

Wicket 10.x Reference Guide

The Apache Software Foundation

Table of Contents

1. Introduction	1
2. How to use the example code	2
3. Why should I learn Wicket?	3
3.1. We all like spaghetti :-)	3
3.2. Component oriented frameworks - an overview	3
3.3. Benefits of component oriented frameworks for web development	4
3.4. Wicket vs the other component oriented frameworks	4
4. Wicket says "Hello world!"	6
4.1. Wicket distribution and modules	6
4.2. Configuration of Wicket applications	8
4.3. The HomePage class	10
4.4. Wicket Links	12
4.5. Summary	13
5. Wicket as page layout manager	15
5.1. Header, footer, left menu, content, etc.....	15
5.2. Here comes the inheritance!	17
5.3. Divide et impera!	21
5.4. Markup inheritance with the wicket:extend tag	24
5.5. Summary	26
6. Keeping control over HTML	28
6.1. Hiding or disabling a component	28
6.2. Modifying tag attributes	28
6.3. Generating tag attribute 'id'	29
6.4. Creating in-line panels with WebMarkupContainer	30
6.5. Working with markup fragments	30
6.6. Adding header contents to the final page	32
6.7. Using stub markup in our pages/panels	33
6.8. How to render component body only	33
6.9. Hiding decorating elements with the wicket:enclosure tag	34
6.10. Surrounding existing markup with Border	35
6.11. Summary	37
7. Components lifecycle	38
7.1. Lifecycle stages of a component	38
7.2. Hook methods for component lifecycle	38
7.3. Initialization stage	39
7.4. Rendering stage	39
7.5. Removed stage	43
7.6. Detached stage	43

7.7. Summary	43
8. Page versioning and caching	44
8.1. Stateful pages vs stateless	44
8.2. Stateful pages	44
8.3. Stateless pages	49
8.4. Summary	50
9. Under the hood of the request processing	51
9.1. Class Application and request processing	51
9.2. Request and Response classes	51
9.3. The “director” of request processing - RequestCycle	51
9.4. Session Class	55
9.5. Exception handling	59
9.6. Summary	61
10. Wicket Links and URL generation	62
10.1. PageParameters	62
10.2. Bookmarkable links	65
10.3. Automatically creating bookmarkable links with tag wicket:link	65
10.4. External links	67
10.5. Stateless links	68
10.6. Generating structured and clear URLs	68
10.7. Summary	73
11. Wicket models and forms	75
11.1. What is a model?	75
11.2. IModel and Lambda	76
11.3. Models and JavaBeans	78
11.4. Wicket forms	82
11.5. Component DropDownChoice	86
11.6. Model chaining	88
11.7. Detachable models	90
11.8. Using more than one model in a component	93
11.9. Use models!	94
11.10. Summary	94
12. Wicket forms in detail	95
12.1. Default form processing	95
12.2. Form validation and feedback messages	95
12.3. Input value conversion	103
12.4. Validation with JSR 303	106
12.5. Submit form with an IFormSubmittingComponent	108
12.6. Nested forms	112
12.7. Multi-line text input	112
12.8. File upload	113

12.9. Creating complex form components with <code>FormComponentPanel</code>	115
12.10. Stateless form	118
12.11. Working with radio buttons and checkboxes	120
12.12. Selecting multiple values with <code>ListMultipleChoices</code> and <code>Palette</code>	124
12.13. Summary	128
13. Displaying multiple items with repeaters	129
13.1. The <code>RepeatingView</code> Component	129
13.2. The <code>ListView</code> Component	130
13.3. The <code>RefreshingView</code> Component	131
13.4. Pageable repeaters	133
13.5. Summary	135
14. Component queueing	137
14.1. Markup hierarchy and code	137
14.2. Improved auto components	140
14.3. When are components dequeued?	141
14.4. Restrictions of queueing	142
14.5. Summary	142
15. Internationalization with Wicket	143
15.1. Localization	143
15.2. Localization in Wicket	144
15.3. Bundles lookup algorithm	149
15.4. Localization of component's choices	153
15.5. Internationalization and Models	155
15.6. Summary	157
16. Resource management with Wicket	158
16.1. Static vs dynamic resources	158
16.2. Resource references	158
16.3. Package resources	158
16.4. Adding resources to page header section	163
16.5. Context-relative resources	165
16.6. Resource dependencies	166
16.7. Aggregate multiple resources with resource bundles	166
16.8. Put JavaScript inside page body	167
16.9. Header contributors positioning	169
16.10. Custom resources	169
16.11. Mounting resources	171
16.12. Lambda support	171
16.13. Shared resources	172
16.14. Customizing resource loading	173
16.15. <code>CssHeaderItem</code> and <code>JavaScriptHeaderItem</code> compression	175
16.16. NIO resources	176

16.17. Resources derived through models	178
16.18. Summary	179
17. An example of integration with JavaScript	180
17.1. What we want to do... ..	180
17.2. ...and how we will do it	181
17.3. Summary	185
18. Wicket advanced topics	186
18.1. Enriching components with behaviors	186
18.2. Generating callback URLs with IRequestListener	187
18.3. Wicket events infrastructure	190
18.4. Initializers	191
18.5. Using JMX with Wicket	192
18.6. Generating HTML markup from code	195
18.7. Summary	197
19. Working with AJAX	198
19.1. How to use AJAX components and behaviors	198
19.2. Build-in AJAX components	200
19.3. Built-in AJAX behaviors	210
19.4. Using an activity indicator	213
19.5. AJAX request attributes and call listeners	214
19.6. Creating custom AJAX call listener	216
19.7. Stateless AJAX components/behaviors	221
19.8. Lambda support for components	222
19.9. Lambda support for behaviors	223
19.10. Summary	223
20. Integration with enterprise containers	224
20.1. Integrating Wicket with EJB	224
20.2. Integrating Wicket with Spring	225
20.3. JSR-330 annotations	227
20.4. Summary	227
21. Native WebSockets	228
21.1. How does it work ?	228
21.2. How to use	228
21.3. Client-side APIs	231
21.4. Testing	232
21.5. FAQ	232
22. Security with Wicket	233
22.1. Authentication	233
22.2. Authorizations	237
22.3. Using HTTPS protocol	244
22.4. URLs encryption in detail	245

22.5. CSRF protection	246
22.6. Content Security Policy (CSP)	248
22.7. Cross Origin Isolation with COOP and COEP	251
22.8. Package Resource Guard	252
22.9. External Security Checks	253
22.10. Summary	254
23. Test Driven Development with Wicket	256
23.1. Utility class WicketTester	256
23.2. Testing Wicket forms	263
23.3. Testing markup with TagTester	266
23.4. Summary	267
24. Test Driven Development with Wicket and Spring	268
24.1. Configuration of the runtime environment	268
24.2. Configuration of the JUnit based integration test environment	270
24.3. Summary	275
25. Wicket Best Practices	276
25.1. Encapsulate components correctly	276
25.2. Put models and page data in fields	278
25.3. Correct naming for Wicket IDs	279
25.4. Avoid changes at the component tree	279
25.5. Implement visibilities of components correctly	280
25.6. Always use models	281
25.7. Do not unwrap models within the constructor hierarchy	282
25.8. Pass models extended components	282
25.9. Validators must not change any data or models	283
25.10. Do not pass components to constructors	283
25.11. Use the Wicket session only for global data	284
25.12. Do not use factories for components	285
25.13. Every page and component must be tested	287
25.14. Avoid interactions with other servlet filters	287
25.15. Cut small classes and methods	287
25.16. The argument "Bad documentation"	288
25.17. Summary	289
26. Wicket Internals	290
26.1. Page storing	290
26.2. Markup parsing and Autocomponents	292
27. Wicket HTTP/2 Support (Experimental)	295
27.1. Example Usage	295
27.2. Create server specific http/2 push support	296
28. Wicket Metrics Monitoring (Experimental)	299
28.1. Example setup	299

28.2. Visualization with Graphite	301
28.3. Measured data	303
28.4. Write own measurements	304
Appendix A: Working with Maven	306
A.1. Switching Wicket to DEPLOYMENT mode	306
A.2. Creating a Wicket project from scratch and importing it into our favourite IDE	307
Appendix B: Project WicketStuff	314
B.1. What is project WicketStuff	314
B.2. Module tinymce	315
B.3. Module wicketstuff-gmap3	316
B.4. Module wicketstuff-googlecharts	317
B.5. Module wicketstuff-inmethod-grid	318
B.6. Module wicketstuff-rest-annotations	319
B.7. Module wicketstuff-lambda-components	321
Appendix C: Lost In Redirection With Apache Wicket	322
Appendix D: Working with Karaf	327
D.1. Wicket feature	327
Appendix E: Contributing to this guide	329

Chapter 1. Introduction

Wicket has been around since 2004 and it has been 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 many live examples and many technical articles on the Internet (most of them at [Wicket Examples Site](#) and at [Wicket in Action](#)), 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 in [the according repository on Github](#). 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 for this framework.

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

Editors:

Andrea Del Bene, adelbene@apache.org

Martin Grigorov

Tobias Soloschenko

Igor Vaynberg

Carsten Hufe

Christian Kroemer

Daniel Bartl

Paul Bor

Joachim Rohde

Emond Papegaaij

PS: this guide is based on Wicket 9. However if you are using an older version you should find this guide useful as well, but it's likely that the code and the snippets won't work with your version.

PPS: although we try to do our best working on this guide, 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!

Project started by

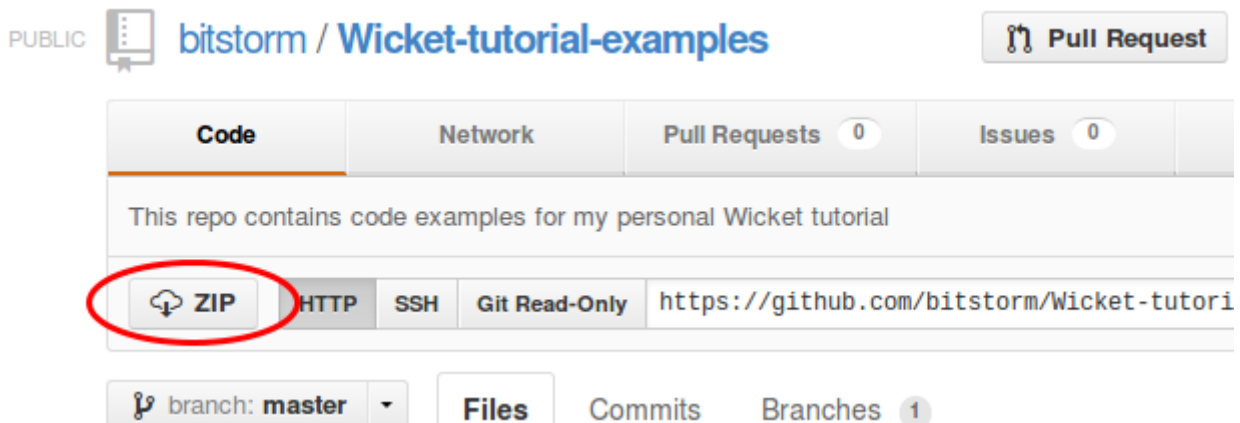


Chapter 2. How to use the example code

Most of the code you will find in this document is available as a [Git repository](#) and is licensed under the ASF 2.0. Examples are hosted live at <https://wicket-guide.herokuapp.com/>. 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:

TestAjaxEventsExample	9 days ago	Added license header [bitstorm]
UploadSingleFile	22 days ago	Clean up [bitstorm]
UsernameCustomValidator	22 days ago	Clean up [bitstorm]
.gitignore	4 months ago	Added PageDataViewExample [andrea]
LICENSE	5 months ago	Added Apache License 2.0 header [andrea]
header.txt	5 months ago	Component JQueryDateField was made self-contained [andrea]
pom.xml	9 days ago	-Fixed project StatelessPage [bitstorm]

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

Chapter 3. 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’s ecosystem is already full of many well-known web frameworks, so why should we spend our time learning Wicket?

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

3.1. 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 them (like Struts, Spring MVC, Velocity, etc...) have become widely adopted and they made the MVC pattern very popular among developers. However, none of these frameworks offers a real object-oriented (OO) abstraction for web pages and we still have to take care of web-related tasks such as HTTP request/response handling, URL mapping, storing data into user sessions and so on.

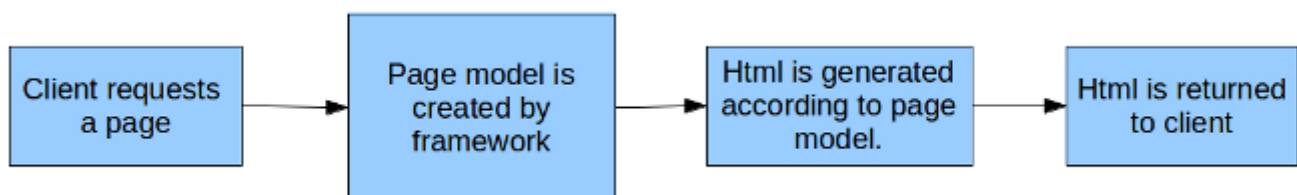
The biggest limit of MVC frameworks is that they don’t do much to overcome the impedance mismatch between the stateless nature of HTTP protocol and the need for our web applications to handle (a very complex) state.

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.

3.2. Component oriented frameworks - an overview

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

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



General schema of page request handling for a component oriented framework

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 the JVM heap and we can handle them as we 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. Most of those kind of frameworks hide the details of the HTTP protocol and naturally solve the problem of its stateless nature.

3.3. Benefits of component oriented frameworks for web development

At this point some people may still wonder why OOP is so important 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 not 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 4.2](#)).
- **We don't have to worry about an 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 the *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 23](#)).

3.4. Wicket vs the other component oriented frameworks

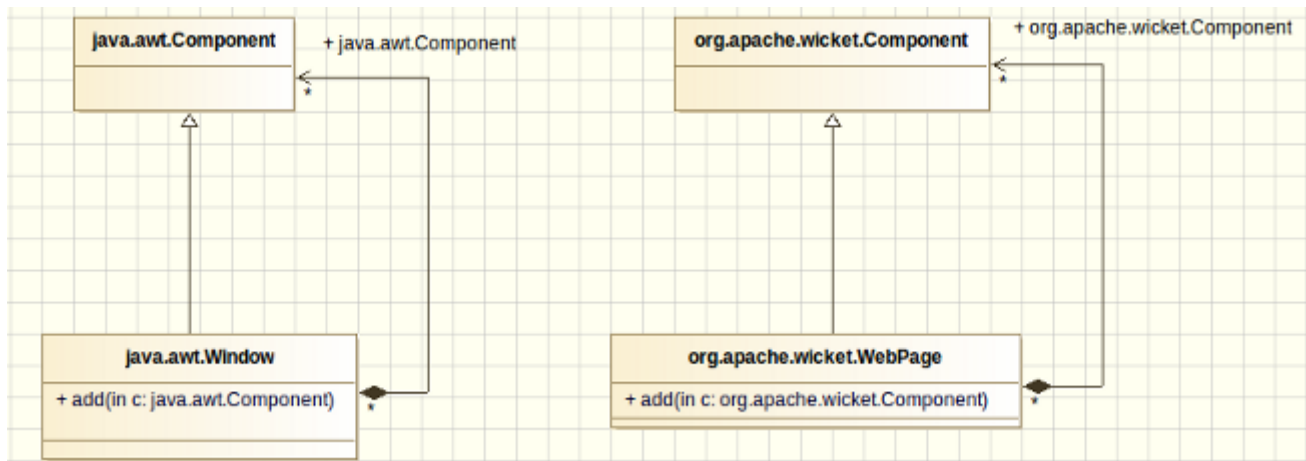
Wicket is not the only component oriented framework available in the Java ecosystem. Among its competitors we can find GWT (from Google), JSF (from Oracle), Vaadin (from Vaadin Ltd.), etc... Even if Wicket and all those other frameworks have their pros and cons, there are good reasons to prefer Wicket 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:** The Wicket team supports and promotes the dialogue with the framework's users through two mailing lists ([one for users and another one for framework developers](#)) and an [Apache JIRA](#) (the issue tracking system). Moreover, as any other Apache project, Wicket is developed paying great attention to user feedback and to suggested features.

- **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 the 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 in our web tier:** one of the most annoying and error-prone tasks 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 the inverse path (load data from Java to the web form). Moreover, most times our forms will use a JavaBean or a POJO as a 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 10](#) will introduce a Wicket model concept 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` (unless you are using Servlet 3 or a later version. See [Chapter 4](#) for more details).

Chapter 4. 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, etc...), 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 when working with this framework. We’ll learn more about custom components in the next chapters.

4.1. 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’s name	Description	Dependencies
wicket-core	Contains the main classes of the framework, like class <i>Component</i> and <i>Application</i> .	wicket-request, wicket-util
wicket-tester	Contains common classes for unit testing (like <i>WicketTester</i>).	wicket-core
wicket-core-tests	Contains test cases for wicket-core.	wicket-core, wicket-tester
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-bean-validation	Provides support for JSR 303 standard validation.	wicket-core, wicket-tester
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-tester
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-tester
wicket-auth-roles	Provides support for role-based authorization.	wicket-core, wicket-tester
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-tester
wicket-guice	This module provides integration with the dependency injection framework developed by Google.	wicket-core, wicket-ioc, wicket-tester
wicket-spring	This module provides integration with Spring framework.	wicket-core, wicket-ioc, wicket-tester
wicket-velocity	This module provides panels and utility class to integrate Wicket with Velocity template engine.	wicket-core, wicket-tester
wicket-jmx	This module provides panels and utility class to integrate Wicket with Java Management Extensions.	wicket-core, wicket-tester
wicket-objectsizeof-agent	Provides integration with Java agent libraries and instrumentation tools.	wicket-core

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

4.2. 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 the HelloWorldExample project). 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.



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 these 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.

4.2.1. 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`:

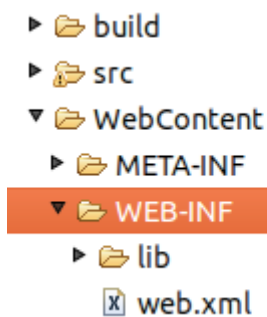


Illustration: The standard directory structure of a Wicket application

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.

If we are using Servlet 3 or a later version, we can of course use a class in place of web.xml to configure our application. The following example uses annotation *WebFilter*.

```
@WebFilter(value = "/*", initParams = {
    @WebInitParam(name = "applicationClassName", value =
"com.mycompany.WicketApplication"),
    @WebInitParam(name="filterMappingUrlPattern", value="/*") })
public class ProjectFilter extends WicketFilter {

}
```



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. [Chapter 24.1](#) contains the 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!

4.2.2. The application class

If we look back at *web.xml* we can see that we have provided the 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 a homepage for our application. Another important method is *init()*:

```
protected void init()
```


This method is called when our application is loaded by the web server (Tomcat, Jetty, etc...) and is the ideal place to put our configuration code. The *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, which will be gradually introduced in the next chapters when covering related settings.

The current application's instance can be retrieved at any time by calling static method *Application.get()* in our code. We will give more details about this method in [chapter 9.3](#). The content of the application class from the HelloWorldExample project 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 a Wicket application, we don't need to specify anything inside the *init* method. The home page of the application is the *HomePage* class. In the next paragraph we will see how this page is implemented and what conventions we have to follow to create a page in Wicket.



Declaring a *WicketFilter* inside web.xml descriptor is not the only way we have to kick-start our application. If we prefer to use a servlet instead of a filter, we can use class *org.apache.wicket.protocol.http.WicketServlet*. See the JavaDoc for further details.

4.3. The HomePage class

To complete our first Wicket application we must explore the home page class that is returned by the *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:

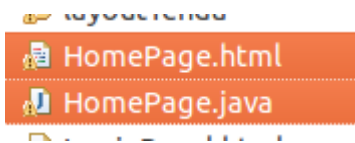


Illustration: Page class and its related HTML file

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 [chapter 16.14](#).

The Java code for the *HomePage* class 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 5.2](#). 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 the component id. If we run this example we will see the text *Hello WicketWorld!* Inside a *<div>* tag.



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)



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

4.4. 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 time 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 the user clicks on it.

That said, you shouldn't be surprised to find an abstract method called *onClick()* inside the *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<Void>("id"){
            @Override
            public void onClick() {
                //link code goes here
            }
        });
    }
}
```

By default after *onClick* has been executed, Wicket will send back to the current page to the client web browser. If we want to navigate to another page we must use method *setResponsePage* of class *Component*:

```
public class HomePage extends WebPage {
    public HomePage(){
        add(new Link<Void>("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 as input the class of the target page. In this way a new instance of *AnotherPage* will be created each time we click on the link. The other version of *setResponsePage* 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 8](#), when we will introduce the topic of stateful and stateless pages. For now, we can consider them as equivalent.

Since Wicket 8 is built on Java 8, we can choose to leverage lambda expressions to specify handler method:

```

//create a standard link component
add(ComponentFactory.link("id", (newlink) -> { /*do stuff*/ }));

```

Factory class *ComponentFactory* is provided by the WicketStuff project. You can find more information on this project, as well as the instructions to use its modules, in [Appendix B](#).

Wicket comes with a rich set of link components suited for every need (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 10](#).



We can specify the content of a link (i.e. the text inside it) with its method *setBody*. This method takes in input a generic Wicket model, which will be the topic of [chapter 11](#).

4.5. Summary

In this chapter we have seen the basic elements that compose a Wicket application. We have started preparing the configuration artifacts needed for our applications. As promised in [chapter 2.4](#), we needed to put in place just a minimal amount of XML with an application class and a 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 navigate from a page to another.

Chapter 5. 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 get out of Wicket, but it is surely the simplest one so it's the best way to get our hands dirty with some code.

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

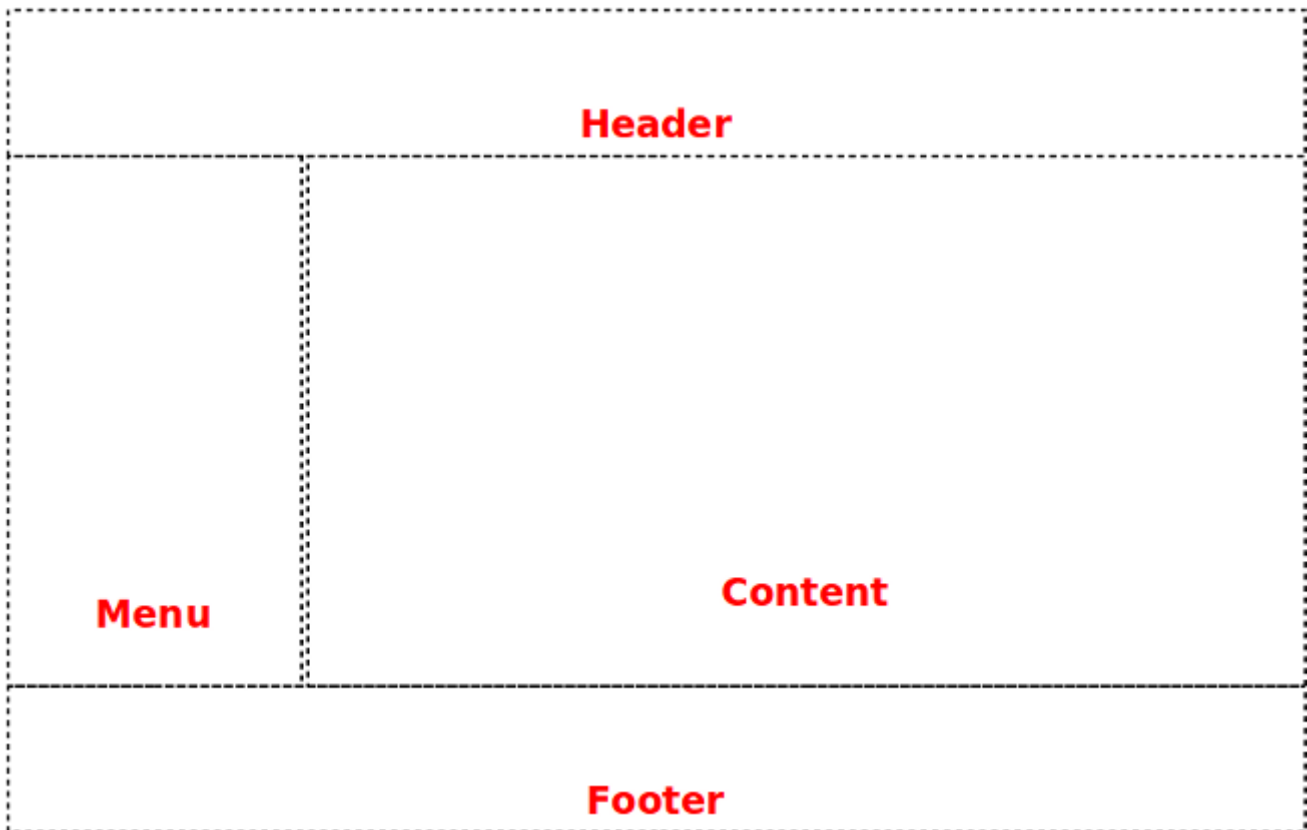
There was a time in the 90s when Internet was just a buzzword and watching a plain HTML page being rendered by a browser was a new and amazing experience. In those days we used to organize our page layout using the `<frame>` HTML tag. Over the years this tag has almost disappeared from our code and it survives only in few specific domains. For example is still being used by JavaDoc.

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.

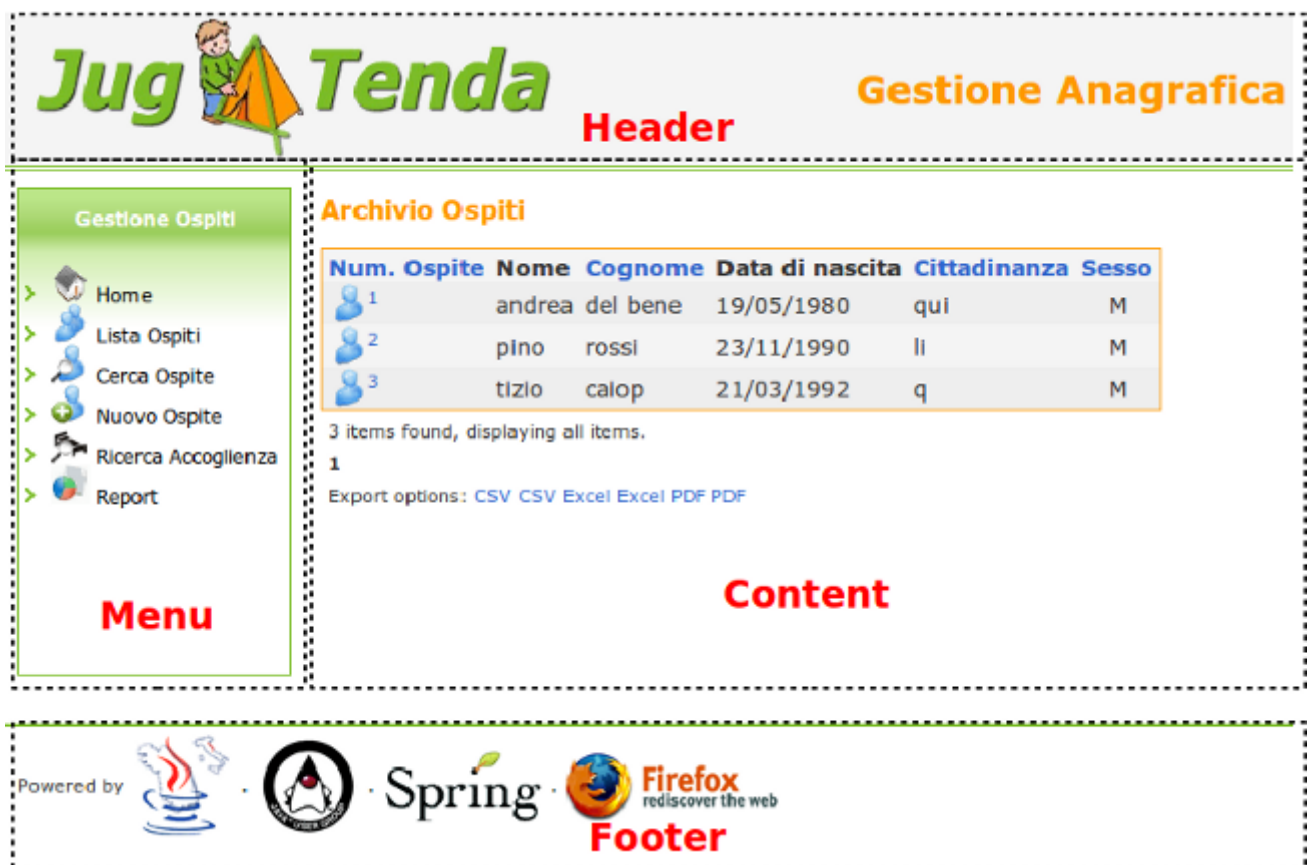
In this chapter we will see how to use Wicket to build a site layout. The sample layout we will use is a typical page layout consisting of the following areas:

- **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:



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:



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):

The screenshot shows a web application layout for 'Jug Tenda'. The header includes the site logo and the title 'Gestione Anagrafica'. The main content area is divided into two sections: 'Gestione Ospiti' (Guest Management) on the left and 'Archivio Ospiti' (Guest Archive) on the right. The 'Gestione Ospiti' section contains a sidebar menu with links to Home, Lista Ospiti, Cerca Ospite, Nuovo Ospite, Ricerca Accoglienza, and Report. The 'Archivio Ospiti' section displays a table of guests with columns for Num. Ospite, Nome, Cognome, Data di nascita, Cittadinanza, and Sesso. Below the table, it indicates '3 items found, displaying all items.' and provides export options: CSV, Excel, PDF. The footer includes logos for 'Powered by', Spring, and Firefox, along with the text 'rediscover the web'.

Key JSP include directives and HTML elements are highlighted in red:

- Header: `<%@include file="../common/Jug4TendaHeader.jsp"%>`
- Sidebar menu: `<%@include file="gestioneOspiteMenu.htm"%>`
- Main content area: `<div id="Content">`
- Footer: `<%@include file="../common/Jug4TendaFooter.jsp"%>`



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

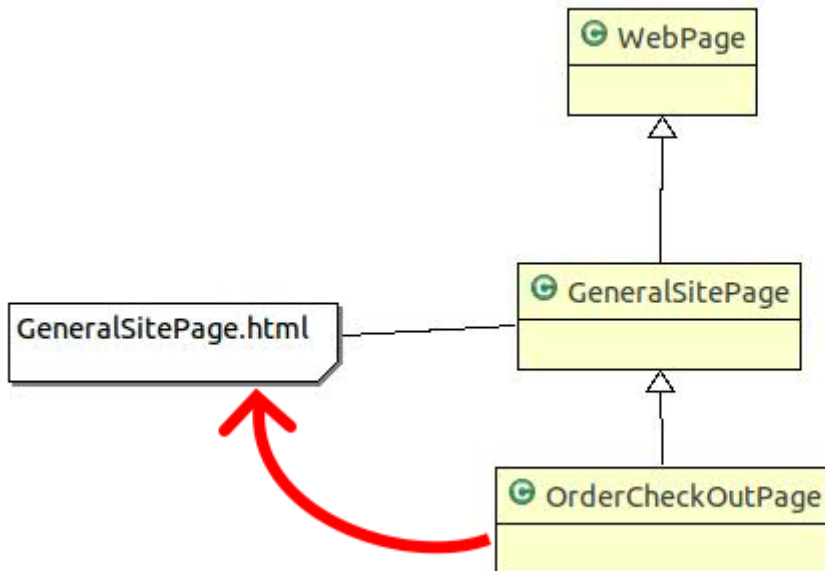
Now let's see how we can handle the layout of our web application using Wicket.

5.2. Here comes the inheritance!

The need of ensuring a consistent layout across our pages unveiled a serious limit of the HTML: 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 goals of Wicket is to overcome this kind of limit.

5.2.1. 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 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 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:



Markup inheritance comes in handy for page layout management as it helps us avoid the burden of checking that each page conforms to the 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.



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

5.2.2. Panel class

Class *org.apache.wicket.markup.html.panel.Panel* is a special component which lets 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: Hierarchy of WebPage and Panel classes

Subclasses of *MarkupContainer* can contain children components that can be added with method *add(Component...)* (seen in [chapter 3.3](#)). *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 method *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 *MarkupContainers*. 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 those 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 through children components. This can be done in the old way (pre-Wicket 8) using method *iterator* or using visitor pattern with method *visitChildren*. Starting from Wicket 8 the same task can be accomplished using the *stream* object returned by methods *stream* (which contains only the direct children) and *streamChildren* (which contains all children).

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>  
</html>
```

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

5.3. Divide et impera!

Let's go back to our layout example. In [chapter 5.1](#) 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 pages and panels. The code examples are from project MarkupInheritanceExample.

5.3.1. 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>
            
          </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.

5.3.2. 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="contentComponent">content</div>
</div>
<div id="footer" wicket:id="footerPanel">footer</div>
</body>
</html>
```

The HTML code for this page implements the generic left-menu layout 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 the page and panels:

```
package helloWorld.layoutTenda;
```

```

import org.apache.wicket.markup.html.WebPage;
import org.apache.wicket.Component;
import org.apache.wicket.markup.html.basic.Label;

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.

5.3.3. 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 *LoginPanel* 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>

```

```

    </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 11 and 12. Since this is a login page we don't want it to display the 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 the 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:



5.4. Markup inheritance with the wicket:extend tag

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

component. An example of a parent page using the 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 the 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>
```

5.4.1. 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 the tag `<wicket:child>`. Going forward with our example we can build a login page creating class *SimpleLoginPage* which extends the *JugTemplate* page, 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.

5.5. 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 pages on the web site. During the chapter we

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

Chapter 6. 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 prevent you from having any initial misleading feeling about Wicket showing you how to control and manipulate the generated HTML with the built-in tools shipped with the framework.

6.1. 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 ``) and it can not fire JavaScript events.

Class *Component* provides two getter methods to determine 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 determine 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.



Overriding *isVisible* or *isEnabled* can lead to unpredictable results when querying visibility information via *isVisibleInHierarchy* or *isEnabledInHierarchy*.

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

6.2. Modifying tag attributes

To modify tag attributes in a component's HTML markup we can use class *org.apache.wicket.AttributeModifier*. This class extends *org.apache.wicket.behavior.Behavior* and can be added to any component via the *Component*'s *add* method. Class *Behavior* is used to expand component functionalities and it can also modify component markup. We will see this class in detail later in [chapter 19.1](#).

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 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"));
```

6.3. 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()*.

Wicket generates markup ids using an instance of interface *org.apache.wicket.IMarkupIdGenerator*. The default implementation is *org.apache.wicket.DefaultMarkupIdGenerator* and it uses a session-scoped counter to generate the final id. A different generator can be set with the markup settings class *org.apache.wicket.settings.MarkupSettings* available in the application class:

```
@Override
public void init()
{
    super.init();
    getMarkupSettings().setMarkupIdGenerator(myGenerator);
}
```

6.4. Creating in-line panels with WebMarkupContainer

Creating 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 *org.apache.wicket.markup.html.WebMarkupContainer* component 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 have 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 those conditions we can consider using 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.

Note also that we may later choose to make information box visible by calling *setVisible(true)*, upon for example an AJAX request (we will be covering such an example in [chapter 19.2.8](#)).

6.5. 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 small fragments of markup in a page. 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 its own markup file but it uses a markup fragment defined in the markup file of its parent 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 as 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">
        <p>News available</p>
    </wicket:fragment>
</body>
</html>
```

Java code:

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

When the page is rendered, markup inside the fragment will be inserted **inside div element**:

```
<html>
...
<body>
...
    <div wicket:id="contentArea">
        <p>News available</p>
    </div>
</body>
</html>
```

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 the user has submitted 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 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>
```

Java code:

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

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

6.6. 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 the `<wicket:head>` tag.

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



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



The content of the `<wicket:head>` tag is added to the 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>`.



The `<wicket:head>` tag 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 16](#).

6.7. Using stub markup in our pages/panels

Wicket's `<wicket:remove>` tag 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>
```

6.8. 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:

HTML 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.

6.9. Hiding decorating elements with the `wicket:enclosure` tag

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 than 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>
```



`<wicket:enclosure>` is nice and prevents that users have to add boilerplate to their application. But it is not without problems. The child components are children in the markup, but the auto-component generated for the enclosure tag will not magically re-parent the child components. Thus the markup hierarchy and the component hierarchy will be out of sync. The automatically created enclosure container will be created along side its "children" with both attached to the very same parent container. That leads to a tricky situation since e.g. *onBeforeRender()* will be called for enclosure children even if the enclosure is made invisible by its controlling child. On top auto-components cannot keep any state. A new instance is created during each render process and automatically deleted at the end. That implies that we cannot prevent *validation()* from being called, since *validation()* is called before the actual render process has started. Where any of these problems apply, you may replace the tag and manually add a [EnclosureContainer](#) which basically does the same. But instead of adding the children to the Page, Panel, whatever, you must add the children to this container in order to keep the component hierarchy in sync.

6.10. 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, borders also have their own markup file which is defined following the same rules seen for panels and pages. In this file `<wicket:border>` tag 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:body> tag will be discarded...-->
    <wicket:body>
        <div>
            foo<br />
            <wicket:body/><br />
            buz <br />

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

```

The `<wicket:body>` tag 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 the border component with method `addToBorder(Component...)`, while in the second case we must use the `add(Component...)` method.

The following example illustrates both use 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:

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

6.11. Summary

In this chapter we have seen the tools provided by Wicket to gain 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 loops (like *while* or *for*) inside our pages to achieve this result. 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 first learn more of Wicket's basics. That's why repeaters will be introduced later in [chapter 13](#).

Chapter 7. Components lifecycle

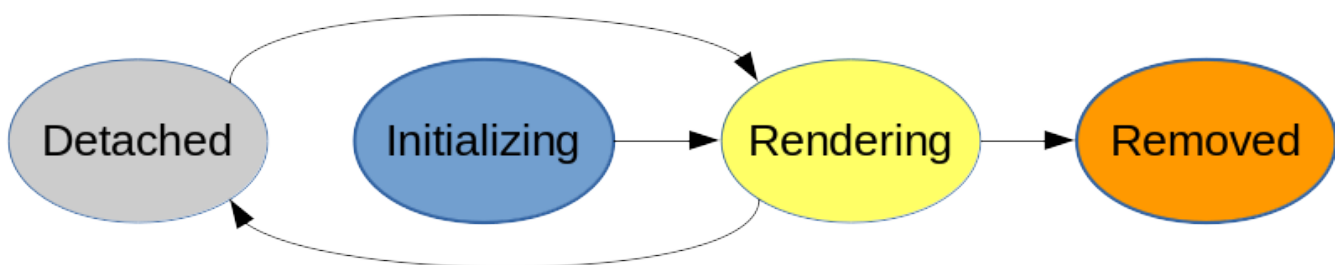
Just like applets and servlets, also Wicket components follow a lifecycle during their existence. In this chapter we will analyze each stage of this cycle and we will learn how to make the most of the hook methods that are triggered when a component moves from one stage to another.

7.1. Lifecycle stages of a component

During its life a Wicket component goes through the following stages:

1. **Initialization:** a component is instantiated and initialized by Wicket.
2. **Rendering:** components are prepared for rendering and generate markup. If a component contains children (i.e. is a subclass of *MarkupContainer*) their rendering result is included in the resulting markup.
3. **Removed:** 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.
4. **Detached:** after request processing has ended all components are notified to detach any state that is no longer needed.

The following picture shows the state diagram of component lifecycle:



Once a component has been removed it could be added again to a container, but the initialization stage won't be executed again - it is easier to just create a new component instance instead.



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.

7.2. 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. In the following table these methods are grouped according to the stage in which they are invoked (and they are sorted by execution order):

Cycle stage	Involved methods
Initialization	constructor, <i>onInitialize()</i>
Rendering	<i>onConfigure()</i> , <i>onBeforeRender()</i> , <i>renderHead()</i> , <i>onRender()</i> , <i>onComponentTag()</i> , <i>onComponentTagBody()</i> , <i>onAfterRender()</i>
Removed	<i>onRemove()</i>
Detached	<i>onDetach()</i>

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

7.3. Initialization stage

This stage is the beginning of the component lifecycle.

A component is instantiated by application code (or by Wicket in case of bookmarkable page) and added to a parental component. As soon as the component is contained in a component tree rooted in a page, a “post”-constructor *onInitialize()* is called where we can execute custom initialization of our component.

When we override this method we have to call *super.onInitialize()*, usually before anything else in that method.

7.4. Rendering stage

This stage is reached each time a component is rendered, typically when a page is requested or when the component or one of its ancestors is refreshed via AJAX.

7.4.1. Method *onConfigure*

Method *onConfigure()* has been introduced in order to provide a good point to manage the component states such as its visibility or enabled state. This method is called on all components whose parent is visible.

As stated in [chapter 6.1](#), *isVisible()* and *isEnabled()* are called multiple times when a page or a component is rendered, so it's highly recommended not to directly override these method, but rather to use *onConfigure()* to change component states. On the contrary method *onBeforeRender* (see the next paragraph) is not indicated for this task because it will not be invoked if component visibility is set to false.

7.4.2. Method *onBeforeRender*

The most important hook method of this stage is probably *onBeforeRender()*. This method is called on all visible components before any of them are rendered. It is our last chance to change a component's state prior to rendering - no change to a component's state is allowed afterwards.

If we want to add/remove child 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");

        add(firstLabel);
        add(new Link<Void>("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 the 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".



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
```

7.4.3. Method `renderHead`

This method gives all components the possibility to add items to the page header through its argument of type *org.apache.wicket.markup.head.IHeaderResponse*

7.4.4. Method `onRender`

This method does the actual rendering—you will rarely have to implement it, since most components already contain a specific implementation to produce their markup.

7.4.5. Method `onComponentTag`

Method *onComponentTag(ComponentTag)* is called to process a 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 or rename 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");
            }
        });
    }
}
```

Generated 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 of the super class because also this class may also 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 [chapter 6.2](#) how to use behavior *AttributeModifier* to manipulate the tag's attribute. In [chapter 19.1](#) we will see that base class *Behavior* offers also a callback method named *onComponentTag(ComponentTag, Component)* that can be used in place of the hook method *onComponentTag(ComponentTag)*.

7.4.6. Methods onComponentTagBody

Method *onComponentTagBody(MarkupStream, ComponentTag)* is called to process the component tag's body. Just like *onComponentTag* it takes as input a *ComponentTag* parameter representing the component tag. In addition, we also find 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 the *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 the 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);
            }
        });
    }
}

```

```
}
```

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

7.4.7. Methods *onAfterRender*

Called on each rendered component immediately after it has been rendered - *onAfterRender()* will even be called when rendering failed with an exception.

7.5. Removed stage

This stage is entered when a component is removed from its container 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 a different one. Starting from version 6.18.0 Wicket added a further hook method called *onReAdd()* which is triggered every time a previously removed component is re-added to a container. Please note that while *onInitialize()* is called only the very first time a component is added, *onReAdd()* is called every time it is re-added after having been removed.

7.6. Detached stage

When a request has finished, the page and all its contained components move to the detached stage:

The hook method *onDetach()* notifies each component that it should release all held resources no longer needed until the next request.

7.7. Summary

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

Chapter 8. Page versioning and caching

This chapter explains how Wicket manages page instances, underlining the difference between stateful and stateless pages. The chapter also introduces 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 concepts.

8.1. 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 their 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 Wicket's point of view the biggest difference between these two page types 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 the 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 call the *isPageStateless()* method of class *Page*.

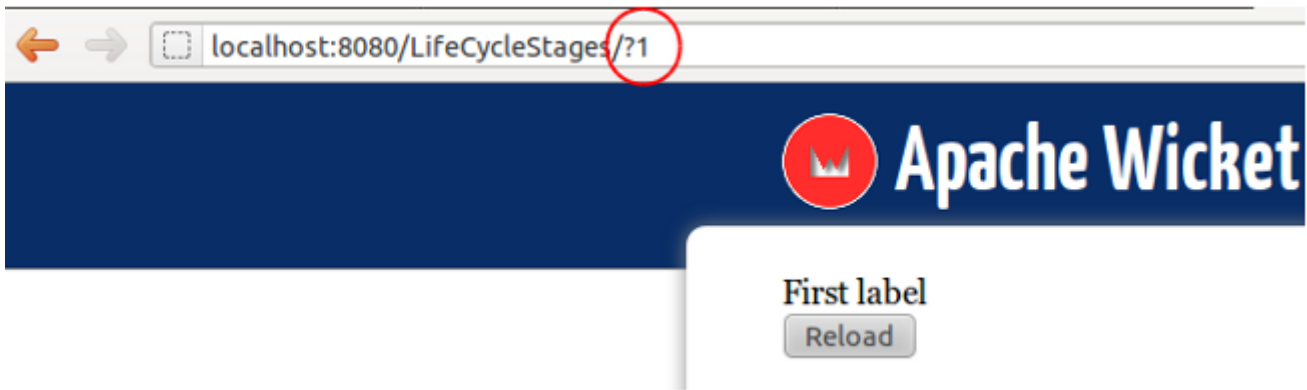
In order to build a stateless page we must comply with some rules to ensure that the page won't need to use user session. These rules are illustrated in paragraph 8.3 but before talking about stateless pages we must first understand how stateful pages are handled and why they are versioned.

8.2. Stateful pages

Stateful pages are versioned in order to support browser's back button: when this button is pressed Wicket must respond by 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 unique 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 end of URL. This number is the page id we are talking about:



In this chapter we will use a revised version of this example project where the component hierarchy is modified inside the Link's *onClick()* method. This is necessary because Wicket creates a new page version only if the 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<Void>("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 the “Reload” button, a new page version is created and the page id is increased by one:



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



For more details about page storing you can take a look at paragraph "Page storing" from chapter "Wicket Internals" The content of this paragraph is from wiki page <https://cwiki.apache.org/confluence/display/WICKET/Page+Storage>.

As we have stated at the beginning of this chapter, page versions are stored using Java serialization, therefore every object referenced inside a page must be serializable. In [paragraph 11.6](#) we will see how to overcome this limit and work with non-serializable objects in our components using detachable Wicket models.

8.2.1. Using a specific page version with PageReference

To retrieve a specific page version in our code we can use class *org.apache.wicket.PageReference* by providing its constructor with the corresponding page id:

```
//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 the method *getPage*.

8.2.2. 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)*.

8.2.3. 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](#).

We can access this class inside the method `init` of the class `Application` using the `getFrameworkSettings()` method :

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

A serializer based on Kryo library and another one based on Fast are provided by the WicketStuff project. You can find more information on this project, as well as the instructions to use its modules, in Appendix B.

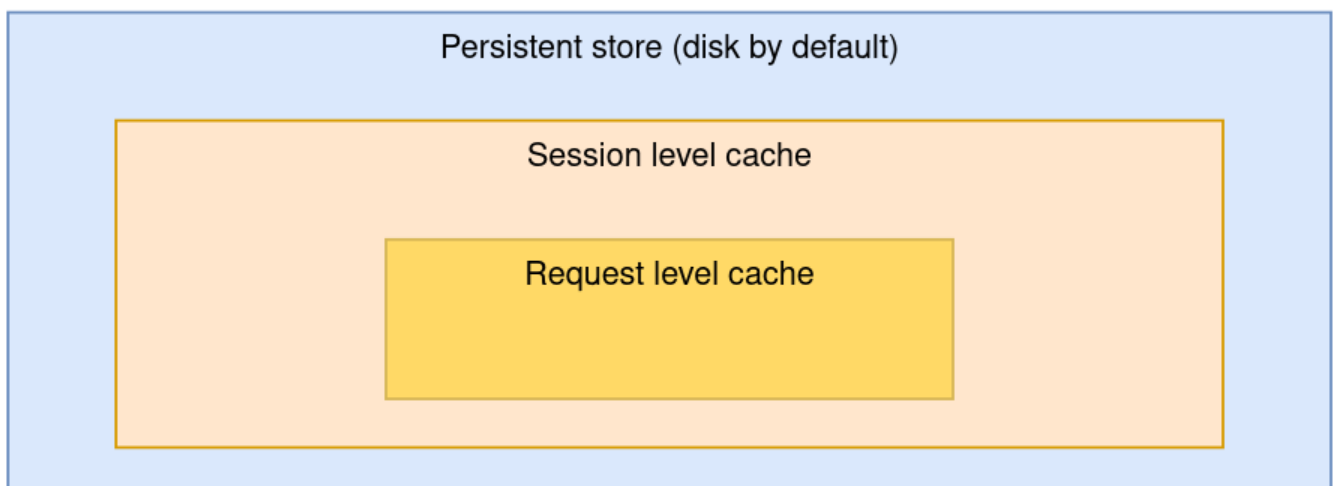
8.2.4. Page caching

By default Wicket persists versions of pages into a session-relative file on disk, but it uses a two-level cache to speed up the access. The first level of the cache contains all the pages involved in the current requests as we may visit more than one page during a single request, for example if we have been redirected with `setResponsePage`. The second level cache stores the last rendered page into a session-scoped variables.



Scoped variables will be introduced in [chapter 9.4.6](#) which is about Wicket metadata.

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



Wicket allows us to set the maximum size of the file used to store pages with setting class *org.apache.wicket.settings.StoreSettings*. This class provides the *setMaxSizePerSession(Bytes bytes)* method to set the size of the file. The Bytes parameter is the maximum size allowed for this file:

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

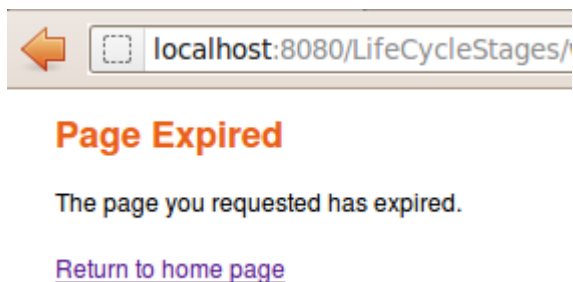
Class *org.apache.wicket.util.lang.Bytes* is an utility class provided by Wicket to express size in bytes (for further details refer to the JavaDoc).



More insights on internal page storing will be covered in [chapter 26](#)

8.2.5. Page expiration

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



This error page can be customized with the *setPageExpiredErrorPage* method of class *org.apache.wicket.settings.ApplicationSettings*:

```
@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 as input a single *PageParameters* argument (the page must be bookmarkable as described in [paragraph 10.1.1](#)).

8.3. Stateless pages

Wicket makes it very easy to build stateful pages, but sometimes we might want to use an “old school” stateless page that doesn’t keep memory of its state in the user session. For example consider the public area of a site or a login page: in those cases a stateful page would be a waste of resources or even a security threat, as we will see in [paragraph 12.10](#).

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 as input a single `PageParameters` argument (class `PageParameters` will be covered in [chapter 10.1](#)).
2. All its children components (and behaviors) 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 the visitor pattern to iterate over components and test if their method `isStateless` actually returns `true`:

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

    visitChildren((component, visit) -> {
        if(!component.isStateless()) {
            System.out.println("Component " + component.getId() + " is not stateless");
        }
    });
}
```

Alternatively, we could use the `StatelessComponent` utility annotation along with the `StatelessChecker` class (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 `StatelessComponent` annotation we must first add the `StatelessChecker` to our application as a component render listener:

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




Most of the Wicket's built-in components 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 the *setStatelessHint(true)* method. 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 log message:



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

8.4. 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 10](#) will cover these missing topics.

Chapter 9. Under the hood of the 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 in the user session.

Wicket provides 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 the web requests initiated by the user’s browser.

9.1. Class Application and request processing

Beside configuring and initializing our application, the *Application* class 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.

9.2. Request and Response classes

The *Request* and *Response* classes are located in package *org.apache.wicket.request* and they provide an abstraction of the concrete request and response used by our web 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 located inside the package *org.apache.wicket.protocol.http.servlet*. *ServletWebRequest* and *ServletWebResponse* wrap respectively a *HttpServletRequest* and a *HttpServletResponse* object. If we need to access these low-level objects, we can call *Request*’s method *getContainerRequest()* and *Response*’s method *getContainerResponse()*.

9.3. 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 via the *setRequestCycleProvider* method.

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 4.2.2](#)) and *org.apache.wicket.Session* in order to get the application and the session in use by the current thread.



The implementation of the *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 the *getRequestCycle()* method which is a convenience method that internally invokes *RequestCycle.get()*:

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

9.3.1. RequestCycle and request processing

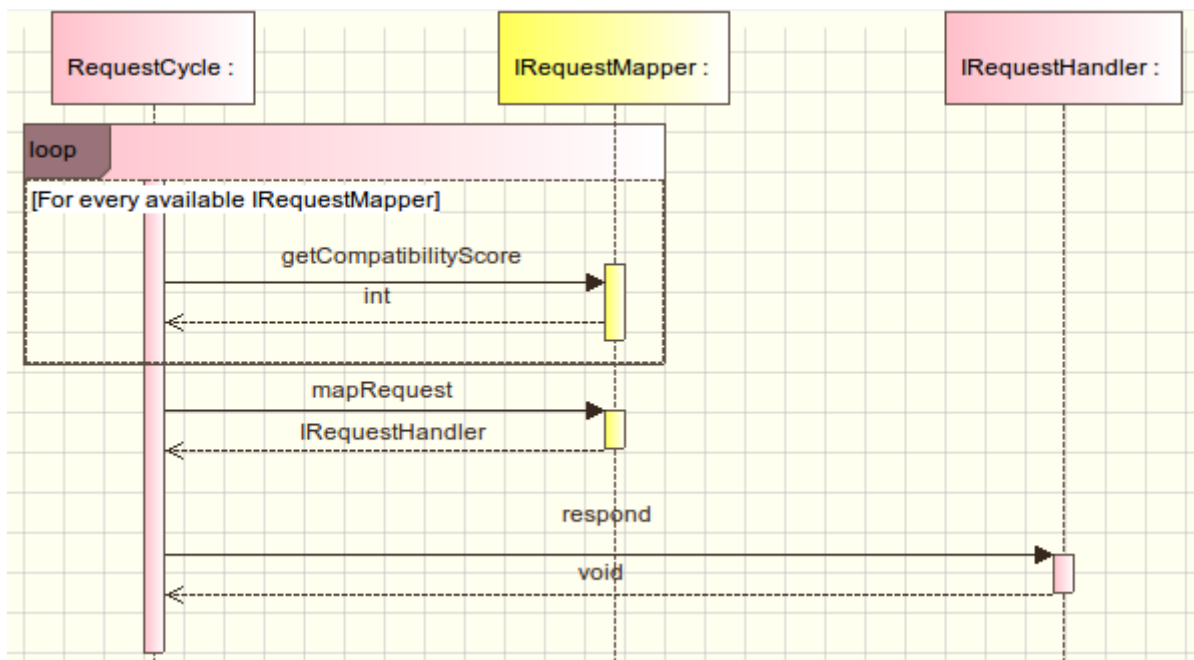


This paragraph will provide just the basic informations about what happens behind the scenes of request processing. When you work with Wicket it's unlikely to have a need for customizing this process, so we won't cover this topic in detail.

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 HTTP request, the *RequestCycle* uses a set of objects implementing the *org.apache.wicket.request.IRequestMapper* interface. The mapping interface defines the *getCompatibilityScore(Request request)* method 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 *mapRequest(Request request)* method 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 the *RequestCycle*:



Developers can create additional implementations of *IRequestMapping* and add them to their application via the *mount(IRequestMapping mapper)* method of the *WebApplication* class. In paragraph 10.6 we will see how Wicket uses this method to add built-in mappers for mounted pages.

9.3.2. Generating URL with the *urlFor* and *mapUrlFor* methods

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

- a page class, via the *urlFor(Class<C> pageClass, PageParameters parameters)* method
- an *IRequestHandler* via the *urlFor(IRequestHandler handler)* method
- a *ResourceReference* via the *urlFor(ResourceReference reference, PageParameters params)* method (resource entities will be introduced in [chapter 19](#)).

The overloaded *urlFor* method from above also has 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).

9.3.3. Method *setResponsePage*

The *RequestCycle* class contains the implementation of the *setResponsePage* method we use to redirect a user to a specific page (see [paragraph 4.4](#)). 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);
}
  
```

9.3.4. RequestCycle's hook methods and listeners

The *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 the *RequestCycle* is about to start handling the request.
- **onEndRequest():** called when the *RequestCycle* has finished to handle the request
- **onDetach():** called after the request handling has completed and the *RequestCycle* is about to be detached from its thread. The default implementation of this method invokes *detach()* on the current session (the *Session* class will be shortly discussed in paragraph 9.4).

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 the 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 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 the request cycle listeners we must add them to our application which in turn will pass them to the new *RequestCycle*'s instances created with *createRequestCycle* method:

```
@Override
public void init() {

    super.init();

    IRequestCycleListener myListener;
```

```
//listener initialization...
getRequestCycleListeners().add(myListener)
}
```

The `getRequestCycleListeners` method 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](#).

9.4. Session Class

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 8.2), etc...

In addition, we know from paragraph 8.1 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 the `Application` class with the `newSession(Request request, Response response)` method. This method is not declared as final, hence it can be overridden if we need to use a custom implementation of the `Session` class.

By default if our custom application class is a subclass of `WebApplication`, method `newSession` will return an instance of class `org.apache.wicket.protocol.http.WebSession`. As we have mentioned talking about `RequestCycle`, also class `Session` provides a static `get()` method which returns the session associated to the current thread.

9.4.1. Session and listeners

Similar to the `RequestCycle`, class `org.apache.wicket.Session` also offers support for listener entities. With `Session` these entities must implement the callback interface `org.apache.wicket.ISessionListener` which exposes only the `onCreated(Session session)` method. 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
public void init(){

    super.init();

    //listener initialization...
    ISessionListener myListener;
    //add a custom session listener
    getSessionListeners().add(myListener)

}
```

9.4.2. Handling session attributes

The Session class 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 the 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 the WebApplication's method getSessionAttributePrefix().

9.4.3. Accessing the 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 might 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:

Sessions Administration for /admin/myApp

Tips:

- Click on a column to sort.
- To view a session details and/or remove a session attributes, click on its id.

Active HttpSession informations

Refresh Sessions list 1 active Sessions

Session Id	Type	Guessed Locale	Guessed User name	Creation Time
<input type="checkbox"/> BC56322A3DEF48E8B568B086F97F7FFE	Primary	ENGLISH	Mr BadGuy	2012-06-15 12:04:00

Invalidate selected Sessions

Return to main page

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 via Wicket’s Session class because they would not 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");
```

9.4.4. Temporary and permanent sessions

Wicket doesn’t need to store data into user session as long as the 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 the `isTemporary()` method:

```
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 the `getId()` method. This value will be *null* if the 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 the user to another page which displays with a label the session attribute previously created. The code of the two pages is as follows:

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 bind() method which converts temporary session into a permanent one. If the home page has not invoked this method, the session with its attribute would have been discarded at the end of the request and the page *DisplaySessionParameter* would have displayed an empty value in its label.

9.4.5. Discarding session data

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



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

9.4.6. Storing arbitrary objects with metadata

JavaServer Pages Specification¹ 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 in 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 declared 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 the *setMetaData* method which takes two parameters as input: the key used to store the value and the value itself. 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 the *getMetaData(MetadataKey<T> key)* method. In the following example we set a *java.sql.Connection* object in the 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...
    /**
     * Metadata key definition
     */
    public static MetadataKey<Connection> connectionKey = new MetadataKey<> (){};

    /**
     * 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 the metadata:

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

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

9.5. Exception handling

Wicket uses a number of custom exceptions during the regular running of an application. We have already seen *PageExpiredException* raised when a page version is expired. Other examples of such

exceptions are *AuthorizationException* and *RestartResponseException*. We will see them later in the next chapters. All the other exceptions raised during rendering phase are handled by an implementation of *org.apache.wicket.request.IExceptionMapper* which by default is class *org.apache.wicket.DefaultExceptionMapper*. If we are working in DEVELOPMENT mode this mapper will redirect us to a page that shows the exception stacktrace (page *ExceptionHandlerPage*). On the contrary, if application is running in DEPLOYMENT mode *DefaultExceptionMapper* will display an internal error page which by default is *org.apache.wicket.markup.html.pages.InternalErrorPage*. To use a custom internal error page we can change application settings like this:

```
getApplicationSettings().setInternalErrorPage(MyInternalErrorPage.class);
```

We can also manually set if Wicket should display the exception with *ExceptionHandlerPage* or if we want to use the internal error page or if we don't want to display anything at all when an unexpected exception is thrown:

```
//show default developer page
getExceptionSettings().setUnexpectedExceptionDisplay(
ExceptionSettings.SHOW_EXCEPTION_PAGE );
//show internal error page
getExceptionSettings().setUnexpectedExceptionDisplay(
ExceptionSettings.SHOW_INTERNAL_ERROR_PAGE );
//show no exception page when an unexpected exception is thrown
getExceptionSettings().setUnexpectedExceptionDisplay(
ExceptionSettings.SHOW_NO_EXCEPTION_PAGE );
```

Developers can also decide to use a custom exception mapper instead of *DefaultExceptionMapper*. To do this we must override *Application*'s method *getExceptionMapperProvider*:

```
@Override
public IProvider<IExceptionMapper> getExceptionMapperProvider()
{
    //...
}
```

The method returns an instance of *org.apache.wicket.util.IProvider* that should return our custom exception mapper.

9.5.1. Ajax requests

To control the behavior in Ajax requests the application may use *org.apache.wicket.settings.ExceptionSettings1*.

setAjaxErrorHandlingStrategy(ExceptionSettings.AjaxErrorStrategy). By default if an error occurs during the processing of an Ajax request Wicket will render the configured error page. By configuring *org.apache.wicket.settings.ExceptionSettings.AjaxErrorStrategy2.INVOKE_FAILURE_HANDLER* as the default strategy the application will call the JavaScript *onFailure* callback(s) instead.

9.6. 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 23](#).

Chapter 10. 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 quite similar to a “click” event handler (see [paragraph 4.4](#)).

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 a basic `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.

10.1. PageParameters

A common practice in web development is to pass data to a page using query string parameters (like `?paramName1=paramValue1¶mName2=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.

10.1.1. PageParameters and bookmarkable pages

Base class `Page` comes with a constructor which takes as input a `PageParameters` instance. If we use

this superclass constructor in our page, `PageParameters` will be used to build the page URL and it can be retrieved at a later time with the Page's `getPageParameters()` method.

In the following example taken from the `PageParametersExample` project we have a home page with a link to a second page that uses a version of `setResponsePage` method that takes as 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<Void>("pageWithIndexParam") {

    @Override
    public void onClick() {

        PageParameters pageParameters = new PageParameters();
        pageParameters.add("foo", "foo");
        pageParameters.add("bar", "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&bar=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 explained in paragraph 8.3, 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 the 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 the page class (`org.wicketTutorial.PageWithParameters` in our example). Segment `wicket` is another fixed prefix added by default during URL generation. In paragraph 10.6 we will see how to customize fixed prefixes with a custom implementation of `IMapperContext` interface.

10.1.2. Indexed parameters

Besides named parameters, Wicket also supports 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 by the `PageParameters` class. 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 the `PageParameters`. The following is an example 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.

10.2. Bookmarkable links

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

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

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

10.3. Automatically creating bookmarkable links with tag `wicket:link`

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

```
<!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:

```
<a disabled="disabled">HomePage</a>
```

The markup used to render disabled links can be customized using the markup settings (class `org.apache.wicket.settings.MarkupSettings`) available in the application class:

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

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

10.4. External links

Since Wicket uses plain HTML markup files as templates, we can place an anchor to an external page directly inside the markup file. 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 the href attribute 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 a dynamic value for the text inside the anchor, we can pass it as an 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>
```

10.5. 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 *org.apache.wicket.markup.html.link.StatelessLink* component 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 the `onClick()` method can not depend on instance variables. To illustrate this potential issue let's consider the following code (from the 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 the user clicks on the `statelessLink` link.

10.6. Generating structured and clear URLs

Having structured URLs in our site is a basic requirement if we want to build an efficient SEO 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 `IMapperContext` and `IPageParametersEncoder` interfaces. In the next paragraphs we will learn both of these two techniques.

10.6.1. Mounting a single page

With Wicket we can mount a page to a given path in much the same way as we map a servlet filter to a desired path inside file `web.xml` (see [paragraph 4.2](#)). Using `mountPage(String path, Class<T> pageClass)` method of the `WepApplication` class we tell Wicket to respond with a new instance of `pageClass` whenever a user navigates to the given path. In the application class of the project `MountedPagesExample` we mount `MountedPage` to the `"/pageMount"` path:

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

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

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

Under the hood the `mountPage` method mounts an instance of the 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 the Application's method `mount` have been introduced in the previous chapter ([paragraph 9.3](#)).

10.6.2. 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 the named parameter `foo` that must be provided when the 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 the '#' 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
```

10.6.3. Mounting a package

In addition to mounting a single page, Wicket allows to mount all of 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 the `MountedPagesExample` project we have mounted all pages inside the subpackage `org.tutorialWicket.subPackage` with this line of code:

```
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 the `mountPage` method, the implementation of the `mountPackage` method mounts an instance of *org.apache.wicket.request.mapper.PackageMapper* to the given path.

10.6.4. 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 non-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 10.1.1](#)).
- 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 16](#).

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 the `newMapperContext()` method declared in the `Application` class and make it return our custom implementation of `IMapperContext`:

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

10.6.5. 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 the URL.

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

This built-in encoder is `org.apache.wicket.request.mapper.parameter.UrlPathPageParametersEncoder` class. In the `PageParametersEncoderExample` project we have manually mounted a `MountedMapper` that takes as 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 the `MountedPage` web page. The code of the link and the resulting page URL are:

Link code:

```
add(new Link<Void>("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
```

10.6.6. 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 making it more vulnerable to malicious users.

To avoid this kind of security threat we can use the *CryptoMapper* request mapper which wraps an existing mapper and encrypts the original URL producing a single 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);
}
```

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.



By default *CryptoMapper* encrypts page URLs with a cipher that might not be strong enough for production environment. Paragraph ["Security with Wicket"](#) will provide a more detailed description of how Wicket encrypts page URLs and we will see how to use stronger ciphers.

10.7. 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

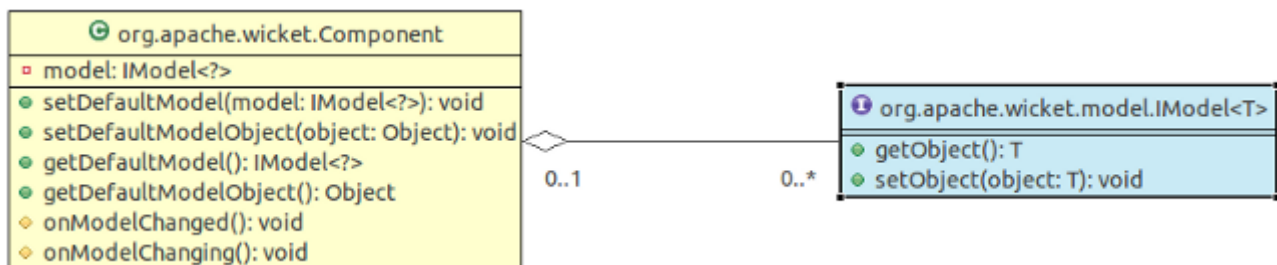
parameters to target pages with PageParameters. In the second part of the chapter we focused on mounting pages to a specific path and on controlling how parameters are encoded by Wicket. Finally, we have also seen how to encrypt URLs to prevent security vulnerabilities.

Chapter 11. Wicket models and forms

In Wicket the concept of “model” 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 forms. That’s why we haven’t talked about models so far, and why this chapter discusses these two topics together.

11.1. What is a model?

Model is essentially a [facade](#) 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*:



The main goal of *IModel* interface is to decouple components from concrete details about the persistence strategy adopted for their data. In order to achieve this level of abstraction *IModel* defines the two methods required to get and set a data object: *getObject()* and *setObject()*. The level of indirection introduced by models allows access data object only when it is really needed (for example during the rendering phase) and not earlier when it may not be ready to be used. In addition to *getObject()* and *setObject()*, *IModel* defines a richer set of methods, mostly meant to work with Java 8 lambdas. We will introduce them in the next paragraph.

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 the 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 as 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 mention label model in the previous examples. Here is the code of this constructor:

```
public Label(final String id, String label) {
    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 in the web session, and we know from chapter 6 that data are stored into session using serialization.



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

Just like any other Wicket components, `Label` provides a constructor that takes as 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!")));
```

The `Model` class comes with a bunch of factory methods that makes it easier to build new model instances. For example the `of(T object)` method creates a new instance of `Model` which wraps any `Object` instance 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 these factory methods in our examples.

It's quite clear that if our `Label` must display a static text it doesn't make much sense to build 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 provided by a user or a value read from a database. Wicket models are designed to solve these kinds of problems.

By default the 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](#) (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 a model, we can tell the `Component` class not to escape characters by calling the `setEscapeModelStrings(false)` method.

11.2. IModel and Lambda

With Wicket 8 `IModel` has been extended with new methods to fully leverage lambdas. The most

interesting thing of the new version of *IModel* is that it provides a default implementation for all of its methods (included *setObject()*), with the only exception of *getObject()*. In this way *IModel* is eligible as functional interface and this greatly simplify the creation of custom models. As long as we need to display a static text it doesn't make much sense building a custom model, but if we need to display a dynamic value (like the input provided by a user or a value read from a database), defining a model with a lambda expression comes quite in handy.

Let's say we need a label to display the current time stamp each time a page is rendered. This could be a possible solution:

```
add(new Label("timeStamp", () -> java.time.LocalDate.now()));
```

As mentioned above, method *setObject()* comes with a default implementation. The code is the following:

```
default void setObject(final T object)
{
    throw new UnsupportedOperationException(
        "Override this method to support setObject(Object)");
}
```

This means that models obtained using *IModel* as lambda expressions are *read-only*. When we work with forms we need to use a model that support also data storing. In the next paragraph we will see a couple of models shipped with Wicket that allow us to easily use JavaBeans as backing objects.

11.2.1. Lambda Goodies

Most of the default methods we find in *IModel* are meant to leverage Lambda expressions to transform model object. The following is a short reference for such methods:

- **filter(predicate):** Returns a *IModel* checking whether the predicate holds for the contained object, if it is not null. If the predicate doesn't evaluate to true, the contained object will be null. Example:

```
//the filtered model will have a null model object if person's name
//is not "Jane"
IModel<Person> janeModel = Model.of(person)
    .filter((p) -> p.getName().equals("Jane"));
```

- **map(mapperFunction):** Returns an *IModel* applying the given mapper to the contained object, if it is not null. Example:

```
//the new read-only model will contain the person's first name
IModel<String> personNameModel = Model.of(person).map(Person::getName);
```

- **flatMap(mapperFunction)**: Returns an *IModel* applying the given *IModel*-bearing mapper to the contained object, if it is not null. Example:

```
//returns a read/write model for person's first name
//NOTE: LambdaModel will be discussed later.
IModel<String> personNameModel = Model.of(person).flatMap(targetPerson ->
LambdaModel.of(
    () -> targetPerson::getName, targetPerson::setName
));
```

- **combineWith(otherModel, combiner)**: Returns an *IModel* applying the given combining function to the current model object and to the one from the other model, if they are not null. Example:

```
IModel<String> hello = Model.of("hello");
IModel<String> world = Model.of("world");
IModel<String> combinedModel = hello.combineWith(
    world, (thisObj, otherObj) -> thisObj + " " + otherObj);

assertEquals("hello world", combinedModel.getObject());
```

- **orElseGet(supplier)**: Returns a read-only *IModel* using either the contained object or invoking the given supplier to get a default value. Example:

```
IModel<String> nullObj = new Model();
assertEquals("hello!", nullObj.orElseGet(() -> "hello!"));
```

11.3. 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 the 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
 */
}

```

11.3.1. PropertyModel

Let's say we want to display the name field of a Person instance with a label. We could, of course, use the 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 always display the current value of a class field, we should use the *org.apache.wicket.model.PropertyModel* model class:

```

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.Property* Resolver which can access any kind of property, private fields included.

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

```

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

```



`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 keys can be also specified using squared brackets:

```
children[0].name ...
mapField[key].subfield ...
```

11.3.2. LambdaModel

PropertyModel uses textual expressions to resolve object properties. That's nice but it comes with some drawbacks. For example the expression can not be checked at compile time and is not refactoring-friendly. To overcome these problems with Wicket 8 a new kind of lambda-based model has been introduced: *org.apache.wicket.model.LambdaModel*. This model uses lambda expressions to get/set model object. Here is the signature of its constructor:

```
public LambdaModel(SerializableSupplier<T> getter, SerializableConsumer<T> setter)
```

In the following code we use method references to operate on a specific object property:

```
Person person = new Person();
IModel<String> personNameModel = new LambdaModel<>(person::getName, person::setName);
```

As we have seen for *Model* also *LambdaModel* comes with factory method *LambdaModel.of*:

```
Person person = new Person();
IModel<String> personNameModel = LambdaModel.of(person::getName, person::setName);
```

11.3.3. CompoundPropertyModel and model inheritance

Class *org.apache.wicket.model.CompoundPropertyModel* is a particular kind of model which is usually used in conjunction with another Wicket feature called model inheritance. With this feature, when a component needs to use a model but none 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 containers 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 a further example let's say we want to extend the code above to display all of 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. The 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 the 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` is particularly useful when used in combination with Wicket forms, as we will see in the next paragraph.



Model is referred to as static model because the result of its method `getObject` is fixed and it is not dynamically evaluated each time the method is called. In contrast, models like `PropertyModel` and `CompoundProperty Model` are called dynamic models.

11.4. Wicket forms

Web applications use HTML forms to collect user input and send it to the server. Wicket provides *`org.apache.wicket.markup.html.form.Form`* class to handle web forms. This component must be bound to `<form>` tag. 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>
```

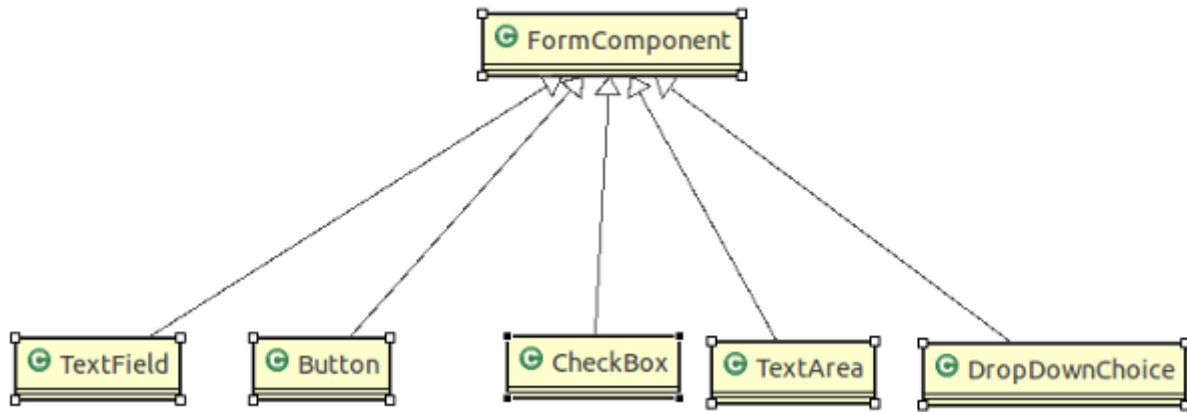
Java code:

```
Form<Void> form = new Form<Void>("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 does not have 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`*.

11.4.1. Form and models

A form should contain some input fields (like text fields, check boxes, radio buttons, drop-down 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`*:



The purpose of `FormComponent` is to store the corresponding user input into its model when the form is submitted. The form is responsible for mapping input values to the corresponding components, avoiding us the burden of manually synchronizing models with input fields and vice versa.

11.4.2. Login form

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



The topic of security will be discussed later in chapter 22. 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. 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's constructor we build the three components used in the form and we assign them a model containing an empty string:

```

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 the `<input>` tag with the attribute `type` set to `"text"`. `PasswordTextField` is a subtype of `TextField` and it must be used with an `<input>` tag with the attribute `type` set to `"password"`. For security reasons component `PasswordTextField` cleans its value at each request, so it will be always empty after the 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. 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);
        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 (the LoginFormRevisited project):

```

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 its `CompoundPropertyModel`. This allows children components to have direct access to form fields and use them as backing objects, without explicitly creating a model for themselves.



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 the id of the visited child. 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 spouseContainer = new WebMarkupContainer("spouse");
Label name;
spouseContainer.add(name = new Label("name"));

add(spouseContainer);

```

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, its container (WebMarkup Container with id 'spouse') doesn't own a model, hence the request for a model is forwarded to the parent container, which in this case is the page. In the end the 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.

11.5. 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 `<select>` tag:

Html:

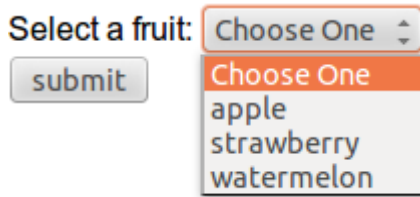
```
<form wicket:id="form">
```

```
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 `CompoundPropertyModel` 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 via the `setNullValid` method. Please note that the placeholder text ("Chose one") can be localized, as we will see in chapter 15. 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 the user is obtained by 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 the option id and the one to display to user. Wicket has solved this problem with `org.apache.wicket.markup.html.form.IChoiceRender` interface. 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 the DropDownChoice using one of its constructors that accepts this type of parameter (like we did in the example above) or after its creation invoking *setChoiceRenderer* method.

11.6. Model chaining

Models that extend *org.apache.wicket.model.ChainingModel* can be used to build a chain of models. These 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](#). 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:



List of persons Choose One

Name:

Surname:

Address:

Email:

Save

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. The DropDownChoice component can be configured to automatically update its model each time we change the selected item on the client side. All we have to do is to add a *FormComponentUpdatingBehavior* to it: The behavior will submit the components value every time JavaScript event "change" occurs, and its model will be consequently updated. To leverage this functionality, the form component 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>
    </div>
    <input type="submit" value="Save"/>
  </form>
</body>
```

The initialization code for DropDownChoice is the following:

```
Model<Person> listModel = new Model<Person>();
ChoiceRenderer<Person> personRenderer = new ChoiceRenderer<Person>("fullName");
personsList = new DropDownChoice<Person>("persons", listModel, loadPersons(),
personRenderer);
personsList.add(new FormComponentUpdatingBehavior());
```

As choice renderer we have used the basic implementation provided with the `org.apache.wicket.markup.html.form.ChoiceRenderer` class 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 the Model class.

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

```
public class PersonListDetails extends WebPage {
    private Form<Void> 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);
        personsList.add(new FormComponentUpdatingBehavior());

        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()
    //...
}
```

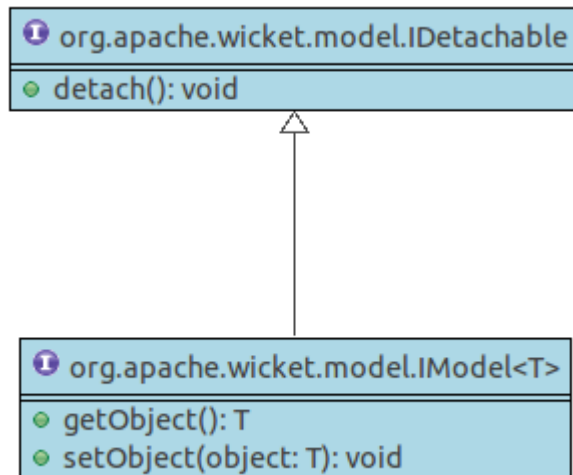
The two models work together as a pipeline where the output of method getObject of Model is the model object of CompoundPropertyModel. As we have seen, model chaining allows us to combine the actions of two or more models without creating new custom implementations.

11.7. Detachable models

In chapter 6 we have seen how Wicket uses serialization to store page instances. When an object is serialized, all its referenced objects are recursively serialized. For a page this means that all its children components, their related models as well as the model objects 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 a POJO as backing object, but [POJOs](#) are ordinary objects with no prespecified interface, annotation or superclass, hence they are not required to implement the standard Serializable interface.

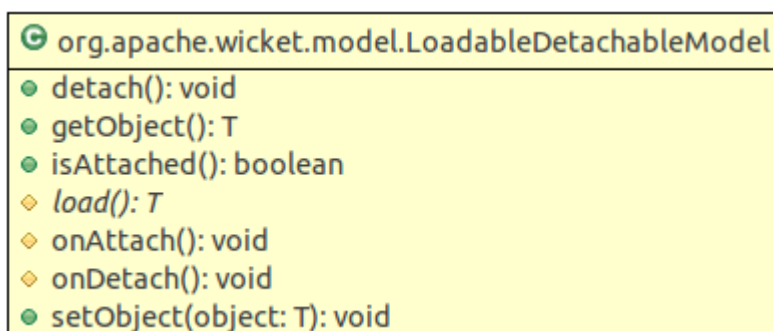
To cope with these problems IModel extends another interface called IDetachable.



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 (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 the `Model` class). 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 a model object which is initialized the first time `getObject()` is called to process a request. The concrete data loading is delegated to abstract method `T load()`. The reference to a model object is automatically set to null at the end of the request by the `detach()` method.

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 via [JPA](#). To understand the following code a basic knowledge of JPA is

required even if we won't go into the detail of this standard.



The following model is provided for example purposes only and is not intended to be used in production environment. Important aspects such as 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);
        }
        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 as 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 used by the load method.

`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 lose these changes we must explicitly persist the entity before the detaching phase occurs.



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

11.8. 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 the Component's `onDetach` method 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.

11.9. 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. Taking your first steps with Wicket you may be tempted to pass raw objects to your components instead of using models:

```
/**
 *
 * NOT TO DO: passing raw 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 a bad practice and you must avoid it. By using models we do not only decouple our components from the data source, but we can also rely 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 we force model objects to be serialized.

11.10. Summary

Models are at the core of Wicket and they are the basic ingredient needed to taste the real power of the 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.

Chapter 12. Wicket forms in detail

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

One of such features is the ability to work with nested forms (they will be discussed in [paragraph 12.6](#)).

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.

12.1. Default form processing

In [paragraph 11.3](#) we have seen a very basic usage of the 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 with 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.



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 at the step that updates the form field models.

The whole form processing is started invoking public method `process(IFormSubmitter)` (Later in [paragraph 12.5](#) we will introduce interface `IFormSubmitter`).

12.2. Form validation and feedback messages

A basic example of a validation rule is to make a field required. In [paragraph 11.4.2](#) we have already seen how this can be done calling `setRequired(true)` on a field. However, to set a validation

rule on a `FormComponent` we must add the corresponding validator to it.

A validator is an implementation of the `org.apache.wicket.validation.IValidator` interface and the `FormComponent` has a version of method `add` which takes as input a reference of 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 the inserted input will respect the email format `local-part@domain`:

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

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

12.2.1. 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 the default value.

The entire infrastructure of feedback messages is built on top of the Java internationalization (i18N) support and it uses [resource bundles](#) to store messages.



The topics of internationalization will be covered in [chapter 15](#). 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 15.2](#).

Default feedback messages (like the one above for required fields) are stored in the file `Application.properties` placed inside Wicket the `org.apache.wicket` package. Opening this file we can find the key and the localized value of the message:

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

We can note the key (Required in our case) and the label parameter written in the [expression language](#) (`${label}`). Scrolling down this file we can also find the message used by the `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).



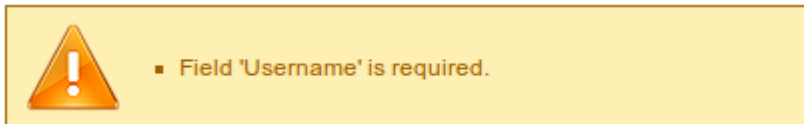
Remember that component model is updated with the user input only if validation succeeds! As a consequence, we can't retrieve the wrong value inserted for a field from its model. Instead, we should use `getValue()` method of `FormComponent` class. (This method will be introduced in the example used later in this chapter)

12.2.2. Displaying 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:



The component can be freely placed inside the page and we can set the maximum amount of displayed messages with the `setMaxMessages()` method.

Error messages can be 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 later in this chapter.

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 its constructor or using the `setFilter` method. Custom filters can be created implementing the `IFeedbackMessageFilter` interface. An example of custom filter is illustrated later in this paragraph.

12.2.3. 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:

EmailAddressValidator

Checks if input respects the format local-part@domain.

Message:

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://).

Message:

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 greater 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)).

Messages:

The value of '\${label}' is less than the minimum of \${minimum}.

The value of '\${label}' is larger than the maximum of \${maximum}.

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.

Messages:

The value of '\${label}' must be at least \${minimum}.

The value of '\${label}' must be at most \${maximum}.

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 `maxLength(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)`.

Messages:

The value of '{label}' is shorter than the minimum of {minimum} characters.

The value of '{label}' is longer than the maximum of {maximum} characters.

The value of '{label}' is not between {minimum} and {maximum} characters long.

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”).

Message:

The credit card number is invalid.

EqualPasswordInputValidator

This validator checks if two password fields have the same value.

Message:

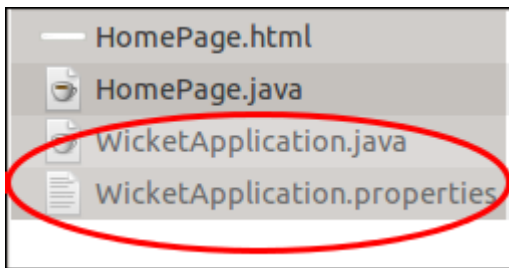
{label0} and {label1} must be equal.

12.2.4. Overriding standard feedback messages with custom bundles

If we don't like the default validation feedback 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 the 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 also reports the value that failed validation:

EmailAddressValidator=The value '\${input}' inserted for field '\${label}' is not a valid email address.

Email:

- The value 'no good' inserted for field 'email' is not a valid email address.

12.2.5. Creating custom validators

If our web application requires a complex validation logic and built-in validators are not enough, we can 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.

For the sake of simplicity, the validator of our example will check the given username against a fixed list of three existing usernames.

A custom validator must simply implement interface `IValidator`:

```
public class UsernameValidator implements IValidator<String> {
    List<String> existingUsernames = Arrays.asList("bigJack", "anonymous", "mrSmith");

    public void validate(IValidatable<String> validatable) {
        String chosenUserName = validatable.getValue();

        if(existingUsernames.contains(chosenUserName)){
            ValidationError error = new ValidationError(this);
            Random random = new Random();

            error.setVariable("suggestedUserName",
                validatable.getValue() + random.nextInt());
            validatable.error(error);
        }
    }
}
```

The only method defined inside `IValidator` is `validate(IValidatable<T> validatable)` and is invoked during validation's step. Interface `IValidatable` represents the component being validated and it can be used to retrieve the component model (`getModel()`) or the value to validate (`getValue()`).

The custom validation logic is all inside `IValidator`'s method `validate`. When validation fails a validator must use `IValidatable`'s method `error(IValidationError error)` to generate the appropriate feedback message. In the code above we used the `ValidationError` class as convenience implementation of the `IValidationError` interface which represents the validation error that must be displayed to the user. This class provides a constructor that uses the class name of the validator in input as key for the resource to use as feedback message (i.e. 'UsernameValidator' in the example). If we want to specify more than one key to use to locate the error message, we can use method `addKey(String key)` of `ValidationError` class.

In our example when 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 a variable named `suggestedUserName`. The message is inside application's properties file:

UsernameValidator=The username '\${input}' is already in use. Try with '\${suggestedUserName}'

To provide further variables to our feedback message we can use method `setVariable(String name, Object value)` of class `ValidationError` as we did in our example.

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.

12.2.6. Using flash messages

So far we have considered just the error messages generated during validation step. However `Wicket's 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<Void> form = new Form<Void>("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`).



Since version 6.13.0 Wicket provides the additional filter class `org.apache.wicket.feedback.ExactLevelFeedbackMessageFilter` to accept only feedback messages of a certain error level.

12.3. Input value conversion

Working with Wicket we will rarely need to worry about conversion between input values (which are strings because the underlying HTTP protocol) 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 `convertInput()` method. 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. The `PropertyModel` and `CompoundPropertyModel` models 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 `org.apache.wicket.util.convert.IConverter` interface 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);
```



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`) use their choice render and their collection of possible choices to perform input conversion.

12.3.1. 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:

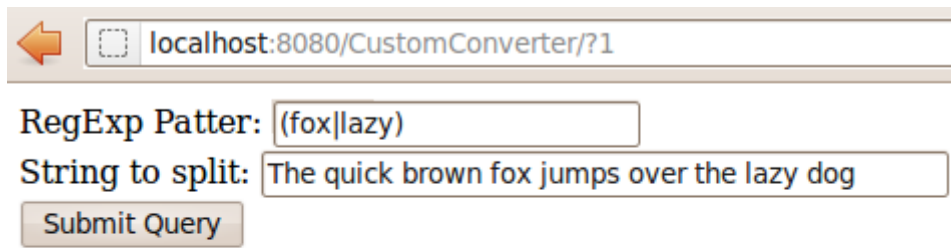
```
public ConverterLocator()
{
    set(Boolean.TYPE, BooleanConverter.INSTANCE);
    set(Boolean.class, BooleanConverter.INSTANCE);
    set(Byte.TYPE, ByteConverter.INSTANCE);
    set(Byte.class, ByteConverter.INSTANCE);
    set(Character.TYPE, CharacterConverter.INSTANCE);
    set(Character.class, 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 input a regular expression pattern and another one to input a string value that will be split with the given pattern.

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 the CustomConverter project):



localhost:8080/CustomConverter/?1

RegExp Patter: (fox|lazy)

String to split: The quick brown fox jumps over the lazy dog

Submit Query

- Tokens for the given string and pattern:
 - The quick brown
 - jumps over the
 - dog

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 as input a `Locale` parameter in order to deal with locale-sensitive data and conversions. We will learn more about locales and internationalization in [Chapter 15](#).

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 regExpPattern;
    private String stringToSplit;

    public HomePage(final PageParameters parameters) {
        TextField regExpPatternTxt;
        TextField stringToSplitTxt;

        Form<Void> form = new Form<Void>("form"){
            @Override
            protected void onSubmit() {
                super.onSubmit();
                String messageResult = "Tokens for the given string and pattern:<br/>";
                String[] tokens = regExpPattern.split(stringToSplit);

                for (String token : tokens) {
                    messageResult += "- " + token + "<br/>";
                }
                success(messageResult);
            }
        };

        form.setDefaultModel(new CompoundPropertyModel(this));
        form.add(regExpPatternTxt = new TextField("regExpPattern"));
        form.add(stringToSplitTxt = new TextField("stringToSplit"));
        add(new FeedbackPanel("feedbackMessage").setEscapeModelStrings(false));

        add(form);
    }
}

```



If the user 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`.

12.4. Validation with JSR 303

Standard JSR 303 defines a set of annotations and APIs to validate our domain objects at field-level. Wicket has introduced an experimental support for this standard since version 6.4.0 and with version 6.14.0 it has become an official Wicket module (named *wicket-bean-validation*). In this paragraph we will see the basic steps needed to use JSR 303 validation in our Wicket application. Code snippets are from example project *JSR303validation*.

In the example application we have a form to insert the data for a new *Person* bean and its relative *Address*. The code for class *Person* is the following

```

public class Person implements Serializable{

```

```

@NotNull
private String name;

//regular expression to validate an email address
@Pattern(regexp = "^[_A-Za-z0-9-]+(.[_A-Za-z0-9-]+)*[A-Za-z0-9-]+(.[A-Za-z0-9-]+)*((.[A-Za-z]{2,}){1}$)")
private String email;

@Range(min = 18, max = 150)
private int age;

@Past @NotNull
private Date birthDay;

@NotNull
private Address address;
}

```

You can note the JSR 303 annotations used in the code above to declare validation constraints on class fields. Class *Address* has the following code:

```

public class Address implements Serializable {

    @NotNull
    private String city;

    @NotNull
    private String street;

    @Pattern(regexp = "\\d+", message = "{address.invalidZipCode}")
    private String zipCode;
}

```

You might have noted that in class *Address* we have used annotation *Pattern* using also attribute *message* which contains the key of the bundle to use for validation message. Our custom bundle is contained inside *HomePage.properties*:

```

address.invalidZipCode=The inserted zip code is not valid.

```

To tell Wicket to use JSR 303, we must register bean validator on Application's startup:

```

public class WicketApplication extends WebApplication {
    @Override
    public void init(){
        super.init();

        new BeanValidationConfiguration().configure(this);
    }
}

```

```
}  
}
```

The last step to harness JSR 303 annotations is to add validator *org.apache.wicket.bean.validation.PropertyValidator* to our corresponding form components:

```
public HomePage(final PageParameters parameters) {  
    super(parameters);  
  
    setDefaultModel(new CompoundPropertyModel<Person>(new Person()));  
  
    Form<Void> form = new Form<Void>("form");  
  
    form.add(new TextField("name").add(new PropertyValidator()));  
    form.add(new TextField("email").add(new PropertyValidator()));  
    form.add(new TextField("age").add(new PropertyValidator()));  
    //...  
}
```

Now we can run our application and see that JSR 303 annotations are fully effective:

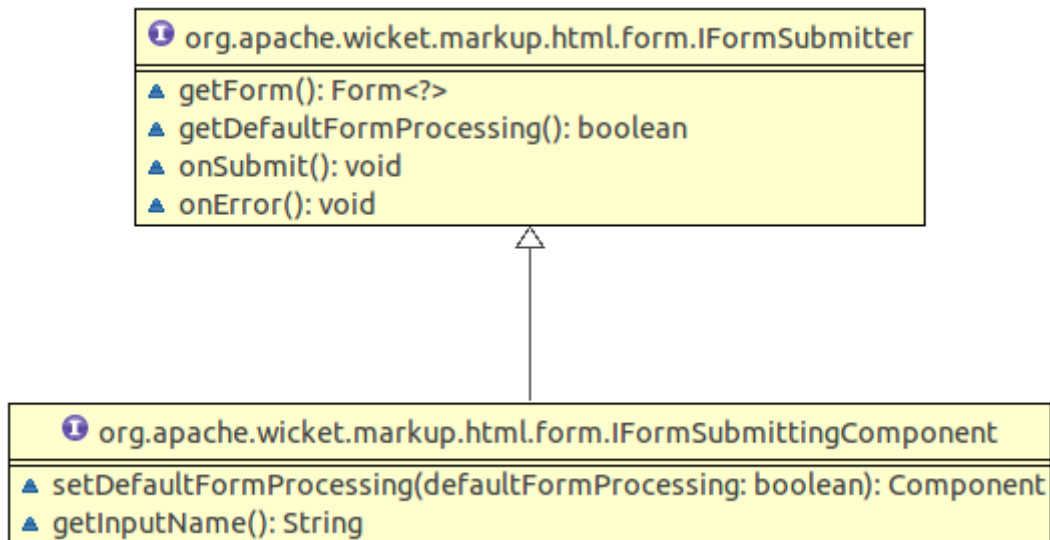
JSR303 validation form

Name	<input type="text" value="bitstorm"/>
Email	<input type="text" value="t@t.com"/>
Age	<input type="text" value="0"/>
Birthday	<input type="text"/>
Street	<input type="text" value="avenue"/>
Zip code	<input type="text" value="dsdas"/>
City	<input type="text" value="NY"/>

- 'age' must be between 18 and 150
- 'birthDay' is required.
- The inserted zip code is not valid.

12.5. 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 process method which takes as input an instance of `IFormSubmitter`. This parameter corresponds to the `IFormSubmittingComponent` clicked by a 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 by calling the 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 a hidden `<div>` tag 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, 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 form's one.



Starting with 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.

12.5.1. 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 the `<input>` or `<button>` tags. The string model received as 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 an `<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<Void> 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 the `<a>` tag but it can be also bound to any other tag that supports the event handler `onclick`. When used with the `<a>` tag, 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<Void> form = new Form<>("form");

        form.add(new PasswordTextField("password", Model.of("")));
        //specify the form to submit
        add(new SubmitLink("externalSubmitter", form));
        add(form);
    }
}

```

12.5.2. Disabling default form processing

With an `IFormSubmittingComponent` we can choose to skip the default form submission process by setting the appropriate flag to false with the `setDefaultFormProcessing` method. When the default form processing is disabled only the 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 the form processing by calling the `process(IFormSubmittingComponent)` method.

12.6. Nested forms

As you might already know, HTML doesn't allow to have nested forms. However with Wicket we can overcome this limitation by 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 in the 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.

12.7. Multi-line text input

HTML provides a multi-line text input control with `<textarea>` tag. The Wicket counterpart for this kind of control is `org.apache.wicket.markup.html.form.TextArea` component:

HTML:

```
<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 create new attribute modifiers and add them to our `TextArea` component.

12.8. File upload

Wicket supports file uploading with the `FileUploadField` component which must be used with the `<input>` tag whose type attribute must be set 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 BootstrapBasePage {
    private FileUploadField fileUploadField;

    public HomePage(final PageParameters parameters) {

        fileUploadField = new FileUploadField("fileUploadField");

        Form<Void> form = new Form<Void>("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);
        info("Upload completed!");
    } catch (Exception e) {
        e.printStackTrace();
        error("Upload failed!");
    }
}

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 to the temporary directory is inside the `onSubmit` method of the `Form` class. The uploaded file is handled with an instance of class `FileUpload` returned by the `getFileUpload()` method of the `FileUploadField` class. 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 `setMaxSize(size)` method. In the example we have set this limit to 100 kb to prevent users from uploading files bigger than this size.



The maximum size for uploaded files can also be set at application's level using the `setDefaultMaximumUploadSize(Bytes maxSize)` method of class `ApplicationSettings`:

```

@Override
public void init()
{
    getApplicationSettings().setDefaultMaximumUploadSize(Bytes.kilobytes(100));
}

```

12.8.1. Upload multiple files

If we need to upload multiple files at once and our clients support HTML5, we can still use `FileUploadField` adding attribute "multiple" to its tag. If we can not rely on HTML5, we can use the `MultiFileUploadField` component which allows the user to upload an arbitrary number of files using a JavaScript-based solution. An example showing how to use this component can be found in Wicket module `wicket-examples` in file `MultiUploadPage.java`. The live example is hosted on the [examples site](#).

12.9. Creating complex form components with `FormComponentPanel`

In [chapter 5.2.2](#) we have seen 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 in the form's submission workflow.

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 the `org.apache.wicket.markup.html.form.FormComponentPanel` component which combines the features of a `Panel` (it has its 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 a form where a user can insert a temperature that will be recorded after being converted to Kelvin degrees (see the example project `CustomFormComponentPanel`).

The Kelvin scale is wildly adopted among the scientific community and it is one of the seven base units of the [International System of Units](#) , 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 the user.

In order to implement such a component, we can make a subclass of `FormComponentPanel` and leverage the `convertInput` and `onBeforeRender` methods: in the implementation of the `convertInput` method we will convert input value to Kelvin degrees while in the implementation of `onBeforeRender` method we will take care of converting the Kelvin value to the temperature scale adopted by the 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="measurementUnit"></label>
    </wicket:panel>
</body>
</html>

```

As shown in the markup above `FormComponentPanel` uses the same `<wicket:panel>` tag used by `Panel` to define its markup. Now let's see the Java code of the new form component starting with the `onInitialize()` method:

```

public class TemperatureDegreeField extends FormComponentPanel<Double> {

    private TextField<Double> userDegree;

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

    public TemperatureDegreeField(String id, IModel<Double> model) {
        super(id, model);
    }

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

        IModel<String> labelModel = () -> getLocale().equals(Locale.US) ? "°F" : "°C";

        add(new Label("measurementUnit", labelModel));
        add(userDegree=new TextField<Double>("registeredTemperature", new
            Model<Double>()));
        userDegree.setType(Double.class);
    }
}

```

Inside the `onInitialize` method we have created a read-only model for the label that displays the letter corresponding to the user's temperature scale. To determine which temperature scale is in use, we retrieve the `Locale` from the session by calling `Component's getLocale()` method (we will talk more about this method in [Chapter 15](#)). 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 the `convertInput` and `onBeforeRender` methods:

```
// continued example
@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 `getConvertedInput()` method 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 the `getDefaultModelObject()` method, then we convert it to the temperature scale adopted by the user and finally we use this value to set the model object of the text field.

12.10. Stateless form

In [chapter 8](#) we have seen how Wicket pages can be divided into two categories: stateful and stateless. Pages that are stateless don't need to be stored in the user session and they should be used when we don't need to save any user data in the user session (for example in the public area of a site).

Besides saving resources on server-side, stateless pages can be 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 the user tries to login without a valid session assigned to him. This could happen if the user leaves the login page opened for a period of time bigger than the session's timeout and then he decides to log in. Under these conditions the user will be redirected to a 'Page expired' error page, which is not exactly a nice thing for user experience.

The second problem occurs when a malicious user or a web crawler program attempts to login into our web application, generating a huge number of page versions and consequently increasing the size of the user session.

To avoid these kinds of problems we should build a stateless login page which does not depend on a user session. Wicket provides a special version of the 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/>
      </fieldset>
    </form>
  </body>
</html>
```

```

        <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<Void> form = new StatelessForm<Void>("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.

12.11. 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:

Select a car

SUV ☐ Minivan ☐ Station wagon ☒

Select some fruits

☒ Apple ☒ Watermelon ☒ Strawberry

A check box can be used as single component to set a boolean property. For this purpose Wicket provides the *org.apache.wicket.markup.html.form.CheckBox* component which must be attached to `<input type="checkbox"/>` tag. In the next example (project SingleCheckBox) we will consider a form similar to the one used in [paragraph 11.5](#) to edit a Person object, but with an additional checkbox to let the 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:

HTML:

```
<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>

```

Java code:

```

public HomePage(final PageParameters parameters) {
    RegistrationInfo registrtionInfo = new RegistrationInfo();
    registrtionInfo.setSubscribeList(true);

    Form<Void> 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 the model object creation (with instruction `registrtionInfo.setSubscribeList(true)`):

Name:

Surname:

Address:

Email:

Subscribe list: ☒

12.11.1. Working with grouped checkboxes

When we need to display a given number of options with checkboxes, we can use the *org.apache.wicket.markup.html.form.CheckBoxMultipleChoice* component. 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:

- ☐ apple
- ☐ strawberry
- ☐ watermelon

This component can be attached to a `<div>` tag or to a `` 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 class *org.apache.wicket.model.util.ListModel* which is specifically designed to wrap a List object.

CheckBoxMultipleChoice can insert a prefix and a suffix before and after each option. To configure them we can use 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 11.5](#):

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>(new
ArrayList<String>()),
                                theSmiths, render));
```

Screenshot:

- ☐ John
- ☐ Bob
- ☐ Jill

12.11.2. 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:

What genres are you interested in?

- ☒ All of them
- ☒ Fantasy ☒ Science Fiction ☒ Children's ☒ Humour ☒ Science & Technology

Wicket comes with a couple of utility components that make it easy to implement such a feature. They are `CheckBoxMultipleChoiceSelector` and `CheckBoxSelector` classes, 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);
```

12.11.3. Working with grouped radio buttons

For groups of radio buttons we can use the *org.apache.wicket.markup.html.form.RadioChoice* component 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));
```

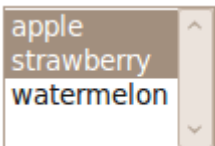
Screenshot:

☐ apple
☒ strawberry
☐ watermelon

Just like *CheckBoxMultipleChoice*, this component provides the *setPrefix* and *setSuffix* methods to configure the prefix and suffix for our options and it supports *IChoiceRender* as well.

12.12. 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 the `<select>` tag switching it to multiple-choice mode with attribute `multiple="multiple"`



Now the user can select multiple options by holding down `Ctrl` key (or `Command` key for Mac) and selecting them.

To work with multiple choice list Wicket provides the *org.apache.wicket.markup.html.form.ListMultipleChoice* component:

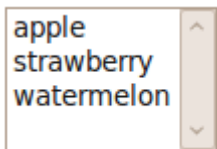
HTML:

```
<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:

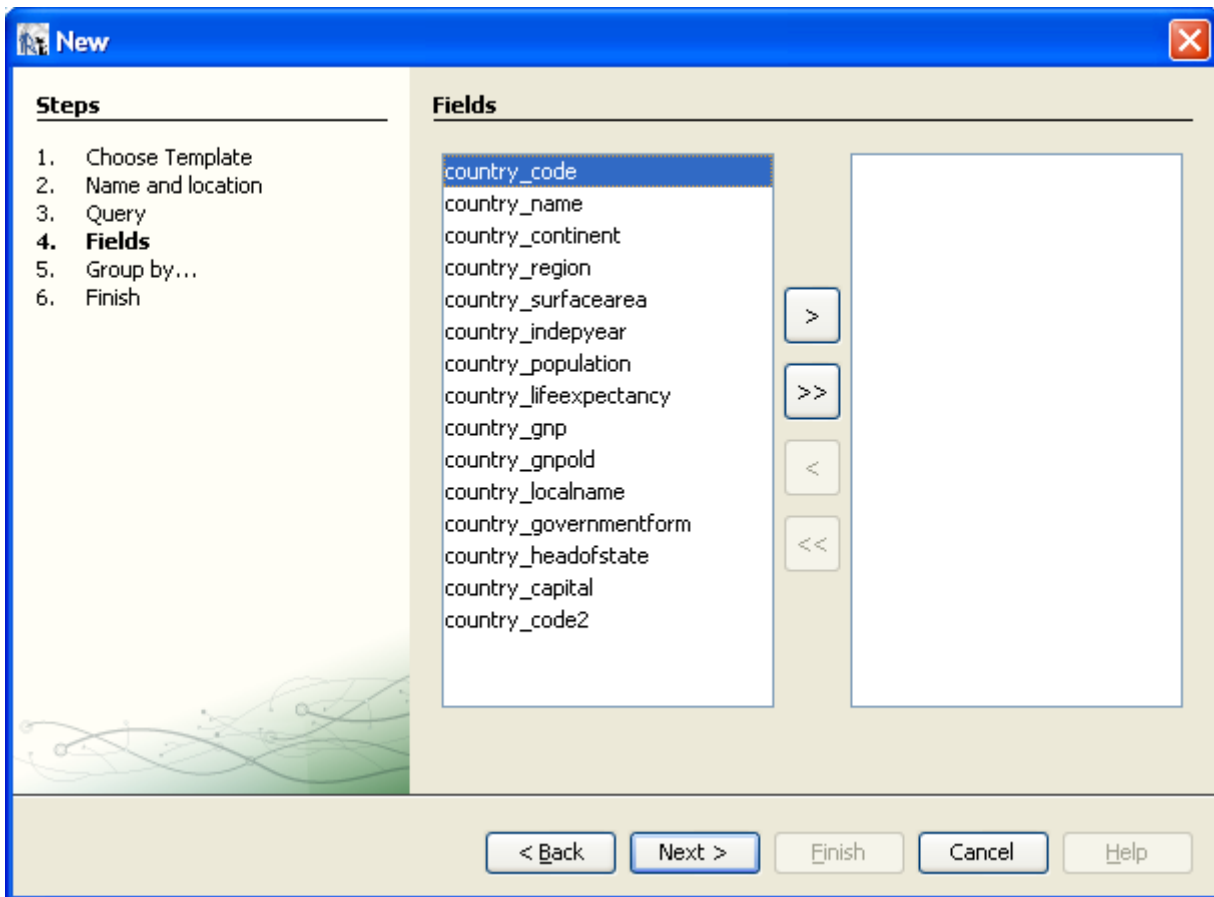


This component must be bound to a `<select>` tag but the attribute `multiple="multiple"` is not required as it will automatically be added by the component.

The number of visible rows can be set with the `setMaxRows(int maxRows)` method.

12.12.1. 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):



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 *org.apache.wicket.extensions.markup.html.form.palette.Palette* component 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:



The last two parameters of the 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/overridden 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 cellpadding="2" cellspacing="0" class="palette">
<tr>
  <td class="header headerAvailable"><span wicket:id="availableHeader">[available
header]</span></td>
  <td>&#160;</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>
```

12.13. Summary

Forms are the standard solution to let users interact with our web applications. In this chapter we have seen the three steps involved with the form processing workflow 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 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.

Chapter 13. 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.

13.1. The RepeatingView Component

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.

13.2. The ListView Component

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 type *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 the *item* object and not directly to the *ListView*.

13.2.1. 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.

13.3. The RefreshingView Component

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();
        }
    });
}
```

13.3.1. Item reuse strategy

Similar to *ListView*, the default behavior of the *RefreshingView* is to replace 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*.

13.4. 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 implement 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 13.4.2](#).

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+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*).

13.4.1. 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>Surname</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>("rows", 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.getSurname()));
        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.

13.4.2. Data paging

To enable data paging on a pageable repeater, 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. 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 is 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 [examples site](#) where you can find live examples of these components.



Wicket provides also component `PageableListView` which is a subclass of `ListView` that implements interface `IPageable`, hence it can be considered a pageable repeater even if it doesn't use interface `IDataProvider` as data source.

13.5. 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 the user.

Chapter 14. Component queueing

So far to build component hierarchy we have explicitly added each component and container in accordance with the corresponding markup. This necessary step can involve repetitive and boring code which must be changed every time we decide to change markup hierarchy. Component queueing is a new feature in Wicket 7 that solves this problem allowing Wicket to build component hierarchy in Java automatically making your code simpler and more maintainable. This chapter should serve as a short introduction to what Component Queueing is and what problems it is trying to solve.

14.1. Markup hierarchy and code

With Wicket as developers we use to define the hierarchy of components in the markup templates:

```
<form wicket:id='customer'>
  <input wicket:id='first' type='text' />
  <input wicket:id='last' type='text' />
  <div wicket:id="child">
    <input wicket:id='first' type='text' />
    <input wicket:id='last' type='text' />
    <input wicket:id='dob' type='date' />
  </div>
</form>
```

and then we repeat the same hierarchy in Java code:

```
Form<Void> form = new Form<>("customer");
add(form);

form.add(new TextField("first"));
form.add(new TextField("last"));

WebMarkupContainer child=new WebMarkupContainer("child");
form.add(child);

child.add(new TextField("first"));
child.add(new TextField("last"));
child.add(new TextField("dob"));
```

The need for the hierarchy in the markup is obvious, it is simply how the markup works. On the Java side of things it may not be immediately apparent. After all, why can we not write the code like this?

```
add(new Form<Void>("customer"));
add(new TextField("first"));
add(new TextField("last"));
```



```
WebMarkupContainer child=new WebMarkupContainer("child");
add(child);
add(new TextField("first"));
add(new TextField("last"));
add(new TextField("dob"));
```

There are a couple of reasons:

- Ambiguities that happen with duplicate ids
- Inheriting state from parent to child

We will examine these below.

14.1.1. Markup Id Ambiguities

In the example above we have a form that collects the name of a customer along with the name of their child and the child's date of birth. We mapped the name of the customer and child to form components with wicket ids *first* and *last*. If we were to add all the components to the same parent we would get an error because we cannot have two components with the same wicket id under the same parent (two components with id *first* and two with id *last*). Without hierarchy in Java we would have to make sure that all wicket ids in a markup file are unique, no small feat in a non-trivial page or panel. But, with hierarchy on the Java side we just have to make sure that no parent has two children with the same id, which is trivial.

14.1.2. Inheriting State From Parents

Suppose we wanted to hide form fields related to the child in the example above when certain conditions are met. Without hierarchy we would have to modify the *first*, *last*, and *dob* fields to implement the visibility check. Worse, whenever we would add a new child related field we would have to remember to implement the same check; this is a maintenance headache. With hierarchy this is easy, simply hide the parent container and all children will be hidden as well — the code lives in one place and is automatically inherited by all descendant components. Thus, hierarchy on the Java side allows us to write succinct and maintainable code by making use of the parent-child relationship of components.

14.1.3. Pain Points of the Java-Side Hierarchy

While undeniably useful, the Java-side hierarchy can be a pain to maintain. It is very common to get requests to change things because the designer needs to wrap some components in a *div* with a dynamic style or class attribute. Essentially we want to go from:

```
<form wicket:id='customer'>
  <input wicket:id='first' type='text' />
  <input wicket:id='last' type='text' />
```

To:

```
<form wicket:id='customer'>
  <div wicket:id='container'>
    <input wicket:id='first' type='text' />
    <input wicket:id='last' type='text' />
  </div>
```

Seems simple enough, but to do so we need to create the new container, find the code that adds all the components that have to be relocated and change it to add to the new container instead. This code:

```
Form<Void> form = new Form<>("customer");
add(form);

form.add(new TextField("first"));
form.add(new TextField("last"));
```

Will become:

```
Form<Void> form = new Form<>("customer");
add(form);

WebMarkupContainer container=new WebMarkupContainer("container");
form.add(container);

container.add(new TextField("first"));
container.add(new TextField("last"));
```

Another common change is to tweak the nesting of markup tags. This is something a designer should be able to do on their own if the change is purely visual, but cannot if it means Wicket components will change parents.

In large pages with a lot of components these kinds of simple changes tend to cause a lot of annoyance for the developers.

14.1.4. Component Queueing To The Rescue

The idea behind component queueing is simple: instead of adding components to their parents directly, the developer can queue them in any ancestor and have Wicket automatically ‘dequeue’ them to the correct parent using the hierarchy defined in the markup. This will give us the best of both worlds: the developer only has to define the hierarchy once in markup, and have it automatically constructed in Java land.

That means we can go from code like this:

```
Form<Void> form = new Form<>("customer");
add(form);
```

```

form.add(new TextField("first"));
form.add(new TextField("last"));

WebMarkupContainer child=new WebMarkupContainer("child");
form.add(child);

child.add(new TextField("first"));
child.add(new TextField("last"));
child.add(new TextField("dob"));

```

To code like this:

```

queue(new Form("customer"));
queue(new TextField("first"));
queue(new TextField("last"));

WebMarkupContainer child=new WebMarkupContainer("child");
queue(child);
child.queue(new TextField("first"));
child.queue(new TextField("last"));
child.queue(new TextField("dob"));

```



Note that we had to queue child's *first* and *last* name fields to the *child* container in order to disambiguate their wicket ids.

The code above does not look shorter or that much different, so where is the advantage?

Suppose our designer wants us to wrap the customer's first and last name fields with a *div* that changes its styling based on some condition. We saw how to do that above, we had to create a container and then reparent the two *TextField* components into it. Using queueing we can skip the second step, all we have to do is add the following line:

```

queue(new WebMarkupContainer("container"));

```

When dequeuing Wicket will automatically reparent the first and last name fields into the container for us.

If the designer later wanted to move the first name field out of the *div* we just added for them they could do it all by themselves without requiring any changes in the Java code. Wicket would dequeue the first name field into the form and the last name field into the container *div*.

14.2. Improved auto components

Auto components, such as Enclosure, are a very useful feature of Wicket, but they have always been a pain to implement and use.

Suppose we have:

```
<wicket:enclosure childId="first">
  <input wicket:id="first" type="text"/>
  <input wicket:id="last" type="text"/>
</wicket:enclosure>
```

Together with:

```
add(new TextField("first").setRequired(true).setVisible(false));
add(new TextField("last").setRequired(true));
```

When developing auto components the biggest pain point is in figuring out who the children of the auto component are. In the markup above the enclosure is a parent of the text fields, but in Java it would be a sibling because auto components do not modify the java-side hierarchy. So when the Enclosure is looking for its children it has to parse the markup to figure out what they are. This is not a trivial task.

Because auto components do not insert themselves properly into the Java hierarchy they are also hard for users to use. For example, the documentation of Enclosure does not recommend it to be used to wrap form components like we have above. When the page renders the enclosure will be hidden because *first* component is not visible. However, when we submit the form, *last* component will raise a required error. This is because *last* is not made a child of the hidden enclosure and therefore does not know its hidden — so it will try to process its input and raise the error.

Had we used *queue* instead of *add* in the code above, everything would work as expected. As part of Queueing implementation Wicket will properly insert auto components into the Java hierarchy. Furthermore, auto components will remain in the hierarchy instead of being added before render and removed afterwards. This is a big improvement because developers will no longer have to parse markup to find the children components — since children will be added to the enclosure by the dequeueing. Likewise, user restrictions are removed as well; the code above would work as expected.

14.3. When are components dequeued?

Once you call *queue()*, when are the components dequeued into the page hierarchy? When is it safe to call *getParent()* or use methods such as *isVisibleInHierarchy()* which rely on component's position in hierarchy?

The components are dequeued as soon as a path is available from *Page* to the component they are queued into. The dequeue operation needs access to markup which is only available once the Page is known (because the *Page* object controls the extension of the markup).

If the *Page* is known at the time of the *queue()* call (eg if its called inside *onInitialize()*) the components are dequeued before *queue()* returns.

14.4. Restrictions of queueing

14.4.1. Ancestors

Suppose on a user profile panel we have the following code:

```
queue(new Label("first"));
queue(new Label("last"));

WebMarkupContainer secure=new WebMarkupContainer("secure") {
    void onConfigure() {
        super.onConfigure();
        setVisible(isViewingOwnProfile());
    }
};

queue(secure);
secure.queue(new Label("creditCardNumber"));
secure.queue(new Label("creditCardExpiry"));
```

What is to prevent someone with access to markup from moving the *creditCardNumber* label out of the *secure* div, causing a big security problem for the site?

Wicket will only dequeue components either to the component they are queued to or any of its descendants.

In the code above this is the reason why we queued the *creditCardNumber* label into the *secure* container. That means it can only be dequeued into the *secure* container's hierarchy.

This restriction allows developers to enforce certain parent-child relationships in their code.

14.4.2. Regions

Dequeuing of components will not happen across components that implement the *org.apache.wicket.IQueueRegion* interface. This interface is implemented by all components that provide their own markup such as: *Page*, *Panel*, *Border*, *Fragment*. This is done so that if both a page and panel contain a component with id *foo* the one queued into the page will not be dequeued into the panel. This minimizes confusion and debugging time. The rule so far is that if a component provides its own markup only components queued inside it will be dequeued into it.

14.5. Summary

Component queueing is a new and improved way of creating the component hierarchy in Wicket 7. By having to define the hierarchy only once in markup we can make the Java-side code simpler and more maintainable.

Chapter 15. Internationalization with Wicket

In [chapter 12.2](#) 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.

15.1. Localization

As we have seen in [paragraph 12.2](#), 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.



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](#).

15.1.1. 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. 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 [paragraph 12.2](#) 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 15.3](#) 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.

15.2. 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)
```

15.2.1. 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>]]]
```

15.2.2. Using UTF-8 for resource bundles

Java uses the standard character set [ISO 8859-11](#) 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, 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`. For this reason ISO 8859-11 is being replaced with another Unicode-compliant character encoding called UTF-8. Text files created with this encoding can contain Unicode symbols in plain format. Wicket provides a useful convention to use properties file encoded with UTF-8. We just have to add prefix `.utf8.` to file extension (i.e. `.utf8.properties`).



If you want to use UTF-8 with your text files, make sure that your editor/IDE is actually using this character encoding. Some OS like Windows use a different encoding by default.

15.2.3. Using XML files as resource bundles

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">欢迎来到网站!</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 follows the same rules seen so far for bundle names.

15.2.4. 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 15.3](#). 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");
```

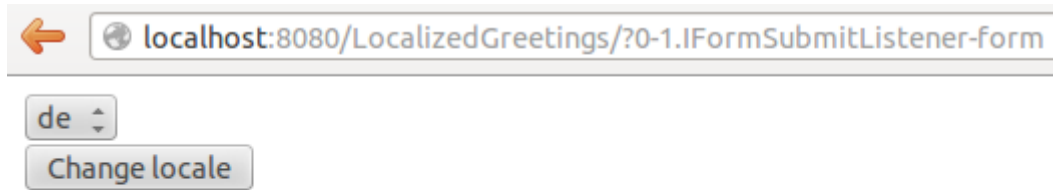
15.2.5. Localization of bundles in Wicket

In [paragraph 12.2](#) 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:



Willkommen auf der Webseite!

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:

```
IModel<String> model = () -> getString("greetingMessage");

add(new Label("greetingMessage", model));
```

The rest of the code of the home page builds the stateless form and the drop-down menu used to change the locale.

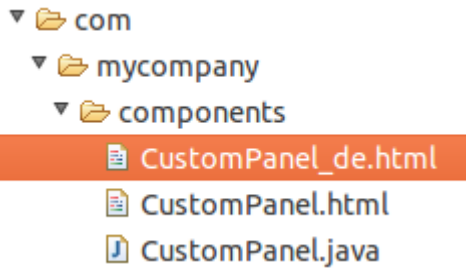
```
List<Locale> locales = Arrays.asList(Locale.ENGLISH, Locale.CHINESE, Locale.GERMAN);
final DropDownChoice<Locale> changeLocale =
    new DropDownChoice<Locale>("changeLocale", new Model<Locale>(), locales);

StatelessForm<Void> form = new StatelessForm<Void>("form"){
    @Override
    protected void onSubmit() {
        Session.get().setLocale(changeLocale.getModelObject());
    }
};

setStatelessHint(true);
add(form.add(changeLocale))
```

15.2.6. 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:



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.

15.2.7. 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>
```

By default the resource value is not escaped for HTML entities. To do that use the *escape* attribute:

```
<wicket:message key="greetingMessage" escape="true">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 comma:

```
<input type="submit" value="Preview value" wicket:message="value:key4value, title:key4title"/>
```

Finally, we can work with more complex text templates nesting components within a `wicket:message` element. For example:

```
<wicket:message key="myKey">
  This text will be replaced with text from the properties file.
  <span wicket:id="amount">[amount]</span>.
  <a wicket:id="link">
    <wicket:message key="linkText"/>
  </a>
</wicket:message>
```

```
myKey=Your balance is ${amount}. Click ${link} to view the details.  
linkText=here
```

and

```
add(new Label("amount",new Model("$5.00")));  
add(new BookmarkablePageLink("link",DetailsPage.class));
```

Results in:

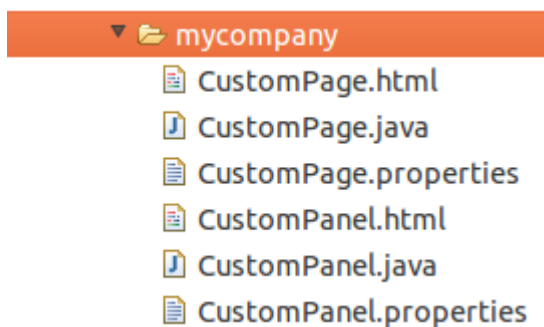
```
Your balance is $5.00. Click <a href="...">here</a> to view the details.
```

15.3. 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.

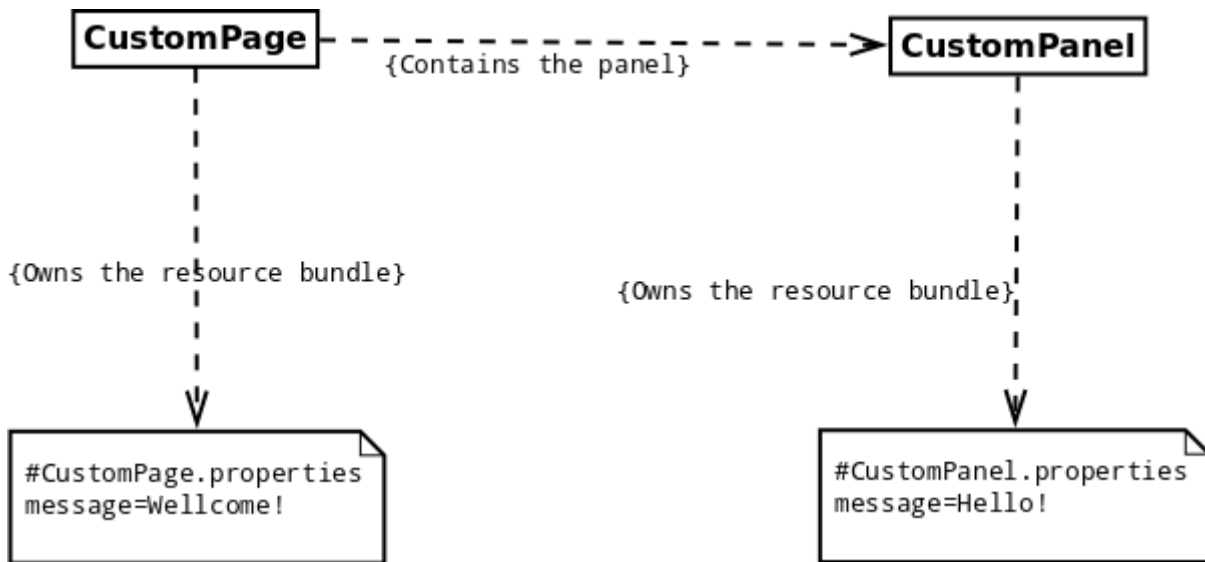
15.3.1. Localizing pages and panels

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:



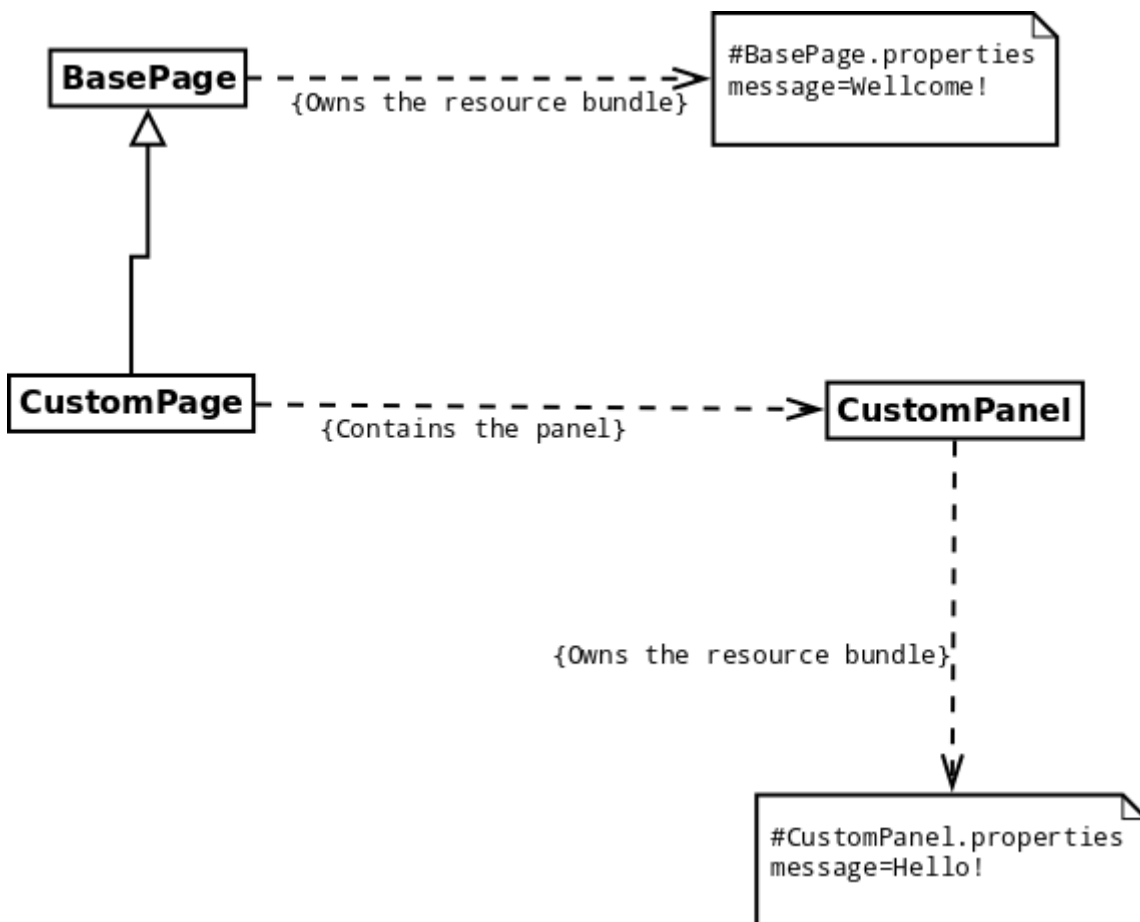
One fundamental thing to keep in mind when we work with these kinds of bundles is that the lookup algorithm gives priority to the bundles of the containers of the component that is requesting a localized resource. The more a container is higher in the hierarchy, the bigger is its priority over the other 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:



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:



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).

15.3.2. 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<Void> form = new Form<>("form");  
form.add(new Label("anotherLabel",new ResourceModel("labelValue")));  
add(form);
```

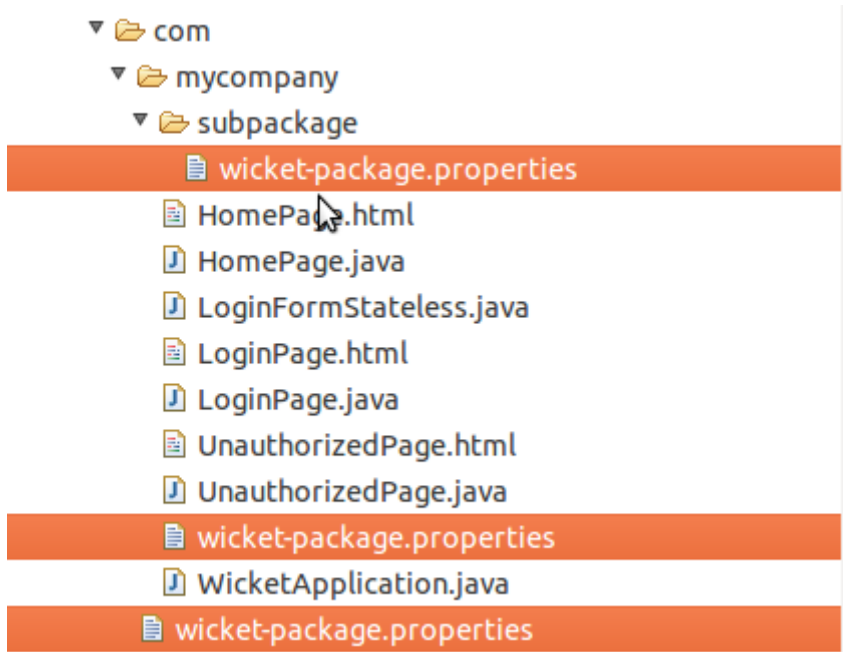
Page bundle:

```
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'.

15.3.3. Package bundles

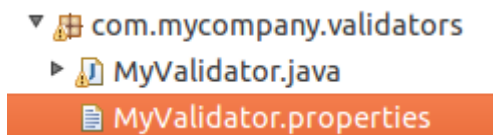
If no one of the previous steps can find a resource for the given key, the algorithm will look for package bundles. These bundles have *wicket-package* as base name and they can be placed in one of the package of our application:



Packages are traversed starting from the one containing the component requesting for a resource and going up to the root package.

15.3.4. 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:



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.

15.3.5. Extending the default lookup algorithm

Wicket implements the default lookup algorithm using the strategy pattern. 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 18.3](#).
6. **NestedStringResourceLoader:** allows to replace nested Strings and can be chained up with other resource loader

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 class `org.apache.wicket.settings.ResourceSettings`:

```
@Override
public void init()
{
    super.init();
    //retrieve ResourceSettings and then the list of resource loaders
    List<IStringResourceLoader> resourceLoaders = getResourceSettings().
                                                getStringResourceLoaders();

    //customize the list...
```

15.4. 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);

    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<Void> form = new StatelessForm<Void>("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=Wählen Sie eine Farbe
green=Grün
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 (“Select a color ”) used for null value. The resource key for this resource is 'null' or '<component id>.null' if we want to make it component-specific.

15.5. 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`.

15.5.1. ResourceModel

Model `org.apache.wicket.model.ResourceModel` acts just like the read-only model we have implemented in [paragraph 15.3](#). 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.");
```

15.5.2. 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 determine 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 as a vararg argument with method `setParameters(Object... parameters)`. 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").setParameters(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);
```

Default value is supported as well, both as string model or as string value:

```
new StringResourceModel("myKey", myComponent, myModel).setDefaultValue("default");
```

15.6. 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.

Chapter 16. 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 6.6](#) 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.

16.1. 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 16.10](#), developers can build custom resources extending base class *org.apache.wicket.request.resource.AbstractResource*.

16.2. 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.

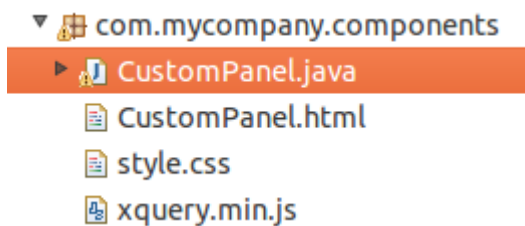
16.3. Package resources

With HTML we use to include static resources in our pages using tags like `<script>`, `<link>` or ``.

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 (a CSS and a JavaScript file in this screenshot):



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 `` 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>
```

Jave 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 classname>/<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 identifier for resources seen in [paragraph 10.6.4](#).

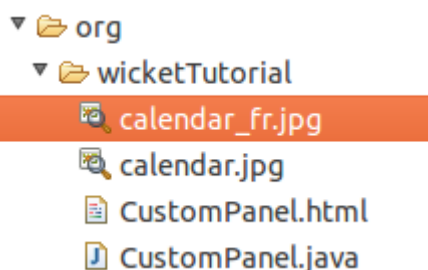
The fourth segment is the fully qualified name of the class used to locate the resource and it is the scope of the package resource. In the last segment of the URL we can find the name of the resource (the file name).

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.



Package resources can be localized following the same rules seen for resource bundles and markup files:



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`.

16.3.1. Responsive images - multiple resource references use in one component

Since Wicket 7.0.0 the build-in component *org.apache.wicket.markup.html.image.Image* allows you to add several *ResourceReferences* via varargs and to provide sizes for each image so that the browser is able to pick the best image source.

HTML:

```
...
Package resource image: <img wicket:id="packageResPicture"/>
...
```

Java Code:

```
...
    Image image = new Image("packageResPicture",
        new PackageResourceReference(getClass(), "small.jpg"),
        new PackageResourceReference(getClass(), "large.jpg"),
        new PackageResourceReference(getClass(), "medium.jpg"),
        new PackageResourceReference(getClass(), "small.jpg"));
    image.setXValues("1024w", "640w", "320w");
    image.setSizes("(min-width: 36em) 33.3vw", "100vw");

    this.add(image);
...
```

The component *org.apache.wicket.markup.html.image.Picture* is used to provide a fallback image *org.apache.wicket.markup.html.image.Image* and several source components *org.apache.wicket.markup.html.image.Source* which gives a developer the control as to when and if those images are presented to the user.

HTML:

```
...
<picture wicket:id="picture">
    <source wicket:id="big" />
    <source wicket:id="small" />
    <img wicket:id="fallback" />
</picture>
...
```

Java Code:


```

...
    Picture picture = new Picture("picture");

    Source big = new Source("big", new PackageResourceReference(getClass(),
"big.jpg"), new PackageResourceReference(getClass(), "big-hd.jpg"));
    big.setXValues("1x","2x");
    big.setMedia("(min-width: 40em)");
    picture.add(big);

    Source small = new Source("small", new PackageResourceReference(getClass(),
"small.jpg"), new PackageResourceReference(getClass(), "small-hd.jpg"));
    small.setXValues("1x","2x");
    picture.add(small);

    Image image = new Image("fallback", new PackageResourceReference(getClass(),
"fallback.jpg"));
    picture.add(image);

    this.add(picture);
...

```

16.3.2. Inline Image - embedded resource reference content

In some components like in the inline image resource references are going to be translated to other representations like base64 content.

Java Code:

```

...
    add(new InlineImage("inline", new
PackageResourceReference(getClass(),"image2.gif")));
...

```

16.3.3. Media tags - resource references with content range support

Since Wicket 7.0.0 the `PackageResource` and the `PackageResourceReference` support *Range* HTTP header for the request and *Content-Range* / *Accept-Range* HTTP headers for the response, which are used for videos / audio tags. The *Range* header allows the client to only request a specific byte range of the resource. The server provides the *Content-Range* and tells the client which bytes are going to be send.

If you want the resource not to be load into memory apply `readBuffered(false)` - this way the stream is written directly to the response. (*org.apache.wicket.resource.ITextResourceCompressor* will not be applied if `readBuffered` is set to false)

HTML:

```
...
    <video wicket:id="video" />
...
```

Java Code:

```
...
    Video video = new Video("video", new
PackageResourceReference(getClass(),"video.mp4").readBuffered(false));
...
```

16.3.4. Using package resources with tag `<wicket:link>`

In [paragraph 10.3](#) 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>
    
</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.

16.4. 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 18.1](#)) 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. The returned type *JavaScriptReferenceHeaderItem* exposes some interesting setting methods like *setDefer* and *setAsync* which can be used to set the corresponding attributes for [script tag](#).
- **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 (or the element itself), 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)*.
- **MetaDataHeaderItem:** starting from version 6.17.0, Wicket provides this class to handle meta informations such as <meta> tags or [canonical link element](#)
- **HtmlImportHeaderItem:** introduced in Wicket 6.19.0, provides a HTML5 functionality to include other wicket pages (other html files) into the current generated. Factory methods provided by this class are *forImportLinkTag* which takes the page class or the url of the page / html to be included.

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.

```
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);
    }
}
```

16.5. Context-relative resources

In web applications, it's quite common to have one or more root context folders containing css/js files. These resources are normally referenced with an absolute path inside link/script tags:

```
<script src="/misc/js/jscript.js"></script>
<link type="text/css" rel="stylesheet" href="/misc/css/themes/style.css" />
```

To handle this kind of resources from code we can use resource reference class *org.apache.wicket.request.resource.ContextRelativeResourceReference*. To build a new instance of this class we must specify the root context path of the resource we want to use:

```
ContextRelativeResourceReference resource = new
ContextRelativeResourceReference("/misc/js/jscript.js");
```

By default when our application runs in DEPLOYMENT mode *ContextRelativeResourceReference* will automatically load the minified version of the specified resource using 'min' as postfix. In the example above it will load '/misc/js/jscript.min.js'. We can force *ContextRelativeResourceReference* to always use the not-minified resource passing an additional flag to class constructor:

```
//it will always use '/misc/js/jscript.js'
ContextRelativeResourceReference resource = new
ContextRelativeResourceReference("/misc/js/jscript.js", false);
```

The minified postfix can be customized with an optional string parameter:

```
//it will use '/misc/js/jscript.minified.js' in DEPLOYMENT mode
ContextRelativeResourceReference resource = new
ContextRelativeResourceReference("/misc/js/jscript.js", "minified");
```

ContextRelativeResourceReference is usually used with the header item classes we have seen before in this chapter to create entries for the page header section.

16.5.1. Picture files

For picture files Wicket provides a specific component with class *org.apache.wicket.markup.html.image.ContextImage* which is meant to be used with tag

```
//build the component specifying its id and picture's context path
ContextImage image = new ContextImage("myPicture", "/misc/imgs/mypic.png");
```

16.6. Resource dependencies

Class *ResourceReference* allows to specify the resources it depends on overriding method *getDependencies()*. The method returns a list of *HeaderItemS* 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 jqueryuiUrl = Url.parse("https://ajax.googleapis.com/ajax/libs/jqueryui/" +
                             "1.10.2/jquery-
                             ui.min.js");

UrlResourceReference jqueryuiRef = new UrlResourceReference/jqueryuiUrl){
    @Override
    public List<HeaderItem> getDependencies() {
        Application application = Application.get();
        ResourceReference jqueryRef = application.getJavaScriptLibrarySettings().
            getJQueryReference();

        return Arrays.asList(JavaScriptHeaderItem.forReference/jqueryuiRef));
    }
};
```

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.



Wicket already provides base class *org.apache.wicket.resource.JQueryPluginResourceReference* for those JavaScript resources that depend on JQuery. This class uses the JQuery version bundled with Wicket.



The same method *getDependencies()* is defined also for class *HeaderItem*.

16.7. Aggregate multiple resources with resource bundles

One of the best practices to make our web application faster and reduce its latency is to reduce the number of requests to the server to load page resources like JavaScript or CSS files. To achieve this goal some JavaScript-based build tools (like Grunt) allow to merge multiple files used in a page into a single file that can be loaded in a single request. Wicket provides class *org.apache.wicket.ResourceBundles* to aggregate multiple resource references into a single one. A resource bundle can be declared during application initialization listing all the resources that compose it:

```

@Override
public void init() {
    super.init();

    getResourceBundles().addJavaScriptBundle(WicketApplication.class,
        "jqueryUis",
        jqueryJsReference,
        jqueryUisReference);

    getResourceBundles().addCssBundle(WicketApplication.class,
        "jqueryUiCss",
        jqueryCssReference,
        jqueryUiCssReference);
}

```

To declare a new resource bundle we need to provide a *scope* class (*WicketApplication.class* in our example) and an unique name. Now, when one of the resources included in the bundle is requested, the entire bundle is rendered instead.



A specific resource reference can not be shared among different resource bundles (i.e. it can be part of only one bundle).

16.8. Put JavaScript inside page body

Some web developers prefer to put their `<script>` tags at the end of page body and not inside the `<head>` tags:

```

<html>

<head>
//no <script> tag here...
</head>

<body>
...
<script>
//one or more <script> tags at the end of the body
</script>
</body>
</html>

```

In Wicket we can achieve this result providing a custom *IHeaderResponseDecorator* to a our application and using Wicket tag `<wicket:container/>` to indicate where we want to render our scripts inside the page. Interface *IHeaderResponseDecorator* defines method *IHeaderResponse decorate(IHeaderResponse response)* which allows to decorate or add functionalities to Wicket *IHeaderResponse*. Our custom *IHeaderResponseDecorator* can be registered in the application via

the method *getHeaderResponseDecorators*. Anytime Wicket creates an instance of *IHeaderResponse*, it will call the registered *IHeaderResponseDecorators* to decorate the header response.

In the example project *ScriptInsideBody* we can find a custom *IHeaderResponseDecorator* that renders CSS into the usual `<head>` tag and put JavaScript header items into a specific container (tag `<wicket:container/>`) Wicket already comes with class *JavaScriptFilteredIntoFooterHeaderResponse* which wraps a *IHeaderResponse* and renders in a given container all the instances of *JavaScriptHeaderItem*. The following code is taken from the Application class of the project:

```
//...
@Override
public void init()
{
    getHeaderResponseDecorators().add(response -> new
JavaScriptFilteredIntoFooterHeaderResponse(response, "footer-container"));
}
```

As you can see in the code above the *bucket* that will contain JavaScript tags is called *footer-container*. To make a use of it the developer have to add a special component called *HeaderResponseContainer* in his page:

```
add(new HeaderResponseContainer("someId", "filterName"));
```

Please note that *HeaderResponseContainer*'s needs also a name for the corresponding header response's filter. The markup of our page will look like this:

```
<html>

<header>
<!-- no <script> tag here... -->
</header>

<body>
<h1 id="click-me">Click me!</h1>
<!-- here we will have our JavaScript tags -->
<wicket:container wicket:id="someId"/>
</body>
</html>
```

The code of the home page is the following:

```
public HomePage(final PageParameters parameters) {
    super(parameters);

    add(new HeaderResponseContainer("footer-container", "footer-container"));
```

```

    }

    @Override
    public void renderHead(IHeaderResponse response) {
        response.render(JavaScriptHeaderItem.forReference(new
            PackageResourceReference(getClass(),
                "javascriptLibrary.js")));

        response.render(OnEventHeaderItem.forScript("'click-me'", "click",
            "alert('Clicked me!')"));
    }

```

Looking at the code above you can note that our page adds two script to the header section: the first is an instance of *JavaScriptHeaderItem* and will be rendered in the *HeaderResponseContainer* while the second will follow the usual behavior and will be rendered inside `<head>` tag.

16.9. Header contributors positioning

Starting from version 6.15.0 we can specify where header contributors must be rendered inside `<head>` tag using the placeholder tag `<wicket:header-items/>`:

```

<head>
  <meta charset="UTF-8"/>
  <wicket:header-items/>
  <script src="my-monkey-patch-of-wicket-ajax.js"></script>
</head>

```

With the code above all header contributions done by using *IHeaderResponse* in your Java code or the special `<wicket:head>` tag will be put between the `<meta>` and `<script>` elements, i.e. in the place of `<wicket:header-items/>`.

This way you can make sure that some header item is always before or after the header items managed by Wicket.

`<wicket:header-items/>` can be used only in the page's `<head>` element and there could be at most one instance of it.

16.10. 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 framework and its utility classes.

As hinted above in [paragraph 16.1](#), 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");

        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.

16.11. 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 10.6.1](#):

```
@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);
}
```

16.12. Lambda support

Since interface *IResource* is marked as functional interface, a custom resource can also be implemented with a simple lambda expression that consumes a *IResource.Attributes* parameter:

```
IResource helloWorldRes = (attributes) ->
    attributes.getResponse().write("Hello world!");
```

Lambda expressions come in handy also with *ResourceReference* factory methods *of* that accept a resource supplier as argument. Let's say we want to mount the resource of the previous example.

Using lambdas the code looks like this:

```
@Override
public void init() {
    super.init();

    IResource helloWorldRes = (attributes) ->
        attributes.getResponse().write("Hello world!");

    ResourceReference resRef = ResourceReference.of("helloworld", () -> helloWorldRes);

    mountResource("/helloworld", resRef);
}
```

As first argument for factory methods we can specify the name of the resource reference or a key for it (an instance of *ResourceReference.Key*)

16.13. 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).



Package resources are also application-shared resources but they don't need to be explicitly registered.



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*.

16.14. 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 class *ResourceSettings*:

```
//init application's method
@Override
public void init(){
    //get the resource locator
    getResourceSettings().getResourceStreamLocator();
    //set the resource locator
    getResourceSettings().setResourceStreamLocator(myLocator);
}
```

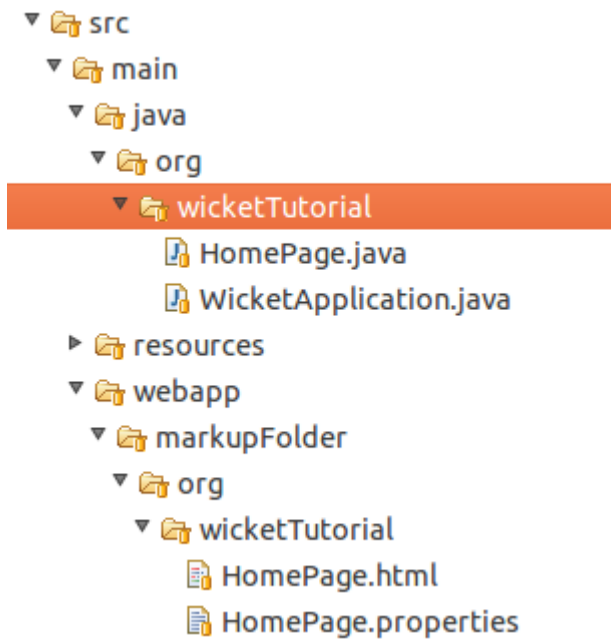
The default locator used by Wicket is class *ResourceStreamLocator* which in turn tries to load a requested resource using a set of implementations of interface *IResourceFinder*. This interface defines method *find(Class class, String pathname)* which tries to resolve a resource corresponding to the 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:



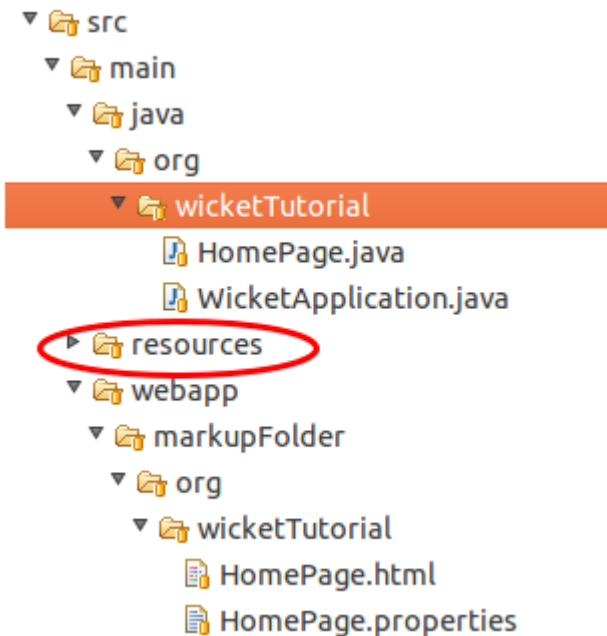
As we can see in the picture above, we must preserve the package structure also in the custom folder used as resource container. The code used inside application class to configure WebApplicationPath is the following:

```
@Override
public void init()
{
    getResourceSettings().getResourceFinders().add(
        new WebApplicationPath(getServletContext(), "markupFolder"));
}
```

Method getResourceFinders() defined by setting class ResourceSettings returns the list of resource finders defined in our application. The constructor of WebApplicationPath takes in input also an instance of standard interface jakarta.servlet.ServletContext which can be retrieved with WebApplication's method getServletContext().



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.

16.15. CssHeaderItem and JavaScriptHeaderItem compression

Introduced in Wicket 6.20.0 / Wicket 7.0.0 there is a default way to be used in which the output of all `CssHeaderItems` / `JavaScriptHeaderItems` is modified before they are cached and delivered to the client. You can add a so called Compressor by receiving the resource settings and invoke `setJavaScriptCompressor(...)` / `setJavaScriptCompressor(...)`. If you want to add several Compressors use `org.apache.wicket.resource.CompositeCssCompressor` or `org.apache.wicket.resource.CompositeJavaScriptCompressor`

Java Code:

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

    @Override
    public void init()
    {
        super.init();
        getResourceSettings().setCssCompressor(new CssUrlReplacer());
    }
}
...
```

In the previous example you see that a *org.apache.wicket.resource.CssUrlReplacer* is added which does not compress the content, but replaces all urls in CSS files and applies a Wicket representation for them by automatically wrapping them into *PackageResourceReferences*. Here is an example where you can see what Wicket does with the url representation.

HomePage (in package my/company/): **Java Code:**

```
...
response.render(CssReferenceHeaderItem.forReference(new
PackageResourceReference(HomePage.class, "res/css/mycss.css")));
...
```

mycss.css (in package my/company/res/css/): **CSS:**

```
...
body{
    background-image:url('../images/some.png');
}
...
```

some.png (in package my/company/res/images/):

Output of mycss.css: **CSS:**

```
...
body{
    background-image:url('../images/some-ver-1425904170000.png');
}
...
```

If you add a url which looks like this `background-image:url('../images/some.png?embedBase64');` Wicket is going to embed the complete image as base64 string with its corresponding mime type into the css file. It looks like the following code block demonstrates.

Output of mycss.css: **CSS:**

```
...
body{
    background-image: url(data:image/png;base64,R0lGODlh1wATAX....);
}
...
```

16.16. NIO resources

The `FileSystemResourceReference` comes along with the `FileSystemResource`,

`FileSystemResourceStreamReference` and the `FileSystemResourceStream`. Those classes provide a simple way to handle resources with Java's NIO API in Wicket starting from JDK version 7.0. (Available since Wicket 7.2.0 / Wicket 8.0.0)

Example: To include a resource which is zipped into a file and located in a specific folder in the file system you can simply write code like this:

Java:

```
URI uri = URI.create("jar:file:///videosFolder/videos.zip!/folderInZip/Video.mp4");
Path path = FileSystemResourceReference.getPath(uri);
FileSystemResourceReference ref = new FileSystemResourceReference("video",path);
Video video = new Video("video",ref);
add(video);
```

HTML:

```
<video wicket:id="video"/>
```

Using `FileSystemResourceReference` mounted:

Java:

```
mountResource("/filecontent/${name}", new FileSystemResourceReference("filesystem")
{
    private static final long serialVersionUID = 1L;

    @Override
    public IResource getResource()
    {
        return new FileSystemResource()
        {
            private static final long serialVersionUID = 1L;

            protected ResourceResponse newResourceResponse(Attributes attributes)
            {
                try
                {
                    String name = attributes.getParameters().get("name").toString("");
                    URI uri = URI.create(
                        "jar:file:///folder/example.zip!/zipfolder/" + name);
                    return createResourceResponse(
                        FileSystemResourceReference.getPath(uri));
                }
                catch (IOException | URISyntaxException e)
                {
                    throw new WicketRuntimeException("Error while reading the file.",
e);
                }
            }
        }
    }
});
```



```

        }
    };
}
});

```

`FileSystemResourceReference.getPath(uri)` uses a `FileSystemPathService` to setup a path the resource reference can work on.

So if you write a custom file system you can easily handle every path by adding a **`org.apache.wicket.resource.FileSystemPathService`** text file into **`META-INF/services`** and put in your implementation.

A reference implementation can be found in the java class `org.apache.wicket.resource.FileSystemJarPathService`.

Further `FileSystemProviders` and the corresponding `FileSystems` can be implemented as described here:

<http://docs.oracle.com/javase/7/docs/technotes/guides/io/fsp/filesystemprovider.html>

16.17. Resources derived through models

Another way to receive external image resources is to use the corresponding component with a model which contains the target URL.

The `ExternalImage` and `ExternalSource` components which are available since Wicket 7.2.0 / Wicket 8.0.0 fulfill that task.

The following example demonstrates the usage of a `CompoundPropertyModel` with the model object `ImageSrc`. The model object, bound to surrounding component / page, contains an attribute named `url` which is read by the component:

Java:

```

ImageSrc imageSrc = new ImageSrc();
imageSrc.setUrl("http://www.google.de/test.jpg");
setDefaultModel(new CompoundPropertyModel<>(imageSrc));
add(new ExternalImage("url"));

```

HTML:

```

<img wicket:id="url" />

```

The `ExternalImage` can also be constructed by passing in a `Model` (`src`) and `Model of List` (`srcSet`). For `ExternalSource` only the `srcSet` is available.

16.18. 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.

Chapter 17. An example of integration with JavaScript

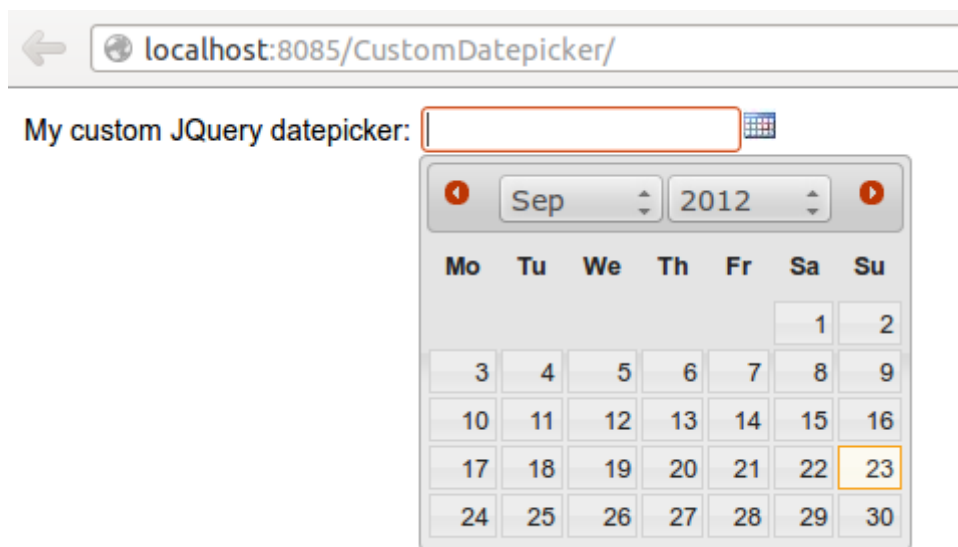
It's time to put into practice what we have learnt so far in this guide. To do this we will build a custom date component consisting of a text field to edit a date value and a fancy calendar icon to open a JavaScript datepicker. This chapter will also illustrate an example of integration of Wicket with a JavaScript library like [jQuery](#) and its child project [jQuery UI](#).

17.1. 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, 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:



On Internet you can find different libraries that already offer a strong integration between Wicket and JQuery. 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.

17.1.1. What features we want to implement

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.

17.2. ...and how we will do it

Our new component will extend the built-in text field *org.apache.wicket.extensions.markup.html.form.DateField* 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.

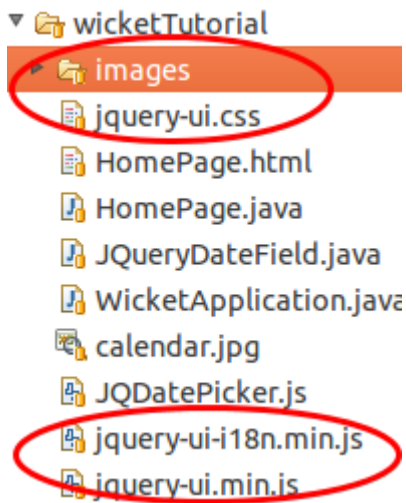
To make JQuery UI available we should instead go to its official site, download the required artifacts and use them as package resources of our component.

17.2.1. 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>.

17.2.2. Initialization code

The component is based on *LocalDateTextField* which supports pattern-based conversion for dates. The initialization code is the following:

```
public class JQueryDateField extends LocalDateTextField {

    /**
     *
     */
    private static final long serialVersionUID = 5088998263851588184L;
    private final String datePattern;
    private final String countryIsoCode;
    private CharSequence urlForIcon;
    private static final PackageResourceReference JQDatePickerRef =
        new PackageResourceReference(JQueryDateField.class,
        "JQDatePicker.js");

    public JQueryDateField(String id, IModel<LocalDate> dateModel,
        String datePattern, String countryIsoCode){
        super(id, dateModel, datePattern);
        this.datePattern = datePattern;
        this.countryIsoCode = countryIsoCode;
    }

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

        setOutputMarkupId(true);

        PackageResourceReference resourceReference = new
        PackageResourceReference(getClass(), "calendar.jpg");

        urlForIcon = urlFor(resourceReference, new PageParameters());

        add(AttributeModifier.replace("size", "12"));
    }
    ...
}
```

The first thing we 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, to generate the relative URL for the calendar icon we load its file as package resource reference and then we use *Component*'s method `urlFor` to get the URL value (we have seen this method in [paragraph 9.3.2](#)).



as you might have noted in the constructor we need to pass the ISO language code along with the date pattern. The ISO code will be used to generate the JavaScript

for the calendar.

17.2.3. 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
    JavaScriptLibrarySettings 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 class *JavaScriptLibrarySettings* 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.



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.



Our datepicker is not ready yet to be used with AJAX. In [chapter 19](#) we will see how to modify it to make it AJAX-compatible.

17.3. Summary

In this brief chapter we have seen how custom components can be integrated with [DHTML](#)

Chapter 18. 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.

18.1. 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 asteriskHtml = new StringBuffer(200);

        if(component 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.

18.2. Generating callback URLs with IRequestListener

With Wicket it's quite easy to build a callback URL that is handled on server side by a component or a behavior. What we have to do is to implement interface *org.apache.wicket.IRequestListener*:

```

public interface IRequestListener extends IClusterable
{

    /**
     * Does invocation of this listener render the page.
     *
     * @return default {@code true}, i.e. a {@link RenderPageRequestHandler} is
     schedules after invocation
     */
    default boolean rendersPage()
    {
        return true;
    }

    /**

```

```

    * Called when a request is received.
    */
    void onRequest();
}

```

Method *onRequest* is the handler that is executed to process the callback URL while *rendersPage* tells if the whole page should be re-rendered after *onRequest* has been executed (if we have a non-AJAX request).

An example of a component that implements *IRequestListener* can be seen in the Wicket standard link component. Here is an excerpt from its code:

```

public abstract class Link<T> extends AbstractLink implements IRequestListener ...
{
    /**
     * Called when a link is clicked.
     */
    public abstract void onClick();

    /**
     * THIS METHOD IS NOT PART OF THE WICKET API. DO NOT ATTEMPT TO OVERRIDE OR CALL
     IT.
     *
     * Called when a link is clicked. The implementation of this method is currently
     to simply call
     * onClick(), but this may be augmented in the future.
     */
    @Override
    public void onRequest()
    {
        // Invoke subclass handler
        onClick();
    }
}

```

Callback URLs can be generated with *Component*'s method *urlForListener(PageParameters)* or with method *urlForListener(Behavior, PageParameters)* if we are using a request listener on a component or behavior respectively (see the following example).

Project *CallbackURLExample* contains a behavior (class *OnChangeSingleChoiceBehavior*) that implements *org.apache.wicket.IRequestListener* to update the model of an *AbstractSingleSelectChoice* component when user changes the selected option (it provides the same functionality as *FormComponentUpdatingBehavior*). The following is the implementation of *onRequest()* 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 (modifying 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:

```

public CharSequence getCallbackUrl() {
    if (boundComponent == null) {
        throw new IllegalArgumentException(

```

```

        "Behavior must be bound to a component to create the URL");
    }

    return boundComponent.urlForListener(this, new PageParameters());
}

```

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:

Radio choices. Model value: Green

☐ Red

☐ Blue

☒ Green

☐ Yellow

Select choices. Model value: Red

Red



Implementing interface *IRequestListener* makes a behavior stateful because its callback URL is specific for a given instance of component.

18.3. 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 an 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:

Click on the links below to send an event to the container in the middle using one of the supported broadcast methods.

A panel at the bottom of the page will display the order in which the event has been received by sinks (page, components, session and application).

[Breadth mode](#)
[Depth mode](#)
[Bubble mode](#)
[Exact mode](#)

I'm the container in the middle

I'm the inner component.

- I'm the container in the middle and I received an event.
- I'm the page and I received an event.
- I'm the session and I received an event.
- I'm the application and I received an event.

18.4. 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 placed in one of the '/META-INF/wicket/' folder visible to the application classpath. 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 a 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 properties file. To overcome this limit we can create a main initializer that in turn executes every initializer we need:

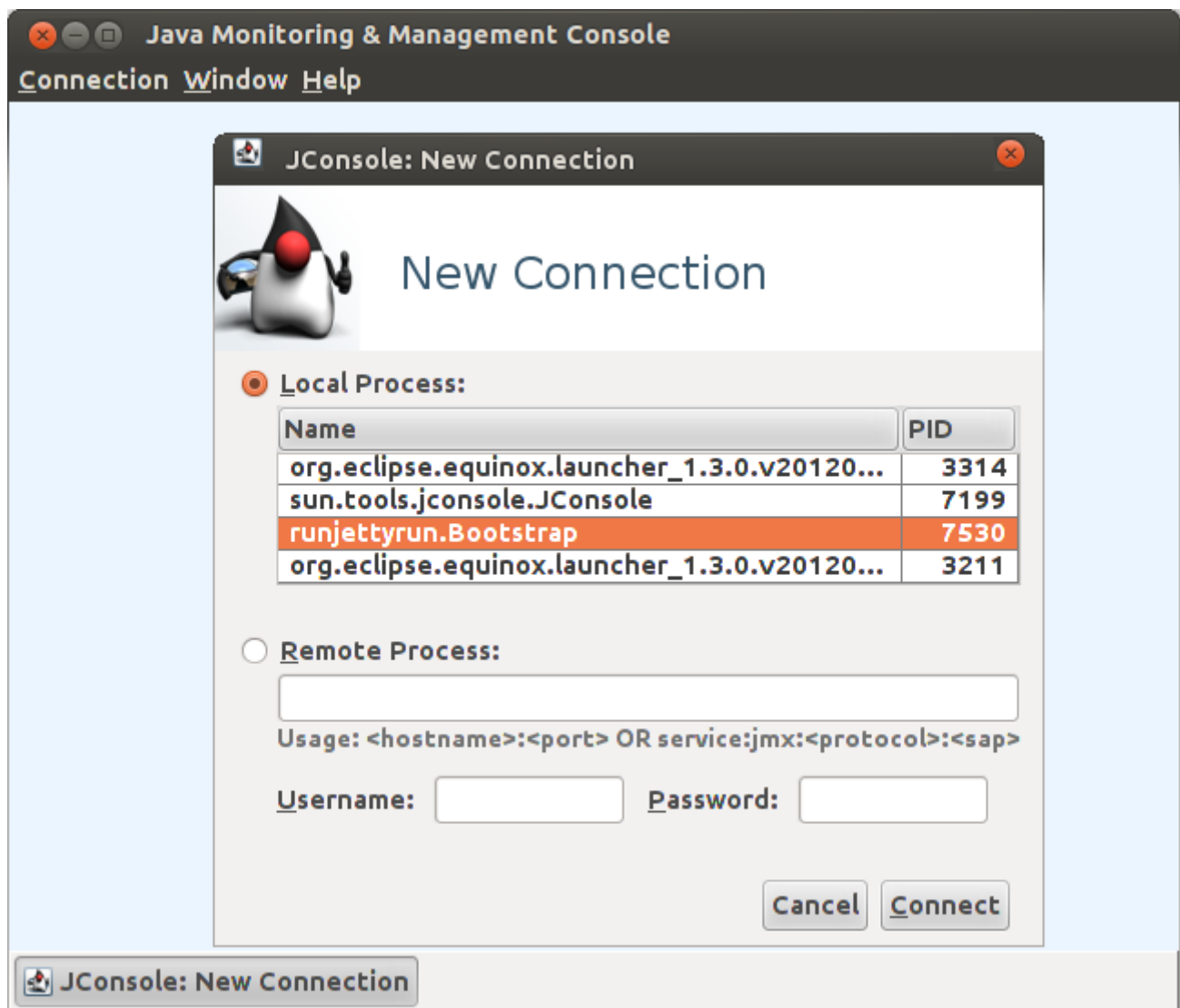
```
public class MainInitializer implements IInitializer{

    public void init(Application application) {
        new AnotherInitializer().init(application);
        new YetAnotherInitializer().init(application);
        //...
    }
    //destroy...
}
```

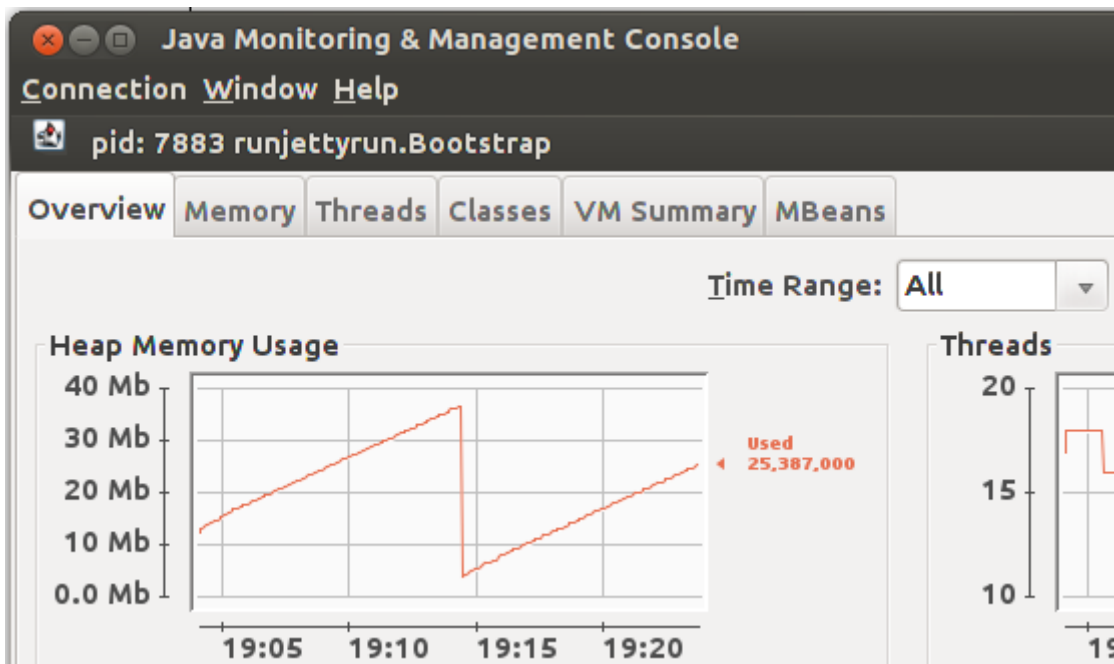
18.5. Using JMX with Wicket

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>*.

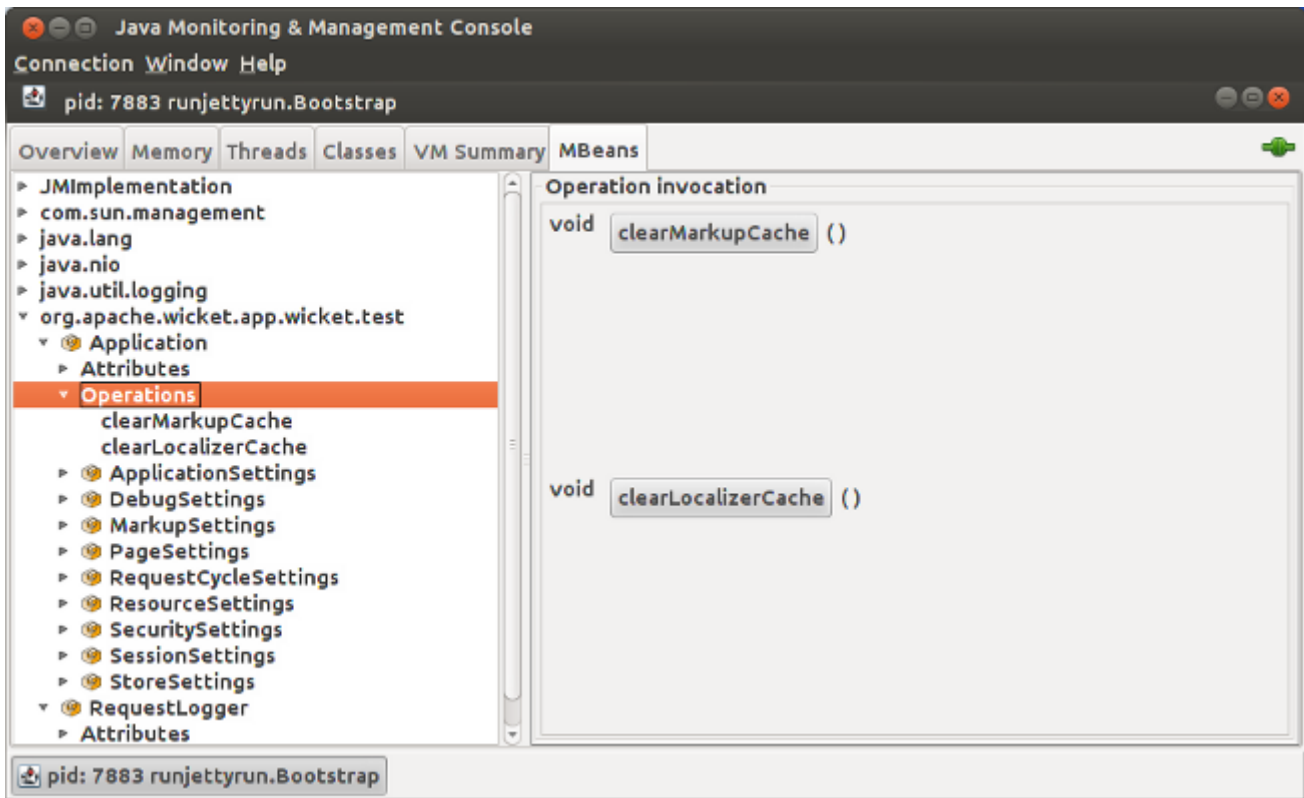
In our example we have used wicket.test as filter name for our application:

The screenshot shows the 'Java Monitoring & Management Console' window with the 'MBeans' tab selected. The left pane shows a tree view of MBeans. The 'org.apache.wicket.app.wicket.test' MBean is selected and circled in red. The right pane shows the 'Attribute values' table for this MBean.

Name	Value
ApplicationClass	com.mycompany.WicketApplication
ConfigurationType	DEPLOYMENT
HomePageClass	com.mycompany.subpackage.ValidatorPage
MarkupCacheSize	0
WicketVersion	n/a

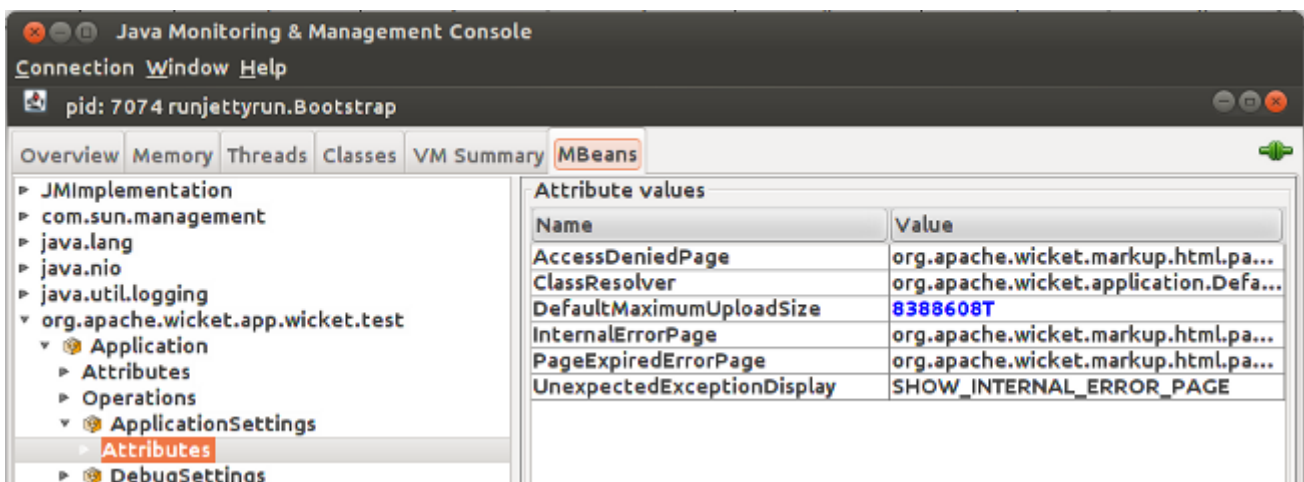
Below the table is a 'Refresh' button.

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*:



18.6. 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 = "<wicket:panel><div>Panel markup</div></wicket:panel>";
        StringResourceStream resourceStream = new StringResourceStream(markup);

        return resourceStream;
    }
}
```

Class *StringResourceStream* is a resource stream that uses a *String* instance as backing object.

18.6.1. 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 = "<wicket:panel><div>Panel with current nanotime: " +
        System.nanoTime() +
        "</div></wicket:panel>";
    StringResourceStream resourceStream = new StringResourceStream(markup);

    return resourceStream;
}

/**
 * Avoid markup caching for this component
 */
@Override
public String getCacheKey(MarkupContainer arg0, Class<?> arg1) {
    return null;
}
}

```

18.7. 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.

Chapter 19. 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](#) compliant.

19.1. 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<Void>("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<Void>("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 6.3](#):

```
final Label label = new Label("labelComponent", "Initial value.");
//autogenerate a markup id
label.setOutputMarkupId(true);
add(label);
//...
new AjaxLink<Void>("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<Void>("ajaxLink"){
    @Override
    public void onClick(AjaxRequestTarget target) {
        target.appendJavaScript(";alert('Hello!!');");
    }
};
```



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*.

The standard implementation of *AjaxRequestTarget* used by Wicket is class *org.apache.wicket.ajax.AjaxRequestHandler*. To create new instances of *AjaxRequestTarget* a Wicket application uses the provider object registered with method *setAjaxRequestTargetProvider*:

```
setAjaxRequestTargetProvider(
    Function<Page, AjaxRequestTarget> ajaxRequestTargetProvider)
```

The provider is an implementation of interface *java.util.function.Function*, hence to use custom implementations of *AjaxRequestTarget* we must register a custom provider that returns the desired implementation:

```
private static class MyCustomAjaxRequestTargetProvider implements
    Function<Page, AjaxRequestTarget>
{
    @Override
    public AjaxRequestTarget apply(Page page)
    {
        return new MyCustomAjaxRequestTarget();
    }
}
```



During request handling *AjaxRequestHandler* sends an event to its application to notify the entire component hierarchy of the current page:

```
//'page' is the associated Page instance
page.send(app, Broadcast.BREADTH, this);
```

The payload of the event is the *AjaxRequestHandler* itself.

19.2. Build-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 prepending "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.

19.2.1. 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*.

19.2.2. 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.

19.2.3. 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 19.3.3](#)).

19.2.4. AJAX editable labels

An editable label is a special label that can be edited by the 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 overridden:

- **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 on [examples site](#):

[\[go back\]](#)

Click on the area with a green outline to begin an inplace edit. Press enter or outside the area to save, pless esc to cancel.

The quick brown fox jumped over the lazy dog.

click to edit multiple lines:

multiple

lines of

textual content

click to select another site: The Server Side

This page has been refreshed: 0 times. [refresh](#)

19.2.5. 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:

builtin

[\[go back\]](#)

The textfield below will autocomplete country names. It utilizes `AutoCompleteTextField` in `wicket-extensions`.

Selected value is: g

Country:

Guatemala

Greece

Germany

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)*:

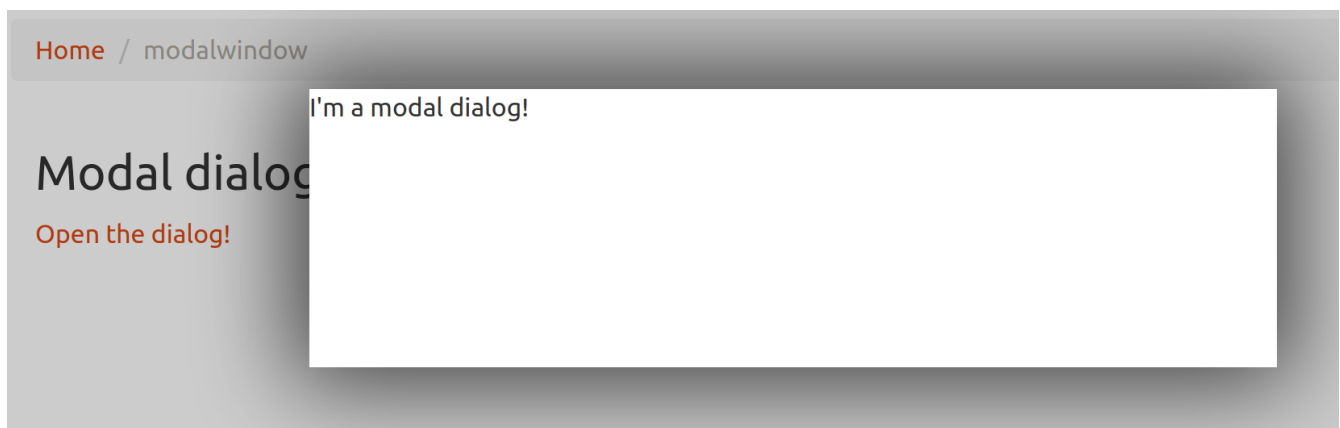
```
AutoCompleteSettings settings = new AutoCompleteSettings();
//set throttle to 400 ms: component will wait 400ms before displaying the options
settings.setThrottleDelay(400);
//...
AutoCompleteTextField field = new AutoCompleteTextField<T>("field", model) {

    @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 on [examples site](#).

19.2.6. Modal dialog

Class *org.apache.wicket.extensions.ajax.markup.html.modal.ModalDialog* is an implementation of a [modal dialog](#) based on AJAX:



The content of a modal dialog is another component. The id of this component used as content must be *ModalDialog#CONTENT_ID*.

To display a modal dialog we must call its method *open(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 *BasicModalDialogExample* and illustrate a basic usage of a modal dialog:

HTML:

```
<body>
  <h2>Modal Dialog example</h2>
  <a wicket:id="open">Open the Dialog!</a>
  <div wicket:id="modal"></div>
</body>
```

Java Code:

```
public HomePage(final PageParameters parameters) {
    super(parameters);
    final ModalDialog modal = new ModalDialog("modal");
    modal.add(new DefaultTheme());
    modal.closeOnClick();
    Label label = new Label(ModalDialog.CONTENT_ID, "I'm a modal dialog!");

    modal.setContent(label);

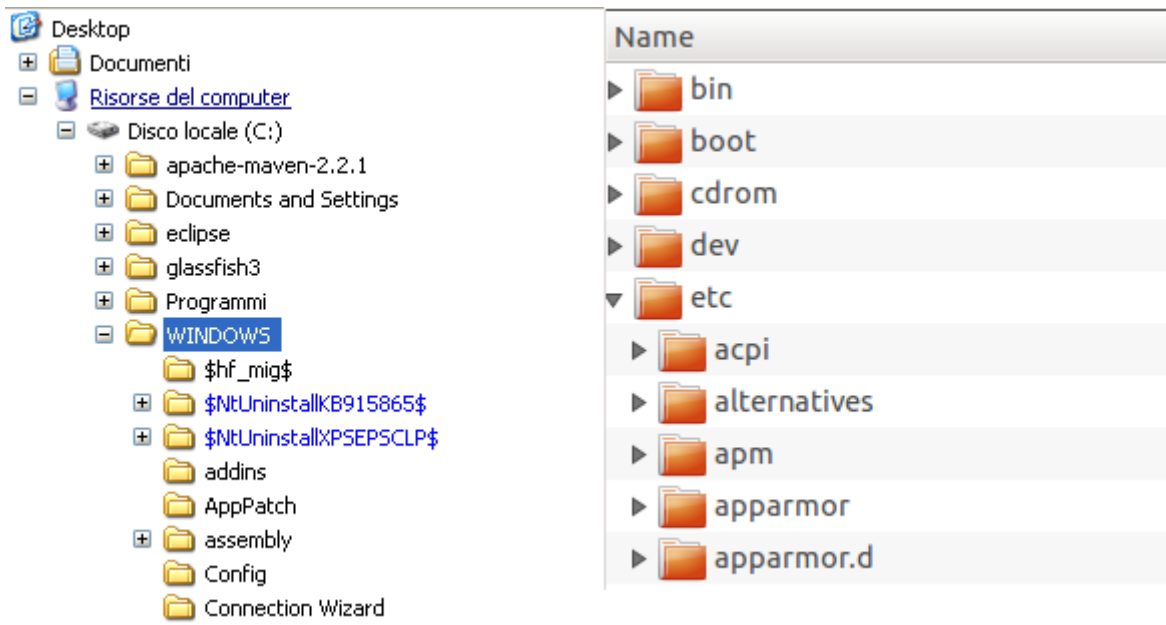
    add(modal);
    add(new AjaxLink<Void>("open") {
        @Override
        public void onClick(AjaxRequestTarget target) {
            modal.open(target);
        }
    });
}
```

Just like any other component also *ModalDialog* must be added to a markup tag, like we did in our example using a `<div>` tag. Wicket will automatically hide the content of this tag in the final markup, as long as the dialog is not opened. This component does not provide any styling by itself, so you have can add a *DefaultTheme* to this component if aren't styling these CSS classes by yourself.

The modal dialog can be closed from code using its method *close(AjaxRequestTarget target)*.

19.2.7. 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:

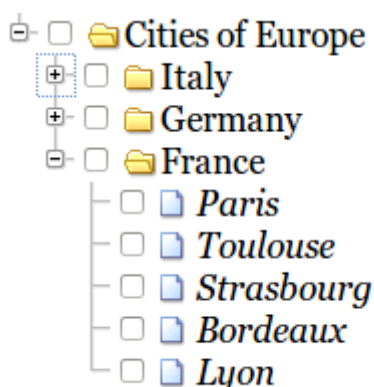


Because of their highly interactive nature, tree repeaters are implemented as AJAX components, meaning that they are updated via AJAX when we expand or collapse their nodes.

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 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:



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) {
        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 *DefaultMutableTreeNodes* 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 tree 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));
}
```


19.2.8. 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 `` 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<Void>("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.

19.3. 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`.

19.3.1. 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 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*.

19.3.2. 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<Void> 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);
```



AjaxFormSubmitBehavior does not prevent JavaScript default event handling. For `<input type=submit>` you'll have to call *AjaxRequestAttributes.setPreventDefault(true)* to prevent the form from being submitted twice.

19.3.3. 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<Void> 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));
```

19.3.4. 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);

```



By default AJAX components and behaviors are *stateful*, but as we will see very soon they can be easily turned to stateless if we need to use them in stateless pages.

19.4. Using an activity indicator

One of the things 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.

19.5. 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](#) 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

dt	Specifies the type of data returned by the AJAX call (XML, HTML, JSON, etc...).	XML
wr	Boolean flag which indicates whether the response is <ajax-response> which is handled by wicket-ajax.js or custom response type which can be handled by application's code (e.g. in IAjaxCallListener's success handler).	true
ih, bh, pre, bsh, ah, sh, fh, coh, dh	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



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 *IAjaxCallListeners* returned by method *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:

- **getInitHandler(Component)**: returns the JavaScript code that will be executed on initialization of the Ajax call, immediately after the causing event. 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.

- **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*.
- **getDoneHandler(Component)**: returns the JavaScript code that will be executed after the Ajax call is done, regardless whether it was sent or not. 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*.

In the next paragraph we will see an example of custom *IAjaxCallListener* designed to disable a component during AJAX request processing.

19.6. 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*.

19.6.1. 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:

Press 'submit' to disable the form for (at least) three seconds.



19.6.2. 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.

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.

19.6.3. 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;">'
      + ''
      + '</div>');

    hiderId = "#" + hiderId;
    //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();
}
};

```

19.6.4. Java 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));
    }
}
//...
```

19.6.5. 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>', <callback Function>)*. The first parameter of this function is the name of the event we want to handle. The possible names are:

- *'/ajax/call/init'*: called on initialization of an ajax call
- *'/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.
- *'/ajax/call/done'*: called when the AJAX call is done.
- *'/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 19](#). 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`.

19.7. Stateless AJAX components/behaviors

Wicket makes working with AJAX easy and pleasant with its component-oriented abstraction. However as side effect, AJAX components and behaviors make their hosting page stateful. This can be quite annoying if we are working on a page that must be stateless (for example a login page). Starting from version 7.4.0 Wicket has made quite easy forcing existing AJAX components to be stateless. All we have to do is to override component's method `getStatelessHint` returning `true`:

```
final Link<?> incrementLink = new AjaxFallbackLink<Void>("incrementLink")
{
    ...

    @Override
    protected boolean getStatelessHint()
```

```

    {
        return true;
    }
};

```

Just like components also AJAX behaviors can be turned to stateless overriding *getStatelessHint(Component component)*

```

final AjaxFormSubmitBehavior myBehavior = new AjaxFormSubmitBehavior(form, event)
{
    ...

    @Override
    protected boolean getStatelessHint(Component component)
    {
        return true;
    }
};

```

19.7.1. Usage

Stateless components and behaviors follows the same rules and conventions of their standard stateful version, so they must have a markup id in order to be manipulated via JavaScript. However in this case calling *setOutputMarkupId* on a component is not enough. Since we are working with a stateless page, the id of the component to refresh must be unique but also static, meaning that it should not depend on page instance. In other words, the id should be constant through different instances of the same page. By default calling *setOutputMarkupId* we generate markup ids using a session-level counter and this make them not static. Hence, to refresh component in a stateless page we must provide them with static ids, either setting them in Java code (with *Component.setMarkupId*) or simply writing them directly in the markup:

```
<span id="staticIdToUse" wicket:id="componentWicketId"></span>
```

See [examples](#) page for a full showcase of AJAX-stateless capabilities.

19.8. Lambda support for components

Just like we have seen for regular links, WicketStuff project offers a lambda-based factory class to build Ajax links and Ajax submitting component:

```

//create an AJAX link component
add(ComponentFactory.ajaxLink("id", (ajaxLink, ajaxTarget) -> {/*do stuff*/}));

//create a submit link with error handler
add(ComponentFactory.ajaxSubmitLink("id", (ajaxLink, ajaxTarget) -> {/*do submit
stuff*/},

```

```
(ajaxLink, ajaxTarget) -> {/*do error stuff*/});
```

For more examples see WicketStuff module [wicketstuff-lambda-components](#).

19.9. Lambda support for behaviors

Ajax behaviors classes come with lambda-based factory methods which make their creation easier and less verbose. For example AjaxEventBehavior can be instantiated like this:

```
AjaxEventBehavior.onEvent("click", ajaxtarget -> //some lambda stuff)
```

In the following table are listed these factory methods along with their behavior classes:

Table 1. Factory methods

Class Name	Method Name
AbstractAjaxTimerBehavior	onTimer
AjaxEventBehavior	onEvent
AjaxNewWindowNotifyingBehavior	onNewWindow
AjaxSelfUpdatingTimerBehavior	onSelfUpdate
AjaxFormChoiceComponentUpdatingBehavior	onUpdateChoice
AjaxFormComponentUpdatingBehavior	onUpdate
AjaxFormSubmitBehavior	onSubmit
OnChangeAjaxBehavior	onChange

19.10. 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.

Chapter 20. 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](#)) and new specifications (like [EJB 3.1](#)) 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](#) 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](#) already provide a set of built-in listeners to integrate our applications with EJB 3.1 compliant containers (like JBoss Seam) or with some of the most popular enterprise frameworks like [Guice](#) 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](#)) and then using Spring.

20.1. 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 *IComponentInstantiationListeners*.

The following initialization code is taken from project *EjbInjectionExample*:

```
public class WicketApplication extends WebApplication
{
    //Constructor...

    @Override
```

```

public void init()
{
    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:



Welcome to the EJB world!

20.2. 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;
```

20.3. JSR-330 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:

```
//inject a bean specifying its name with JSR-330 annotations
@Inject
@Named("anotherName")
private EnterpriseMessage enterpriseMessage;
```

20.4. Summary

During their lifecycle Wicket components are serialized into the session or some secondary storage. But in most cases injected dependencies are not serializable, as these are typically singletons like services and repositories. Because of this Wicket uses [Byte Buddy](#) to generate proxies that are injected instead. These will serialize a replacement and re-lookup the dependency after deserialization.



By default injected types need a default constructor without arguments, otherwise proxies can not be instantiated. You can remedy this limitation by adding [Objenesis](#) to your project dependencies.

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`).

Chapter 21. Native WebSockets

WebSockets is a technology that provides full-duplex communications channels over a single TCP connection. This means that once the browser establish a web socket connection to the server the server can push data back to the browser without the browser explicitly asking again and again whether there is something new for it.

Wicket Native WebSockets modules provide functionality to integrate with the non-standard APIs provided by different web containers (like [Apache Tomcat](#))



Native WebSocket works only when both the browser and the web containers support WebSocket technology. There are no plans to add support to fallback to long-polling, streaming or any other technology that simulates two way communication. Use it only if you really know that you will run your application in an environment that supports WebSockets. Currently supported web containers are Jetty 7.5+ , Tomcat 7.0.27+ and JBoss WildFly 8.0.0+. Supported browsers can be found at [caniuse.com](#)

21.1. How does it work ?

Each of the modules provide a specialization of *org.apache.wicket.protocol.http.WicketFilter* that registers implementation specific endpoint when an HTTP request is [upgraded](#).

WebSockets communication can be used in a Wicket page by using *org.apache.wicket.protocol.ws.api.WebSocketBehavior* or in a *IResource* by extending *org.apache.wicket.protocol.ws.api.WebSocketResource*. When a client is connected it is being registered in a application scoped registry using as a key the application name, the client http session id, and the id of the page or the resource name that registered it. Later when the server needs to push a message it can use this registry to filter out which clients need to receive the message.

When a message is received from the client Wicket wraps it in *IWebSocketMessage* and calls *WebSocketBehavior.onMessage()* or *WebSocketResource.onMessage()* where the application logic can react on it. The server can push plain text and binary data to the client, but it can also add components for re-render, prepend/append JavaScript as it can do with [Ajax](#).

21.2. How to use

- **Classpath dependency**

Add the following dependency to your application to get access to the *wicket-native-websocket* API on any JSR356 compliant application server (at the moment are supported: Tomcat 8.0+, Tomcat 7.0.47+, Jetty 9.1.0+ and JBoss Wildfly 8.0.0+):

```
<dependency>
  <groupId>org.apache.wicket</groupId>
  <artifactId>wicket-native-websocket-javax</artifactId>
```

```
<version>...</version>
</dependency>
```

- for [Spring Boot](#) applications also add

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-websocket</artifactId>
</dependency>
```



The examples above show snippets for Maven's pom.xml but the application can use any other dependency management tool like [Gradle](#)

- **web.xml**

In *WEB-INF/web.xml* replace the usage of **WicketFilter** with the following:

```
<filter-class>org.apache.wicket.protocol.ws.javax.JavaxWebSocketFilter</filter-class>
```

For [Spring Boot](#) application:

```
@Bean
    public FilterRegistrationBean wicketFilter() {
        final FilterRegistrationBean wicketFilter = new
FilterRegistrationBean();
        wicketFilter.setDispatcherTypes(DispatcherType.REQUEST,
DispatcherType.ERROR, DispatcherType.FORWARD, DispatcherType.ASYNC);
        wicketFilter.setAsyncSupported(true);
        wicketFilter.setFilter(new JavaxWebSocketFilter());
        wicketFilter.addInitParameter(WicketFilter.APP_FACT_PARAM,
SpringWebApplicationFactory.class.getName());
        wicketFilter.addInitParameter(WicketFilter.FILTER_MAPPING_PARAM,
"/");
        wicketFilter.addUrlPatterns("/");
        return wicketFilter;
    }

@Bean
    public ServerEndpointExporter serverEndpointExporter() {
        return new ServerEndpointExporter();
    }

@Bean
    public WicketServerEndpointConfig wicketServerEndpointConfig() {
        return new WicketServerEndpointConfig();
    }
```

- **WebSocketBehavior**

org.apache.wicket.protocol.ws.api.WebSocketBehavior is similar to Wicket Ajax behaviors that you may have used. Add *WebSocketBehavior* to the page (or to any component in the page) that will use web socket communication:

```
public class MyPage extends WebPage {

    public MyPage()
    {
        add(new WebSocketBehavior() {
            @Override
            protected void onMessage(WebSocketRequestHandler handler, TextMessage message)
            {
                String msg = message.getText();
                // do something with msg
            }
        });
    }
}
```

Use *message.getText()* to read the message sent by the client and use *handler.push(String)* to push a text message to the connected client. Additionally you can use *handler.add(Component...)* to add Wicket components for re-render, *handler.prependJavaScript(CharSequence)* and *handler.appendJavaScript(CharSequence)* as you do with *AjaxRequestTarget*.

- **WebSocketResource**

Wicket allows one thread at a time to use a page instance to simplify the usage of the pages in multithreaded environment. When a WebSocket message is sent to a page Wicket needs to acquire the lock to that page to be able to pass the *IWebSocketMessage* to the *WebSocketBehavior*. This may be problematic when the application needs to send many messages from the client to the server. For this reason Wicket provides *WebSocketResource* - an *IResource* implementation that provides the same APIs as *WebSocketBehavior*. The benefit is that there is no need of synchronization as with the pages and the drawback is that *WebSocketRequestHandler.add(Component...)* method cannot be used because there is no access to the components in an *IResource*.

To register such WebSocket resource add such line to *YourApplication1.init()* method:

```
getSharedResources().add("someName", new MyWebSocketResource());
```

and

```
page.add(new BaseWebSocketBehavior("someName"));
```

to any page. This will prepare the JavaScript connection for you.

- **WebSocket connection registry**

To push data to one or more clients the application can use the *IWebSocketConnectionRegistry* to find all registered connections and send data to all/any of them:

```
Application application = Application.get(applicationName);
WebSocketSettings webSocketSettings = WebSocketSettings.Holder.get(application);
IWebSocketConnectionRegistry webSocketConnectionRegistry =
webSocketSettings.getConnectionRegistry();
IWebSocketConnection connection =
webSocketConnectionRegistry.getConnection(application, sessionId, key);
```

21.3. Client-side APIs

By adding a *(Base)WebSocketBehavior* to your component(s) Wicket will contribute *wicket-websocket-jquery.js* library which provides some helper functions to write your client side code. There is a default websocket connection per Wicket Page opened for you which you can use like:

```
Wicket.WebSocket.send('{msg: "my message"}').
```

To close the default connection:

```
Wicket.WebSocket.close()
```

Wicket.WebSocket is a simple wrapper around the native window.WebSocket API which is used to intercept the calls and to fire special JavaScript events (Wicket.Event PubSub). Once a page that contributes *(Base)WebSocketBehavior* is rendered the client may react on messages pushed by the server by subscribing to the */websocket/message* event:

```
Wicket.Event.subscribe("/websocket/message", function(jqEvent, message) {
    var data = JSON.parse(message);
    processData(data); // does something with the pushed message
});
```

Here is a table of all events that the application can subscribe to:

Event name	Arguments	Description
/websocket/open	jqEvent	A WebSocket connection has been just opened
/websocket/message	jqEvent, message	A message has been received from the server
/websocket/closed	jqEvent	A WebSocket connection has been closed

/websocket/error	jqEvent	An error occurred in the communication. The connection will be closed
------------------	---------	---

21.4. Testing

The module provides *org.apache.wicket.protocol.ws.util.testers.WebSocketTester* which gives you the possibility to emulate sending and receiving messages without the need to run in a real web container, as WicketTester does this for HTTP requests. Check [WebSocketTesterBehaviorTest](#)

21.5. FAQ

1. Request and session scoped beans do not work. The Web Socket communication is not processed by Servlet Filters and Listeners and thus the Dependency Injection libraries have no chance to export the request and session bean proxies.

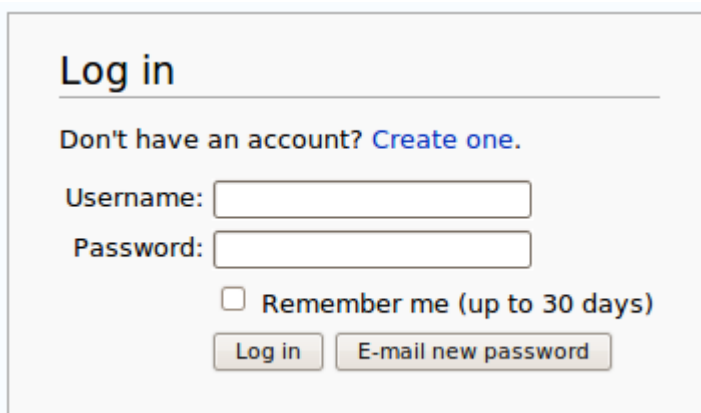
Chapter 22. Security with Wicket

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.

22.1. 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:



The image shows a login form with the following elements:

- Title: **Log in**
- Link: [Don't have an account? Create one.](#)
- Username field:
- Password field:
- Remember me checkbox: ☐ **Remember me (up to 30 days)**
- Buttons:

Wicket supports form-based authentication with session class *AuthenticatedWebSession* and application class *AuthenticatedWebApplication*, both placed inside package *org.apache.wicket.authroles.authentication*.

22.1.1. 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*.
- **invalidate():** sets the flag *signedIn* to false and invalidates session.
- **signOut():** an alias of *invalidate()*.

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.

22.1.2. AuthenticatedWebApplication

Class *AuthenticatedWebApplication* provides the following methods to support form-based authentication:

- **getSessionClass():** 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 *Component*'s method *continueToOriginalDestination()*.

The other methods implemented inside *AuthenticatedWebApplication* will be introduced when we talk about authorization.

22.1.3. 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() {
        super.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 BookmarkablePageLink<Void>("goToHomePage",
        getApplication().getHomePage()));

        add(new Link<Void>("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<Void> form = new StatelessForm<Void>("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.setModel(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 new Roles();
    }
}
```

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*.

22.1.4. 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 and the parameters 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);
```



Since both *restartResponseAtSignInPage* and *redirectToInterceptPage* internally throw an exception, the code placed after them will not be executed.

22.2. 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 (class *SecuritySettings*) 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 pattern for authorization strategies.

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.

22.2.1. 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.

22.2.2. Role-based strategies

At the end of [paragraph 22.1](#) 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...

        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 *AnnotationsRoleAuthorizationStrategy* uses Java annotations.



Application class *AuthenticatedWebApplication* already sets *MetaDataRoleAuthorizationStrategy* and *AnnotationsRoleAuthorizationStrategy* as its own authorization strategies (it uses a compound strategy as we will see in [paragraph 22.2](#)).

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.

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;
    }
}

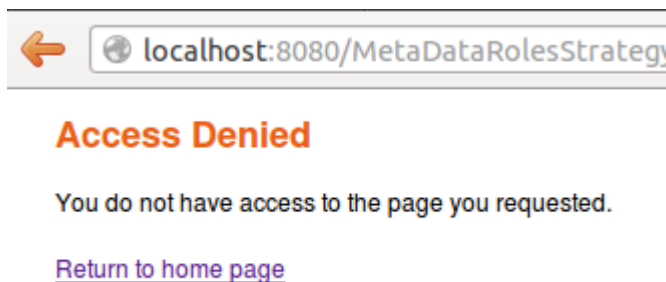
```


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 class *ApplicationSettings*:

```
//Application class code...
@Override
public void init(){
    getApplicationSettings().setAccessDeniedPage(
        MyCustomAccessDeniedPage.class);
}
```

Just like custom “Page expired” page (see [chapter 8.2.5](#)), also custom “Access denied” page must be bookmarkable.

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 *MetaDataRolesStrategyExample* 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...
}
```

22.2.3. 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 class *SecuritySettings*. 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.

22.2.4. Strategy *RoleAuthorizationStrategy*

Class *RoleAuthorizationStrategy* is a compound strategy that combines both *MetaDataRoleAuthorizationStrategy* and *AnnotationsRoleAuthorizationStrategy*.

This is the strategy used internally by *AuthenticatedWebApplication*.

22.3. 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 annotated with *RequireHttps*. Most of the times the wrapped mapper will be the root one, just like we saw before for *CryptoMapper* in [paragraph 10.6](#).

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, 8443)));  
}
```

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...
}
```

22.4. URLs encryption in detail

In chapter [10.6](#) we have seen how to encrypt URLs using *CryptoMapper* request mapper. To encrypt/decrypt page URLs *CryptoMapper* uses an instance of *org.apache.wicket.util.crypt.ICrypt* interface:

```
public interface ICrypt
{
    String encryptUrlSafe(final String plainText);

    String decryptUrlSafe(final String encryptedText);

    ...
}
```

The default implementation for this interface is class *org.apache.wicket.util.crypt.SunJceCrypt*. It provides password-based cryptography using *PBEWithMD5AndDES* algorithm coming with the standard security providers in the Java Runtime Environment.



For better security it is recommended to install Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction [Policy Files](#)

By using *CryptoMapper(IRequestMapper wrappedMapper, Application application)* constructor the mapper will use the configured *org.apache.wicket.util.crypt.ICryptFactory* from *org.apache.wicket.settings.SecuritySettings.getCryptFactory()*. To use a stronger cryptography mechanism there are the following options:

- The first option is to use constructor *CryptoMapper(IRequestMapper wrappedMapper, Supplier<ICrypt> cryptProvider)* and give it an implementation of *java.util.function.Supplier* that returns a custom *org.apache.wicket.util.crypt.ICrypt*.
- The second option is to register a cipher factory at application level with method *setCryptFactory(ICryptFactory cryptFactory)* of class *SecuritySettings*:

```
@Override
public void init() {
    super.init();
    getSecuritySettings().setCryptFactory(new SomeCryptFactory());
    setRootRequestMapper(new CryptoMapper(getRootRequestMapper(), this));
}
```

Since version 6.19.0 Wicket uses *org.apache.wicket.core.util.crypt.KeyInSessionSunJceCryptFactory* as a default factory for *ICrypt* objects. This factory generates a unique key for each user that is stored in her HTTP session. This way it helps to protect the application against [CSRF](#) for each user of the application. The url itself serves as [encrypted token](#)



org.apache.wicket.core.util.crypt.KeyInSessionSunJceCryptFactory binds the http session if it is not already bound! If the application needs to run in stateless mode then the application will have to provide a custom implementation of *ICryptFactory* that stores the user specific keys by other means.

22.5. CSRF protection

CryptoMapper helps preventing CSRF attacks by making the urls impossible to be guessed by an attacker but still there is some theoretical chance this to happen.

To further help against this kind of vulnerability Wicket provides *ResourceIsolationRequestCycleListener* - a *IRequestCycleListener* that uses *IResourceIsolationPolicy* objects to decide whether to allow or reject cross-origin requests. Just like any RequestCycle listener *ResourceIsolationRequestCycleListener* must be registered on application initialization:

```
@Override
```

```
protected void init() {
    super.init();
    getRequestCycleListeners().add(new ResourceIsolationRequestCycleListener());
    // ...
}
```

By default *ResourceIsolationRequestCycleListener* checks only event handlers requests, i.e. a cross-origin requests cannot execute *Link.onClick()* or submit forms (*Form.onSubmit()*). Any request to render pages are still allowed so Wicket pages could be easily embedded in other applications. To extend CSRF protection to pages we can simply override *isChecked(IRequestHandler handler)* method to make it return always *true*:

```
@Override
protected void init() {
    super.init();
    getRequestCycleListeners().add(new ResourceIsolationRequestCycleListener() {
        @Override
        protected boolean isChecked(IRequestHandler handler) {
            //check everything
            return true;
        }
    });
    // ...
}
```

ResourceIsolationRequestCycleListener is highly configurable. It allows to add exempted paths that will not be checked with the *addExemptedPath* method. It can also be configured with multiple *ResourceIsolationPolicy* objects to be checked in order.

An *IResourceIsolationPolicy* returns a *ResourceIsolationOutcome* after processing a request, which can be one of 3 values (*ALLOWED*, *DISALLOWED*, *UNKNOWN*). The *ResourceIsolationRequestCycleListener* checks the *IResourceIsolationPolicy* objects in order and uses the first outcome that is not *UNKNOWN* to trigger the appropriate action. If all return *UNKNOWN* *unknownOutcomeAction* is applied. The actions can be configured through the listener.

The default constructor uses the *FetchMetadataResourceIsolationPolicy*, which checks Fetch Metadata headers, and the *OriginResourceIsolationPolicy* which uses the Origin and Referer headers to forbid requests made from a different origin, in order. The *OriginResourceIsolationPolicy* contains the refactored logic of the now deprecated *CsrfPreventionRequestCycleListener*. The listener can be configured to include custom *IResourceIsolationPolicy* objects.

For example:

```
@Override
protected void init() {
    super.init();
    getRequestCycleListeners().add(
```

```

new ResourceIsolationRequestCycleListener(
    new FetchMetadataResourceIsolationPolicy(),
    new OriginResourceIsolationPolicy(),
    new MyCustomResourceIsolationPolicy()
));
// ...
}

```

ResourceIsolationRequestCycleListener is not an alternative to *CryptoMapper*! Both of them could be used separately or in tandem to prevent CSRF attacks depending on the application requirements.



In the next chapter we will cover unit testing with Wicket. If your application is protected with *ResourceIsolationRequestCycleListener* you have to properly set request header "sec-fetch-site" to make you unit tests pass. In [paragraph 23.1.10](#) you will learn how to do it.

Wicket also provides the deprecated (since version 9.1.0) *CsrfPreventionRequestCycleListener* - a *IRequestCycleListener* that forbids requests made from a different origin. Similar to the *ResourceIsolationRequestCycleListener* by default only actions are forbidden, i.e. a request coming from different origin cannot execute *Link.onClick()* or submit forms (*Form.onSubmit()*). Any request to render pages are still allowed so Wicket pages could be easily embedded in other applications.

MyApplication.java

```

@Override
protected void init() {
    super.init();
    getRequestCycleListeners().add(new CsrfPreventionRequestCycleListener());
    // ...
}

```

CsrfPreventionRequestCycleListener is highly configurable. It allows to define a whitelist of allowed origins via *addAcceptedOrigin(String acceptedOrigin)*, to enable/disable it dynamically by overriding *isEnabled()*, to define different kind of actions when a request is rejected or allowed, to set custom error message and code for the rejected requests.

CsrfPreventionRequestCycleListener is not an alternative to *CryptoMapper*! Both of them could be used separately or in tandem to prevent CSRF attacks depending on the application requirements.

22.6. Content Security Policy (CSP)

In Wicket 9 support for a Content Security Policy (or CSP) has been added. CSP is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross Site Scripting (XSS) and data injection attacks. These attacks are used for everything from data theft to site defacement to distribution of malware. See [MDN](#) for more information on CSP.

The default CSP set by Wicket is very strict. It requires all scripts and stylesheets to be rendered

with a nonce. Wicket will automatically attach the nonce to all header contributions that support this (i.e. subclasses of the *AbstractCspHeaderItem* class). Images, fonts and (i)frames are allowed from *self*, the current host. All other resources are blocked. This includes any inline styling or Javascript (such as an *onclick* attribute). The exact CSP is:

```
Content-Security-Policy: default-src 'none'; script-src 'strict-dynamic' 'nonce-XYZ';
style-src 'nonce-XYZ'; img-src 'self'; connect-src 'self'; font-src 'self'; manifest-
src 'self'; child-src 'self'; frame-src 'self'; base-uri 'self';
```

In developer mode, the CSP is extended with a reporting directive that reports violations at a special endpoint in the application that logs the violation. This is convenient while developing an application, but care should be taken when this is enabled on production. The *cspviolation* endpoint must be an open endpoint and all data sent to that URL will end up in the logs. To prevent the server log from filling the disk, make sure the *org.apache.wicket.csp.ReportCSPViolationMapper* logger has a limit on its disk usage.

22.6.1. Configuring the Content Security Policy

The Content Security Policy is managed by the *ContentSecurityPolicySettings* that can be accessed via *WebApplication.getCspSettings()*. This class maintains two instances of *CSPHeaderConfiguration*, each of which contains the directives for the CSP HTTP headers *Content-Security-Policy* and *Content-Security-Policy-Report-Only*. The first header defines the policies that are actually enforced by the browser, whereas the second header defines a policy for which the browser will only report violations. Note that violations can also be reported on the enforced policy and that reporting requires a *report-uri* directive.

For applications that cannot adhere to a CSP, the CSP can be disabled with the following call in your *Application* class:

```
@Override
protected void init() {
    super.init();
    getCspSettings().blocking().disabled();
    // ...
}
```

As mentioned before, Wicket uses a very strict CSP by default. This preset can be selected with the following code:

```
getCspSettings().blocking().strict();
```

A third preset is available that allows unsafe inline Javascript and styling and the use of unsafe *eval*. As can be inferred from the names, use of *unsafe* is not recommended. It removes the most important protection offered by CSP. However, older applications may not be ready to apply a strict CSP. For those applications, *CSPHeaderConfiguration.unsafeInline()* can be a starting point for the path to a strict CSP.

CSPHeaderConfiguration defines several methods to tune the Content Security Policy for your application. Additional sources can be whitelisted for certain via the *add(CSPDirective, ...)* methods. For example, the code below whitelists images rendered via a *data:* url, fonts loaded from <https://maxcdn.bootstrapcdn.com> and a single CSS file.

```
getCspSettings().blocking()
    .add(CSPDirective.IMG_SRC, "data:")
    .add(CSPDirective.FONT_SRC, "https://maxcdn.bootstrapcdn.com")
    .add(CSPDirective.STYLE_SRC,
        "https://maxcdn.bootstrapcdn.com/font-awesome/4.3.0/css/font-
awesome.min.css");
```

22.6.2. Guidelines for strict CSP support

To comply to the strict CSP set by default, you'll need to follow a few rules. Most importantly, you cannot use inline styling or Javascript. This includes:

- *style* attributes in the markup or via an *AttributeModifier*.
- Inline event handlers, such as *onclick* or *onsubmit*, in the markup or via an *AttributeModifier*.
- Including stylesheets directly from the markup without whitelisting them.
- Including Javascript directly from the markup. Whitelisting is not possible due to the *strict-dynamic* rule.
- Rendering *style* attributes from Javascript in dynamically generated markup. Modifying the *style* DOM property is still possible.
- Load other resources from external domains without whitelisting them.

For most of these restrictions Wicket provides alternative solutions that do work with a strict CSP. First of all, replace all inline styling with proper stylesheets and use CSS selectors to target the elements in the DOM. Replace the *AttributeModifier* with a *style* attribute for one with the *class* attribute. For Javascripts that manipulate the *style* attribute, you may have to update the script to use the DOM property instead.

When a component includes a stylesheet directly from the markup as seen below, there are two possible solutions:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns:wicket="http://wicket.apache.org">
<head>
    <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/font-awesome/4.3.0/css/font-awesome.min.css"
    />
</head>
<body>...</body>
</html>
```

The first solution is to whitelist the stylesheet, as seen above. The second solution is to move the stylesheet to a header contribution in *renderHead*. This will allow Wicket to render the required *nonce* attribute. For Javascript resources, this is the only possible solution:

```
@Override
public void renderHead(IHeaderResponse response) {
    super.renderHead(response);
    response.render(CssHeaderItem.forReference(
        new CssResourceReference(MyComponent.class, "mycomponent.css")));
}
```

When your component relies on inline event handlers, such as *onclick* or *onsubmit*, you can convert these to an *OnEventHeaderItem*. Again, this will allow Wicket to add the required *nonce* attribute.

```
@Override
public void renderHead(IHeaderResponse response) {
    super.renderHead(response);
    response.render(OnEventHeaderItem.forComponent(this, "submit",
        "return confirm('Do you really want to submit?')"));
}
```

22.7. Cross Origin Isolation with COOP and COEP

A major consequence of transient execution attacks for the web is the risk of exposing private authenticated data to untrusted sites via microarchitectural side channels. Some web APIs increase the risk of side-channel attacks like [Spectre](#). To mitigate that risk, browsers offer two opt-in mechanisms to effectively isolate resources from third-party origins through Cross-Origin Opener Policy and Cross-Origin Embedder Policy.

[Cross-Origin Opener Policy \(COOP\)](#) is a security mitigation that lets developers isolate their resources against side-channel attacks and information leaks. The COOP response header allows a document to request a new browsing context group to better isolate itself from other untrustworthy origins.

[Cross-Origin Embedder Policy \(COEP\)](#) prevents a document from loading any cross-origin resources which don't explicitly grant the document permission to be loaded.

COOP and COEP are independent mechanisms that can be enabled, tested and deployed separately. While enabling one doesn't require developers to enable the other, when set together COOP and COEP allows developers to use powerful features (such as *SharedArrayBuffer*, *performance.measureMemory()* and the JS Self-Profiling API) securely, while still mitigating information leaks and side channel attacks like Spectre. Read more about COOP/COEP on [COOP/COEP](#) and [why you need cross-origin isolation](#).

Wicket provides highly configurable support for COOP and COEP. Users can specify the mode for each policy. For COOP, the *CoopMode* represents the three valid header values *same-origin* (*SAME_ORIGIN* mode), *same-origin-allow-popups* (*SAME_ORIGIN_ALLOW_POPUPS* mode), *unsafe-*

none (*UNSAFE_NONE* mode) and disabling COOP entirely (*DISABLED*). For COEP the *CoepMode* represents whether the policy will be *Report-Only* (*REPORTING* mode), enforcing (*ENFORCING* mode) or disabled entirely (*DISABLED*). For both COOP and COEP, exempted paths can be specified, the response headers will not be set for these paths. For exempted paths, the *IRequestCycleListeners* perform exact string matching against the path associated with URL requests. When setting exempted paths the parameter should only receive relative paths with a trailing slash. Exemptions can include paths that use features that need to hold JavaScript references to windows opened through the *window.open* function.

To set preferred policies use the *CrossOriginOpenerPolicyConfiguration* and *CrossOriginEmbedderPolicyConfiguration* in *SecuritySettings*. The configuration should be set in the *init()* method, the values are read once at startup (in *WebApplication#validateInit()*), and if the configurations indicate the policies aren't *DISABLED* the respective listeners (*CrossOriginOpenerPolicyRequestCycleListener*, *CrossOriginEmbedderPolicyRequestCycleListener*) will be added automatically and these *IRequestCycleListeners* will add the appropriate headers to every response.

MyApplication.java

```
@Override
protected void init() {
    super.init();
    // configure COOP
    getSecuritySettings().setCrossOriginOpenerPolicyConfiguration(CoopMode.SAME_ORIGIN,
"exemptions");
    // configure COEP
    getSecuritySettings().setCrossOriginEmbedderPolicyConfiguration(CoepMode.ENFORCING,
"exemptions");
    // ...
}
```

22.8. 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):

File	Extensions
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
Web font files	.eot, .ttf, .woff

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 pattern "+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");
    }
}
```

22.9. External Security Checks

Since Mozilla released their site to check if web pages have security issues named [Mozilla Observatory](#) a few things which can be done to get a high grade within this ranking without using further frameworks.

Add a request cycle listener to your web application and adjust the headers to fit your requirements:

```

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

    getRequestCycleListeners().add(new IRequestCycleListener(){

        @Override
        public void onEndRequest(RequestCycle cycle)
        {
            WebResponse response = (WebResponse) cycle.getResponse();
            response.setHeader("X-XSS-Protection", "1; mode=block");
            response.setHeader("Strict-Transport-Security", "max-age=31536000;
includeSubDomains; preload");
            response.setHeader("X-Content-Type-Options", "nosniff");
            response.setHeader("X-Frame-Options", "sameorigin");
            response.setHeader("Content-Security-Policy", "default-src https:");
        }
    });
}

```

Add this configuration to your web.xml (or let your server redirect to https):

```

<?xml version="1.0" encoding="UTF-8"?>
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Entire Application</web-resource-name>
        <url-pattern>*/</url-pattern>
    </web-resource-collection>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>

```

After this changes you have to check if your web application continues to work because it fits the requirements given with these headers. For example that resources could not be requested from other domains anymore.

22.10. 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, we continued with how to configure our applications to support HTTPS and how to specify which pages must be served over this protocol.

We've explored the protection Wicket offers against CSRF attacks with URL encryption and the *ResourceIsolationRequestCycleListener*, or the (now deprecated) *CsrfPreventionRequestCycleListener*. This was followed by an explanation of the Content Security Policy used by Wicket and how to tune this for your application. We've also explained Cross-Origin Opener Policy and Cross-Origin Embedder policy to achieve cross-origin isolation, and how to configure the policies for your application.

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.

Chapter 23. Test Driven Development with Wicket

Test Driven Development 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 5.x.



from version 10 module *wicket-tester* is required as test dependency in order to access the utility classes used in this chapter.

23.1. 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:

```
▼ src
  ► main
  ▼ test
    ▼ java
      ▼ org
        ▼ wicketTutorial
          Start.java
          TestHomePage.java
```

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.test.WicketTester*. This 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...).

23.1.1. 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 switchLabelText(){
    //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);
//...
```

23.1.2. 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");
}
//...
```

23.1.3. Testing components in isolation

Method *startComponentInPage(Component)* can be used to test a component in isolation. The target component is rendered in an automatically generated page and both *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.startComponentInPage(field);
    Assert.assertNotNull(field.get("mesuramentUnit"));
}
//...
```

23.1.4. 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 information 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.

getLastResponse() should be used to assert the status code, response headers, binary content and anything that is part of the HTTP response.

23.1.5. 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");
//...
```

23.1.6. 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));
//...
```

23.1.7. 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);
}

```

23.1.8. 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));
//...

```

23.1.9. Using a custom servlet context

In [paragraph 16.13](#) 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...
}
```



After a test method has been executed, we may need to clear any possible side effect occurred to the *Application* and *Session* objects. This can be done invoking *WicketTester*'s method *destroy()*:

```
@After
public void tearDown(){
    //clear any side effect occurred during test.
    tester.destroy();
}
```

23.1.10. Setting request headers

In some cases you might need to set one or more specific request headers to make your test pass. This holds true when your application is protected against CSRF attacks as explained in [paragraph 22.5](#). In this particular case in order to make your tests green you must set header request `_sec-fetch-site` to `same-site` before clicking on a page link or before invoking a callback URL:

```
import static
org.apache.wicket.protocol.http.FetchMetadataResourceIsolationPolicy.SAME_SITE;
import static
org.apache.wicket.protocol.http.FetchMetadataResourceIsolationPolicy.SEC_FETCH_SITE_HE
```

```

ADER;

public class TestHomePage
{
    private WicketTester tester;

    @BeforeEach
    public void setUp()
    {
        tester = new WicketTester(new WicketApplication());
    }

    @Test
    public void homepageRendersSuccessfully()
    {
        //start and render the test page
        tester.startPage(HomePage.class);

        tester.addRequestHeader(SEC_FETCH_SITE_HEADER, SAME_SITE);
        tester.clickLink("click");
    }
}

```



keep in mind that request headers are immediately discarded after the use and thus are not re-used for following requests.

23.2. 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.

23.2.1. 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();
}
```

23.2.2. 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");
}
```

23.2.3. 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());
}
```


23.3. 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.test*.

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 `` 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 class attribute. In the

code above we have used this method to test that our markup contains two tags having attribute class equal to myClass.

Another utility class that comes in handy when we want to test components markup is *ComponentRenderer* in package *org.apache.wicket.core.util.string*. The purpose of this class is to render a page or a component in isolation with its static methods *renderComponent* and *renderPage*. Both methods return the generated markup as *CharSequence*:

```
@Test
public void customComponentMarkupTest()
{
    //instantiate MyComponent
    MyComponent myComponent = //...

    //render and save component markup
    String componentMarkup = ComponentRenderer.renderComponent(myComponent);

    //perform test operations
    //...
}
```

23.4. 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.

Chapter 24. Test Driven Development with Wicket and Spring

Since the development of many web applications is mostly based on the Spring framework for dependency injection and application configuration in general, it's especially important to get these two frameworks running together smoothly not only when deployed on a running server instance itself but rather during the execution of JUnit based integration tests as well. Thanks to the *WicketTester* API provided by the Wicket framework itself, one can easily build high-quality web applications while practicing test driven development and providing a decent set of unit and integration tests to be executed with each build. As already mentioned previously, integration and configuration of our web applications is based on a lightweight Spring container meaning that the integration of Spring's *ApplicationContext* and a WicketTester API is essential to get our integration tests running. In order to explain how to achieve that integration in an easy and elegant fashion in your integration test environment, we'll first take a look at a configuration of these 2 framework beauties in a runtime environment.

24.1. Configuration of the runtime environment

In order to get the Wicket framework up to speed when your server is up and running, you usually configure a *WicketFilter* instance in your web application deployment descriptor file (*web.xml*) while passing it a single init parameter called *applicationClassName* that points to your main implementation class extending *org.apache.wicket.protocol.http.WebApplication* where all of your application-wide settings and initialization requirements are dealt with:

```
<filter>
  <filter-name>wicketfilter</filter-name>
  <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
  <init-param>
    <param-name>applicationClassName</param-name>
    <param-value>com.comsys.to.webapp.MyWebApplication</param-value>
  </init-param>
</filter>
```

In case you want to get Wicket application up and running while leaving the application configuration and dependency injection issues to the Spring container, the configuration to be provided within the deployment descriptor looks slightly different though:

```
<web-app>
  <filter>
    <filter-name>wicketfilter</filter-name>
    <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
    <init-param>
      <param-name>applicationFactoryClassName</param-name>
      <param-value>org.apache.wicket.spring.SpringWebApplicationFactory</param-
value>
    </init-param>
```

```

</filter>
<listener>
  <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml</param-value>
  </context-param>
</web-app>

```

The additional configuration part containing listener and context parameter definition is a usual Spring container related configuration detail. ContextLoaderListener is an implementation of standard Servlet API ServletContextListener interface provided by the Spring framework itself and is responsible for looking up an according bean definition file(s) specified by the context param above and creating an ApplicationContext instance during servlet context initialization accordingly. When integrating an ApplicationContext instance with Wicket, one of the beans defined in the above mentioned Spring bean definition file has to be your own specific extension of *org.apache.wicket.protocol.http.WebApplication*. You can either define a suitable bean in the bean definition file itself:

```

<beans>
  <bean id="myWebApp" class="com.comsysto.webapp.MyWebApplication"/>
</beans>

```

or use powerful classpath scanning feature of the Spring framework and annotate the MyWebApplication implementation with the appropriate *Component* annotation accordingly while enabling the Spring container to scan the according package(s) of your application for relevant bean definitions:

```

<beans>
  <context:component-scan base-package="com.comsysto.webapp" />
  <context:component-scan base-package="com.comsysto.webapp.service" />
  <context:component-scan base-package="com.comsysto.webapp.repository" />
</beans>

```

Either way, if everything goes well, you'll get a pre-configured ApplicationContext all set up during the startup of your web container. One of the beans in the ApplicationContext will be your own extension of Wicket's WebApplication type. SpringWebApplicationFactory implementation provided by the Wicket framework itself that you have defined as the *applicationFactoryClassName* in the configuration of your WicketFilter will then be used in order to retrieve that very same WebApplication bean out of your Spring ApplicationContext. The Factory expects one and only one extension of Wicket's very own WebApplication type to be found within the ApplicationContext instance at runtime. If no such bean or more than one bean extending WebApplication is found in the given ApplicationContext an according IllegalStateException will be raised and initialization of your web application will fail:

```

Map<?,?> beans =
BeanFactoryUtils.beansOfTypeIncludingAncestors(ac,WebApplication.class, false, false);
if (beans.size() == 0)
{
    throw new IllegalStateException("bean of type [" + WebApplication.class.getName()
+
        "]" not found");
}
if (beans.size() > 1)
{
    throw new IllegalStateException("more than one bean of type [" +
        WebApplication.class.getName() + "]" found, must have only one");
}

```

After the `WebApplication` bean has been successfully retrieved from the `ApplicationContext` via `SpringWebApplicationFactory`, `WicketFilter` will then, as part of its own initialization process, trigger both `internalInit()` and `init()` methods of the `WebApplication` bean. The latter one is the exact spot where the last piece of the runtime configuration puzzle between Wicket and Spring is to be placed :

```

@Component
public class MyWebApplication extends WebApplication {
    @Override
    protected void init() {
        super.init();

        getComponentInstantiationListeners().add(new SpringComponentInjector(this));
    }
}

```

`SpringComponentInjector` provided by the Wicket framework enables you to get dependencies from the `ApplicationContext` directly injected into your Wicket components by simply annotating these with the according *SpringBean* annotation.

24.2. Configuration of the JUnit based integration test environment

One of the main features of Apache Wicket framework is the ability to easily write and run plain unit tests for your Pages and all other kinds of Components that even include the verification of the rendering process itself by using JUnit framework and the `WicketTester` API only. When using Spring framework for application configuration together with Wicket, as we do, you can even use the same tools to easily write and run full blown integration tests for your web application as well. All you have to do is use [Spring's TestContext](#) framework additionally to configure and run your JUnit based integration tests. The Spring Framework provides a set of Spring specific annotations that you can use in your integration tests in conjunction with the `TestContext` framework itself in

order to easily configure an according `ApplicationContext` instance for your tests as well as for appropriate transaction management before, during and after your test execution. Following code snippet represents a simple JUnit 4 based test case using Spring's specific annotations in order to initialize an `ApplicationContext` instance prior to executing the test itself:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"classpath:WEB-INF/applicationContext.xml"})
@TransactionConfiguration(transactionManager = "txManager", defaultRollback = false)
public class LoginPageTest {

    private WicketTester tester;

    @Autowired
    private ApplicationContext ctx;

    @Autowired
    private MyWebApplication myWebApplication;

    @Before
    public void setUp() {
        tester = new WicketTester(myWebApplication);
    }

    @Test
    @Transactional
    @Rollback(true)
    public void testRenderMyPage() {
        tester.startPage(LoginPage.class);
        tester.assertRenderedPage(LoginPage.class);
        tester.assertComponent("login", LoginComponent.class);
    }
}
```

By defining three annotations on the class level (see code snippet above) in your test, Spring's `TestContext` framework takes care of preparing and initializing an `ApplicationContext` instance having all the beans defined in the according Spring context file as well as the transaction management in case your integration test includes some kind of database access. Fields marked with *Autowired* annotation will be automatically dependency injected as well so that you can easily access and use these for your testing purposes. Since `MyWebApplication`, which extends `Wicket's WebApplication` type and represents the main class of our web application, is also a bean within the `ApplicationContext` managed by Spring, it will also be provided to us by the test framework itself and can be easily used in order to initialize a `WicketTester` instance later on during the execution of the test's `setUp()` method. With this kind of simple, annotation based test configuration we are able to run an integration test that verifies whether a `LoginPage` gets started and initialized, whether the rendering of the page runs smoothly and whether the page itself contains a `LoginComponent` that we possibly need in order to process user's login successfully.

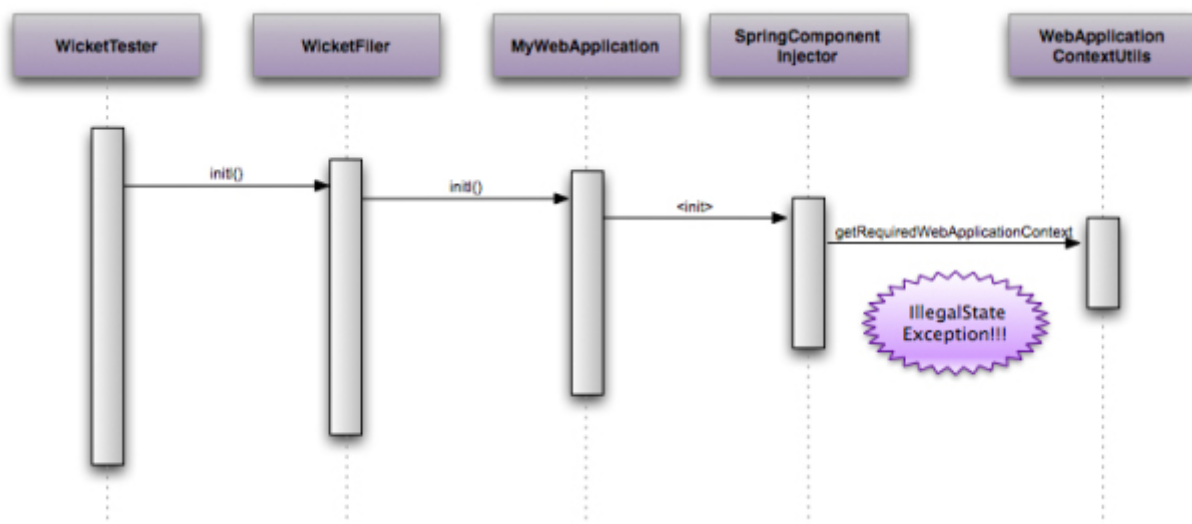
When you run this test though, you'll unfortunately get the following exception raised:

```

java.lang.IllegalStateException: No WebApplicationContext found: no
ContextLoaderListener registered?
    at org.springframework.web.context.support.WebApplicationContextUtils.
    getRequiredWebApplicationContext(WebApplicationContextUtils.java:84)
    at org.apache.wicket.spring.injection.annot.
    SpringComponentInjector.<init>(SpringComponentInjector.java:72)
    at com.comsys.to.serviceplatform.uiwebapp.MyWebApplication.
    initializeSpringComponentInjector(MyWebApplication.java:59)
    at com.comsys.to.serviceplatform.uiwebapp.MyWebApplication.
    init(MyWebApplication.java:49)
    at org.apache.wicket.protocol.http.WicketFilter.
    init(WicketFilter.java:719)
    at org.apache.wicket.protocol.http.MockWebApplication.
    <init>(MockWebApplication.java:168)
    at org.apache.wicket.util.test.BaseWicketTester.
    <init>(BaseWicketTester.java:219)
    at org.apache.wicket.util.test.WicketTester.
    <init>(WicketTester.java:325)
    at org.apache.wicket.util.test.WicketTester.
    <init>(WicketTester.java:308)

```

As you can see above, the Exception gets raised during the initialization of the *WicketTester* instance even before the actual test method gets executed. Even though we have applied rather cool and simple annotation based test configuration already described and passed in perfectly well prepared *ApplicationContext* instance to the *WicketTester* instance in the constructor, somewhere down the rabbit hole someone complained that no *WebApplicationContext* instance could have been found which seems to be required in order to initialize the *WicketTester* properly.



The problem that we run against here is due to the fact that *SpringComponentInjector* during its own initialization is trying to get hold of an according Spring's *ApplicationContext* instance that would normally be there in a runtime environment but does not find any since we are running in a test environment currently. *SpringComponentInjector* delegates to Spring's own *WebApplicationContextUtils* class to retrieve the instance of *ApplicationContext* out of the *ServletContext* which is perfectly fine for a runtime environment but is unfortunately failing in a

test environment:

```
public static WebApplicationContext getRequiredWebApplicationContext(ServletContext
sc)
    throws IllegalStateException {

    WebApplicationContext wac = getWebApplicationContext(sc);
    if (wac == null) {
        throw new IllegalStateException("No WebApplicationContext found: no
ContextLoaderListener registered?");
    }
    return wac;
}
```

If you still remember we defined a `ContextLoaderListener` in our `web.xml` file as part of the configuration of our runtime environment that makes sure an according `WebApplicationContext` instance gets initialized and registered against the `ServletContext` properly. Luckily, this problem can easily be solved if we slightly change the way we initialize `SpringComponentInjector` in our main `MyWebApplication` class. Apart from the constructor that we have used so far, there is another constructor in the `SpringComponentInjector` class that expects the caller to provide it with an according `ApplicationContext` instance rather than trying to resolve one on its own:

```
public SpringComponentInjector(WebApplication webapp, ApplicationContext ctx,
    boolean wrapInProxies)
{
    if (webapp == null)
    {
        throw new IllegalArgumentException("Argument [[webapp]] cannot be null");
    }

    if (ctx == null)
    {
        throw new IllegalArgumentException("Argument [[ctx]] cannot be null");
    }

    // store context in application's metadata ...
    webapp.setMetaData(CONTEXT_KEY, new ApplicationContextHolder(ctx));

    // ... and create and register the annotation aware injector
    InjectorHolder.setInjector(new AnnotSpringInjector(new ContextLocator(),
wrapInProxies));
}
```

In order to use this constructor instead of the one we used previously, we now obviously need to get hold of the *ApplicationContext* instance on our own in our *MyWebApplication* implementation. The easiest way to do this is to use Spring's own concept of [lifecycle callbacks](#) provided to the beans managed by the Spring container. Since our *MyWebApplication* is also a bean managed by the Spring container at runtime (enabled by the classpath scanning and *Component* annotation on a

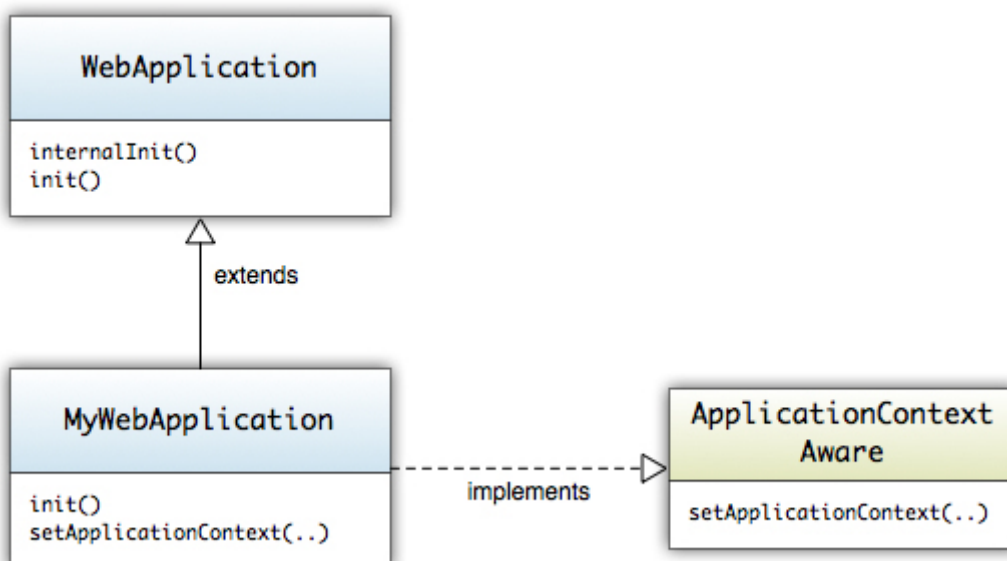
type level), we can declare it to implement *ApplicationContextAware* interface which ensures that it gets provided with the *ApplicationContext* instance that it runs in by the Spring container itself during startup.

```
public interface ApplicationContextAware {  
  
    void setApplicationContext(ApplicationContext applicationContext) throws  
    BeansException;  
  
}
```

So the relevant parts of *MyWebApplication* type will now look something like the following code snippet:

```
@Component  
public class MyWebApplication extends WebApplication implements  
ApplicationContextAware {  
    @Override  
    protected void init() {  
        addComponentInstantiationListener(new SpringComponentInjector(this, ctx,  
true));  
    }  
  
    public void setApplicationContext(ApplicationContext applicationContext) throws  
BeansException {  
        this.ctx = applicationContext;  
    }  
}
```

For additional clarification of how *MyWebApplication* now relates to both Wicket and Spring framework here is an according class diagram:



24.3. Summary

With the configuration outlined above, no additional modifications are required to the test itself. It's going to turn green now. This way you can use exactly the same Spring context configuration that you'd use in your runtime environment for running your JUnit based integration tests as well.

Chapter 25. Wicket Best Practices

This section is addressed to developers, who have already made their first experiences with Apache Wicket. Developers who get into Wicket often have difficulties with it because they apply the typical JSF and Struts patterns and approaches. These frameworks primarily use procedural programming methods. In contrast Wicket is strongly based on object oriented patterns. So forget all Struts and JSF patterns, otherwise you won't have fun with Wicket in the long run.

25.1. Encapsulate components correctly

A component should be self-contained. The user of a component should neither have to know nor care about its internal structure. She should just be familiar with its external interfaces and its documentation in order to be able to use it. This means in detail: Every component that extends Wicket's own Panel type (thus is a Panel itself) must provide its own HTML template. In contrast, when a component extends the classes *WebMarkupContainer* or *Form*, there is no HTML template. This implies that you should add components through composition in *WebMarkupContainer* or *Form*.

Listing 1:

```
// Poor component
public class RegistrationForm extends Form<Registration> {
    public RegistrationForm(String id, IModel<Registration> regModel) {
        super(id, new CompoundPropertyModel<Registration>(regModel))
        // Wrong: RegistrationForm provides its own components
        add(new TextField("username"));
        add(new TextField("firstname"));
        add(new TextField("lastname"));
    }
}
```

This snippet is an example for a poor component. The user of the *RegistrationForm* must know the internal structure of the markup and component in order to use it.

Listing 2:

```
public class RegistrationPage extends Page {
    public RegistrationPage(IModel<Registration> regModel) {
        Form<?> form = new RegistrationForm("form");
        form.add(new SubmitButton("register") {
            public void onSubmit() {
                // do something
            }
        });
        add(form);
    }
}
```

```

<html>
<body>
  <form wicket:id="form">
    <!-- These are internal structure information from RegistrationForm -->
    Username <input type="text" wicket:id="username"/>
    First name <input type="text" wicket:id="firstname"/>
    Last name <input type="text" wicket:id="lastname"/>
    <!-- Above new components from page which the user knows -->
    <input type="submit" wicket:id="register" value="Register"/>
  </form>
</body>
</html>

```

The code above shows the usage of the poor component in the *RegistrationPage*. You can see that the input fields *firstname*, *lastname* and *username* get used, even though these components are not added explicitly to the *RegistrationPage*. Avoid this, because other developers cannot directly see that the components were added in *RegistrationPage* class.

Listing 3:

```

// Good component
public class RegistrationInputPanel extends Panel{
    public RegistrationInputPanel(String id, IModel<Registration> regModel) {
        super(id, regModel);
        IModel<Registration> compound = new
CompoundPropertyModel<Registration>(regmodel);
        Form<Registration> form = new Form<Registration>("form", compound);
        // Correct: Add components to Form over the instance variable
        form.add(new TextField("username"));
        form.add(new TextField("firstname"));
        form.add(new TextField("lastname"));
        add(form);
    }
}

```

```

<html>
<body>
  <wicket:panel>
    <form wicket:id="form">
      Username <input type="text" wicket:id="username"/>
      First name <input type="text" wicket:id="firstname"/>
      Last name <input type="text" wicket:id="lastname"/>
    </form>
  </wicket:panel>
</body>
</html>

```

Now we have a properly encapsulated input component which provides its own markup. Furthermore you can see the correct usage of a Wicket *Form*. The components get added by calling *form.add(Component)* on the instance variable. On the other hand, it is allowed to add behaviours and validators over inheritance, because those do not have markup ids which must be bound.

With that, the usage of *RegistrationInputPanel* is much more intuitive. There is no markup of other embedded components present anymore, just markup of components which get directly added. The *RegistrationPage* provides its own form that delegates the submit to all Wicket nested forms which are contained in the component tree.

Listing 4:

```
public class RegistrationPage extends Page {
    public RegistrationPage(IModel<Registration> regModel) {
        Form<?> form = new Form("form");
        form.add(new RegistrationInputPanel("registration", regModel);
        form.add(new SubmitButton("register") {
            public void onSubmit() {
                // do something
            }
        });
        add(form);
    }
}
```

```
<html>
<body>
    <form wicket:id="form">
        <div wicket:id="registration">
            Display the RegistrationInputPanel
        </div>
        <input type="submit" wicket:id="register" value="Register"/>
    </form>
</body>
</html>
```

25.2. Put models and page data in fields

In contrast to Struts, Wicket pages and components are no singletons, they are stateful and session-scoped. This enables us to store user-specific information within pages and components. The information should be stored in fields. This way you can access the information within a class while avoiding long method signatures only for passing the same information around. Instances of components can exist for several requests. For example, a page with a form which gets submitted and produces validation errors uses the same page instance. Furthermore the same page instance gets used when the user presses the back button of the browser and resubmits this formular again. Information which gets passed by the constructor should be assigned to fields (normally this must be models). When storing information in fields you should consider that the information is

serializable, because the pages are stored using Java serialization. By default Wicket stores pages on the hard disk. A non-serializable object leads to *NullPointerExceptions* and *NonSerializableExceptions*. Additionally, big data (like binary stuff) should not be stored directly in fields because this can cause performance losses and memory leaks during serialization and deserialization. In this case, you should use the *LoadableDetachableModel* which can be assigned to a field because this provides an efficient mechanism to load and detach data.

25.3. Correct naming for Wicket IDs

For many developers, naming is a dispensable thing, but I think it is one of the major topics in software development. With the help of correct naming, you can easily identify the business aspects of a software component. Additionally good naming avoids unnecessary and bad comments.

Bad namings for Wicket-IDs are *birthdateTextField*, *firstnameField* and *addressPanel*. Why? The naming contains two aspects: A technical aspect ("*TextField*") and the business aspect ("*birthdate*"). Only the the business aspect is relevant because both the HTML template as well as the Java code already contain the technical details (*new TextField("birthdate")*). Additionally, such names add a lot of effort when you do technical refactorings, e.g. if you have to replace a *TextField* by a *DatePicker* and the Wicket ID *birthdateTextField* becomes *birthdateDatePicker*. Another reason for avoiding technical aspects in Wicket IDs is the *CompoundPropertyModel*. This model delegates the properties to its child components named by Wicket IDs (see listing 3). For example the *TextField username* automatically calls *setUsername()* and *getUsername()* on the *Registration* object. A setter like *setUsernameTextfield()* would be very inconvenient here.

25.4. Avoid changes at the component tree

You should consider the component tree as a constant and fixed skeleton which gets revived when its model is filled with data like a robot without brain. Without brain the robot is not able to do anything and is just a dead and fixed skeleton. However, when you fill it with data, it becomes alive and can act. There is no need for changing hardware when filling it with data.

In Wicket, you should manipulate the component tree as little as possible. Consequently, you should avoid calling methods like *Component.replace(Component)* and *Component.remove(Component)*. Calling these methods indicates missing usage or misuse of Wicket's models. Furthermore the component trees should not be constructed using conditions (see listing 5). This reduces the possibility of reusing the same instance significantly.

Listing 5:

```
// typical for struts
if(MySession.get().isNotLoggedIn()) {
    add(new LoginBoxPanel("login"))
}
else {
    add(new EmptyPanel("login"))
}
```

Instead of constructing *LoginBoxPanel* conditionally, it is recommended to always add the panel and set the visibility within *onConfigure()*. So the component *LoginBoxPanel* is responsible for displaying itself. We move the responsibility into the same component which executes the login. Brilliant! Cleanly encapsulated business logic. There is no decision from outside, the component handles all the logic. You can see another example in "Implement visibilities of components correctly".

25.5. Implement visibilities of components correctly

Visibility of components is an important topic. In Wicket you control any component's visibility via the methods *isVisible()* and *setVisible()*. These methods are within Wicket's base class *Component* and therefore it is applicable for every component and page. Let's have a look at a concrete example of *LoginBoxPanel*. The panel just gets displayed when the user is not logged in.

Listing 6:

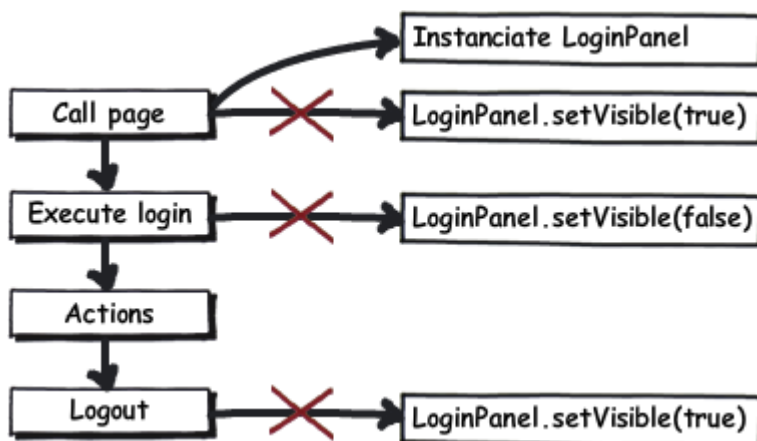
```
// Poor implementation
LoginBoxPanel loginBox = new LoginBoxPanel("login");
loginBox.setVisible(MySession.get().isNotLoggedIn());
add(loginBox);
```

Listing 6 shows a poor implementation, because a decision about the visibility is made while instantiating the component. Again, in Wicket instances of components exist for several requests.

To reuse the same instance you have to call *loginBox.setVisible(false)*. This is very unhandy, because we always have to call *setVisible()* and manage the visibility manually from the outside.

This approach is error-prone and fragile, because we always have to pay attention to setting the correct information every time. But this is often forgotten because the logic might be widely spread over the code.

The solution is the Hollywood principle: "Don't call us, we'll call you". Take a look at the following diagram illustrating an application flow with some calls. We avoid three calls through the [Hollywood-Principle](#) and we just have to instantiate the *LoginBoxPanel*.



Listing 7:

```
public class LoginBoxPanel {
    // constructor etc.
    @Override
    public void onConfigure() {
        setVisible(MySession.get().isNotLoggedIn());
    }
};
```

Now the control over visibility has been inverted, the *LoginBoxPanel* decides on its visibility autonomously.

For each call of *onConfigure()* there is a refreshed interpretation of the login state. Hence, there is no additional state that might be outdated. The logic is centralized in one line code and not spread throughout the application. Furthermore, you can easily identify that the technical aspect of visibility correlates to the business aspect "logged in".



The same approach can be used to control when a component is enabled.



Forms which are within an inactive or invisible component do not get executed.

Note that there are cases in which calls to *setVisible()* and *setEnabled()* seem unavoidable. For example, the user presses a button to display an inlined registration form. In general, you can apply the following rules: data driven components are configured by their model, and the button modifies the model. Listing 8 presents a model-bound visibility.

Listing 8:

```
new Label("headline", headlineModel) {
    @Override
    public void onConfigure() {
        // Headline visible only if text starts with "Berlusconi"
        String headline = getModelObject();
        setVisible(headline.startsWith("Berlusconi"));
    }
}
```



While possible, overriding *isVisible()* may lead to unpredictable results under some conditions. See [WICKET-3171](#), [WICKET-6946](#) and dev@wicket.apache.org mailing list: [overriding isVisible bad?](#).

25.6. Always use models

Always use models - period! Do not pass raw objects directly to components. Instances of pages and components can exist for several requests. If you use raw objects, you cannot replace them later. An example is an entity which gets loaded at each request within a *LoadableDetachableModel*. The entity manager creates a new object reference, but the page would keep the obsolete instance.

Always pass *IModel* in the constructor of your components:

Listing 9:

```
public class RegistrationInputPanel extends Panel {
    // Correct: The class Registration gets wrapped by IModel
    public RegistrationInputPanel(String id, IModel<Registration> regModel) {
        // add components
    }
}
```

This code can use any implementation of *IModel*, e.g. the class *Model*, a *PropertyModel* or a custom implementation of *LoadableDetachableModel* which loads and persists the values automatically. The model implementations gets very easy to replace. You - as a developer - just need to know: if I call *IModel.getObject()*, I will get an object of type *Registration*. Where the object comes from is within the responsibility of the model implementation and the calling component. For example you can pass the model while instanciating the component. If you avoid using models, you will almost certainly have to modify the component tree sooner or later which forces you to duplicate states and thus produce unmaintainable code. Additionally, you should use models due to serialization issues. Objects which get stored in fields of pages and components get serialized and deserialized on each request. This can be inefficient in some cases.

25.7. Do not unwrap models within the constructor hierarchy

Avoid unwrapping models within the constructor hierarchy, i.e. do not call *IModel.getObject()* within any constructor. As already mentioned, a page instance can exist for several page requests, so you might store obsolete and redundant infomation. It is reasonable to unpack Wicket Models at events (user actions), that are methods like *onUpdate()*, *onClick()* or *_onSubmit()*:

Listing 10:

```
new Form<Void>("register") {
    public void onSubmit() {
        // correct, unwrap model in an event call
        Registration reg = registrationModel.getObject()
        userService.register(reg);
    }
}
```

An additional possibility to unwrap models is via overriding methods like *isVisible()*, *isEnabled()* or *onBeforeRender()*.

25.8. Pass models extended components

Always try to pass models on to the parent component. By that, you ensure that at the end of every

request the method *IModel.detach()* gets called. This method is responsible for a data cleanup. Another example: you have implemented your own model which persists the data in the *detach()* method. So the call of *detach()* is necessary for that your data gets persisted. You can see an exemplary passing to the super constructor here:

Listing 11:

```
public class RegistrationInputPanel extends Panel{
    public RegistrationInputPanel(String id, IModel<Registration> regModel) {
        super(id, regModel)
        // add components
    }
}
```

25.9. Validators must not change any data or models

Validators should just validate. Consider a bank account form which has a *BankFormValidator*. This validator checks the bank data over a webservice and corrects the bank name. Nobody would expect that a validator modifies information. Such logic has to be located in *Form.onSubmit()* or in the event logic of a button.

25.10. Do not pass components to constructors

Do not pass entire components or pages to constructors of other components.

Listing 12:

```
// Bad solution
public class SettingsPage extends Page {
    public SettingsPage (IModel<Settings> settingsModel, final Webpage backToPage) {
        Form<?> form = new Form("form");
        // add components
        form.add(new SubmitButton("changeSettings") {
            public void onSubmit() {
                // do something
                setResponsePage(backToPage);
            }
        });
        add(form);
    }
}
```

The *SettingsPage* expects the page which should be displayed after a successful submit to be passed to its constructor. This solution works, but is very bad practice. You need to know during the instantiation of *SettingsPage* where you want to redirect the user. This requires a predetermined order of instantiation. It is better to order the instantiation based on business logic (e.g. the order in the HTML template). Furthermore, you need an unnecessary instance of the next success page

which might never be displayed. The solution is once again the Hollywood principle. For this you create an abstract method or a hook:

Listing 13:

```
// Good solution
public class SettingsPage extends Page {
    public SettingsPage (IModel<Settings> settingsModel) {
        Form<?> form = new Form("form");
        // add components
        form.add(new SubmitButton("changeSettings") {
            public void onSubmit() {
                // do something
                onSettingsChanged();
            }
        });
        add(form);
    }

    // hook
    protected void onSettingsChanged() {
    }

    // The usage of the new component
    Link<Void> settings = new Link<Void>("settings") {
        public void onClick() {
            setResponsePage(new SettingsPage(settingsModel) {
                @Override
                protected void onSettingsChanged() {
                    // reference to the current page
                    setResponsePage(this);
                }
            });
        }
    }
    add(settings);
}
```

This solution has more code, but it is more flexible and reuseable. We can see there is an event *onSettingsChanged()* and this event is called after a successful change. Furthermore, there is the possibility to execute additional code besides setting the next page. For example, you can display messages or persist information.

25.11. Use the Wicket session only for global data

The Wicket session is your own extension of Wicket's base session. It is fully typed. There is no map structure to store information unlike the servlet session. You just should use Wicket's session for global data. Authentication is a good example for global data. The login and user information is required on nearly each page. For a blog application it would be good to know whether the user is an author who is allowed to compose blog entries. So you are able to hide or or show links to edit a

blog entry. In general you should store the whole authorization logic in Wicket's session, because it is a global thing and you would expect it there. Data of forms and flows which only span certain pages should not be stored in the session. This data can be passed from one page to the next via the constructor (see listing 14). As a consequence of this, the models and data have a clearly defined lifecycle that reflects the corresponding page flow.

Listing 14:

```
public class MyPage extends WebPage {
    IModel<MyData> myDataModel;

    public MyPage(IModel<MyData> myDataModel) {
        this.myDataModel = myDataModel;
        Link<Void> next = new Link<Void>("next") {
            public void onClick() {
                // do something
                setResponsePage(new NextPage(myDataModel));
            }
        };
        add(next);
    }
}
```

You should pass concrete information to the page. All models can simply be stored in fields because Wicket pages are user-specific instances and not singletons in contrast to Struts. The big advantage of this approach is that the data gets automatically cleaned up when a user completes or exits the page flow. No manual cleanup anymore! This is basically an automatic garbage collector for your session.

25.12. Do not use factories for components

The factory pattern is useful, but nevertheless not suitable for Wicket components.

Listing 15:

```
public class CmsFactory {
    public Label getCmsLabel(String markupId, final String url) {
        IModel<String> fragment = () -> loadSomeContent(url);
        Label result = new Label(markupId, fragment);
        result.setRenderBodyOnly(true);
        result.setEscapeModelStrings(false);
        return result;
    }

    public String loadContent(String url) {
        // load some content
    }
}
```

```
// create the component within the page:
public class MyPage extends WebPage {
    @SpringBean
    CmsFactory cmsFactory;

    public MyPage() {
        add(cmsFactory.getCmsLabel("id", "http://url.to.load.from"));
    }
}
```

This approach for adding a label from the *CmsFactory* to a page seems to be okay at first glance, but it comes with some disadvantages. There is no possibility to use inheritance anymore. Furthermore, there is no possibility to override *isVisible()* and *isEnabled()*. The factory could also be a Spring service which instantiates the component. A better solution is to create a *CmsLabel*.

Listing 16:

```
public class CmsLabel extends Label {
    @SpringBean
    CmsResource cmsResource;
    public CmsLabel(String id, IModel<String> urlModel) {
        super(id, urlModel);
        IModel<String> fragment = () ->
cmsResource.loadSomeContent(urlModel.getObject());
        setRenderBodyOnly(true);
        setEscapeModelStrings(false);
    }
}

// create the component within a page
public class MyPage extends WebPage {
    public MyPage() {
        add(new CmsLabel("id", Model.of("http://url.to.load.from")));
    }
}
```

The label in listing 16 is clearly encapsulated in a component without using a factory. Now you can easily create inline implementations and override *isVisible()* or other stuff. Naturally, you might claim "I need a factory to initialize some values in the component, e.g. a Spring service". For this you can create a implementation of *IComponentInstantiationListener*. This listener gets called on the super-constructor of every component. The most popular implementation of this interface is the *SpringComponentInjector* which injects Spring beans in components when the fields are annotated with *SpringBean*. You can easliy write and add your own implementation of *IComponentInstantiationListener*. So there is no reason for using a factory anymore. More information about the instantiation listener is located in Wicket's JavaDoc.

25.13. Every page and component must be tested

Every page and component should have a test. The simplest test just renders the component and validates its technical correctness. For example, a child component should have a matching wicket id in the markup. If the wicket id is not correctly bound - through a typo or if it was just forgotten - the test will fail. An advanced test could test a form, where a backend call gets executed and validated over a mock. So you can validate your component's behaviour. This is a simple way to detect and fix technical and business logic bugs during the build process. Wicket is very suitable for a test driven development approach. For instance, if you run a unit test which fails and shows a message that the wicket id not bound, you will avoid an unnecessary server startup (a server startup takes longer than running a unit test). This reduces the development turnaround. A disadvantage is the difficult testing possibility of AJAX components. However, the testing features of Wicket are much more sophisticated than in other web frameworks.

25.14. Avoid interactions with other servlet filters

Try to get within the Wicket world whenever possible. Avoid the usage of other servlet filters. For this you can use the *RequestCycle* and override the methods *onBeginRequest()* and *onEndRequest()*. You can apply the same to the *HttpSession*. The equivalent in Wicket is the *WebSession*. Just extend the *WebSession* and override the *newSession()*-method from the *Application* class. There are very few reasons to access the servlet interfaces. An example could be to read an external cookie to authenticate a user. Those parts should be properly encapsulated and avoided when possible. For this example, you could do the handling within the Wicket session because this is an authentication.

25.15. Cut small classes and methods

Avoid monolithic classes. Often I have seen that developers put the whole stuff into constructors. These classes are getting very unclear and chaotic because you use inline implementations over several levels. It is recommended to group logical units and extract methods with a correct business naming. This enhances the clarity and the understandability of the business aspect. When a developer navigates to a component, he is not interested in the technical aspect at first, however he just need the business aspect. To retrieve technical information of a component you can navigate to the method implementation. In case of doubt you should consider to extract separate components. Smaller components increase the chances of reuse and make testing easier. Listing 17 shows an example of a possible structuring.

Listing 17:

```
public class BlogEditPage extends WebPage {
    private IModel<Blog> blogModel;

    public BlogEditPage(IModel<Blog> blogModel) {
        super(new PageParameters());
        this.blogModel = blogModel;
        add(createBlogEditForm());
    }
}
```

```
private Form<Blog> createBlogEditForm() {
    Form<Blog> form = newBlogEditForm();
    form.add(createHeadlineField());
    form.add(createContentField());
    form.add(createTagField());
    form.add(createViewRightPanel());
    form.add(createCommentRightPanel());
    form.setOutputMarkupId(true);
    return form;
}

// more methods here
}
```

25.16. The argument "Bad documentation"

It is a widespread opinion that Wicket has a bad documentation. This argument is just partly correct. There are a lot of code samples and snippets which can be used as code templates. Furthermore, there is a big community that answers complex questions very quickly. In Wicket it is very hard to document everything, because nearly everything is extensible and replaceable. If a component is not completely suitable, you will extend or replace it. Working with Wicket means permanently navigating through code. For example, just consider validators. How can I find all validators that exist? Open the interface *IValidator* (Eclipse: Ctrl + Shift + T) and then open the type hierarchy (Ctrl + T). Now we can see all the validators existing in Wicket and our project.

Type hierarchy of 'org.apache.wicket.validation.IValidator':



25.17. Summary

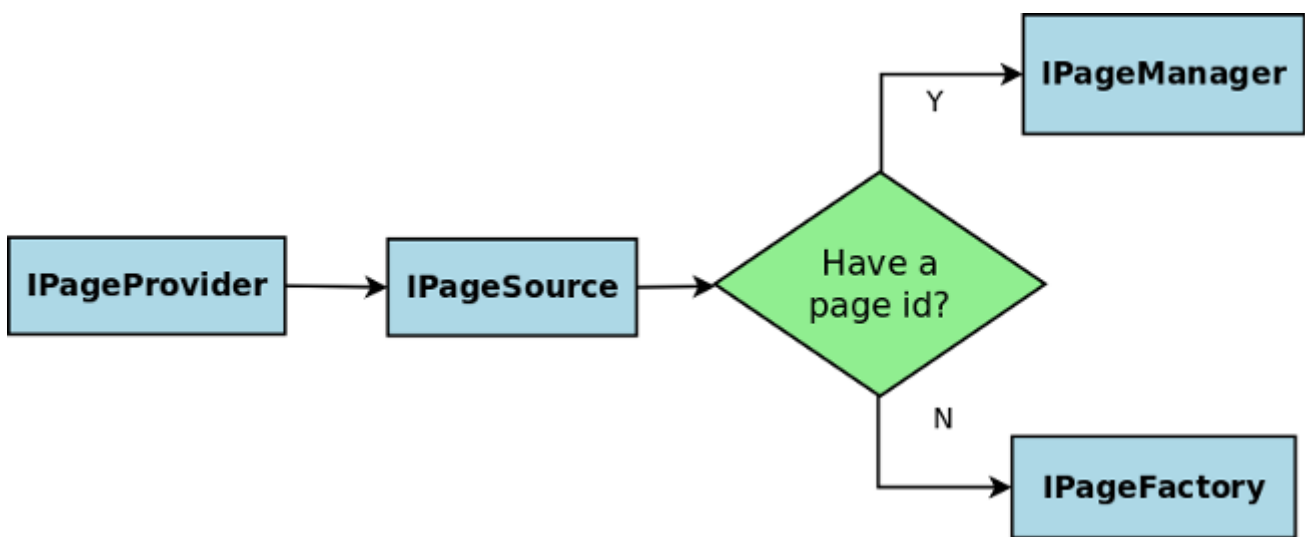
The best practices presented in this chapter should help you to write better and more maintainable code in Wicket. All described methodologies were already proven in a few Wicket projects. If you follow these advices, your Wicket projects will get future-proof and hopefully successful.

Chapter 26. Wicket Internals

26.1. Page storing

During request handling, Wicket manages page instances through interface *org.apache.wicket.request.handler.IPageProvider*. This interface creates a new page instance or loads a previously serialized page instance if we provide the corresponding page id. *IPageProvider* delegates page creation and retrieval to interface *org.apache.wicket.request.mapper.IPageSource*. When page class is provided *IPageSource* delegates page creation to interface *org.apache.wicket.IPageFactory*, while when page id is provided it uses interface *org.apache.wicket.page.IPageManager* to load the previously serialized page.

The following workflow diagram summarizes the mechanism seen so far:



26.1.1. IPageManager

org.apache.wicket.page.IPageManager's task is to manage which pages have been used in a request and store their last state in the backing stores, namely *IPageStore*. The default implementation *org.apache.wicket.page.PageManager* uses a chain of *IPageStore* to collect all stateful pages which have been used in the request cycle and store them for a later use.



Keep in mind that more than one page can be used in a single request if, for example, *setResponsePage()* or *RestartResponseException* are used.

As said on [page_caching](#) paragraph 8.2.4 stateful pages are stored in a session-relative file using a two-levels cache to speedup the access. This process is made possible by the different implementations of *IPageStore* that are part of the default chain and that will be introduced in the next paragraph.



Wicket gets the default *IPageManager* using a supplier interface called *IPageManagerProvider*, hence to use a custom *IPageManager* implementation we must register a specific *IPageManagerProvider* via *org.apache.wicket.Application.setPageManagerProvider(IPageManagerProvider)*.

26.1.2. Default IPageStore chain

org.apache.wicket.pageStore.IPageStore's role is to mediate the storing and loading of page instances. The default chain of *IPageStore* used by Wicket contains the following ordered list of *IDataStore*:

- **RequestPageStore:** collect all page instances involved in the last request. During the detach stage, stateful pages are passed to the other steps of the chain to be persisted on file.
- **InSessionPageStore:** with *InSessionPageStore* the default chain keeps the last rendered page instance into the HTTP session for fast access.
- **SerializingPageStore:** *SerializingPageStore* turns page instances into a more serialization-friendly format represented by class *org.apache.wicket.pageStore.SerializedPage*. This is a struct of:

```
{
    pageType: String,
    pageId : int,
    data : byte[]
}
```

i.e. this is the serialized page instance (data) plus additional information needed to be able to easily find it later (pageId, pageType).

- **AsynchronousPageStore:** The role of *AsynchronousPageStore* is to detach the http worker thread from waiting for the write of the page bytes to the disk. To disable it use: *org.apache.wicket.settings.StoreSettings.setAsynchronous(false)*. *AsynchronousPageStore* can delay the storage of page's bytes for at most *org.apache.wicket.settings.StoreSettings.setAsynchronousQueueCapacity(int)* pages. If this capacity is exceeded then the page's bytes are written synchronously to the backing *IPageStore*.
- **CryptingPageStore:** page instances might contain sensible informations, therefore it's important to have the chance to encrypt their content before persist them on disk. *CryptingPageStore* encrypts *SerializedPage*'s with a 256 bit AES before passing them to the underlying *DiskPageStore*. By default this stage is disabled and not added to the default chain. To change this behavior we can use *org.apache.wicket.settings.StoreSettings.setEncrypted*.
- **DiskPageStore:** stores *SerializedPage* on a session-scoped file on disk. The location of the folder where the files are stored is configurable via *org.apache.wicket.settings.StoreSettings.setFileStoreFolder(File)*, by default the web container's work folder is used (ServletContext attribute 'javax.servlet.context.tempdir'). In this folder a sub-folder is created named '*applicationName-filestore*'. This folder contains a sub-folder for each active http session. This session folder contains a single file named 'data' which contains the bytes for the pages. The size of this 'data' file is configurable via *org.apache.wicket.settings.StoreSettings.setMaxSizePerSession(Bytes)*. When this size is exceeded the newly stored files overwrite the oldest ones.

26.1.3. InMemoryPageStore

An alternative *IPageStore* we can use in a custom chain is *org.apache.wicket.pageStore.InMemoryPageStore*. This implementation stores pages in a session-relative variable which can be limited by size or by page instance number.

26.1.4. DebugBar

Further insights which can be valuable during debugging can be retrieved using the *org.apache.wicket.devutils.debugbar.DebugBar* from *wicket-devutils.jar*. It's a panel which you simply add:

Java:

```
add(new DebugBar("debug"));
```

HTML:

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

26.2. Markup parsing and Autocomponents

26.2.1. Markup loading and parsing

Before rendering any component Wicket must retrieve its markup calling method *getMarkup()* of class *org.apache.wicket.Component*. This markup is an instance of interface *org.apache.wicket.markup.IMarkupFragment*. Markup is lazy loaded the first time we render the relative component and is cached at application level. The internal class that actually loads the markup is *org.apache.wicket.markup.MarkupFactory* and is part of application's markup settings:

```
//get current markup factory  
Application.get().getMarkupSettings().getMarkupFactory()
```

After the markup has been loaded by *MarkupFactory*, it's parsed with class *org.apache.wicket.markup.MarkupParser*. *MarkupFactory* creates a new *MarkupParser* with method *newMarkupParser(MarkupResourceStream resource)*. The effective markup parsing is performed with a chain of entities implementing interface *org.apache.wicket.markup.parser.IMarkupFilter*. The default set of *IMarkupFilters* used by *MarkupParser* takes care of different tasks such as HTML validation, comments removing, Wicket tags handling, etc...

To customize the set of *IMarkupFiltersS* used in our application we can create a subclass of *MarkupFactory* overriding method *newMarkupParser(MarkupResourceStream resource)*:

```
public MyMarkupFactory  
{
```

```
...
    public MarkupParser newMarkupParser(final MarkupResourceStream resource)
    {
        MarkupParser parser = super.newMarkupParser(resource);
        parser.add(new MyFilter());
        return parser;
    }
}
```

This custom class must be registered in the markup settings during application's initialization:

```
@Override
public void init()
{
    super.init();
    getMarkupSettings().setMarkupFactory(myMarkupFactory)
}
```

Usually we won't need to change the default configuration of *IMarkupFiltersS*, but it's important to be aware of this internal mechanism before we talk about another advanced feature, which is building auto components resolvers.

26.2.2. Auto components resolvers

Even if Wicket encourages developers to use just standard HTML in their markup code, in this guide we have seen a number of "special" tags (those starting with *wicket:*) that help us for specific tasks (e.g. *wicket:enclosure* tag). Wicket handles most of these tags creating a corresponding special component called *auto* component. This kind of components are resolved in two steps:

1. first their tag is identified by a *IMarkupFilters* which also takes care of assigning a unique tag id.
2. then during rendering phase when an auto-component is found a new component is created for it using one of the registered *org.apache.wicket.markup.resolver.IComponentResolver*:

```
public interface IComponentResolver extends IClusterable
{
    /**
     * Try to resolve a component.
     *
     * @param container
     *         The container parsing its markup
     * @param markupStream
     *         The current markupStream
     * @param tag
     *         The current component tag while parsing the markup
     * @return component or {@code null} if not found
     */
    public Component resolve(final MarkupContainer container, final MarkupStream
markupStream,
```

```
        final ComponentTag tag);  
    }
```

Registered *IComponentResolverS* can be retrieved through Application's settings:

```
Application.get()  
    .getPageSettings()  
    .getComponentResolvers()
```



An internal utility class named *org.apache.wicket.markup.resolver.ComponentResolvers* is also available to resolve autocomponents for the current markup tag.

Chapter 27. Wicket HTTP/2 Support (Experimental)

With Wicket 8.0.0-M2 the new HTTP/2 push API is supported which uses the PushBuilder.

The advantage of this is that you reduce the latency and thus save a lot of time in waiting for requests.

27.1. Example Usage

Currently there are different implementations for each server to be used until the Servlet 4.0 (JSR 369) specification reaches the final state.

Current supported servers are: * Eclipse Jetty 9.3+ * Apache Tomcat 8.5+ * RedHat Undertow 2+

For the setup you need to follow those steps:

1. Setup your server to use HTTP/2 and follow the instructions provided by the vendor specific documentation. (Because of HTTP/2 a HTTPS setup is also required)
2. Add the respective dependency for your web server to provide the push functionality.

```
<dependency>
  <groupId>org.apache.wicket.experimental.wicket8</groupId>
  <artifactId>wicket-http2-jetty</artifactId>
  <!--<artifactId>wicket-http2-tomcat</artifactId>-->
  <!--<artifactId>wicket-http2-undertow</artifactId>-->
  <version>0.X-SNAPSHOT</version>
</dependency>
```

3. Use the PushHeader Item like in this example page: Example:

```
public class HTTP2Page extends WebPage
{
    private static final long serialVersionUID = 1L;

    private transient Response webPageResponse;

    private transient Request webPageRequest;

    public HTTP2Page()
    {
        webPageResponse = getRequestCycle().getResponse();
        webPageRequest = getRequestCycle().getRequest();
        add(new Label("label", "Label"));
    }

    @Override
```

```

public void renderHead(IHeaderResponse response)
{
    super.renderHead(response);
    TestResourceReference instance = TestResourceReference.getInstance();
    response.render(CssHeaderItem.forReference(instance));
    response.render(new PushHeaderItem(this, webPageRequest, webPageResponse)
        .push(Arrays.asList(new PushItem(instance))));
}

@Override
protected void setHeaders(WebResponse response)
{
    // NOOP just disable caching
}
}

```

Basically the resource is pushed before the actual response of the component is send to the client (browser) and because of this the client does not need to send an additional request.

The PushHeaderItem behaves like explained in the following steps:

- When a browser requests the page with an initial commit everything is going to be pushed with (200)
- When a browser requests the page a second time resources are not pushed (304) not modified, because of the actual ResourceReferences headers
- When a browser requests the page a second time and the markup of the page has changed everything is going to be pushed again (200)
- When a browser requests the page a second time and resource references has been changed but not the page markup, all changed resource references are shipped via separate requests

Note: Chrome does not set cache headers if the https connection is not secure (self signed) / valid - so ensure that a valid https connection is available with your server. [Browser not caching files if HTTPS is used even if it's allowed by webserver via response headers](#) If you want to change the cache behavior to not only look at the markup of the page and based on this proceed the push, override the method **protected Time getPageModificationTime()** of the PushHeaderItem (for more information have a look at the javadoc)

To change the cache headers override the method **protected void applyPageCacheHeader()** of the PushHeaderItem

27.2. Create server specific http/2 push support

To create a server specific http/2 push support of the Wicket PushBuilder API just follow these steps:

1. Add the following dependency to your projects pom.xml (and of course adjust the version)

```
<dependency>
  <groupId>org.apache.wicket.experimental.wicket8</groupId>
  <artifactId>wicket-http2-core</artifactId>
  <version>0.X-SNAPSHOT</version>
</dependency>
```

2. Add a text file called *org.apache.wicket.IInitializer* into the folder `src/main/resources/META-INF/services/`
3. Add a single line with the name of the *IInitializer* class example: *org.apache.wicket.http2.Initializer* to the created file
4. Implement your own server specific PushBuilder class which implements the interface *org.apache.wicket.http2.markup.head.PushBuilder* This is an example how it was done for jetty:

```
public class Jetty9PushBuilder implements PushBuilder
{
    @Override
    public void push(HttpServletRequest httpRequest, String... paths)
    {
        Request request = RequestCycle.get().getRequest();
        HttpServletRequest httpRequest = (HttpServletRequest)
request.getContainerRequest();
        org.eclipse.jetty.server.PushBuilder pushBuilder =

org.eclipse.jetty.server.Request.getBaseRequest(httpRequest).getPushBuilder();
        for (String path : paths)
        {
            pushBuilder.path(path);
        }
        pushBuilder.push();
    }
}
```

5. Implement the class within the package *org.apache.wicket.http2.Initializer* and add your own server specific PushBuilder class to the `Http2Settings`. This is an example how it was done for jetty:

```
public class Initializer implements IInitializer
{
    /**
     * Initializes the push builder API of Jetty 9.3+
     */
    @Override
    public void init(Application application)
    {
        Http2Settings http2Settings = Http2Settings.Holder.get(application);
        http2Settings.setPushBuilder(new Jetty9PushBuilder());
    }
}
```



```
@Override
public void destroy(Application application)
{
    // NOOP
}
}
```

Chapter 28. Wicket Metrics Monitoring (Experimental)

The wicket-metrics module is available since Wicket 7.3.0 and contains a life measurement implementation to collect data of applications and visualize it.

You can see how many request your application served, how often components are created, initialized, configured or their detach method has been invoked and a lot of other additional information.

The module itself is using [Metrics of dropwizard](#) and [AspectJ](#) so that if you turn of the measurement it has no longer any effect

to your web application.

Keep in mind that AspectJ is licensed under the Eclipse Public License and you should provide the required license information.

28.1. Example setup

This is a little example how to setup wicket-metrics within a Apache Tomcat.

(1) Add the maven dependency to your project

```
<dependency>
  <groupId>org.apache.wicket.experimental.wicket8</groupId>
  <artifactId>wicket-metrics</artifactId>
  <version>0.X-SNAPSHOT</version>
</dependency>
```

(2) Just drop the jars of aspectjrt and aspectjweaver into the tomcat lib folder - you can download it from here <http://mvnrepository.com/artifact/org.aspectj/> (the metrics dependency is shipped with the project)

(3) Add the java agent to the jvm start options of your tomcat:
-javaagent:/pathToServer/lib/aspectjweaver-x.x.x.jar

(4) Add an aop.xml to your project's META-INF folder at the root of your classpath with the metrics you want to use (aspect tags) - if you don't want to enable a metrics just remove the aspect tag:

```
<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN"
"http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
<aspectj>
  <weaver options="-nowarn">
    <include within="org.apache.wicket..*" />
  </weaver>
  <aspects>
```

```

<!-- required -->
<aspect name="org.apache.wicket.metrics.aspects.WicketFilterInitAspect" />

<!-- optional -->
<aspect
name="org.apache.wicket.metrics.aspects.model.LoadableDetachableModelLoadAspect" />
<aspect
name="org.apache.wicket.metrics.aspects.requesthandler.IRequestHandlerDetachAspect" />
<aspect
name="org.apache.wicket.metrics.aspects.requesthandler.IRequestHandlerRespondAspect"
/>
<aspect
name="org.apache.wicket.metrics.aspects.resource.IResourceCreateAspect" />
<aspect name="org.apache.wicket.metrics.aspects.behavior.BehaviorCreateAspect"
/>
<aspect
name="org.apache.wicket.metrics.aspects.component.ComponentCreateAspect" />
<aspect
name="org.apache.wicket.metrics.aspects.component.ComponentOnConfigureAspect" />
<aspect
name="org.apache.wicket.metrics.aspects.component.ComponentOnDetachAspect" />
<aspect
name="org.apache.wicket.metrics.aspects.component.ComponentOnInitializeAspect" />
<aspect
name="org.apache.wicket.metrics.aspects.component.ComponentOnRenderAspect" />
<aspect
name="org.apache.wicket.metrics.aspects.component.ComponentSetResponsePageAspect" />
<aspect
name="org.apache.wicket.metrics.aspects.ajax.IPartialPageRequestHandlerAddAspect" />
<aspect
name="org.apache.wicket.metrics.aspects.ajax.IPartialPageRequestHandlerAppendJavaScriptAspect" />
<aspect
name="org.apache.wicket.metrics.aspects.ajax.IPartialPageRequestHandlerPrependJavaScriptAspect" />
<aspect
name="org.apache.wicket.metrics.aspects.resource.ResourceReferenceCreateAspect" />
<aspect name="org.apache.wicket.metrics.aspects.markup.WicketTagCreateAspect"
/>
<aspect
name="org.apache.wicket.metrics.aspects.request.WicketFilterRequestCycleUrlAspect" />
<aspect
name="org.apache.wicket.metrics.aspects.request.WicketFilterRequestCycleAspect" />
<aspect
name="org.apache.wicket.metrics.aspects.session.SessionCountListenerAspect" />
</aspects>
</aspectj>

```

- If you use the SessionCountListenerAspect you have to ensure that metadata-complete= [false] is set otherwise you have to add the listener yourself:

```
<listener>
  <listener-class>
    org.apache.wicket.metrics.aspects.session.SessionCountListener
  </listener-class>
</listener>
```

(5 - optional) To enable the JMX measurement write the following line into your init method of your Application (Now you are able to connect with jvisualvm to your server and have a look at the data):

```
WicketMetrics.getSettings().startJmxReporter();
```

To deactivate:

```
WicketMetrics.getSettings().stopJmxReporter();
```

To disable measurement:

```
WicketMetrics.getSettings().setEnabled(false);
```



IMPORTANT INFORMATION It is only possible to collect metrics for **one wicket filter per webapp** - don't declare more than one if you want to use wicket-metrics. The `WicketFilterInitAspect` is required so that the application can be resolved - otherwise runtime exceptions will be thrown. If you use the `SessionCountListener` you have to clear the session store if you restart the server - otherwise physically stored session will corrupt the data, because the count is initialized with 0. If you have set wicket-metrics as dependency you can open *wicket-metrics.template.xml* to get a full template of the *aop.xml*. For the weaver options refer to the AspectJ LTW configuration documentation: <https://eclipse.org/aspectj/doc/next/devguide/ltw-configuration.html>

28.2. Visualization with Graphite

To visualize the metrics with Graphite a little additional configuration is required:

(1) Add the additional maven dependency to your project:

```
<dependency>
  <groupId>io.dropwizard.metrics</groupId>
  <artifactId>metrics-graphite</artifactId>
  <version>${metrics.graphite.version}</version>
</dependency>
```

- the `metrics.graphite.version` should be the same as the `metrics` version of the `wicket-metrics` dependency. Check the Maven dependencies to ensure this.

(2) Add the following code to your Application's `init` method:

```
private GraphiteReporter reporter;

@Override
protected void init()
{
    MetricRegistry metricRegistry = WicketMetrics.getMetricRegistry();
    final Graphite graphite = new Graphite(new InetSocketAddress("127.0.0.1",
2003));
    reporter =
GraphiteReporter.forRegistry(metricRegistry).prefixedWith("WebApplications")

.convertRatesTo(TimeUnit.SECONDS).convertDurationsTo(TimeUnit.MILLISECONDS)
    .filter(MetricFilter.ALL).build(graphite);

    // Collects data every 5 seconds
    reporter.start(5, TimeUnit.SECONDS);
}

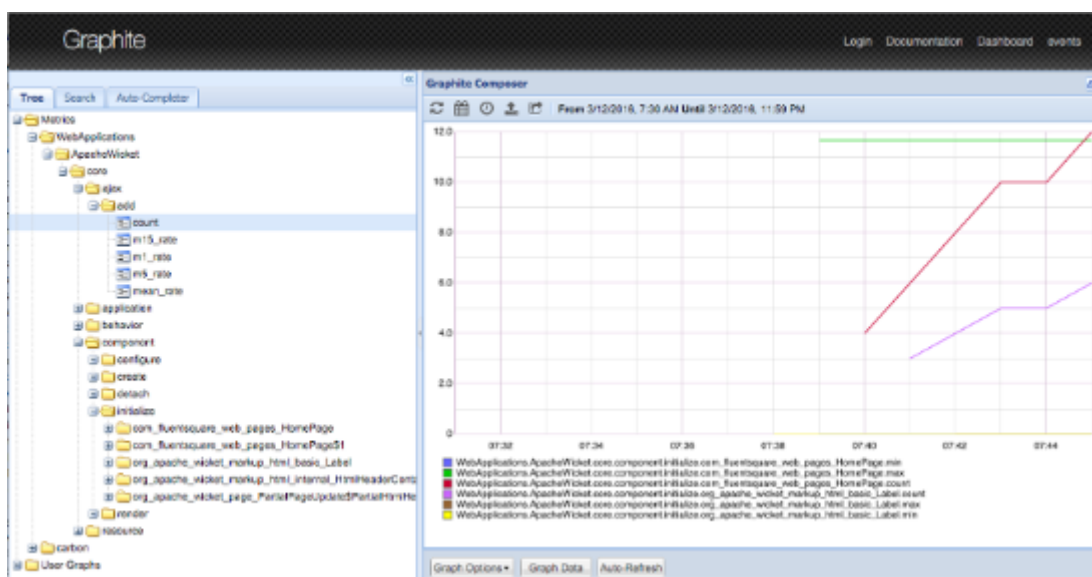
@Override
protected void onDestroy()
{
    super.onDestroy();
    reporter.stop();
}
```

(3) Install and setup graphite on your system. Example installation for Mac (beware that this is only a quickstart setup!):

- (1) Install homebrew: [brew](#)
- (2) Install [Git](#)
- (3) `brew install python`
- (4) `brew install cairo`
- (5) `brew install py2cairo`
- (6) `pip install Django==1.5`
- (7) `pip install "django-tagging<0.4"`
- (8) `sudo pip install carbon`
- (9) `pip install whisper`
- (10) `sudo pip install graphite-web`
- (11) `sudo pip install Twisted==11.1.0`
- (12) `sudo chown -R <your username>:staff /opt/graphite`

- (13) `cp /opt/graphite/conf/carbon.conf{.example,}`
- (14) `cp /opt/graphite/conf/storage-schemas.conf{.example,}`
- (15) `cd /opt/graphite/webapp/graphite`
- (16) `cp local_settings.py{.example,}`
- (17) `python manage.py syncdb`
- (18) `python /opt/graphite/bin/carbon-cache.py start`
- (19) `python /opt/graphite/bin/run-graphite-devel-server.py /opt/graphite`
- (20) Go to <http://localhost:8080>
 - (18) and (19) have to be executed if the mac has been restarted

(4) Now start your tomcat server configured like mentioned in the previous chapter.



28.3. Measured data

The data which is going to be measured depends on the wicket-metrics implementation. So it doesn't make any sense to collect time data

about `setResponsePage`, but it does for the constructor of components, to see if a component needs a long time to be created. You can

get the information about which data has been collected from out of the mbeans.

Here are some information about them:

- max - the maximal time for a task (created, initialized, etc.)
- min - the minimal time for a task (created, initialized, etc.)
- count - how often something happened (request count)

The structure is separated in the way that under core there are the kind of components measured and below that the type of operation

(created, initialized, detached). In this category every component is listed dynamically.

28.4. Write own measurements

There are only a two steps required to write own measurements for life data statistics in Wicket:

(1) Write a class which is named very close to what it measures. This class should extends `WicketMetrics` and should annotated with `@Aspect` and provide one method with a join point scanning for the target signature.

```
@Aspect
public class MySpecialAspect extends WicketMetrics
{
    @Around("execution(* my.package.MyClass.myMethod(..))")
    public Object aroundRequestProcessed(ProceedingJoinPoint joinPoint) throws
    Throwable
    {
        return measureTime("mycategory/someinformation/", joinPoint);
    }
}
```

- To measure time you need `@Around` because `measureTime` of `WicketMetrics` requires the `joinPoint` - the class name is appended with a slash at the end
- To only mark that a method is called you can use `mark` of `WicketMetrics` and apply `null` as a second parameter - if you apply a join point to mark the class name is appended with a slash at the end

(2) Add the class to your `aop.xml` and of course the package to scan for classes that are target for your measurements:

```
<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN"
"http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
<aspectj>
    <weaver options="-nowarn">
        <include within="org.apache.wicket..*" />
        <include within="my.components.package..*" />
    </weaver>
    <aspects>
        <!-- required -->
        <aspect name="org.apache.wicket.metrics.aspects.WicketFilterInitAspect" />

        <!-- own aspects -->
        <aspect name="my.aspect.package.MySpecialAspect" />

        <!-- wickets own metrics -->
        .....
    </aspects>
```

</aspectj>

Appendix A: Working with Maven

A.1. Switching Wicket to DEPLOYMENT mode

As pointed out in the note in [paragraph 4.2](#), 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 four 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 named *configuration*. 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 ...
```

The last option is to override `getConfigurationType()` method in your class that extends `WebApplication`):

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

    @Override
    public RuntimeConfigurationType getConfigurationType()
    {
        return RuntimeConfigurationType.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



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. 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> :

Create a Wicket Quickstart

With the quickstart you'll be up and running in seconds

Use the following wizard to generate a Quick Start Project using Maven. Paste the generated command line into a shell (DOS prompt or unix shell) and create a project with Wicket in a jiffy.

Before you start

The Quick Start Wizard uses [Apache Maven](#) to make it really fast to get started. You should have Maven installed and working before you can use the Quick Start wizard.

5 small steps to a web application

Use the following steps to quickly generate a project to get you started:

- 1 Fill in the Maven coordinates for your project in the wizard and select the appropriate Wicket version
- 2 Copy the generated commandline to your clipboard and paste it in a terminal (or a DOS box)
- 3 Open the project in your IDE of choice
- 4 Start the `Start` class in the `src/test/java` folder
- 5 Open your browser to <http://localhost:8080>

And you're done!

Quick Start Wizard

Fill in your project details in the wizard below and copy the generated command line to your clipboard.

Group ID

com.mycompany

Artifact ID

myproject

Wicket Version

7.6.0

Server to deploy on

Any but WildFly

generated command line

mvn archetype:generate -DarchetypeGroupId=org.apache.wicket
-DarchetypeArtifactId=wicket-archetype-quickstart -DarchetypeVersion=7.6.0
-DgroupId=com.mycompany -DartifactId=myproject
-DarchetypeRepository=https://repository.apache.org/ -DinteractiveMode=false

copy to clipboard

Table of Contents

[1 Before you start](#)

[2 5 small steps to a web application](#)

[3 Quick Start Wizard](#)

[4 Import the Quick Start in your IDE](#)

[4.1 Eclipse](#)

[4.2 IntelliJ IDEA](#)

[4.3 Netbeans](#)

Here we have to specify the root package of our project (GroupId), the project name (ArtifactId) and 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
| | | |
| | | | \---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>
...

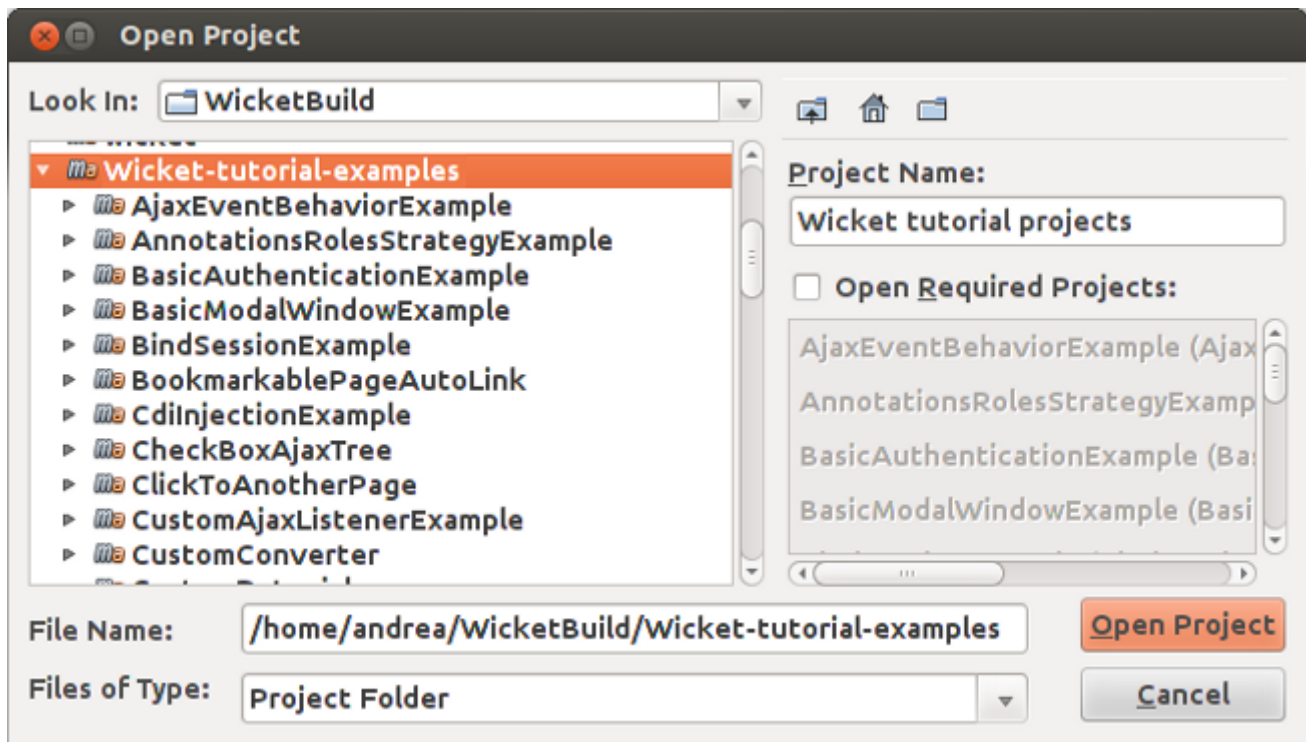
```

If we need to use more Wicket modules or additional libraries, we can add the appropriate XML fragments here.

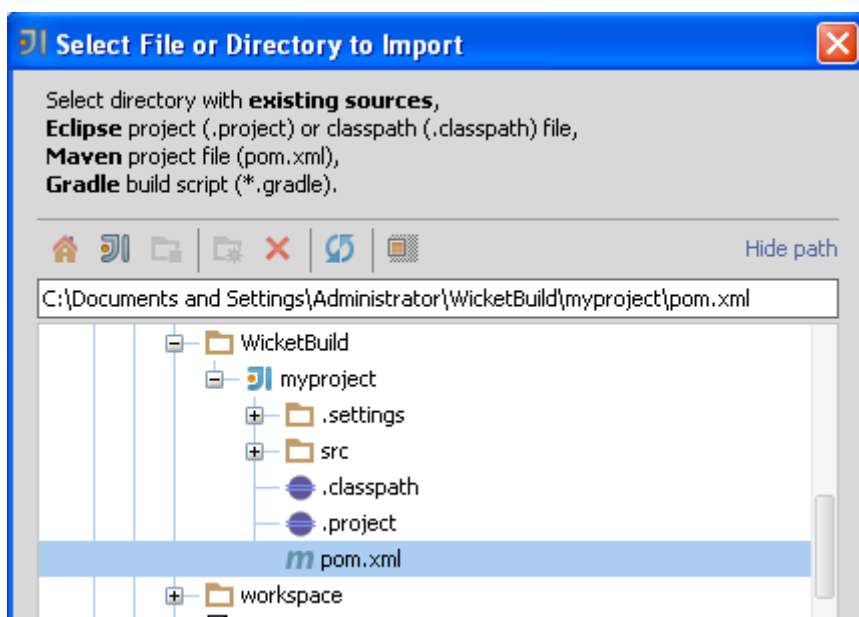
A.2.2. Importing a Maven project into our IDE

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:

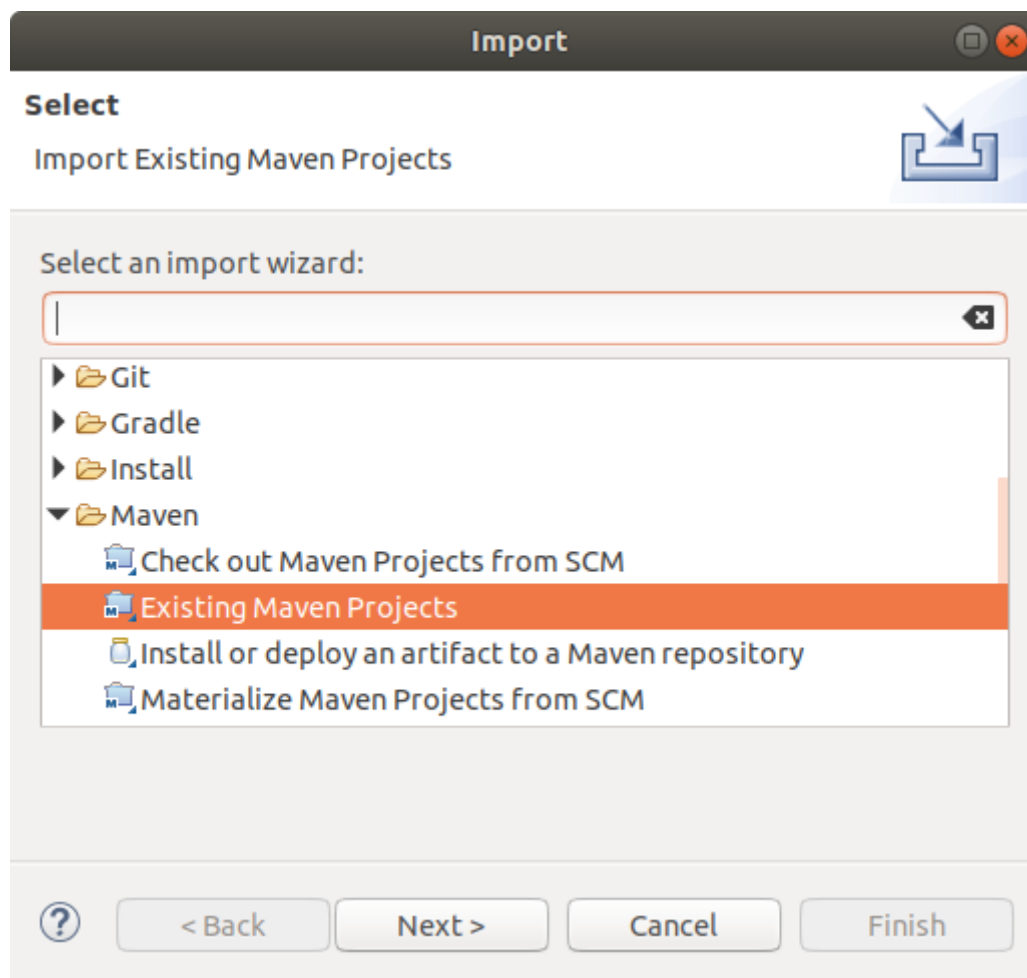


IntelliJ IDEA IntelliJ IDEA comes with a Maven importing functionality that can be started under “File/New Project/Import from external model/Maven”. Then, we just have to select the pom.xml file of our project:

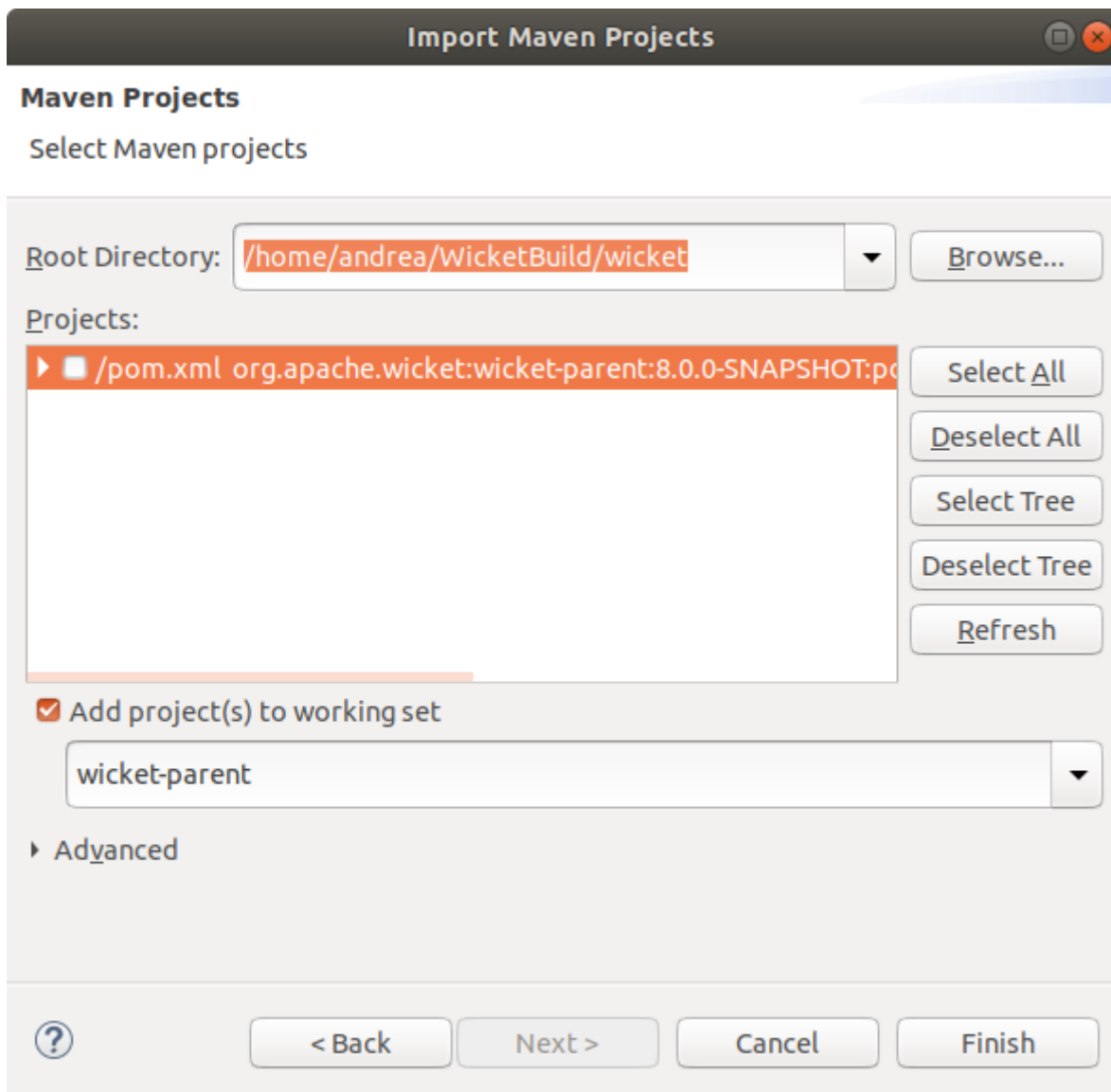


Eclipse Just like the other IDEs Eclipse supports Maven projects out of the box. Open the

“File/Import...” dialog and search for Maven:



then, select the project folder containing the POM file:



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](#)).

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 through 'NetBeans Plugin for Wicket' hosted at <http://plugins.netbeans.org/plugin/3586/wicket-1-4-support> . 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> .

IntelliJ IDEA For JetBrains 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 <https://github.com/count-negative/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
```

In order to enjoy extra components, utilities and/or functionality introduced by WicketStuff modules in our Wicket projects, we can import the respective module dependency in our **pom.xml** as shown below:

```
<dependency>
  <groupId>org.wicketstuff</groupId>
  <artifactId>wicketstuff-<module name></artifactId>
  <version><wicketstuff version></version>
</dependency>
```

where *<wicketstuff version>* is the version of WicketStuff artifact (e.g. 8.0.0-SNAPSHOT), and *<module name>* corresponds to the name of WicketStuff module we want to use. As an illustration, to have access to Java 8 lambda style *ComponentFactory* methods for adding *Links* or *AjaxButtons* to our pages, the following dependency declaration will suffice:

```
<dependency>
  <groupId>org.wicketstuff</groupId>
  <artifactId>wicketstuff-lambda-components</artifactId>
  <version>8.0.0-SNAPSHOT</version>
</dependency>
```

Please refer to [Appendix B.7](#) for more details about Lambda Components.

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. 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](#)) and JavaScript libraries (like [TinyMCE](#)).

This appendix provides a quick overview of what WicketStuff offers to enhance the usability and

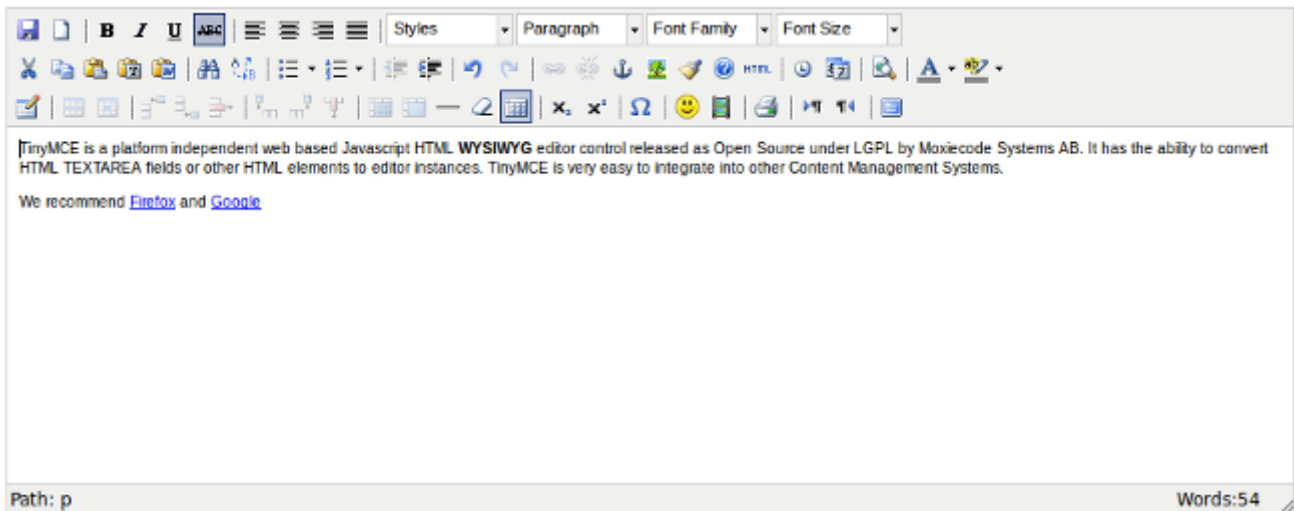
the visually-appealing of our pages.



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.

B.2. Module tinymce

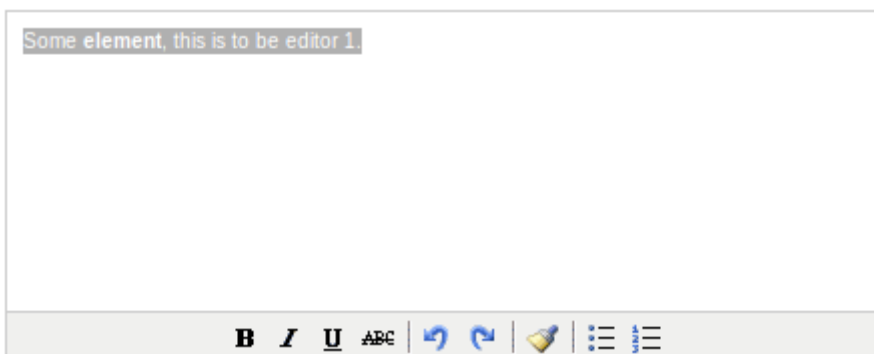
Module tinymce offers integration with the namesake JavaScript library that turns our “humble” text-areas into a full-featured HTML WYSIWYG editor:



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](#) 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`:

HTML:

```
...
<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](#) 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:

HTML:

```

...
<h2>Hello World</h2>
<img wicket:id="helloWorld"/>
...

```

Java code:

```

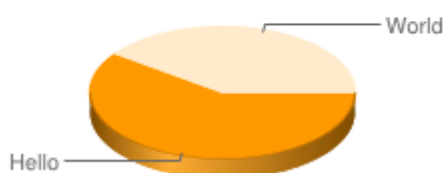
IChartData data = new AbstractChartData(){
    public double[][] getData(){
        return new double[][] { { 34, 22 } };
    }
};

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:

Hello World



As we can see in the snippet above, component Chart must be used with 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 13.4](#)) 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:

<input type="checkbox"/>	ID	First Name	Last Name	Home Phone	Cell Phone	Edit
<input type="checkbox"/>	347	Abby	Gonzalez	710-555-1577	677-555-1601	
<input type="checkbox"/>	360	Abby	Moore	510-555-4672	471-555-7145	
<input type="checkbox"/>	374	Abby	Allen	883-555-5658	328-555-5650	
<input type="checkbox"/>	383	Abby	Murray	876-555-6527	673-555-2368	
<input type="checkbox"/>	389	Abby	Clark	251-555-7726	312-555-7068	
<input type="checkbox"/>	521	Abby	Gomez	326-555-3855	221-555-6578	
<input type="checkbox"/>	572	Abby	Davis	226-555-2267	504-555-1256	
<input type="checkbox"/>	579	Abby	Williams	623-555-5207	736-555-7468	
<input checked="" type="checkbox"/>	580	Abby	Wilson	738-555-8637	524-555-7745	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	616	Abby	Fisher	487-555-3461	265-555-5456	<input checked="" type="checkbox"/>

The following snippet illustrate how to use DataGrid and is taken from wiki page <https://github.com/wicketstuff/core/wiki/InMethodGrid> :

HTML:

```
...  
<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>.

B.6. Module wicketstuff-rest-annotations

REST-based API are becoming more and more popular around the web and the number of services based on this architecture is constantly increasing.

Wicket is well-known for its capability of transparently handling the state of web applications on server side, but it can be also easily adopted to create RESTful services. WicketStuff module for REST provides a special resource class and a set of annotations to implement REST APIs/services in much the same way as we do it with Spring MVC or with the standard JAX-RS.

The module provides class *AbstractRestResource* as generic abstract class to implement a Wicket resource that handles the request and the response using a particular data format (XML, JSON, etc...). Subclassing *AbstractRestResource* we can create custom resources and map their public methods to a given subpath with annotation *MethodMapping*. The following snippet is taken from resource *PersonsRestResource* inside module 'restannotations-examples':

```
@MethodMapping("/persons")
public List<PersonPojo> getAllPersons() {
    //method mapped at subpath "/persons" and HTTP method GET
}

@MethodMapping(value = "/persons/{personIndex}", httpMethod = HttpMethod.DELETE)
public void deletePerson(int personIndex) {
    //method mapped at subpath "/persons/{personIndex}" and HTTP method DELETE.
    //Segment {personIndex} will contain an integer value as index.
}
```

```

@MethodMapping(value = "/persons", httpMethod = HttpMethod.POST)
public void createPerson(@RequestBody PersonPojo personPojo) {
    //creates a new instance of PersonPojo reading it from request body
}

```

MethodMapping requires to specify the subpath we want to map the method to. In addition we can specify also the HTTP method that must be used to invoke the method via REST (GET, POST, DELETE, PATCH, etc...). This value can be specified with enum class *HttpMethod* and is GET by default. In the code above we can see annotation *RequestBody* which is used to extract the value of a method parameter from the request body (method *createPerson*). To write/read objects to response/from request, *AbstractRestResource* uses an implementation of interface *IWebSerialDeserial* which defines the following methods:

```

public interface IWebSerialDeserial {

    public void objectToResponse(Object targetObject, WebResponse response, String
    mimeType) throws Exception;

    public <T> T requestToObject(WebRequest request, Class<T> argClass, String
    mimeType) throws Exception;

    public boolean isMimeTypeSupported(String mimeType);
}

```

To convert segments value (which are strings) to parameters type, *AbstractRestResource* uses the standard Wicket mechanism based on the application converter locator:

```

//return the converter for type clazz
IConverter converter =
Application.get().getConverterLocator().getConverter(clazz);
//convert string to object
return converter.convertToObject(value, Session.get().getLocale());

```

In order to promote the principle of convention over configuration, we don't need to use any annotation to map method parameters to path parameters if they are declared in the same order. If we need to manually bind method parameters to path parameters we can use annotation *PathParam*.

```

@MethodMapping(value = "/variable/{p1}/order/{p2}", produces =
RestMimeTypes.PLAIN_TEXT)
public String testParamOutOfOrder(@PathParam("p2") String textParam,
@PathParam("p1") int intParam) {
    //method parameter textParam is taken from path param 'p2', while intParam
    uses 'p1'
}

```

As JSON is de-facto standard format for REST API, the project comes also with a ready-to-use resource (*GsonRestResource*) and a serial/deserial (*GsonSerialDeserial*) that work with JSON format (both inside module *'restannotations-json'*). These classes use Gson as JSON library.

AbstractRestResource supports role-based authorizations for mapped method with annotation *AuthorizeInvocation*:

```
@RequestMapping(value = "/admin", httpMethod = HttpMethod.GET)
@AuthorizeInvocation("ROLE_ADMIN")
public void testMethodAdminAuth() {

}
```

To use annotation *AuthorizeInvocation* we must specify in the resource constructor an instance of Wicket interface *IRoleCheckingStrategy*.

To read the complete documentation of the module and to discover more advanced feature please refer to the [project homepage](#)

B.7. Module wicketstuff-lambda-components

This module comes with class *org.wicketstuff.lambda.components.ComponentFactory* which exposes a number of factory method to build components using Lambda expressions as event handler. This can be useful to create components with simple behavior. For example:

```
//create a standard link component
add(ComponentFactory.link("id", (link) -> { /*do stuff*/ }));

//create an AJAX link component
add(ComponentFactory.ajaxLink("id", (ajaxLink, ajaxTarget) -> { /*do stuff*/ }));
```

The factory uses library [jdk-serializable-functional](#) to convert lambda expressions into a serializable version of *java.util.function.** interfaces.

AjaxButton and *AjaxSubmitLink* are also supported:

```
//create a submit link
add(ComponentFactory.ajaxSubmitLink("id", (ajaxLink, ajaxTarget) -> { /*do submit stuff*/ }));

//create a submit link with error handler
add(ComponentFactory.ajaxSubmitLink("id", (ajaxLink, ajaxTarget) -> { /*do submit stuff*/ },
                                     (ajaxLink, ajaxTarget) -> { /*do error stuff*/ }));
```

See *ComponentFactory* JavaDoc for a full list of factory methods.

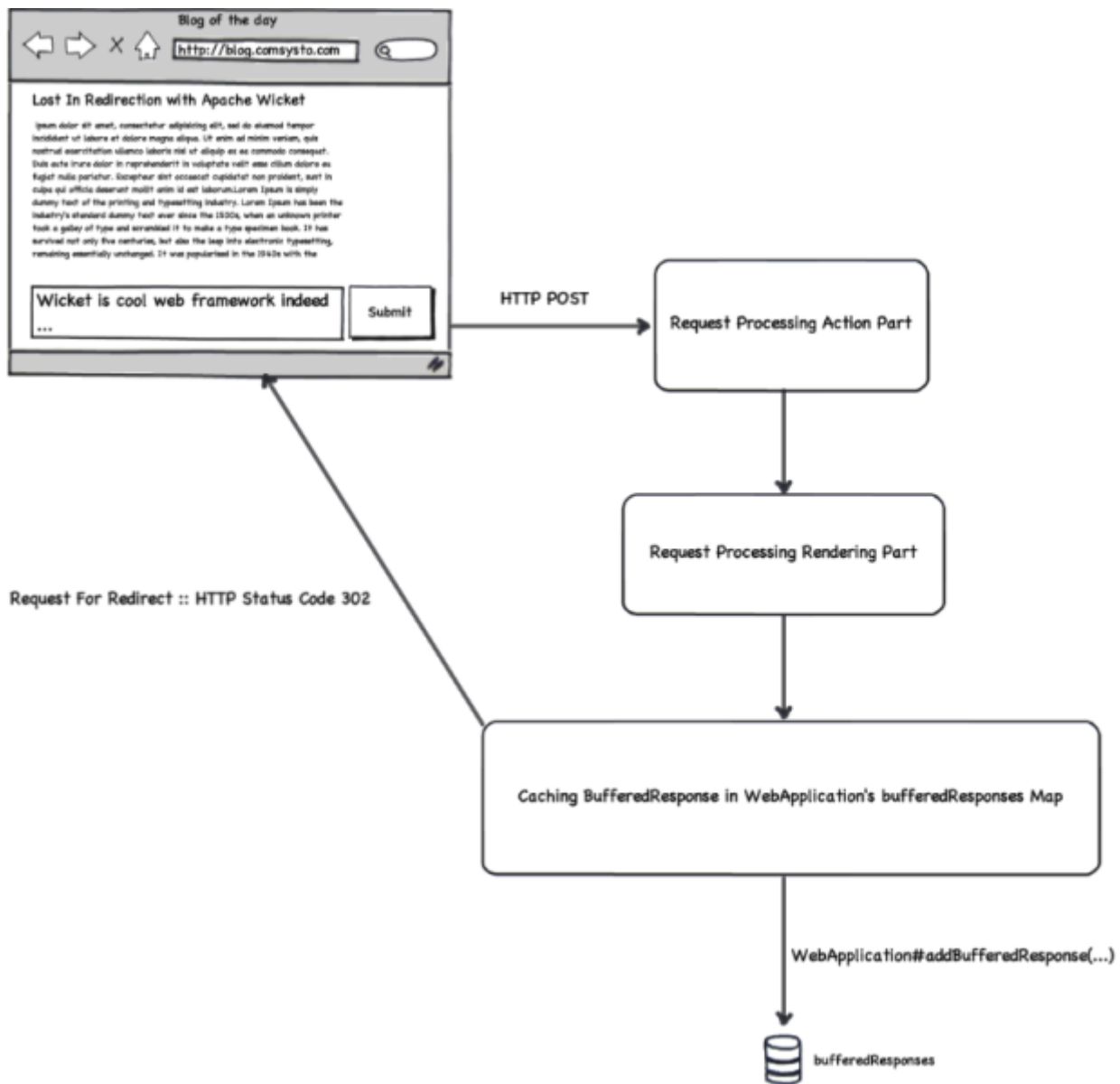
Appendix C: Lost In Redirection With Apache Wicket

Quite a few teams have already got stuck into the following problem when working with wicket forms in a clustered environment while having 2 (or more) tomcat server with enabled session replication running.

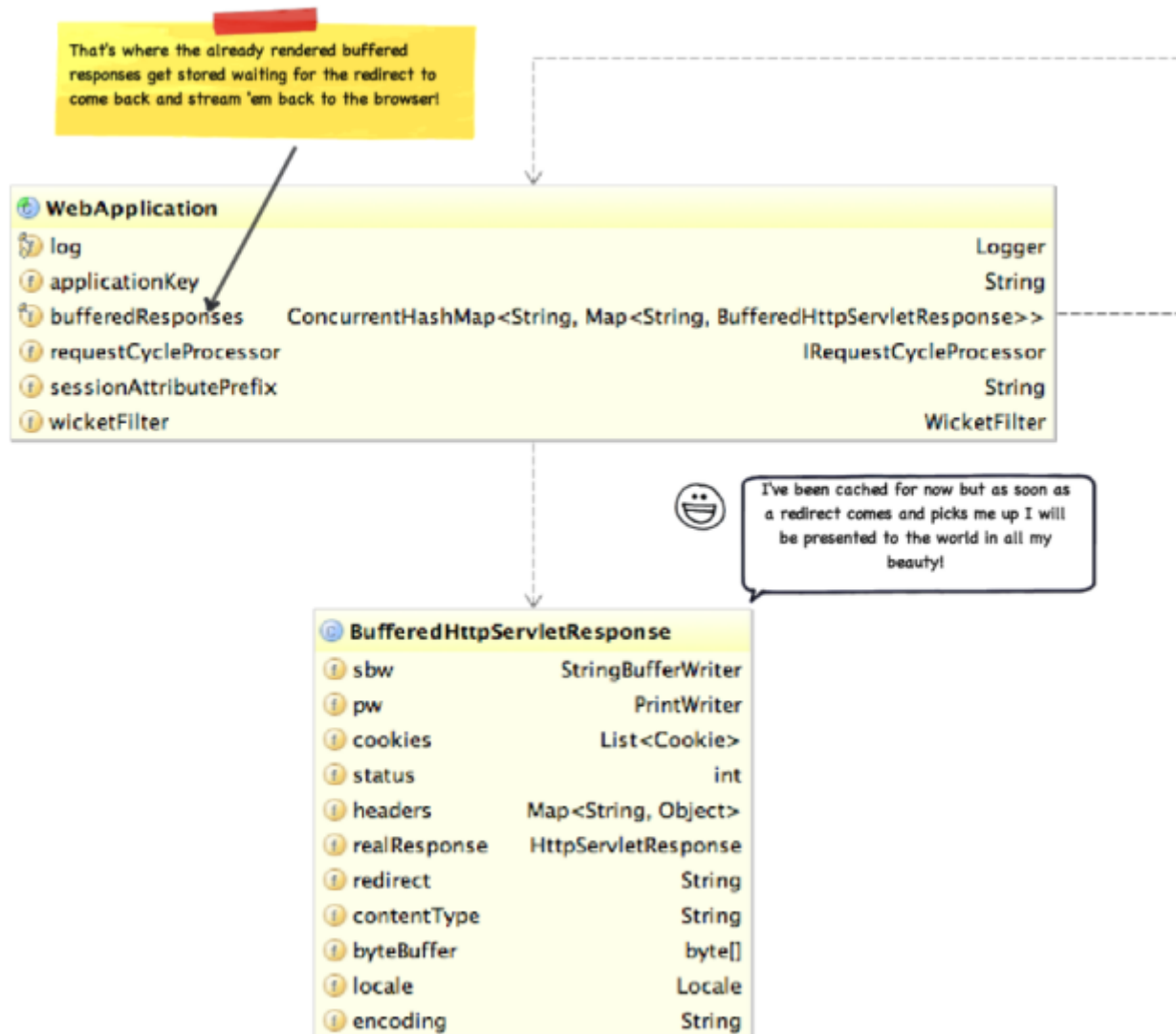
In case of invalid data being submitted with a form instance for example, it seemed like according error messages wouldn't be presented when the same form page gets displayed again. Sometimes! And sometimes they would! One of those nightmares of rather deterministic programmer's life. This so called Lost In Redirection problem, even if it looks like a wicket bug at first, is rather a result of a default setting in wicket regarding the processing of form submissions in general. In order to prevent another wide known problem of double form submissions, Wicket uses a so called REDIRECT_TO_BUFFER strategy for dealing with rendering a page after web form's processing (see *RequestCycleSettings.RenderStrategy*).

What does the default RenderStrategy actually do?

Both logical parts of a single HTTP request, an action and a render part get processed within the same request, but instead of streaming the render result to the browser directly, the result is cached on the server first.



Wicket will create an according `BufferedHttpServletResponse` instance that will be used to cache the resulting `HttpServletResponse` within the `WebApplication`.



After the buffered response is cached the HTTP status code of 302 gets provided back to the browser resulting in an additional GET request to the redirect URL (which Wicket sets to the URL of the Form itself). There is a special handling code for this case in the WicketFilter instance that then looks up a Map of buffered responses within the WebApplication accordingly. If an appropriate already cached response for the current request is found, it gets streamed back to the browser immediately. No additional form processing happens now. The following is a code snippet taken from WicketFilter:

```

// Are we using REDIRECT_TO_BUFFER?
if (webApplication.getRequestCycleSettings().getRenderStrategy() ==
RequestCycleSettings.REDIRECT_TO_BUFFER)
{
    // Try to see if there is a redirect stored
    // try get an existing session
    ISessionStore sessionStore = webApplication.getSessionStore();
    String sessionId = sessionStore.getSessionId(request, false);
    if (sessionId != null)
    {
        BufferedHttpServletResponse bufferedResponse = null;
        String queryString = servletRequest.getQueryString();
    }
}

```

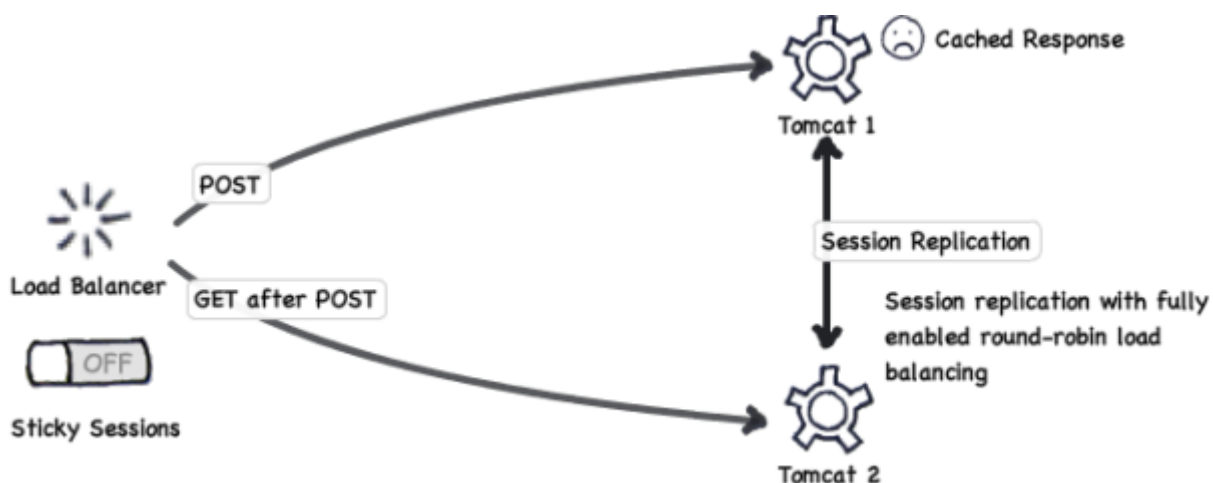
```

// look for buffered response
if (!Strings.isEmpty(queryString))
{
    bufferedResponse = webApplication.popBufferedResponse(sessionId,
        queryString);
}
else
{
    bufferedResponse = webApplication.popBufferedResponse(sessionId,
        relativePath);
}
// if a buffered response was found
if (bufferedResponse != null)
{
    bufferedResponse.writeTo(servletResponse);
    // redirect responses are ignored for the request
    // logger...
    return true;
}
}
}

```

So what happens in case you have 2 server running your application with session replication and load balancing turned on while using the default RenderStrategy described above?

Since a Map of buffered responses is cached within a WebApplication instance that does not get replicated between the nodes obviously, a redirect request that is suppose to pick up the previously cached response (having possibly form violation messages inside) potentially get's directed to the second node in your cluster by the load balancer. The second node does not have any responses already prepared and cached for your user. The node therefore handles the request as a completely new request for the same form page and displays a fresh new form page instance to the user accordingly.



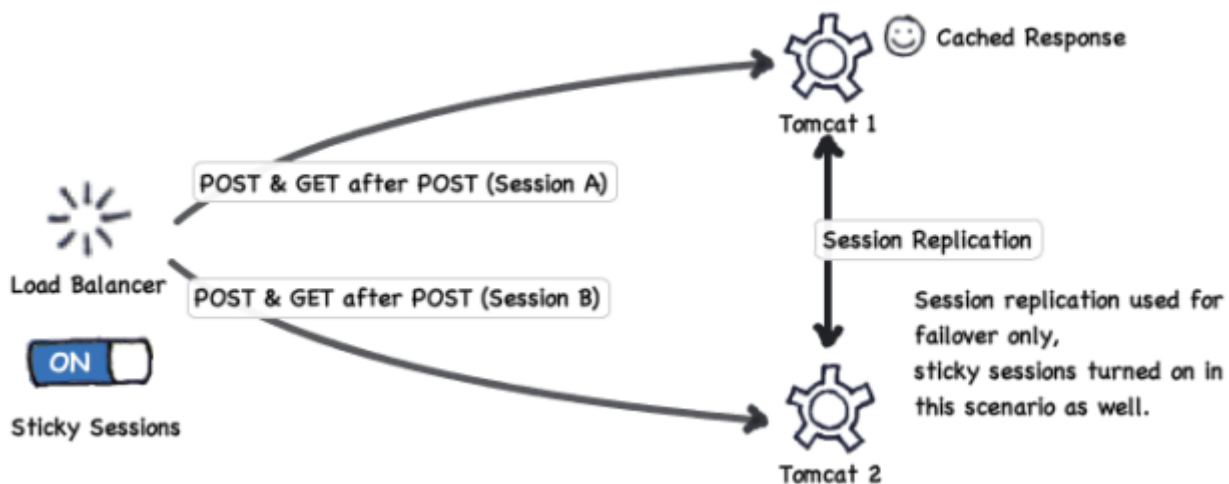
Unfortunately, there is currently no ideal solution to the problem described above. The default RenderStrategy used by Apache Wicket simply does not work well in a fully clustered environment with load balancing and session replication turned on. One possibility is to change the default

render strategy for your application to a so called ONE_PASS_RENDER RenderStrategy which is the more suitable option to use when you want to do sophisticated (non-sticky session) clustering. This is easily done in the init method of your own subclass of Wicket's WebApplication :

```
@Override
protected void init() {
    getRequestCycleSettings().setRenderStrategy(
        RequestCycleSettings.ONE_PASS_RENDER);
}
```

ONE_PASS_RENDER RenderStrategy does not solve the double submit problem though! So this way you'd only be trading one problem for another one actually.

You could of course turn on the session stickiness between your load balancer (apache server) and your tomcat server additionally to the session replication which would be the preferred solution in my opinion.



Session replication would still provide you with failover in case one of the tomcat server dies for whatever reason and sticky sessions would ensure that the Lost In Redirection problem does not occur any more.

Appendix D: Working with Karaf

D.1. Wicket feature

Wicket jar files are packaged as OSGi bundles and are ready to be installed in OSGi environments, such as Apache Karaf.

Wicket provides a Karaf feature file to assist in installation and building Karaf distributions that include Wicket.

D.1.1. Add Wicket feature

```
karaf@root(>) repo-add mvn:org.apache.wicket/wicket/version/xml/features
Adding feature url mvn:org.apache.wicket/wicket/version/xml/features
```

D.1.2. Verify Wicket feature

```
karaf@root(>) feature:list | grep -i wicket
wicket-core      | version |      | Uninstalled | org.apache.wicket.wicket-version |
```

D.1.3. Install Wicket feature

```
karaf@root(>) feature:install wicket-core
```

D.1.4. Troubleshooting

Inspect the wicket-core feature

```
karaf@root(>) feature:info wicket-core
Feature wicket-core version
Feature has no configuration
Feature has no configuration files
Feature has no dependencies.
Feature contains followed bundles:
... wicket dependency bundles ...
mvn:org.apache.wicket/wicket-util/version
mvn:org.apache.wicket/wicket-request/version
mvn:org.apache.wicket/wicket-core/version
mvn:org.apache.wicket/wicket-auth-roles/version
mvn:org.apache.wicket/wicket-devutils/version
mvn:org.apache.wicket/wicket-extensions/version
mvn:org.apache.wicket/wicket-jmx/version
Feature has no conditionals.
```

Wicket feature Maven coordinates

The Wicket feature may be referenced from Maven to include in Karaf assemblies.

```
<dependency>
  <groupId>org.apache.wicket</groupId>
  <artifactId>wicket</artifactId>
  <version>...version...</version>
  <type>xml</type>
  <classifier>features</classifier>
</dependency>
```

D.1.5. Reference links

[Apache Karaf website](#)

[Apache Karaf features](#)

Appendix E: Contributing to this guide

You can contribute to this guide by following these steps:

- The guide uses AsciiDoctor to generate the final HTML/PDF so you should consult with its [syntax](#)
- Fork Apache Wicket's GIT [repository](#) to your own github account
- Clone your forked copy of Apache Wicket's repository into your machine

```
git clone https://github.com/<<your github username>>/wicket.git
```

- Edit the *.adoc* files in `wicket/wicket-user-guide/src/main/asciidoc` folder
- To preview your changes run `mvn clean package -P guide` in the `wicket/wicket-user-guide` folder (You may use a run configuration in your IDE)
- Navigate to `wicket/wicket-user-guide/target/generated-docs` and open one of the following files in a browser / pdf viewer:
 - *single.html* (single page version)
 - *single.pdf* (single page pdf version)
- Create a ticket in Apache Wicket's [JIRA](#)
- **Commit and push the changes** to your forked Apache Wicket's GIT repository and **create a pull request** on github (Enter the created JIRA ticket id into your pull request's title)

Thank you!