WICKET

Wicket user guide

by Andrea Del Bene (an.delbene@gmail.com)

A free guide to Apache Wicket

License

This document is licensed under the Attribution-NonCommercial-ShareAlike 3.0 Unported

You are free:

- to Share — to copy, distribute and transmit the work
- to Remix — to adapt the work

Under the following conditions:

- Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- Noncommercial — You may not use this work for commercial purposes.
- Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

The full license is available at http://creativecommons.org/licenses/by-nc-sa/3.0/.

“We should invent a miracle...yes, we should get to the point where our egoism magically coincides with the happiness of others.”

-Giorgio Gaber, singer, poet and actor.

Table of Contents

Preface 1
How to use the example code 2
Graphic conventions 3
1 Why should I learn Wicket? 4
1.1 We all like spaghetti :-) 4
1.2 Component oriented frameworks: an overview 4
1.3 Benefits of component oriented frameworks for web development 5
1.4 Wicket vs the other component oriented frameworks 5
2 Wicket says “Hello world!” 7
2.1 Wicket distribution and modules 7
2.2 Configuration of Wicket applications 8
2.2.1 Wicket application structure 8
2.2.2 The application class 9
2.3 The HomePage class 11
2.4 Wicket Links 12
2.5 Summary 13
3 Wicket as page layout manager 15
3.1 Header, footer, left menu, content, etc 15
3.2 Here comes the inheritance! 17
3.2.1 Markup inheritance 17
3.2.2 Panel class 18
3.3 Divide et impera! 20
3.3.1 Panels and layout areas 20
3.3.2 Template page 21
3.3.3 Final example 21
3.4 Markup inheritance with <wicket:extend> tag 23
3.4.1 Our example revisited 24
3.5 Summary 25
4 Keeping control over HTML 26
4.1 Hiding or disabling a component 26
4.2 Modifing tag attributes 26
4.3 Generating tag attribute 'id' 27
4.4 Creating panels “on the fly” with WebMarkupContainer 27
4.5 Working with markup fragments 28
4.6 Adding header contents to the final page 29
4.7 Using stub markup in our pages/panels 30
4.8 How to render component body only 30
4.9 Hiding decorating elements with tag <wicket:enclosure> 31
4.10 Surrounding existing markup with Border 32
4.11 Summary 33
5 Components lifecycle 35
5.1 Lifecycle stages of a component 35
5.2 Hook methods for component lifecycle 35
5.3 Initialization stage 36
5.4 Rendering stage 36
5.4.1 Method onBeforeRender 36
5.4.2 Method onConfigure 37
5.4.3 Method onComponentTag 37
5.4.4 Methods onComponentTagBody 38
5.5 Removing stage 39
5.6 Summary 39
6 Page versioning and caching 40
6.1 Stateful pages VS stateless 40
6.2 Stateful pages 40
6.2.1 Using a specific page version with PageReference 42
6.2.2 Turning off page versioning 42
6.2.3 Pluggable serialization 42
6.2.4 Page caching 42
6.2.5 Page expiration 43
6.3 Stateless pages 44

6.4 Summary....45
7 Under the hood of request processing....46
7.1 Class Application and request processing....46
7.2 Classes Request and Response....46
7.3 The “director” of request processing: RequestCycle....46
7.3.1 RequestCycle and request processing....47
7.3.2 Generating url with methods urlFor and mapUrlFor....48
7.3.3 Method setResponsePage....48
7.3.4 RequestCycle's hook methods and listeners....48
7.4 Class Session....49
7.4.1 Session and listeners....49
7.4.2 Handling session attributes....50
7.4.3 Accessing to http session....50
7.4.4 Temporary and permanent sessions....51
7.4.5 Discarding session data....52
7.5 Storing arbitrary objects with metadata....52
7.6 Summary....53
8 Wicket Links and URL generation....54
8.1 PageParameters....54
8.1.1 PageParameters and bookmarkable pages....54
8.1.2 Indexed parameters....55
8.2 Bookmarkable links....56
8.3 Automatically creating bookmarkable links with tag <wicket:link>....56
8.4 External links....58
8.5 Stateless links....59
8.6 Generating structured and clear URLs....59
8.6.1 Mounting a single page....59
8.6.2 Using parameter placeholders with mounted pages....60
8.6.3 Mounting a package....61
8.6.4 Providing custom mapper context to request mappers....61
8.6.5 Controlling how page parameters are encoded with IPageParametersEncoder....62
8.6.6 Encrypting page URLs....63
8.7 Summary....64
9 Wicket models and forms....65
9.1 What is a model?....65
9.2 Models and JavaBeans....67
9.2.1 PropertyModel....67
9.2.2 CompoundPropertyModel and model inheritance....68
9.3 Wicket forms....70
9.3.1 Form and models....70
9.3.2 Login form....71
9.4 Component DropDownChoice....74
9.5 Model chaining....75
9.6 Detachable models....78
9.7 Using more than one model in a component....80
9.8 Use models!....81
9.9 Summary....81
10 Wicket forms in detail....82
10.1 Default form processing....82
10.2 Form validation and feedback messages....82
10.2.1 Feedback messages and localization....83
10.2.2 Displaing feedback messages and filtering them....84
10.2.3 Built-in validators....84
10.2.4 Overriding standard feedback messages with custom bundles....86
10.2.5 Creating custom validators....86
10.2.6 Using flash messages....88
10.3 Input value conversion....89
10.3.1 Creating custom application-scoped converters....90
10.4 Submit form with an IFormSubmittingComponent....92
10.4.1 Components Button and SubmitLink....93
10.4.2 Disabling default form processing....95
10.5 Nested forms....95
10.6 Multi-line text input....95
10.7 File upload....96
10.7.1 Upload multiple files....97

10.8 Creating complex form components with FormComponentPanel....97
10.9 Stateless form....100
10.10 Working with radio buttons and checkboxes....102
10.10.1 Working with grouped checkboxes....104
10.10.2 How to implement a “Select all” checkbox....105
10.10.3 Working with grouped radio buttons....106
10.11 Selecting multiple values with ListMultipleChoices and Palette....106
10.11.1 Component Palette....107
10.12 Summary....109
11 Displaying multiple items with repeaters....110
11.1 Component RepeatingView....110
11.2 Component ListView....111
11.2.1 ListView and Form....112
11.3 Component RefreshingView....112
11.3.1 Item reuse strategy....113
11.4 Pageable repeaters....113
11.4.1 Component DataView....113
11.4.2 Data paging....114
11.5 Summary....115
12 Internationalization with Wicket....117
12.1 Localization....117
12.2 Class Locale and ResourceBundle....117
12.3 Localization in Wicket....118
12.3.1 Style and variation parameters for bundles....119
12.3.2 Using XML files as resource bundles....119
12.3.3 Reading bundles from code....120
12.3.4 Localization of bundles in Wicket....120
12.3.5 Localization of markup files....121
12.3.6 Reading bundles with tag <wicket:message>....121
12.4 Bundles lookup algorithm....122
12.4.1 Localizing pages and panels....122
12.4.2 Component-specific resources....123
12.4.3 Package bundles....124
12.4.4 Bundles for feedback messages....124
12.4.5 Extending the default lookup algorithm....125
12.5 Localization of component's choices....125
12.6 Internationalization and Models....126
12.6.1 ResourceModel....127
12.6.2 StringResourceModel....127
12.7 Summary....128
13 Resource management with Wicket....130
13.1 Static vs dynamic resources....130
13.2 Resource references....130
13.3 Package resources....130
13.3.1 Using package resources with tag <wicket:link>....132
13.4 Adding resources to page header section....133
13.5 Resource dependencies....134
13.6 Custom resources....134
13.7 Mounting resources....135
13.8 Shared resources....136
13.9 Customizing resource loading....137
13.10 Summary....139
14 An example of integration with JavaScript....140
14.1 What we want to do....140
14.1.1 What features we want to implement....140
14.2 ...and how we will do it....141
14.2.1 Component package resources....141
14.2.2 Initialization code....142
14.2.3 Header contributor code....143
14.3 Summary....144
15 Wicket advanced topics....145
15.1 Enriching components with behaviors....145
15.2 Generating callback URLs with IRequestListener....146
15.3 Wicket events infrastructure....148
15.4 Initializers....150

15.5 Using JMX with Wicket....150
15.6 Generating HTML markup from code....152
15.6.1 Avoiding markup caching....153
15.7 Summary....154
16 Working with AJAX....155
16.1 How to use AJAX components and behaviors....155
16.2 Built-in AJAX components....156
16.2.1 Links and buttons....156
16.2.2 Fallback components....156
16.2.3 AJAX Checkbox....157
16.2.4 AJAX editable labels....157
16.2.5 Autocomplete text field....158
16.2.6 Modal window....158
16.2.7 Tree repeaters....160
16.2.8 Working with hidden components....165
16.3 Built-in AJAX behaviors....165
16.3.1 AjaxEventBehavior....165
16.3.2 AjaxFormSubmitBehavior....167
16.3.3 AjaxFormComponentUpdatingBehavior....167
16.3.4 AbstractAjaxTimerBehavior....167
16.4 Using an activity indicator....168
16.5 Ajax request attributes and call listeners....168
16.6 Creating custom AJAX call listener....170
16.6.1 What we want for our listener....170
16.6.2 How to implement the listener....171
16.6.3 JavaScript code....171
16.6.4 Class code....172
16.6.5 Global listeners....173
16.7 Summary....174
17 Integration with enterprise containers....175
17.1 Integrating Wicket with EJB....175
17.2 Integrating Wicket with Spring....176
17.3 JSR-330 annotations....177
17.4 Summary....178
18 Security with Wicket....179
18.1 Authentication....179
18.1.1 AuthenticatedWebSession....179
18.1.2 AuthenticatedWebApplication....180
18.1.3 A basic example of authentication....180
18.1.4 Redirecting user to an intermediate page....182
18.2 Authorizations....183
18.2.1 SimplePageAuthorizationStrategy....184
18.2.2 Role-based strategies....184
18.2.2.1 Using roles with metadata....185
18.2.2.2 Using roles with annotations....187
18.2.3 Catching an unauthorized component instantiation....188
18.2.4 Strategy RoleAuthorizationStrategy....189
18.3 Using HTTPS protocol....189
18.4 Package Resource Guard....190
18.5 Summary....191
19 Test Driven Development with Wicket....192
19.1 Utility class WicketTester....192
19.1.1 Testing links 193
19.1.2 Testing component status....194
19.1.3 Testing components in isolation....194
19.1.4 Testing the response....194
19.1.5 Testing URLs....195
19.1.6 Testing AJAX components....195
19.1.7 Testing AJAX events....195
19.1.8 Testing AJAX behaviors....196
19.1.9 Using a custom servlet context....197
19.2 Testing Wicket forms....197
19.2.1 Setting form components input....198
19.2.2 Testing feedback messages....198
19.2.3 Testing models....199

19.3 Testing markup with TagTester....199
19.4 Summary....200
Appendix A: working with Maven....202
A.1Switching Wicket to DEPLOYMENT mode....202
A.2Creating a Wicket project from scratch and importing it into our favourite IDE....203
A.2.1From Maven to our IDE....203
A.2.2Importing a Maven project into our IDE....205
A.2.3Speeding up development with plugins....207
Appendix B: project WicketStuff....209
B.1What is project WicketStuff?....209
B.2Module tinymce....209
B.3Module wicketstuff-gmap3....210
B.4Module wicketstuff-googlecharts....211
B.5Module wicketstuff-inmethod-grid....212
Alphabetical Index....215

Preface

Wicket has been around since 2004 and it's an Apache project since 2007. During these years it has proved to be a solid and valuable solution for building enterprise web applications.

Wicket core developers have done a wonderful job with this framework and they continue to improve it release after release.

However Wicket never provided a freely available documentation and even if you can find on Internet many live examples and many technical articles on it (most of them at http://www.wicket-library.com/ and at http://wicketinaction.com/), the lack of an organized and freely available documentation has always been a sore point for this framework.

That's quite an issue because many other popular frameworks (like Spring, Hibernate or Struts) offer a vast and very good documentation which substantially contributed to their success.

This document is not intended to be a complete reference for Wicket but it simply aims to be a straightforward introduction to the framework that should significantly reduce its learning curve. What you will find here reflects my experience with Wicket and it's strictly focused on the framework.

The various Wicket-related topics are gradually introduced using pragmatic examples of code that you can find at https://github.com/bitstorm/Wicket-tutorial-examples.

However remember that Wicket is a vast and powerful tool, so you should feel confident with the topics exposed in this document before starting to code your real applications!

For those who need further documentation on Wicket, there are many good books available on this framework. You can find an exhaustive list of these books at http://wicket.apache.org/learn/books/

Hope you'll find this guide helpful. Have fun with Wicket!

Andrea Del Bene, an.delbene@gmail.com

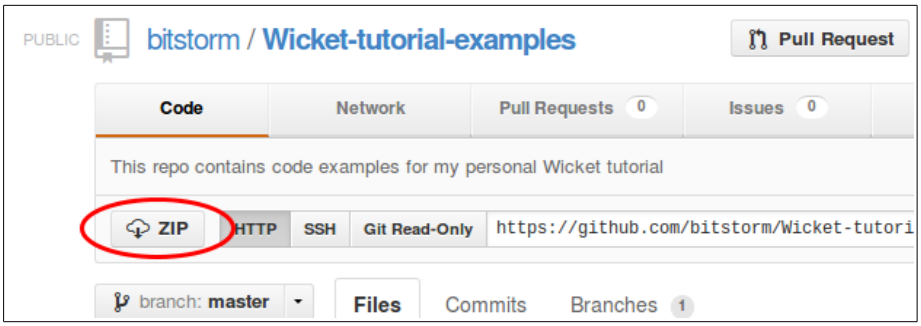
*PS: this guide is based on **Wicket 6**. However if you are using an older version you should find this guide useful as well, but it's likely that code examples and snippets won't work with your version.*
PPS: although I've did my best working on this tutorial, this document is a work in progress and may contain errors and/or omissions. That's why any feedback of any kind is REALLY appreciated!

How to use the example code

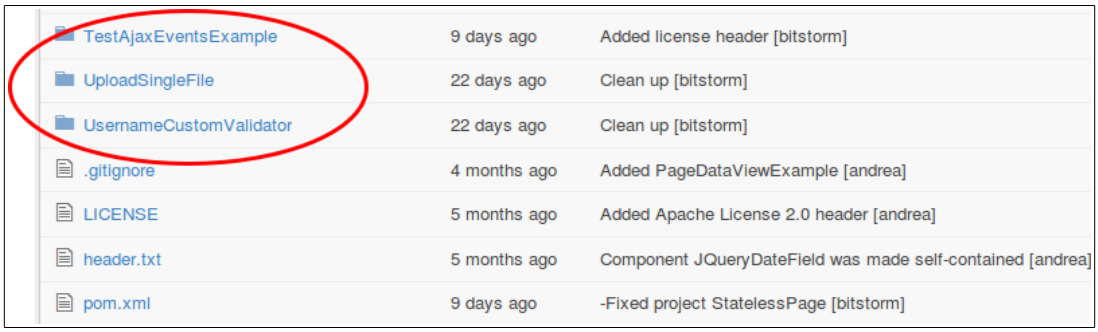
Most of the code you will find in this document is available as Git repository at https://github.com/bitstorm/Wicket-tutorial-examples and is licensed under the ASF 2.0[1]. To get a local copy of the repository you can run the clone command from shell:

```
git clone https://github.com/bitstorm/Wicket-tutorial-examples.git
```

If you aren't used to Git, you can simply download the whole source as a zip archive:



The repository contains a multi-module Maven project. Every subproject is contained in the relative folder of the repository:



When the example code is used in the document, you will find the name of the subproject it belongs to. If you haven't any experience with Maven, you can read Appendix A where you can learn the basic commands needed to work with example projects and to import them into your favourite IDE (NetBeans, IDEA or Eclipse).

1 http://www.apache.org/licenses/LICENSE-2.0

Graphic conventions

To make reading easier, some graphic conventions have been adopted:

- Code reference inside text are written with a different font (`FreeMono`):

  ...the variable `message` is....

- Code blocks are formatted and coloured following the default Eclipse style:

```
/**
 * This is about <code>ClassName</code>.
 * {@link com.yourCompany.aPackage.Interface}
 * @author author
 * @deprecated use <code>OtherClass</code>
 */
public class ClassName<E> implements InterfaceName<String> {
	enum Color { RED, GREEN, BLUE };
	/* This comment may span multiple lines. */
	static Object staticField;
	// This comment may span only this line
	private E field;
	// TASK: refactor
	@SuppressWarnings(value="all")
	public int foo(Integer parameter) {
		abstractMethod();
		int local= 42*hashCode();
		staticMethod();
		return bar(local) + parameter;
	}
}
```

- Important informations and warnings are written inside blocks like these:

Note

bla...bla...bla....

Warning

bla...bla...bla....

1 Why should I learn Wicket?

Software development is a challenging activity and developers must keep their skills up-to-date with new technologies.

But before starting to learn the last “coolest” framework we should always ask ourself if it is the right tool for us and how it can improve our everyday job.

Java ecosystem is already full of many well-known web frameworks, so why should we spend our time learning Wicket?

This chapter will show you why Wicket is different from other web frameworks you may know and it will explain also how Wicket can improve your life as web developer.

1.1 We *all* like spaghetti :-) ...

...but we all hate spaghetti code! That's why in the first half of the 2000s we have seen the birth of so many web frameworks. Their mission was to separate our business code from presentation layer (like JSP pages).

Some of theme (like Struts, Spring MVC, Velocity, ecc...) have become widely adopted and they made the MVC pattern very popular among developers.

However no one of these frameworks offers a real OO[2] abstraction for web pages and we still have to take care of web-related tasks such as HTTP request/response handling, URLs mapping, storing data into user session and so on.

But the biggest limit of MVC frameworks is that they don't do much to overcome the *impedance mismatch* between the stateless nature of HTTP protocol and the need of our web applications of handling a (very complex) state.

To overcome these limits developers have started to adopt a new generation of ***component oriented*** web frameworks designed to provide a completely different approach to web development.

1.2 Component oriented frameworks: an overview

Component oriented frameworks differ from classic web frameworks in that they build a model of requested page on server side and the HTML sent back to the client is generated according to this model. You can think at this model as if it was an “*inverse*” JavaScript DOM, meaning that:

1. is built *on server-side*
2. is built *before* HTML is sent to client
3. HTML code is generated using this model and not vice versa.

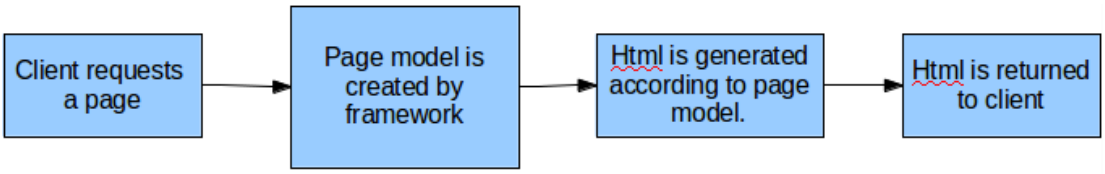


Illustration 1.1: General schema of page request handling for a component oriented framework

2 *Object Oriented*

With this kind of framework our web pages and their HTML components (forms, input controls, links, etc...), are pure class instances.

Since pages are class instances they live inside JVM heap and we can handle them as we use to do with any other Java class.

This approach is very similar to what GUI frameworks (like Swing or SWT) do with desktop windows and their components. Wicket and the other component oriented frameworks bring to web development the same kind of abstraction that GUI frameworks offer when we build a desktop application. But most of all this kind of framework hides the details of HTTP protocol and naturally solves the problem of its stateless nature.

1.3 Benefits of component oriented frameworks for web development

At this point some people may still wonder why OOP is so important also for web development and what benefits it can bring to developers.

Let's quickly review the main advantages that this paradigm can offer us:

- **Web pages are objects**: web pages are no more just text files sent back to the client. They are object instances and we can harness OOP to design web pages and their components. With Wicket we can also apply inheritance to HTML markup in order to build a consistent graphic layout for our applications (we will see *markup inheritance* in chapter 3).

- **We don't have to worry about application's state**: pages and components can be considered *stateful* entities. They are Java objects and they can keep a state inside them and reference other objects. We can stop worrying about keeping track of user data stored inside `HttpSession` and we can start managing them in a natural and transparent way.

- **Testing web applications is much easier**: since pages and components are pure objects, you can use JUnit to test their behavior and to ensure that they render as expected.
  Wicket has a set of utility classes for unit testing that simulate user interaction with web pages, hence we can write acceptance tests using just JUnit without any other test framework (unit testing is covered in chapter 20).

1.4 Wicket vs the other component oriented frameworks

Wicket is not the only component oriented framework available in the Java ecosystem. Among its competitors we can find GWT (from Google), JSF (from Oracle), Vaadin (from Vaadin Ltd.), etc...

Even if Wicket and all these frameworks have their pros and cons, there are good reasons to prefer Wicket over them:

- **Wicket is 100% open source:** Wicket is a top Apache project and it doesn't depend on any private company. You don't have to worry about future licensing changes, Wicket will always be released under Apache license 2.0 and freely available.

- **Wicket is a community driven project:** Wicket team supports and promotes the dialogue with framework users through two mailing lists (one for users and another one for framework developers)[3] and an Apache JIRA[4] (the issue tracking system).
  Moreover, as any other Apache project, Wicket is developed paying great attention to user feedbacks and to suggested features.

3 See http://wicket.apache.org/help/email.html
4 https://issues.apache.org/jira/browse/WICKET

- **Wicket is just about Java and good old HTML:** almost all web frameworks force users to adopt special tags or to use server side code inside HTML markup. This is clearly in contrast with the concept of separation between presentation and business logic and it leads to a more confusing code in our pages.
  In Wicket we don't have to take care of generating HTML *inside* the page itself, and we won't need to use any tag other than standard HTML tags.
  All we have to do is to attach our components (Java instances) to HTML tags using a simple tag attribute called **wicket:id** (we will shortly see how to use it).

- **With Wicket we can easily use JavaBeans and POJO[5] in our web tire:** one of the most annoying and error-prone task in web development is collecting user input through a form and keeping form fields updated with previously inserted values. This usually requires a huge amount of code to extract input from request parameters (which are strings), parse them to Java types and store them into some kind of variable. And this is just half of the work we have to do as we must implement also the inverse path (load data from Java to the web form).
  Moreover, most of the times our forms will use a JavaBeans or a POJO as backing object, meaning that we must manually map form fields with the corresponding object fields and vice versa.
  Wicket comes with an intuitive and flexible mechanism that does this mapping for us without any configuration overhead (using a *convention over configuration* approach) and in a transparent way.
  Chapter 9 will introduce the concept of Wicket *model* and we will learn how to harness this entity with forms.

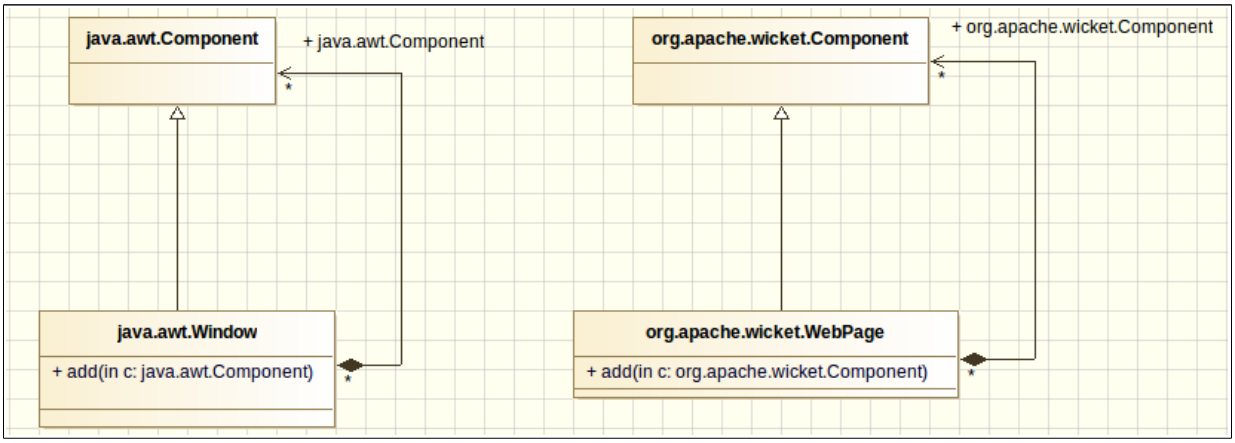
- **No complex XML needed:** Wicket was designed to minimize the amount of configuration files needed to run our applications. No XML file is required except for the standard deployment descriptor *web.xml*.

5 *For a definition of POJO see http://en.wikipedia.org/wiki/Plain_Old_Java_Object*

2 Wicket says *“Hello world!”*

Wicket allows us to design our web pages in terms of components and containers, just like AWT does with desktop windows.

Both frameworks share the same component-based architecture: in AWT we have a `Windows` instance which represents the physical windows containing GUI components (like text fields, radio buttons, drawing areas, ecc...), in Wicket we have a `WebPage` instance which represents the physical web page containing HTML components (pictures, buttons, forms, etc...) .



In both frameworks we find a base class for GUI components called `Component`. Wicket pages can be composed (and usually are) by many components, just like AWT windows are composed by Swing/AWT components.

Both frameworks promote the reuse of presentation code and GUI elements building custom components. Even if Wicket already comes with a rich set of ready-to-use components, building custom components is a common practice working with this framework. We'll learn more about custom components in the next chapters.

2.1 Wicket distribution and modules

Wicket is available as a binary package on the main site http://wicket.apache.org. Inside this archive we can find the distribution jars of the framework. Each jar corresponds to a sub-module of the framework. The following table reports these modules along with a short description of their purpose and with the related dependencies:

Module'sname	Description	Dependencies
wicket-core	Contains the main classes of the framework, like class `Component` and `Application`.	- wicket-request - wicket-util
wicket-request	This module contains the classes involved into web request processing.	- wicket-util
wicket-util	Contains general-purpose utility classes for functional areas such as I/O, lang, string manipulation, security, etc...	None.

wicket-datetime	Contains special purpose components designed to work with date and time.	-wicket-core
wicket-devutils	Contains utility classes and components to help developers with tasks such as debugging, class inspection and so on.	-wicket-core -wicket-extensions
wicket-extensions	Contains a vast set of built-in components to build a rich UI for our web application (Ajax support is part of this module).	-wicket-core
wicket-auth-roles	Provides support for role-based authorization.	-wicket-core
wicket-ioc	This module provides common classes to support Inversion Of Control. It's used by both Spring and Guice integration module.	-wicket-core
wicket-guice	This module provides integration with the dependency injection framework developed by Google.	-wicket-core -wicket-ioc
wicket-spring	This module provides integration with Spring framework.	-wicket-core -wicket-ioc
wicket-velocity	This module provides panels and utility class to integrate Wicket with Velocity template engine.	-wicket-core
wicket-jmx	This module provides panels and utility class to integrate Wicket with *Java Management Extensions*.	-wicket-core
wicket-objectsizeof-agent	Provides integration with Java agent libraries and instrumentation tools.	-wicket-core

Please note that *core* module depends on *utility* and *request* modules, hence it cannot be used without them.

2.2 Configuration of Wicket applications

In this chapter we will see a classic *Hello World!* example implemented using a Wicket page with a built-in component called `Label` (the code is from project *HelloWorldExample*)

Since this is the first example of the guide, before looking at Java code we will go through the common artifacts needed to build a Wicket application from scratch.

Note

All the example projects presented in this document have been generated using Maven and the utility page at http://wicket.apache.org/start/quickstart.html.

Appendix A contains the instructions needed to use this projects and build a quickstart application using Apache Maven. All the artifacts used in the next example (files web.xml, HomePage.class and HomePage.html) are automatically generated by Maven.

2.2.1 Wicket application structure.

A Wicket application is a standard Java EE web application, hence it is deployed through a *web.xml* file placed inside folder WEB-INF[6]:

6 *See "Directory Structure" paragraph of Servlet Specification document*

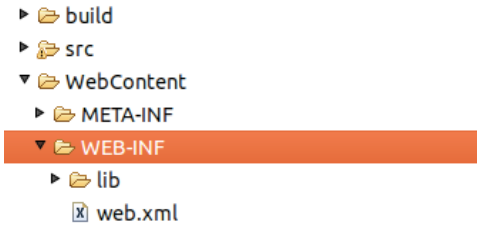


Illustration 2.1: The standard directory structure of a Wicket application

The content of web.xml declares a servlet filter (class `org.apache.wicket.Protocol.http.WicketFilter`) which dispatches web requests to our Wicket application:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    <display-name>Wicket Test</display-name>
    <filter>
        <filter-name>TestApplication</filter-name>
        <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
        <init-param>
          <param-name>applicationClassName</param-name>
          <param-value>org.wicketTutorial.WicketApplication</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>TestApplication</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

Since this is a standard servlet filter we must map it to a specific set of URLs through the `<filter-mapping>` tag). In the xml above we have mapped every URL to our Wicket filter.

Note

Wicket can be started in two modes named respectively **DEVELOPMENT** and **DEPLOYMENT**. The first mode activates some extra features which help application development, like resources monitoring and reloading, full stack trace rendering of exceptions, an AJAX debugger window, etc...

The DEPLOYMENT mode turns off all these features optimizing performances and resource consumption.

In our example projects we will use the default mode which is DEVELOPMENT. Appnedix A contains chapter “Switching Wicket to DEPLOYMENT mode“ where we can find further details about these two modes as well as the possible ways we have to set the desired one.

In any case, **DO NOT** deploy your applications in a production environment without switching to DEPLOYMENT mode!

2.2.2 The application class

If we look back to *web.xml* we can see that we have provided Wicket filter with a parameter called `applicationClassName`. This value must be the fully qualified class name of a subclass of `org.`

apache.wicket.Application. This subclass represents our web application built upon Wicket and it's responsible for configuring it when the server is starting up. Most of the times our custom application class won't inherit directly from class Application, but rather from class org.apache.wicket.protocol.http.WebApplication which provides a closer integration with servlet infrastructure.

Class Application comes with a set of configuration methods that we can override to customize our application's settings. One of these methods is getHomePage() that must be overridden as it is declared abstract:

```
public abstract Class<? extends Page> getHomePage()
```

As you may guess from its name, this method specifies which page to use as homepage for our application.

Another important method is init():

```
protected void init()
```

This method is called when our application is loaded by web server (Tomcat, Jetty, ecc...) and is the ideal place to put our configuration code. Application class exposes its settings grouping them into interfaces (you can find them in package org.apache.wicket.settings). We can access these interfaces through getter methods that will be gradually introduced in the next chapters when we will cover the related settings.

The current application's instance can be retrieved at any time calling static method Application.get() in our code. We will give more details about this method in chapter 7.3.

The content of the application class from project *HelloWorldExample* is the following:

```
public class WicketApplication extends WebApplication
{
	@Override
	public Class<? extends WebPage> getHomePage()
	{
		return HomePage.class;
	}

	@Override
	public void init()
	{
		super.init();
		// add your configuration here
	}
}
```

Since this is a very basic example of Wicket application, we don't need to specify anything inside init method. The home page of the application is class HomePage.java. In the next paragraph we will see how this page is implemented and which conventions we have to follow to create a page in Wicket.

Note

Declaring a WicketFilter inside *web.xml* descriptor is not the only way we have to kickstart our application.

If we prefer to use a servlet instead of a filter, we can use class org.apache.

wicket.protocol.http.WicketServlet. See JavaDoc for further details.

2.3 The HomePage class

To complete our first Wicket application we must explore the home page class that is returned by Application's method getHomePage() seen above.

In Wicket a web page is a subclass of org.apache.wicket.WebPage. This subclass must have a corresponding HTML file which will be used by the framework as template to generate its HTML markup. This file is a regular plain HTML file (its extension must be html).

By default this HTML file must have the same name of the related page class and must be in the same package:

Illustration 2.2:Page class and its related HTML file

If you don't like to put class and html side by side (let's say you want all your HTML files in a separated folder) you can use Wicket settings to specify where HTML files can be found. We will cover this topic later in paragraph 13.9.

The Java code for HomePage.java is the following:

```
package org.wicketTutorial;

import org.apache.wicket.request.mapper.parameter.PageParameters;
import org.apache.wicket.markup.html.basic.Label;
import org.apache.wicket.markup.html.WebPage;

public class HomePage extends WebPage {
    public HomePage() {
        add(new Label("helloMessage", "Hello WicketWorld!"));
    }
}
```

Apart from subclassing WebPage, HomePage defines a constructor that adds a Label component to itself.

Method add(Component component) is inherited from ancestor class org.apache.wicket.MarkupContainer and is used to add children components to a web page. We'll see more about MarkupContainer later in chapter 3.2.2.

Class org.apache.wicket.markup.html.basic.Label is the simplest component shipped with Wicket. It just inserts a string (the second argument of its constructor) inside the corresponding HTML tag.

Just like any other Wicket component, Label needs a textual id ('helloMessage' in our example) to be instantiated. At runtime Wicket will use this value to find the HTML tag we want to bind to the component. This tag must have a special attribute called wicket:id and its value must be identical to the component id (comparison is case-sensitive!).

Here is the HTML markup for HomePage (file HomePage.html):

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="utf-8" />
                <title>Apache Wicket HelloWorld</title>
        </head>
        <body>

                <div wicket:id="helloMessage">
                [Label's message goes here]
                </div>
        </body>
</html>
```

We can see that the `wicket:id` attribute is set according to the value of component id. If we run this example we will see the text `Hello WicketWorld!` inside `<div>` tag.

Note

Please note that Label replaces the original content of its tag (in our example `[Label's message goes here]`) with the string passed as value (`Hello WicketWorld!` in our example).

Warning

If we specify a `wicket:id` attribute for a tag without adding the corresponding component in our Java code, Wicket will throw a `ComponentNotFoundException`.

On the contrary if we add a component in our Java code without specifying a corresponding `wicket:id` attribute in our markup, Wicket will throw a `WicketRuntimeException`.

2.4 Wicket Links

The basic form of interaction offered by web applications is to navigate through pages using links. In HTML a link is basically a pointer to another resource that most of the times is another page. Wicket implements links with component `org.apache.wicket.markup.html.link.Link`, but due to the component-oriented nature of the framework, this component is quite different from classic HTML links.

Following the analogy with GUI frameworks, we can consider Wicket link as a “click” event handler: its purpose is to perform some actions (on server side!) when user clicks on it.

That said, you shouldn't be surprised to find an abstract method called `onClick()` inside `Link` class. In the following example we have a page with a `Link` containing an empty implementation of `onClick`:

```
public class HomePage extends WebPage {
        public HomePage(){
                add(new Link("id"){
                        @Override
                        public void onClick() {
                           //link code goes here
                        }
```

```
                });
        }
}
```

By default after `onClick` has been executed, Wicket will send back to client the current page. If we want to navigate to another page we must use `Component`'s method `setResponsePage`:

```
public class HomePage extends WebPage {
        public HomePage(){
                add(new Link("id"){

                        @Override
                        public void onClick() {
                           //we redirect browser to another page.
                           setResponsePage(AnotherPage.class);
                        }
                });
        }
}
```

In the example above we used a version of `setResponsePage` which takes in input the class of the target page. In this way a new instance of `AnotherPage` will be create each time we click on the link. The other version of `setResponsePage` implemented by `Component` takes in input a page instance instead of a page class:

```
        //...
        @Override
        public void onClick() {
           //we redirect browser to another page.
           AnotherPage anotherPage = new AnotherPage();
           setResponsePage(anotherPage);
        }
        //...
```

The difference between using the first version of `setResponsePage` rather than the second one will be illustrated in chapter 6, when we will introduce the topic of *stateful* and *stateless* pages. For now, we can consider them as equivalent.

Note

Wicket comes with a rich set of link components suited for every needs (links to static url, Ajax-enhanced links, links to a file to download, links to external pages and so on). We will see them in chapter 8.

2.5 Summary

In this chapter we have seen the very basic elements that compose a Wicket application. We have started preparing the configuration artifacts needed for our applications. As promised in paragraph 1.4, we needed to put in place just a minimal amount of XML with an *application* class and an home page.

Then we have continued our “first contact” with Wicket learning how to build a simple page with a label component as child. This example page has shown us how Wicket maps components to HTML tags and

how it uses both of them to generate the final HTML markup.

In the last paragraph we had a first taste of Wicket links and we have seen how they can be considered as a “click” event listener and how they can be used to move from a page to another.

3 Wicket as page layout manager

Before going ahead with more advanced topics, we will see how to maintain a consistent layout across our site using Wicket and its component-oriented features.

Probably this is not the most interesting use we can do of Wicket, but it is surely the simplest one so it's the best way to start dirtying our hands with some code.

3.1 Header, footer, left menu, content, etc...

There was a time in the 90s when Internet was just a buzzword and watching a plain HTML page being rendered by a browser was a new and amazing experience. In those days we used to organize our page layout using HTML tag `<frame>`.

By the years this tag has almost disappeared from our code and it survives only in few specific domains. For example is still be used for JavaDoc:

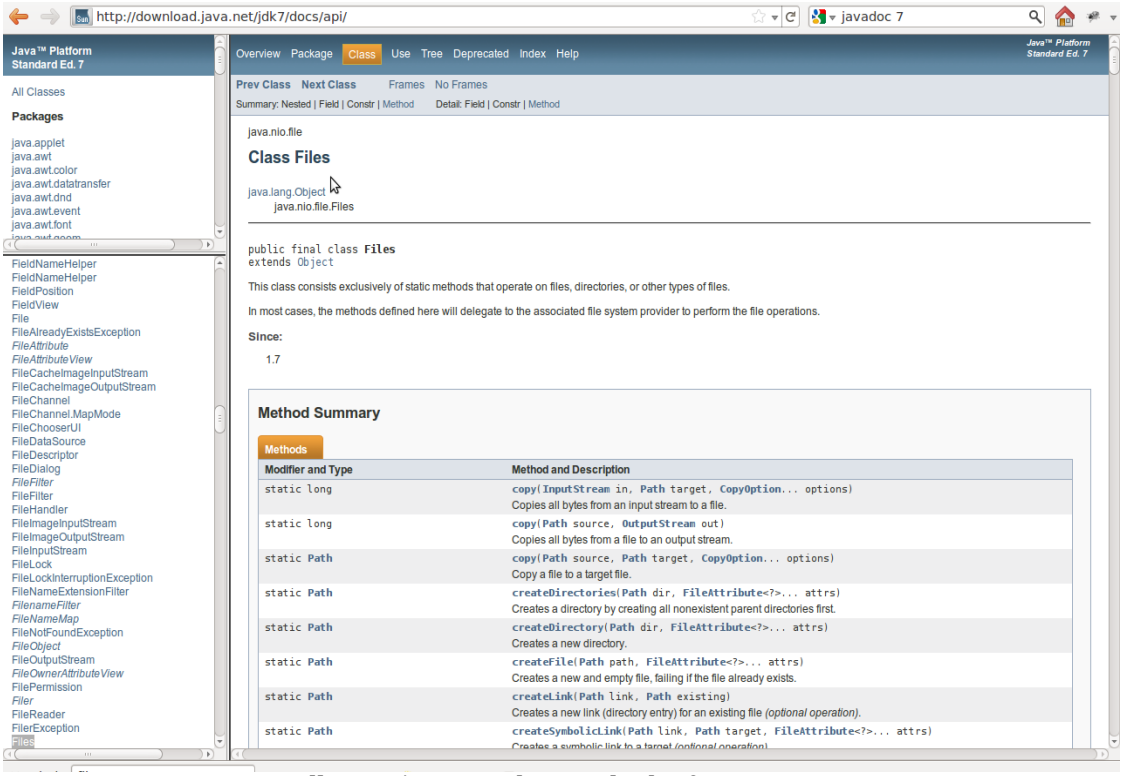


Illustration 3.1: The new look of JavaDoc 7

With the adoption of server side technologies like JSP, ASP or PHP the tag `<frame>` has been replaced by a template-based approach where we divide our page layout into some common areas that will be present in each page of our web application. Then, we manually insert these areas in every page including the appropriate markup fragments.

In this chapter we will see how to use Wicket to build a site layout. The sample layout we will use is a typical page layout consisting of the following areas:

- **a header** which could contain site title, some logos, a navigation bar, etc...
- **a left menu** with a bunch of links to different areas/functionalities of the site.
- **a footer** with generic informations like web master's email, the company address, etc...

- **a content area** which usually contains the functional part of the page.

The following picture summarises the layout structure:

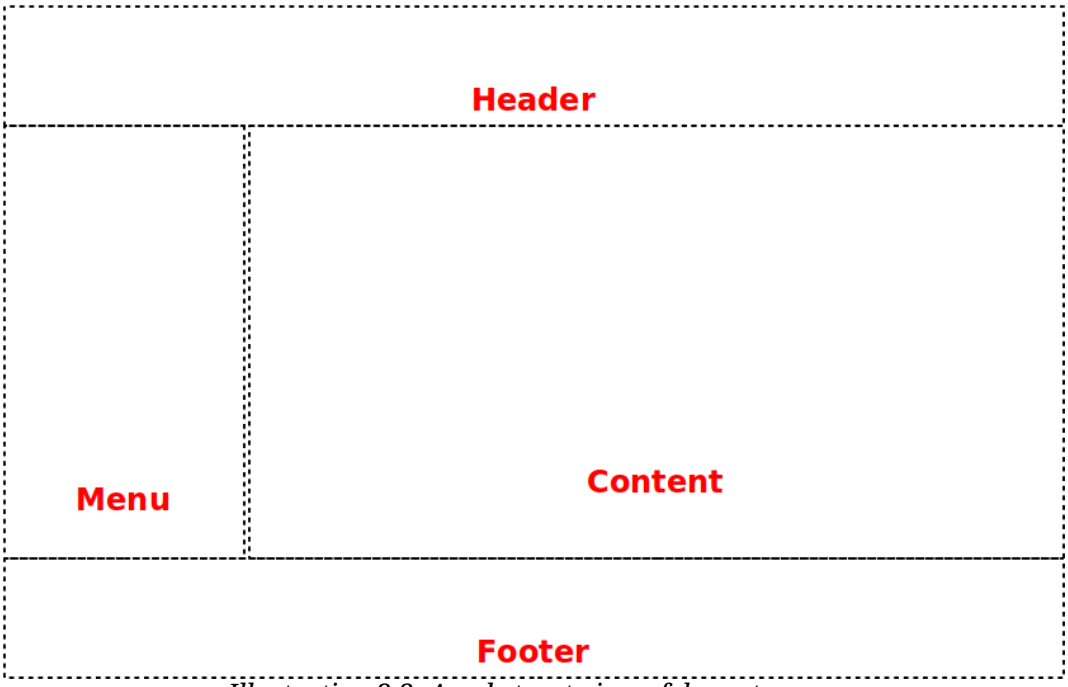


Illustration 3.2: An abstract view of layout areas

Once we have chosen a page layout, our web designer can start building up the site theme. The result is a beautiful mock of our future web pages. Over this mock we can map the original layout areas[7]:

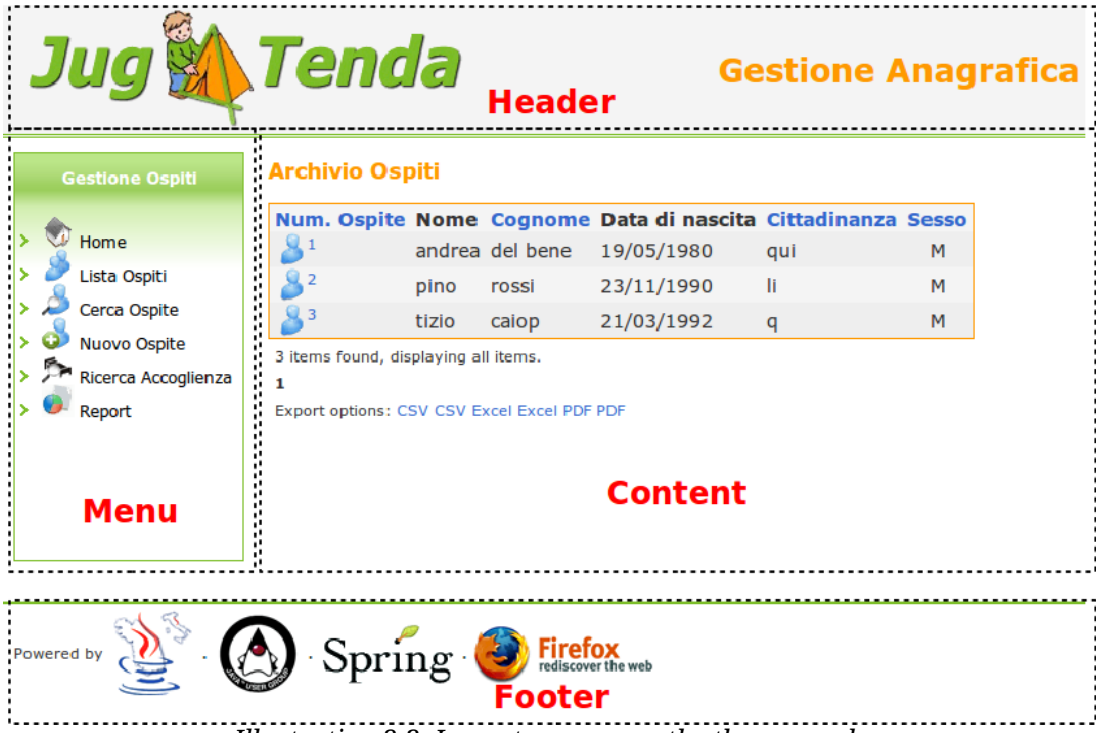


Illustration 3.3: Layout areas over the theme mock

7 *The mock is taken from charity project Jug4Tenda (http://java.net/projects/jugancona), lead by Italian Jug Marche.*

Now in order to have a consistent layout across all the site, we must ensure that each page will include the layout areas seen above. With an old template-based approach we must manually put them inside every page. If we were using JSP we would probably end up using `include` directive to add layout areas in our pages. We would have one `include` for each of the areas (except for the content):

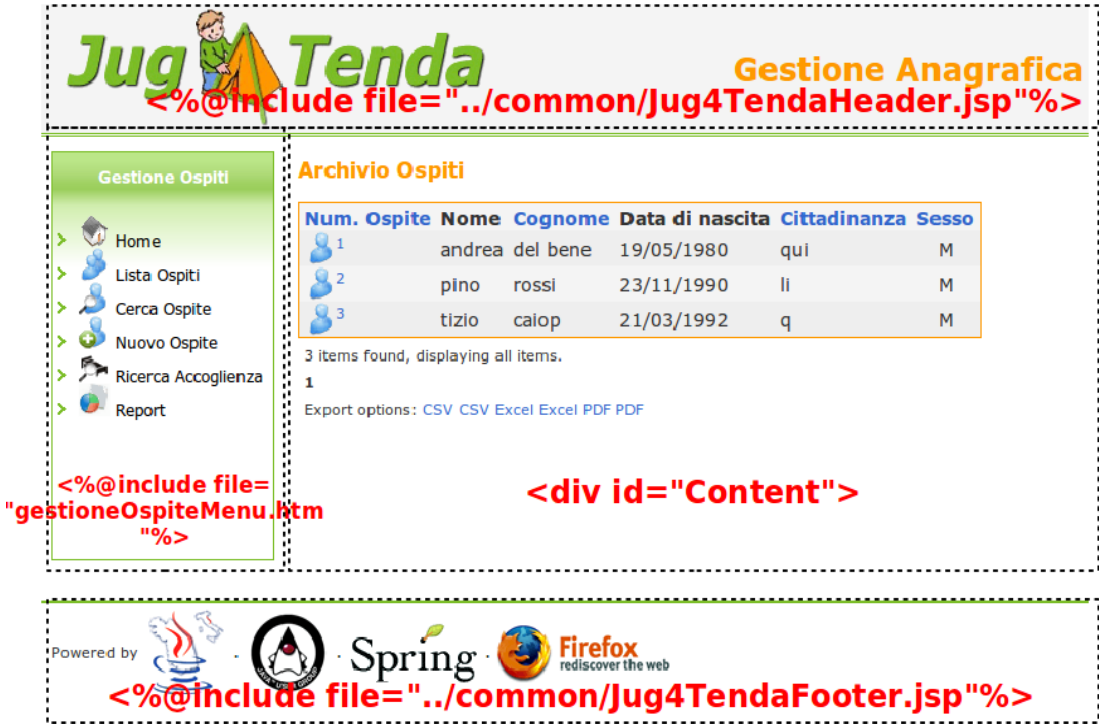


Illustration 3.4: Layout areas are assembled with `include` directive

Note

For the sake of simplicity we can consider each included area as a static HTML fragment.

Now let's see how we can handle the layout of our web application using Wicket.

3.2 Here comes the inheritance!

The need of ensuring a consistent layout across our pages unveiled a serious limit of HTML language: the inability to apply *inheritance* to web pages and their markup. Wouldn't be great if we could write our layout *once* in a page and then *inherit* it in the other pages of our application?

One of the goal of Wicket is to overcome this kind of limit.

3.2.1 Markup inheritance

As we have seen in the previous chapter, Wicket pages are pure Java classes, so we can easily write a page which is a subclass of another *parent* page.

But in Wicket inheritance is not limited to the classic object-oriented *code inheritance*. When a class subclasses a `WebPage` it also inherits the HTML file of to the parent class. This type of inheritance is called *markup inheritance.*

To better illustrate this concept let's consider the following example where we have a page class called `GenericSitePage` with the corresponding HTML file `GenericSitePage.html`. Now let's create a specific page called `OrderCheckOutPage` where users can check out their orders on our our web site.

This class extends `GenericSitePage` but we don't provide it with any corresponding HTML file.
In this scenario `OrderCheckOutPage` will use `GenericSitePage.html` as markup file:

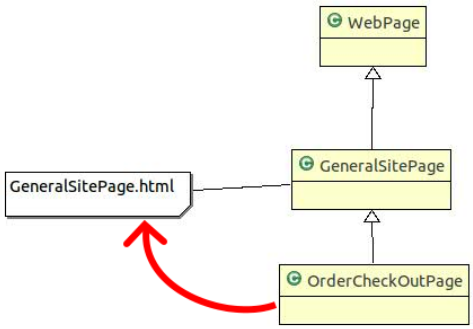


Illustration 3.5: Page OrderCheckOutPage hasn't a corresponding HTML file. It will use the one from parent page (GeneralSitePage.html).

Markup inheritance comes in handy for page layout management as it avoids us the burden of checking that each page conforms to site layout. However to fully take advantage of markup inheritance we must first learn how to use another important component of the framework that supports this feature: the *panel*.

Warning

If no markup is found (nor directly assigned to the class, neither inherited from an ancestor) a `MarkupNotFoundException` is thrown.

3.2.2 Panel class

Class `org.apache.wicket.markup.html.panel.Panel` is a special component which let us reuse GUI code and HTML markup across different pages and different web applications. It shares a common ancestor class with `WebPage` class, which is `org.apache.wicket.MarkupContainer`:

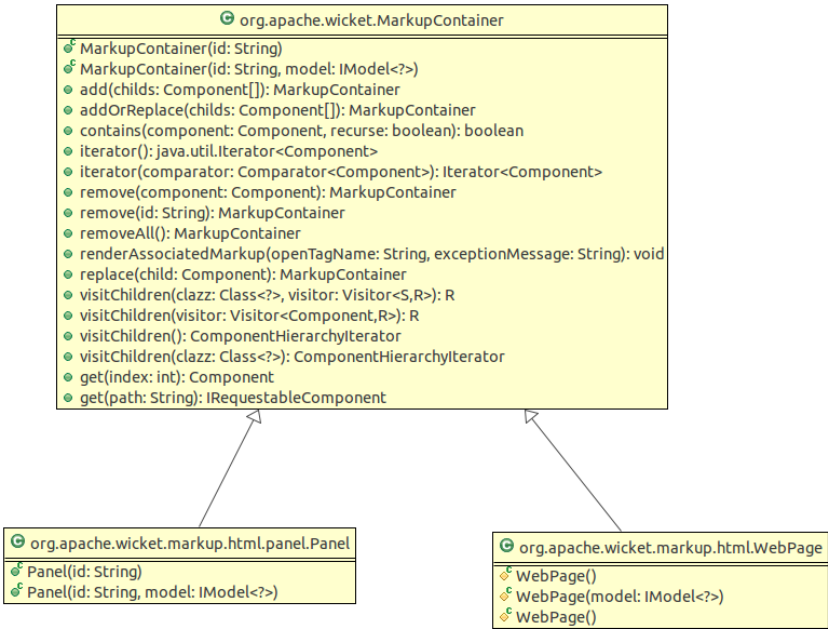


Illustration 3.6: Hierarchy of WebPage and Panel classes

Subclasses of `MarkupContainer` can contain children components that can be added with method `add(Componet...)` (seen in chapter 2.3).

`MarkupContainer` implements a full set of methods to manage children components. The basic operations we can do on them are:

- add one or more children components (with method `add`).
- remove a specific child component (with methods `remove`).
- retrieve a specific child component with method `get(String)`. The string parameter is the id of the component or its relative *path* if the component is nested inside other `MarkupContainer`s. This path is a colon-separated string containing also the ids of the intermediate containers traversed to get to the child component. To illustrate an example of component path, let's consider the code of the following page:

```
MyPanel myPanel = new MyPanel ("innerContainer");
add(myPanel);
```

  Component `MyPanel` is a custom panel containing only a label having `"name"` as id. Under these conditions we could retrieve this label from the container page using the following path expression:

```
Label name = (Label)get("innerContainer:name");
```

- replace a specific child component with a new component **having the same id** (with method `replace`).
- iterate thought children components with the iterator returned by method `iterator` or using visitor pattern[8] with methods `visitChildren`.

Both `Panel` and `WebPage` have their own associated markup file which is used to render the corresponding component. If such file is not provided, Wicket will apply *markup inheritance* looking for a markup file through their ancestor classes. When a panel is attached to a container, the content of its markup file is inserted into its related tag.

While panels and pages have much in common, there are some notable differences between these two components that we should keep in mind.

The main difference between them is that pages can be rendered as standalone entities while panels must be placed inside a page to be rendered.

Another important difference is the content of their markup file: for both `WebPage` and `Panel` this is a standard HTML file, but `Panel` uses a special tag to indicate which part of the whole file will be considered as markup source. This tag is `<wicket:panel>`. A markup file for a panel will typically look like this:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
...
</head>
<body>
   <wicket:panel>
      <!-- Your markup goes here -->
   </wicket:panel>
</body>
```

The HTML outside tag `<wicket:panel>` will be removed during rendering phase. The space outside this

8 http://en.wikipedia.org/wiki/Visitor_pattern

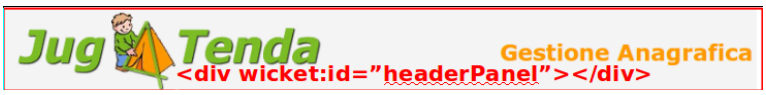
tag can be used by both web developers and web designers to place some mock HTML to show how the final panel should look like.

3.3 Divide et impera!

Let's go back to our layout example. In chapter 3.1 we have divided our layout in common areas that must be part of every page. Now we will build a reusable template page for our web application combining page and panels. The code examples are from project *MarkupInheritanceExample*.

3.3.1 Panels and layout areas

First, let's build a custom panel for each layout area (except for 'content' area). For example given the header area



we can build a panel called `HeaderPanel` with a related markup file called `HeaderPanel.html` containing the HTML for this area:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
...
</head>
<body>
 <wicket:panel>
  <table width="100%" style="border: 0px none;">
  <tbody>
   <tr>
   <td>
      <img alt="Jug4Tenda" src="wicketLayout_files/logo_jug4tenda.gif">
    </td>
  <td>
   <h1>Gestione Anagrafica</h1>
  </td>
  </tr>
  </tbody>
  </table>
 </wicket:panel>
</body>
<html>
```

The class for this panel simply extends base class `Panel`:

```
package helloWorld.layoutTenda;

import org.apache.wicket.markup.html.panel.Panel;

public class HeaderPanel extends Panel {

        public HeaderPanel(String id) {
                super(id);
        }
}
```

For each layout area we will build a panel like the one above that holds the appropriate HTML markup. In the end we will have the following set of panels:

- **HeaderPanel**

- **FooterPanel**
- **MenuPanel**

Content area will change from page to page, so we don't need a reusable panel for it.

3.3.2 Template page

Now we can build a generic template page using our brand new panels. Its markup is quite straightforward :

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
...
<!--Include CSS-->
...
</head>
<body>
<div id="header" wicket:id="headerPanel">header</div>
<div id="body">
        <div id="menu" wicket:id="menuPanel">menu</div>
        <div id="content" wicket:id="content">content</div>
</div>
<div id="footer" wicket:id="footerPanel">footer</div>
</body>
</html>
```

The HTML code for this page implements the generic left-menu laytout of our site. You can note the 4 `<div>` tags used as containers for the corresponding areas.
The page class contains the code to physically assemble page and panels:

```
package helloWorld.layoutTenda;

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

public class JugTemplate extends WebPage {
        public static final String CONTENT_ID = "contentComponent";

        private Component headerPanel;
        private Component menuPanel;
        private Component footerPanel;

        public JugTemplate(){
                add(headerPanel = new HeaderPanel("headerPanel"));
                add(menuPanel = new MenuPanel("menuPanel"));
                add(footerPanel = new FooterPanel("footerPanel"));
                add(new Label(CONTENT_ID, "Put your content here"));
        }

        //getters for layout areas
        //...
}
```

Done! Our template page is ready to be used. Now all the pages of our site will be subclasses of this parent page and they will inherit the layout and the HTML markup. They will only substitute the `Label` inserted as content area with their custom content.

3.3.3 Final example

As final example we will build the login page for our site. We will call it `SimpleLoginPage`. First, we

need a panel containing the login form. This will be the content area of our page. We will call it `LoginFormPanel` and the markup is the following:

```
<html>
<head>
...
</head>
<body>
   <wicket:panel>
    <div style="margin: auto; width: 40%;">
       <form  id="loginForm" method="get">
         <fieldset id="login" class="center">
            <legend >Login</legend>
            <span >Username: </span><input type="text" id="username"/><br/>
            <span >Password: </span><input type="password" id="password" />
            <p>
               <input type="submit" name="login" value="login"/>
            </p>
         </fieldset>
      </form>
    </div>
   </wicket:panel>
</body>
</html>
```

The class for this panel just extends `Panel` class so we won't see the relative code. The form of this panel is for illustrative purpose only. We will see how to work with Wicket forms in chapters 9 and 10.

Since this is a login page we don't want it to display left menu area. That's not a big deal as `Component` class exposes a method called `setVisible` which sets whether the component and its children should be displayed.

The resulting Java code for login page is the following:

```
package helloWorld.layoutTenda;
import helloWorld.LoginPanel;
import org.apache.wicket.event.Broadcast;
import org.apache.wicket.event.IEventSink;

public class SimpleLoginPage extends JugTemplate {
        public SimpleLoginPage(){
                super();
                replace(new LoginPanel(CONTENT_ID));
                getMenuPanel().setVisible(false);
        }
}
```

Obviously this page doesn't come with a related markup file. You can see the final page in the following picture:

Illustration 3.7: Final login page

3.4 Markup inheritance with `<wicket:extend>` tag

With Wicket we can apply markup inheritance using another approach based on tag `<wicket:child>`. This tag is used inside parent's markup to define where children pages/panels can “inject” their custom markup extending the markup inherited from parent component.

An example of parent page using tag `<wicket:child>` is the following:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  ...
</head>
<body>
       This is parent body!
       <wicket:child/>
</body>
</html>
```

The markup of a child page/panel must be placed inside tag `<wicket:extend>`. Only the markup inside `<wicket:extend>` will be included in final markup. Here is an example of child page markup:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  ...
</head>
<body>
       <wicket:extend>
          This is child body!
       </wicket:extend>
</body>
</html>
```

Considering the two pages seen above, the final markup generated for child page will be the following:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  ...
</head>
<body>
```

```
            This is parent body!
        <wicket:child>
        <wicket:extend>
            This is child body!
        </wicket:extend>
        </wicket:child>
</body>
</html>
```

3.4.1 Our example revisited

Applying `<wicket:child>` tag to our layout example, we obtain the following markup for the main template page:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  ...
</head>
<body>
<div id="header" wicket:id="headerPanel">header</div>
<div id="body">
        <div id="menu" wicket:id="menuPanel">menu</div>
        <wicket:child/>
</div>
<div id="footer" wicket:id="footerPanel">footer</div>
</body>
</html>
```

We have replaced the `<div>` tag of the content area with tag `<wicket:child>`. Going ahead with our example we can build a login page creating class `SimpleLoginPage` which extends page `JugTemplate`, but with a related markup file like this:

```
<html>
<head>
...
</head>
<body>
   <wicket:extend>
    <div style="margin: auto; width: 40%;">
       <form  id="loginForm" method="get">
         <fieldset id="login" class="center">
            <legend >Login</legend>
            <span >Username: </span><input type="text" id="username"/><br/>
            <span >Password: </span><input type="password" id="password" />
            <p>
               <input type="submit" name="login" value="login"/>
            </p>
         </fieldset>
      </form>
    </div>
   </wicket:extend>
</body>
</html>
```

As we can see this approach doesn't require to create custom panels to use as content area and it can

be useful if we don't have to handle a GUI with a high degree of complexity.

3.5 Summary

Wicket applies inheritance also to HTML markup making layout management much easier and less error-prone. Defining a master template page to use as base class for the other pages is a great way to build a consistent layout and use it across all the site.

During the chapter we have also introduced component `Panel`, a very important Wicket class that is primarily designed to let us divide our pages in smaller and reusable UI components.

4 Keeping control over HTML

Many Wicket newbies are initially scared by its approach to web development because they have the impression that the component-oriented nature of the framework prevents them from having direct control over the generated markup. This is due to the fact that many developers come from other server-side technologies like JSP where we physically implement the logic that controls how the final HTML is generated.

This chapter will avoid you any initial misleading feeling about Wicket showing how to control and manipulate the generated HTML with the built-in tools shipped with the framework.

4.1 Hiding or disabling a component

At the end of the previous chapter we have seen how to hide a component calling its method `setVisible`. In a similar fashion, we can also decide to *disable* a component using method `setEnabled`. When a component is disabled all the links inside it will be in turn disabled (they will be rendered as `<span>`)[9] and it can not fire JavaScript events.

Class `Component` provides two getter methods to determinate if a component is visible or enabled: `isVisible` and `isEnabled`.

Even if nothing prevents us from overriding these two methods to implement a custom logic to determinate the state of a component, we should keep in mind that methods `isVisible` and `isEnabled` are called multiple times before a component is fully rendered. Hence, if we place non-trivial code inside these two methods, we can sensibly deteriorate the responsiveness of our pages.

As we will see in the next chapter, class `Component` provides method `onConfigure` which is more suited to contain code that contributes to determinate component states because it is called just once during rendering phase.

4.2 Modifing tag attributes

To modify tag attributes we can use class `org.apache.wicket.AttributeModifier`. This class extends `org.apache.wicket.behavior.Behavior` and can be added to any component with `Component`'s method `add`. Class `Behavior` is used to expand component functionalities and it can also modify component markup. We will see this class in detail later in paragraph 15.1.

As first example of attribute manipulation let's consider a `Label` component bound to the following markup:

```
<span wicket:id="simpleLabel"></span>
```

Suppose we want to add some style to label content making it red and bolded. We can add to the label an `AttributeModifier` which creates the tag attribute `style` with value `"color:red;font-weight:bold"`:

```
label.add(new AttributeModifier("style", "color:red;font-weight:bold"));
```

If attribute `style` already exists in the original markup, it will be replaced with the value specified by `AttributeModifier`. If we don't want to overwrite the existing value of an attribute we can use

9 *The markup used to render disabled links can be customized as described at the end of paragraph 8.3*

subclass `AttributeAppender` which will *append* its value to the existing one:

```
label.add(new AttributeAppender("style", "color:red;font-weight:bold"));
```

We can also create attribute modifiers using factory methods provided by class `AttributeModifier` and it's also possible to *prepend* a given value to an existing attribute:

```
//replaces existing value with the given one
label.add(AttributeModifier.replace("style", "color:red;font-weight:bold"));
```

```
//appends the given value to the existing one
label.add(AttributeModifier.append("style", "color:red;font-weight:bold"));
```

```
//prepends the given value to the existing one
label.add(AttributeModifier.prepend("style", "color:red;font-weight:bold"));
```

4.3 Generating tag attribute 'id'

Tag attribute `id` plays a crucial role in web development as it allows JavaScript to identify a DOM element. That's why class `Component` provides two dedicated methods to set this attribute.

With method `setOutputMarkupId(boolean output)` we can decide if the `id` attribute will be rendered or not in the final markup (by default is not rendered). The value of this attribute will be automatically generated by Wicket and it will be unique for the entire page. If we need to specify this value by hand, we can use method `setMarkupId(String id)`.

The value of the id can be retrieved with method `getMarkupId()`.

4.4 Creating panels “on the fly” with WebMarkupContainer

Create custom panels is a great way to handle complex user interfaces. However, sometimes we may need to create a panel which is used only by a specific page and only for a specific task.

In situations like these component `org.apache.wicket.markup.html.WebMarkupContainer` is better suited than custom panels because it can be directly attached to a tag in the parent markup without needing a corresponding html file (hence it is less reusable).

Let's consider for example the main page of a mail service where users can see a list of received mails. Suppose that this page shows a notification box where user can see if new messages are arrived. This box must be hidden if there are no messages to display and it would be nice if we could handle it as if it was a Wicket component.

Suppose also that this information box is a `<div>` tag like this inside the page:

```
<div wicket:id="informationBox">
//here's the body
...
   You've got <span wicket:id="messagesNumber"></span> new messages.
...
</div>
```

Under these conditions we can consider to use a `WebMarkupContainer` component rather than implementing a new panel. The code needed to handle the information box inside the page could be the following:

```
//Page initialization code
...
WebMarkupContainer informationBox = new WebMarkupContainer ("informationBox");
informationBox.add(new Label("messagesNumber", messagesNumber));
add(informationBox);
...
//If there are no new messages, hide informationBox
informationBox.setVisible(false);
```

As you can see in the snippet above we can handle our information box from Java code as we do with any other Wicket component.

4.5 Working with markup fragments

Another circumstance in which we may prefer to avoid the creation of custom panels is when we want to conditionally display in a page small fragments of markup. In this case if we decided to use panels, we would end up having a huge number of small panel classes with their related markup file.

To better cope with situations like this, Wicket defines component `Fragment` in package `org.apache.wicket.markup.html.panel`. Just like its parent component `WebMarkupContainer`, `Fragment` doesn't have an own markup file but it uses a markup *fragment* defined in the markup file of its container, which can be a page or a panel. The fragment must be delimited with tag `<wicket:fragment>` and must be identified by a `wicket:id` attribute. In addition to the component id, `Fragment`'s constructor takes in input also the id of the fragment and a reference to its container.

In the following example we have defined a fragment in a page and we used it as content area:

Page markup:

```
<html>
  ...
<body>
...
       <div wicket:id="contentArea"></div>
       <wicket:fragment wicket:id="fragmentId">
          <!-- Fragment markup goes here -->
       </wicket:fragment>
</body>
</html>
```

Page code:

```
Fragment fragment = new  Fragment ("contentArea", "fragmentId", this);
add(fragment);
```

Fragments can be very helpful with complex pages or components. For example let's say that we have a page where users can register to our forum. This page should first display a form where user must insert his/her personal data (name, username, password, email and so on), then, once user has submitted[10] the form, the page should display a message like “Your registration is complete! Please check your mail to activate your user profile.”.

Instead of displaying this message with a new component or in a new page, we can define two

10 *Form component will be introduced in chapter 9*

fragments: one for the initial form and one to display the confirmation message. The second fragment will replace the first one after the form has been submitted:

Page markup:

```
<html>
  ...
<body>
...
        <div wicket:id="contentArea"></div>
        <wicket:fragment wicket:id="formFrag">
           <!-- Form markup goes here -->
        </wicket:fragment>
        <wicket:fragment wicket:id="messageFrag">
           <!-- Message markup goes here -->
        </wicket:fragment>
</body>
</html>
```

Page code:

```
Fragment fragment = new  Fragment ("contentArea", "formFrag", this);
add(fragment);
//...
//form has been submitted
Fragment fragment = new  Fragment ("contentArea", "messageFrag", this);
replace(fragment);
```

4.6 Adding header contents to the final page

Panel's markup can also contain HTML tags which must go inside header section of the final page, like tags `<script>` or `<style>`. To tell Wicket to put these tags inside page `<head>`, we must surround them with tag `<wicket:head>`.

Considering the markup of a generic panel, we can use tag `<wicket:head>` in this way:

```
<wicket:head>
        <script type="text/javascript">
                function myPanelFunction(){
                }
          </script>

        <style>
         .myPanelClass{
                font-weight: bold;
                color: red;
          }
        </style>
</wicket:head>
<body>
        <wicket:panel>
         ...
        </wicket:panel>
         ...
</body>
```

Wicket will take care of placing the content of `<wicket:head>` inside the `<head>` tag of the final page.

Note

Tag `<wicket:head>` can be used also with children pages/panels which extend parent markup using tag `<wicket:extend>`.

Note

The content of tag `<wicket:head>` is added to header section *once per component class*. In other words, if we add multiple instances of the same panel to a page, the `<head>` tag will be populated just once with the content of `<wicket:head>`.

Warning

Tag `<wicket:head>` is ideal if we want to define small in-line blocks of CSS or JavaScript. However Wicket provides also a more sophisticated technique to let components contribute to header section with in-line blocks and resource files like CSS or JavaScript files. We will see this technique later in chapter 13.

4.7 Using stub markup in our pages/panels

Wicket tag `<wicket:remove>` can be very useful when our web designer needs to show us how a page or a panel should look like. The markup inside this tag will be stripped out in the final page, so it's the ideal place for web designers to put their stub markup:

```
<html>
<head>
...
</head>
<body>
...
       <wicket:remove>
          <!-- Stub markup goes here -->
       </wicket:remove>
</body>
</html>
```

4.8 How to render component body only

When we bind a component to its corresponding tag we can choose to get rid of this outer tag in the final markup. If we call method `setRenderBodyOnly(true)` on a component Wicket will remove the surrounding tag.

For example given the following markup and code:

Markup:

```
<html>
<head>
  <title>Hello world page</title>
</head>
```

```
<body>
<div wicket:id="helloWorld">[helloWorld]</div>
</body>
</html>
```

Java code:

```
Label label = new Label("helloWorld", "Hello World!");
label.setRenderBodyOnly(true);
add(label);
```

the output will be:

```
<html>
<head>
  <title>Hello world page</title>
</head>
<body>
 Hello World!
</body>
</html>
```

As you can see the `<div>` tag used for component `Label` is not present in the final markup.

4.9 Hiding decorating elements with tag *<wicket:enclosure>*

Our data are rarely displayed alone without a caption or other graphic elements that make clear the meaning of their value. For example:

```
<label>Total amount: </label><span wicket:id="totalAmount"></span>
```

Wicket comes with a nice utility tag called `<wicket:enclosure>` that automatically hides those decorating elements if the related data value is not visible. All we have to do is to put the involved markup inside this tag. Applying `<wicket:enclosure>` to the previous example we get the following markup:

```
<wicket:enclosure>
    <label>Total amount: </label><span wicket:id="totalAmount"></span>
</wicket:enclosure>
```

Now if component `totalAmount` is not visible, its description (`Total amount:`) will be automatically hidden. If we have more then a Wicket component inside `<wicket:enclosure>` we can use `child` attribute to specify which component will control the overall visibility:

```
<wicket:enclosure child="totalAmount">
    <label>Total amount: </label><span wicket:id="totalAmount"></span><br/>
    <label>Expected delivery date: </label><span wicket:id="delivDate"></span>
</wicket:enclosure>
```

`child` attribute supports also nested components with a colon-separated path:

```
<wicket:enclosure child="totalAmountContainer:totalAmount">
    <div wicket:id="totalAmountContainer">
        <label>Total amount: </label><span wicket:id="totalAmount"></span>
    </div>
    <label>Expected delivery date: </label><span wicket:id="delivDate"></span>
</wicket:enclosure>
```

4.10 Surrounding existing markup with Border

Component `org.apache.wicket.markup.html.border.Border` is a special purpose container created to enclose its tag body with its related markup. Just like panels and pages also borders have an own markup file which is defined following the same rules seen form panels and pages. In this file tag `<wicket:border>` is used to indicate which part of the content is to be considered as border markup:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:wicket="http://wicket.apache.org">
<head></head>
<body>
    <!--  everything above <wicket:border> tag will be discarded...-->
    <wicket:border>
        <div>
           foo<br />
           <wicket:body/><br />
          buz <br />

         </div>
    </wicket:border>
    <!--  everything below </wicket:border> tag will be discarded...-->
</body>
</html>
```

The tag `<wicket:body/>` used in the example above is used to indicate where the body of the tag will be placed inside border markup. Now if we attached this border to the following tag

```
<span wicket:id="myBorder">
  bar
</span>
```

we would obtain the following resulting HTML:

```
<span wicket:id="myBorder">
       <div>
         foo<br />
         bar<br />
         buz <br />
       </div>
</span>
```

`Border` can also contain children components which can be placed either inside its markup file or

inside its corresponding HTML tag. In the first case children must be added to border component with method `addToBorder(Component...)`, while in the second case we must use the usual method `add(Component...)`.

In the following table you can see an example that illustrates both cases:

Border class:

```
public class MyBorder extends Border {

        public MyBorder(String id) {
                super(id);
        }

}
```

Border markup:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:wicket="http://wicket.apache.org">
<head></head>
<body>
    <wicket:border>
        <div>
           <div wicket:id="childMarkup"></div>
           <wicket:body/><br />
         </div>
    </wicket:border>
</body>
</html>
```

Border tag:

```
<div wicket:id="myBorder">
  <span wicket:id="childTag"></span>
</div>
```

Initialization code for border[11]:

```
MyBorder myBorder = new MyBorder("myBorder");

myBorder.addToBorder(new Label("childMarkup", "Child inside markup."));
myBorder.add(new Label("childTag", "Child inside tag."));

add(myBorder);
```

4.11 Summary

In this chapter we have seen the tools provided by Wicket to gain a complete control over the generated HTML. However we didn't see yet how we can *repeat* a portion of HTML with Wicket. With classic server-side technologies like PHP or JSP we use to do this by using loop constructors (like `while` or `for`) inside our pages.

11 *Initialization code should go inside component's constructor or inside its method onInitialize (we will see this method in the next chapter)*

To perform this task Wicket provides a special-purpose family of components called *repeaters* and designed to repeat their markup body to display a set of items.

But to fully understand how these components work, we must learn first some other basic topics of Wicket. That's why repeaters will be introduced later in chapter 11.

5 Components lifecycle

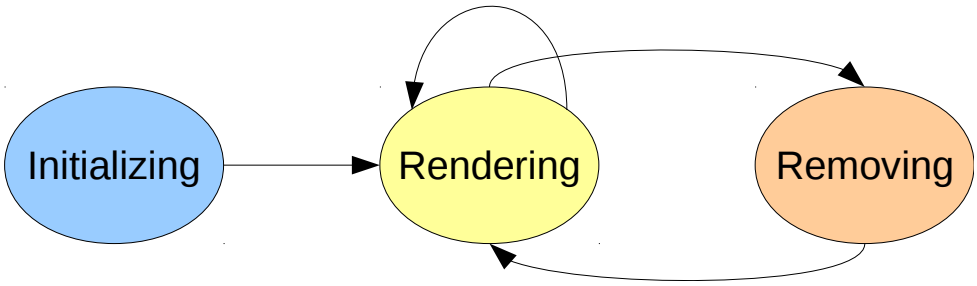
Just like applets[12] and servlets[13], also Wicket components follow a lifecycle during their existence. In this chapter we will analyse each stage of this cycle and we will learn how to make the most of the hook methods that are triggered when a component moves from a stage to another.

5.1 Lifecycle stages of a component

During its life a Wicket component goes through three basic stages:

1. **Initialization:** Component is instantiated by Wicket and prepared for rendering phase.
2. **Rendering:** in this stage Wicket generates component markup. If component contains children (i.e. is a subclass of `MarkupContainer`) it must first wait for them to be rendered before starting its own rendering.
3. **Removing:** this stage is triggered when a component is explicitly removed from its component hierarchy, i.e. when its parent invokes `remove(component)` on it. This stage is facultative and is never triggered for pages.

The following picture shows the state diagram of component lifecycle:



Once a component has been removed it can be added again to a container, but the initialization stage won't be executed again.

Note

If you read the JavaDoc of class `Component` you will find a more detailed description of component lifecycle.
However this description introduces some advanced topics we didn't covered yet hence, to avoid confusion, in this chapter some details have been omitted and they will be covered later in the next chapters.
For now you can consider just the simplified version of the lifecycle described above.

5.2 Hook methods for component lifecycle

Class `Component` comes with a number of hook methods that can be overridden in order to customize component behavior during its lifecycle.

12 *See http://download.oracle.com/javase/tutorial/deployment/applet/lifeCycle.html*
13 *See "Servlet Life Cycle " paragraph of Servlet Specification document*

In the following table these methods are grouped according to the stage in which they are invoked:

Cycle stage	Involved methods
Initialization	• onInitialize
Rendering	• onConfigure • onBeforeRender • onRender • onAfterRender • onAfterRenderChildren • onComponentTag • onComponentTagBody
Removing	• onRemove

Now let's take a closer look at each stage and to its hook methods.

5.3 *Initialization* stage

This stage is performed at the beginning of component lifecycle. During initialization, component has already been inserted into its component hierarchy so we can safely access to its parent container or to its page with methods `getParent()` or `getPage()`.

The only method triggered during this stage is `onInitialize()`. This method is a sort of “special” constructor where we can execute a more sophisticated initialization of our component.

Since `onInitialize` is similar to a regular constructor, when we override this method we have to call `super.onInitialize` inside its body, usually as first instruction.

5.4 *Rendering* stage

This stage is triggered each time a component is rendered by Wicket, typically when its page is requested or when it is refreshed via AJAX[14].

5.4.1 Method onBeforeRender

The most important hook method of this stage is probably `onBeforeRender()`. This method is called before component starts its rendering phase and it is our last chance to change its children hierarchy. If we want add/remove children components this is the right place to do it.

In the next example (project *LifeCycleStages*) we will create a page which alternately displays two different labels, swapping between them each time it is rendered:

```
public class HomePage extends WebPage
{
        private Label firstLabel;
        private Label secondLabel;

        public HomePage(){
                firstLabel = new Label("label", "First label");
                secondLabel = new Label("label", "Second label");
```

14 *AJAX support will be discussed in chapter 16.*

```
                add(firstLabel);
                add(new Link("reload"){
                        @Override
                        public void onClick() {
                        }
                });
        }

        @Override
        protected void onBeforeRender() {
                if(contains(firstLabel, true))
                        replace(secondLabel);
                else
                        replace(firstLabel);

                super.onBeforeRender();
        }
}
```

The code inside `onBeforeRender()` is quite trivial as it just checks which label among `firstLabel` and `secondLabel` is currently inserted into component hierarchy and it replaces the inserted label with the other one.

This method is also responsible for invoking children `onBeforeRender()` so if we decide to override it we have to call `super.onBeforeRender()`. However, unlike `onInitialize()`, the call to superclass method should be placed at the end of method's body in order to affect children's rendering with our custom code.

Please note that in the example above we can trigger the rendering stage pressing F5 key or clicking on link “reload”.

Warning

If we forget to call superclass version of methods `onInitialize()` or `onBeforeRender()`, Wicket will throw an `IllegalStateException` with the following message:

```
java.lang.IllegalStateException: org.apache.wicket.Component has not been
properly initialized. Something in the hierarchy of <page class name> has
not called super.onInitialize()/onBeforeRender() in the override of
onInitialize()/ onBeforeRender() method
```

5.4.2 Method onConfigure

Method `onConfigure()` has been introduced in order to provide a good point to manage component states such as visibility or enabling. This method is called before the beginning of rendering phase.

As stated in paragraph 4.1, `isVisible` and `isEnabled` are called multiple times when a page or a component is rendered, so is highly recommended to not directly override these method, but rather to use `onConfigure` to change component states.

On the contrary method `onBeforeRender` is not indicated for this task because it will not be invoked if component visibility is set to false.

5.4.3 Method onComponentTag

Method `onComponentTag(ComponentTag)` is called to process component tag, which can be freely manipulated through its argument of type `org.apache.wicket.markup.ComponentTag`. For example we can add/remove tag attributes with methods `put(String key, String value)` and

`remove (String key)`, or we can even decide to change the tag renaming it with method `setName(String)` (the following code is taken from project *OnComponentTagExample*):

Markup code:

```
<head>
  <meta charset="utf-8" />
  <title></title>
</head>
<body>
  <h1 wicket:id="helloMessage"></h1>
</body>
```

Java code:

```
public class HomePage extends WebPage {
   public HomePage() {
      add(new Label("helloMessage", "Hello World"){
         @Override
         protected void onComponentTag(ComponentTag tag) {
            super.onComponentTag(tag);
            //Turn the h1 tag to a span
            tag.setName("span");
            //Add formatting style
            tag.put("style", "font-weight:bold");
         }
      });
      //...
    }
}
```

Gnerated markup:

```
<head>
  <meta charset="utf-8" />
  <title></title>
</head>
<body>
  <span wicket:id="helloMessage" style="font-weight:bold">Hello World</span>
</body>
```

Just like we do with `onInitialize`, if we decide to override `onComponentTag` we must remember to call the same method on the super class because also this class may need to customize the tag.

Overriding `onComponentTag` is perfectly fine if we have to customize the tag of a specific component, but if we wanted to reuse the code across different components we should consider to use a behavior in place of this hook method.

We have already seen in paragraph 4.2 how to use behavior `AttributeModifier` to manipulate tag's attribute. In paragraph 15.1 we will see that base class `Behavior` offers also a callback method named `onComponentTag(ComponentTag, Component)` that can fully replace hook method `onComponentTag(ComponentTag)`.

5.4.4 Methods onComponentTagBody

Method `onComponentTagBody(MarkupStream, ComponentTag)` is called to process component tag's body. Just like `onComponentTag` it takes in input a `ComponentTag` parameter representing the

component tag. In addition, we find also a `MarkupStream` parameter which represents the page markup stream that will be sent back to the client as response.

`onComponentTagBody` can be used in combination with `Component`'s method `replaceComponentTagBody` to render a custom body under specific conditions. For example (taken from project *OnComponentTagExample*) we can display a brief description instead of the body if label component is disabled:

```
public class HomePage extends WebPage {
   public HomePage() {
      //...
      add(new Label("helloMessage", "Hello World"){
         @Override
         protected void onComponentTagBody(MarkupStream markupStream, ComponentTag tag) {

           if(!isEnabled())
               replaceComponentTagBody(markupStream, tag, "(the component is disabled)");
           else
               super.onComponentTagBody(markupStream, tag);
         }
      });
    }
}
```

Please note that the original version of `onComponentTagBody` is invoked only when we want to preserve the standard rendering mechanism for tag's body (in our example this happens when the component is enabled).

5.5 *Removing* stage

This stage is triggered when a component is removed from its component hierarchy. The only hook method for this phase is `onRemove()`. If our component still holds some resources needed during rendering phase, we can override this method to release them.

Once a component has been removed we are free to add it again to the same container or to another one.

5.6 Summary

In this chapter we have seen which stages compose the lifecycle of a Wicket component and which hook methods they provide. Overriding these methods we can dynamically modify component hierarchy and we can enrich the behavior of our custom components.

6 Page versioning and caching

This chapter explains how Wicket manages page instances, underlining the difference between stateful and stateless pages. The chapter introduces also some advanced topics like Java Serialization and multi-level cache. However, to understand what you will read you are not required to be familiar with these arguments.

6.1 Stateful pages VS stateless

Wicket pages can be divided into two categories: *stateful* and *stateless* pages. Stateful pages are those which rely on user session to store they internal state and to keep track of user interaction.

On the contrary stateless pages are those which don't change their internal state during their lifecycle and they don't need to occupy space into user session.

From the point of view of the framework the biggest difference between these two types of page is that stateful pages are *versioned,* meaning that they will be saved into user session every time their internal state has changed. Wicket automatically assigns a session to user the first time a stateful page is requested. Page versions are stored into user session using **Java Serialization** mechanism.

Stateless pages are never versioned and that's why they don't require a valid user session. If we want to know whether a page is stateless or not, we can use method `isPageStateless()` of class `Page`.

In order to build a stateless page we must comply with some rules to ensure that page won't need to use user session. These rules are illustrated in paragraph 6.3 but before talking about stateless pages we must first understand how stateful pages are handled and why they are versioned.

6.2 Stateful pages

Stateful pages are versioned in order to support browser's back button: when this button is pressed Wicket must respond rendering the same page instance previously used.

A new page version is created when a stateful page is requested for the first time or when an existing instance is modified (for example changing its component hierarchy). To identify each page version Wicket uses a session-relative identifier called **page id**. This is a progressive number and it is increased every time a new page version is created.

In the final example of the previous chapter (project *LifeCycleStages*), you may have noticed the number appended at the and of URL. This number is the page id we are talking about:

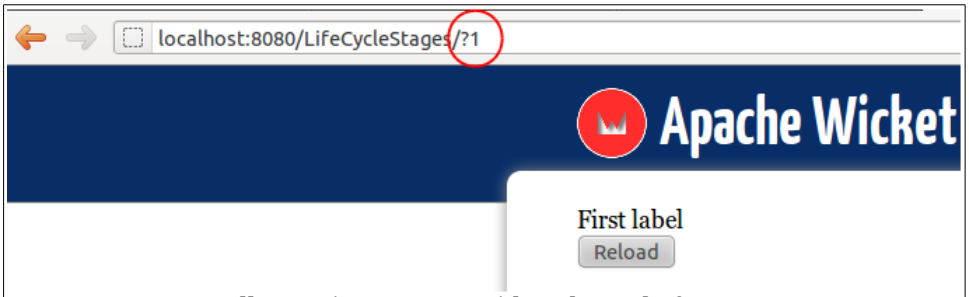


Illustration 6.1: Page id at the end of URL

In this chapter we will use a revisited version of this example project where component hierarchy is modified inside `Link`'s `onClick()` method. This is necessary because Wicket creates a new page version only if page is modified before its method `onBeforeRender()` is invoked. The code of the new

home page is the following:

```
public class HomePage extends WebPage
{
	private static final long serialVersionUID = 1L;
	private Label firstLabel;
	private Label secondLabel;

	public HomePage(){
		firstLabel = new Label("label", "First label");
		secondLabel = new Label("label", "Second label");

		add(firstLabel);

		add(new Link("reload"){
			@Override
			public void onClick() {
				if(getPage().contains(firstLabel, true))
					getPage().replace(secondLabel);
				else
					getPage().replace(firstLabel);
			}
		});

	}
}
```

Now if we run the new example (project *LifeCycleStagesRevisited*) and we click on “Reload” button, a new page version is created and page id is increased by one:

Illustration 6.2: pressing “Reload” button causes page id to increment

If we press the back button the page version previously rendered will be retrieved and it will be used again to respond to our request (and page id is decremented):

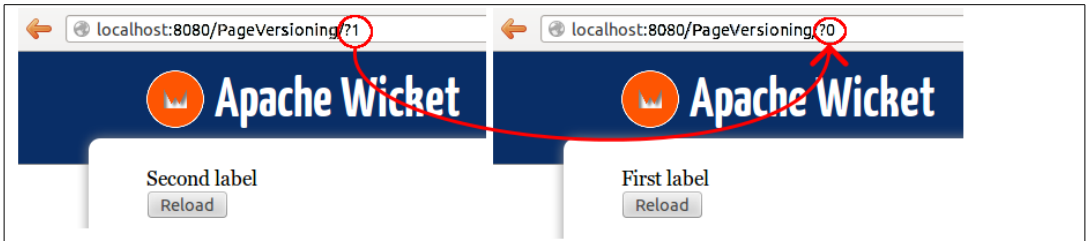


Illustration 6.3: pressing “Back” button we can go back to the previous page version (page id is decremented)

cc BY NC SA

Note

For more details about page storing you can visit the wiki page at https://cwiki.apache.org/confluence/display/WICKET/Page+Storage.
On this page you can find which classes are involved into page storing mechanism and how they work together.

As we have previously told at the beginning of this chapter, page versions are stored using Java serialization, therefore every object referenced inside a page must be serializable[15]. In paragraph 9.6 we will see how to overcome this limit and work with non-serializable objects in our components using Wicket *models*.

6.2.1 Using a specific page version with PageReference

To retrieve a specific page version in our code we can use class `org.apache.wicket.PageReference` using the corresponding page id as argument for its constructor:

```
//load page version with page id = 3
PageReference pageReference = new PageReference(3);
//load the related page instance
Page page = pageReference.getPage();
```

To get the related page instance we must use method `getPage`.

6.2.2 Turning off page versioning

If for any reason we need to switch off versioning for a given page, we can call its method `setVersioned(false)`.

6.2.3 Pluggable serialization

Starting from version 1.5 it is possible to choose which implementation of Java serialization will be used by Wicket to store page versions. Wicket serializes pages using an implementation of interface `org.apache.wicket.serialize.ISerializer`. The default implementation is `org.apache.wicket.serialize.java.JavaSerializer` and it uses the standard Java serialization mechanism based on classes `ObjectOutputStream` and `ObjectInputStream`. However on Internet we can find other interesting serialization libraries like Kryo[16] which performs faster then the standard implementation.

The serializer in use can be customized with method `setSerializer(ISerializer)` defined by setting interface `org.apache.wicket.settings.IFrameworkSettings`. We can access this interface inside `init` method of class `Application` using method `getFrameworkSettings()`:

```
@Override
public void init()
{
        super.init();
        getFrameworkSettings().setSerializer(yourSerializer);
}
```

A serializer based on Kryo library is available with project WicketStuff. You can find more information on this project, as well as the instructions to use its modules, in Appendix B.

6.2.4 Page caching

15 *It must implement standard interface java.io.Serialization*
16 *http://code.google.com/p/kryo/*

By default Wicket persists pages versions into a session-relative file on disk, but it uses a two-levels cache to speed up this process. The first level of the cache uses a http session attribute called "wicket:persistentPageManagerData-<APPLICATION_NAME>" to store pages. The second level cache stores pages into application-scoped variables which are identified by a session id and a page id.

The following picture is an overview of these two caching levels:

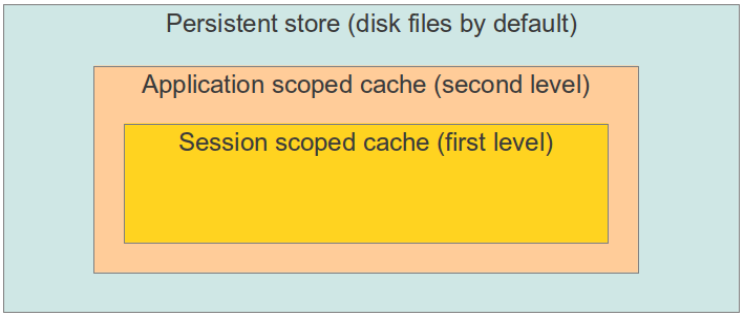


Illustration 6.4: An overview of Wicket cache structure

The session-scoped cache is faster then the other memory levels but it contains only the pages used to serve the last request.

Wicket allows us to set the maximum amount of memory allowed for the application-scoped cache and for the page store file. Both parameters can be configured using setting interface `org.apache.wicket.settings.IStoreSettings`.

The interface provides method `setMaxSizePerSession(Bytes bytes)` to set the size for page store file. The `Bytes` parameter is the maximum size allowed for this file:

```
@Override
public void init()
{
        super.init();
        getStoreSettings().setMaxSizePerSession(Bytes.kilobytes(500));
}
```

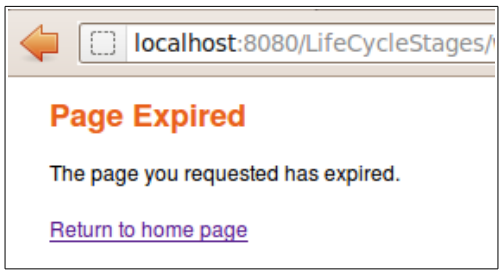
Class `org.apache.wicket.util.lang.Bytes` is an utility class provided by Wicket to express size in bytes (for further details see JavaDoc).

For the second level cache we can use method `setInmemoryCacheSize(int inmemoryCacheSize)`. The integer parameter is the maximum number of page instances that will be saved into application-scoped cache:

```
@Override
public void init()
{
        super.init();
        getStoreSettings().setInmemoryCacheSize(50);
}
```

6.2.5 Page expiration

Page instances are not kept into user session forever. They can be discarded when the limit set with method `setMaxSizePerSession` is reached or (more often) when user session expires. When we ask Wicket for a page id corresponding to a page instance removed from session, we bump into a `PageExpiredException` and we get the following default error page:



This error page can be customized with method `setPageExpiredErrorPage` of setting interface `org.apache.wicket.settings.IApplicationSettings`:

```
@Override
public void init()
{
	super.init();
	getApplicationSettings().setPageExpiredErrorPage(CustomExpiredErrorPage.class);
}
```

The page class provided as custom error page must have a public constructor with no argument or a constructor that takes in input a single `PageParameters` argument (the page must be *bookmarkable* as described in paragraph 8.1.1).

6.3 Stateless pages

Wicket makes it very easy building stateful pages, but sometimes we might want to use an “old school” stateless page that doesn't keep memory of its state into session. Think for example at the public area of a site or at a login page: in cases like these a stateful page would be a waste of resources or even a security threat, as we will see in paragraph 10.9.

In Wicket a page can be stateless only if it satisfies the following requirements:

1. it has been instantiated by Wicket (i.e. we don't create it with operator `new`) using a constructor with no argument or a constructor that takes in input a single `PageParameters` argument (class `PageParameters` will be covered in chapter 8).
2. All its children components (and behaviors[17]) are in turn stateless, which means that their method `isStateless` must return true.

The first requirement implies that, rather than creating a page by hand, we should rely on Wicket's capability of resolving page instances, like we do when we use method `setResponsePage(Class page)`.

In order to comply with the second requirement it could be helpful to check if all children components of a page are stateless. To do this we can leverage method `visitChildren` and visitor pattern to iterate over components and test if their method `isStateless` actually returns true:

```
@Override
protected void onInitialize() {
	super.onInitialize();

	visitChildren(new IVisitor<Component, Void>() {
		@Override
		public void component(Component component, IVisit<Void> arg1) {
```

17 *See method getStatelessHint in paragraph 15.1*

```
                        if(!component.isStateless())
                          System.out.println("Component " + component.getId()
                                              + " is not stateless");
                }
        });
}
```

Alternatively, we could use the utility annotation `StatelessComponent` along with class `StatelessChecker` (they are both in package `org.apache.wicket.devutils.stateless`). `StatelessChecker` will throw an `IllegalArgumentException` if a component annotated with `StatelessComponent` doesn't respect the requirements for being stateless. To use annotation `StatelessComponent` we must first add `StatelessChecker` to our application as component render listener:

```
@Override
public void init()
{
        super.init();
        getComponentPostOnBeforeRenderListeners().add(new StatelessChecker());
}
```

Note

Most of the built-in components of Wicket are stateful, hence they can not be used with a stateless page. However some of them have also a stateless version which can be adopted when we need to keep a page stateless. In the rest of the guide we will point out when a built-in component comes also with a stateless version.

A page can be also explicitly declared as *stateless* setting the appropriate flag to true with method `setStatelessHint(true)`. This method will not prevent us from violating the requirements for a stateless page, but if we do so we will get the following warning message into log stream:

```
Page '<page class>' is not stateless because of component with path '<component path>'
```

6.4 Summary

In this chapter we have seen how page instances are managed by Wicket. We have learnt that pages can be divided into two families: stateless and stateful pages. Knowing the difference between the two types of pages is important to build the right page for a given task.

However, to complete the discussion about stateless pages we still have to deal with two topics we have just outlined in this chapter: class `PageParameters` and *bookmarkable* pages. The first part of chapter 8 will cover these missing topics.

7 Under the hood of request processing

Although Wicket was born to provide a reliable and comprehensive object oriented abstraction for web development, sometimes we might need to work directly with “raw” web entities such as user session, web request, query parameters, and so on. For example this is necessary if we want to store an arbitrary parameter into user session.

Wicket provides some wrapper classes that allow us to easily access to web entities without the burden of using the low-level APIs of Java Servlet Specification. However it will always be possible to access standard classes (like `HttpSession`, `HttpServletRequest`, etc...) that lay under our Wicket application.

This chapter will introduce these wrapper classes and it will explain how Wicket uses them to handle web requests coming from user.

7.1 Class Application and request processing

Besides configuring and initializing our application, class `Application` is responsible for creating the internal entities used by Wicket to process a request. These entities are instances of the following classes: `RequestCycle`, `Request`, `Response` and `Session`.

The next paragraphs will illustrate each of these classes, explaining how they are involved into request processing.

7.2 Classes *Request* and *Response*

Classes `Request` and `Response` inside package `org.apache.wicket.request` provide an abstraction of the concrete request and response used by our application.

Both classes are declared as abstract but if our application class inherits from `WebApplication` it will use their sub classes `ServletWebRequest` and `ServletWebResponse`, both of them inside package `org.apache.wicket.protocol.http.servlet`.

`ServletWebRequest` and `ServletWebResponse` wrap respectively a `HttpServletRequest` and a `HttpServletResponse` object. If we need to access to these low-level objects we can call `Request`'s method `getContainerRequest()` and `Response`'s method `getContainer Response()`.

7.3 The “director” of request processing: RequestCycle

Class `org.apache.wicket.request.cycle.RequestCycle` is the entity in charge of serving a web request. Our application class creates a new `RequestCycle` on every request with its method `createRequestCycle(request, response)`.

Method `createRequestCycle` is declared as `final`, so we can't override it to return a custom subclass of `RequestCycle`. Instead, we must build a *request cycle provider* implementing interface `org.apache.wicket.IRequestCycleProvider`, and then we must tell our application class to use it with `Application`'s method `setRequestCycleProvider`.

The current running request cycle can be retrieved at any time by calling its static method `RequestCycle.get()`. Strictly speaking this method returns the request cycle *associated with the current (or local) thread,* which is the thread that is serving the current request.

A similar `get()` method is also implemented in classes `org.apache.wicket.Application` (as we

have seen in paragraph 2.2.2) and `org.apache.wicket.Session` to get the application and the session in use with the current thread.

Note

The implementation of `get` method takes advantage of the standard class `java.lang.ThreadLocal`. See its JavaDoc for an introduction to local-thread variables.

Class `org.apache.wicket.Component` provides method `getRequestCycle()` which is a convenience method that internally invokes `RequestCycle.get()`:

```
public final RequestCycle getRequestCycle()
{
        return RequestCycle.get();
}
```

7.3.1 RequestCycle and request processing

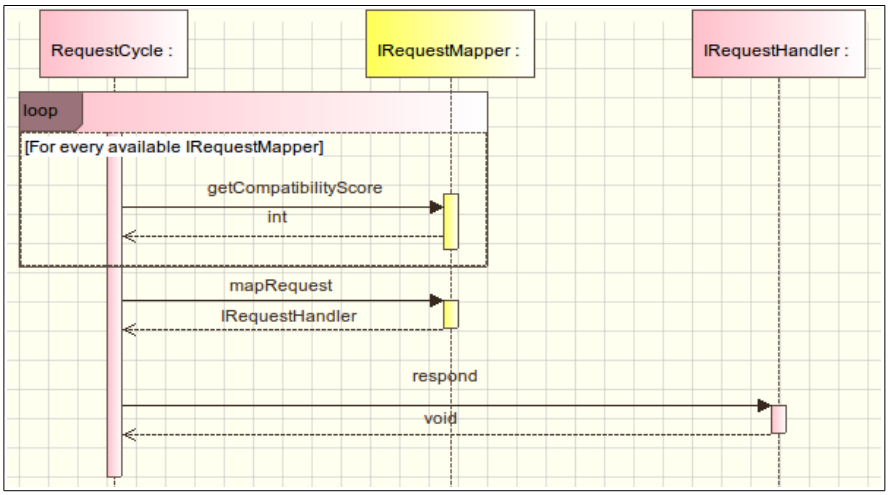
Note

This paragraph will provide just the basic informations about what happens behind the scenes of request processing. Working with Wicket it's likely that we will rarely need to customize this process, so we won't cover this topic in detail.

In order to process a request, `RequestCycle` delegates the task to another entity which implements interface `org.apache.wicket.request.IRequestHandler`. There are different implementations of this interface, each suited for a particular type of requested resource (a page to render, an AJAX request, an URL to an external page, etc...).

To resolve the right handler for a given request `RequestCycle` uses a set of objects implementing interface `org.apache.wicket.request.IRequestMapper`. This interface defines method `getCompatibilityScore(Request request)` which returns a score indicating how compatible the request mapper is for the current request. `RequestCycle` will choose the mapper with the highest score and it will call its method `mapRequest(Request request)` to get the proper handler for the given request. Once `RequestCycle` has resolved a request handler, it invokes its method `respond(IRequestCycle requestCycle)` to start request processing.

The following sequence diagram recaps how a request handler is resolved by `RequestCycle`:



Developers can create additional implementations of `IRequestMapper` and add them to their application with `WebApplication`'s method `mount(IRequestMapper mapper)`. In paragraph 8.6 we will see how Wicket uses this method to add built-in mappers for *mounted* pages.

7.3.2 Generating url with methods *urlFor* and *mapUrlFor*

`RequestCycle` is also responsible for generating the URL value (as `CharSequence`) for the following entities:

- a page class, with method `urlFor(Class<C> pageClass, PageParameters parameters)`
- an `IRequestHandler` with method `urlFor(IRequestHandler handler)`
- a `ResourceReference` with method `urlFor(ResourceReference reference, PageParameters params)` (resource entities will be introduced in chapter 13).

The methods above have also a corresponding version that returns an instance of `org.apache.wicket.request.Url` instead of a `CharSequence`. This version has the prefix 'map' in its name (i.e. it has `mapUrlFor` as full name).

7.3.3 Method setResponsePage

Class `RequestCycle` contains the implementation of method `setResponsePage` we use to redirect user to a specific page (see paragraph 2.4). The namesake method of class `org.apache.wicket.Component` is just a convenience method that internally invokes the actual implementation on current request cycle:

```
public final void setResponsePage(final Page page)
{
        getRequestCycle().setResponsePage(page);
}
```

7.3.4 RequestCycle's hook methods and listeners

`RequestCycle` comes with some hook methods which can be overridden to perform custom actions when request handling reaches a specific stage. These methods are:

- **onBeginRequest():** called when `RequestCycle` is about to start handling the request.
- **onEndRequest():** called when `RequestCycle` has finished to handle the request
- **onDetach():** called after request handling has completed and `RequestCycle` is about to be detached from its thread. The default implementation of this method invokes `detach()` on current session (`Session` class will be shortly discussed in paragraph 7.4).

Methods `onBeforeRequest` and `onEndRequest` can be used if we need to execute custom actions before and after business code is executed, such as opening a Hibernate/JPA session and closing it when code has terminated.

A more flexible way to interact with request processing is to use the listener interface `org.apache.wicket.request.cycle.IRequestCycleListener`. In addition to the three methods already seen for `RequestCycle`, this interface offers some further hooks into request processing:

- **onBeginRequest(RequestCycle cycle):** *(see the description above)*
- **onEndRequest(RequestCycle cycle):** *(see the description above)*
- **onDetach(RequestCycle cycle):** *(see the description above)*
- **onRequestHandlerResolved(RequestCycle cycle, IRequestHandler handler):** called when an `IRequestHandler` has been resolved.
- **onRequestHandlerScheduled(RequestCycle cycle, IRequestHandler handler):** called when

an `IRequestHandler` has been scheduled for execution.

- **onRequestHandlerExecuted(RequestCycle cycle, IRequestHandler handler):** called when an `IRequestHandler` has been executed.
- **onException(RequestCycle cycle, Exception ex):** called when an exception has been thrown during request processing.
- **onExceptionRequestHandlerResolved(RequestCycle rc, IRequestHandler rh, Exception ex):** called when an `IRequestHandler` has been resolved and will be used to handle an exception.
- **onUrlMapped(RequestCycle cycle, IRequestHandler handler, Url url):** called when an URL has been generated for an `IRequestHandler` object.

To use request cycle listeners we must add them to our application which in turn will pass them to the new `RequestCycle`'s instances created by `createRequestCycle`:

```
@Override
public void init(){
       super.init();
       IRequestCycleListener myListener;
       //listener initialization...
       getRequestCycleListeners().add(myListener)
}
```

Method `getRequestCycleListeners` returns an instance of class `org.apache.wicket.request.cycle.RequestCycleListenerCollection`. This class is a sort of typed collection for `IRequestCycleListener` and it also implements the Composite pattern[18].

7.4 Class Session

In Wicket we use class `org.apache.wicket.Session` to handle session-relative informations such as client informations, session attributes, session-level cache (seen in paragraph 6.2.4), etc...

In addition, we know from paragraph 6.1 that Wicket creates a user session to store versions of stateful pages. Similarly to what happens with `RequestCycle`, the new `Session`'s instances are generated by `Application` class with method `newSession(Request request, Response response)`. This method is not declared as `final`, hence it can be overridden if we need to use a custom implementation of class `Session`.

By default if our custom application class is a subclass of `WebApplication`, `newSession` will return an instance of class `org.apache.wicket.protocol.http.WebSession`.

As we have already mentioned talking about `RequestCycle`, also class `Session` provides a static `get()` method which returns the session associated to the current thread.

7.4.1 Session and listeners

Just like `RequestCycle` also class `org.apache.wicket.Session` offers support for listener entities. With `Session` these entities must implement the callback interface `org.apache.wicket.ISessionListener` which exposes only method `onCreated(Session session)`. As you might guess from its name, this method is called when a new session is created. Session listeners must be added to our application using a typed collection, just like we have done before with request cycle listeners:

```
@Override
```

18 http://en.wikipedia.org/wiki/Composite_pattern

```
public void init(){
        super.init();
        //listener initialization...
        ISessionListener myListener;
        //add a custom session listener
        getSessionListeners().add(myListener)
}
```

7.4.2 Handling session attributes

Class `Session` handles session attributes in much the same way as the standard interface `javax.servlet.http.HttpSession`. The following methods are provided to create, read and remove session attributes:

- `setAttribute(String name, Serializable value)`: creates an attribute identified by the given name. If session already contains an attribute with the same name, the new value will replace the existing one. The value must be a serializable object.
- `getAttribute(String name)`: returns the value of the attribute identified by the given name, or `null` if the name does not correspond to any attribute.
- `removeAttribute(String name)`: removes the attribute identified by the given name.

By default class `WebSession` will use the underlying http session to store attributes. Wicket will automatically add a prefix to the name of the attributes. This prefix is returned by `WebApplication`'s method `getSessionAttributePrefix()`.

7.4.3 Accessing to http session

If for any reason we need to directly access to the underlying `HttpSession` object, we can retrieve it from the current request with the following code:

```
HttpSession session = ((ServletWebRequest)RequestCycle.get().getRequest())
                      .getContainerRequest().getSession();
```

Using the raw session object can be necessary if we have to set a session attribute with a particular name without the prefix added by Wicket. Let's say for example that we are working with Tomcat as web server. One of the administrative tools provided by Tomcat is a page listing all the active user sessions of a given web application:

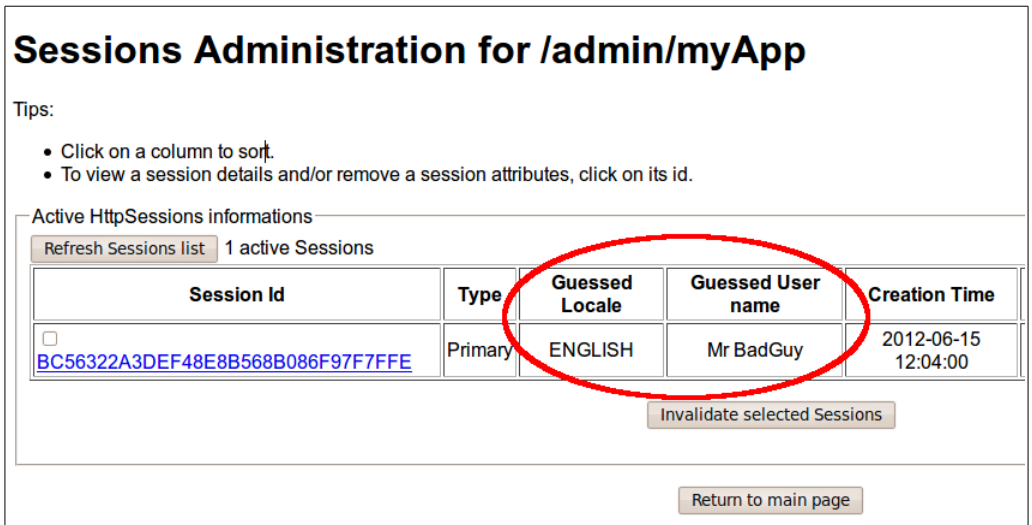


Illustration 7.1: Tomcat Sessions Administration page with custom values.

Tomcat allows us to set the values that will be displayed in columns “Guessed locale” and “Guessed User name”. One possible way to do this is to use session attributes named “Locale” and “userName” but we can't create them using Wicket `Session` class because they wouldn't have exactly the name required by Tomcat. Instead, we must use the raw `HttpSession` and set our attributes on it:

```
HttpSession session = ((ServletWebRequest)RequestCycle.get().getRequest())
                      .getContainerRequest().getSession();

session.setAttribute("Locale", "ENGLISH");
session.setAttribute("userName", "Mr BadGuy");
```

7.4.4 Temporary and permanent sessions

Wicket doesn't need to store data into user session as long as user visits only stateless pages. Nonetheless, even under these conditions, a *temporary* session object is created to process each request but it is discarded at the end of the current request. To know if the current session is temporary, we can use method `isTemporary()`:

```
Session.get().isTemporary();
```

If a session is not temporary (i.e. it is *permanent*), it's identified by an unique id which can be read calling method `getId()`. This value will be `null` if session is temporary.

Although Wicket is able to automatically recognize when it needs to replace a temporary session with a permanent one, sometimes we may need to manually control this process to make our initially temporary session permanent.

To illustrate this possible scenario let's consider project *BindSessionExample* where we have a stateless home page which sets a session attribute inside its constructor and then it redirects user to another page which displays with a label the session attribute previously created. The code of the two pages is the following:

Home page:

```
public class HomePage extends WebPage {
    public HomePage(final PageParameters parameters) {
        Session.get().setAttribute("username", "tommy");
        Session.get().bind();

        setResponsePage(DisplaySessionParameter.class);
    }
}
```

Target page:

```
public class DisplaySessionParameter extends WebPage {

        public DisplaySessionParameter() {
           super();
           add(new Label("username", (String) Session.get().getAttribute("username")));
        }
}
```

Again, we kept page logic very simple to not over-bloat the example with unnecessary code. In the snippet above we have also bolded `Session`'s method `bind()` which converts temporary session into a

permanent one. If home page hadn't invoked this method, the session with its attribute would have been discarded at the end of the request and page `DisplaySessionParameter` would have displayed an empty value in its label.

7.4.5 Discarding session data

Once a user has finished doing her work inside our application, she must be able to log out and clean her session data. To be sure that a permanent session will be discarded at the end of the current request, class `Session` provides method `invalidate()`. If we want to immediately invalidate a given session without waiting for the current request to complete, we can use method `invalidateNow()`.

Warning

Remember that `invalidateNow()` will immediately remove also any instance of components (and pages) from session, meaning that once we have called this method we won't be able to work with them for the rest of request processing.

7.5 Storing arbitrary objects with metadata

JavaServer Pages Specification[19] defines 4 scopes in which a page can create and access a variable. These scopes are:

- ***request***: variables declared in this scope can be seen only by pages processing the same request. The lifespan of these variables is (at most) equal to the one of the related request. They are discarded when the full response has been generated or when the request is forwarded somewhere else.
- ***page***: variables declared this scope can be seen only by the page that has created them.
- ***session***: variables in session scope can be created and accessed by every page used in the same session where they are defined.
- ***application***: this is the widest scope. Variables defined in this scope can be used by any page of a given web application.

Although Wicket doesn't implement the JSP Specification (it is rather an alternative to it), it offers a feature called *metadata* which resembles scoped variables but is much more powerful. Metadata is quite similar to a Java `Map` in that it stores pairs of key-value objects where the key must be unique. In Wicket each of the following classes has its own metadata store: `RequestCycle`, `Session`, `Application` and `Component`.

The key used for metadata is an instance of class `org.apache.wicket.MetaDataKey<T>`. To put an arbitrary object into metadata we must use method `setMetaData` which takes two parameters in input: the key to use to store data and the data object.

If we are using metadata with classes `Session` or `Component`, data object must be serializable because Wicket serializes both session and component instances. This constraint is not applied to metadata of classes `Application` and `RequestCycle` which can contain a generic object. In any case, the type of data object must be compatible with the type parameter `T` specified by the key.

To retrieve a previously inserted object we must use method `getMetaData(MetaDataKey<T> key)`. In the following example we set a `java.sql.Connection` object into application's metadata so it can be used by any page of the application:

Application class code:

```
public static MetaDataApp extends WebApplication{
        //Do some stuff...
```

19 *Paragraph 1.8.2 'Objects and Scopes' of JavaServer Pages 2.1 Specification*

```
        /**
        * Metadata key definition
        */
        public static MetaDataKey<Conncetion> connectionKey = new
        MetaDataKey<Conncetion>(){};

        /**
        * Application's initialization
        */
        @Override
        public void init(){
                super.init();
                Connection connection;
                //connection initialization...
                setMetaData(connectionKey, connection);
                //Do some other stuff..
        }
}
```

Code to get the object from metadata:

```
Connection connection = Application.get().getMetaData(MetaDataApp.connectionKey);
```

Since class `MetaDataKey<T>` is declared as `abstract`, we must implement it with a subclass or with an anonymous class (like we did in the example above).

7.6 Summary

In this chapter we had a look at how Wicket internally handles a web request. Even if most of the time we won't need to customize this internal process, knowing how it works is essential to use the framework at 100%.

Entities like `Application` and `Session` will come in handy again when we will tackle the topic of security in chapter 19.

8 Wicket Links and URL generation

Up to now we used component `Link` to move from a page to another and we have seen that it is quiet similar to a “click” event handler (see paragraph 2.4).

However this component alone is not enough to build all possible kinds of links we may need in our pages. Therefore, Wicket offers other link components suited for those tasks which can not be accomplished with `Link`.

Besides learning new link components, in this chapter we will also see how to customize the page URL generated by Wicket using the *encoding* facility provided by the framework and the *page parameters* that can be passed to a target page.

8.1 PageParameters

A common practice in web development is to pass data to a page using *query string parameters* (like `?paramName1=paramValu1&paramName2=paramValue2`...). Wicket offers a more flexible and object oriented way to do this with *models* (we will see them in the next chapter). However, even if we are using Wicket, we still need to use query string parameters to exchange data with other Internet-based services. Consider for example a classic confirmation page which is linked inside an email to let users confirm important actions like password changing or the subscription to a mailing list. This kind of page usually expects to receive a query string parameter containing the id of the action to confirm.

Query string parameters can also be referred to as *named parameters*. In Wicket they are handled with class `org.apache.wicket.request.mapper.parameter.PageParameters`. Since named parameters are basically name-value pairs, `PageParameters` works in much the same way as Java `Map` providing two methods to create/modify a parameter (`add(String name, Object value)` and `set(String name, Object value)`), one method to remove an existing parameter (`remove(String name)`) and one to retrieve the value of a given parameter (`get(String name)`). Here is a snippet to illustrate the usage of `PageParameters`:

```
PageParameters pageParameters = new PageParameters();
//add a couple of parameters
pageParameters.add("name", "John");
pageParameters.add("age", 28);
//retrieve the value of 'age' parameter
pageParameters.get("age");
```

Now that we have seen how to work with page parameters, let's see how to use them with our pages.

8.1.1 PageParameters and bookmarkable pages

Base class `Page` comes with a constructor which takes in input a `PageParameters` instance. If we use this superclass constructor in our page, `PageParameters` will be used to build page URL and it can be retrieved at a later time with `Page`'s method `getPageParameters()`.

In the following example taken from project *PageParametersExample* we have a home page with a link to a second page that uses a version of method `setResponsePage` that takes in input also a `PageParameters` to build the target page (named `PageWithParameters`). The code for the link and for the target page is the following:

Link code:

```
add(new Link("pageWithIndexParam") {

        @Override
        public void onClick() {
                PageParameters pageParameters = new PageParameters();
                pageParameters.add("foo", "foo");
                pageParameters.add("bor", "bar");

                setResponsePage(PageWithParameters.class, pageParameters);
        }

});
```

Target page code:

```
 public class PageWithParameters extends WebPage {
        //Override superclass constructor
        public PageWithParameters(PageParameters parameters) {
                super(parameters);
        }
 }
```

The code is quite straightforward and it's more interesting to look at the URL generated for the target page:

```
<app root>/PageParametersExample/wicket/bookmarkable/org.wicketTutorial.PageWithParameters
                                                                        ?foo=foo&bor=bar
```

At first glance the URL above could seem a little weird, except for the last part which contains the two named parameters used to build the target page.

The reason for this "strange" URL is that, as we know from paragraph 6.2.5, when a page is instantiated using a constructor with no argument or using a constructor that accepts only a `PageParameters`, Wicket will try to generate a static URL for it, with no session-relative informations. This kind of URL is called *bookmarkable* because it can be saved by users as a bookmark and accessed at a later time.

A bookmarkable URL is composed by a fixed prefix (which by default is `bookmarkable`) and the qualified name of page class (`org.wicketTutorial.PageWithParameters` in our example). Segment `wicket` is another fixed prefix added by default during URL generation. In paragraph 8.6.4 we will see how to customize fixed prefixes with a custom implementation of interface `IMapperContext`.

8.1.2 Indexed parameters

Besides named parameters, Wicket supports also *indexed parameters*. These kinds of parameters are rendered as URL segments placed before named parameters. Let's consider for example the following URL:

```
<application path>/foo/bar?1&baz=baz
```

The URL above contains two indexed parameters (`foo` and `bar`) and a query string consisting of the page id and a named parameter (`baz`). Just like named parameters also indexed parameters are

handled with class `PageParameters`. The methods provided by `PageParameters` for indexed parameters are `set(int index, Object object)` (to add/modify a parameter), `remove(int index)` (to remove a parameter) and `get(int index)` (to read a parameter).

As their name suggests, indexed parameters are identified by a numeric index and they are rendered following the order in which they have been added to `PageParameters`. The following is an example of usage of indexed parameters:

```
PageParameters pageParameters = new PageParameters();
//add a couple of parameters
pageParameters.set(0, "foo");
pageParameters.set(1, "bar");
//retrieve the value of the second parameter ("bar")
pageParameters.get(1);
```

Project *PageParametersExample* comes also with a link to a page with both indexed parameters and a named parameter:

```
        add(new Link("pageWithNamedIndexParam") {

                @Override
                public void onClick() {
                        PageParameters pageParameters = new PageParameters();
                        pageParameters.set(0, "foo");
                        pageParameters.set(1, "bar");
                        pageParameters.add("baz", "baz");

                        setResponsePage(PageWithParameters.class, pageParameters);
                }

        });
```

The URL generated for the linked page (`PageWithParameters`) is the one seen at the beginning of the paragraph.

8.2 Bookmarkable links

A link to a bookmarkable page can be built with link component `org.apache.wicket.markup.html.link.BookmarkablePageLink`:

```
BookmarkablePageLink bpl=new BookmarkablePageLink(PageWithParameters.class, pageParameters);
```

The specific purpose of this component is to move user to a bookmarkable page, hence we don't have to implement any abstract method like we do with `Link` component.

8.3 Automatically creating bookmarkable links with tag <wicket:link>

Bookmarkable pages can be linked directly inside markup files without writing any Java code. Using tag `<wicket:link>` we ask Wicket to automatically add bookmarkable links for the anchors wrapped inside this tag. Here is an example of usage of tag `<wicket:link>` taken from the home page of project *BookmarkablePageAutoLink*:

```
<!DOCTYPE html>
<html xmlns:wicket="http://wicket.apache.org">
        <head>
                <meta charset="utf-8" />
                <title>Apache Wicket Quickstart</title>
        </head>
        <body>
           <div id="bd">
              <wicket:link>
                      <a href="HomePage.html">HomePage</a><br/>
                      <a href="anotherPackage/SubPackagePage.html">SubPackagePage</a>
              </wicket:link>
           </div>
        </body>
</html>
```

The key part of the markup above is the `href` attribute which must contain the package-relative path to a page. The home page is inside package `org.wicketTutorial` which in turns contains the sub package `anotherPackage`. This package hierarchy is reflected into `href` attributes: in the first anchor we have a link to the home page itself while the second anchor points to page `SubPackagePage` which is placed into sub package `anotherPackage`. Absolute paths are supported as well and we can use them if we want to specify the full package of a given page. For example the link to `SubPackagePage` could have been written also in the following (more verbose) way :

```
<a href="/org/wicketTutorial/anotherPackage/SubPackagePage.html">SubPackagePage</a>
```

If we take a look also at the markup of `SubPackagePage` we can see that it contains a link to the home page which uses the parent directory selector (two dots):

```
<!DOCTYPE html>
<html xmlns:wicket="http://wicket.apache.org">
        <head>
                <meta charset="utf-8" />
                <title>Apache Wicket Quickstart</title>
        </head>
        <body>
                <div id="bd">
                        <wicket:link>
                                <a href="../HomePage.html">HomePage</a><br/>
                                <a href="SubPackagePage.html">SubPackagePage</a>
                        </wicket:link>
                </div>
        </body>
</html>
```

Please note that any link to the current page (aka *self link*) is disabled. For example in the home page the self link is rendered like this:

```
<span><em>HomePage</em></span
```

The markup used to render disabled links can be customized using the markup settings (interface `IMarkupSettings`) available inside application class:

```
@Override
public void init()
{
        super.init();
        //wrap disabled links with <b> tag
        getMarkupSettings().setDefaultBeforeDisabledLink("<b>");
        getMarkupSettings().setDefaultAfterDisabledLink("</b>");
}
```

The purpose of tag `<wicket:link>` is not limited to just ease the usage of bookmarkable pages. As we we will see in chapter 13, this tag can be also adopted to manage web resources like pictures, CSS files, JavaScript files and so on.

8.4 External links

Since Wicket uses plain HTML markup files as templates, we can place an anchor to an external page directly into markup file. But when we need to dynamically generate external anchors, we can use link component `org.apache.wicket.markup.html.link.ExternalLink`. In order to build an external link we must specify the value of attribute `href` using a model or a plain string. In the next snippet, given an instance of `Person`, we generate a Google search query for its full name:

Html:

```
<a wicket:id="externalSite">Search me on Google!</a>
```

Java code:

```
Person person = new Person("John", "Smith");
String fullName = person.getFullName();
//Space characters must be replaced by character '+'
String googleQuery = "http://www.google.com/search?q=" + fullName.replace(" ", "+");
add(new ExternalLink("externalSite", googleQuery));
```

Generated anchor:

```
<a href="http://www.google.com/search?q=John+Smith">Search me on Google!</a>
```

If we need to specify also a dynamic value for the text inside the anchor, we can pass it as additional constructor parameter:

Html:

```
<a wicket:id="externalSite">Label goes here...</a>
```

Java code:

```
Person person = new Person("John", "Smith");
String fullName = person.getFullName();
String googleQuery = "http://www.google.com/search?q=" + fullName.replace(" ", "+");
String linkLabel = "Search '" + fullName + "' on Google.";
add(new ExternalLink("externalSite", googleQuery, linkLabel));
```

Generated anchor:

```
<a href="http://www.google.com/search?q=John+Smith">Search 'John Smith' on Google.</a>
```

8.5 Stateless links

Component `Link` has a stateful nature, hence it cannot be used with stateless pages. To use links with these kinds of pages Wicket provides the convenience component `org.apache.wicket.markup.html.link.StatelessLink` which is basically a subtype of `Link` with the stateless hint set to true.

Please keep in mind that Wicket generates a new instance of a stateless page also to serve stateless links, so the code inside method `onClick()` can not depend on instance variables. To illustrate this potential issue let's consider the following code (from project *StatelessPage*) where the value of the variable `index` is used inside `onclick()`:

```
public class StatelessPage extends WebPage {
	private int index = 0;

	public StatelessPage(PageParameters parameters) {
		super(parameters);
	}

	@Override
	protected void onInitialize() {
		super.onInitialize();
		setStatelessHint(true);

		add(new StatelessLink("statelessLink") {

			@Override
			public void onClick() {
				//It will always print zero
				System.out.println(index++);
			}

		});
	}
}
```

The printed value will always be zero because a new instance of the page is used every time user clicks on link `statelessLink`.

8.6 Generating structured and clear URLs

Having structured URLs in our site is a basic requirement if we want to build an efficient SEO[20] strategy, but it also contributes to improve user experience with more intuitive URLs. Wicket provides two different ways to control URL generation. The first (and simplest) is to “mount” one or more pages to an arbitrary path, while a more powerful technique is to use custom implementations of interfaces `IMapperContext` and `IPageParametersEncoder`. In the next paragraphs we will learn both these two techniques.

8.6.1 Mounting a single page

20 *http://en.wikipedia.org/wiki/Search_engine_optimization*

With Wicket we can mount a page to a given path in much the same way as we map a servlet filer to a desired path inside file *web.xml* (see page 9). Using method `mountPage(String path, Class<T> pageClass)` of class `WepApplication` we tell Wicket to respond with a new instance of `pageClass` whenever a user navigates to the given `path`. In the application class of project *MountedPagesExample* we mount page `MountedPage` to path `"/pageMount"`:

```
@Override
public void init()
{
        super.init();
        mountPage("/pageMount", MountedPage.class);
        //Other initialization code...
}
```

The path provided to `mountPage` will be also used to generate the URL for any page of the specified class:

```
//it will return "/pageMount"
RequestCycle.get().urlFor(MountedPage.class);
```

Under the hood `mountPage` mounts an instance of request mapper `org.apache.wicket.request.mapper.MountedMapper` configured for the given path:

```
public final <T extends Page> void mountPage(final String path,final Class<T> pageClass)
{
        mount(new MountedMapper(path, pageClass));
}
```

Request mappers and `Application`'s method `mount` have been introduced in the previous chapter (paragraph 7.3.1).

8.6.2 Using parameter placeholders with mounted pages

The path specified for mounted pages can contain dynamic segments which are populated with the values of the named parameters used to build the page. These segments are declared using special segments called *parameter placeholders*. Consider the path used in the following example:

```
        mountPage("/pageMount/${foo}/otherSegm", MountedPageWithPlaceholder.class);
```

The path used above is composed by three segments: the first and the last are fixed while the second will be replaced by the value of named parameter `foo` that must be provided when page `MountedPageWithPlaceholder` is instantiated:

Java code:

```
PageParameters pageParameters = new PageParameters();
pageParameters.add("foo", "foo");

setResponsePage(MountedPageWithPlaceholder.class, pageParameters);
```

Generated URL:

```
<Application path>/pageMount/foo/otherSegm
```

On the contrary if we manually insert an URL like '`<web app path>/pageMount/bar/otherSegm`', we can read value '`bar`' retrieving the named parameter `foo` inside our page.
Place holders can be declared as optional using character '#' in place of '$':

```
mountPage("/pageMount/#{foo}/otherSegm", MountedPageOptionalPlaceholder.class);
```

If the named parameter for an optional placeholder is missing, the corresponding segment is removed from the final URL:

Java code:

```
PageParameters pageParameters = new PageParameters();
setResponsePage(MountedPageWithPlaceholder.class, pageParameters);
```

Generated URL:

```
<Application path>/pageMount/otherSegm
```

8.6.3 Mounting a package

In addition to mounting a single page, Wicket offers also the possibility of mounting all the pages inside a package to a given path. Method `mountPackage(String path, Class<T> pageClass)` of class `WepApplication` will mount every page inside `pageClass`'s package to the specified `path`.
The resulting URL for package-mounted pages will have the following structure:

```
<Application path>/mountedPath/<PageClassName>[optional query string]
```

For example in project *MountedPagesExample* we have mounted all pages inside package `org.tutorialWicket.subPackage` with this instruction:

```
mountPackage("/mountPackage", StatefulPackageMount.class);
```

`StatefulPackageMount` is one of the pages placed into the desired package and its URL will be:

```
<Application path>/mountPackage/StatefulPackageMount?1
```

Similarly to what is done by `mountPage`, the implementation of method `mountPackage` mounts an instance of `org.apache.wicket.request.mapper.PackageMapper` to the given path.

8.6.4 Providing custom *mapper context* to request mappers

Interface `org.apache.wicket.request.mapper.IMapperContext` is used by request mappers to create new page instances and to retrieve static URL segments used to build and parse page URLs. Here is the list of these segments:

- *Namespace*: it's the first URL segment of not-mounted pages. By default its value is `wicket`.
- *Identifier for non bookmarkable URLs*: it's the segment that identifies non bookmarkable pages. By default its value is `page`.

- *Identifier for bookmarkable URLs*: it's the segment that identifies bookmarkable pages. By default its value is `bookmarkable` (as we have seen before in paragraph 8.1.1).
- *Identifier for resources*: it's the segment that identifies Wicket resources. Its default value is `resources`. The topic of resource management will be covered in chapter 13.

`IMapperContext` provides a getter method for any segment listed above. By default Wicket uses class `org.apache.wicket.DefaultMapperContext` as mapper context.

Project *CustomMapperContext* is an example of customization of mapper context where we use `index` as identifier for non-bookmarkable pages and `staticURL` as identifier for bookmarkable pages. In this project, instead of implementing our mapper context from scratch, we used `DefaultMapperContext` as base class overriding just the two methods we need to achieve the desired result (`getBookmarkableIdentifier()` and `getPageIdentifier()`). The final implementation is the following:

```
public class CustomMapperContext extends DefaultMapperContext{
	@Override
	public String getBookmarkableIdentifier() {
		return "staticURL";
	}
	@Override
	public String getPageIdentifier() {
		return "index";
	}
}
```

Now to use a custom mapper context in our application we must override `Application`'s method `newMapperContext()` and make it return our implementation of `IMapperContext`:

```
@Override
protected IMapperContext newMapperContext() {
	return new CustomMapperContext();
}
```

8.6.5 Controlling how page parameters are encoded with IPageParametersEncoder

Some request mappers (like `MountedMapper` and `PackageMapper`) can delegate page parameters encoding/decoding to interface `org.apache.wicket.request.mapper.parameter.IPageParametersEncoder`.

This entity exposes two methods: `encodePageParameters` and `decodePageParameters`: the first one is invoked to encode page parameters into an URL while the second one extracts parameters from URL.

Wicket comes with a built-in implementation of this interface which encodes named page parameters as URL segments using the following patter: `/paramName1/paramValue1/paramName2/paramValue2...`

This built-in encoder is class `org.apache.wicket.request.mapper.parameter.`*`UrlPathPageParametersEncoder`*. In project *PageParametersEncoderExample* we have manually mounted a `MountedMapper` that takes in input also an `UrlPathPageParametersEncoder`:

```
@Override
public void init()
{
```

```
        super.init();
        mount(new MountedMapper("/mountedPath", MountedPage.class, new
                  UrlPathPageParametersEncoder()));
}
```

The home page of the project contains just a link to page `MountedPage`. The code of the link and the resulting page URL are the following:

Link code:

```
add(new Link("mountedPage") {

        @Override
        public void onClick() {
                PageParameters pageParameters = new PageParameters();
                pageParameters.add("foo", "foo");
                pageParameters.add("bar", "bar");

                setResponsePage(MountedPage.class, pageParameters);
        }
});
```

Generated URL:

```
<Application path>/mountedPath/foo/foo/bar/bar?1
```

8.6.6 Encrypting page URLs

Sometimes URLs are a double–edged sword for our site because they can expose too many details about the internal structure of our web application and malicious users could exploit them to perform a cross-site request forgery[21].

To avoid this kind of security threat we can use request mapper `CryptoMapper` which wraps an existing mapper and encrypts the original URL producing a single encrypted segment:

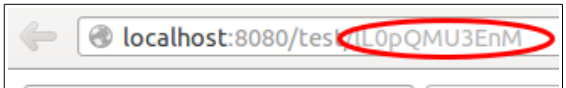


Illustration 8.1: An URL with the encrypted segment

Typically, `CryptoMapper` is registered into a Wicket application as the root request mapper wrapping the default one:

```
@Override
public void init()
{
 super.init();
 setRootRequestMapper(new CryptoMapper(getRootRequestMapper(), this));
 //pages and resources must be mounted after we have set CryptoMapper
 mountPage("/foo/", HomePage.class);
}
```

21 http://en.wikipedia.org/wiki/Cross-site_request_forgery

As pointed out in the code above, pages and resources must be mounted after having set `CryptoMapper` as root mapper, otherwise the mounted paths will not work.

8.7 Summary

Links and URLs are not trivial topics as they may seem and in Wicket they are strictly interconnected. Developers must choose the right trade-off between producing structured URLs and avoiding to make them verbose and vulnerable.

In this chapter we have explored the tools provided by Wicket to control how URLs are generated. We have started with static URLs for bookmarkable pages and we have seen how to pass parameters to target pages with class `PageParameters`. In the second part of the chapter we have focused on mounting pages to a specific path and on controlling how parameters are encoded by Wicket. Finally, we have also seen how to encrypt URLs to prevent security vulnerabilities.

9 Wicket models and forms

In Wicket the concept of "model"[22] is probably the most important topic of the entire framework and it is strictly related to the usage of its components. In addition, models are also an important element for internationalization, as we will see in paragraph 12.6.

However, despite their fundamental role in Wicket, models are not difficult to understand but the best way to learn how they work is to use them with Wicket forms.

That's why we haven't talked about models so far, and why this chapter discusses these two topics together.

9.1 What is a *model*?

Model is essentially a facade[23] interface which allows components to access and modify their data without knowing any detail about how they are managed or persisted. Every component has at most one related model, while a model can be shared among different components. In Wicket a model is any implementation of the interface `org.apache.wicket.model.IModel`:

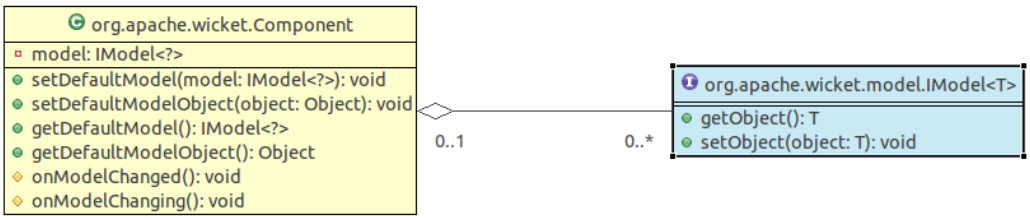


Illustration 9.1: UML class diagram of Component and IModel

The `IModel` interface defines just the methods needed to get and set a data object (`getObject()` and `setObject()`), decoupling components from concrete details about the persistence strategy adopted for data. In addition, the level of indirection introduced by models allows to access data object only when it is really needed (for example during rendering phase) and not earlier when it may not be ready to be used.

Any component can get/set its model as well as its data object using the 4 public shortcut methods listed in the class diagram above.

The two methods `onModelChanged()` and `onModelChanging()` are triggered by Wicket each time a model is modified: the first one is called after model has been changed, the second one just before the change occurs.

In the examples seen so far we have worked with `Label` component using its constructor which takes in input two string parameters, the component id and the text to display:

```
add(new Label("helloMessage", "Hello WicketWorld!"));
```

This constructor internally builds a model which wraps the second string parameter. That's why we didn't noticed label model in the previous examples. Here is the code of this constructor:

```
public Label(final String id, String label)
```

22 *Wicket models have nothing to do with the model we talked about in paragraph 1.2!!*
23 *For an introduction to Facade pattern see* http://en.wikipedia.org/wiki/Facade_pattern

```
{
        this(id, new Model<String>(label));
}
```

Class `org.apache.wicket.model.Model` is a basic implementation of `IModel`. It can wrap any object that implements the interface `java.io.Serializable`. The reason of this constraint over data object is that this model is stored into web session, and we know from chapter 6 that data are stored into session using serialization.

Note

In general, Wicket models support a *detaching* capability that allows to work also with non-serializable objects as data model. We will see detaching mechanism later in this chapter.

Just like any other Wicket components, `Label` provides also a constructor that takes in input the component id and the model to use with the component. Using this constructor the previous example becomes:

```
add(new Label("helloMessage", new Model<String>("Hello WicketWorld!")));
```

Note

Class `Model` comes with a bunch of factory methods that makes easier building new model instances. For example method `of(T object)` creates a new instance of `Model` which wraps `object` parameter inside it.

So instead of writing

```
new Model<String>("Hello WicketWorld!")
```

we can write

```
Model.of("Hello WicketWorld!")
```

If the data object is a `List`, a `Map` or a `Set` we can use similar methods called `ofList`, `ofMap` and `ofSet`.
From now on we will use this factory methods in our examples.

It's quite clear that if our `Label` must display a static text it doesn't make much sense building a model by hand like we did in the last code example.

However is not unusual to have a `Label` that must display a dynamic value, like the input inserted by a user or a value read from database. Wicket models are designed to solve these kind of problems.

Let's say we need a label to display the current time stamp each time a page is rendered. We can implement a custom model which returns a new `Date` instance when method `getObject()` is called:

```
IModel timeStampModel = new Model<String>(){
        @Override
        public String getObject() {
                return new Date().toString();
        }
};
```

```
add(new Label("timeStamp", timeStampModel));
```

Even if sometimes writing a custom model could be a good choice to solve a specific problem, Wicket already provides a set of `IModel` implementations which should fit most of our needs. In the next paragraph we will see a couple of models that allow us to easily integrate JavaBeans with our web applications and in particular with our forms.

Note

By default class `Component` escapes HTML sensitive characters (like '<', '>' or '&') from the textual representation of its model object. The term 'escape' means that these characters will be replaced with their corresponding HTML entity[24] (for example '<' becomes '< ').

This is done for security reasons as a malicious user could attempt to inject markup or JavaScript into our pages.

If we want to display the raw content stored inside model, we can tell `Component` to not escape characters calling method `setEscapeModelStrings(false)`.

9.2 Models and JavaBeans

One of the main goals of Wicket is to use JavaBeans and POJO as data model, overcoming the impedance mismatch between web technologies and OO paradigm. In order to make this task as easy as possible, Wicket offers two special model classes: `org.apache.wicket.model.PropertyModel` and `org.apache.wicket.model.CompoundPropertyModel`. We will see how to use them in the next two examples, using the following JavaBean as data object:

```
public class Person implements Serializable{
       private String name;
       private String surname;
       private String address;
       private String email;
       private String passportCode;

       private Person spouse;
       private List<Person> children;

       public Person(String name, String surname) {
              this.name = name;
              this.surname = surname;
       }

       public String getFullName(){
          return name + " " + surname;
       }

       /*
         Getters and setters for private fields
         ...
        */
}
```

24 http://en.wikipedia.org/wiki/Character_entity_reference

9.2.1 PropertyModel

Let's say we want to display the `name` field of a `Person` instance with a label. We could, of course, use `Model` class like we did in the previous example, obtaining something like this:

```
Person person = new Person();
//load person's data...

Label label = new Label("name", new Model(person.getName()));
```

However this solution has a huge drawback: the text displayed by the label will be static and if we change the value of the field, the label won't update its content. Instead, to display always the current value of a class field, we should use model class `org.apache.wicket.model.PropertyModel`:

```
Person person = new Person();
//load person's data...

Label label = new Label("name", new PropertyModel(person, "name"));
```

`PropertyModel` has just one constructor with two parameters: the model object (`person` in our example) and the name of the property we want to read/write (`"name"` in our example). This last parameter is called *property expression*.

Internally, methods *getObject*/*setObject* use property expression to get/set property's value. To resolve class properties `PropertyModel` uses class `org.apache.wicket.util.lang.PropertyResolver` which can access any kind of property, private fields included.

Just like Java language, property expressions support dotted notation to select sub properties. So if we want to display the name of `Person`'s spouse we can write:

```
Label label = new Label("spouseName", new PropertyModel(person, "spouse.name"));
```

Note

`PropertyModel` is null-safe, which means we don't have to worry if property expression includes a null value in its path. If such a value is encountered, an empty string will be returned.

If property is an array or a `List`, we can specify an index after its name. For example, to display the name of the first child of a `Person` we can write the following property expression:

```
Label label = new Label("firstChildName", new PropertyModel(person, "children.0.name"));
```

Indexes and map keyes can be also specified using squared brackets like “children[0].name” or “mapField[key].subfield”.

9.2.2 CompoundPropertyModel and model inheritance

Class `org.apache.wicket.model.CompoundPropertyModel` is a particular kind of model which is usually used along with another feature of Wicket called *model inheritance*.

With this feature, when a component needs to use a model but no one has been assigned to it, it will search through the whole container hierarchy for a parent with an *inheritable model*. Inheritable models are those which implement interface `org.apache.wicket.model.IComponentInheritedModel`

and `CompoundPropertyModel` is one of them.

Once a `CompoundPropertyModel` has been inherited by a component, it will behave just like a `PropertyModel` using the id of the component as property expression. As a consequence, to make the most of `CompoundPropertyModel` we must assign it to one of the container of a given component, rather than directly to the component itself.

For example if we use `CompoundPropertyModel` with the previous example (display spouse's name), the code would become like this:

```
//set CompoundPropertyModel as model for the container of the label
setDefaultModel(new CompoundPropertyModel(person));

Label label = new Label("spouse.name");

add(label);
```

Note that now the id of the label is equal to the property expression previously used with `PropertyModel`. Now as further example let's say we want to extend the code above displaying also all the main informations of a person (name, surname, address and email). All we have to do is to add one label for every additional information using the relative property expression as component id:

```
//Create a person named 'John Smith'
Person person = new Person("John", "Smith");
setDefaultModel(new CompoundPropertyModel(person));

add(new Label("name"));
add(new Label("surname"));
add(new Label("address"));
add(new Label("email"));
add(new Label("spouse.name"));
```

`CompoundPropertyModel` can save us a lot of boring coding if we choose the id of components according to properties name. However it's also possible to use this type of model even if the id of a component does not correspond to a valid property expression. Method `bind(String property)` allows to create a property model from a given `CompoundPropertyModel` using the provided parameter as property expression. For example if we want to display spouse's name in a label having *"xyz"* as id, we can write the following code:

```
//Create a person named 'John Smith'
Person person = new Person("John", "Smith");
CompoundPropertyModel compoundModel;
setDefaultModel(compoundModel = new CompoundPropertyModel(person));

add(new Label("xyz", compoundModel.bind("spouse.name")));
```

`CompoundPropertyModel` are particularly useful when used in combination with Wicket forms, as we will see in the next paragraph.

Note

`Model` is referred to as *static* model because the result of its method `getObject` is fixed an it is not dynamically evaluated each time the method is

called. In contrast, models like `PropertyModel` and `CompoundPropertyModel` are called *dynamic*.

9.3 Wicket forms

Web applications use HTML forms to collect user input and send it to the server. Wicket provides class `org.apache.wicket.markup.html.form.Form` to handle web forms. This component must be bound to tag `<form>`. The following snippet shows how to create a very basic Wicket form in a page:

Html:

```
<form wicket:id="form">
    <input type="submit" value="submit"/>
</form>
```

Java code:

```
Form form = new Form("form"){
    @Override
    protected void onSubmit() {
        System.out.println("Form submitted.");
    }
};
add(form);
```

Method `onSubmit` is called whenever a form has been submitted and it can be overridden to perform custom actions. Please note that a Wicket form can be submitted using a standard HTML submit button which is not mapped to any component (i.e. it hasn't a `wicket:id` attribute).

In the next chapter we will continue to explore Wicket forms and we will see how to submit forms using special components which implement interface `org.apache.wicket.markup.html.form.IFormSubmitter`.

9.3.1 Form and models

A form should contain some input fields (like text fields, check boxes, radio buttons, drop-dpwn lists, text areas, etc...) to interact with users. Wicket provides an abstraction for all these kinds of elements with component `org.apache.wicket.markup.html.form.FormComponent`:

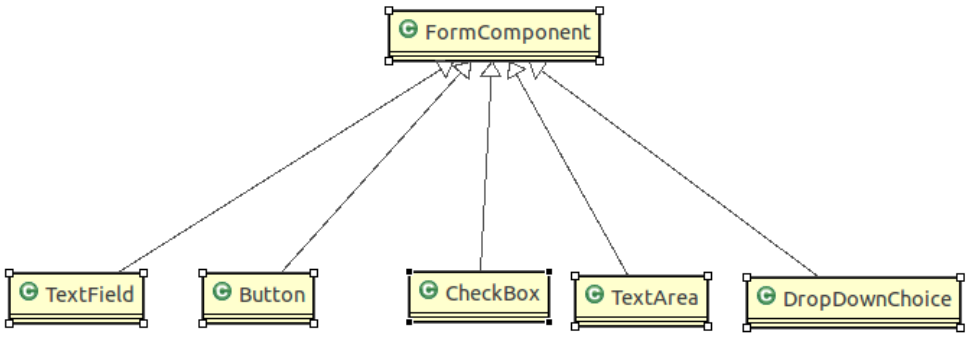


Illustration 9.2: FormComponent and some of the most common input components. All classes are under package org.apache.wicket.markup.html.form.

The purpose of `FormComponent` is to store the corresponding user input into its model when form is

submitted. Form is responsible for mapping input values to the corresponding component, avoiding us the burden of manually synchronizing models with input fields and vice versa.

9.3.2 Login form

As first example of interaction between form and models, we will build a classic login form which asks for username and password (project *LoginForm*).

Warning

The topic of security will be discussed later in chapter 19. The following form is for example purposes only and is not suited for a real application.
If you need to use a login form you should consider to use component `org.apache.wicket.authroles.authentication.panel.SignInPanel` shipped with Wicket.

This form needs two text fields, one of which must be a password field. We should also use a label to display the result of login process[25]. For the sake of simplicity, the login logic is all inside `onSubmit` and is quite trivial.
The following is a possible implementation of our form:

```
public class LoginForm extends Form{
	private TextField usernameField;
	private PasswordTextField passwordField;
	private Label loginStatus;

	public LoginForm(String id) {
		super(id);

		usernameField = new TextField("username", Model.of(""));
		passwordField = new PasswordTextField("password", Model.of(""));
		loginStatus = new Label("loginStatus", Model.of(""));

		add(usernameField);
		add(passwordField);
		add(loginStatus);
	}

	public final void onSubmit() {
		String username = (String)usernameField.getDefaultModelObject();
		String password = (String)passwordField.getDefaultModelObject();

		if(username.equals("test") && password.equals("test"))
			loginStatus.setDefaultModelObject("Congratulations!");
		else
			loginStatus.setDefaultModelObject("Wrong username or password!");

	}
}
```

Inside form constructor we build the three components used in the form and we assign them a model containing an empty string:

25 *In chapter 10 we will see that Wicket offers a built-in mechanism to display feedback messages to user.*

```
                usernameField = new TextField("username", Model.of(""));
                passwordField = new PasswordTextField("password", Model.of(""));
                loginStatus = new Label("loginStatus", Model.of(""));
```

If we don't provide a model to a form component, we will get the following exception on form submission:

```
java.lang.IllegalStateException: Attempt to set model object on null model of component:
```

Component `TextField` corresponds to the standard text field, without any particular behavior or restriction on the allowed values. We must bind this component to tag `<input>` with attribute `type` equals to *"text"*.

`PasswordTextField` is a subtype of `TextFiled` and it must be used with an `<input>` tag with attribute `type` equals to *"password"*. For security reasons component `PasswordTextField` cleans its value at each request, so it wil be always empty after form has been rendered.

By default `PasswordTextField` fields are *required,* meaning that if we left them empty, the form won't be submitted (i.e. `onSubmit` won't be called).

Class `FormComponent` provides method `setRequired(boolean required)` to change this behavior. [26]

Inside `onSubmit`, to get/set model objects we have used shortcut methods `setDefaultModelObject` and `getDefaultModelObject`. Both methods are defined in class `Component` (see class diagram from Illustration 9.1). The following are the possible markup and code for the login page:

Html:

```
<html>
<head>
  <title>Login page</title>
</head>
<body>
<form id="loginForm" method="get" wicket:id="loginForm">
  <fieldset>
    <legend style="color: #F90">Login</legend>
    <p wicket:id="loginStatus"></p>
    <span >Username: </span><input wicket:id="username" type="text" id="username" /><br/>
    <span >Password: </span><input wicket:id="password" type="password" id="password" />
    <p>
    <input type="submit" name="Login" value="Login"/>
    </p>

  </fieldset>
</form>
</body>
</html>
```

Java code:

```
public class HomePage extends WebPage {
    public HomePage(final PageParameters parameters) {
        super(parameters);
```

26 *Chapter 10 will cover form validation in detail.*

```
        add(new LoginForm("loginForm"));
    }
}
```

The example shows how Wicket form components can be used to store user input inside their model. However we can dramatically improve the form code using `CompoundPropertyModel` and its ability to access the properties of its model object. The revisited code is the following (project *LoginForm Revisited*):

```
public class LoginForm extends Form{
        private String username;
        private String password;
        private String loginStatus;

        public LoginForm(String id) {
                super(id);
                setDefaultModel(new CompoundPropertyModel(this));

                add(new TextField("username"));
                add(new PasswordTextField("password"));
                add(new Label("loginStatus"));
        }

        public final void onSubmit() {
                if(username.equals("test") && password.equals("test"))
                        loginStatus = "Congratulations!";
                else
                        loginStatus = "Wrong username or password !";
        }
}
```

In this version the form itself is used as model object for `CompoundPropertyModel`. This allows children components to use directly form's fields as backing objects, without explicitly creating a model for them.

Note

Keep in mind that when `CompoundPropertyModel` is inherited, it does not consider the ids of traversed containers for the final property expression, but it will always use just the id of the inheritor component.
To understand this potential pitfall, let's consider the following initialization code of a page:

```
//Create a person named 'John Smith'
Person person = new Person("John", "Smith");
//Create a person named 'Jill Smith'
Person spouse = new Person("Jill", "Smith");
//Set Jill as John's spouse
person.setSpouse(spouse);

setDefaultModel(new CompoundPropertyModel(person));
```

```
WebMarkupContainer spouse = new WebMarkupContainer("spouse");
Label name;
spouse.add(name = new Label("name"));

add(spouse);
```

The value displayed by label `"name"` will be `"John"` and not the spouse's name `"Jill"` as you may expect.
In this example the label doesn't own a model, so it must search up its container hierarchy for an inheritable model. However, also its container (`WebMarkupContainer` with id '`spouse`') doesn't own a model, hence the request for a model is forwarded to the upper container, which is the page. Finally, label inherits `CompoundPropertyModel` from page but only its own id is used for the property expression.
The containers in between are never taken into account for the final property expression.

9.4 Component DropDownChoice

Class `org.apache.wicket.markup.html.form.DropDownChoice` is the form component needed to display a list of possible options as a drop-down list where users can select one of the proposed options. This component must be used with tag `<select>`:

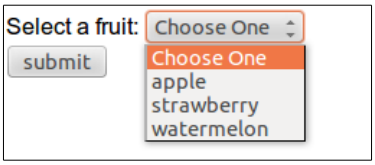
Html:

```
<form wicket:id="form">
        Select a fruit: <select wicket:id="fruits"></select>
        <div><input type="submit" value="submit"/></div>
</form>
```

Java code:

```
List<String> fruits = Arrays.asList("apple", "strawberry", "watermelon");
form.add(new DropDownChoice<String>("fruits", new Model(), fruits));
```

Screenshot of generated page:



In addition to the component id, in order to build a `DropDownChoice` we need to provide to its constructor two further parameters:

- a model containing the current selected item. This parameter is not required if we are going to inherit a `CompundPropertyModel` for this component.

- a list of options to display which can be supplied as a model or as a regular `java.util.List`.

In the example above the possible options are provided as a list of `String` objects. Now let's take a look at the markup generated for them:

```
<select name="fruits" wicket:id="fruits">
        <option value="" selected="selected">Choose One</option>
        <option value="0">apple</option>
        <option value="1">strawberry</option>
        <option value="2">watermelon</option>
</select>
```

The first option is a placeholder item corresponding to a `null` model value. By default `DropDownChoice` cannot have a `null` value so users are forced to select a not-null option. If we want to change this behavior we can set the `nullValid` flag to `true` with method `setNullValid`. Please note that the placeholder text ("Chose one") can be localized, as we will see in chapter 12.

The other options are identified by the attribute `value`. By default the value of this attribute is the index of the single option inside the provided list of choices, while the text displayed to user is obtained by simply calling `toString()` on the choice object. This default behavior works fine as long as our options are simple objects like strings, but when we move to more complex objects we may need to implement a more sophisticated algorithm to generate the value to use as option id and the one to display to user.

Wicket has solved this problem introducing interface `org.apache.wicket.markup.html.form.IChoiceRender`. This interface defines method `getDisplayValue(T object)` that is called to generate the value to display for the given choice `object`, and method `getIdValue(T object, int index)` that is called to generate the option id.

The built-in implementation of this interface is class `org.apache.wicket.markup.html.Form.ChoiceRenderer` which renders the two values using property expressions.

In the following code we want to show a list of `Person` objects using their full name as value to display and using their passport code as option id:

Java code:

```
List<Person> persons;
//Initialize the list of persons here...
ChoiceRenderer personRenderer = new ChoiceRenderer("fullName", "passportCode");
form.add(new DropDownChoice<String>("persons", new Model<Person>(), persons, personRenderer));
```

The choice renderer can be assigned to `DropDownChoice` using one of its constructor that accepts this type of parameter (like we did in the example above) or after its creation with method `setChoiceRenderer`.

9.5 Model chaining

Models that implement the interface `org.apache.wicket.model.IChainingModel` can be used to build a *chain of models*.

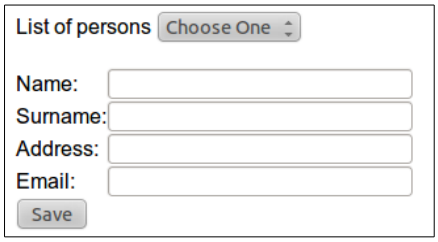
These kinds of models are able to recognize whether their model object is itself an implementation of `IModel` and if so, they will call `getObject` on the wrapped model and the returned value will be the actual model object. In this way we can combine the action of an arbitrary number of models, making exactly a chain of models.

Chaining models allows to combine different data persistence strategies, similarly to what we do with

chains of I/O streams.[27]

To see model chaining in action we will build a page that implements the List/Detail View pattern, where we have a drop-down list of `Person` objects and a form to display and edit the data of the current selected `Person`.

The example page will look like this:



What we want to do in this example is to chain the model of the `DropDownChoice` (which contains the selected `Person`) with the model of the `Form`. In this way the `Form` will work with the selected `Person` as backing object.

Component `DropDownChoice` can be configured to automatically update its model each time we change the selected item on client side. All we have to do is to override method `wantOnSelectionChangedNotifications` to make it return `true`. In practice, when this method returns `true`, `DropDownChoice` will submit its value every time JavaScript event `onChange` occurs, and its model will be consequently updated. To leverage this functionality, `DropDownChoice` doesn't need to be inside a form.

The following is the resulting markup of the example page:

```
...
<body>
        List of persons <select wicket:id="persons"></select> <br/>
        <br/>
        <form wicket:id="form">
                <div style="display: table;">
                        <div style="display: table-row;">
                                <div style="display: table-cell;">Name: </div>
                                <div style="display: table-cell;">
                                        <input type="text" wicket:id="name"/>
                                </div>
                        </div>
                        <div style="display: table-row;">
                                <div style="display: table-cell;">Surname: </div>
                                <div style="display: table-cell;">
                                        <input type="text" wicket:id="surname"/>
                                </div>
                        </div>
                        <div style="display: table-row;">
                                <div style="display: table-cell;">Address: </div>
                                <div style="display: table-cell;">
                                        <input type="text" wicket:id="address"/>
                                </div>
                        </div>
                        <div style="display: table-row;">
                                <div style="display: table-cell;">Email: </div>
                                <div style="display: table-cell;">
                                        <input type="text" wicket:id="email"/>
                                </div>
                        </div>
```

27 http://java.sun.com/developer/technicalArticles/Streams/ProgIOStreams

```
                </div>
                <input type="submit" value="Save"/>
        </form>
</body>
...
```

The initialization code for `DropDownChoice` is the following:

```
Model<Person> listModel = new Model<Person>();
ChoiceRenderer<Person> personRender = new ChoiceRenderer<Person>("fullName");

 personsList = new DropDownChoice<Person>("persons", listModel, loadPersons(),
                                          personRender){
       @Override
       protected boolean wantOnSelectionChangedNotifications() {
              return true;
       }
};
```

As choice render we have used the basic implementation provided with class `org.apache.wicket.markup.html.form.ChoiceRenderer` that we have seen in the previous paragraph. `loadPersons()` is just an utility method which generates a list of `Person` instances. The model for `DropDownChoice` is a simple instance of `Model` class.

Here is the whole code of the page (except method `loadPersons()`):

```
public class PersonListDetails extends WebPage {
  private Form form;
  private DropDownChoice<Person> personsList;

  public PersonListDetails(){
    Model<Person> listModel = new Model<Person>();
    ChoiceRenderer<Person> personRender = new ChoiceRenderer<Person>("fullName");

    personsList = new DropDownChoice<Person>("persons", listModel, loadPersons(),
                                                          personRender){
      @Override
      protected boolean wantOnSelectionChangedNotifications() {
        return true;
      }
    };

    add(personsList);

    form = new Form("form", new CompoundPropertyModel<Person>(listModel));
    form.add(new TextField("name"));
    form.add(new TextField("surname"));
    form.add(new TextField("address"));
    form.add(new TextField("email"));

    add(form);
  }
       //loadPersons()
       //...
```

```
}
```

The two models work together as a pipeline where the output of method `getObject` of `Model` is the model object of `CompoundPropertyModel`. As we have seen, model chaining allows to combine the actions of two or more models without creating new custom implementations.

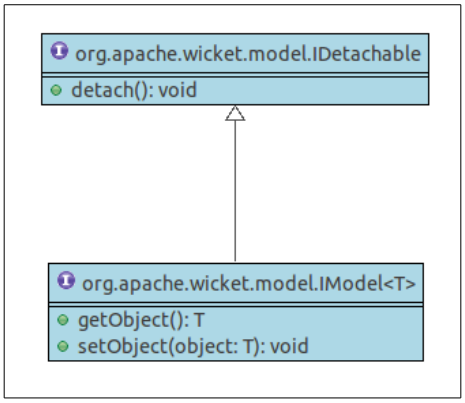
9.6 Detachable models.

In chapter 6 we saw how Wicket uses serialization to store page instances. When an object is serialized also all its referenced objects are recursively serialized. For a page this means that also all its children components, their related models and the model object inside them will be serialized.

For model objects this could be a serious issue for (at least) two main reasons:

1. The model object could be a very large instance, hence serialization would become very expensive in terms of time and memory.
2. We simply may not be able to use a serializable object as model object. In paragraphs 1.4 and 9.2 we stated that Wicket allows us to use POJO as backing object, but POJOs are ordinary objects with no prespecified interface, annotation or superclass[28], hence they are not required to implement standard interface `Serializable`.

To cope with these problems `IModel` extends another interface called `IDetachable`.



This interface provides a method called `detach()` which is invoked by Wicket at the end of web request processing, when data model is no more needed but before serialization occurs.

Overriding this method we can clean any reference to data object, keeping just the information needed to retrieve it later (like for example the id of the table row where our data are stored). In this way we can avoid the serialization of the object wrapped into the model, overcoming both the problem with non-serializable objects and the one with large data objects.

Since `IModel` inherits from `IDetachable`, every model of Wicket is “detachable”, although not all of them implement a detaching policy (like class `Model`).

Usually detaching operations are strictly dependent on the persistence technology adopted for model objects (like a relational db, a NoSQL db, a queue, etc...), so it's not unusual to write a custom detachable model suited for the persistence technology chosen for a given project.

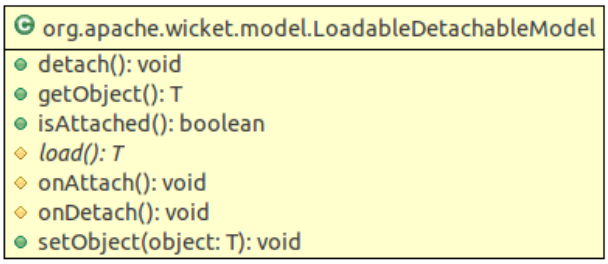
To ease this task Wicket provides abstract model `LoadableDetachableModel`. This class internally holds a transient reference to model object which is initialized the first time `getObject()` is called to precess a request. The concrete data loading is delegated to abstract method `T load()`.

The reference to model object is automatically set to `null` at the end of the request by method

28 *See definition at http://en.wikipedia.org/wiki/Plain_Old_Java_Object#Definition*

`detach()`.

The following class diagram summarizes the methods defined inside `LoadableDetachableModel`.



`onDetach` and `onAttach` can be overridden in order to obtain further control over the detaching procedure.

Now as example of a possible use of `LoadableDetachableModel`, we will build a model designed to work with entities managed with JPA[29]. To understand the following code a basic knowledge of JPA is required even if we won't go into the detail of this standard.

Warning

The following model is provided for example purposes only and is not intended to be used in production environment. Important aspects like transaction management are not taken into account and you should rework the code before considering to use it.

```
public class JpaLoadableModel<T> extends LoadableDetachableModel<T> {

  private EntityManagerFactory entityManagerFactory;
  private Class<T> entityClass;
  private Serializable identifier;
  private List<Object> constructorParams;

  public JpaLoadableModel(EntityManagerFactory entityManagerFactory, T entity) {
     super();
     PersistenceUnitUtil util = entityManagerFactory.getPersistenceUnitUtil();

      this.entityManagerFactory = entityManagerFactory;
      this.entityClass = (Class<T>) entity.getClass();
      this.identifier = (Serializable) util.getIdentifier(entity);

      setObject(entity);
  }

  @Override
  protected T load() {
     T entity = null;

     if(identifier != null){
       EntityManager entityManager = entityManagerFactory.createEntityManager();
       entity = entityManager.find(entityClass, identifier);
     }
```

29 http://en.wikipedia.org/wiki/Java_Persistence_API

```
        return entity;
    }

    @Override
    protected void onDetach() {
        super.onDetach();

        T entity = getObject();
        PersistenceUnitUtil persistenceUtil = entityManagerFactory.getPersistenceUnitUtil();

        if(entity == null) return;

        identifier = (Serializable) persistenceUtil.getIdentifier(entity);
    }
}
```

The constructor of the model takes in input two parameters: an implementation of the JPA interface `javax.persistence.EntityManagerFactory` to manage JPA entities and the entity that must be handled by this model. Inside its constructor the model saves the class of the entity and its id (which could be `null` if the entity has not been persisted yet). These two informations are required to retrieve the entity at a later time and are use by method `load`.

`onDetach` is responsible for updating the entity id before detachment occurs. The id can change the first time an entity is persisted (JPA generates a new id and assigns it to the entity).

Please note that this model is not responsible for saving any changes occurred to the entity object before it is detached. If we don't want to loose these changes we must explicitly persist the entity before the detaching phase occurs.

Warning

Since the model of this example holds a reference to interface `EntityManagerFactory`, the implementation in use must be serializable.

9.7 Using more than one model in a component

Sometimes our custom components may need to use more than a single model to work properly. In such a case we must manually detach the additional models used by our components. In order to do this we can overwrite `Component`'s method `onDetach` that is called at the end of the current request. The following is the generic code of a component that uses two models:

```
/**
 *
 * fooModel is used as main model while beeModel must be manually detached
 *
 */
public class ComponetTwoModels extends Component{

        private IModel<Bee> beeModel;

        public ComponetTwoModels(String id, IModel<Foo> fooModel, IModel<Bee> beeModel) {
                super(id, fooModel);
                this.beeModel = beeModel;
        }
```

```
        @Override
        public void onDetach() {
                if(beeModel != null)
                   beeModel.detach();

                 super.onDetach();
        }
}
```

When we overwrite `onDetach` we must call the super class implementation of this method, usually as last line in our custom implementation.

9.8 Use models!

Like many people new to Wicket, you may need a little time to fully understand the power and the advantages of using models. Moving your first steps with Wicket you may be tempted to pass row objects to your components instead of using models:

```
/**
 *
 * NOT TO DO: passing row objects to components instead of using models!
 *
 */
public class CustomComponent extends Component{
        private FooBean fooBean;

        public CustomComponent(String id, FooBean fooBean) {
                super(id);
                this.fooBean = fooBean;
        }
        //...some other ugly code :)...
}
```

That's absolutely a bad practice and you must avoid it. Using models we not only decouple our components from data source, but we can also relay on them (if they are dynamic) to work with the most up-to-date version of our model object. If we decide to bypass models we lose all these advantages and, moreover, we force model objects to be serialized.

9.9 Summary

Models are one of the core concepts of Wicket and they are the basic ingredient needed to taste the real power of this framework. In this chapter we have seen how to use models to bring data to our components without littering their code with technical details about their persistence strategy.

We have also introduced Wicket forms as complementary topic. With forms and models we are able to bring our applications to life allowing them to interact with users.

But what we have seen in this chapter about Wicket forms is just the tip of the iceberg. That's why the next chapter is entirely dedicated to them.

10 Wicket forms in detail

In the previous chapter we have only scratched the surface of Wicket forms. Component `Form` was designed to not simply collect user input but also to extend the semantic of the classic HTML forms with new features.

For example one of such features is the ability to work with *nested forms* (they will be discussed in paragraph 10.5).

In this chapter we will continue to explore Wicket forms learning how to master them and how to build effective and user-proof forms for our web applications.

10.1 Default form processing

In paragraph 9.3 we have seen a very basic usage of `Form` component and we didn't pay much attention to what happens behind the scenes of form submission. In Wicket when we submit a form we trigger the following steps on server side:

1. **Form validation:** user input is checked to see if it satisfies the validation rules set on the form. If validation fails, step number 2 is skipped and the form should display a feedback message to explain to user what went wrong. During this step input values (which are simple strings sent with a web request) are converted into Java objects.
   In the next paragraphs we will explore the infrastructures provided by Wicket for the three sub-tasks involved into form validation, which are: *conversion of user input into objects*, *validation of user input*, and *visualization of feedback messages*.
2. **Updating of models:** if validation succeeds, the form updates the model of its children components with the converted values obtained in the previous step.
3. **Invoking callback methods *onSubmit()* or *onError()*:** if we didn't have any validation error, method `onSubmit()` is called, otherwise `onError()` will be called. The default implementation of both these methods is left empty and we can override them to perform custom actions.

Note

Please note that the model of form components is updated only if no validation error occurred (i.e. step two is performed only if validation succeeds).

Without going into too much detail, we can say that the first two steps of form processing correspond to the invocation of one or more `Form`'s internal methods (which are declared `protected` and `final`). Some examples of these methods are `validate()`, which is invoked during validation step, and `updateFormComponentModels()`, which is used during models updating step.

The whole form processing is started invoking public method `process(IFormSubmitter)` (Later in paragraph 10.4 we will introduce interface `IFormSubmitter`).

10.2 Form validation and feedback messages

A basic example of validation rule is to make a field required. In paragraph 9.3.2 we have already seen how this can be done calling `setRequired(true)` on a field. However, more generally, to set a validation rule on a `FormComponent` we must add the corresponding *validator* to it.

A *validator* is an implementation of interface `org.apache.wicket.validation.IValidator` and `FormComponent` has a version of method `add` which takes in input this interface.

For example if we want to use a text field to insert an email address, we could use the built-in validator `EmailAddressValidator` to ensure that inserted input will respect the email format `local-part@domain`[30]:

```
TextField email = new TextField("email");
email.add(new EmailAddressValidator());
```

Wicket comes with a set of built-in validators that should suit most of our needs. We will see them in paragraph 10.2.3.

10.2.1 Feedback messages and localization

Wicket generates a feedback message for each field that doesn't satisfy one of its validation rules. For example the message generated when a required field is left empty is the following

```
Field '<label>' is required.
```

`<label>` is the value of the *label model* set on a `FormComponent` with method `setLabel(IModel <String> model)`. If such model is not provided, component id will be used as default value.

The entire infrastructure of feedback messages is built on top of Java internationalization (I18N) support and it uses *resource bundles*[31] to store messages.

Note

The topics of internationalization will be covered in chapter 12. For now we will give just few notions needed to understand the examples from this chapter.

By default resource bundles are stored into *properties files* but we can easily configure other sources as described later in paragraph 12.4.5.

Default feedback messages (like the one above for required fields) are stored into file *Application.properties* placed inside Wicket package `org.apache.wicket`. Opening this file we can find the message used for required fields:

```
Required=Field '${label}' is required.
```

We can note the key used to identify the bundle (`Required` in our case) and the `label` parameter written with *expression language*[32] (`${label}`). Scrolling down this file we can also find the message used by validator `EmailAddressValidator`:

```
EmailAddressValidator=The value of '${label}' is not a valid email address.
```

By default `FormComponent` provides 3 parameters for feedback message: `input` (the value that failed validation), `label` and `name` (this later is the id of the component).

Warning

Remember that component model is updated with user input only if validation succeeds! As a consequence, we can't retrieve the wrong value inserted for a

30 http://en.wikipedia.org/wiki/Email_address
31 http://docs.oracle.com/javase/tutorial/i18n/resbundle/index.html
32 http://en.wikipedia.org/wiki/Expression_Language

field from its model. Instead, we should use method `getValue()` of class `FormComponent`. (This method will be introduced in the example used in paragraph 10.2.5)

10.2.2 Displaing feedback messages and filtering them

To display feedback messages we must use component `org.apache.wicket.markup.html.panel.FeedbackPanel`. This component automatically reads all the feedback messages generated during form validation and displays them with an unordered list:

```
<ul class="feedbackPanel">
        <li class="feedbackPanelERROR">
                <span class="feedbackPanelERROR">Field 'Username' is required.</span>
        </li>
</ul>
```

CSS classes "feedbackPanel" and "feedbackPanelERROR" can be used in order to customize the style of the message list[33]:

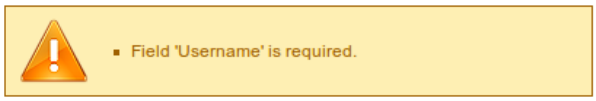


Illustration 10.1: Example of styling of feedback messages list

The component can be freely placed inside the page and we can set the maximum amount of displayed messages with method `setMaxMessages()`.

Error messages can be also filtered using three built-in filters:

- **ComponentFeedbackMessageFilter:** shows only messages coming from a specific component.
- **ContainerFeedbackMessageFilter:** shows only messages coming from a specific container or from any of its children components.
- **ErrorLevelFeedbackMessageFilter:** shows only messages with a level of severity equals or greater than a given lower bound. Class `FeedbackMessage` defines a set of static constants to express different levels of severity: DEBUG, ERROR, WARNING, INFO, SUCCESS, etc.... Levels of severity for feedback messages are discussed in paragraph 10.2.6.

These filters are intended to be used when there are more than one feedback panel (or more than one form) in the same page. We can pass a filter to a feedback panel via constructor or using its setter method `setFilter`. Custom filters can be created implementing interface `IFeedbackMessageFilter`. An example of custom filter is illustrated on page 89.

10.2.3 Built-in validators

Wicket already provides a number of built-in validators ready to be used. The following table is a short reference where validators are listed along with a brief description of what they do. The default feedback message used by each of them is reported as well:

33 *The style of Illustration 10.1 was created by Janko Jovanovic. See http://css.dzone.com/news/css-message-boxes-different-me*

Name	Description	Message
EmailAddressValidator	Checks if input respects the format `local-part@domain`	`The value of '${label}' is not a valid email address.`
UrlValidator	Checks if input is a valid URL. We can specify in the constructor which protocols are allowed (`http://`, `https://`, and `ftp://`).	`The value of '${label}' is not a valid URL.`

DateValidator	Validator class that can be extended or used as a factory class to get date validators to check if a date is bigger than a lower bound (method `minimum(Date min)`), smaller than a upper bound (method `maximum(Date max)`) or inside a range (method `range(Date min, Date max)`).	**For minimum validator:**
		`The value of '${label}' is less than the minimum of ${minimum}.`
		For maximum validator:
		`The value of '${label}' is larger than the maximum of ${maximum}.`
		For range validator:
		`The value of '${label}' is not between ${minimum} and ${maximum}.`

RangeValidator	Validator class that can be extended or used as a factory class to get validators to check if a value is bigger than a given lower bound (method `minimum(T min)`), smaller than a upper bound (method `maximum(T max)`) or inside a range (method `range(T min,T max)`).
	The type of the value is a generic subtype of `java.lang.Comparable` and must implement `Serializable` interface.
		For minimum validator:
		`The value of '${label}' must be at least ${minimum}.`
		For maximum validator:
		`The value of '${label}' must be at most ${maximum}.`
		For range validator:
		`The value of '${label}' must be between ${minimum} and ${maximum}.`

StringValidator	Validator class that can be extended or used as a factory class to get validators to check if the length of a string value is bigger then a given lower bound (method `minimumLength (int min)`), smaller then a given upper bound (method `maximumLength (int max)`) or within a given range (method `lengthBetween(int min, int max)`).
	To accept only string values consisting of exactly *n* characters, we must use method `exactLength(int length)`.
		For minimum validator:
		`The value of '${label}' is shorter than the minimum of ${minimum} characters.`
		For maximum validator:
		`The value of '${label}' is longer than the maximum of ${maximum} characters.`
		For range validator:
		`The value of '${label}' is not between ${minimum} and ${maximum} characters long.`

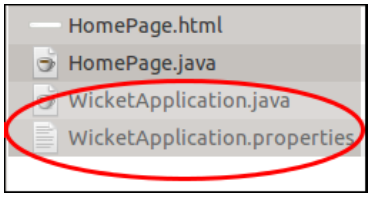
		For exact validator: `The value of '${label}' is not exactly ${exact} characters long.`
CreditCardValidator	Checks if input is a valid credit card number. This validator supports some of the most popular credit cards (like “American Express", "MasterCard", “Visa” or “Diners Club”).	`The credit card number is invalid.`
EqualPasswordInputValidator	This validator checks if two password fields have the same value.	`${label0} and ${label1} must be equal.`

10.2.4 Overriding standard feedback messages with custom bundles

If we don't like the default validation messages, we can override them providing custom properties files. In these files we can write our custom messages using the same keys of the messages we want to override. For example if we wanted to override the default message for invalid email addresses, our properties file would contain a line like this:

```
EmailAddressValidator=Man, your email address is not good!
```

As we will see in the next chapter, Wicket searches for custom properties files in various positions inside application's class path, but for now we will consider just the properties file placed next to our application class. The name of this file must be equal to the name of our application class:



The example project *OverrideMailMessage* overrides email validator's message with a new one which reports also the value that failed validation:

```
EmailAddressValidator=The value '${input}' inserted for field '${label}' is not a valid
                      email address.
```

10.2.5 Creating custom validators

If our application requires a complex validation logic and built-in validators are not enough, we can always implement our own custom validators. For example (project *UsernameCustomValidator*) suppose we are working on the registration page of our site where users can create their profile choosing their username. Our registration form should validate the new username checking if it was

already chosen by another user. In a situation like this we may need to implement a custom validator that queries a specific data source to check if a username is already in use.

Although validators are nothing but implementations of interface `IValidator`, Wicket provides the convenience class `org.apache.wicket.validation.validator.AbstractValidator` as base class for custom validators. For the sake of simplicity, the validator of our example will check the given username against a fixed list of three existing usernames:

```
public class UsernameValidator extends AbstractValidator<String> {
	List<String> existingUsernames = Arrays.asList("bigJack", "anonymous", "mrSmith");

	@Override
	protected void onValidate(IValidatable<String> validatable) {
		String chosenUserName = validatable.getValue();

		if(existingUsernames.contains(chosenUserName))
			error(validatable);
	}

	@Override
	protected String resourceKey() {
		return "UsernameValidator";
	}

	@Override
	protected Map<String, Object> variablesMap(IValidatable<String> validatable) {
		Map<String, Object> map = super.variablesMap(validatable);
		Random random = new Random();

		map.put("suggestedUserName", validatable.getValue() + random.nextInt());

		return map;
	}
}
```

Class `AbstractValidator` comes with three methods that can be overridden to implement a custom validation logic:

- `onValidate`: this method contains the concrete validation logic and must call method `error(Ivalidatable)` when validation fails. It takes in input an instance of interface `IValidatable` which represents the component being validated.
- `resourceKey`: returns the key of the resource corresponding to the feedback message for this validator. The default implementation of this method returns the name of the class, hence in the example above the overridden version of `resourceKey` is redundant.
- `variablesMap`: this method can be overridden to provide further variables to the feedback message.

In our example if validation fails, we suggest a possible username concatenating the given input with a pseudo-random integer. This value is passed to the feedback message with variable `suggestedUserName`. The message is inside application's properties file:

```
UsernameValidator=The username '${input}' is already in use. Try with
                  '${suggestedUserName}'
```

The code of the home page of the project will be examined in the next paragraph after we have introduced the topic of *flash messages*.

10.2.6 Using flash messages

So far we have considered just the error messages generated during validation step. However Wicket `Component` class provides a set of methods to explicitly generate feedback messages called *flash messages*. These methods are:

- `debug(Serializable message)`
- `info(Serializable message)`
- `success(Serializable message)`
- `warn(Serializable message)`
- `error(Serializable message)`
- `fatal(Serializable message)`

Each of these methods corresponds to a *level of severity* for the message. The list above is sorted by increasing level of severity.

In the example seen in the previous paragraph we have a form which uses `success` method to notify user when the inserted username is valid. Inside this form there are two `FeedbackPanel` components: one to display the error message produced by custom validator and the other one to display the success message. The code of the example page is the following:

Html:

```
<body>
        <form wicket:id="form">
                Username: <input type="text" wicket:id="username"/>
                <br/>
                <input type="submit"/>
        </form>
        <div style="color:green" wicket:id="succesMessage">
        </div>
        <div style="color:red" wicket:id="feedbackMessage">
        </div>
</body>
```

Java code:

```
public class HomePage extends WebPage {

    public HomePage(final PageParameters parameters) {
        Form form = new Form("form"){
                @Override
                protected void onSubmit() {
                        super.onSubmit();
                        success("Username is good!");
                }
        };

        TextField mail;

        form.add(mail = new TextField("username", Model.of("")));
        mail.add(new UsernameValidator());

        add(new FeedbackPanel("feedbackMessage",
```

```
                new ExactErrorLevelFilter(FeedbackMessage.ERROR)));
        add(new FeedbackPanel("succesMessage",
                new ExactErrorLevelFilter(FeedbackMessage.SUCCESS)));

        add(form);
    }

    class ExactErrorLevelFilter implements IFeedbackMessageFilter{
        private int errorLevel;

        public ExactErrorLevelFilter(int errorLevel){
                this.errorLevel = errorLevel;
        }

        public boolean accept(FeedbackMessage message) {
                return message.getLevel() == errorLevel;
        }

    }
    //UsernameValidator definition
    //...
}
```

The two feedback panels must be filtered in order to display just the messages with a given level of severity (ERROR for validator message and SUCCESS for form's flash message). Unfortunately the built-in message filter `ErrorLevelFeedbackMessageFilter` is not suitable for this task because its filter condition does not check for an exact error level (the given level is used as lower bound value). As a consequence, we had to build a custom filter (inner class `ExactErrorLevelFilter`) to accept only the desired severity level (see method `accept` of interface `IFeedbackMessageFilter`).

10.3 Input value conversion

Working with Wicket we will rarely need to worry about conversion between input values (which are strings) and Java types because in most cases the default conversion mechanism will be smart enough to infer the type of the model object and perform the proper conversion. However, sometimes we may need to work under the hood of this mechanism to make it properly work or to perform custom conversions. That's why this paragraph will illustrate how to control input value conversion.

The component that is responsible for converting input is the `FormComponent` itself with its method `convertInput()`. In order to convert its input a `FormComponent` must know the type of its model object. This parameter can be explicitly set with method `setType(Class<?> type)`:

```
//this field must receive an integer value
TextField integerField = new TextField("number", new
Model()).setType(Integer.class));
```

If no type has been provided, `FormComponent` will try to ask its model for this information. Models `PropertyModel` and `CompoundPropertyModel` can use *reflection* to get the type of object model. By default, if `FormComponent` can not obtain the type of its model object in any way, it will consider it as a simple `String`.

Once `FormComponent` has determined the type of model object, it can look up for a *converter*, which is the entity in charge of converting input to Java object and vice versa. Converters are instances of interface `org.apache.wicket.util.convert.IConverter` and are registered by our application

class on start up.

To get a converter for a specific type we must call method `getConverter(Class<C> type)` on the interface `IConverterLocator` returned by `Application`'s method `getConverterLocator()`:

```
//retrieve converter for Boolean type
Application.get().getConverterLocator().getConverter(Boolean.class);
```

Note

Components which are subclasses of `AbstractSingleSelectChoice` don't follow the schema illustrated above to convert user input.
These kinds of components (like `DropDownChoice` and `RadioChoice`[34]) use their choice render and their collection of possible choices to perform input conversion.

10.3.1 Creating custom application-scoped converters

The default converter locator used by Wicket is `org.apache.wicket.ConverterLocator`. This class provides converters for the most common Java types. Here we can see the converters registered inside its constructor:

```
public ConverterLocator()
{
        set(Boolean.TYPE, BooleanConverter.INSTANCE);
        set(Boolean.class, BooleanConverter.INSTANCE);
        set(Byte.TYPE, ByteConverter.INSTANCE);
        set(Byte.class, ByteConverter.INSTANCE);
        set(Character.TYPE, CharacterConverter.INSTANCE);
        set(Character.class, CharacterConverter.INSTANCE);
        set(Double.TYPE, DoubleConverter.INSTANCE);
        set(Double.class, DoubleConverter.INSTANCE);
        set(Float.TYPE, FloatConverter.INSTANCE);
        set(Float.class, FloatConverter.INSTANCE);
        set(Integer.TYPE, IntegerConverter.INSTANCE);
        set(Integer.class, IntegerConverter.INSTANCE);
        set(Long.TYPE, LongConverter.INSTANCE);
        set(Long.class, LongConverter.INSTANCE);
        set(Short.TYPE, ShortConverter.INSTANCE);
        set(Short.class, ShortConverter.INSTANCE);
        set(Date.class, new DateConverter());
        set(Calendar.class, new CalendarConverter());
        set(java.sql.Date.class, new SqlDateConverter());
        set(java.sql.Time.class, new SqlTimeConverter());
        set(java.sql.Timestamp.class, new SqlTimestampConverter());
        set(BigDecimal.class, new BigDecimalConverter());
}
```

If we want to add more converters to our application, we can override `Application`'s method `newConverterLocator` which is used by application class to build its converter locator.

To illustrate how to implement custom converters and use them in our application, we will build a form with two text field: one to insert a regular expression pattern and another one to insert a string value that will be split with the given pattern.

34 *RadioChoice is introduced in paragraph 10.10.3*

The first text field will have an instance of class `java.util.regex.Pattern` as model object. The final page will look like this (the code of this example is from project *CustomConverter.*):

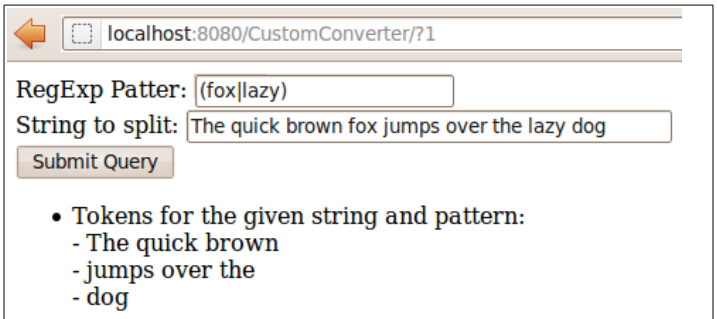


Illustration 10.2: A form with a TextField using a java.util.regex.Pattern as model object

The conversion between `Pattern` and `String` is quite straightforward. The code of our custom converter is the following:

```
public class RegExpPatternConverter implements IConverter<Pattern> {
	@Override
	public Pattern convertToObject(String value, Locale locale) {
		return Pattern.compile(value);
	}

	@Override
	public String convertToString(Pattern value, Locale locale) {
		return value.toString();
	}
}
```

Methods declared by interface `IConverter` take in input also a `Locale` parameter in order to deal with locale-sensitive data and conversions. We will learn more about locales and internationalization in chapter 12.

Once we have implemented our custom converter, we must override method `newConverterLocator()` inside our application class and tell it to add our new converter to the default set:

```
@Override
protected IConverterLocator newConverterLocator() {
	ConverterLocator defaultLocator = new ConverterLocator();

	defaultLocator.set(Pattern.class, new RegExpPatternConverter());

	return defaultLocator;
}
```

Finally, in the home page of the project we build the form which displays (with a flash message) the tokens obtained splitting the string with the given pattern:

```
public class HomePage extends WebPage {
```

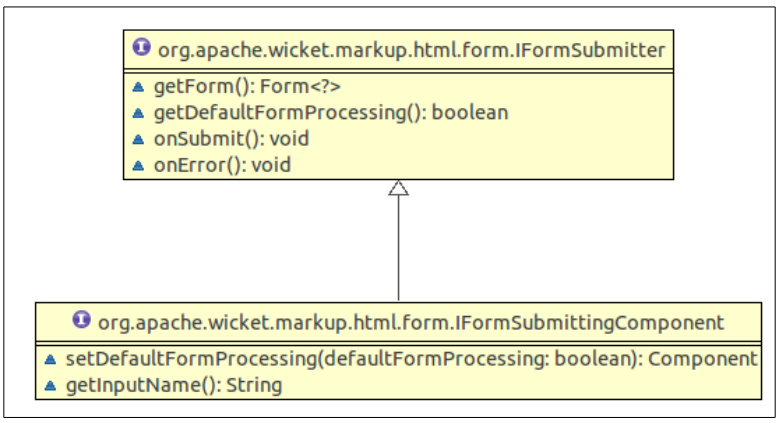
```
    private Pattern regExpPatter;
    private String stringToSplit;

    public HomePage(final PageParameters parameters) {
       TextField mail;
       TextField stringToSplitTxt;

       Form form = new Form("form"){
              @Override
              protected void onSubmit() {
                     super.onSubmit();
                     String messageResult = "Tokens for the given string and pattern:<br/>";
                     String[] tokens = regExpPatter.split(stringToSplit);

                     for (String token : tokens) {
                            messageResult += "- " + token + "<br/>";
                     }
                     success(messageResult);
              }
       };

       form.setDefaultModel(new CompoundPropertyModel(this));
       form.add(mail = new TextField("regExpPatter"));
       form.add(stringToSplitTxt = new TextField("stringToSplit"));
       add(new FeedbackPanel("feedbackMessage").setEscapeModelStrings(false));

       add(form);
    }
}
```

Note

If the inserted input can not be converted to the target type, `FormComponent` will generate the default error message "`The value of '${label}' is not a valid ${type}.`". The bundle key for this message is `IConverter`.

10.4 Submit form with an IFormSubmittingComponent

Besides submitting forms with a standard HTML submit button, Wicket allows us to use special components which implement interface `IFormSubmittingComponent`. This entity is a subinterface of `IFormSubmitter`:



At the beginning of this chapter we have seen that form processing is started by method `process` which takes in input an instance of `IFormSubmitter`. This parameter corresponds to the `IFormSubmittingComponent` clicked by user to submit the form and it is `null` if we have used a standard HTML submit button (like we have done so far).

A submitting component is added to a form just like any other child component using method `add(Component...)`.

A form can have any number of submitting components and we can specify which one among them is the default one calling `Form`'s method `setDefaultButton(IFormSubmittingComponent component)`. The default submitter is the one that will be used when user presses 'Enter' key in a field of the form. In order to make the default button work, Wicket will add to our form an hidden `<div>` containing a text field and a submit button with some JavaScript code to trigger it:

```
<div style="width:0px;height:0px;position:absolute;left:-100px;top:-100px;
            overflow:hidden">
       <input type="text" autocomplete="off"/>
       <input type="submit" name="submit2" onclick=" var b=document...."/>
</div>
```

Just like Wicket forms, also interface `IFormSubmitter` defines methods `onSubmit` and `onError`. These two methods have the priority over the namesake methods of the form, meaning that when a form is submitted with an `IFormSubmitter`, the `onSubmit` of the submitter is called before the one of the form. Similarly, if validation errors occurs during the first step of form processing, submitter's method `onError` is called before the one of the form.

Note

Starting from Wicket version 6.0 interface `IFormSubmitter` defines a further callback method called `onAfterSubmit()`. This method is called after form's method `onSubmit()` has been executed.

10.4.1 Components Button and SubmitLink

Component `org.apache.wicket.markup.html.form.Button` is a basic implementation of a form submitter. It can be used with either tag `<input>` or `<button>`. The string model taken in input by its constructor is used as button label and it will be the value of the markup attribute `value`.

In the following snippet we have a form with two submit buttons bound to a `<input>` tag. One of them is set as default button and both have a string model for the label:

Html:

```
<body>
       <form wicket:id="form">
              Username: <input type="text" wicket:id="username"/>
              <br/>
              <input type="submit" wicket:id="submit1"/>
              <input type="submit" wicket:id="submit2"/>
       </form>
</body>
```

Java code:

```
public class HomePage extends WebPage {

    public HomePage(final PageParameters parameters) {
```

```
        Form form = new Form("form");

        form.add(new TextField("username", Model.of("")));
        form.add(new Button("submit1", Model.of("First submitter")));
        Button secondSubmitter;
        form.add(secondSubmitter = new Button("submit2", Model.of("Second submitter")));

        form.setDefaultButton(secondSubmitter);
        add(form);
    }
}
```

Generated markup:

```
<form wicket:id="form" id="form1" method="post" action="?0-1.IFormSubmitListener-form">
   <div>
      ...
      <!-- Code generated by Wicket to handle the default button -->
      ...
   </div>
   Username: <input type="text" wicket:id="username" value="" name="username"/>
   <br/>
   <input type="submit" wicket:id="submit1" name="submit1" id="submit13" value="First
submitter"/>
   <input type="submit" wicket:id="submit2" name="submit2" id="submit22" value="Second
submitter"/>
</form>
```

Another component that can be used to submit a form is `org.apache.wicket.markup.html.form.SubmitLink`. This component uses JavaScript to submit the form. Like the name suggests, the component can be used with tag `<a>`, but it can be also bound to any other tag that supports the event handler `onclick`. When used with tag `<a>`, the JavaScript code needed to submit the form will be placed inside `href` attribute while with other tags the script will go inside the event handler `onclick`.

A notable difference between this component and `Button` is that `SubmitLink` can be placed outside the form it must submit. In this case we must specify the form to submit in its constructor:

Html:

```
<html xmlns:wicket="http://wicket.apache.org">
        <head>
        </head>
        <body>
                <form wicket:id="form">
                        Password: <input type="password" wicket:id="password"/>
                        <br/>
                </form>
                <button wicket:id="externalSubmitter">
                        Submit
                </button>
        </body>
</html>
```

Java code:

```
public class HomePage extends WebPage {

    public HomePage(final PageParameters parameters) {
                Form form = new Form("form");

                form.add(new PasswordTextField("password", Model.of("")));
                //specify the form to submit
                add(new SubmitLink("externalSubmitter", form));
                add(form);
    }
}
```

10.4.2 Disabling default form processing

With an `IFormSubmittingComponent` we can choose to skip the default form submission process setting the appropriate flag to false with method `setDefaultFormProcessing`. When the default form processing is disabled only submitter's `onSubmit` is called while form's validation and models updating are skipped.

This can be useful if we want to implement a “Cancel” button on our form which redirects user to another page without validating his/her input.

When we set this flag to false we can decide to manually invoke form processing by calling method `process(IFormSubmittingComponent)`.

10.5 Nested forms

As you might already know, HTLM doesn't allow to have nested forms[35]. However with Wicket we can overcome this limitation adding one or more form components to a parent form.

This can be useful if we want to split a big form into smaller ones in order to reuse them and to better distribute responsibilities among different components.

Forms can be nested to an arbitrary level:

```
<form wicket:id="outerForm">
        ...
        <form wicket:id="innerForm">
                ...
                <form wicket:id="veryInnerForm">
                        ...
                </form>
        </form>
</form>
```

When a form is submitted also its nested forms are submitted and they participate to validation step. This means that if a nested form contains invalid input values, the outer form won't be submitted. On the contrary, nested forms can be singularly submitted without depending on the status of their outer form.

To submit a parent form when one of its children forms is submitted, we must override its method `wantSubmitOnNestedFormSubmit` and make it return `true`.

10.6 Multi-line text input

35 *See http://www.w3.org/MarkUp/html3/forms.html where it is stated: 'There can be several forms in a single document, but the FORM element can't be nested.'*

HTML provides a multi-line text input control with tag `<textarea>`. The Wicket counterpart for this kind of control is component `org.apache.wicket.markup.html.form.TextArea`:

Markup code:

```
<textarea wicket:id="description" rows="5" cols="40"></textarea>
```

Java code:

```
form.add(new TextArea("description", Model.of("")));
```

Component `TextArea` is used just like any other single-line text field. To specify the size of the text area we can write attributes `rows` and `cols` directly in the markup file or we can use attribute modifiers (seen in paragraph 4.2) in the Java code.

10.7 File upload

Wicket supports file uploading with component `FileUploadField` which must be used with tag `<input>` and attribute `type` equals to *"file"*. In order to send a file on form submission we must enable *multipart mode* calling `setMultiPart(true)` on our form.

In the next example (project *UploadSingleFile*) we will see a form which allows users to upload a file into the temporary directory of the server (path `/tmp` on Unix/Linux systems):

Html:

```
<html>
        <head>
        </head>
        <body>
        <h1>Upload your file here!</h1>
        <form wicket:id="form">
                <input type="file" wicket:id="fileUploadField"/>
                <input type="submit" value="Upload"/>
        </form>
        <div wicket:id="feedbackPanel">
        </div>
        </body>
</html>
```

Java code:

```
public class HomePage extends WebPage {
   private FileUploadField fileUploadField;

    public HomePage(final PageParameters parameters) {
       fileUploadField = new FileUploadField("fileUploadField");

       Form form = new Form("form"){
               @Override
               protected void onSubmit() {
                 super.onSubmit();

                 FileUpload fileUpload = fileUploadField.getFileUpload();
```

```
                    try {
                        File file = new File(System.getProperty("java.io.tmpdir") + "/" +
                                                    fileUpload.getClientFileName());

                            fileUpload.writeTo(file);
                        } catch (IOException e) {
                           e.printStackTrace();
                        }
                }
        };

        form.setMultiPart(true);
        //set a limit for uploaded file's size
        form.setMaxSize(Bytes.kilobytes(100));
        form.add(fileUploadField);
        add(new FeedbackPanel("feedbackPanel"));
        add(form);
    }
}
```

The code that copies the uploaded file into the temporary directory is inside form's `onSubmit`. The uploaded file is handled with an instance of class `FileUpload` returned by `FileUploadField`'s method `getFileUpload()`. This class provides a set of methods to perform some common tasks like getting the name of the uploaded file (`getClientFileName()`), coping the file into a directory (`writeTo(destinationFile)`), calculating file digest (`getDigest(digestAlgorithm)`) and so on.

`Form` component can limit the size for uploaded files using its method `setMaxSize(size)`. In the example we have set this limit to 100 kb to prevent users from uploading files bigger then this size.

Note

The maximum size for uploaded files can be also set at application level using method `setDefaultMaximumUploadSize(Bytes maxSize)` of setting interface `IApplicationSettings`:

```
@Override
public void init()
{
 getApplicationSettings().setDefaultMaximumUploadSize(Bytes.
                                                    kilobytes(100));
}
```

10.7.1 Upload multiple files

If we need to upload multiple files at once, we can use component `MultiFileUploadField` which allows user to select an arbitrary number of files to send on form submission.

An example showing how to use this component can be found in Wicket module `wicket-examples` in page `MultiUploadPage.java`. The live example is hosted at http://www.wicket-library.com/wicket-examples-6.0.x/upload/multi.

10.8 Creating complex form components with FormComponentPanel

In chapter 3.2.2 we have learnt how to use class `Panel` to create custom components with their own markup and with an arbitrary number of children components.

While it's perfectly legal to use `Panel` also to group form components, the resulting component won't be itself a form component and it won't participate to form submission's steps.

This could be a strong limitation if the custom component needs to coordinate its children during sub-tasks like input conversion or model updating. That's why in Wicket we have component `org.apache.wicket.markup.html.form.FormComponentPanel` which combines the features of both a `Panel` (it has it's own markup file) and a `FormComponent` (it is a subclass of `FormComponent`).

A typical scenario in which we may need to implement a custom `FormComponentPanel` is when our web application and its users work with different units of measurement for the same data.

To illustrate this possible scenario, let's consider for example a form where user can insert a temperature that will be recorded after being converted in Kelvin degrees (see example project *CustomFormComponentPanel*).

Kelvin scale is wildly adopted among the scientific community and it is one of the seven base units of the International System of Units[36], so it makes perfect sense to store temperatures expressed with this unit of measurement.

However, in our everyday life we still use other temperature scales like Celsius or Fahrenheit, so it would be nice to have a component which internally works with Kelvin degrees and automatically applies conversion between Kelvin temperature scale and the one adopted by user.

In order to implement such a component, we can make a subclass of `FormComponentPanel` and leverage methods `convertInput` and `onBeforeRender`: inside `convertInput` we will convert input value to Kelvin degrees while method `onBeforeRender` will take care of converting the Kelvin value to the temperature scale adopted by user.

Our custom component will contain two children components: a text field to let user insert and edit a temperature value and a label to display the letter corresponding to user's temperature scale (F for Fahrenheit and C for Celsius). The resulting markup file is the following:

```
<html>
<head>
</head>
<body>

  <wicket:panel>
       Registered temperature: <input size="3" maxlength="3"
                                   wicket:id="registeredTemperature"/>
       <label wicket:id="mesuramentUnit"></label>
  </wicket:panel>
</body>
</html>
```

As shown in the markup above `FormComponentPanel` uses the same tag `<wicket:panel>` used by `Panel` to define its markup. Now let's see the Java code of the new form component starting from method `onInitialize()`:

```
public class TemperatureDegreeField extends FormComponentPanel<Double> {

       private TextField<Double> userDegree;

       public TemperatureDegreeField(String id) {
              super(id);
```

36 *http://en.wikipedia.org/wiki/International_System_of_Units*

```
        }

        public TemperatureDegreeField(String id, IModel<Double> model) {
                super(id, model);
        }

        @Override
        protected void onInitialize() {
                super.onInitialize();

             AbstractReadOnlyModel<String> labelModel=new AbstractReadOnlyModel<String>(){
                        @Override
                        public String getObject() {
                                if(getLocale().equals(Locale.US))
                                        return "°F";
                                return "°C";
                        }
                };

                add(new Label("mesuramentUnit", labelModel));
                add(userDegree=new TextField<Double>("registeredTemperature", new
                                Model<Double>()));
                userDegree.setType(Double.class);
        }
```

Inside method `onInitialize` we have created a read-only model for the label that displays the letter corresponding to the user's temperature scale. To determinate which temperature scale is used by user, we retrieve the `Locale` from session with `Component`'s method `getLocale()` (we will talk more about this method in chapter 12). Then, if locale is the one corresponding to the United States, the chosen scale will be Fahrenheit, otherwise it will be considered as Celsius.

In the final part of `onInitialize()` we add the two components to our custom form component. You may have noticed that we have explicitly set the type of model object for the text field to double. This is necessary as the starting model object is a `null` reference and this prevents the component from automatically determining the type of its model object.

Now we can look at the rest of the code containing methods `convertInput` and `onBeforeRender`:

```
        @Override
        protected void convertInput() {
                Double userDegreeVal = userDegree.getConvertedInput();
                Double kelvinDegree;

                if(getLocale().equals(Locale.US)){
                        kelvinDegree = userDegreeVal +  459.67;
                        BigDecimal bdKelvin = new BigDecimal(kelvinDegree);
                        BigDecimal fraction = new BigDecimal(5).divide(new BigDecimal(9));

                        kelvinDegree = bdKelvin.multiply(fraction).doubleValue();
                }else{
                        kelvinDegree = userDegreeVal + 273.15;
                }

                setConvertedInput(kelvinDegree);
```

```
        }

        @Override
        protected void onBeforeRender() {
                super.onBeforeRender();

                Double kelvinDegree = (Double) getDefaultModelObject();
                Double userDegreeVal = null;

                if(kelvinDegree == null) return;

                if(getLocale().equals(Locale.US)){
                        BigDecimal bdKelvin = new BigDecimal(kelvinDegree);
                        BigDecimal fraction = new BigDecimal(9).divide(new BigDecimal(5));

                        kelvinDegree = bdKelvin.multiply(fraction).doubleValue();
                        userDegreeVal = kelvinDegree - 459.67;
                }else{
                        userDegreeVal = kelvinDegree - 273.15;
                }

                userDegree.setModelObject(userDegreeVal);
        }
}
```

Since our component does not directly receive the user input, `convertInput()` must read this value from the inner text field using `FormComponent`'s method `getConvertedInput()` which returns the input value already converted to the type specified for the component (`Double` in our case). Once we have the user input we convert it to kelvin degrees and we use the resulting value to set the converted input for our custom component (using method `setConvertedInput(T convertedInput)`).

Method `onBeforeRender()` is responsible for synchronizing the model of the inner textfield with the model of our custom component. To do this we retrieve the model object of the custom component with method `getDefaultModelObject()`, then we convert it to the temperature scale adopted by user and finally we use this value to set the model object of the text field.

10.9 Stateless form

In chapter 6 we have seen how Wicket pages can be divided into two categories: *stateful* and *stateless*. Pages that are stateless doesn't need to be stored into user session and they should be used instead of stateful pages when we don't need to save any user data into session (for example in the public area of a site).

Besides saving resources on server-side, stateless pages can be also adopted to improve user experience and to avoid security weaknesses. A typical situation where a stateless page can bring these benefits is when we have to implement a login page.

For this kind of page we might encounter two potential problems if we chose to use a stateful page. The first problem occurs when user tries to login without a valid session assigned to him. This could happen if user leaves the login page opened for a period of time bigger than session timeout and then he decides to enter into the site. Under these conditions the user will be redirected to a 'Page expired' error page, which is not exactly a good thing for user experience.

The second problem occurs when a malicious user or a web crawler program starts to attempt to login to our application, generating a huge number of page versions and consequently increasing the size of user session.

To avoid these kinds of problems we should build a stateless login page which does not depend on user session. Wicket provides a special version of `Form` component called `StatelessForm` which is stateless by default (i.e its method `getStatelessHint()` returns true), hence it's an ideal solution when we want to build a stateless page with a form. A possible implementation of our login form is the following (example project *StatelessLoginForm*):

Html:

```
<html>
   <head>
      <meta charset="utf-8" />
   </head>
   <body>
      <div>Session is <b wicket:id="sessionType"></b></div>
      <br/>
      <div>Type 'user' as correct credentials</div>
      <form wicket:id="form">
         <fieldset>
            Username: <input type="text" wicket:id="username"/> <br/>
            Password: <input type="password" wicket:id="password"/><br/>
            <input type="submit"/>
         </fieldset>
      </form>
      <br/>
      <div wicket:id="feedbackPanel"></div>
   </body>
</html>
```

Java code:

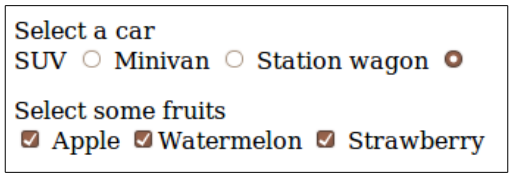
```
public class HomePage extends WebPage {
    private Label sessionType;
    private String password;
    private String username;

    public HomePage(final PageParameters parameters) {
       StatelessForm form = new StatelessForm("form"){
        @Override
        protected void onSubmit() {
           //sign in if username and password are “user”
           if("user".equals(username) && username.equals(password))
              info("Username and password are correct!");
           else
              error("Wrong username or password");
        }
      };

      form.add(new PasswordTextField("password"));
      form.add(new TextField("username"));

      add(form.setDefaultModel(new CompoundPropertyModel(this)));

      add(sessionType = new Label("sessionType", Model.of("")));
```

```
      add(new FeedbackPanel("feedbackPanel"));
    }

    @Override
    protected void onBeforeRender() {
       super.onBeforeRender();

       if(getSession().isTemporary())
          sessionType.setDefaultModelObject("temporary");
       else
          sessionType.setDefaultModelObject("permanent");
    }
}
```

Label `sessionType` shows if current session is temporary or not and is set inside `onBeforeRender()`: if our page is really stateless the session will be always temporary. We have also inserted a feedback panel in the home page that shows if the credentials are correct. This was done to make the example form more interactive.

10.10 Working with radio buttons and checkboxes

In this paragraph we will see which components can be used to handle HTML radio buttons and checkboxes. Both these input elements are usually grouped together to display a list of possible choices:



A check box can be also used as single component to set a boolean property. For this purpose Wicket provides component `org.apache.wicket.markup.html.form.CheckBox` which must be attached to tag `<input type="checkbox".../>`. In the next example (project *SingleCheckBox*) we will consider a form similar to the one used in paragraph 9.5 to edit a `Person` object, but with an additional checkbox to let user decide if she wants to subscribe to our mailing list or not. The form uses the following bean as backing object:

```
public class RegistrationInfo implements Serializable {

       private String name;
       private String surname;
       private String address;
       private String email;
       private boolean subscribeList;

       /*Getters and setters*/
}
```

The markup and the code for this example are the following:

Form's markup:

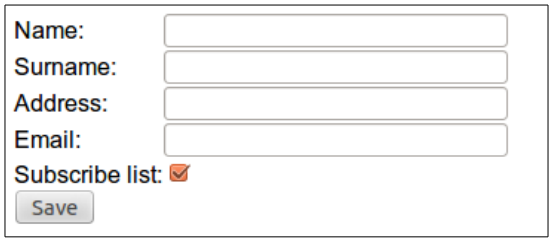
```
<form wicket:id="form">
        <div style="display: table;">
                <div style="display: table-row;">
                        <div style="display: table-cell;">Name: </div>
                        <div style="display: table-cell;">
                                <input type="text" wicket:id="name"/>
                        </div>
                </div>
                <div style="display: table-row;">
                        <div style="display: table-cell;">Surname: </div>
                        <div style="display: table-cell;">
                                <input type="text" wicket:id="surname"/>
                        </div>
                </div>
                <div style="display: table-row;">
                        <div style="display: table-cell;">Address: </div>
                        <div style="display: table-cell;">
                                <input type="text" wicket:id="address"/>
                        </div>
                </div>
                <div style="display: table-row;">
                        <div style="display: table-cell;">Email: </div>
                        <div style="display: table-cell;">
                                <input type="text" wicket:id="email"/>
                        </div>
                </div>
                <div style="display: table-row;">
                        <div style="display: table-cell;">Subscribe list:</div>
                        <div style="display: table-cell;">
                                <input type="checkbox" wicket:id="subscribeList"/>
                        </div>
                </div>
        </div>
        <input type="submit" value="Save"/>
</form>
```

Page constructor:

```
public HomePage(final PageParameters parameters) {
        RegistrationInfo registrtionInfo = new RegistrationInfo();
        registrtionInfo.setSubscribeList(true);

        Form form = new Form("form",
                        new CompoundPropertyModel<RegistrationInfo>(registrtionInfo));

        form.add(new TextField("name"));
        form.add(new TextField("surname"));
        form.add(new TextField("address"));
        form.add(new TextField("email"));
        form.add(new CheckBox("subscribeList"));

        add(form);
```

```
}
```

Please note that the checkbox will be initially selected because we have set to true the subscribe flag during model object creation (with instruction `registrtionInfo.setSubscribeList(true)`):



10.10.1 Working with grouped checkboxes

When we need to display a given number of options with checkboxes, we can use component `org.apache.wicket.markup.html.form.CheckBoxMultipleChoice`. For example, If our options are a list of strings, we can display them in this way:

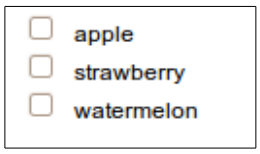
Html:

```
<div wicket:id="checkGroup">
        <input type="checkbox"/>It will be replaced by the actual checkboxes...
</div>
```

Java code:

```
List<String> fruits = Arrays.asList("apple", "strawberry", "watermelon");

form.add(new CheckBoxMultipleChoice("checkGroup", new ListModel<String>(new
                                        ArrayList<String>()), fruits));
```

Screenshot of generated page:



The component can be attached to a `<div>` tag or to a `<span>` tag. No specific content is required for this tag as it will be populated with the actual checkboxes. Since this component allows multiple selection, its model object is a list. In the example above we have used model `org.apache.wicket.model.util.ListModel` which is specifically designed to wrap a `List` object.

By default `CheckBoxMultipleChoice` inserts a `<br/>` tag as suffix after each option. We can configure both the suffix and the prefix used by the component with methods `setPrefix` and `setSuffix`.

When our options are more complex objects than simple strings, we can render them using an

`IChoiceRender`, as we did for `DropDownChoice` in paragraph 9.4:

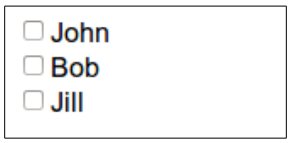
Html:

```
<div wicket:id="checkGroup">
        <input type="checkbox"/>It will be replaced by actual checkboxes...
</div>
```

Java code:

```
Person john = new Person("John", "Smith");
Person bob = new Person("Bob", "Smith");
Person jill = new Person("Jill", "Smith");
List<Person> theSmiths = Arrays.asList(john, bob, jill);
ChoiceRenderer render = new ChoiceRenderer("name");
form.add(new CheckBoxMultipleChoice("checkGroup", new ListModel<String>(ArrayList<String>()),
                                    theSmiths, render));
```

Screenshot of generated page:



10.10.2 How to implement a “Select all” checkbox

A nice feature we can offer to users when we have a group of checkboxes is a “special” checkbox which selects/unselects all the other options of the group:

Wicket comes with a couple of utility components that make easy to implement such a feature. They are classes `CheckboxMultipleChoiceSelector` and `CheckBoxSelector`, both inside package `org.apache.wicket.markup.html.form`. The difference between these two components is that the first works with an instance of `CheckBoxMultipleChoice` while the second takes in input a list of `CheckBox` objects:

CheckboxMultipleChoiceSelector usage:

```
CheckBoxMultipleChoice checkGroup;
//checkGroup initialization...
CheckboxMultipleChoiceSelector cbmcs = new CheckboxMultipleChoiceSelector("id", checkGroup);
```

CheckBoxSelector usage:

```
CheckBox checkBox1, checkBox2, checkBox3;
//checks initialization...
CheckBoxSelector cbmcs = new CheckBoxSelector("id", checkBox1, checkBox2, checkBox3);
```

10.10.3 Working with grouped radio buttons

For groups of radio buttons we can use component `org.apache.wicket.markup.html.form.RadioChoice` which works in much the same way as `CheckBoxMultipleChoice`:

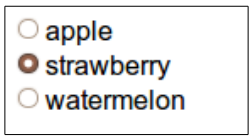
Html:

```
<div wicket:id="radioGroup">
       <input type="radio"/>It will be replaced by actual radio buttons...
</div>
```

Java code:

```
List<String> fruits = Arrays.asList("apple", "strawberry", "watermelon");

form.add(new RadioChoice("radioGroup", Model.of(""), fruits));
```

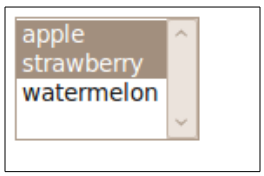
Screenshot of generated page:



Just like `CheckBoxMultipleChoice`, this component provides methods `setPrefix` and `setSuffix` to configure the prefix and suffix for our options and it supports `IChoiceRender` as well. In addition, `RadioChoice` provides method `wantOnSelectionChangedNotifications()` to notify server when the selected option changes (this is the same method seen for `DropDownChoice` in paragraph 9.4).

10.11 Selecting multiple values with ListMultipleChoices and Palette

Checkboxes work well when we have a small amount of options to display, but they quickly become chaotic as the number of options increases. To overcome this limit we can use `<select>` tag switching it to multiple-choice mode with attribute `multiple="multiple"`:



Now user can select multiple options by holding down Ctrl key (or Command key for Mac) and clicking on them.

To work with multiple choice list Wicket provides component `org.apache.wicket.markup.html.form.ListMultipleChoice`:

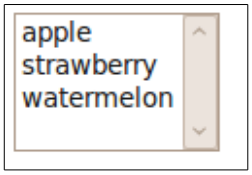
Html:

```
<select wicket:id="fruits">
        <option>choice 1</option>
        <option>choice 2</option>
</select>
```

Java code:

```
List<String> fruits = Arrays.asList("apple", "strawberry", "watermelon");
form.add(new ListMultipleChoice("fruits", new ListModel<String>(new ArrayList<String>()),
                                                                  fruits));
```

Screenshot of generated page:



The component must be bound to a `<select>` tag but the attribute `multiple="multiple"` is not required as it will be automatically added by the component.

The number of visible rows can be set with method `setMaxRows(int maxRows)`.

10.11.1 Component Palette

While multiple choice list solves the problem of handling a big number of multiple choices, it is not much intuitive for end users. That's why desktop GUIs have introduced a more complex component which can be generally referred to as *multi select transfer* component (it doesn't have an actual official name):

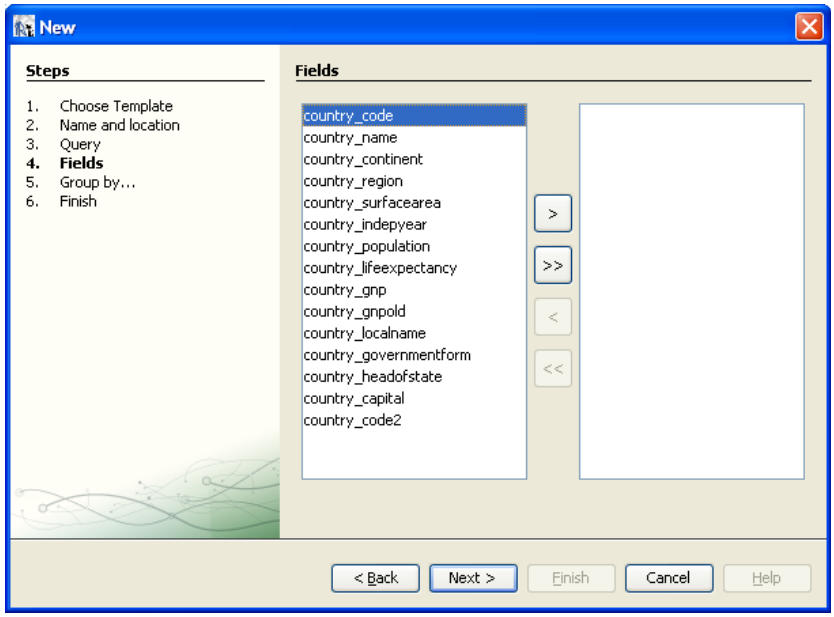


Illustration 10.3: An example of multi select transfer component from Jasper iReport

This kind of component is composed by two multiple-choice lists, one on the left displaying the available options and the other one on the right displaying the selected options. User can move options from a list to another by double clicking on them or using the buttons placed between the two list.

Built-in component `org.apache.wicket.extensions.markup.html.form.palette.Palette` provides an out-of-the-box implementation of a multi select transfer component. It works in a similar way to `ListMultipleChoice`:

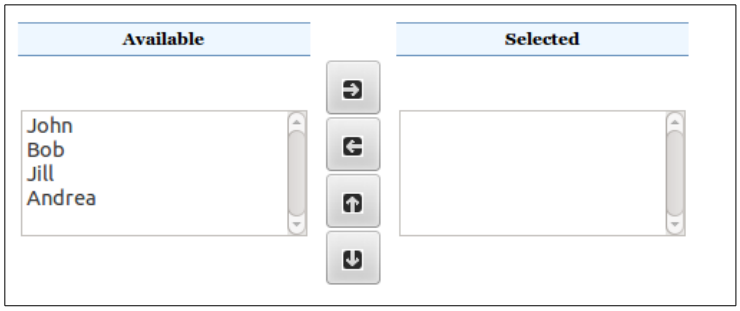
Html:

```
<div wicket:id="palette">
   Select will be replaced by the actual content...
   <select multiple="multiple">
     <option>option1</option>
     <option>option2</option>
     <option>option3</option>
</div>
```

Java code:

```
Person john = new Person("John", "Smith");
Person bob = new Person("Bob", "Smith");
Person jill = new Person("Jill", "Smith");
Person andrea = new Person("Andrea", "Smith");

List<Person> theSmiths = Arrays.asList(john, bob, jill, andrea);
ChoiceRenderer render = new ChoiceRenderer("name");

form.add(new Palette("palette", Model.of(new ArrayList<String>()), new ListModel<String>
(theSmiths), render, 5, true));
```

Screenshot of generated page:



The last two parameters of `Palette`'s constructor (an integer value and a boolean value) are, respectively, the number of visible rows for the two lists and a flag to choose if we want to display the two optional buttons which move selected options up and down. The descriptions of the two lists ("Available" and "Selected") can be customized providing two resources with keys `palette.available` and `palette.selected`.

The markup of this component uses a number of CSS classes which can be extended/overriden to customize the style of the component. We can find these classes and see which tags they decorate in the default markup file of the component:

```
<table cellspacing="0" cellpadding="2" class="palette">
<tr>
        <td class="header headerAvailable"><span wicket:id="availableHeader">[available
header]</span></td>
        <td> </td>
        <td class="header headerSelected"><span wicket:id="selectedHeader">[selected
header]</span>
         </td>
</tr>
<tr>
        <td class="pane choices">
                <select wicket:id="choices" class="choicesSelect">[choices]</select>
        </td>
        <td class="buttons">
                <button type="button" wicket:id="addButton" class="button add"><div/>
                </button><br/>
                <button type="button" wicket:id="removeButton" class="button remove"><div/>
                </button><br/>
                <button type="button" wicket:id="moveUpButton" class="button up"><div/>
                </button><br/>
                <button type="button" wicket:id="moveDownButton" class="button down"><div/>
                </button><br/>
        </td>
        <td class="pane selection">
                <select class="selectionSelect" wicket:id="selection">[selection]</select>
        </td>
</tr>
</table>
```

10.12 Summary

Forms are the standard solution to let users interact with our applications. In this chapter we have seen the three steps involved into form processing in Wicket. We have started looking at form validation and

feedback messages generation, then we have seen how Wicket converts input values into Java objects and vice versa.

In the second part of the chapter we have learnt how to build reusable form components and how to implement a stateless form. We have ended the chapter with an overview of the built-in form components needed to handle standard input form elements like checkboxes, radio buttons and multiple selections lists.

11 Displaying multiple items with repeaters

A common task for web applications is to display a set of items. The most typical scenario where we need such kind of visualization is when we have to display some kind of search result. With the old template-based technologies (like JSP) we used to accomplish this task using classic `for` or `while` loops:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
  <%
    for(int i = 12; i<=32; i++) {
      %>
      <div>Hello! I'm index n°<%= %></div>
  <%
    }
  %>
</body>
```

To ease this task Wicket provides a number of special-purpose components called *repeaters* which are designed to use their related markup to display the items of a given set in a more natural and less chaotic way.

In this chapter we will see some of the built-in repeaters that come with Wicket.

11.1 Component RepeatingView

Component `org.apache.wicket.markup.repeater.RepeatingView` is a container which renders its children components using the tag it is bound to. It can contain an arbitrary number of children elements and we can obtain a new valid id for a new child calling its method `newChildId()`. This component is particularly suited when we have to repeat a simple markup fragment, for example when we want to display some items as a HTML list:

Html:

```
<ul>
    <li wicket:id="listItems"></li>
</ul>
```

Java code:

```
RepeatingView listItems = new RepeatingView("listItems");

listItems.add(new Label(listItems.newChildId(), "green");
listItems.add(new Label(listItems.newChildId(), "blue");
listItems.add(new Label(listItems.newChildId(), "red");
```

Generated markup:

```
<ul>
    <li>green</li>
    <li>blue</li>
    <li>red</li>
</ul>
```

As we can see in this example, each child component has been rendered using the parent markup as if it was its own.

11.2 Component ListView

As its name suggests, component `org.apache.wicket.markup.html.list.ListView` is designed to display a given list of objects which can be provided as a standard Java `List` or as a model containing the concrete `List`. `ListView` iterates over the list and creates a child component of class `org.apache.wicket.markup.html.list.ListItem` for every encountered item.

Unlike `RepeatingView` this component is intended to be used with complex markup fragments containing nested components.

To generate its children, `ListView` calls its abstract method `populateItem(ListItem<T> item)` for each item in the list, so we must provide an implementation of this method to tell the component how to create its children components. In the following example we use a `ListView` to display a list of `Person` objects:

Html:

```
...
<body>
        <div id="bd" style="display: table;">
                <div wicket:id="persons" style="display: table-row;">
                        <div style="display: table-cell;"><b>Full name: </b></div>
                        <div wicket:id="fullName" style="display: table-cell;"></div>
                </div>
        </div>
</body>
...
```

Java code (page constructor):

```
public HomePage(final PageParameters parameters) {
   List<Person> persons = Arrays.asList(new Person("John", "Smith"),
                                        new Person("Dan", "Wong"));

   add(new ListView<Person>("persons", persons) {
       @Override
       protected void populateItem(ListItem<Person> item) {
          item.add(new Label("fullName", new PropertyModel(item.getModel(), "fullName")));
       }
   });
}
```

Screenshot of generated page:

Full name: John Smith
Full name: Dan Wang

In this example we have displayed the full name of two `Person`'s instances. The most interesting part of the code is the implementation of method `populateItem` where parameter `item` is the current child component created by `ListView` and its model contains the corresponding element of the list. Please note that inside `populateItem` we must add nested components to `item` object and not directly to the `ListView`.

11.2.1 ListView and Form

By default `ListView` replaces its children components with new instances every time is rendered. Unfortunately this behavior is a problem if `ListView` is inside a form and it contains form components. The problem is caused by the fact that children components are replaced by new ones before form is rendered, hence they can't keep their input value if validation fails and, furthermore, their feedback messages can not be displayed.

To avoid this kind of problem we can force `ListView` to reuse its children components using its method `setReuseItems` and passing `true` as parameter. If for any reason we need to refresh children components after we have invoked `setReuseItems(true)`, we can use `MarkupContainer`'s method `removeAll()` to force `ListView` to rebuild them.

11.3 Component RefreshingView

Component `org.apache.wicket.markup.repeater.RefreshingView` is a subclass of `RepeatingView` that comes with a customizable rendering strategy for its children components.

`RefreshingView` defines abstract methods `populateItem(Item)` and `getItemModels()`. The first method is similar to the namesake method seen for `ListView`, but it takes in input an instance of class `org.apache.wicket.markup.repeater.Item` which is a subclass of `ListItem`. `RefreshingView` is designed to display a collection of models containing the actual items. An iterator over these models is returned by the other abstract method `getItemModels`.

The following code is a version of the previous example that uses `RefreshingView` in place of `ListView`:

Html:

```
...
<body>
        <div id="bd" style="display: table;">
                <div wicket:id="persons" style="display: table-row;">
```

```
                        <div style="display: table-cell;"><b>Full name: </b></div>
                        <div wicket:id="fullName" style="display: table-cell;"></div>
                </div>
        </div>
</body>
...
```

Java code (page constructor):

```
public HomePage(final PageParameters parameters) {
   //define the list of models to use
    final List<IModel<Person>> persons = new ArrayList<IModel<Person>>();

    persons.add(Model.of(new Person("John", "Smith"));
    persons.add(Model.of(new Person("Dan", "Wong"));

   add(new RefreshingView<Person>("persons") {
       @Override
       protected void populateItem(Item<Person> item) {
          item.add(new Label("fullName", new PropertyModel(item.getModel(), "fullName")));
       }

       @Override
       protected Iterator<IModel<Person>> getItemModels() {
          return persons.iterator();
       }
   });
}
```

11.3.1 Item reuse strategy

By default, just like `ListView`, `RefreshingView` replaces its children with new instances every time is rendered. The strategy that decides if and how children components must be refreshed is returned by method `getItemReuseStrategy`. This strategy is an implementation of interface `IItemReuseStrategy`. The default implementation used by `RefreshingView` is class `DefaultItemReuseStrategy` but Wicket provides also strategy `ReuseIfModelsEqualStrategy` which reuses an item if its model has been returned by the iterator obtained with method `getItemModels`.

To set a custom strategy we must use method `setItemReuseStrategy`.

11.4 Pageable repeaters

Wicket offers a number of components that should be used when we have to display a big number of items (for example the results of a `select` SQL query).

All these components implements interface `org.apache.wicket.markup.html.navigation.paging.IPageable` and use interface `IDataProvider` (placed in package `org.apache.wicket.markup.repeater.data`) as *data source*. This interface is designed to support *data paging*. We will see an example of data paging later in paragraph 11.4.2.

The methods defined by `IDataProvider` are the following:

- **iterator(long first, long count)**: returns an iterator over a subset of the entire dataset. The subset starts from the item at position `first` and includes all the next `count` items (i.e. it's the closed interval [`first`,`first+count`]).
- **size()**: gets the size of the entire dataset.
- **model(T object)**: this method is used to wrap an item returned by the iterator with a model. This

can be necessary if, for example, we need to wrap items with a detachable model to prevent them from being serialized.

Wicket already provides implementations of `IDataProvider` to work with a `List` as data source (`ListDataProvider`) and to support data sorting (`SortableDataProvider`).

11.4.1 Component DataView

Class `org.apache.wicket.markup.repeater.data.DataView` is the simplest pageable repeater shipped with Wicket. `DataView` comes with abstract method `populateItem(Item)` that must be implemented to configure children components. In the following example we use a `DataView` to display a list of `Person` objects in a HTML table:

Html:

```
<table>
        <tr>
           <th>Name</th><th>Surename</th><th>Address</th><th>Email</th>
        </tr>
        <tr wicket:id="rows">
           <td wicket:id="dataRow"></td>
        </tr>
</table>
```

Java code:

```
//method loadPersons is defined elsewhere
List<Person> persons = loadPersons();
ListDataProvider<Person> listDataProvider = new ListDataProvider<Person>(persons);

DataView<Person> dataView = new DataView<Person>("row", listDataProvider) {

  @Override
  protected void populateItem(Item<Person> item) {
    Person person = item.getModelObject();
    RepeatingView repeatingView = new RepeatingView("dataRow");

    repeatingView.add(new Label(repeatingView.newChildId(), person.getName()));
    repeatingView.add(new Label(repeatingView.newChildId(), person.getSurename()));
    repeatingView.add(new Label(repeatingView.newChildId(), person.getAddress()));
    repeatingView.add(new Label(repeatingView.newChildId(), person.getEmail()));
    item.add(repeatingView);
  }
};
add(dataView);
```

Please note that in the code above we have used also a `RepeatingView` component to populate the rows of the table.

In the next paragraph we will see a similar example that adds support for data paging.

11.4.2 Data paging

To enable data paging on a pageable repeaters, we must first set the number of items to display per page with method `setItemsPerPage(long items)`. Then, we must attach the repeater to panel `PagingNavigator` (placed in package `org.apache.wicket.markup.html.navigation`

`.paging`) which is responsible for rendering a navigation bar containing the links illustrated in the following picture:

<< < 1 2 3 4 5 6 7 8 9 10 > >>

Project *PageDataViewExample* mixes a `DataView` component with a `PagingNavigator` to display the list of all countries of the world sorted by alphabetical order[37]. Here is the initialization code of the project home page:

Html:

```
<table>
  <tr>
    <th>ISO 3166-1</th><th>Name</th><th>Long name</th><th>Capital</th><th>Population</th>
  </tr>
  <tr wicket:id="rows">
    <td wicket:id="dataRow"></td>
  </tr>
</table>
```

Java code:

```
public HomePage(final PageParameters parameters) {
  super(parameters);
  //method loadCountriesFromCsv is defined elsewhere in the class.
  //It reads countries data from a csv file and returns each row as an array of Strings.
  List<String[]> countries = loadCountriesFromCsv();
  ListDataProvider<String[]> listDataProvider = new ListDataProvider<String[]>(countries);

  DataView<String[]> dataView = new DataView<String[]>("rows", listDataProvider) {
    @Override
    protected void populateItem(Item<String[]> item) {
      String[] countriesArr = item.getModelObject();
      RepeatingView repeatingView = new RepeatingView("dataRow");

      for (int i = 0; i < countriesArr.length; i++){
         repeatingView.add(new Label(repeatingView.newChildId(), countriesArr[i]));
      }
      item.add(repeatingView);
    }
  };

  dataView.setItemsPerPage(15);

  add(dataView);
  add(new PagingNavigator("pagingNavigator", dataView));
}
```

The data of a single country (ISO code, name, long name, capital and population) are handled with an array of strings. The usage of `PagingNavigator` it's quite straightforward as we need to simply pass the pageable repeater to its constructor.

To explore the other pageable repeaters shipped with Wicket you can visit the page at

37 *The list of countries is read from a csv file downloaded from* http://opengeocode.org/download/cow.php

http://www.wicket-library.com/wicket-examples/repeater/ where you can find live examples of these components.

Note

Wicket provides also component `PageableListView` which is a sublcass of `ListView` that implements interface `IPageable`, hence it can be considered a pageable repeaters even if it doesn't use interface `IDataProvider` as data source.

11.5 Summary

In this chapter we have explored the built-in set of components called *repeaters* which are designed to repeat their own markup in output to display a set of items. We have started with component `RepeatingView` which can be used to repeat a simple markup fragment.

Then, we have seen components `ListView` and `RefreshingView` which should be used when the markup to repeat contains nested components to populate.

Finally, we have discussed those repeaters that support data paging and that are called *pageable* repeaters. We ended the chapter looking at an example where a pageable repeater is used with panel `PagingNavigator` to make its dataset navigable by user.

12 Internationalization with Wicket

In chapter 10 we have seen how the topic of localization is involved in the generation of feedback messages and we had a first contact with resource bundles.

In this chapter we will continue to explore the localization support provided by Wicket and we will learn how to build pages and components ready to be localized in different languages.

12.1 Localization

As we have seen in chapter 10, the infrastructure of feedback messages is built on top of Java internationalization (i18n) support, so it should not be surprising that the same infrastructure is used also for localization purpose.

However, while so far we have used only the *<ApplicationClassName>.properties* file to store our custom messages, in this chapter we will see that also pages, components, validators and even Java packages can have their own resource bundles. This allows us to split bundles into multiple files keeping them close to where they are used.

But before diving into the details of internationalization with Wicket, it's worthwhile to quickly review how i18n works under Java, see what classes are involved and how they are integrated into Wicket.

Note

Providing a full description of Java support for i18n is clearly out of the scope of this document. If you need more informations about this topic you can find them in the JavaDocs and in the official i18n tutorial[38].

12.2 Class Locale and ResourceBundle

Class `java.util.Locale` represents a specific country or language of the world and is used in Java to retrieve other locale-dependent informations like numeric and date formats, the currency in use in a country and so on. Such kind of informations are accessed through special entities called *resource bundles* which are implemented by class `java.util.ResourceBundle`.

Every resource bundle is identified by a full name which is built using four parameters: a *base name* (which is required), a *language code*, a *country code* and a *variant* (which are all optional). These three optional parameters are provided by an instance of `Locale` with its three corresponding getter methods: `getLanguage()`, `getCountry()` and `getVariant()`. Parameter *language code* is a lowercase ISO 639 2-letter code (like `zh` for Chinese, `de` for German and so on) while *country code* is an uppercase ISO 3166 2-letter code (like `CN` for China, `DE` for Germany and so on).

The final full name will have the following structure (NOTE: tokens inside squared brackets are optional):

```
<base name>[_<language code>[_<COUNTRY_CODE>[_<variant code>]]]
```

For example a bundle with `MyBundle` as base name and localized for Mandarin Chinese (language code `zh`, country code `CH`, variant `cmn`) will have `MyBundle_zh_CH_cmn` as full name.

38 http://docs.oracle.com/javase/tutorial/i18n/index.html

A base name can be a fully qualified class name, meaning that it can include a package name before the actual base name. The specified package will be the container of the given bundle. For example if we use `org.foo.MyBundle` as base name, the bundle named `MyBundle` will be searched inside package `org.foo`. The actual base name (`MyBundle` in our example) will be used to build the full name of the bundle following the same rules seen above.

`ResourceBundle` is an abstract factory class, hence it exposes a number of factory methods named `getBundle` to load a concrete bundle. Without going into too much details we can say that a bundle corresponds to a file in the classpath. To find a file for a given bundle, `getBundle` needs first to generate an ordered list of *candidate bundle names*. These names are the set of all possible full names for a given bundle. For example if we have `org.foo.MyBundle` as base name and the current locale is the one seen before for Mandarin Chinese, the candidate names will be:

1. `org.foo.MyBundle_zh_CH_cmn`
2. `org.foo.MyBundle_zh_CH`
3. `org.foo.MyBundle_zh`
4. `org.foo.MyBundle`

The list of these candidate names is generated starting from the most specific one and subtracting an optional parameter at each step. The last name of the list corresponds to the *default* resource bundle which is the most general name and is equal to the base name.

Once that `getBundle` has generated the list of candidate names, it will iterate over them to find the first one for which is possible to load a class or a properties file. The class must be a subclass of `ResourceBundle` having as class name the full name used in the current iteration. If such a class is not found, `getBundle` will try to locate a `properties` file having a file name equals to the current full name (Java will automatically append extension `.properties` to the full name).

For example given the resource bundle of the previous example, Java will search first for class `org.foo.MyBundle_zh_CH_cmn` and then for file `MyBundle_zh_CH_cmn.properties` inside package `org.foo`. If no file is found for any of the candidate names, a `MissingResourceException` will be thrown.

Bundles contains local-dependent string resources identified by a key that is unique in the given bundle. So once we have obtained a valid bundle we can access these objects with method `getString (String key)`.

As we have seen before working with feedback messages, in Wicket most of the times we will work with properties files rather than with bundle classes. In chapter 10 we used a properties file having as base name the class name of the application class and without any information about the locale. This file is the *default* resource bundle for a Wicket application. In paragraph 12.4 we will explore the algorithm used in Wicket to locate the available bundles for a given component. Once we have learnt how to leverage this algorithm, we will be able to split our bundles into more files organized in a logical hierarchy.

12.3 Localization in Wicket

A component can get the current locale in use calling its method `getLocale()`. By default this method will be recursively called on component's parent containers until one of them returns a valid locale. If no one of them returns a locale, this method will get the one associated with the current user session. This locale is automatically generated by Wicket in accordance with the language settings of the browser.

Developers can change the locale of the current session with `Session`'s method `setLocale (Locale locale)`:

```
Session.get().setLocale(locale)
```

12.3.1 *Style* and *variation* parameters for bundles

In addition to locale's informations, Wicket supports two further parameters to identify a resource bundle: *style* and *variation*. Parameter *style* is a string value and is defined at session-level. To set/get the style for the current session we can use the corresponding setter and getter of class `Session`:

```
Session.get().setStyle("myStyle");
Session.get().getStyle();
```

If set, style's value contributes to the final full name of the bundle and it is placed between the base name and the locale's informations:

```
<base name>[_style][_<language code>[_<COUNTRY_CODE>[_<variant code>]]]
```

Wicket gives the priority to candidate names containing the style information (if available).

The other parameter we can use for localization is *variation*. Just like *style* also *variation* is a string value, but it is defined at component-level. The value of variation is returned by `Component`'s method `getVariation()`. By default this method returns the variation of the parent component or a `null` value if a component hasn't a parent (i.e. it's a page). If we want to customize this parameter we must overwrite method `getVariation` and make it return the desired value.

Variation's value contributes to the final full name of the bundle and is placed before style parameter:

```
<base name>[_variation][_style][_<language code>[_<COUNTRY_CODE>[_<variant code>]]]
```

12.3.2 Using XML files as resource bundles

Java uses the standard character set ISO 8859-1[39] to encode text files like properties files. Unfortunately ISO 8859-1 does not support most of the extra-European languages like Chinese or Japanese. The only way to use properties files with such languages is to use escaped Unicode characters[40], but this leads to not human-readable files. For example if we wanted to write the word 'website' in simplified Chinese (the ideograms are 网站) we should write the Unicode characters `\u7F51\u7AD9`.

That's why starting from version 1.5, Java introduced the support for XML files as resource bundles. XML files are generally encoded with character sets UTF-8 or UTF-16 which support every symbol of the Unicode standard.

In order to be a valid resource bundle the XML file must conform to the DTD available at http://java.sun.com/dtd/properties.dtd.

Here is an example of XML resource bundle taken from project *LocalizedGreetings* (file `WicketApplication_zh.properties.xml`) containing the translation in simplified Chinese of the greeting message “Welcome to the website!”:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
        <entry key="greetingMessage">欢迎光临本网站！</entry>
</properties>
```

To use XML bundles in Wicket we don't need to put in place any additional configuration. The only rule we have to respect with these files is to use `properties.xml` as extension while their base name

39 http://en.wikipedia.org/wiki/ISO/IEC_8859-1
40 http://en.wikipedia.org/wiki/List_of_Unicode_characters

follows the same rules seen so far for bundle names.

12.3.3 Reading bundles from code

Class `Component` makes reading bundles very easy with method `getString(String key)`. This method searches for a resource with the given key looking into the resource bundles visited by the lookup algorithm illustrated in paragraph 12.4.

For example if we have a greeting message with key `greetingMessage` in our application's resource bundle, we can read it from our component code with this instruction:

```
getString("greetingMessage");
```

12.3.4 Localization of bundles in Wicket.

In chapter 10 we have used as resource bundle the properties file placed next to our application class. This file is the default resource bundle for the entire application and it is used by the lookup algorithm if it doesn't find any better match for a given component and locale.

If we want to provide localized versions of this file we must simply follow the rules of Java i18n and put our translated resources into another properties file with a name corresponding to the desired locale.

For example project *LocalizedGreetings* comes with the default application's properties file (`WicketApplication.properties`) containing a greeting message:

```
greetingMessage=Welcome to the site!
```

Along with this file we can also find a bundle for German (`WicketApplication_de.properties`) and another one in XML format for simplified Chinese (`WicketApplication_zh.properties.xml`).

The example project consists of a single page (`HomePage.java`) displaying the greeting message. The current locale can be changed with a drop-down list and the possible options are English (the default one), German and simplified Chinese:

Illustration 12.1: The home page of project LocalizedGreetings localized in German

The label displaying the greeting message has a custom read-only model which returns the message with method `getString`. The initialization code for this label is this:

```
AbstractReadOnlyModel<String> model = new AbstractReadOnlyModel<String>() {
        @Override
        public String getObject() {
                return getString("greetingMessage");
        }
};

add(new Label("greetingMessage", model));
```

Class `org.apache.wicket.model.AbstractReadOnlyModel` is a convenience class for implementing read-only models. In this project we have implemented a custom read-only model for illustrative purposes only because Wicket already provides built-in models for the same task. We will see them in paragraph 12.6.

The rest of the code of the home page builds the stateless form and the drop-down menu used to change the locale.

```
List<Locale> locales = Arrays.asList(Locale.ENGLISH, Locale.CHINESE, Locale.GERMAN);
final DropDownChoice<Locale> changeLocale =
              new DropDownChoice<Locale>("changeLocale", new Model<Locale>(), locales);

StatelessForm form = new StatelessForm("form"){
       @Override
       protected void onSubmit() {
              Session.get().setLocale(changeLocale.getModelObject());
       }
};

setStatelessHint(true);
add(form.add(changeLocale))
```

12.3.5 Localization of markup files

Although resource bundles exist to extract local-dependent elements from our code and from UI components, in Wicket we can decide to provide different markup files for different locale settings. Just like standard markup files, by default localized markup files must be placed next to component's class and their file name must contain the locale's informations. In the following picture, `CustomPanel` comes with a standard (or default) markup file and with another one localized for German:

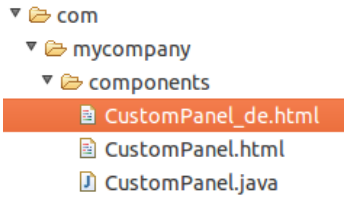


Illustration 12.2: Component with markup localized in German

When the current locale corresponds to German country (language code `de`), markup file CustomPanel_de.html will be used in place of the default one.

12.3.6 Reading bundles with tag *<wicket:message>*

String resources can be also retrieved directly from markup code using tag `<wicket:message>`. The key of the desired resource is specified with attribute `key`:

```
<wicket:message key="greetingMessage">message goes here</wicket:message>
```

`wicket:message` can be adopted also to localize the attributes of a tag. The name of the attribute and the resource key are expressed as a colon-separated value. In the following markup the content of attribute `value` will be replaced with the localized resource having '`key4value`' as key:

```
<input type="submit" value="Preview value" wicket:message="value:key4value"/>
```

If we want to specify multiple attributes at once, we can separate them with a coma:

```
<input type="submit" value="Preview value" wicket:message="value:key4value,title:key4title"/>
```

12.4 Bundles lookup algorithm

As we hinted at the beginning of this chapter, by default Wicket provides a very flexible algorithm to locate the resource bundles available for a given component. In this paragraph we will learn how this default lookup algorithm works and which options it offers to manage our bundle files.

12.4.1 Localizing pages and panels

Similarly to application class, also component classes can have their own bundle files having as base name the class name of the related component and placed in the same package. So for example if class `CustomPanel` is a custom panel we created, we can provide it with a default bundle file called `CustomPanel.properties` containing the textual resources used by this panel. This rule applies to page classes as well:

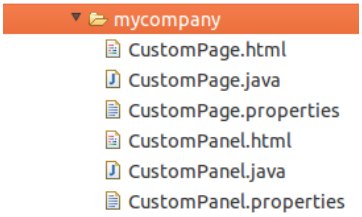


Illustration 12.3: Page and panel with their own bundle

One fundamental thing to keep in mind when we work with these kinds of bundles is that the lookup algorithm gives priority to the bundles of the containers of the component that is requesting a localized resource. The more a container is higher in the hierarchy, the bigger is its priority over the other components. This mechanism was made to allow containers to overwrite resources used by children components. As a consequence the values inside the resource bundle of a page will have the priority over the other values with the same key defined in the bundles of children components.

To better grasp this concept let's consider the component hierarchy depicted in the following picture:

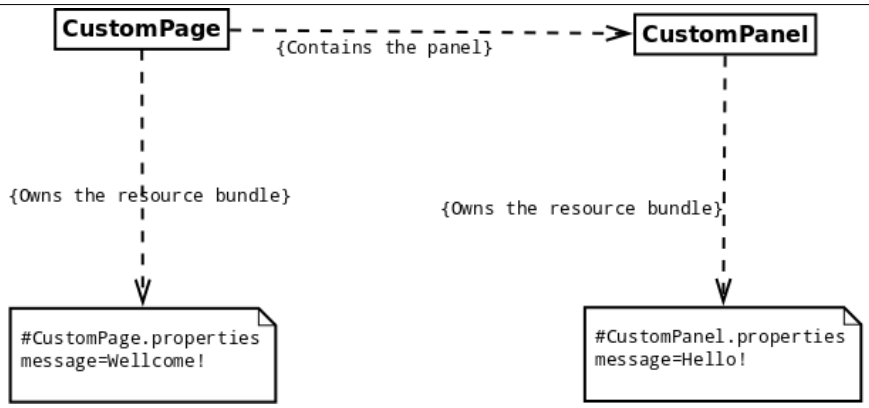


Illustration 12.4: The bundle of CustomPanel is overwritten by the one of CustomPage.

If `CustomPanel` tries to retrieve the string resource having `'message'` as key, it will get the value `'Wellcome!'` and not the one defined inside its own bundle file.

The default message-lookup algorithm is not limited to component hierarchy but it also includes the class hierarchy of every component visited in the search strategy described so far. This makes bundle files *inheritable*, just like markup files. When the hierarchy of a container component is explored, any ancestor has the priority over children components. Consider for example the hierarchy in the following picture:

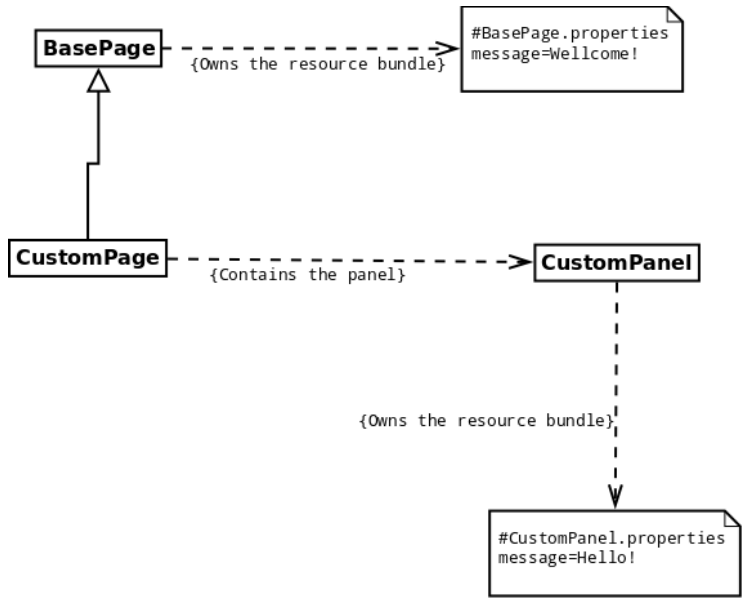


Illustration 12.5: The bundle of CustomPanel is overwritten by the one of page class BasePage.

Similarly to the previous example, the bundle owned by `CustomPanel` is overwritten by the bundle of page class `BasePage` (which has been inherited by `CustomPage`).

12.4.2 Component-specific resources

In order to make a resource specific for a given child component, we can prefix the message key with the id of the desired component. Consider for example the following code and bundle of a generic page:

Page code:

```
add(new Label("label",new ResourceModel("labelValue")));
add(new Label("anotherLabel",new ResourceModel("labelValue")));
```

Page bundle:

```
labelValue=Default value
anotherLabel.labelValue=Value for anotherLabel
```

Label with id `anotherLabel` will display the value `'Value for anotherLabel'` while label `label` will display `'Default value'`. In a similar fashion, parent containers can specify a resource for a nested child component prepending also its relative path (the path is dot-separated):

Page code:

```
Form form = new Form("form");
form.add(new Label("anotherLabel",new ResourceModel("labelValue")));
add(form);
```

Page bundle:

```
labelValue=Default value
anotherLabel.labelValue=Value for anotherLabel
form.anotherLabel.labelValue=Value for anotherLabel inside form
```

With the code and the bundle above, the label inside the form will display the value `'Value for anotherLabel inside form'`.

12.4.3 Package bundles

If no one of the previous steps can find a resource for the given key, the algorithm will look for *package bundles*. These bundles have `package` as base name and they can be placed in one of the package of our application:

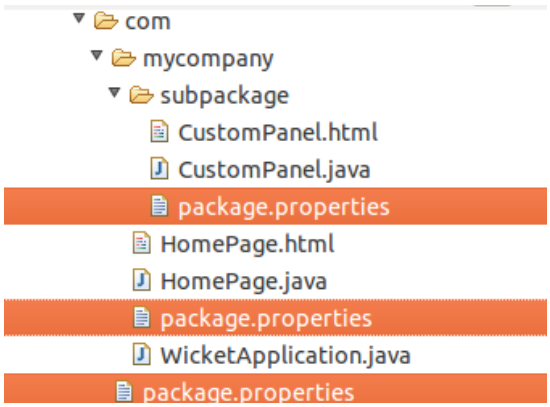


Illustration 12.6: Package bundles in each package of the application

Packages are traversed starting from the one containing the component requesting for a resource and going up to the root package.

12.4.4 Bundles for feedback messages

The algorithm described so far applies to feedback messages as well. In case of validation errors, the component that has caused the error will be considered as the component which the string resource is relative to.

Furthermore, just like application class and components, validators can have their own bundles placed next to their class and having as base name their class name. This allows us to distribute validators along with the messages they use to report errors:

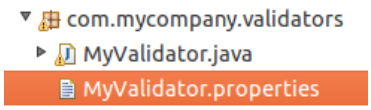


Illustration 12.7: Validator with its own bundle

Validator's resource bundles have the lowest priority in the lookup algorithm. They can be overwritten by resource bundles of components, packages and application class.

12.4.5 Extending the default lookup algorithm

Wicket implements the default lookup algorithm using the *strategy pattern*[41]. The concrete strategies are abstracted with the interface `org.apache.wicket.resource.loader.IStringResourceLoader`. By default Wicket uses the following implementations of `IStringResourceLoader` (sorted by execution order):

1. **ComponentStringResourceLoader**: implements most of the default algorithm. It searches for a given resource across bundles from the container hierarchy, from class hierarchy and from the given component.
2. **PackageStringResourceLoader**: searches into package bundles.
3. **ClassStringResourceLoader**: searches into bundles of a given class. By default the target class is the application class.
4. **ValidatorStringResourceLoader**: searches for resources into validator's bundles. A list of validators is provided by the form component that failed validation.
5. **InitializerStringResourceLoader**: this resource allows internationalization to interact with the initialization mechanism of the framework that will be illustrated in paragraph 15.4.

Developer can customize lookup algorithm removing default resource loaders or adding custom implementations to the list of the resource loaders in use. This task can be accomplished using method `getStringResourceLoaders` of setting interface `org.apache.wicket.settings.IResourceSettings`:

```
@Override
public void init()
{
  super.init();
  //retrieve IResourceSettings and then the list of resource loaders
  List<IStringResourceLoader> resourceLoaders= getResourceSettings().
                                                        getStringResourceLoaders();
  //customize the list...
```

12.5 Localization of component's choices

Components that inherit from `AbstractChoice` (such as `DropDownChoice`, `CheckBoxMultipleChoice` and `RadioChoice`) must override method `localizeDisplayValues` and make it return `true` to localize the values displayed for their choices. By default this method return `false` so values are displayed as they are.

Once localization is activated we can use display values as key for our localized string resources. In project *LocalizedChoicesExample* we have a drop-down list that displays four colors (green, red, blue, and yellow) which are localized in three languages (English, German and Italian). The current locale can be changed with another drop-down menu (in a similar fashion to project *LocalizedGreetings*). The code of the home page and the relative bundles are the following:

Java code:

```
public HomePage(final PageParameters parameters) {
       super(parameters);
```

41 http://en.wikipedia.org/wiki/Strategy_pattern

```
        List<Locale> locales = Arrays.asList(Locale.ENGLISH, Locale.ITALIAN, Locale.GERMAN);
        List<String> colors = Arrays.asList("green", "red", "blue", "yellow");

        final DropDownChoice<Locale> changeLocale = new DropDownChoice<Locale>("changeLocale",
                                                        new Model<Locale>(), locales);

        StatelessForm form = new StatelessForm("form"){
                @Override
                protected void onSubmit() {
                        Session.get().setLocale(changeLocale.getModelObject());
                }
        };

        DropDownChoice<String> selectColor = new DropDownChoice<String>("selectColor", new
                                                         Model<String>(), colors){
                @Override
                protected boolean localizeDisplayValues() {
                        return true;
                }
        };

        form.add(selectColor);
        add(form.add(changeLocale));
    }
```

Default bundle (English):

```
selectColor.null=Select a color
green=Green
red=Red
blue=Blue
yellow=Yellow
```

German bundle:

```
selectColor.null=Wahlen sie eine farbe
green=Grun
red=Rot
blue=Blau
yellow=Gelb
```

Italian bundle:

```
selectColor.null=Scegli un colore
green=Verde
red=Rosso
blue=Blu
yellow=Giallo
```

Along with the localized versions of colors names, in the bundles above we can also find a custom value for the placeholder text (“Chose one ”) used for `null` value. The resource key for this resource is `'null'` or `'<component id>.null'` if we want to make it component-specific.

12.6 Internationalization and Models

Internationalization is another good chance to taste the power of models. Wicket provides two built-in

models to better integrate our components with string resources: they are `ResourceModel` and `StringResourceModel`.

12.6.1 ResourceModel

Model `org.apache.wicket.model.ResourceModel` acts just like the read-only model we have implemented in paragraph 12.3.4. It simply retrieves a string resource corresponding to a given key:

```
//build a ResourceModel for key 'greetingMessage'
new ResourceModel("greetingMessage");
```

We can also specify a default value to use if the requested resource is not found:

```
//build a ResourceModel with a default value
new ResourceModel("notExistingResource", "Resource not found.");
```

12.6.2 StringResourceModel

Model `org.apache.wicket.model.StringResourceModel` allows to work with complex and dynamic string resources containing parameters and property expressions. The basic constructor of this model takes in input a resource key and another model. This further model can be used by both the key and the related resource to specify dynamic values with property expressions.

For example let's say that we are working on an e-commerce site which has a page where users can see an overview of their orders. To handle the state of user's orders we will use the following bean and enum (the code is from project *StringResourceModelExample*):

Bean:

```
public class Order implements Serializable{
	private Date orderDate;
	private ORDER_STATUS status;

	public Order(Date orderDate, ORDER_STATUS status) {
		super();
		this.orderDate = orderDate;
		this.status = status;
	}
	//Getters and setters for private fields
}
```

Enum:

```
public enum ORDER_STATUS {
	PAYMENT_ACCEPTED(0),
	IN_PROGRESS(1),
	SHIPPING(2),
	DELIVERED(3);

	private int code;
	//Getters and setters for private fields
}
```

Now what we want to do in this page is to print a simple label which displays the status of an order and the date on which the order has been submitted. All the informations about the order will be passed to a `StringResourceModel` with a model containing the bean `Order`. The bundle in use contains the

following key/value pairs:

```
orderStatus.0=Your payment submitted on ${orderDate} has been accepted.
orderStatus.1=Your order submitted on ${orderDate} is in progress.
orderStatus.2=Your order submitted on ${orderDate} has been shipped.
orderStatus.3=Your order submitted on ${orderDate} has been delivered.
```

The values above contain a property expression (`${orderDate}`) that will be evaluated on the data object of the model. The same technique can be applied to the resource key in order to load the right resource according to the state of the order:

```
Order order = new Order(new Date(), ORDER_STATUS.IN_PROGRESS);
add(new Label("orderStatus", new StringResourceModel("orderStatus.${status.code}",
                              Model.of(order))));
```

As we can see in the code above also the key contains a property expression (`${status.code}`) which makes its value dynamic. In this way the state of an object (an `Order` in our example) can determinate which resource will be loaded by `StringResourceModel`.

If we don't use properties expressions we can provide a `null` value as model and in this case `StringResourceModel` will behave exactly as a `ResourceModel`.

`StringResourceModel` supports also the same parameter substitution used by standard class `java.text.MessageFormat`. Parameters can be generic objects but if we use a model as parameter, `StringResourceModel` will use the data object inside it as actual value (it will call `getObject` on the model). Parameters are passed to constructor as a vararg argument. Here is an example of usage of parameter substitution:

Java code:

```
PropertyModel propertyModel = new PropertyModel<Order>(order, "orderDate");
//build a string model with two parameters: a property model and an integer value
StringResourceModel srm = new StringResourceModel("orderStatus.delay", null,
                                                                  propertyModel, 3);
```

Bundle:

```
orderStatus.delay=Your order submitted on ${0} has been delayed by {1} days.
```

One further parameter we can specify when we build a `StringResourceModel` is the component that must be used by the lookup algorithm. Normally this parameter is not relevant, but if we need to use a particular bundle owned by a component not considered by the algorithm, we can specify this component as second parameter.

If we pass all possible parameters to `StringResourceModel`'s constructor we obtain something like this:

```
new StringResourceModel("myKey", myComponent, myModel, param1, param2, param3,...);
```

12.7 Summary

Internationalization is a mandatory step if we want to take our applications (and our business!) abroad. Choosing the right strategy to manage our localized resources is fundamental to avoid to make a mess of them. In this chapter we have explored the built-in support for localization provided by Wicket, and we

have learnt which solutions it offers to manage resource bundles.

In the final part of the chapter we have seen how to localize the options displayed by a component (such as `DropDownChoice` or `RadioChoice`) and we also introduced two new models specifically designed to localize our components without introducing in their code any detail about internationalization.

13 Resource management with Wicket

One of the biggest challenge for a web framework is to offer an efficient and consistent mechanism to handle internal resources such as CSS/JavaScript files, picture files, pdf and so on. Resources can be *static* (like an icon used across the site) or *dynamic* (they can be generated on the fly) and they can be made available to users as a download or as a simple URL.

In paragraph 4.6 we have already seen how to add CSS and JavaScript contents to the header section of the page. In the first half of this chapter we will learn a more sophisticated technique that allows us to manage static resources directly from code and “pack” them with our custom components.

Then, in the second part of the chapter we will see how to implement custom resources to enrich our web application with more complex and dynamic functionalities.

13.1 Static vs dynamic resources

In Wicket a *resource* is an entity that can interact with the current request and response and It must implement interface `org.apache.wicket.request.resource.IResource`. This interface defines just method `respond(IResource.Attributes attributes)` where the nested class `IResource.Attributes` provides access to request, response and page parameters objects.

Resources can be *static* or *dynamic*. Static resources don't entail any computational effort to be generated and they generally correspond to a resource on the filesystem. On the contrary dynamic resources are generated on the fly when they are requested, following a specific logic coded inside them.

An example of dynamic resource is the built-in class `CaptchaImageResource` in package `org.apache.wicket.extensions.markup.html.captcha` which generates a captcha image each time is rendered.

As we will see in paragraph 13.6, developers can build custom resources extending base class `org.apache.wicket.request.resource.AbstractResource`.

13.2 Resource references

Most of the times in Wicket we won't directly instantiate a resource but rather we will use a *reference* to it. Resource references are represented by abstract class `org.apache.wicket.request.resource.ResourceReference` which returns a concrete resource with factory method `getResource()`. In this way we can lazy-initialize resources loading them only the first time they are requested.

13.3 Package resources

With HTML we use to include static resources in our pages using tags like `<script>`, `<link>` or `<img>`. This is what we have done so far writing our custom panels and pages. However, when we work with a component-oriented framework like Wicket, this classic approach becomes inadequate because it makes custom components hardly reusable. This happens when a component depends on a big number of resources. In such a case, if somebody wanted to use our custom component in his application, he would be forced to know which resources it depends on and make them available.

To solve this problem Wicket allows us to place static resource files into component package (like we do with markup and properties files) and load them from component code.

These kinds of resources are called *package resources*:

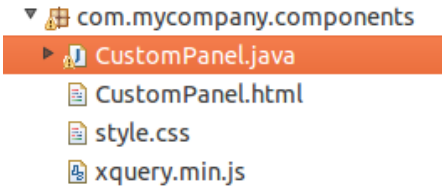


Illustration 13.1: A component with two package resources (a CSS and a JavaScript file).

With package resources custom components become independent and *self-contained* and client code can use them without worrying about their dependencies.

To load package resources Wicket provides class `org.apache.wicket.request.resource.PackageResourceReference`.

To identify a package resource we need to specify a class inside the target package and the name of the desired resource (most of the times this will be a file name).

In the following example taken from project *ImageAsPackageRes*, `CustomPanel` loads a picture file available as package resource and it displays it in a `<img>` tag using the built-in component `org.apache.wicket.markup.html.image.Image`:

Html:

```
<html>
<head>...</head>
<body>
<wicket:panel>
        Package resource image: <img wicket:id="packageResPicture"/>
</wicket:panel>
</body>
</html>
```

Java code:

```
public class CustomPanel extends Panel {

        public CustomPanel(String id) {
                super(id);
                PackageResourceReference resourceReference =
                     new PackageResourceReference(getClass(), "calendar.jpg");
                add(new Image("packageResPicture", resourceReference));
        }
}
```

Wicket will take care of generating a valid URL for file `calendar.jpg`. URLs for package resources have the following structure:

```
<path to application root>/wicket/resource/<fully qualified class name>/<resource file name>
-<ver-<id>>[.<file extension>]
```

In our example the URL for our picture file `calendar.jpg` is the following:

```
./wicket/resource/org.wicketTutorial.CustomPanel/calendar-ver-1297887542000.jpg
```

The first part of the URL is the relative path to the application root. In our example our page is already at the application's root so we have only a single-dotted segment. The next two segments, `wicket` and `resource`, are respectively the *namespace* and the i*dentifier for resources* seen in paragraph 8.6.4.

The fourth segment is the fully qualified name of the class used to locate the resource and it is the *scope* of the package resource. In the last segment of the URL we can find the name of the resource (the file name).

As you can see Wicket has automatically appended to the file name a version identifier (`ver-1297887542000`). When Wicket runs in DEVELOPMENT mode this identifier contains the timestamp in millisecond indicating the last time the resource file was modified. This can be useful when we are developing our application and resource files are frequently modified. Appending the timestamp to the original name we are sure that our browser will use always the last version of the file and not an old, out of date, cached version.

When instead Wicket is running in DEPLOYMENT mode, the version identifier will contain the MD5 digest of the file instead of the timestamp. The digest is computed only the first time the resource is requested. This perfectly makes sense as static resources don't change so often when our application runs into production environment and when this appends the application is redeployed.

Note

Package resources can be localized following the same rules seen for resource bundles and markup files:

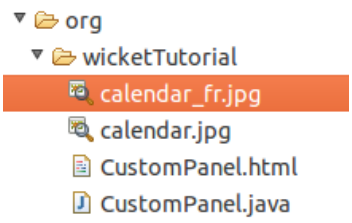


Illustration 13.2: Package resource localized for French language

In the example illustrated in the picture above, if we try to retrieve package resource `calendar.jpg` when the current locale is set to French, the actual file returned will be `calendar_fr.jpg`.

13.3.1 Using package resources with tag *<wicket:link>*

In paragraph 8.3 we have used tag `<wicket:link>` to automatically create links to bookmarkable pages. The same technique can be used also for package resources in order to use them directly from markup file. Let's assume for example that we have a picture file called `icon.png` placed in the same package of the current page. Under these conditions we can display the picture file using the following markup fragment:

```
<wicket:link>
   <img src="icon.png"/>
</wicket:link>
```

In the example above Wicket will populate the attribute `src` with the URL corresponding to the package

resource `icon.png`. `<wicket:link>` supports also tag `<link>` for CSS files and tag `<script>` for JavaScript files.

13.4 Adding resources to page header section

Wicket comes with interface `org.apache.wicket.markup.html.IHeaderContributor` which allows components and behaviors (which will be introduced later in paragraph 15.1) to contribute to the header section of their page. The only method defined in this interface is `renderHead (IHeaderResponse response)` where `IHeaderResponse` is an interface which defines method `render(HeaderItem item)` to write static resources or free-form text into the header section of the page.

Header entries are instances of abstract class `org.apache.wicket.markup.head.HeaderItem`. Wicket provides a set of built-in implementations of this class suited for the most common types of resources. With the exception of `PriorityHeaderItem`, every implementation of `HeaderItem` is an abstract factory class:

- **CssHeaderItem**: represents a CSS resource. Factory methods provided by this class are `forReference` which takes in input a resource reference, `forUrl` which creates an CSS item from a given URL and `forCSS` which takes in input an arbitrary CSS string and an optional id value to identify the resource.
- **JavaScriptHeaderItem**: represents a JavaScript resource. Just like `CssHeaderItem` it provides factory methods `forReference` and `forUrl` along with method `forScript` which takes in input an arbitrary string representing the script and an optional id value to identify the resource.
- **OnDomReadyHeaderItem**: it adds JavaScript code that will be executed after the DOM has been built, but before external files (such as picture, CSS, etc...) have been loaded. The class provides a factory method `forScript` which takes in input an arbitrary string representing the script to execute.
- **OnEventHeaderItem**: the JavaScript code added with this class is executed when a specific JavaScript event is triggered on a given DOM element. The factory method is `forScript(String target, String event, CharSequence javaScript)`, where `target` is the id of a DOM element, `event` is the event that must trigger our code and `javaScript` is the code to execute.
- **OnLoadHeaderItem**: the JavaScript code added with this class is executed after the whole page is loaded, external files included. The factory method is `forScript(CharSequence javaScript)`.
- **PriorityHeaderItem**: it wraps another header item and ensures that it will have the priority over the other items during rendering phase.
- **StringHeaderItem**: with this class we can add an arbitrary text to the header section. Factory method is `forString(CharSequence string)`.

In the following example our custom component loads a CSS file as a package resource (placed in the same package) and it adds it to header section.

Java code:

```
public class MyComponent extends Component{

        @Override
        public void renderHead(IHeaderResponse response) {
                PackageResourceReference cssFile =
                                new PackageResourceReference(this.getClass(), "style.css");
```

```
                CssHeaderItem cssItem = CssHeaderItem.forReference(cssFile);

                response.render(cssItem);
        }
}
```

13.5 Resource dependencies

Class `ResourceReference` allows to specify the resources it depends on overriding method `getDependencies()`. The method returns an iterator over the set of `HeaderItem`s that must be rendered before the resource referenced by `ResourceReference` can be used. This can be really helpful when our resources are JavaScript or CSS libraries that in turn depend on other libraries.

For example we can use this method to ensure that a custom reference to JQueryUI library will find JQuery already loaded in the page:

```
Url jqueyuiUrl = Url.parse("https://ajax.googleapis.com/ajax/libs/jqueryui/" +
                                                                "1.10.2/jquery-ui.min.js");

UrlResourceReference jqueryuiRef = new UrlResourceReference(jqueyuiUrl){
        @Override
        public Iterable<? extends HeaderItem> getDependencies() {
                Application application = Application.get();
                ResourceReference jqueryRef = application.getJavaScriptLibrarySettings().
                                              getJQueryReference();

                return Arrays.asList(JavaScriptHeaderItem.forReference(jqueryRef));
        }
};
```

Please note that in the code above we have built a resource reference using a URL to the desired library instead of a package resource holding the physical file.

The same method `getDependencies()` is defined also for class `HeaderItem`.

13.6 Custom resources

In Wicket the best way to add dynamic functionalities to our application (such as csv export, a pdf generated on the fly, etc...) is implementing a custom resource. In this paragraph as example of custom resource we will build a basic RSS feeds generator which can be used to publish feeds on our site (project *CustomResourceMounting*). Instead of generating a RSS feed by hand we will use Rome[42] framework and its utility classes.

As hinted above in paragraph 13.1, class `AbstractResource` can be used as base class to implement new resources. This class defines abstract method `newResourceResponse` which is invoked when the resource is requested. The following is the code of our RSS feeds generator:

```
public class RSSProducerResource extends AbstractResource {

  @Override
  protected ResourceResponse newResourceResponse(Attributes attributes) {
    ResourceResponse resourceResponse = new ResourceResponse();
    resourceResponse.setContentType("text/xml");
    resourceResponse.setTextEncoding("utf-8");
```

42 http://rometools.org/

```
    resourceResponse.setWriteCallback(new WriteCallback()
    {
      @Override
      public void writeData(Attributes attributes) throws IOException
      {
        OutputStream outputStream = attributes.getResponse().getOutputStream();
        Writer writer = new OutputStreamWriter(outputStream);
        SyndFeedOutput output = new SyndFeedOutput();
            try {
          output.output(getFeed(), writer);
        } catch (FeedException e) {
          throw new WicketRuntimeException("Problems writing feed to response...");
        }
      }
    });

    return resourceResponse;
  }
  // method getFeed()...
}
```

Method `newResourceResponse` returns an instance of `ResourceResponse` representing the response generated by the custom resource. Since RSS feeds are based on XML, in the code above we have set the type of the response to `text/xml` and the text encoding to `utf-8`.

To specify the content that will be returned by our resource we must also provide an implementation of inner class `WriteCallback` which is responsible for writing content data to response's output stream. In our project we used class `SyndFeedOutput` from Rome framework to write our feed to response. Method `getFeed()` is just an utility method that generates a sample RSS feed (which is an instance of interface `com.sun.syndication.feed.synd.SyndFeed`).

Now that we have our custom resource in place, we can use it in the home page of the project. The easiest way to make a resource available to users is to expose it with link component `ResourceLink`:

```
add(new ResourceLink("rssLink", new RSSProducerResource()));
```

In the next paragraphs we will see how to register a resource at application-level and how to mount it to an arbitrary URL.

13.7 Mounting resources

Just like pages also resources can be mounted to a specific path. Class `WebApplication` provides method `mountResource` which is almost identical to `mountPage` seen in paragraph 8.6.1:

```
@Override
public void init() {
  super.init();
  //resource mounted to path /foo/bar
  ResourceReference resourceReference = new ResourceReference("rssProducer"){
     RSSReaderResource rssResource = new RSSReaderResource();
     @Override
      public IResource getResource() {
        return rssResource;
```

```
  }};
  mountResource("/foo/bar", resourceReference);
}
```

With the configuration above (taken from project *CustomResourceMounting*) every request to `/foo/bar` will be served by the custom resource built in the previous paragraph.

Parameter placeholders are supported as well:

```
@Override
public void init() {
  super.init();
  //resource mounted to path /foo with a required indexed parameter
  ResourceReference resourceReference = new ResourceReference("rssProducer"){
     RSSReaderResource rssResource = new RSSReaderResource();
     @Override
      public IResource getResource() {
        return rssResource;
  }};
  mountResource("/bar/${baz}", resourceReference);
}
```

13.8 Shared resources

Resources can be added to a global registry in order to share them at application-level. Shared resources are identified by an application-scoped key and they can be easily retrieved at a later time using reference class `SharedResourceReference`. The global registry can be accessed with `Application`'s method `getSharedResources`. In the following excerpt of code (taken again from project *CustomResourceMounting*) we register an instance of our custom RSS feeds producer as application-shared resource:

```
  //init application's method
  @Override
  public void init(){
    RSSProducerResource rssResource = new RSSProducerResource();
    // ...
    getSharedResources().add("globalRSSProducer", rssResource);
  }
```

Now to use an application-shared resource we can simply retrieve it using class `SharedResourceReference` and providing the key previously used to register the resource:

```
add(new ResourceLink("globalRssLink", new SharedResourceReference("globalRSSProducer")));
```

The URL generated for application shared resources follows the same pattern seen for package resources:

```
./wicket/resource/org.apache.wicket.Application/globalRSSProducer
```

The last segment of the URL is the key of the resource while the previous segment contains the scope of the resource. For application-scoped resources the scope is always the fully qualified name of class `Application`. This should not be surprising since global resources are visible at application level (i.e.

the scope *is* the application).

Note

Package resources are also application-shared resources but they don't need to be explicitly registered.

Note

Remember that we can get the URL of a resource reference using method `urlFor(ResourceReference resourceRef, PageParameters params)` available with both class `RequestCycle` and class `Component`.

13.9 Customizing resource loading.

Wicket loads application's resources delegating this task to a *resource locator* represented by interface `org.apache.wicket.core.util.resource.locator.IResourceStreamLocator`. To retrieve or modify the current resource locator we can use the getter and setter methods defined by setting interface `IResourceSettings`:

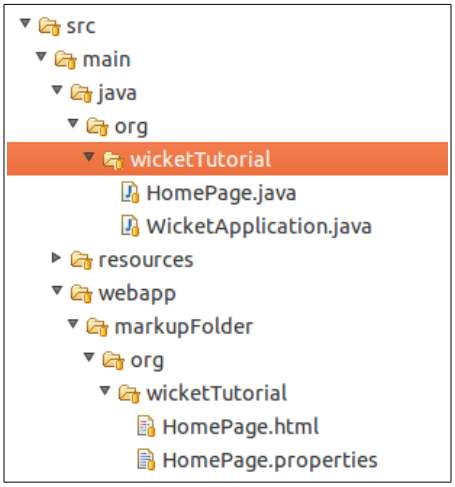
```
//init application's method
@Override
public void init(){
  //get the resource locator
  getResourceSettings().getResourceStreamLocator();
  //set the resource locator
  getResourceSettings().setResourceStreamLocator(myLocator);
}
```

The default locator used by Wicket is class `ResourceStreamLocator` which in turn tries to load a requested resource using a set of implementations of interface `IResourceFinder`. This interface defines method `find(Class class, String pathname)` which tries to resolve a resource corresponding to the given class and path.

The default implementation of `IResourceFinder` used by Wicket is `ClassPathResourceFinder` which searches for resources into the application class path. This is the implementation we have used so far in our examples. However some developers may prefer storing markup files and other resources in a separate folder rather than placing them side by side with Java classes.

To customize resource loading we can add further resource finders to our application in order to extend the resource-lookup algorithm to different locations. Wicket already comes with two other implementations of `IResourceFinder` designed to search for resources into a specific folder on the file system. The first is class `Path` and it's defined in package `org.apache.wicket.util.file`. The constructor of this class takes in input an arbitrary folder that can be expressed as a string path or as an instance of Wicket utility class `Folder` (in package `org.apache.wicket.util.file`). The second implementation of interface `IResourceFinder` is class `WebApplicationPath` which looks into a folder placed inside webapp's root path (but not inside folder WEB-INF).

Project *CustomFolder4MarkupExample* uses `WebApplicationPath` to load the markup file and the resource bundle for its home page from a custom folder. The folder is called `markupFolder` and it is placed in the root path of the webapp. The following picture illustrates the file structure of the project:



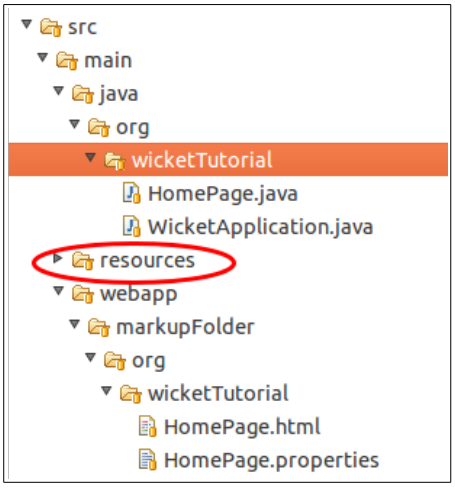
As we can see in the picture above, we must preserve the package structure also in the custom folder used as resource container. The code used inside application class to configure `WebApplicationPath` is the following:

```
@Override
public void init()
{
        getResourceSettings().getResourceFinders().add(
                        new WebApplicationPath(getServletContext(), "markupFolder"));
}
```

Method `getResourceFinders()` defined by setting interface `IResourceSettings` returns the list of resource finders defined in our application. The constructor of `WebApplicationPath` takes in input also an instance of standard interface `javax.servlet.ServletContext` which can be retrieved with `WebApplication`'s method `getServletContext()`.

Note

By default, if resource files can not be found inside application classpath, Wicket will search for them inside “resources” folder. You may have noted this folder in the previous picture. It is placed next to the folder “java” containing our source files:



This folder can be used to store resource files without writing any configuration code.

13.10 Summary

In this chapter we have learnt how to manage resources with the built-in mechanism provided by Wicket. With this mechanism we handle resources from Java code and Wicket will automatically take care of generating a valid URL for them. We have also seen how resources can be bundled as *package resources* with a component that depends on them to make it *self-contained*.

Then, in the second part of the chapter, we have built a custom resource and we have learnt how to mount it to an arbitrary URL and how to make it globally available as *shared resource*.

Finally, in the last part of the paragraph we took a peek at the mechanism provided by the framework to customize the locations where the resource-lookup algorithm searches for resources.

14 An example of integration with JavaScript

It's time to put into practice what we have learnt so far in this guide. To do this we will build a custom date component consisting of a text field to edit a date value and a fancy calendar icon to open a JavaScript datepicker. This chapter will also illustrate an example of integration of Wicket with a JavaScript library like JQuery[43] and its child project JQuery UI[44].

14.1 What we want to do...

For end-users a datepicker is one of the most appreciated widget. It allows to simply edit a date value with the help of a user-friendly pop-up calendar. That's why nearly all UI frameworks provide a version of this widget.

Popular JavaScript libraries like YUI and JQuery come with a ready-to-use datepicker to enrich the user experience of our web applications. Wicket already provides a component which integrates a text field with a calendar widget from YUI library[45], but there is no built-in component that uses a datepicker based on JQuery library.

As both JQuery and its child project JQueryUI have gained a huge popularity in the last years, it's quite interesting to see how to integrate them in Wicket building a custom component. In this chapter we will create a custom datepicker based on the corresponding widget from JQueryUI project:

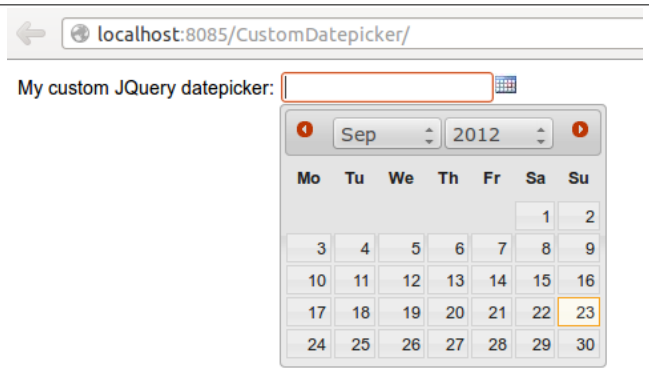


Illustration 14.1: How our custom datepicker will look like (localized for English).

Warning

On Internet you can find different libraries that already offer a strong integration between Wicket and JQuery[46]. The goal of this chapter is to see how to integrate Wicket with a JavaScript framework building a simple homemade datepicker which is not intended to provide every feature of the original JavaScript widget.

14.1.1 What features we want to implement.

43 http://jquery.com/
44 http://jqueryui.com/
45 *See component org.apache.wicket.datetime.markup.html.form.DateTextField*
46 *See jqwicket (http://code.google.com/p/jqwicket/) , wiquery (http://code.google.com/p/wiquery/) or Wicket - JQuery UI (http://www.7thweb.net /wicket-jquery-ui/)*

Before starting to write code, we must clearly define what features we want to implement for our component. The new component should:

- **Be self-contained**: we must be able to distribute it and use it in other projects without requiring any kind of additional configuration.
- **Have a customizable date format**: developer must be able to decide the date format used to display date value and to parse user input.
- **Be localizable**: the pop-up calendar must be localizable in order to support different languages.

That's what we'd like to have with our custom datepicker. In the rest of the chapter we will see how to implement the features listed above and which resources must be packaged with our component.

14.2 ...and how we will do it.

Our new component will extend the a built-in text field `org.apache.wicket.extensions.markup.html.form.DateTextField` which already uses a `java.util.Date` as model object and already performs conversion and validation for input values. Since the component must be self-contained, we must ensure that the JavaScript libraries it relies on (JQuery and JQuery UI) will be always available.

Starting from version 6.0 Wicket has adopted JQuery as backing JavaScript library so we can use the version bundled with Wicket for our custom datepicker.

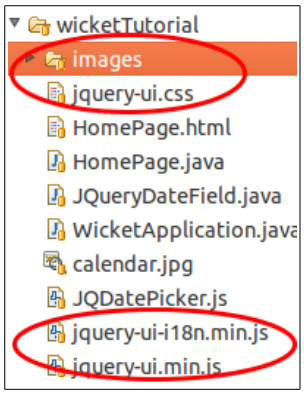
To make JQuery UI available we should instead go to its official site, download the required artifacts and use them as package resources of our component.

14.2.1 Component package resources

JQuery UI needs the following static resources in order to work properly:

- **jquery-ui.min.js**: the minified version of the library.
- **jquery-ui.css**: the CSS containing the style used by JQuery UI widgets.
- **jquery-ui-i18n.min.js**: the minified JavaScript containing the built-in support for localization.
- **Folder 'images'**: the folder containing picture files used by JQuery UI widgets.

In the following picture we can see these package resources with our component class (named `JQueryDateField`):



Along with the four static resources listed above, we can find also file calendar.jpg, which is the calendar icon used to open the pop up calendar, and file JQDatePicker.js which contains the following custom JavaScript code that binds our component to a JQuery UI datepicker:

```
function initJQDatepicker(inputId, countryIsoCode, dateFormat,  calendarIcon) {
```

```
        var localizedArray = $.datepicker.regional[countryIsoCode];
        localizedArray['buttonImage'] = calendarIcon;
        localizedArray['dateFormat'] = dateFormat;
        initCalendar(localizedArray);
        $("#" + inputId).datepicker(localizedArray);
};

function initCalendar(localizedArray){
         localizedArray['changeMonth']= true;
         localizedArray['changeYear']= true;
         localizedArray['showOn'] = 'button';
         localizedArray['buttonImageOnly'] = true;
};
```

Function `initJQDatepicker` takes in input the following parameters:

- **inputId**: the id of the HTML text field corresponding to our custom component instance.
- **countryIsoCode**: a two-letter low-case ISO language code. It can contain also the two-letter upper-case ISO country code separated with a minus sign (for example `en-GB`)
- **dateFormat**: the date format to use for parsing and displaying date values.
- **calendarIcon**: the relative URL of the icon used as calendar icon.

As we will see in the next paragraphs, its up to our component to generate this parameters and invoke the `initJQDatepicker` function.

Function `initCalendar` is a simple utility function that sets the initialization array for datepicker widget. For more details on JQuery UI datepicker usage see the documentation at http://jqueryui.com/datepicker.

14.2.2 Initialization code

The initialization code for our component is contained inside its method `onInitialize` and is the following:

```
@Override
protected void onInitialize() {
        super.onInitialize();
        setOutputMarkupId(true);

        datePattern =  new ResourceModel("jqueryDateField.shortDatePattern", "mm/dd/yy")
                                        .getObject();
        countryIsoCode = new ResourceModel("jqueryDateField.countryIsoCode", "en-GB")
                                        .getObject();

        PackageResourceReference resourceReference =
                                new PackageResourceReference(getClass(), "calendar.jpg");

        urlForIcon = urlFor(resourceReference, new PageParameters());
        dateConverter = new PatternDateConverter(datePattern, false);
}

@Override
public <Date> IConverter<Date> getConverter(Class<Date> type) {
```

```
        return (IConverter<Date>) dateConverter;
}
```

The first thing to do inside `onInitialize` is to ensure that our component will have a markup id for its related text field. This is done invoking `setOutputMarkupId(true)`.

Next, `JQueryDateField` tries to retrieve the date format and the ISO language code that must be used as initialization parameters. This is done using class `ResourceModel` which searches for a given resource in the available bundles. If no value is found for date format or for ISO language code, default values will be used ('`mm/dd/yy`' and '`en-GB`').

To generate the relative URL for calendar icon, we load it as package resource reference and then we use `Component`'s method `urlFor` to get the URL value (we have seen this method in paragraph 7.3.2).

The last configuration instruction executed inside `onInitialize` is the instantiation of the custom converter used by our component. This converter is an instance of the built-in class `org.apache.wicket.datetime.PatternDateConvert` and must use the previously retrieved date format to perform conversion operations. Now to tell our component to use this converter we must return it overriding `FormComponent`'s method `getConverter`.

14.2.3 Header contributor code

The rest of the code of our custom component is inside method `renderHeader`, which is responsible for adding to page header the bundled JQuery library, the three files from JQuery UI distribution, the custom file JQDatePicker.js and the invocation of function `initJQDatepicker`:

```
@Override
public void renderHead(IHeaderResponse response) {
        super.renderHead(response);

        //if component is disabled we don't have to load the JQueryUI datepicker
        if(!isEnabledInHierarchy())
                return;
        //add bundled JQuery
        IJavaScriptLibrarySettings javaScriptSettings =
                        getApplication().getJavaScriptLibrarySettings();
        response.render(JavaScriptHeaderItem.
                        forReference(javaScriptSettings.getJQueryReference()));
        //add package resources
        response.render(JavaScriptHeaderItem.
                forReference(new PackageResourceReference(getClass(), "jquery-ui.min.js")));
        response.render(JavaScriptHeaderItem.
                forReference(new PackageResourceReference(getClass(), "jquery-ui-i18n.min.js")));
        response.render(CssHeaderItem.
                forReference(new PackageResourceReference(getClass(), "jquery-ui.css")));
        //add custom file JQDatePicker.js. Reference JQDatePickerRef is a static field
        response.render(JavaScriptHeaderItem.forReference(JQDatePickerRef));

        //add the init script for datepicker
        String jqueryDateFormat = datePattern.replace("yyyy", "yy").toLowerCase();
        String initScript = ";initJQDatepicker('" + getMarkupId() + "', '" + countryIsoCode +
                        "', '" + jqueryDateFormat + "', " + "'" + urlForIcon +"');";
        response.render(OnLoadHeaderItem.forScript(initScript));
}
```

If component is disabled the calendar icon must be hidden and no datepicker must be displayed.

That's why `renderHeader` is skipped if component is not enabled.

To get a reference to the bundled JQuery library we used the JavaScript setting interface `IJavaScriptLibrarySettings` and its method `getJQueryReference`.

In the last part of `renderHeader` we build the string to invoke function `initJQDatepicker` using the values obtained inside `onInitialize`. Unfortunately the date format used by JQuery UI is different from the one adopted in Java so we have to convert it before building the JavaScript code. This init script is rendered into header section using a `OnLoadHeaderItem` to ensure that it will be executed after all the other scripts have been loaded.

Note

If we add more than one instance of our custom component to a single page, static resources are rendered to the header section just once. Wicket automatically checks if a static resource is already referenced by a page and if so, it will not render it again.
This does not apply to the init script which is dynamically generated and is rendered for every instance of the component.

Warning

Our datepicker is not ready yet to be used with AJAX. In chapter 16 we will see how to modify it to make it AJAX-compatible.

14.3 Summary

In this brief chapter we have seen how custom components can be integrated with DHTML[47] technologies. To do so we have used most of what we have learnt in this guide. Now we are able to build complex components with a rich user experience. However this is not enough yet to develop Web 2.0[48] applications. We still have to cover a fundamental technology like AJAX and some other Wicket-related topics that will help us building our application in more modular and efficient way.

47 http://en.wikipedia.org/wiki/Dynamic_HTML
48 http://en.wikipedia.org/wiki/Web_2.0

15 Wicket advanced topics

In this chapter we will learn some advanced topics which have not been covered yet in the previous chapters but which are nonetheless essential to make the most of Wicket and to build sophisticated web applications.

15.1 Enriching components with behaviors

With class `org.apache.wicket.behavior.Behavior` Wicket provides a very flexible mechanism to share common features across different components and to enrich existing components with further functionalities. As the class name suggests, `Behavior` adds a generic *behavior* to a component modifying its markup and/or contributing to the header section of the page (`Behavior` implements the interface `IHeaderContributor`).

One or more behaviors can be added to a component with `Component`'s method `add(Behavior...)`, while to remove a behavior we must use method `remove(Behavior)`.

Here is a partial list of methods defined inside class `Behavior` along with a brief description of what they do:

- **beforeRender(Component component)**: called when a component is about to be rendered.
- **afterRender(Component component)**: called after a component has been rendered.
- **onComponentTag(Component component, ComponentTag tag)**: called when component tag is being rendered.
- **getStatelessHint(Component component)**: returns if a behavior is stateless or not.
- **bind(Component component)**: called after a behavior has been added to a component.
- **unbind(Component component)**: called when a behavior has been removed from a component.
- **detach(Component component)**: overriding this method a behavior can detach its state before being serialized.
- **isEnabled(Component component)**: tells if the current behavior is enabled for a given component. When a behavior is disabled it will be simply ignored and not executed.
- **isTemporary(Component component)**: tells component if the current behavior is temporary. A temporary behavior is discarded at the end of the current request (i.e it's executed only once).
- **onConfigure(Component component)**: called right after the owner component has been configured.
- **onRemove(Component component)**: called when the owner component has been removed from its container.
- **renderHead(Component component, IHeaderResponse response)**: overriding this method behaviors can render resources to the header section of the page.

For example the following behavior prepends a red asterisk to the tag of a form component if this one is required:

```
public class RedAsteriskBehavior extends Behavior {

  @Override
  public void beforeRender(Component component) {
```

```
      Response response = component.getResponse();
      StringBuffer asterisktHtml = new StringBuffer(200);

      if(componet instanceof FormComponent
            && ((FormComponent)component).isRequired()){
        asteriskHtml.append(" <b style=\"color:red;font-size:medium\">*</b>");
      }
      response.write(asteriskHtml);
  }
}
```

Since method `beforeRender` is called before the coupled component is rendered, we can use it to prepend custom markup to component tag. This can be done writing our markup directly to the current `Response` object, as we did in the example above.

Please note that we could achieve the same result overriding component method `onBeforeRender`. However using a behavior we can easily reuse our custom code with any other kind of component without modifying its source code. As general best practice we should always consider to implement a new functionality using a behavior if it can be shared among different kinds of component.

Behaviors play also a strategic role in the built-in AJAX support provided by Wicket, as we will see in the next chapter.

15.2 Generating callback URLs with IRequestListener

With Wicket it's quite easy to build a *callback URL* that executes a specific method on server side. This method must be defined in a *functional interface*[49] that inherits from built-in `org.apache.wicket.IRequestListener` and it must be a void method with no parameters in input:

```
public interface IMyListener extends IRequestListener
{
       /**
        * Called when the relative callback URL is requested.
        */
       void myCallbackMethod();
}
```

To control how the method will be invoked we must use class `org.apache.wicket.RequestListenerInterface`. In Wicket is a common practice to instantiate this class as a public static field inside the relative callback interface:

```
public interface IMyListener extends IRequestListener
{
       /**RequestListenerInterface instance*/
       public static final RequestListenerInterface INTERFACE = new
                                        RequestListenerInterface(IMyListener.class);
       /**
        * Called when the relative callback URL is requested.
        */
       void myCallbackMethod();
}
```

By default `RequestListenerInterface` will respond rendering the current page after the callback

49 *A functional interface is an an interface that defines just one method.*

method has been executed (if we have a non-AJAX request). To change this behavior we can use setter method `setRenderPageAfterInvocation(boolean)`.

Now that our callback interface is complete we can generate a callback URL with `Component`'s method `urlFor(RequestListenerInterface, PageParameters)` or with method `urlFor(Behavior, RequestListenerInterface, PageParameters)` if we are using a callback interface with a behavior (see the following example).

Project *CallbackURLExample* contains a behavior (class `OnChangeSingleChoiceBehavior`) that implements a callback interface to update the model of an `AbstractSingleSelectChoice` component when user changes the selected option (it provides the same functionality of method `wantOnSelectionChangedNotifications`).

Instead of a custom callback interface, `OnChangeSingleChoiceBehavior` implements built-in interface `org.apache.wicket.behavior.IBehaviorListener` which is designed to generate a callback URL for behaviors. The callback method defined in this interface is `onRequest()` and the following is the implementation provided by `OnSelectionChangedNotifications`:

```
@Override
public void onRequest() {
        Request request = RequestCycle.get().getRequest();
        IRequestParameters requestParameters = request.getRequestParameters();
        StringValue choiceId = requestParameters.getParameterValue("choiceId");
        //boundComponent is the component that the behavior it is bound to.
        boundComponent.setDefaultModelObject(convertChoiceIdToChoice(choiceId.toString()));
}
```

When invoked via URL, the behavior expects to find a request parameter (`choiceId`) containing the id of the selected choice. This value is used to obtain the corresponding choice object that must be used to set the model of the component that the behavior is bound to (`boundComponent`). Method `convertChoiceIdToChoice` is in charge of retrieving the choice object given its id and it has been copied from class `AbstractSingleSelectChoice`.

Another interesting part of `OnChangeSingleChoiceBehavior` is its method `onComponentTag` where some JavaScript “magic” is used to move user's browser to the callback URL when event “change” occurs on bound component:

```
@Override
public void onComponentTag(Component component, ComponentTag tag) {
        super.onComponentTag(component, tag);

        CharSequence callBackURL = getCallbackUrl();
        String separatorChar = (callBackURL.toString().indexOf('?') > -1 ? "&" : "?");

        String finalScript = "var isSelect = $(this).is('select');\n" +
                                "var component;\n" +
                                "if(isSelect)\n" +
                                "      component = $(this);\n" +
                                "else \n" +
                                "      component = $(this).find('input:radio:checked');\n" +
                                "window.location.href='" + callBackURL +  separatorChar +
                                "choiceId=' + " + "component.val()";

        tag.put("onchange", finalScript);
}
```

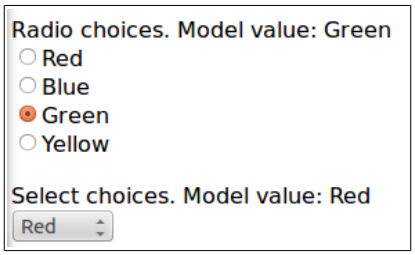
The goal of `onComponentTag` is to build an `onchange` handler that forces user's browser to move to the callback URL (modifing standard property `window.location.href`). Please note that we have appended the expected parameter (`choiceId`) to the URL retrieving its value with a JQuery selector suited for the current type of component (a drop-down menu or a radio group). Since we are using JQuery in our JavaScript code, the behavior comes also with method `renderHead` that adds the bundled JQuery library to the current page.

Method `getCallbackUrl()` is used to generate the callback URL for our custom behavior and it has been copied from built-in class `AbstractAjaxBehavior`:

```
public CharSequence getCallbackUrl(){
        if (boundComponent == null){
                throw new IllegalArgumentException(
                        "Behavior must be bound to a component to create the URL");
        }

        final RequestListenerInterface rli;

        rli = IBehaviorListener.INTERFACE;

        return boundComponent.urlFor(this, rli, new PageParameters());
}
```

Static field `IBehaviorListener.INTERFACE` is the implementation of `RequestListenerInterface` defined inside callback interface `IBehaviorListener`.

The home page of project *CallbackURLExample* contains a `DropDownChoice` and a `RadioChoice` which use our custom behavior. There are also two labels to display the content of the models of the two components:



Note

Implementing interface `IBehaviorListener` makes a behavior stateful because its callback URL is specific for a given instance of component.

15.3 Wicket events infrastructure

Starting from version 1.5 Wicket offers an event-based infrastructure for inter-component communication. The infrastructure is based on two simple interfaces (both in package `org.apache.wicket.event`) : `IEventSource` and `IEventSink`.

The first interface must be implemented by those entities that want to broadcast en event while the second interface must be implemented by those entities that want to receive a broadcast event.

The following entities already implement both these two interfaces (i.e. they can be either sender or receiver): `Component`, `Session`, `RequestCycle` and `Application`.

`IEventSource` exposes a single method named `send` which takes in input three parameters:

- **sink**: an implementation of `IEventSink` that will be the receiver of the event.
- **broadcast**: a `Broadcast` enum which defines the broadcast method used to dispatch the

event to the sink and to other entities such as sink children, sink containers, session object, application object and the current request cycle. It has four possible values:

Value	Description
BREADTH	The event is sent first to the specified sink and then to all its children components following a breadth-first order.
DEPTH	The event is sent to the specified sink only after it has been dispatched to all its children components following a depth-first order.
BUBBLE	The event is sent first to the specified sink and then to its parent containers.
EXACT	The event is sent only to the specified sink.

- **payload**: a generic object representing the data sent with the event.

Each broadcast mode has its own traversal order for `Session`, `RequestCycle` and `Application`. See JavaDoc of class `Broadcast` for further details about this order.

Interface `IEventSink` exposes callback method `onEvent(IEvent<?> event)` which is triggered when a sink receives an event. The interface `IEvent` represents the received event and provides getter methods to retrieve the event broadcast type, the source of the event and its payload. Typically the received event is used checking the type of its payload object :

```
@Override
public void onEvent(IEvent event) {
  //if the type of payload is MyPayloadClass perform some actions
  if(event.getPayload() instanceof MyPayloadClass) {
     //execute some business code.
  }else{
     //other business code
  }
}
```

Project *InterComponetsEventsExample* provides a concrete example of sending an event to a component (named 'container in the middle') using all the available broadcast methods:

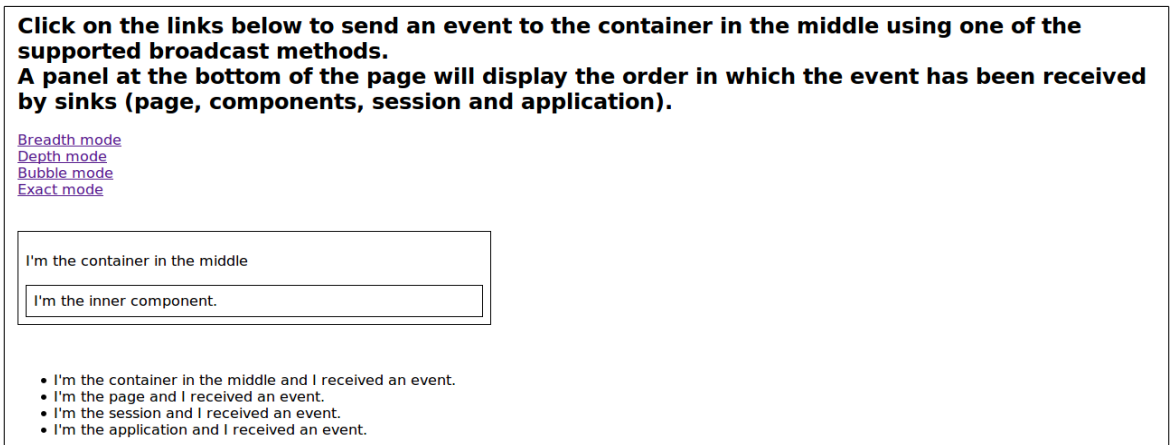


Illustration 15.1: Project InterComponentsEventsExample displaying the order in which the entities are notified using broadcast type BUBBLE

15.4 Initializers

Some components or resources may need to be configured before being used in our applications. While so far we used `Application`'s `init` method to initialize these kinds of entities, Wicket offers a more flexible and modular way to configure our classes.

During application's bootstrap Wicket searches for any properties file named `wicket.properties` placed in one of the classpath roots visible to the application[50]. When one of these files is found, the *initializer* defined inside it will be executed. An initializer is an implementation of interface `org.apache.wicket.IInitializer` and is defined inside `wicket.properties` with a line like this:

```
initializer=org.wicketTutorial.MyInitializer
```

The fully qualified class name corresponds to the initializer that must be executed. Interface `IInitializer` defines method `init(Application)` which should contain our initialization code, and method `destroy(Application)` which is invoked when application is terminated:

```
public class MyInitializer implements IInitializer{

	public void init(Application application) {
		//initialization code
	}

	public void destroy(Application application) {
		//code to execute when application is terminated
	}
}
```

Only one initializer can be defined in a single `wicket.properties` file. To overcome this limit we can create a main initializer that in turn executes every initializer we need:

```
public class MainInitializer implements IInitializer{

	public void init(Application application) {
		new AnotherInitializer().init(application);
```

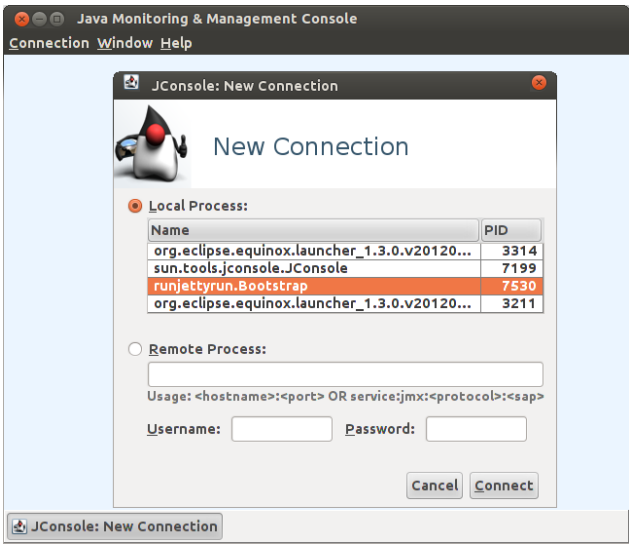
50 *This set of classpath roots includes the root of the package of the application class and the roots of the visible JARs*

```
                new YetAnotherInitializer().init(application);
                //...
        }
        //destroy...
}
```

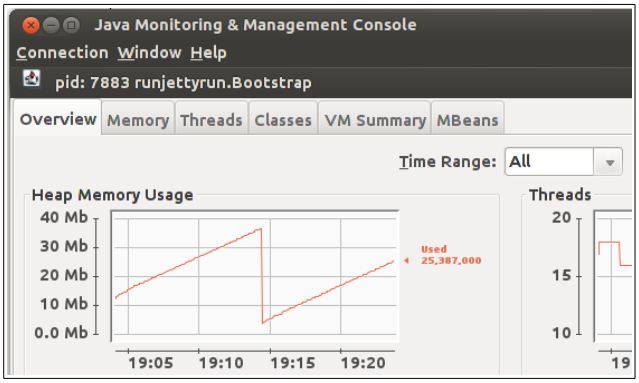
15.5 Using JMX with Wicket

JMX (*Java Management Extensions*) is the standard technology adopted in Java for managing and monitoring running applications or Java Virtual Machines. Wicket offers support for JMX through module `wicket-jmx`. In this paragraph we will see how we can connect to a Wicket application using JMX. In our example we will use JConsole as JMX client. This program is bundled with Java SE since version 5 and we can run it typing `jconsole` in our OS shell.

Once JConsole has started it will ask us to establish a new connection to a Java process, choosing between a local process or a remote one. In the following picture we have selected the process corresponding to the local instance of Jetty server we used to run one of our example projects:



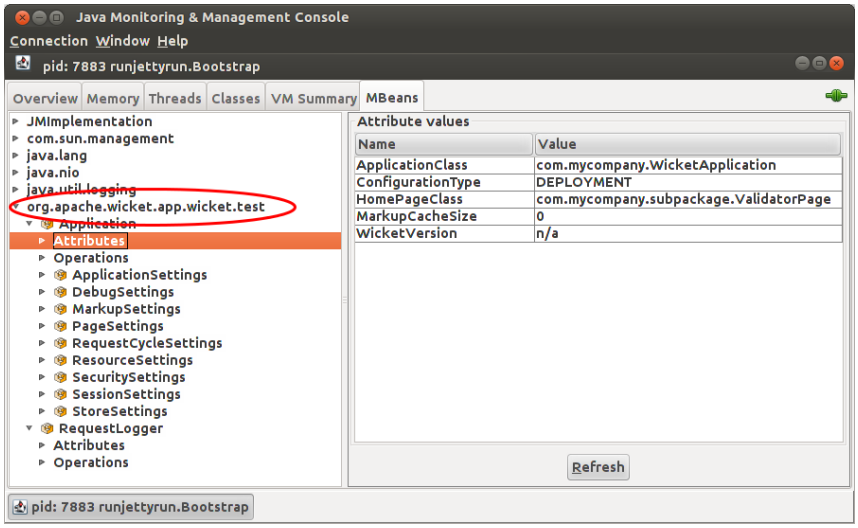
After we have established a JMX connection, JConsole will show us the following set of tabs



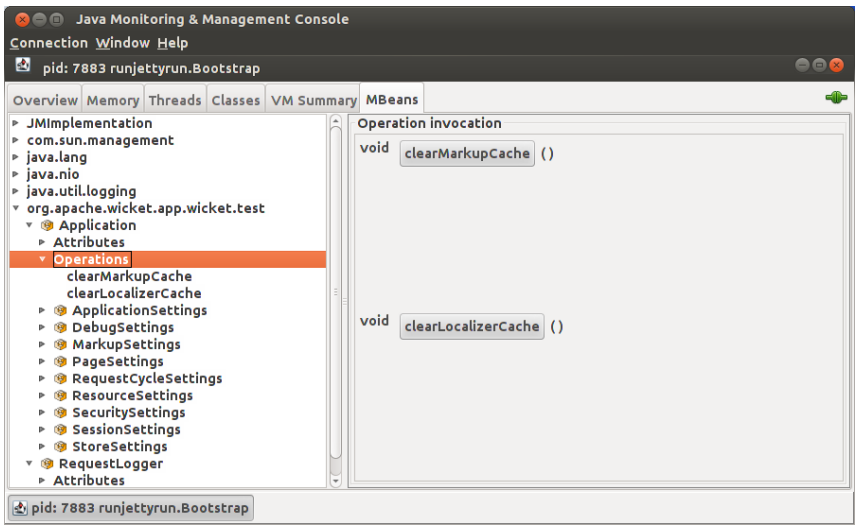
JMX exposes application-specific informations using special objects called MBeans (*Manageable Beans*), hence if we want to control our application we must open the corresponding tab. The MBeans containing the application's informations is named `org.apache.wicket.app.<filter/servlet name>`.

In our example we have used `wicket.test` as filter name[51] for our application:

51 *We have discussed filter name and web.xml in paragraph 2.2.1.*

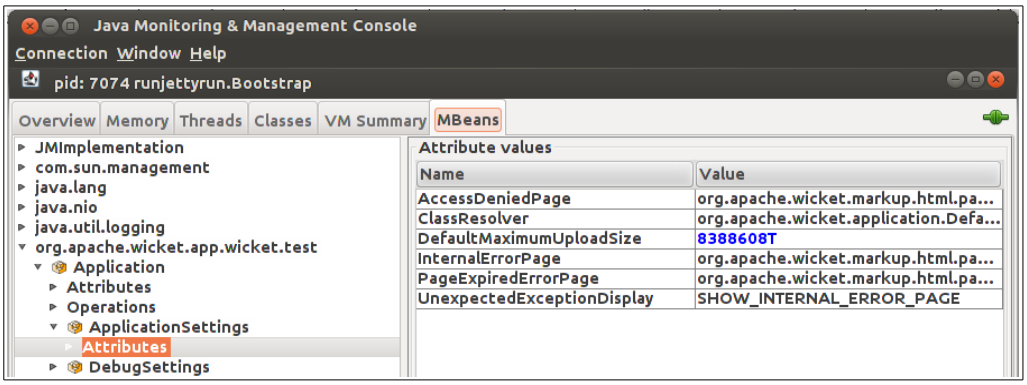


As we can see in the picture above, every MBean exposes a node containing its *attributes* and another node showing the possible *operations* that can be performed on the object. In the case of a Wicket application the available operations are `clearMarkupCache` and `clearLocalizerCache`:



With these two operations we can force Wicket to clear the internal caches used to load components markup and resource bundles. This can be particularly useful if we have our application running in DEPLOYMENT mode and we want to publish minor fixes for markup or bundle files (like spelling or typo corrections) without restarting the entire application. Without cleaning these two caches Wicket would continue to use cached values ignoring any change made to markup or bundle files.

Some of the exposed properties are editable, hence we can tune their values while the application is running. For example if we look at the properties of `ApplicationSettings` we can set the maximum size allowed for an upload modifying the attribute `DefaultMaximumUploadSize`:



15.6 Generating HTML markup from code.

So far, as markup source for our pages/panels we have used a static markup file, no matter if it was inherited or directly associated to the component. Now we want to investigate a more complex use case where we want to *dynamical* generate the markup directly inside component code.

To become a markup producer, a component must simply implement interface `org.apache.wicket.markup.IMarkupResourceStreamProvider`. The only method defined in this interface is `getMarkupResourceStream(MarkupContainer, Class<?>)` which returns an utility interface called `IResourceStream` representing the actual markup.

In the following example we have a custom panel without a related markup file that generates a simple `<div>` tag as markup:

```
public class AutoMarkupGenPanel extends Panel implements IMarkupResourceStreamProvider
```

```
{
        public AutoMarkupGenPanel(String id, IModel<?> model) {
                super(id, model);
        }

        @Override
        public IResourceStream getMarkupResourceStream(MarkupContainer container,
                        Class<?> containerClass) {
                String markup = "<div>Panel markup</div>";
                StringResourceStream resourceStream = new StringResourceStream(markup);

                return resourceStream;
        }
}
```

Class `StringResourceStream` is a resource stream that uses a `String` instance as backing object.

15.6.1 Avoiding markup caching

As we have seen in the previous paragraph, Wicket uses an internal cache for components markup. This can be a problem if our component dynamical generates its markup when it is rendered because once the markup has been cached, Wicket will always use the cached version for the specific component. To overwrite this default caching policy, a component can implement interface `IMarkupCacheKeyProvider`.

This interface defines method `getCacheKey(MarkupContainer, Class<?>)` which returns a string value representing the key used by Wicket to retrieve the markup of the component from the cache. If this value is `null` the markup will not be cached, allowing the component to display the last generated markup each time it is rendered:

```
public class NoCacheMarkupPanel extends Panel implements IMarkupCacheKeyProvider {
        public NoCacheMarkupPanel(String id, IModel<?> model) {
                super(id, model);
        }

        /**
        * Generate a dynamic HTML markup that changes every time
        * the component is rendered
        */
        @Override
        public IResourceStream getMarkupResourceStream(MarkupContainer container,
                        Class<?> containerClass) {
                String markup = "<div>Panel with current nanotime: " + System.nanoTime() +
                                "</div>";
                StringResourceStream resourceStream = new StringResourceStream(markup);

                return resourceStream;
        }

        /**
        * Avoid markup caching for this component
        */
        @Override
        public String getCacheKey(MarkupContainer arg0, Class<?> arg1) {
                return null;
```

```
        }
}
```

15.7 Summary

In this chapter we have introduced some advanced topics we didn't have the chance to cover yet. We have started talking about behaviors and we have seen how they can be used to enrich existing components (promoting a component-oriented approach). Behaviors are also fundamental to work with AJAX in Wicket, as we will see in the next chapter.

After behaviors we have learnt how to generate callback URLs to execute a custom method on server side defined inside a specific callback interface.

The third topic of the chapter has been the event infrastructure provided in Wicket for inter-component communication which brings to our components a desktop-like event-driven architecture.

Then, we have introduced a new entity called initializer which can be used to configure resources and component in a modular and self-contained way.

We have also looked at Wicket support for JMX and we have seen how to use this technology for monitoring and managing our running applications.

Finally we have introduced a new technique to generate the markup of a component from its Java code.

16 Working with AJAX

AJAX has become a *must-have* for nearly all kinds of web application. This technology does not only help to achieve a better user experience but it also allows to improve the bandwidth performance of web applications. Using AJAX usually means writing tons of JavaScript code to handle asynchronous requests and to update user interface, but with Wicket we can leave all this boilerplate code to the framework and we don't even need to write a single line of JavaScript to start using AJAX.

In this chapter we will learn how to leverage the AJAX support provided by Wicket to make our applications fully *web 2.0*[52] compliant.

16.1 How to use AJAX components and behaviors.

Wicket support for AJAX is implemented in file `wicket-ajax-jquery.js` which makes complete transparent to Java code any detail about AJAX communication.

AJAX components and behaviors shipped with Wicket expose one or more callback methods which are executed when they receive an AJAX request. One of the arguments of these methods is an instance of interface `org.apache.wicket.ajax.AjaxRequestTarget`.

For example component `AjaxLink` (in package `org.apache.wicket.ajax.markup.html`) defines abstract method `onClick(AjaxRequestTarget target)` which is executed when user clicks on the component:

```
new AjaxLink("ajaxLink"){
        @Override
        public void onClick(AjaxRequestTarget target) {
            //some server side code...
        }
};
```

Using `AjaxRequestTarget` we can specify the content that must be sent back to the client as response to the current AJAX request. The most commonly used method of this interface is probably `add(Component... components)`. With this method we tell Wicket to render again the specified components and refresh their markup via AJAX:

```
new AjaxLink("ajaxLink"){
        @Override
        public void onClick(AjaxRequestTarget target) {
            //modify the model of a label and refresh it on browser
            label.setDefaultModelObject("Another value 4 label.");
            target.add(label);
        }
};
```

Components can be refreshed via Ajax only if they have rendered a markup id for their related tag. As a consequence, we must remember to set a valid id value on every component we want to add to `AjaxRequestTarget`. This can be done using one of the two methods seen in paragraph 4.3:

52 http://en.wikipedia.org/wiki/Web_2.0

```
final Label label = new Label("labelComponent", "Initial value.");
//autogenerate a markup id
label.setOutputMarkupId(true);
add(label);
//...
new AjaxLink("ajaxLink"){
        @Override
        public void onClick(AjaxRequestTarget target) {
            //modify the model of a label and refresh it on client side
            label.setDefaultModelObject("Another value 4 label.");
            target.add(label);
        }
};
```

Another common use of `AjaxRequestTarget` is to prepend or append some JavaScript code to the generated response. For example the following AJAX link displays an alert box as response to user's click:

```
new AjaxLink("ajaxLink"){
        @Override
        public void onClick(AjaxRequestTarget target) {
            target.appendJavaScript(";alert('Hello!!');");
        }
};
```

Warning

Repeaters component that have `org.apache.wicket.markup.repeater AbstractRepeater` as base class (like `ListView`, `RepeatingView`, etc...) can not be directly updated via AJAX.
if we want to refresh their markup via AJAX we must add one of their parent containers to the `AjaxRequestTarget`.

16.2 Built-in AJAX components

Wicket distribution comes with a number of built-in AJAX components ready to be used. Some of them are the *ajaxified* version of common components like links and buttons, while others are AJAX-specific components.

AJAX components are not different from any other component seen so far and they don't require any additional configuration to be used. As we will shortly see, switching from a classic link or button to the ajaxified version is just a matter of appending “Ajax” to the component class name.

This paragraph provides an overview of what we can find in Wicket to start writing AJAX-enhanced web applications.

16.2.1 Links and buttons

In the previous paragraph we have already introduced component `AjaxLink`. Wicket provides also the ajaxified versions of submitting components `SubmitLink` and `Button` which are simply called `AjaxSubmitLink` and `AjaxButton`. These components come with a version of methods `onSubmit`, `onError` and `onAfterSubmit` that takes in input also an instance of `AjaxRequestTarget`.

Both components are in package `org.apache.wicket.ajax.markup.html.form`.

16.2.2 Fallback components

Building an entire site using AJAX can be risky as some clients may not support this technology. In order to provide an usable version of our site also to these clients, we can use components `AjaxFallbackLink` and `AjaxFallbackButton` which are able to automatically degrade to a standard link or to a standard button if client doesn't support AJAX.

16.2.3 AJAX Checkbox

Class `org.apache.wicket.ajax.markup.html.form.AjaxCheckBox` is a checkbox component that updates its model via AJAX when user changes its value. Its AJAX callback method is `onUpdate(AjaxRequestTarget target)`. The component extends standard checkbox component `CheckBox` adding an `AjaxFormComponentUpdatingBehavior` to itself (we will see this behavior later in paragraph 16.3.3).

16.2.4 AJAX editable labels

An *editable label* is a special label that can be edited by user when she/he clicks on it. Wicket ships three different implementations for this component (all inside package `org.apache.wicket.extensions.ajax.markup.html`):

- **AjaxEditableLabel**: it's a basic version of editable label. User can edit the content of the label with a text field. This is also the base class for the other two editable labels.
- **AjaxEditableMultiLineLabel**: this label supports multi-line values and uses a text area as editor component.
- **AjaxEditableChoiceLabel**: this label uses a drop-down menu to edit its value.

Base component `AjaxEditableLabel` exposes the following set of AJAX-aware methods that can be overriden:

- **onEdit(AjaxRequestTarget target)**: called when user clicks on component. The default implementation shows the component used to edit the value of the label.
- **onSubmit(AjaxRequestTarget target)**: called when the value has been successfully updated with the new input.
- **onError(AjaxRequestTarget target)**: called when the new inserted input has failed validation.
- **onCancel(AjaxRequestTarget target)**: called when user has exited from editing mode pressing escape key. The default implementation brings back the label to its initial state hiding the editor component.

Wicket module `wicket-examples` contains page class `EditableLabelPage.java` which shows all these three components together. You can see this page in action at http://www.wicket-library.com/wicket-examples-6.0.x/ajax/editable-label:

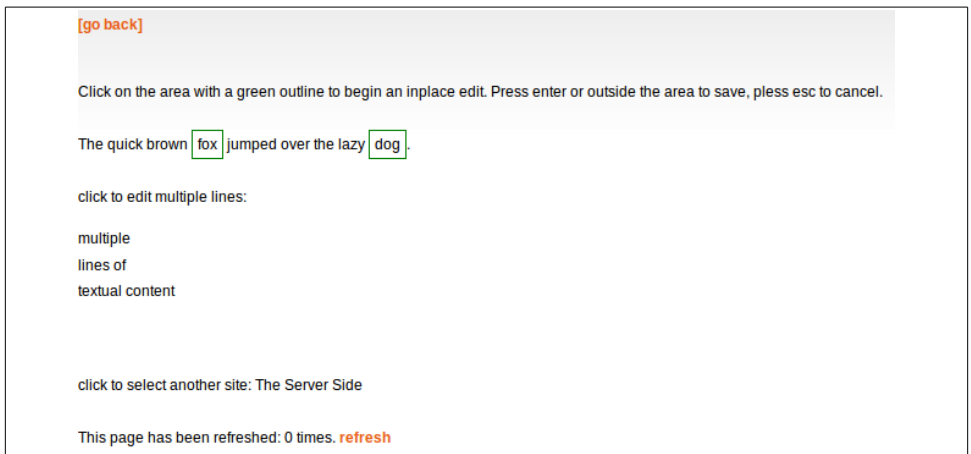


Illustration 16.1: The edit label example bundled with Wicket

16.2.5 Autocomplete text field

On Internet we can find many examples of text fields that display a list of suggestions (or options) while the user types a text inside them. This feature is known as *autocomplete* functionality.

Wicket offers an out-of-the-box implementation of an autocomplete text field with component `org.apache.wicket.extensions.ajax.markup.html.autocomplete.AutoCompleteTextField`.

When using `AutoCompleteTextField` we are required to implement its abstract method `getChoices(String input)` where the `input` parameter is the current input of the component. This method returns an iterator over the suggestions that will be displayed as a drop-down menu:

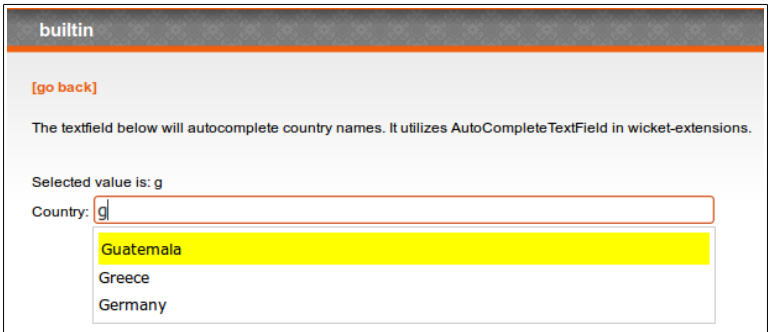


Illustration 16.2: The autocomplete example bundled with Wicket

Suggestions are rendered using a render which implements interface `IAutoCompleteRenderer`. The default implementation simply calls `toString()` on each suggestion object. If we need to work with a custom render we can specify it via component constructor.

`AutoCompleteTextField` supports a wide range of settings that are passed to its constructor with class `AutoCompleteSettings`.

One of the most interesting parameter we can specify for `AutoCompleteTextField` is the throttle delay which is the amount of time (in milliseconds) that must elapse between a change of input value and the transmission of a new Ajax request to display suggestions. This parameter can be set with method `setThrottleDelay(int)`:

```
AutoCompleteSettings settings = new AutoCompleteSettings();
//set throttle to 400 ms: component will wait 400ms before displaying the options
settings.setThrottleDelay(400);
//...
AutoCompleteTextField field = new AutoCompleteTextField<T>("field", model) {

        @Override
        protected Iterator getChoices(String arg0) {
                //return an iterator over the options
        }
};
```

Wicket module `wicket-examples` contains page class `AutoCompletePagePage.java` which shows an example of autocomplete text field. The running example is available at http://www.wicket-library.com/wicket-examples-6.0.x/ajax/autocomplete.

16.2.6 Modal window

Class `org.apache.wicket.extensions.ajax.markup.html.modal.ModalWindow` is an imp-

lementation of a modal window[53] based on AJAX:

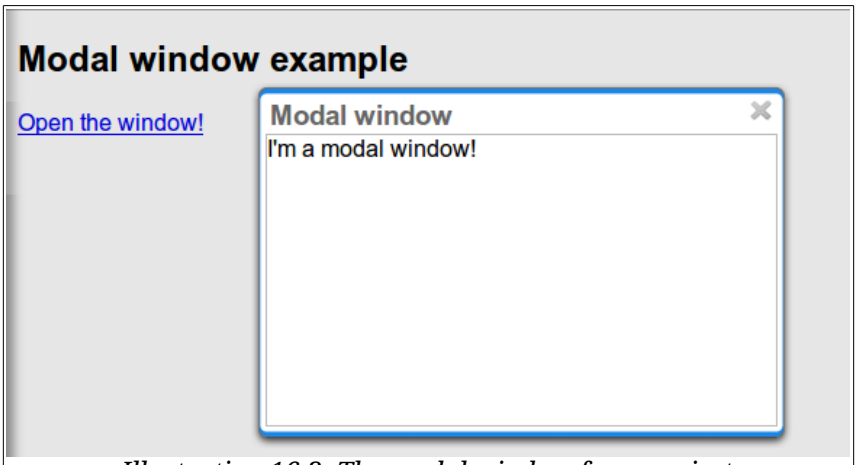


Illustration 16.3: The modal window from project BasicModalWindowExample

The content of a modal window can be either another component or a page. In the first case the id of the component used as content must be retrieved with method `getContentId()`.

If instead we want to use a page as window content, we must implement the inner interface `ModalWindow.`*PageCreator* and pass it to method `setPageCreator`. The page used as content will be embedded in a `<iframe>` tag.

To display a modal window we must call its method `show(AjaxRequestTarget target)`. This is usually done inside the AJAX callback method of another component (like an `AjaxLink`). The following markup and code are taken from project *BasicModalWindowExample* and illustrate a basic usage of a modal window:

Html:

```
<body>
        <h2>Modal Windod example</h2>
        <a wicket:id="openWindow">Open the window!</a>
        <div wicket:id="modalWindow"></div>
</body>
```

Java code:

```
 public HomePage(final PageParameters parameters) {
        super(parameters);
        final ModalWindow modalWindow = new ModalWindow("modalWindow");
        Label label = new Label(modalWindow.getContentId(), "I'm a modal window!");

        modalWindow.setContent(label);
        modalWindow.setTitle("Modal window");

        add(modalWindow);
        add(new AjaxLink("openWindow") {
                @Override
                public void onClick(AjaxRequestTarget target) {
                      modalWindow.show(target);
                }
```

53 http://en.wikipedia.org/wiki/Modal_window

```
                });
    }
```

Just like any other component also `ModalWindow` must be added to a markup tag, like we did in our example using a `<div>` tag. Wicket will automatically hide this tag in the final markup appending the style value `display:`*`none`*.

The component provides different setter methods to customize the appearance of the window:

- **setTitle(String)**: specifies the title of the window
- **setResizable(boolean)**: by default the window is resizeable. If we need to make its size fixed we can use this method to turn off this feature.
- **setInitialWidth(int)** and **setInitialHeight(int)**: set the initial dimensions of the window.
- **setMinimalWidth(int)** and **setMinimalHeight(int)**: specify the minimal dimensions of the window.
- **setCookieName(String)**: this method can be used to specify the name of the cookie used on client side to store size and position of the window when it is closed. The component will use this cookie to restore these two parameters the next time the window will be opened.
  If no cookie name is provided, the component will not remember its last position and size.
- **setCssClassName(String)**: specifies the CSS class used for the window.
- **setAutoSize(boolean)**: when this flag is set to `true` the window will automatically adjust its size to fit content width and height. By default it is `false`.

The modal window can be closed from code using its method `close(AjaxRequestTarget target)`. The currently opened window can be closed also with the following JavaScript instruction:

```
Wicket.Window.get().close();
```

`ModalWindow` gives the opportunity to perform custom actions when window is closing. Inner interface `ModalWindow.`*WindowClosedCallback* can be implemented and passed to window's method `setWindowClosedCallback` to specify the callback that must be executed after window has been closed:

```
modalWindow.setWindowClosedCallback(new ModalWindow.WindowClosedCallback() {

        @Override
        public void onClose(AjaxRequestTarget target) {
          //custom code...
        }
});
```

16.2.7 Tree repeaters

Class `org.apache.wicket.extensions.markup.html.repeater.tree.AbstractTree` is the base class of another family of repeaters called *tree repeaters* and designed to display a data hierarchy as a tree, resembling the behavior and the look & feel of desktop *tree components*. A classic example of tree component on desktop is the tree used by nearly all file managers to navigate file system:

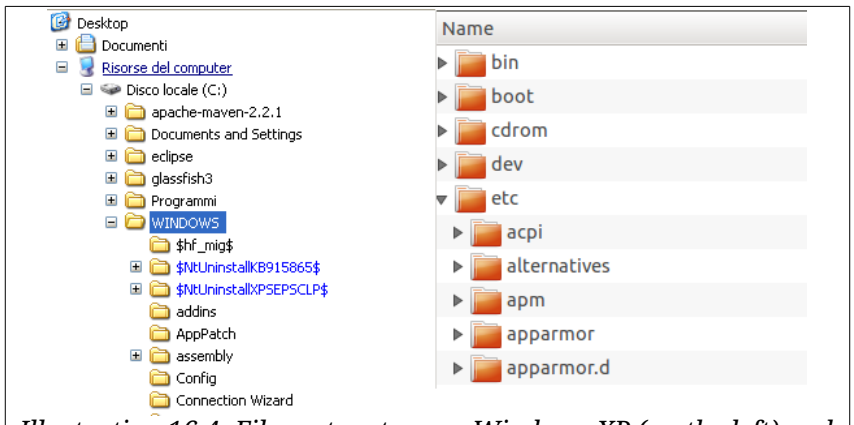


Illustration 16.4: File system tree on Windows XP (on the left) and Ubuntu (on the right)

Because of their highly interactive nature, tree repeaters are implemented as AJAX components, meaning that they are updated via AJAX when we expand or collapse their nodes.

The basic implementation of a tree repeater shipped with Wicket is component `NestedTree`. In order to use a tree repeater we must provide an implementation of interface `ITreeProvider` which is in charge of returning the nodes that compose the tree.

Wicket comes with a built-in implementation of `ITreeProvider` called `TreeModelProvider` that works with the same tree model[54] and nodes used by Swing component `javax.swing.JTree`. These Swing entities should be familiar to you if you have previously worked with the old tree repeaters (components `Tree` and `TreeTable`) that have been deprecated with Wicket 6 and that are strongly dependent on Swing-based model and nodes. `TreeModelProvider` can be used to migrate your code to the new tree repeaters.

In the next example (project *CheckBoxAjaxTree*) we will build a tree that displays some of the main cities of three European countries: Italy, Germany and France. The cities are sub-nodes of a main node representing the relative county. The nodes of the final tree will be also selectable with a checkbox control. The whole tree will have the classic look & feel of Windows XP. This is how our tree will look like:

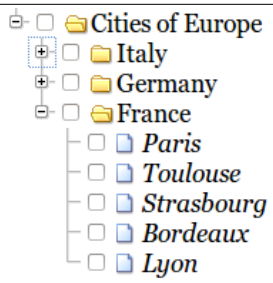


Illustration 16.5: The tree repeater from project CheckBoxAjaxTree

We will start to explore the code of this example from the home page. The first portion of code we will see is where we build the nodes and the `TreeModelProvider` for the three. As tree node we will use Swing class `javax.swing.tree.DefaultMutableTreeNode`:

```
public class HomePage extends WebPage {
    public HomePage(final PageParameters parameters) {
```

54 *Tree model is an implementation of interface javax.swing.tree.TreeModel and it has nothing to do with Wicket models!*

```
    super(parameters);
    DefaultMutableTreeNode root = new DefaultMutableTreeNode("Cities of Europe");

    addNodes(addNodes(root, "Italy"), "Rome", "Venice", "Milan", "Florence");
    addNodes(addNodes(root, "Germany"),"Stuttgart","Munich","Berlin","Dusseldorf", "Dresden");
    addNodes(addNodes(root, "France"), "Paris","Toulouse","Strasbourg","Bordeaux", "Lyon");

    DefaultTreeModel treeModel = new DefaultTreeModel(root);
    TreeModelProvider<DefaultMutableTreeNode> modelProvider = new
                                TreeModelProvider<DefaultMutableTreeNode>(treeModel) {
        @Override
        public IModel<DefaultMutableTreeNode> model(DefaultMutableTreeNode object) {
           return Model.of(object);
       }
    };
    //To be continued...
```

Nodes have been built using simple strings as data objects and invoking custom utility method `addNodes` which converts string parameters into children nodes for a given parent node. Once we have our tree of `DefaultMutableTreeNode`s we can build the Swing tree model (`DefaultTreeModel`) that will be the backing object for a `TreeModelProvider`. This provider wraps each node in a model invoking its abstract method `model`. In our example we have used a simple `Model` as wrapper model.

Scrolling down the code we can see how the tree component is instantiated and configured before being added to the home page:

```
 //Continued from previous snippet...
 NestedTree<DefaultMutableTreeNode> tree = new NestedTree<DefaultMutableTreeNode>("tree",
                                                modelProvider)
  {

   @Override
   protected Component newContentComponent(String id, IModel<DefaultMutableTreeNode>model)
   {
      return new CheckedFolder<DefaultMutableTreeNode>(id, this, model);
   }
  };
  //select Windows theme
  tree.add(new WindowsTheme());

  add(tree);
  }
  //implementation of addNodes
  //...
}
```

To use tree repeaters we must implement their abstract method `newContentComponent` which is called internally by base class `AbstractTree` when a new node must be built. As content component we have used built-in class `CheckedFolder` which combines a `Folder` component with a `CheckBox` form control.

The final step before adding the tree to its page is to apply a theme to it. Wicket comes with two behaviors, `WindowsTheme` and `HumanTheme`, which correspond to the classic Windows XP theme and to the Human theme from Ubuntu.

Our checkable tree is finished but our work is not over yet because the component doesn't offer many functionalities as it is. Unfortunately neither `NestedTree` nor `CheckedFolder` provide a means for collecting checked nodes and returning them to client code. It's up to us to implement a way to keep track of checked nodes.

Another nice feature we would like to implement for our tree is the following user-friendly behavior that should occur when a user checks/unchecks a node:

- When a node is checked also all its children nodes (if any) must be checked. We must also ensure that all the ancestors of the checked node (root included) are checked, otherwise we would get an inconsistent selection.
- When a node is unchecked also all its children nodes (if any) must be unchecked and we must also ensure that ancestors get unchecked if they have no more checked children.

The first goal (keeping track of checked node) can be accomplished building a custom version of `CheckedFolder` that uses a shared Java `Set` to store checked node and to verify if its node has been checked. This kind of solution requires a custom model for checkbox component in order to reflect its checked status when its container node is rendered. This model must implement typed interface `IModel<Boolean>` and must be returned by `CheckedFolder`'s method `newCheckBoxModel`.

For the second goal (auto select/unselect children and ancestor nodes) we can use `CheckedFolder`'s callback method `onUpdate(AjaxRequestTarget)` that is invoked after a checkbox is clicked and its value has been updated. Overriding this method we can handle user click adding/removing nodes to/from the Java `Set`.

Following this implementation plan we can start coding our custom `CheckedFolder` (named `AutocheckedFolder`):

```
public class AutocheckedFolder<T> extends CheckedFolder<T> {

   private ITreeProvider<T> treeProvider;
   private IModel<Set<T>> checkedNodes;
   private IModel<Boolean> checkboxModel;

   public AutocheckedFolder(String id, AbstractTree<T> tree,
                       IModel<T> model, IModel<Set<T>> checkedNodes) {
      super(id, tree, model);
      this.treeProvider = tree.getProvider();
      this.checkedNodes = checkedNodes;
   }

   @Override
   protected IModel<Boolean> newCheckBoxModel(IModel<T> model) {
      checkboxModel =  new CheckModel();
      return checkboxModel;
   }

   @Override
   protected void onUpdate(AjaxRequestTarget target) {
      super.onUpdate(target);
      T node = getModelObject();
      boolean nodeChecked = checkboxModel.getObject();

      addRemoveSubNodes(node, nodeChecked);
      addRemoveAncestorNodes(node, nodeChecked);
```

```
    }

  class CheckModel extends AbstractCheckBoxModel{
      @Override
      public boolean isSelected() {
         return checkedNodes.getObject().contains(getModelObject());
      }

      @Override
      public void select() {
         checkedNodes.getObject().add(getModelObject());
      }

      @Override
      public void unselect() {
         checkedNodes.getObject().remove(getModelObject());
      }
  }
}
```

The constructor of this new component takes in input a further parameter which is the set containing checked nodes.

Class `CheckModel` is the custom model we have implemented for checkbox control. As base class for this model we have used `AbstractCheckBoxModel` which is provided to implement custom models for checkbox controls.

Methods `addRemoveSubNodes` and `addRemoveAncestorNodes` are called to automatically add/remove children and ancestor nodes to/from the current `Set`. Their implementation is mainly focused on the navigation of tree nodes and it heavily depends on the internal implementation of the tree, so we won't dwell on their code.

Now we are just one step away from completing our tree as we still have to find a way to update the checked status of both children and ancestors nodes on client side. Although we could easily accomplish this task by simply refreshing the whole tree via AJAX, we would like to find a better and more performant solution for this task.

When we modify the checked status of a node we don't expand/collapse any node of the three so we can simply update the desired checkboxes rather than updating the entire tree component. This alternative approach could lead to a more responsive interface and to a strong reduction of bandwidth consumption.

With the help of JQuery we can code a couple of JavaScript functions that can be used to check/uncheck all the children and ancestors of a given node. Then, we can append these functions to the current `AjaxRequest` at the end of method `onUpdate`:

```
  @Override
  protected void onUpdate(AjaxRequestTarget target) {
     super.onUpdate(target);
     T node = getModelObject();
     boolean nodeChecked = checkboxModel.getObject();

     addRemoveSubNodes(node, nodeChecked);
     addRemoveAncestorNodes(node, nodeChecked);
```

```
        updateNodeOnClientSide(target, nodeChecked);
    }

    protected void updateNodeOnClientSide(AjaxRequestTarget target,
                        boolean nodeChecked) {
        target.appendJavaScript(";CheckAncestorsAndChildren.checkChildren('" + getMarkupId() +
                                "'," + nodeChecked + ");");

        target.appendJavaScript(";CheckAncestorsAndChildren.checkAncestors('" + getMarkupId() +
                                "'," + nodeChecked + ");");
    }
```

The JavaScript code can be found inside file `autocheckedFolder.js` which is added to the header section as package resource:

```
@Override
public void renderHead(IHeaderResponse response) {
        PackageResourceReference scriptFile = new PackageResourceReference(this.getClass(),
                                                    "autocheckedFolder.js");
        response.render(JavaScriptHeaderItem.forReference(scriptFile));
}
```

16.2.8 Working with hidden components

When a component is not visible its markup and the related id attribute are not rendered in the final page, hence it can not be updated via AJAX. To overcome this problem we must use `Component`'s method `setOutputMarkupPlaceholderTag(true)` which has the effect of rendering a hidden `<span>` tag containing the markup id of the hidden component:

```
final Label label = new Label("labelComponent", "Initial value.");
//make label invisible
label.setVisible(false);
//ensure that label will leave a placeholder for its markup id
label.setOutputMarkupPlaceholderTag(true);
add(label);
//...
new AjaxLink("ajaxLink"){
        @Override
        public void onClick(AjaxRequestTarget target) {
            //turn label to visible
            label.setVisible(true);
            target.add(label);
        }
};
```

Please note that in the code above we didn't invoked method `setOutputMarkupId(true)` as `setOutputMarkupPlaceholderTag` already does it internally.

16.3 Built-in AJAX *behaviors*

In addition to specific components, Wicket offers also a set of built in AJAX behaviors that can be used to easily add AJAX functionalities to existing components. As we will see in this paragraph AJAX behaviors can be used also to ajaxify components that weren't initially designed to work with this

technology. All the following behaviors are inside package `org.apache.wicket.ajax`.

16.3.1 AjaxEventBehavior

`AjaxEventBehavior` allows to handle a JavaScript event (like click, change, etc...) on server side via AJAX. Its constructor takes in input the name of the event that must be handled. Every time this event is fired for a given component on client side, the callback method `onEvent(AjaxRequestTarget target)` is executed. `onEvent` is abstract, hence we must implement it to tell `AjaxEventBehavior` what to do when the specified event occurs.

In project *AjaxEventBehaviorExample* we used this behavior to build a "clickable" `Label` component that counts the number of clicks. Here is the code from the home page of the project:

Html:

```
<body>
  <div wicket:id="clickCounterLabel"></div>
  User has clicked <span wicket:id="clickCounter"></span> time/s on the label above.
</body>
```

Java code:

```
public class HomePage extends WebPage {
   public HomePage(final PageParameters parameters) {
      super(parameters);

      final ClickCounterLabel clickCounterLabel =
         new ClickCounterLabel("clickCounterLabel", "Click on me!");
      final Label clickCounter =
         new Label("clickCounter", new PropertyModel(clickCounterLabel, "clickCounter"));


      clickCounterLabel.setOutputMarkupId(true);
      clickCounterLabel.add(new AjaxEventBehavior("click"){

         @Override
         protected void onEvent(AjaxRequestTarget target) {
            clickCounterLabel.clickCounter++;
            target.add(clickCounter);
         }
      });

      add(clickCounterLabel);
      add(clickCounter.setOutputMarkupId(true));
    }
}

class ClickCounterLabel extends Label{
   public int clickCounter;

   public ClickCounterLabel(String id) {
      super(id);
   }

   public ClickCounterLabel(String id, IModel<?> model) {
```

```
        super(id, model);
    }

    public ClickCounterLabel(String id, String label) {
        super(id, label);
    }
}
```

In the code above we have declared a custom label class named `ClickCounterLabel` that exposes a public integer field called `clickCounter`. Then, in the home page we have attached a `AjaxEventBehavior` to our custom label to increment `clickCounter` every time it receives a click event.

The number of clicks is displayed with another standard label named `clickCounter`.

16.3.2 AjaxFormSubmitBehavior

This behavior allows to send a form via AJAX when the component it is attached to receives the specified event. The component doesn't need to be inside the form if we use the constructor version that, in addition to the name of the event, takes in input also the target form:

```
Form form = new Form("form");
Button submitButton = new Button("submitButton");
//submit form when button is clicked
submitButton.add(new AjaxFormSubmitBehavior(form, "click"){});
add(form);
add(submitButton);
```

16.3.3 AjaxFormComponentUpdatingBehavior

This behavior updates the model of the form component it is attached to when a given event occurs. The standard form submitting process is skipped and the behavior validates only its form component.

The behavior doesn't work with radio buttons and checkboxes. For these kinds of components we must use `AjaxFormChoiceComponentUpdatingBehavior`:

```
Form form = new Form("form");
TextField textField = new TextField("textField", Model.of(""));
//update the model of the text field each time event "change" occurs
textField.add(new AjaxFormComponentUpdatingBehavior("change"){
        @Override
        protected void onUpdate(AjaxRequestTarget target) {
                //...
        }
});
add(form.add(textField));
```

16.3.4 AbstractAjaxTimerBehavior

`AbstractAjaxTimerBehavior` executes callback method `onTimer(AjaxRequestTarget target)` at a specified interval. The behavior can be stopped and restarted at a later time with methods `stop(AjaxRequestTarget target)` and `restart(AjaxRequestTarget target)`:

```
Label dynamicLabel = new Label("dynamicLabel");
```

```
//trigger an AJAX request every three seconds
dynamicLabel.add(new AbstractAjaxTimerBehavior(Duration.seconds(3)) {
        @Override
        protected void onTimer(AjaxRequestTarget target) {
                //...
        }
});
add(dynamicLabel);
```

16.4 Using an activity indicator

One of the thing we must take care of when we use AJAX is to notify user when an AJAX request is already in progress. This is usually done displaying an animated picture as activity indicator while the AJAX request is running.

Wicket comes with a variant of components `AjaxButton`, `AjaxLink` and `AjaxFallbackLink` that display a default activity indicator during AJAX request processing. These components are respectively `IndicatingAjaxButton`, `IndicatingAjaxLink` and `IndicatingAjaxFallbackLink`.

The default activity indicator used in Wicket can be easily integrated in our components using behavior `AjaxIndicatorAppender` (available in package `org.apache.wicket.extensions.ajax.markup.html`) and implementing the interface `IAjaxIndicatorAware` (in package `org.apache.wicket.ajax`).

`IAjaxIndicatorAware` declares method `getAjaxIndicatorMarkupId()` which returns the id of the markup element used to display the activity indicator. This id can be obtained from the `AjaxIndicatorAppender` behavior that has been added to the current component. The following code snippet summarizes the steps needed to integrate the default activity indicator with an ajaxified component:

```
//1-Implement interface IAjaxIndicatorAware
public class MyComponent extends Component implements IAjaxIndicatorAware {
        //2-Instantiate an AjaxIndicatorAppender
        private AjaxIndicatorAppender indicatorAppender =
                        new AjaxIndicatorAppender();

        public MyComponent(String id, IModel<?> model) {
                super(id, model);
                //3-Add the AjaxIndicatorAppender to the component
                add(indicatorAppender);
        }
        //4-Return the markup id obtained from AjaxIndicatorAppender
        public String getAjaxIndicatorMarkupId() {
                return indicatorAppender.getMarkupId();
        }
//...
}
```

If we need to change the default picture used as activity indicator, we can override method `getIndicatorUrl()` of `AjaxIndicatorAppender` and return the URL to the desired picture.

16.5 Ajax request attributes and call listeners

Starting from version 6.0 Wicket has introduced two entities which allow us to control how an AJAX request is generated on client side and to specify the custom JavaScript code we want to execute during

request handling. These entities are class `AjaxRequestAttributes` and interface `IAjaxCallListener`, both placed in package `org.apache.wicket.ajax.attributes`.
`AjaxRequestAttributes` exposes the attributes used to generate the JavaScript call invoked on client side to start an AJAX request. Each attribute will be passed as a JSON[55] parameter to the JavaScript function `Wicket.Ajax.ajax` which is responsible for sending the concrete AJAX request. Every JSON parameter is identified by a short name. Here is a partial list of the available parameters:

Short name	Description	Default value
u	The callback URL used to serve the AJAX request that will be sent.	
c	The id of the component that wants to start the AJAX call.	
e	A list of event (click, change, etc...) that can trigger the AJAX call.	domready
m	The request method that must be used (GET or POST).	GET
f	The id of the form that must be submitted with the AJAX call.	
mp	If the AJAX call involves the submission of a form, this flag indicates whether the data must be encoded using the encoding mode “multipart/form-data”.	`false`
sc	The input name of the submitting component of the form	
async	A boolean parameter that indicates if the AJAX call is asynchronous (`true`) or not.	`true`
wr	Specifies the type of data returned by the AJAX call (XML, HTML, JSON, etc...).	XML
bh, pre, bsh, ah, sh, fh, coh	This is a list of the listeners that are executed on client side (they are JavaScript scripts) during the lifecycle of an AJAX request. Each short name is the abbreviation of one of the methods defined in the interface `IAjaxCallListener` (see below).	An empty list

Note

A full list of the available request parameters as well as more details on the related JavaScript code can be found at https://cwiki.apache.org/confluence/display/WICKET/Wicket+Ajax.

Parameters 'u' (callback URL) and 'c' (the id of the component) are generated by the AJAX behavior that will serve the AJAX call and they are not accessible through `AjaxRequestAttributes`.
Here is the final AJAX function generate for the behavior used in example project *AjaxEventBehavior Example:*

```
Wicket.Ajax.ajax({"u":"./?0-1.IBehaviorListener.0-clickCounterLabel","e":"click",
                  "c":"clickCounterLabel1"});
```

Even if most of the times we will let Wicket generate request attributes for us, both AJAX components and behaviors give us the chance to modify them overriding their method `updateAjaxAttributes(AjaxRequestAttributes attributes)`.
One of the attribute we may need to modify is the list of `IAjaxCallListener`s returned by method

55 *http://en.wikipedia.org/wiki/JSON*

`getAjaxCallListeners()`.

`IAjaxCallListener` defines a set of methods which return the JavaScript code (as a `CharSequence`) that must be executed on client side when the AJAX request handling reaches a given stage:

- **getBeforeHandler(Component)**: returns the JavaScript code that will be executed before any other handlers returned by `IAjaxCallListener`.
  The code is executed in a scope where it can use variable `attrs`, which is an array containing the JSON parameters passed to `Wicket.Ajax.ajax`.
- **getPrecondition(Component)**: returns the JavaScript code that will be used as precondition for the AJAX call. If the script returns false then neither the Ajax call nor the other handlers will be executed.
  The code is executed in a scope where it can use variable `attrs`, which is the same variable seen for `getBeforeHandler`.
- **getBeforeSendHandler(Component)**: returns the JavaScript code that will be executed just before the AJAX call is performed.
  The code is executed in a scope where it can use variables `attrs`, `jqXHR` and `settings`:
  - `attrs` is the same variable seen for `getBeforeHandler`.
  - `jqXHR` is the the jQuery XMLHttpRequest object used to make the AJAX call.
  - `settings` contains the settings used for calling jQuery.ajax().
- **getAfterHandler(Component)**: returns the JavaScript code that will be executed after the AJAX call.
  The code is executed in a scope where it can use variable `attrs`, which is the same variable seen before for `getBeforeHandler`.
- **getSuccessHandler(Component)**: returns the JavaScript code that will be executed if the AJAX call has successfully returned.
  The code is executed in a scope where it can use variables `attrs`, `jqXHR`, `data` and `textStatus`:
  - `attrs` and `jqXHR` are same variables seen for `getBeforeSendHandler`:
  - `data` is the data returned by the AJAX call. Its type depends on parameter `wr` (Wicket AJAX response).
  - `textStatus` it's the status returned as text.
- **getFailureHandler(Component)**: returns the JavaScript code that will be executed if the AJAX call has returned with a failure.
  The code is executed in a scope where it can use variable `attrs`, which is the same variable seen for `getBeforeHandler`.
- **getCompleteHandler(Component)**: returns the JavaScript that will be invoked after success or failure handler has been executed.
  The code is executed in a scope where it can use variables `attrs`, `jqXHR` and `textStatus` which are the same variables seen for `getSuccessHandler`.

In the next paragraph we will see an example of custom `IAjaxCallListener` designed to disable a component during AJAX request processing.

16.6 Creating custom AJAX call listener

Displaying an activity indicator is a nice way to notify user that an AJAX request is already running, but sometimes is not enough. In some situations we may need to completely disable a component during AJAX request processing, for example when we want to avoid that impatient users submit a form multiple times. In this paragraph we will see how to accomplish this goal building a custom and reusable `IAjaxCallListener`. The code used in this example is from project *CustomAjaxListenerExample*.

16.6.1 What we want for our listener

The listener should execute some JavaScript code to disable a given component when the component it is attached to is about to make an AJAX call. Then, when the AJAX request has been completed, the listener should bring back the disabled component to an active state.

When a component is disabled it must be clear to user that an AJAX request is running and that he/she must wait for it to complete. To achieve this result we want to disable a given component covering it with a semi-transparent overlay area with an activity indicator in the middle.

The final result will look like this:

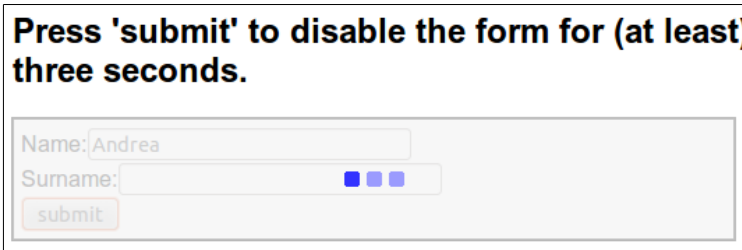


Illustration 16.6: The overlay area on a form (from project CustomAjaxBehaviorExample)

16.6.2 How to implement the listener

The listener will implement methods `getBeforeHandler` and `getAfterHandler`: the first will return the code needed to place an overlay `<div>` on the desired component while the second must remove this overlay when the AJAX call has completed.

To move and resize the overlay area we will use another module from JQueryUI library that allows us to position DOM elements on our page relative to another element[56].

So our listener will depend on four static resources: the JQuery library, the position module of JQuery UI, the custom code used to move the overlay `<div>` and the picture used as activity indicator. Except for the activity indicator, all these resources must be added to page header section in order to be used.

Ajax call listeners can contribute to header section by simply implementing interface `IComponentAwareHeaderContributor`. Wicket provides adapter class `AjaxCallListener` that implements both `IAjaxCallListener` and `IComponentAwareHeaderContributor`. We will use this class as base class for our listener.

16.6.3 JavaScript code

Now that we know what to do on the Java side, let's have a look at the custom JavaScript code that must be returned by our listener (file moveHiderAndIndicator.js):

```
DisableComponentListener = {
   disableElement: function(elementId, activeIconUrl){
      var hiderId = elementId + "-disable-layer";
      var indicatorId = elementId + "-indicator-picture";

      elementId = "#" + elementId;
      //create the overlay <div>
      $(elementId).after('<div id="' + hiderId
         + '" style="position:absolute;">'
         + '<img id="' + indicatorId +  '" src="' + activeIconUrl + '"/>'
         + '</div>');

      hiderId = "#" + hiderId;
```

56 http://jqueryui.com/position/

```
        //set the style properties of the overlay <div>
        $(hiderId).css('opacity', '0.8');
        $(hiderId).css('text-align', 'center');
        $(hiderId).css('background-color', 'WhiteSmoke');
        $(hiderId).css('border', '1px solid DarkGray');
        //set the dimention of the overlay <div>
        $(hiderId).width($(elementId).outerWidth());
        $(hiderId).height($(elementId).outerHeight());
        //positioning the overlay <div> on the component that must be disabled.
        $(hiderId).position({of: $(elementId),at: 'top left', my: 'top left'});

        //positioning the activity indicator in the middle of the overlay <div>
        $("#" + indicatorId).position({of: $(hiderId), at: 'center center',
                                       my: 'center center'});
    },
    //function hideComponent
```

Function `DisableComponentListener.disableElement` places the overlay `<div>` an the activity indicator on the desired component. The parameters in input are the markup id of the component we want to disable and the URL of the activity indicator picture. These two parameters must be provided by our custom listener.

The rest of custom JavaScript contains function `DisableComponentListener.hideComponent` which is just a wrapper around the JQuery function `remove()`:

```
hideComponent: function(elementId){
        var hiderId = elementId + "-disable-layer";
        $('#' + hiderId).remove();
        }
};
```

16.6.4 Class code

The code of our custom listener is the following:

```
public class DisableComponentListener extends AjaxCallListener {
   private static PackageResourceReference customScriptReference = new
   PackageResourceReference(DisableComponentListener.class,"moveHiderAndIndicator.js");

   private static PackageResourceReference jqueryUiPositionRef = new
   PackageResourceReference(DisableComponentListener.class, "jquery-ui-position.min.js");

   private static PackageResourceReference indicatorReference =
         new PackageResourceReference(DisableComponentListener.class, "ajax-loader.gif");

   private Component targetComponent;

   public DisableComponentListener(Component targetComponent){
      this.targetComponent = targetComponent;
   }

   @Override
   public CharSequence getBeforeHandler(Component component) {
      CharSequence indicatorUrl = getIndicatorUrl(component);
      return ";DisableComponentListener.disableElement('" + targetComponent.getMarkupId()
```

```
                                                    + "'," + "'" + indicatorUrl + "');";
    }

    @Override
    public CharSequence getCompleteHandler(Component component) {
        return ";DisableComponentListener.hideComponent('" + targetComponent.getMarkupId() +
                                                                                    "');";
    }

    protected CharSequence getIndicatorUrl(Component component) {
        return component.urlFor(indicatorReference, null);
    }

    @Override
    public void renderHead(Component component, IHeaderResponse response) {
        ResourceReference jqueryReference = Application.get().getJavaScriptLibrarySettings().
                                        getJQueryReference();
        response.render(JavaScriptHeaderItem.forReference(jqueryReference));
        response.render(JavaScriptHeaderItem.forReference(jqueryUiPositionRef));
        response.render(JavaScriptHeaderItem.forReference(customScriptReference));
    }
}
```

As you can see in the code above we have created a function (`getIndicatorUrl`) to retrieve the URL of the indicator picture. This was done in order to make the picture customizable by overriding this method.

Once we have our listener in place, we can finally use it in our example overwriting method `updateAjaxAttributes` of the AJAX button that submits the form:

```
//...
new AjaxButton("ajaxButton"){
        @Override
        protected void updateAjaxAttributes(AjaxRequestAttributes attributes) {
          super.updateAjaxAttributes(attributes);
          attributes.getAjaxCallListeners().add(new DisableComponentListener(form));
        }
}
//...
```

16.6.5 Global listeners

So far we have seen how to use an AJAX call listener to track the AJAX activity of a single component. In addition to these kinds of listeners, Wicket provides also *global* listeners which are triggered for any AJAX request sent from a page.

Global AJAX call events are handled with JavaScript. We can register a callback function for a specific event of the AJAX call lifecycle with function `Wicket.Event.subscribe('<eventName>', <callbackFunction>)`. The first parameter of this function is the name of the event we want to handle. The possible names are:

- `'/ajax/call/before'`: called before any other event handler.
- `'/ajax/call/beforeSend'`: called just before the AJAX call.
- `'/ajax/call/after'`: called after the AJAX request has been sent.
- `'/ajax/call/success'`: called if the AJAX call has successfully returned.

- `'/ajax/call/failure'`: called if the AJAX call has returned with a failure.
- `'/ajax/call/complete'`: called when the AJAX call has completed.
- `'/dom/node/removing'`: called when a component is about to be removed via AJAX. This happens when component markup is updated via AJAX (i.e. the component itself or one of its containers has been added to `AjaxRequestTarget`)
- `'/dom/node/added'`: called when a component has been added via AJAX. Just like `'/dom/node/removing'`, this event is triggered when a component is added to `AjaxRequestTarget`.

The callback function takes in input the following parameters: `attrs`, `jqXHR`, `textStatus`, `jqEvent` and `errorThrown`. The first three parameters are the same seen before with `IAjaxCallListener` while `jqEvent` is an event internally fired by Wicket. The last parameter `errorThrown` indicates if an error has occurred during the AJAX call.

To see a basic example of use of a global AJAX call listener, let's go back to our custom datepicker created in chapter 14. When we built it we didn't think about a possible use of the component with AJAX. When a complex component like our datepicker is refreshed via AJAX, the following two side effects can occur:

- After been refreshed, the component loses every JavaScript handler set on it. This is not a problem for our datepicker as it sets a new JQuery datepicker every time is rendered (inside method `renderHead`).
- The markup previously created with JavaScript is not removed. For our datepicker this means that the icon used to open the calendar won't be removed while a new one will be added each time the component is refreshed.

To solve the second unwanted side effect we can register a global AJAX call listener that completely removes the datepicker functionality from our component before it is removed due to an AJAX refresh (which fires event `'/dom/node/removing'`).

Project CustomDatepickerAjax contains a new version of our datepicker which adds to its JavaScript file JQDatePicker.js the code needed to register a callback function that gets rid of the JQuery datepicker before the component is removed from the DOM:

```
Wicket.Event.subscribe('/dom/node/removing',
     function(jqEvent, attributes, jqXHR, errorThrown, textStatus) {
        var componentId = '#' + attributes['id'];
        if($(componentId).datepicker !== undefined)
              $(componentId).datepicker('destroy');
     }
);
```

The code above retrieves the id of the component that is about to be removed using parameter `attributes`. Then it checks if a JQuery datepicker was defined for the given component and if so, it removes the widget calling function `destroy`.

16.7 Summary

AJAX is another example of how Wicket can simplify web technologies providing a good component and object oriented abstraction of them.

In this chapter we have seen how to take advantage of the AJAX support provided by Wicket to write AJAX-enhanced applications. Most of the chapter has been dedicated to the built-in components and behaviors that let us adopt AJAX without almost any effort.

In the final part of the chapter we have seen how Wicket physically implements an AJAX call on client

side using AJAX request attributes. Then, we have learnt how to use call listeners to execute custom JavaScript during AJAX request lifecycle.

17 Working with WebSocket

WebSocket is a new standard technology defined by HTML 5. It allows to establish a *full-duplex* (or *bidirectional*) communication channel between server and a single web page (using a single TCP connection).

17.1 Native WebSocket support

17.2 Using WebSocket with Atmosphere

17.3 An outlook to JSR 356

18 Integration with enterprise containers

Writing a web application is not just about producing a good layout and a bunch of “cool” pages. We must also integrate our presentation code with enterprise resources like data sources, message queues, business objects, etc...

The first decade of 2000s has seen the rising of new frameworks (like Spring[57]) and new specifications (like EJB 3.1[58]) aimed to simplify the management of enterprise resources and (among other things) their integration with presentation code.

All these new technologies are based on the concepts of *container* and *dependency injection*. *Container* is the environment where our enterprise resources are created and configured while *dependency injection*[59] is a pattern implemented by containers to *inject* into an object the resources it depends on.

Wicket can be easily integrated with enterprise containers using *component instantiation listeners*. These entities are instances of interface `org.apache.wicket.application.IComponentInstantiationListener` and can be registered during application's initialization. `IComponentInstantiationListener` defines callback method `onInstantiation(Component component)` which can be used to provide custom instantiation logic for Wicket components.

Wicket distribution and project WicketStuff[60] already provide a set of built-in listeners to integrate our applications with EJB 3.1 compliant containers (like JBoss Seam[61]) or with some of the most popular enterprise frameworks like Guice[62] or Spring.

In this chapter we will see two basic examples of injecting a container-defined object into a page using first an implementation of the EJB 3.1 specifications (project OpenEJB[63]) and then using Spring.

18.1 Integrating Wicket with EJB

WicketStuff provides a module called `wicketstuff-javaee-inject` that contains component instantiation listener `JavaEEComponentInjector`. If we register this listener in our application we can use standard EJB annotations to inject dependencies into our Wicket components.

To register a component instantiation listener in Wicket we must use `Application`'s method `getComponentInstantiationListeners` which returns a typed collection of `IComponentInstantiationListener`s.

The following initialization code is taken from project *EjbInjectionExample*:

```
public class WicketApplication extends WebApplication
{
        //Constructor...

        @Override
        public void init()
```

57 http://www.springsource.org/
58 http://en.wikipedia.org/wiki/Enterprise_JavaBeans
59 http://en.wikipedia.org/wiki/Dependency_Injection
60 https://github.com/wicketstuff *An overview of WicketStuff project is available in Appendix B*
61 *Seam has its own integration module for Wicket, but it is stuck on Wicket version 1.4*
62 http://code.google.com/p/google-guice/
63 http://openejb.apache.org/

```
        {
                super.init();
                getComponentInstantiationListeners().add(new JavaEEComponentInjector(this));

        }
}
```

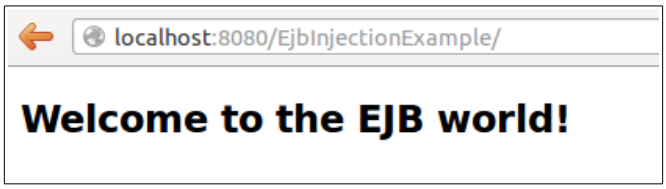
In this example the object that we want to inject is a simple class containing a greeting message:

```
@ManagedBean
public class EnterpriseMessage {
        public String message = "Welcome to the EJB world!";
}
```

Please note that we have used annotation `@ManagedBean` to decorate our object. Now to inject it into the home page we must add a field of type `EnterpriseMessage` and annotate it with annotation `@EJB`:

```
public class HomePage extends WebPage {

        @EJB
        private EnterpriseMessage enterpriseMessage;
        //getter and setter for enterpriseMessage...

        public HomePage(final PageParameters parameters) {
                super(parameters);

                add(new Label("message", enterpriseMessage.message));
        }
}
```

That is all. We can point the browser to the home page of the project and see the greeting message injected into the page:



18.2 Integrating Wicket with Spring

If we need to inject dependencies with Spring we can use listener `org.apache.wicket.spring.injection.annot.SpringComponentInjector` provided by module `wicket-spring`.

For the sake of simplicity in the example project *SpringInjectionExample* we have used Spring class `AnnotationConfigApplicationContext` to avoid any XML file and create a Spring context directly from code:

```
public class WicketApplication extends WebApplication
{
  //Constructor...
```

```
  @Override
  public void init()
  {
    super.init();

    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
    //Scan package for annotated beans
    ctx.scan("org.wicketTutorial.ejbBean");
    ctx.refresh();

    getComponentInstantiationListeners().add(new SpringComponentInjector(this, ctx));
  }
}
```

As we can see in the code above, the constructor of `SpringComponentInjector` takes in input also an instance of Spring context.

The injected object is the same used in the previous project *EjbInjectionExample*, it differs only for the greeting message:

```
@ManagedBean
public class EnterpriseMessage {
       public String message = "Welcome to the Spring world!";
}
```

In the home page of the project the object is injected using Wicket annotation `@SpringBean`:

```
public class HomePage extends WebPage {
  @SpringBean
  private EnterpriseMessage enterpriseMessage;
  //getter and setter for enterpriseMessage...

  public HomePage(final PageParameters parameters) {
       super(parameters);

       add(new Label("message", enterpriseMessage.message));
  }
}
```

By default `@SpringBean` searches into Spring context for a bean having the same type of the annotated field. If we want we can specify also the name of the bean to use as injected object and we can declare if the dependency is required or not. By default dependencies are required and if they can not be resolved to a compatible bean, Wicket will throw an `IllegalStateException`:

```
  //set the dependency as not required, i.e the field can be left null
  @SpringBean(name="anotherName", required=false)
  private EnterpriseMessage enterpriseMessage;
```

18.3 JSR-330[64] annotations

Spring (and Guice) users can use standard JSR-330 annotations to wire their dependencies. This will make their code more interoperable with other containers that support this standard:

64 http://jcp.org/en/jsr/detail?id=330

```
//inject a bean specifying its name with JSR-330 annotations
@Inject @Named("anotherName")
private EnterpriseMessage enterpriseMessage;
```

18.4 Summary

In this chapter we have seen how to integrate Wicket applications with Spring and with an EJB container. Module `wicket-examples` contains also an example of integration with Guice (see application class `org.apache.wicket.examples.guice.GuiceApplication`).

19 Security with Wicket

Security is one of the most important non-functional requirements we must implement in our applications. This is particularly true for enterprise applications as they usually support multiple concurrent users, and therefore they need to have an access control policy.

In this chapter we will explore the security infrastructure provided by Wicket and we will learn how to use it to implement authentication and authorizations in our web applications.

19.1 Authentication

The first step in implementing a security policy is assigning a trusted identity to our users, which means that we must *authenticate* them. Web applications usually adopt a form-based authentication with a login form that asks user for a unique username and the relative password:

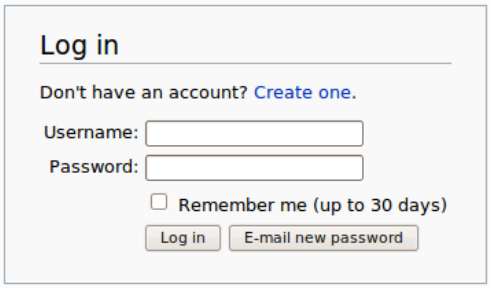


Illustration 19.1: Login form from Wikipedia

Wicket supports form-based authentication with session class `AuthenticatedWebSession` and application class `AuthenticatedWebApplication`, both placed inside package `org.apache.wicket.authroles.authentication`.

19.1.1 AuthenticatedWebSession

Class `AuthenticatedWebSession` comes with the following set of public methods to manage user authentication:

- **authenticate(String username, String password)**: this is an abstract method that must be implemented by every subclass of `AuthenticatedWebSession`. It should contain the actual code that checks for user's identity. It returns a boolean value which is `true` if authentication has succeeded or `false` otherwise.
- **signIn(String username, String password)**: this method internally calls `authenticate` and set the flag `signedIn` to `true` if authentication succeeds.
- **isSignedIn()**:getter method for flag `signedIn`.
- **signOut()**: sets the flag `signedIn` to `false`.
- **invalidate()**: calls `signOut` and invalidates session.

Warning

Remember that `signOut` does not discard any session-relative data. If we want to get rid of these data, we must invoke method `invalidate` instead of `signOut`.

Another abstract method we must implement when we use `AuthenticatedWebSession` is `getRoles` which is inherited from parent class `AbstractAuthenticatedWebSession`. This method can be ignored for now as it will be discussed later when we will talk about role-based authorization.

19.1.2 AuthenticatedWebApplication

Class `AuthenticatedWebApplication` provides the following methods to support form-based authentication:

- **getWebSessionClass()**: abstract method that returns the session class to use for this application. The returned class must be a subclass of `AbstractAuthenticatedWebSession`.
- **getSignInPageClass()**: abstract method that returns the page to use as sign in page when a user must be authenticated.
- **restartResponseAtSignInPage()**: forces the current response to restart at the sign in page. After we have used this method to redirect a user, we can make her/him return to the original page calling `Componet`'s method `continueToOriginalDestination()`.

The other methods implemented inside `AuthenticatedWebApplication` will be introduced when we will talk about authorizations.

19.1.3 A basic example of authentication.

Project *BasicAuthenticationExample* is a basic example of form-based authentication implemented with classes `AuthenticatedWebSession` and `AuthenticatedWebApplication`.

The homepage of the project contains only a link to page `AuthenticatedPage` which can be accessed only if user is signed in. The code of `AuthenticatedPage` is this following:

```
public class AuthenticatedPage extends WebPage {
   @Override
   protected void onConfigure() {
      AuthenticatedWebApplication app = (AuthenticatedWebApplication)Application.get();
      //if user is not signed in, redirect him to sign in page
      if(!AuthenticatedWebSession.get().isSignedIn())
         app.restartResponseAtSignInPage();
   }

   @Override
   protected void onInitialize() {
      super.onInitialize();
      add(new Link("goToHomePage") {

         @Override
         public void onClick() {
            setResponsePage(getApplication().getHomePage());
         }
      });

      add(new Link("logOut") {

         @Override
         public void onClick() {
            AuthenticatedWebSession.get().invalidate();
            setResponsePage(getApplication().getHomePage());
         }
```

```
        });
    }
}
```

Page `AuthenticatedPage` checks inside `onConfigure` if user is signed in and if not, it redirects her/him to the sign in page with method `restartResponseAtSignInPage`. The page contains also a link to the homepage and another link that signs out user.

The sign in page is implemented in class `SignInPage` and contains the form used to authenticate users:

```
public class SignInPage extends WebPage {
    private String username;
    private String password;

    @Override
    protected void onInitialize() {
        super.onInitialize();

        StatelessForm form = new StatelessForm("form"){
            @Override
            protected void onSubmit() {
                if(Strings.isEmpty(username))
                    return;

                boolean authResult = AuthenticatedWebSession.get().signIn(username, password);
                //if authentication succeeds redirect user to the requested page
                if(authResult)
                    continueToOriginalDestination();
            }
        };

        form.setDefaultModel(new CompoundPropertyModel(this));

        form.add(new TextField("username"));
        form.add(new PasswordTextField("password"));

        add(form);
    }
}
```

The form is responsible for handling user authentication inside its method `onSubmit`. The username and password are passed to `AuthenticatedWebSession`'s method `signIn(username, password)` and if authentication succeeds, the user is redirected to the original page with method `continueToOriginalDestination`.

The session class and the application class used in the project are reported here:

Session class:

```
public class BasicAuthenticationSession extends AuthenticatedWebSession {

        public BasicAuthenticationSession(Request request) {
                super(request);
        }
```

```
        @Override
        public boolean authenticate(String username, String password) {
              //user is authenticated if both username and password are equal to 'wicketer'
               return username.equals(password) && username.equals("wicketer");
        }

        @Override
        public Roles getRoles() {
               return null;
        }
}
```

Application class:

```
public class WicketApplication extends AuthenticatedWebApplication{
        @Override
        public Class<HomePage> getHomePage(){
               return HomePage.class;
        }

        @Override
        protected Class<? extends AbstractAuthenticatedWebSession> getWebSessionClass(){
               return BasicAuthenticationSession.class;
        }

        @Override
        protected Class<? extends WebPage> getSignInPageClass() {
               return SignInPage.class;
        }
}
```

The authentication logic inside `authenticate` has been kept quite trivial in order to make the code as clean as possible. Please note also that session class must have a constructor that accepts an instance of class `Request`.

19.1.4 Redirecting user to an intermediate page

Method `restartResponseAtSignInPage` is an example of redirecting user to an intermediate page before allowing him to access to the requested page. This method internally throws exception `org.apache.wicket.RestartResponseAtInterceptPageException` which saves the URL of the requested page into session metadata and then redirects user to the page passed as constructor parameter (the sign in page).

`Component`'s method `redirectToInterceptPage(Page)` works in much the same way as `restartResponseAtSignInPage` but it allows us to specify which page to use as intermediate page:

```
redirectToInterceptPage(intermediatePage);
```

Note

Since both `restartResponseAtSignInPage` and `redirectToIntercept Page` internally throw an exception, the code placed after them will not be executed.

19.2 Authorizations

The authorization support provided by Wicket is built around the concept of *authorization strategy* which is represented by interface `IAuthorizationStrategy` (in package `org.apache.wicket.authorization`):

```
public interface IAuthorizationStrategy
{
  //interface methods
 <T extends IRequestableComponent> boolean isInstantiationAuthorized(Class<T>
componentClass);
 boolean isActionAuthorized(Component component, Action action);

 //default authorization strategy that allows everything
 public static final IAuthorizationStrategy ALLOW_ALL = new IAuthorizationStrategy()
 {
  @Override
  public <T extends IRequestableComponent> boolean isInstantiationAuthorized(final
Class<T> c)
  {
    return true;
  }
  @Override
  public boolean isActionAuthorized(Component c, Action action)
  {
    return true;
  }
 };
}
```

This interface defines two methods:

- `isInstantiationAuthorized` checks if user is allowed to instantiate a given component.
- `isActionAuthorized` checks if user is authorized to perform a given action on a component's instance. The standard actions checked by this method are defined into class `Action` and are `Action.ENABLE` and `Action.RENDER`.

Inside `IAuthorizationStrategy` we can also find a default implementation of the interface (called *`ALLOW_ALL`*) that allows everyone to instantiate every component and perform every possible action on it. This is the default strategy adopted by class `Application`.

To change the authorization strategy in use we must register the desired implementation into security settings (interface `ISecuritySettings`) during initialization phase with method `setAuthorizationStrategy`:

```
  //Application class code...
  @Override
  public void init()
  {
    super.init();
    getSecuritySettings().setAuthorizationStrategy(myAuthorizationStrategy);
  }
//...
```

If we want to combine the action of two or more authorization strategies we can chain them with

strategy `CompoundAuthorizationStrategy` which implements composite patter for authorization strategies[65].

Most of the times we won't need to implement an `IAuthorizationStrategy` from scratch as Wicket already comes with a set of built-in strategies. In the next paragraphs we will see some of these strategies that can be used to implement an effective and flexible security policy.

19.2.1 SimplePageAuthorizationStrategy

Abstract class `SimplePageAuthorizationStrategy` (in package `org.apache.wicket.authorization.strategies.page`) is a strategy that checks user authorizations calling abstract method `isAuthorized` only for those pages that are subclasses of a given supertype. If `isAuthorized` returns `false`, the user is redirected to the sign in page specified as second constructor parameter:

```
SimplePageAuthorizationStrategy authorizationStrategy = new SimplePageAuthorizationStrategy(
                                                PageClassToCheck.class, SignInPage.class)
{
  protected boolean isAuthorized()
  {
    //Authentication code...
  }
};
```

By default `SimplePageAuthorizationStrategy` checks for permissions only on pages. If we want to change this behavior and check also other kinds of components, we must override method `isActionAuthorized` and implement our custom logic inside it.

19.2.2 Role-based strategies

At the end of paragraph 19.1.1 we have introduced `AbstractAuthenticatedWebSession`'s method `getRoles` which is provided to support role-based authorization returning the set of roles granted to the current user.

In Wicket roles are simple strings like “BASIC_USER” or “ADMIN” (they don't need to be capitalized) and they are handled with class `org.apache.wicket.authroles.authorization.strategies.role.Roles`. This class extends standard `HashSet` collection adding some functionalities to check whether the set contains one or more roles. Class `Roles` already defines roles `Roles.USER` and `Roles.ADMIN`.

The session class in the following example returns a custom “SIGNED_IN” role for every authenticated user and it adds an `Roles.ADMIN` role if username is equal to `superuser`:

```
class BasicAuthenticationRolesSession extends AuthenticatedWebSession {
       private String userName;

       public BasicAuthenticationRolesSession(Request request) {
              super(request);
       }

       @Override
       public boolean authenticate(String username, String password) {
              boolean authResult= false;

              authResult = //some authentication logic...
```

65 *On page 49 we have seen the same pattern implemented by RequestCycleListenerCollection for request cycle listeners .*

```
            if(authResult)
                    userName = username;

            return authResult;
        }

        @Override
        public Roles getRoles() {
            Roles resultRoles = new Roles();

            if(isSignedIn())
                    resultRoles.add("SIGNED_IN");

            if(userName.equals("superuser"))
                    resultRoles.add(Roles.ADMIN);

            return resultRoles;
        }
}
```

Roles can be adopted to apply security restrictions on our pages and components. This can be done using one of the two built-in authorization strategies that extend super class `AbstractRoleAuthorizationStrategyWicket`: `MetaDataRoleAuthorizationStrategy` and `AnnotationsRoleAuthorizationStrategy`

The difference between these two strategies is that `MetaDataRoleAuthorizationStrategy` handles role-based authorizations with Wicket metadata while `AnnotationsRoleAuthorization-Strategy` uses Java annotations.

Note

Application class `AuthenticatedWebApplication` already sets `MetaDataRoleAuthorizationStrategy` and `AnnotationsRoleAuthorizationStrategy` as its own authorization strategies (it uses a compound strategy as we will see in paragraph 19.2.4).

The code that we will see in the next examples is for illustrative purpose only. If our application class inherits from `AuthenticatedWebApplication` we won't need to configure anything to use these two strategies.

19.2.2.1 Using roles with metadata

Strategy `MetaDataRoleAuthorizationStrategy` uses application and components metadata to implement role-based authorizations. The class defines a set of static methods `authorize` that can be used to specify which roles are allowed to instantiate a component and which roles can perform a given action on a component.

The following code snippet reports both application and session classes from project *MetaDataRolesStrategyExample* and illustrates how to use `MetaDataRoleAuthorizationStrategy` to allow access to a given page (`AdminOnlyPage`) only to ADMIN role:

Application class:

```
public class WicketApplication extends AuthenticatedWebApplication{
   @Override
   public Class<? extends WebPage> getHomePage(){
```

```
        return HomePage.class;
    }

    @Override
    protected Class<? extends AbstractAuthenticatedWebSession> getWebSessionClass() {
        return BasicAuthenticationSession.class;
    }

    @Override
    protected Class<? extends WebPage> getSignInPageClass() {
        return SignInPage.class;
    }

    @Override
    public void init(){
        getSecuritySettings().setAuthorizationStrategy(new MetaDataRoleAuthorizationStrategy
                                                                                    (this));
        MetaDataRoleAuthorizationStrategy.authorize(AdminOnlyPage.class, Roles.ADMIN);
    }
}
```

Session class:

```
public class BasicAuthenticationSession extends AuthenticatedWebSession {

    private String username;

    public BasicAuthenticationSession(Request request) {
        super(request);
    }

    @Override
    public boolean authenticate(String username, String password) {
        //user is authenticated if username and password are equal
        boolean authResult = username.equals(password);

        if(authResult)
            this.username = username;

        return authResult;
    }

    public Roles getRoles() {
        Roles resultRoles = new Roles();
        //if user is signed in add the relative role
        if(isSignedIn())
            resultRoles.add("SIGNED_IN");
        //if username is equal to 'superuser' add the ADMIN role
        if(username!= null && username.equals("superuser"))
            resultRoles.add(Roles.ADMIN);

        return resultRoles;
    }

    @Override
```

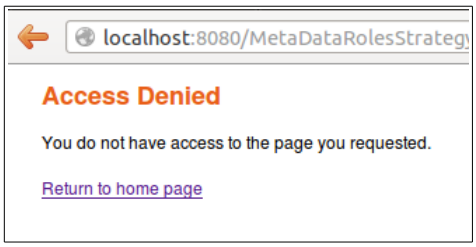
```
    public void signOut() {
        super.signOut();
        username = null;
    }
}
```

The code that instantiates `MetaDataRoleAuthorizationStrategy` and set it as application's strategy is inside application class method `init`.

Any subclass of `AbstractRoleAuthorizationStrategyWicket` needs an implementation of interface `IRoleCheckingStrategy` to be instantiated. For this purpose in the code above we used the application class itself because its base class `AuthenticatedWebApplication` already implements interface `IRoleCheckingStrategy`. By default `AuthenticatedWebApplication` checks for authorizations using the roles returned by the current `AbstractAuthenticatedWebSession`. As final step inside `init` we grant the access to page `AdminOnlyPage` to ADMIN role calling method `authorize`.

The code from session class has three interesting methods. The first is `authenticate` which considers as valid credentials every pair of username and password having the same value. The second notable method is `getRoles` which returns role SIGNED_IN if user is authenticated and it adds role ADMIN if username is equal to `superuser`. Finally, we have method `signOut` which has been overridden in order to clean the `username` field used internally to generate roles.

Now if we run the project and we try to access to `AdminOnlyPage` from the home page without having the ADMIN role, we will be redirected to the default access-denied page used by Wicket:



The access-denied page can be customized using method `setAccessDeniedPage(Class<? extends Page>)` of setting interface `IApplicationSettings`:

```
//Application class code...
@Override
public void init(){
    getApplicationSettings().setAccessDeniedPage(MyCustomAccessDeniedPage.class);
}
```

Just like custom “Page expired” page (see chapter 6.2.5), also custom “Access denied” page must be bookmarkable.

19.2.2.2 Using roles with annotations

Strategy `AnnotationsRoleAuthorizationStrategy` relies on two built-in annotations to handle role-based authorizations. These annotations are `AuthorizeInstantiation` and `AuthorizeAction`. As their names suggest the first annotation specifies which roles are allowed to instantiate the annotated component while the second must be used to indicate which roles are allowed to perform a specific action on the annotated component.

In the following example we use annotations to make a page accessible only to signed-in users and to enable it only if user has the ADMIN role:

```
@AuthorizeInstantiation("SIGNED_IN")
@AuthorizeAction(action = "ENABLE", roles = {"ADMIN"})
public class MyPage extends WebPage {
   //Page class code...
}
```

Remember that when a component is not enabled, user can render it but he can neither click on its links nor interact with its forms.

Example project *AnnotationsRolesStrategyExample* is a revisited version of *MetaDataRolesStrategy Example* where we use `AnnotationsRoleAuthorizationStrategy` as authorization strategy. To ensure that page `AdminOnlyPage` is accessible only to ADMIN role we have used the following annotation:

```
@AuthorizeInstantiation("ADMIN")
public class AdminOnlyPage extends WebPage {
    //Page class code...
}
```

19.2.3 Catching an unauthorized component instantiation

Interface `IUnauthorizedComponentInstantiationListener` (in package `org.apache.wicket.authorization`) is provided to give the chance to handle the case in which a user tries to instantiate a component without having the permissions to do it. The method defined inside this interface is `onUnauthorizedInstantiation(Component)` and it is executed whenever a user attempts to execute an unauthorized instantiation.

This listener must be registered into application's security settings with method `setUnauthorizedComponentInstantiationListener` defined by setting interface `ISecuritySettings`. In the following code snippet we register a listener that redirect user to a warning page if he tries to do a not-allowed instantiation:

```
public class WicketApplication extends AuthenticatedWebApplication{
     //Application code...
     @Override
     public void init(){
        getSecuritySettings().setUnauthorizedComponentInstantiationListener(
                      new IUnauthorizedComponentInstantiationListener() {

           @Override
           public void onUnauthorizedInstantiation(Component component) {
               component.setResponsePage(AuthWarningPage.class);
           }
        });
     }
}
```

In addition to interface `IRoleCheckingStrategy`, class `AuthenticatedWebApplication` implements also `IUnauthorizedComponentInstantiationListener` and registers itself as listener for unauthorized instantiations.

By default `AuthenticatedWebApplication` redirects users to sign-in page if they are not signed-in and they try to instantiate a restricted component. Otherwise, if users are already signed in but they are

not allowed to instantiate a given component, an `UnauthorizedInstantiationException` will be thrown.

19.2.4 Strategy RoleAuthorizationStrategy

Class `RoleAuthorizationStrategy` is a compound strategy that combines both `MetaData RoleAuthorizationStrategy` and `AnnotationsRoleAuthorizationStrategy`.

This is the strategy used internally by `AuthenticatedWebApplication`.

19.3 Using HTTPS protocol

HTTPS is the standard technology adopted on Internet to create a secure communication channel between web applications and their users.

In Wicket we can easily protect our pages with HTTPS mounting a special request mapper called `HttpsMapper` and using annotation `RequireHttps` with those pages we want to serve over this protocol. Both these two entities are in package `org.apache.wicket.protocol.https`.

`HttpsMapper` wraps an existing mapper and redirects incoming requests to HTTPS if the related response must render a page containing annotation `RequireHttps`. Most of the times the wrapped mapper will be the root one, just like we saw before for `CryptoManager` in paragraph 8.6.6.

Another parameter needed to build a `HttpsMapper` is an instance of class `HttpsConfig`. This class allows us to specify which ports must be used for HTTPS and HTTP. By default the port numbers used by these two protocols are respectively 443 and 80.

The following code is taken from project *HttpsProtocolExample* and illustrates how to enable HTTPS in our applications:

```
//Application class code...
@Override
public void init(){
   setRootRequestMapper(new HttpsMapper(getRootRequestMapper(),
                                        new HttpsConfig(8080, 443)));
}
```

Now we can use annotation `RequireHttps` to specify which pages must be served using HTTPS:

```
@RequireHttps
public class HomePage extends WebPage {
    public HomePage(final PageParameters parameters) {
        super(parameters);
    }
}
```

If we want to protect many pages with HTTPS without adding annotation `RequireHttps` to each of them, we can annotate a marker interface or a base page class and implement/extend it in any page we want to make secure:

Marker interface:

```
@RequireHttps
public interface IMarker{
}
```

Base class:

```
@RequireHttps
public class BaseClass extends WebPage{
//Page code...
}
```

Secure page inheriting from `BaseClass`:

```
public class HttpsPage extends BaseClass{
//Page code...
}
```

Secure page implementing `IMarker`:

```
public class HttpsPage implements IMarker{
//Page code...
}
```

19.4 Package Resource Guard

Wicket internally uses an entity called *package resource guard* to protect package resources from external access. This entity is an implementation of interface `org.apache.wicket.markup.html.IPackageResourceGuard`.

By default Wicket applications use as package resource guard class `SecurePackageResourceGuard`, which allows to access only to the following file extensions (grouped by type):

JavaScript files	*.js
Css files	*.css
HTML pages	*.html
Textual files	*.txt
Flash files	*.swf
Picture files	*.png, *.jpg, *.jpeg, *.gif, *.ico, *.cur, *.bmp, *.svg

To modify the set of allowed files formats we can add one or more *patterns* with method `addPattern(String)`. The rules to write a pattern are the following:

- patterns start with either a "+" or a "-". In the first case the pattern will add one or more file to the set while starting a pattern with a “-” we exclude all the files matching the given pattern. For example pattern “-web.xml” excludes all web.xml files in all directories.
- wildcard character “*” is supported as placeholder for zero or more characters. For example pattern “+*.mp4” adds all the mp4 files inside all directories.
- subdirectories are supported as well. For example pattern “+documents/*.pdf” adds all pdf files under “documents” directory. Character “*” can be used with directories to specify a nesting level. For example “+documents/*/*.pdf” adds all pdf files placed one level below “documents” directory.
- a double wildcard character “**” indicates zero or more subdirectories. For example patter “+documents/**/*.pdf” adds all pdf files placed inside “documents” directory or inside any of its subdirectories.

Patterns that allow to access to every file with a given extensions (such as “+*.pdf”) should be always avoided in favour of more restrictive expressions that contain a directory structure:

```
//Application class code...
@Override
public void init()
{
      IPackageResourceGuard packageResourceGuard = application.getResourceSettings()
                                                                .getPackageResourceGuard();
      if (packageResourceGuard instanceof SecurePackageResourceGuard)
      {
         SecurePackageResourceGuard guard = (SecurePackageResourceGuard) packageResourceGuard;
         //Allow to access only to pdf files placed in the “public” directory.
         guard.addPattern("+public/*.pdf");
      }
}
```

19.5 Summary

In this chapter we have seen the components and the mechanisms that allow us to implement security policies in our Wicket-based applications. Wicket comes with an out of the box support for both authorization and authentication.

The central element of authorization mechanism is the interface `IAuthorizationStrategy` which decouples our components from any detail about security strategy. The implementations of this interface must decide if a user is allowed to instantiate a given page or component and if she/he can perform a given action on it.

Wicket natively supports role-based authorizations with strategies `MetaDataRoleAuthorizationStrategy` and `AnnotationsRoleAuthorizationStrategy`. The difference between these two strategies is that the first offers a programmatic approach for role handling while the second promotes a declarative approach using built-in annotations.

After having explored how Wicket internally implements authentication and authorization, in the last part of the chapter we have learnt how to configure our applications to support HTTPS and how to specify which pages must be served over this protocol.

In the last paragraph we have seen how Wicket protects package resources with a guard entity that allows us to decide which package resources can be accessed from users.

20 Test Driven Development with Wicket

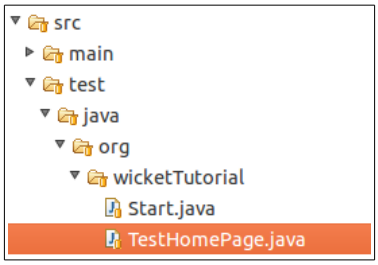
Test Driven Development[66] has become a crucial activity for every modern development methodology. This chapter will cover the built-in support for testing provided by Wicket with its rich set of *helper* and *mock* classes that allows us to test our components and our applications in isolation (i.e without the need for a servlet container) using JUnit, the *de facto* standard for Java unit testing.

In this chapter we will see how to write *unit* tests for our applications and components and we will learn how to use helper classes to simulate user navigation and write *acceptance* tests without the need of any testing framework other than JUnit.

The JUnit version used in this chapter is 4.x.

20.1 Utility class WicketTester

A good way to start getting confident with Wicket unit testing support is looking at the test case class `TestHomePage` that is automatically generated by Maven when we use Wicket archetype to create a new project:



Here is the content of `TestHomePage`:

```
public class TestHomePage{
	private WicketTester tester;

	@Before
	public void setUp(){
		tester = new WicketTester(new WicketApplication());
	}
	@Test
	public void homepageRendersSuccessfully(){
		//start and render the test page
		tester.startPage(HomePage.class);
		//assert rendered page class
		tester.assertRenderedPage(HomePage.class);
	}
}
```

The central class in a Wicket testing is `org.apache.wicket.util.tester.WicketTester`. This

66 http://en.wikipedia.org/wiki/Test-driven_development

utility class provides a set of methods to render a component, click links, check if page contains a given component or a feedback message, and so on.

The basic test case shipped with `TestHomePage` illustrates how `WicketTester` is typically instantiated (inside method `setUp()`). In order to test our components, `WicketTester` needs to use an instance of `WebApplication`. Usually, we will use our application class as `WebApplication`, but we can also decide to build `WicketTester` invoking its no-argument constructor and letting it automatically build a mock web application (an instance of class `org.apache.wicket.mock.MockApplication`).

The code from `TestHomePage` introduces two basic methods to test our pages. The first is method `startPage` that renders a new instance of the given page class and sets it as current rendered page for `WicketTester`. The second method is `assertRenderedPage` which checks if the current rendered page is an instance of the given class. In this way if `TestHomePage` succeeds we are sure that page `HomePage` has been rendered without any problem. The last rendered page can be retrieved with method `getLastRenderedPage`.

That's only a taste of what `WicketTester` can do. In the next paragraphs we will see how it can be used to test every element that composes a Wicket page (links, models, behaviors, etc...).

20.1.1 Testing links

A click on a Wicket link can be simulated with method `clickLink` which takes in input the link component or the page-relative path to it.

To see an example of usage of `clickLink`, let's consider again project *LifeCycleStagesRevisited*. As we know from chapter 5 the home page of the project alternately displays two different labels (“First label” and “Second label”), swapping between them each time button `"reload"` is clicked. The code from its test case checks that label has actually changed after button `"reload"` has been pressed:

```
//...
@Test
public void switchLabelTest(){
        //start and render the test page
        tester.startPage(HomePage.class);
        //assert rendered page class
        tester.assertRenderedPage(HomePage.class);
        //assert rendered label
        tester.assertLabel("label", "First label");
        //simulate a click on "reload" button
        tester.clickLink("reload");
        //assert rendered label
        tester.assertLabel("label", "Second label");
}
//...
```

In the code above we have used `clickLink` to click on the `"reload"` button and force page to be rendered again. In addition, we have used also method `assertLabel` that checks if a given label contains the expected text.

By default `clickLink` assumes that AJAX is enabled on client side. To switch AJAX off we can use another version of this method that takes in input the path to the link component and a boolean flag that indicates if AJAX must be enabled (`true`) or not (`false`).

```
//...
//simulate a click on a button without AJAX support
tester.clickLink("reload", false);
```

```
//...
```

20.1.2 Testing component status

`WicketTester` provides also a set of methods to test the states of a component. They are:

- **assertEnabled(String path)/assertDisabled(String path)**: they test if a component is enabled or not.
- **assertVisible(String path)/assertInvisible(String path)**: they test component visibility.
- **assertRequired(String path)**: checks if a form component is required.

In the test case from project *CustomDatepickerAjax* we used `assertEnabled/assertDisabled` to check if button `"update"` really disables our datepicker:

```
//...
@Test
public void testDisableDatePickerWithButton(){
        //start and render the test page
        tester.startPage(HomePage.class);
        //assert that datepicker is enabled
        tester.assertEnabled("form:datepicker");
        //click on update button to disable datepicker
        tester.clickLink("update");
        //assert that datepicker is disabled
        tester.assertDisabled("form:datepicker");
}
//...
```

20.1.3 Testing components in isolation

Method `startComponent(Component)` can be used to test a component in isolation without having to create a container page for this purpose. The target component is rendered and both its methods `onInitialize()` and `onBeforeRender()` are executed. In the test case from project *CustomFormComponentPanel* we used this method to check if our custom form component correctly renders its internal label:

```
//...
@Test
public void testCustomPanelContainsLabel(){
        TemperatureDegreeField field = new TemperatureDegreeField("field", Model.of(0.00));
        //Use standard JUnit class Assert
        Assert.assertNull(field.get("mesuramentUnit"));
        tester.startComponent(field);
        Assert.assertNotNull(field.get("mesuramentUnit"));
}
//...
```

If test requires a page we can use `startComponentInPage(Component)` which automatically generates a page for our component.

20.1.4 Testing the response

`WicketTester` allows us to access to the last response generated during testing with method `getLastResponse`. The returned value is an instance of class `MockHttpServletResponse` that provides helper methods to extract informations from mocked request.

In the test case from project *CustomResourceMounting* we extract the text contained in the last response with method `getDocument` and we check if it is equal to the RSS feed used for the test:

```
//...
@Test
public void testMountedResourceResponse() throws IOException, FeedException{
        tester.startResource(new RSSProducerResource());
        String responseTxt = tester.getLastResponse().getDocument();
        //write the RSS feed used in the test into a ByteArrayOutputStream
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        Writer writer = new OutputStreamWriter(outputStream);
        SyndFeedOutput output = new SyndFeedOutput();

        output.output(RSSProducerResource.getFeed(), writer);
        //the response and the RSS must be equal
        Assert.assertEquals(responseTxt, outputStream.toString());
}
//...
```

To simulate a request to the custom resource we used method `startResource` which can be used also with resource references.

20.1.5 Testing URLs

`WicketTester` can be pointed to an arbitrary URL with method `executeUrl(String url)`. This can be useful to test mounted pages, resources or request mappers:

```
//...
//the resource was mapped at '/foo/bar'
tester.executeUrl("./foo/bar");
//...
```

20.1.6 Testing AJAX components

If our application uses AJAX to refresh components markup, we can test if `AjaxRequestTarget` contains a given component with `WicketTester`'s method `assertComponentOnAjaxResponse`:

```
//...
//test if AjaxRequestTarget contains a component (using its instance)
tester.assertComponentOnAjaxResponse(amountLabel);
//...
//test if AjaxRequestTarget contains a component (using its path)
tester.assertComponentOnAjaxResponse("pathToLabel:labelId");
```

It's also possible to use method `isComponentOnAjaxResponse(Component cmp)` to know if a component has been added to `AjaxRequestTarget`:

```
//...
//test if AjaxRequestTarget does NOT contain amountLabel
assertFalse(tester.isComponentOnAjaxResponse(amountLabel));
//...
```

20.1.7 Testing AJAX events

Behavior `AjaxEventBehavior` and its subclasses can be tested simulating AJAX events with `WicketTester`'s method `executeAjaxEvent(Component cmp, String event)`. Here is the sample code from project *TestAjaxEventsExample*:

Home page code:

```
public class HomePage extends WebPage {
 public static String INIT_VALUE = "Initial value";
 public static String OTHER_VALUE = "Other value";

 public HomePage(final PageParameters parameters) {
        super(parameters);
        Label label;
        add(label = new Label("label", INIT_VALUE));
        label.add(new AjaxEventBehavior("click") {

                @Override
                protected void onEvent(AjaxRequestTarget target) {
                        //change label's data object
                        getComponent().setDefaultModelObject(OTHER_VALUE);
                        target.add(getComponent());
                }
        }).setOutputMarkupId(true);
        //...
 }
}
```

Test method:

```
@Test
public void testAjaxBehavior(){
        //start and render the test page
        tester.startPage(HomePage.class);
        //test if label has the initial expected value
        tester.assertLabel("label", HomePage.INIT_VALUE);
        //simulate an AJAX "click" event
        tester.executeAjaxEvent("label", "click");
        //test if label has changed as expected
        tester.assertLabel("label", HomePage.OTHER_VALUE);
}
```

20.1.8 Testing AJAX behaviors

To test a generic AJAX behavior we can simulate a request to it using `WicketTester`'s method `executeBehavior(AbstractAjaxBehavior behavior)`:

```
//...
AjaxFormComponentUpdatingBehavior ajaxBehavior = new AjaxFormComponentUpdatingBehavior
                                                                        ("change"){
        @Override
        protected void onUpdate(AjaxRequestTarget target) {
                //...
        }
};
component.add(ajaxBehavior);
```

```
//...
//execute AJAX behavior, i.e. onUpdate will be invoked
tester.executeBehavior(ajaxBehavior));
//...
```

20.1.9 Using a custom servlet context

In paragraph 13.9 we have seen how to configure our application to store resource files into a custom folder placed inside webapp root folder (see project *CustomFolder4MarkupExample*).

In order to write testing code for applications that use this kind of customization, we must tell `WicketTester` which folder to use as webapp root. This is necessary as under test environment we don't have any web server, hence it's impossible for `WicketTester` to retrieve this parameter from servlet context.

Webapp root folder can be passed to `WicketTester`'s constructor as further parameter like we did in the test case of project *CustomFolder4MarkupExample*:

```
public class TestHomePage{
   private WicketTester tester;

   @Before
   public void setUp(){
      //build the path to webapp root folder
      File curDirectory = new File(System.getProperty("user.dir"));
      File webContextDir = new File(curDirectory, "src/main/webapp");

      tester = new WicketTester(new WicketApplication(), webContextDir.getAbsolutePath());
   }
   //test methods...
}
```

20.2 Testing Wicket forms

Wicket provides utility class `FormTester` that is expressly designed to test Wicket forms. A new `FormTester` is returned by `WicketTester`'s method `newFormTester(String, boolean)` which takes in input the page-relative path of the form we want to test and a boolean flag indicating if its form components must be filled with a blank string:

```
//...
//create a new form tester without filling its form components with a blank string
FormTester formTester = tester.newFormTester("form", false);
//...
```

`FormTester` can simulate form submission with method `submit` which takes in input as optional parameter the submitting component to use instead of the default one:

```
//...
//create a new form tester without filling its form components with a blank string
FormTester formTester = tester.newFormTester("form", false);
//submit form with default submitter
formTester.submit();
//...
//submit form using inner component 'button' as alternate button
```

```
formTester.submit("button");
```

If we want to submit a form with an external link component we can use method `submitLink (String path, boolean pageRelative)` specifying the path to the link.

In the next paragraphs we will see how to use `WicketTester` and `FormTester` to interact with a form and with its children components.

20.2.1 Setting form components input

The purpose of a HTML form is to collect user input. `FormTester` comes with the following set of methods that simulate input insertion into form's fields:

- **setValue(String path, String value)**: inserts the given textual value into the specified component. It can be used with components `TextField` and `TextArea`. A version of this method that accepts a component instance instead of its path is also available.
- **setValue(String checkboxId, boolean value)**: sets the value of a given `CheckBox` component.
- **setFile(String formComponentId, File file, String contentType)**: sets a `File` object on a `FileUploadField` component.
- **select(String formComponentId, int index)**: selects an option among a list of possible options owned by a component. It supports components that are subclasses of `AbstractChoice` along with `RadioGroup` and `CheckGroup`.
- **selectMultiple(String formComponentId, int[] indexes)**: selects all the options corresponding to the given array of indexes. It can be used with multiple-choice components like `CheckGroup` or `ListMultipleChoice`.

`setValue` is used inside method `insertUsernamePassword` to set the username and password fields of the form used in project *StatelessLoginForm*:

```
protected void insertUsernamePassword(String username, String password) {
        //start and render the test page
        tester.startPage(HomePage.class);
        FormTester formTester = tester.newFormTester("form");
        //set credentials
        formTester.setValue("username", username);
        formTester.setValue("password", password);
        //submit form
        formTester.submit();
}
```

20.2.2 Testing feedback messages

To check if a page contains one or more expected feedback messages we can use the following methods provided by `WicketTester`:

- **assertFeedback(String path, String... messages)**: asserts that a given panel contains the specified messages
- **assertInfoMessages(String... expectedInfoMessages)**: asserts that the expected info messages are rendered in the page.
- **assertErrorMessages(String... expectedErrorMessages)**: asserts that the expected error messages are rendered in the page.

`assertInfoMessages` and `assertErrorMessages` are used in the test case from project *StatelessLoginForm* to check that form generates a feedback message in accordance with the login

result:

```
@Test
public void testMessageForSuccessfulLogin(){
        inserUsernamePassword("user", "user");
        tester.assertInfoMessages("Username and password are correct!");
}

@Test
public void testMessageForFailedLogin (){
        inserUsernamePassword("wrongCredential", "wrongCredential");
        tester.assertErrorMessages("Wrong username or password");
}
```

20.2.3 Testing models

Component model can be tested as well. With method `assertModelValue` we can test if a specific component has the expected data object inside its model.

This method has been used in the test case of project *ModelChainingExample* to check if the form and the drop-down menu share the same data object:

```
@Test
public void testFormSelectSameModelObject(){
        PersonListDetails personListDetails = new PersonListDetails();
        DropDownChoice dropDownChoice = (DropDownChoice) personListDetails.get("persons");
        List choices = dropDownChoice.getChoices();
        //select the second option of the drop-down menu
        dropDownChoice.setModelObject(choices.get(1));

        //start and render the test page
        tester.startPage(personListDetails);
        //assert that form has the same data object used by drop-down menu
        tester.assertModelValue("form", dropDownChoice.getModelObject());
}
```

20.3 Testing markup with TagTester

If we need to test component markup at a more fine-grained level, we can use class `TagTester` from package `org.apache.wicket.util.tester`.

This test class allows to check if the generated markup contains one or more tags having a given attribute with a given value. `TagTester` can not be directly instantiated but it comes with three factory methods that return one or more `TagTester` matching the searching criteria. In the following test case (from project *TagTesterExample*) we retrieve the first tag of the home page (a `<span>` tag) having attribute `class` equal to `myClass`:

HomePage markup:

```
<html xmlns:wicket="http://wicket.apache.org">
        <head>
                <meta charset="utf-8" />
                <title></title>
        </head>
        <body>
                <span class="myClass"></span>
```

```
                <div class="myClass"></div>
        </body>
</html>
```

Test method:

```
@Test
public void homePageMarkupTest()
{
        //start and render the test page
        tester.startPage(HomePage.class);
        //retrieve response's markup
        String responseTxt = tester.getLastResponse().getDocument();

        TagTester tagTester = TagTester.createTagByAttribute(responseTxt, "class", "myClass");

        Assert.assertNotNull(tagTester);
        Assert.assertEquals("span", tagTester.getName());

        List<TagTester> tagTesterList = TagTester.createTagsByAttribute(responseTxt, "class",
                                                                        "myClass", false);

        Assert.assertEquals(2, tagTesterList.size());
}
```

The name of the tag found by `TagTester` can be retrieved with its method `getName`. Method `createTagsByAttribute` returns all the tags that have the given value on the given attribute. In the code above we have used this method to test that our markup contains two tags having attribute `class` equal to `myClass`.

20.4 Summary

With a component-oriented framework we can test our pages and components as we use to do with any other Java entity. Wicket offers a complete support for writing testing code, offering built-in tools to test nearly all the elements that build up our applications (pages, containers, links, behaviors, etc...).

The main entity discussed in this chapter has been class `WicketTester` which can be used to write unit tests and acceptance tests for our application, but we have also seen how to test forms with `FormTester` and how to inspect markup with `TagTester`.

In addition to learning how to use the utility classes provided by Wicket for testing, we have also experienced the innovative approach of Wicket to web testing that allows to test components in isolation without the need of running our tests with a web server and depending only on JUnit as testing framework.

Appendix A: working with Maven

A.1 Switching Wicket to DEPLOYMENT mode

As pointed out in the note at page 9, Wicket can be started in two modes, DEVELOPMENT and DEPLOYMENT. When we are in DEVELOPMENT mode Wicket warns us at application startup with the following message:

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

As we can read Wicket itself discourages us from using DEVELOPMENT mode into production environment. The running mode of our application can be configured in three different ways. The first one is adding a filter parameter inside deployment descriptor web.xml:

```
<filter>
        <filter-name>wicket.MyApp</filter-name>
        <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
        <init-param>
                <param-name>applicationClassName</param-name>
                <param-value>org.wicketTutorial.WicketApplication</param-value>
        </init-param>
        <init-param>
              <param-name>configuration</param-name>
              <param-value>deployment</param-value>
        </init-param>
</filter>
```

The additional parameter is written in bold. The same parameter can be also expressed as context parameter:

```
<context-param>
    <param-name>configuration</param-name>
    <param-value>deployment</param-value>
</context-param>
```

The third way to set the running mode is using system property `wicket.configuration`. This parameter can be specified in the command line that starts up the server:

```
java -Dwicket.configuration=deployment ...
```

Remember that system properties overwrite other settings, so they are ideal to ensure that on

production machine the running mode will be always set to DEPLOYMENT.

A.2 Creating a Wicket project from scratch and importing it into our favourite IDE.

Note

In order to follow the instructions of this paragraph you must have Maven installed on your system. The installation of Maven is out of the scope of this guide but you can easily find an extensive documentation about it on Internet.

Another requirement is a good Internet connection (a flat ADSL is enough) because Maven needs to connect to its central repository to download the required dependencies.

A.2.1 From Maven to our IDE

Wicket project and its dependencies are managed using Maven[67]. This tool is very useful also when we want to create a new project based on Wicket from scratch. With a couple of shell commands we can generate a new project properly configured and ready to be imported into our favourite IDE.

The main step to create such a project is to run the command which generates project's structure and its artifacts. If we are not familiar with Maven or we simply don't want to type this command by hand, we can use the utility form on Wicket site at http://wicket.apache.org/start/quickstart.html:

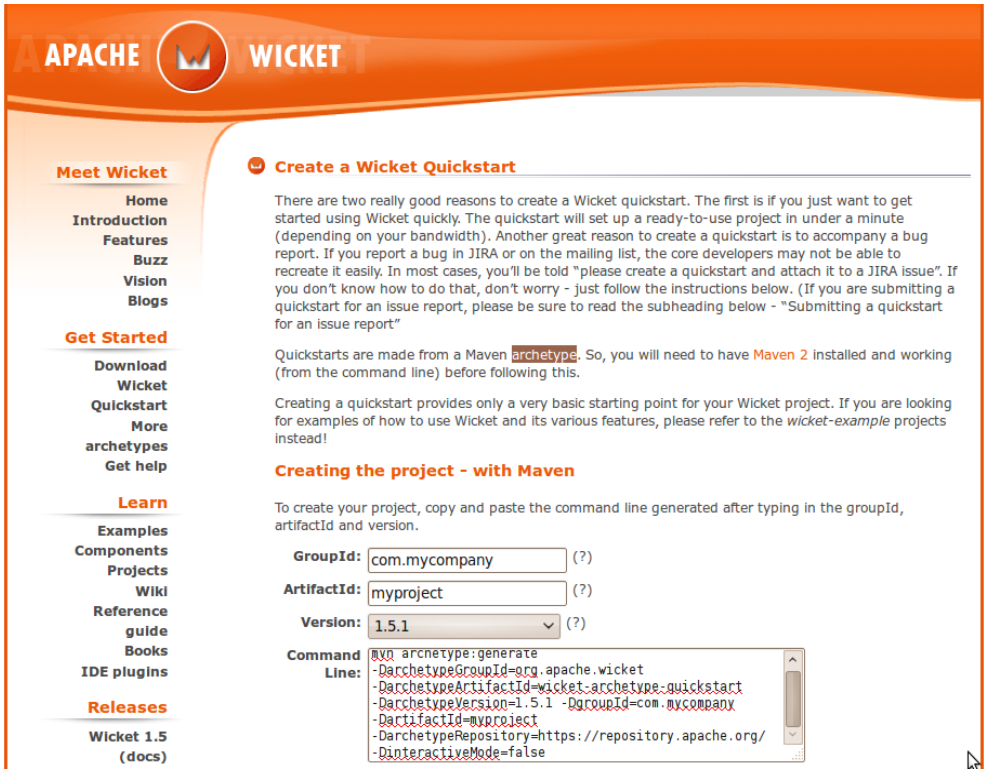


Illustration 1: Wicket quickstart page

Here we have to specify the root package of our project (***GroupId***), the project name (***ArtifactId***) and

67 http://maven.apache.org/

which version of Wicket we want to use (***Version***).

Once we have run the resulting command in the OS shell, we will have a new folder with the same name of the project (i.e the *ArtifactId*). Inside this folder we can find a file called *pom.xml*. This is the main file used by Maven to manage our project. For example, using “org.wicketTutorial” as *GroupId* and “MyProject” as *ArtifactId*, we would obtain the following artifacts:

```
.\MyProject
       |   pom.xml
       |
       \---src
           +---main
           |   +---java
           |   |   \---org
           |   |       \---wicketTutorial
           |   |               HomePage.html
           |   |               HomePage.java
           |   |               WicketApplication.java
           |   |
           |   +---resources
           |   |       log4j.properties
           |   |
           |   \---webapp
           |       \---WEB-INF
           |               web.xml
           |
           \---test
               \---java
                   \---org
                       \---wicketTutorial
                               TestHomePage.java
```

Amongst other things, file pom.xml contains a section delimited by tag `<dependencies>` which declares the dependencies of our project. By default the Maven archetype will add the following Wicket modules as dependencies:

```
...
<dependencies>
       <!--  WICKET DEPENDENCIES -->
       <dependency>
              <groupId>org.apache.wicket</groupId>
              <artifactId>wicket-core</artifactId>
              <version>${wicket.version}</version>
       </dependency>
       <dependency>
              <groupId>org.apache.wicket</groupId>
              <artifactId>wicket-ioc</artifactId>
              <version>${wicket.version}</version>
       </dependency>
       <!-- OPTIONAL DEPENDENCY
       <dependency>
              <groupId>org.apache.wicket</groupId>
              <artifactId>wicket-extensions</artifactId>
              <version>${wicket.version}</version>
```

```
        </dependency>
        -->
        ...
</dependencies>
...
```

If we need to use more Wicket modules or additional libraries, we can add the appropriate XML fragments[68] here.

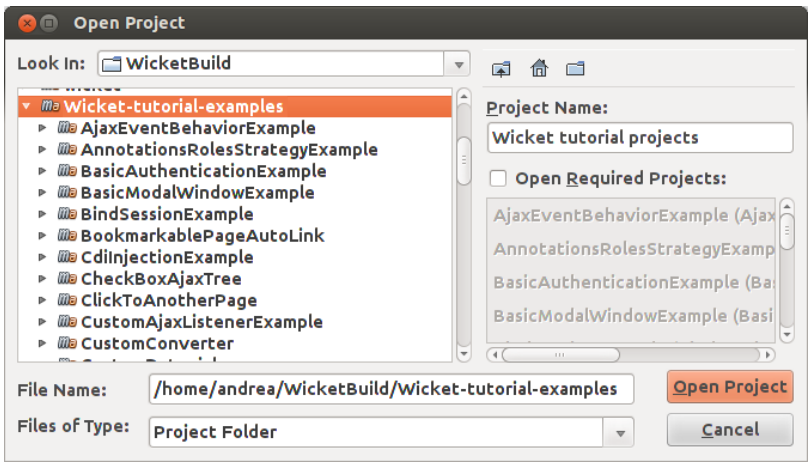
A.2.2 Importing a Maven project into our IDE

Maven projects can be easily imported into the most popular Java IDEs. However, the procedure needed to do this differs from IDE to IDE. In this paragraph we can find the instructions to import Maven projects into three of the most popular IDEs among Java developers : NetBeans, JetBrains IDEA and Eclipse.

NetBeans

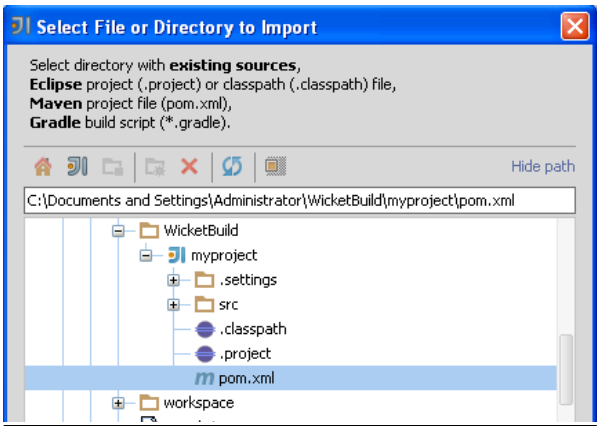
Starting from version 6.7, NetBeans includes Maven support, hence we can start it and directly open the folder containing our project:

68 *As described in Appendix B, the XML needed to include a dependency can be found at* http://mvnrepository.com/



IDEA

IDEA comes with a Maven importing functionality that can be started under “File/New Project/Import from external model/Maven”. Then, we just have to select the pom.xml file of our project:



Eclipse

If our IDE is Eclipse the import procedure is a little more complex. Before opening the new project we must generate the Eclipse project artifacts running the following command from project root:

```
mvn eclipse:eclipse
```

Now to import our project into Eclipse we must create a classpath variable called *M2_REPO* that must point to your local Maven repository. This can be done selecting “Window/Preferences” and searching for “Classpath Variables”. The folder containing our local Maven repository is usually under our user folder and is called .m2 (for example under Unix system is /home/<myUserName>/.m2/repository):

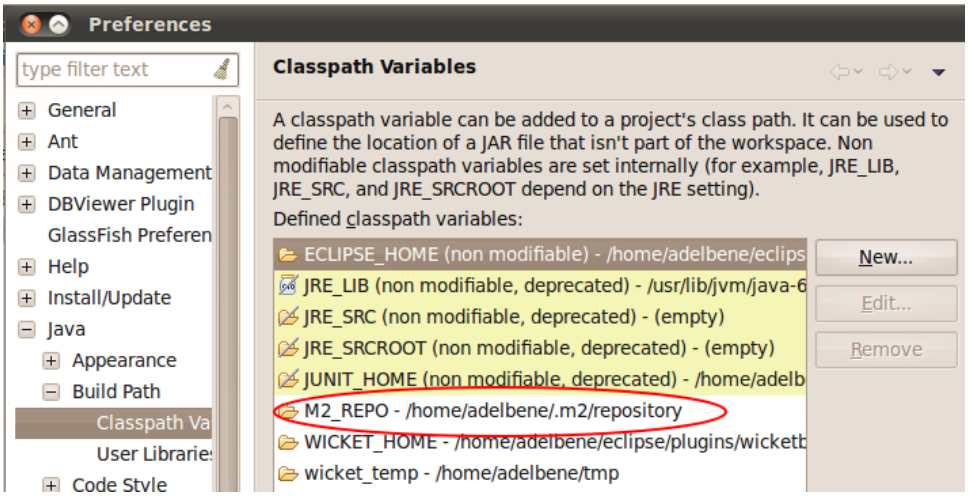


Illustration 2: Setting M2_REPO variable

Once we have created the classpath variable we can go to “File/Import.../Existing Project into Workspace”, select the directory of the project and press “Finish”:

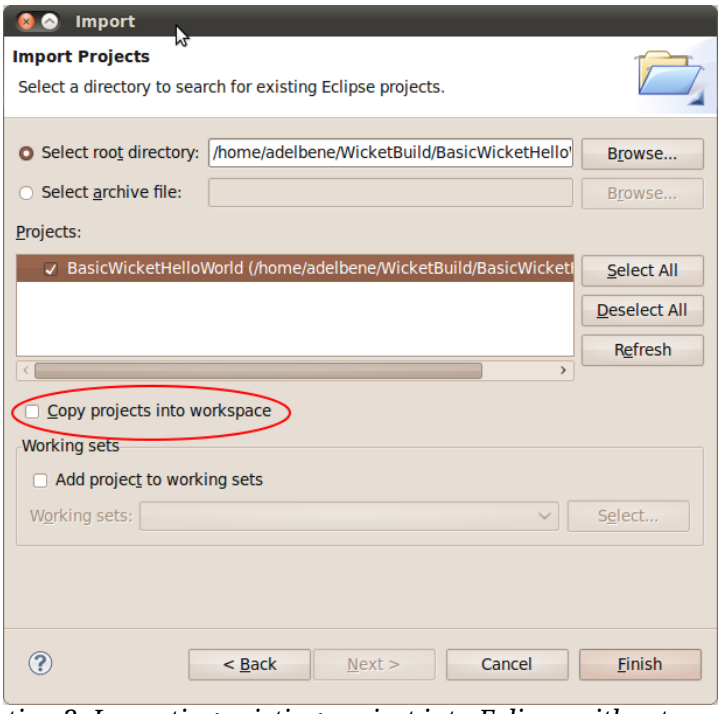


Illustration 3: Importing existing project into Eclipse without coping the original project into workspace (see the note below)

Once the project has been imported into Eclipse, we are free to use our favourite plug-ins to run it or debug it (like for example run-jetty-run: http://code.google.com/p/run-jetty-run/).

Note

Please note the option “Copy projects into workspace” in the previous illustration. If we select it, the original project generated with Maven won't be affected by the changes made inside Eclipse because we will work on a copy of it under the current workspace.

Note

If we modify the pom.xml file (for example adding further dependencies) we must regenerate project's artifacts and refresh the project (F5 key) to reflect changes into Eclipse.

A.2.3 Speeding up development with plugins.

Now that we have our project loaded into our IDE we could start coding our components directly by hand. However it would be a shame to not leverage the free and good Wicket plugins available for our IDE. The following is a brief overview of the most widely used plugins for each of the three main IDEs considered so far.

NetBeans

NetBeans offers Wicket support thought 'NetBeans Plugin for Wicket' hosted at http://java.net/projects/nbwicketsupport/. This plugin is released under CDDL-1.0 license.

You can find a nice introduction guide to this plugin at http://netbeans.org/kb/docs/web/quickstart-webapps-wicket.html.

IDEA

For JetBrain IDEA we can use WicketForge plugin, hosted at Google Code http://code.google.com/p/wicketforge/. The plugin is released under ASF 2.0 license.

Eclipse

With Eclipse we can install one of the plugins that supports Wicket. As of the writing of this document, the most popular is probably Qwickie, available in the Eclipse Marketplace and hosted on Google Code at http://code.google.com/p/qwickie/.

QWickie is released under ASF 2.0 license.

Appendix B: project WicketStuff

B.1 What is project WicketStuff?

WicketStuff is an umbrella project that gathers different Wicket-related projects developed and maintained by the community. The project is hosted on GitHub at https://github.com/wicketstuff/core. Every module is structured as a parent Maven project containing the actual project that implements the new functionality and an example project that illustrates how to use it in our code. The resulting directory structure of each module is the following:

```
\<module name>-parent
        |
        +---<module name>
        \---<module name>-examples
```

So far we have introduced only modules *Kryo Serializer* and *JavaEE Inject*, but WicketStuff comes with many other modules that can be used in our applications[69]. Some of them come in handy to improve the user experience of our pages with complex components or integrating some popular web services (like Google Maps[70]) and JavaScript libraries (like TinyMCE[71]).

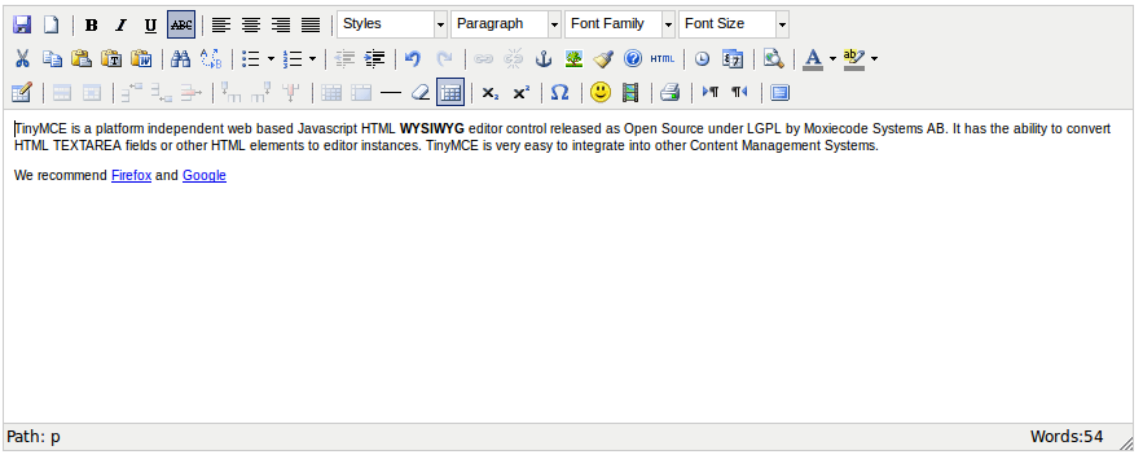
This appendix provides a quick overview of what WicketStuff offers to enhance the usability and the visually-appealing of our pages.

Note

Every WicketStuff module can be downloaded as JAR archive at http://mvnrepository.com. This site provides also the XML fragment needed to include it as a dependency into our pom.xml file[72].

B.2 Module *tinymce*

Module `tinymce` offers integration with the namesake JavaScript library that turns our "humble" text-areas into a full-featured HTML WYSIWYG editor:

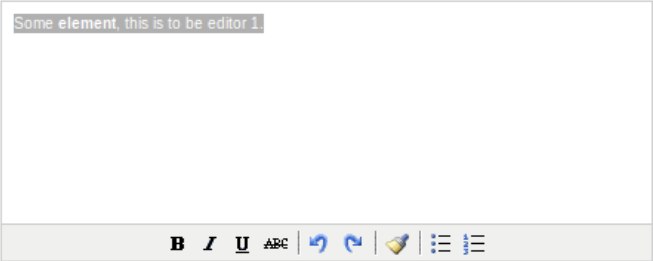


69 *The full list of the available modules can be found at https://github.com/wicketstuff/core/wiki*
70 *http://maps.google.com/*
71 *http://www.tinymce.com/*
72 *In addition to Maven also Ivy, Grape, Gradle ,Buildr and SBT are supported*

To “tinyfy” a textarea component we must use behavior `TinyMceBehavior`:

```
TextArea textArea = new TextArea("textArea", new Model(""));
textArea.add(new TinyMceBehavior());
```

By default `TinyMceBehavior` adds only a basic set of functionalities to our textarea:



To add more functionalities we must use class `TinyMCESettings` to register additional TinyMCE plugins and to customize the toolbars buttons. The following code is an excerpt from example page `FullFeaturedTinyMCEPage`:

```
TinyMCESettings settings = new TinyMCESettings(TinyMCESettings.Theme.advanced);
//...
// first toolbar
//...
settings.add(Button.newdocument, TinyMCESettings.Toolbar.first,
                                        TinyMCESettings.Position.before);
settings.add(Button.separator, TinyMCESettings.Toolbar.first,
                                         TinyMCESettings.Position.before);
settings.add(Button.fontselect, TinyMCESettings.Toolbar.first,
                                         TinyMCESettings.Position.after);
//...
// other settings
settings.setToolbarAlign(TinyMCESettings.Align.left);
settings.setToolbarLocation(TinyMCESettings.Location.top);
settings.setStatusbarLocation(TinyMCESettings.Location.bottom);
settings.setResizing(true);
//...
TextArea textArea = new TextArea("ta", new Model(TEXT));
textArea.add(new TinyMceBehavior(settings));
```

For more configuration examples see pages inside package `wicket.contrib.examples.tinymce` in the example project of the module.

B.3 Module *wicketstuff-gmap3*

Module `wicketstuff-gmap3` integrates Google Maps[73] service with Wicket providing component `org.wicketstuff.gmap.GMap`. If we want to embed Google Maps into one of our pages we just need to add component `GMap` inside the page. The following snippet is taken from example page `SimplePage`:

73 http://maps.google.com/

Markup code:

```
...
<body>
  <div wicket:id="map">Map</div>
</body>
...
```

Java code:

```
public class SimplePage extends WicketExamplePage
{
    public SimplePage()
    {
        GMap map = new GMap("map");
        map.setStreetViewControlEnabled(false);
        map.setScaleControlEnabled(true);
        map.setScrollWheelZoomEnabled(true);
        map.setCenter(new GLatLng(52.47649, 13.228573));
        add(map);
    }
}
```

The component defines a number of setters to customize its behavior and appearance. More info can be found on wiki page https://github.com/wicketstuff/core/wiki/Gmap3.

B.4 Module *wicketstuff-googlecharts*

To integrate the Google Chart[74] tool into our pages we can use module `wicketstuff-googlecharts`. To display a chart we must combine the following entities: component `Chart`, interface `IChartData` and class `ChartProvider`, all inside package `org.wicketstuff.googlecharts`. The following snippet is taken from example page `Home`:

Markup code:

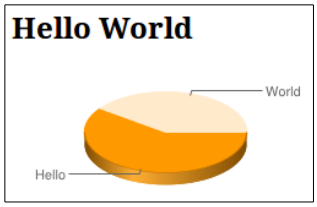
```
...
  <h2>Hello World</h2>
  <img wicket:id="helloWorld"/>
...
```

Java code:

```
IChartData data = new AbstractChartData(){
    public double[][] getData(){
       return new double[][] { { 34, 22 } };
    }
};

ChartProvider provider = new ChartProvider(new Dimension(250, 100), ChartType.PIE_3D,
                                           data);
provider.setPieLabels(new String[] { "Hello", "World" });
add(new Chart("helloWorld", provider));
```

Displayed chart:

74 https://developers.google.com/chart/



As we can see in the snippet above, component `Chart` must be used with `<img>` tag while the input data returned by `IChartData` must be a two-dimensional array of double values.

B.5 Module *wicketstuff-inmethod-grid*

Module `wicketstuff-inmethod-grid` implements a sophisticated grid-component with class `com.inmethod.grid.datagrid.DataGrid`.

Just like pageable repeaters (seen in paragraph 11.4) `DataGrid` provides data pagination and uses interface `IDataProvider` as data source. In addition the component is completely ajaxified:

ID	First Name	Last Name	Home Phone	Cell Phone
347	Abby	Gonzalez	710-555-1577	677-555-1601
360	Abby	Moore	510-555-4672	471-555-7145
374	Abby	Allen	883-555-5658	328-555-5650
383	Abby	Murray	876-555-6527	673-555-2368
389	Abby	Clark	251-555-7726	312-555-7068
521	Abby	Gomez	326-555-3855	221-555-6578
572	Abby	Davis	226-555-2267	504-555-1256
579	Abby	Williams	623-555-5207	736-555-7468
580	Abby	Wilson	738-555-8637	524-555-7745
616	Abby	Fisher	487-555-3461	265-555-5456
656	Abby	Fisher	838-555-3550	475-555-4836
339	Abner	Rose	807-555-6466	422-555-1237
349	Abner	Williams	757-555-6081	852-555-8773
428	Abner	Bailey	370-555-2806	603-555-4278
448	Abner	Clark	571-555-2535	536-555-5675
456	Abner	Fisher	611-555-8481	336-555-1360
470	Abner	Clark	445-555-2035	746-555-2151
488	Abner	Ortiz	737-555-2574	707-555-4505
507	Abner	Rose	864-555-1223	651-555-7400
510	Abner	Black	840-555-7446	584-555-2416

Showing 1 to 20 of 330 << < 1 2

`DataGrid` supports also editable cells and row selection:

ID	First Name	Last Name	Home Phone	Cell Phone	Edit
347	Abby	Gonzalez	710-555-1577	677-555-1601	
360	Abby	Moore	510-555-4672	471-555-7145	
374	Abby	Allen	883-555-5658	328-555-5650	
383	Abby	Murray	876-555-6527	673-555-2368	
389	Abby	Clark	251-555-7726	312-555-7068	
521	Abby	Gomez	326-555-3855	221-555-6578	
572	Abby	Davis	226-555-2267	504-555-1256	
579	Abby	Williams	623-555-5207	736-555-7468	
580	Abby	Wilson	738-555-8637	524-555-7745	
616	Abby	Fisher	487-555-3461	265-555-5456	

The following snippet illustrate how to use `DataGrid` and is taken from wiki page https://github.com/wicketstuff/core/wiki/InMethodGrid:

Markup code:

```
...
```

```
  <div wicket:id="grid">Grid</div>
...
```

Java code:

```
final List<Person> personList = //load a list of Persons
final ListDataProvider listDataProvider = new ListDataProvider(personList);
//define grid's columns
List<IGridColumn> cols = (List) Arrays.asList(
            new PropertyColumn(new Model("First Name"), "firstName"),
            new PropertyColumn(new Model("Last Name"), "lastName"));

DataGrid grid = new DefaultDataGrid("grid", new DataProviderAdapter(listDataProvider),
                                    cols);
add(grid);
```

In the code above we have used convenience class `DefaultDataGrid` that is a subclass of `DataGrid` and it already comes with a navigation toolbar.

The example pages are under package `com.inmethod.grid.examples.pages` in the example project which is hosted at http://www.wicket-library.com/inmethod-grid/data-grid/simple.

Alphabetical Index

A

AbstractAjaxTimerBehavior....167
AbstractCheckBoxModel....164
AbstractResource....130
AbstractResource 134
AbstractRoleAuthorizationStrategyWicket....186
AbstractTree....160
AbstractValidator....87
Acceptance test....193
Access-denied page....188
AJAX....155
AJAX behaviors....165
AJAX call listener....169
  global listeners....173
AjaxButton....156
AjaxCheckBox....157
AjaxEditableChoiceLabel....157
AjaxEditableLabel....157
AjaxEditableMultiLineLabel....157
AjaxEventBehavior....166
AjaxFallbackButton....157
AjaxFallbackLink....157
AjaxFormComponentUpdatingBehavior....167
AjaxFormSubmitBehavior....167
AjaxLink....155, 156
AjaxRequestAttributes....168
AjaxRequestTarget....155, 196
AjaxSubmitLink....156
AnnotationsRoleAuthorizationStrategy....188
Application....46, 148
ApplicationSettings....152
Attribute wicket:id....11
AttributeModifier....26
AuthenticatedWebApplication....180, 181, 189
AuthenticatedWebSession....180
Authentication....180
  authentication example....181
  onConfigure....37
AutoCompleteTextField....158
Autolink ....
  bookmarkable pages....56
  package resources....132

B

Behavior....26, 38
Behaviors....145
Bookmarkable pages....54
BookmarkablePageLink....56
Border....32
Broadcast....148
Built-in validators....84
Bundles lookup algorithm....122
  order of traversed bundles....125
Button....93

C

Callback URLs....146
CaptchaImageResource....130
CheckBox....102, 162
CheckBoxMultipleChoice....104
CheckboxMultipleChoiceSelector....105
CheckBoxSelector....105
CheckedFolder....162
ChoiceRenderer....75, 77
ClassPathResourceFinder....137
Component....7, 65, 145, 148
  and Model....65
ComponentNotFoundException....12
CompoundAuthorizationStrategy....185
CompoundPropertyModel....68, 73
continueToOriginalDestination....181
ConverterLocator....90
CryptoMapper....63

D

Data grid....213
DataView....113
  populateItem....113
DateTextField....141
DefaultItemReuseStrategy....113
DefaultMapperContext....62
DefaultMutableTreeNode....161
Deprecated tree components....161
DropDownChoice....74

E

Event-based component communication....148
ExternalLink....58

F

FileUploadField....96
Flash messages....88
Folder....162
Form....70
  feedback messages....82
  multipart content....96
  processing....82
  validation....82
FormComponent....70
FormComponentPanel....98
FormTester....198
  select....199
  selectMultiple....199
  setFile....199
  setValue....199
  submit....198
  submitLink....199

G

Generating HTML markup from code....152
getDefaultModelObject....72
getMarkupId....27
getPage....36
getParent....36
Google Chart....212
Google Maps....211
Guice....176
GuiceApplication....179
GWT....5

H

Header contribution....133
HeaderItem....133
Hibernate....1
HTTPS....190
HttpSession....50
HttpsMapper....190

I

IAjaxCallListener....168
IAjaxIndicatorAware....168
IApplicationSettings....44
IAuthorizationStrategy....184
IAutoCompleteRenderer....158
IChainingModel....75
IChoiceRender....75
IComponentAwareHeaderContributor....171
IComponentInheritedModel....68
IConverter....89
IConverterLocator....90
IDataProvider....113
IDetachable....78
IEvent....149
IEventSink....148
IEventSource....148
IFormSubmitter....70, 92
IFormSubmittingComponent....92
IHeaderContributor....133, 145
IHeaderResponse....133
IInitializer....150
IItemReuseStrategy....113
IJavaScriptLibrarySettings....143
Image....131
IMapperContext....61
IMarkupResourceStreamProvider....152
IModel....65
Indexed parameters....55
IndicatingAjaxButton....168
IndicatingAjaxFallbackLink....168
IndicatingAjaxLink....168
Initializer....150
Internationalization and Models....126
invalidate....52
invalidateNow....52
IPackageResourceGuard....191
IPageable....113
IRequestCycleListener....48
IRequestCycleProvider....46
IRequestHandler....47
  handler resolving algorithm....47
IRequestListener....146
IRequestMapper....47
IResource....130
  Attributes....130
IResourceFinder....137
IResourceSettings....125, 137, 138
  getResourceFinders....138
  getStringResourceLoaders....125
IResourceStreamLocator....137
IRoleCheckingStrategy....188
ISecuritySettings....189
isEnabled....26
ISerializer....42
ISessionListener....49
isPageStateless....40
isTemporary....51
IStringResourceLoader....125
isVisible....26
Item....112
ITreeProvider....161
IUnauthorizedComponentInstantiationListener....189
IValidator....82

J

JavaBeans....67
JavaEE Inject....210
JavaEE Inject 176
JavaEEComponentInjector....176
JavaSerializer....42
JBoss Seam....176
JMX....150
  Using JMX to control Wicket apps....150
JQuery integration....140
JQueryDateField....141
JQueryUI integration....140
JSF....5
JTree....161
JUnit....5, 193

K

Kryo project....42
Kryo serializer....42, 210

L

Label....8, 11, 65
Link....12, 54
  onClick....12
ListDataProvider....113
ListItem....111
ListView....111
  populateItem....111
  setReuseItems....112
LoadableDetachableModel....78
Locale....117
Localization in Wicket....117

M

Markup file....11
MarkupContainer....11, 18
Maven....8, 193, 204
importing projects into Eclipse....207
importing projects into IDEA....206
importing projects into NetBeans....206
Metadata....52
MetaDataKey....52
MetaDataRoleAuthorizationStrategy....186
MockHttpServletResponse....195
ModalWindow....158
PageCreator....159
WindowClosedCallback....160
Model....65
model chaining....75
model detaching....78
model inheritance....68
using more than one model....80
Model (class)....66
MountedMapper....60
mountPage....60
MultiFileUploadField....97
MVC pattern....4

N

Named parameters....54
NestedTree....161

O

onBeforeRender....36
onInitialize....36
onModelChanged....65
onSubmit....70
OpenEJB....176

P

PackageMapper....61
PackageResourceReference....131
Page mounting....
optional placeholders....61
placeholders....60
Page serialization....40, 42
Page versioning....40
disabling versioning....42
PageParameters....44, 54
PageReference....42
PagingNavigator....114
Palette....107
Panel....18
PasswordTextField....72
Path....137
PatternDateConvert....143
POJO....67
PropertyModel....68
PropertyResolver....68

Q

Query string parameters....54

R

Redirecting user to an intermediate page....183
RefreshingView....112
populateItem....112
Repeaters....110
RepeatingView....110
Request....46
RequestCycle....46, 148
hook methods....48
listeners....48
request processing....47
using a custom request cycle....46
RequestListenerInterface....146
RequireHttps....190
Resource management....
custom resources....134
mounting resources....135
package resources....130
resource dependencies....134
resource references....130
static vs dynamic resources....130
ResourceBundle....117
ResourceReference....130
ResourceResponse....135
WriteCallback....135
ResourceStreamLocator....137
Response....46
RestartResponseAtInterceptPageException....183
restartResponseAtSignInPage....181
ReuseIfModelsEqualStrategy....113
RoleAuthorizationStrategy....190
Roles....185

S

SecurePackageResourceGuard....191
Serializable....66
Session....49, 148
accessing to http session object....50
discarding session data....52
session listener....49
setDefaultModelObject....72
setEnabled....26
setMarkupId....27
setOutputMarkupId....27
setOutputMarkupPlaceholderTag....165
setRenderBodyOnly....30
setResponsePage....13, 48
setVersioned....42
setVisible....22, 26
SharedResourceReference....136
SimplePageAuthorizationStrategy....185
SortableDataProvider....113
Spring....1, 176, 177
SpringComponentInjector....177
Stateful pages....40
Stateless pages....40, 44, 55
StatelessLink....59
Struts....1
SubmitLink....93, 94
Swing....7
Swing/AWT....7
Switching Wicket to DEPLOYMENT mode....203

T

TagTester....200
Test Driven Development....193
Testing with Wicket....
setting form component input....199
testing AJAX behaviors....197
testing AJAX components....196
testing AJAX events....196
testing component status....195
testing components in isolation....195
testing feedback messages....199
testing links 194
testing models....200
testing web response....195
testing Wicket forms....198
TextField....72
TinyMCE....210
Tree repeaters....160
TreeModelProvider....161

U

Unit test....193
Updating hidden components via AJAX....165
urlFor and mapUrlFor....48
UrlPathPagParametersEncoder....62

V

Vaadin....5

W

WebApplication and testing....
WebApplication....194
WebApplicationPath....137, 138
WebMarkupContainer....27, 28
WebPage....7, 11
WebSession....49
Wicket forms....70
Wicket Links....12
Wicket modules....7
Wicket plugins....208
for Eclipse....208
for IDEA....208
for NetBeans....208
Qwickie....208
WicketForge....208
Wicket tags....
<wicket:body/>....32
<wicket:border>....32
<wicket:child>....23
<wicket:enclosure>....31
<wicket:extend>....23
<wicket:fragment>....28
<wicket:head>....29
<wicket:id>....6
<wicket:link>....56, 132
<wicket:message>....121
<wicket:panel>....19
<wicket:remove>....30
WicketRuntimeException....12
WicketServlet....11
WicketStuff....42, 176, 210
WicketTester....193
assertComponentOnAjaxResponse....196
assertEnabled....195
assertErrorMessages....199
assertFeedback....199
assertInfoMessages....199
assertLabel....194
assertModelValue....200
assertRenderedPage....194
assertRequired....195
assertVisible....195
clickLink....194
executeAjaxEvent....197
executeUrl....196
getLastRenderedPage....194
getLastResponse....195
isComponentOnAjaxResponse....196
newFormTester....198
startComponent....195
startComponentInPage....195
startPage....194
startResource....196