DZone Refcardz

CONTENTS INCLUDE:

Getting Started with Apache Wicket

By Andrew Lombardi

ABOUT APACHE WICKET

Apache Wicket is a Java-based web application framework that has rapidly grown to be a favorite among many developers. It features a POJO data model, no XML, and a proper mark-up / logic separation not seen in most frameworks. Apache Wicket gives you a simple framework for creating powerful, reusable components and offers an object oriented methodology to web development while requiring only Java and HTML. This Refcard covers Apache Wicket 1.3 and describes common configuration, models, the standard components, implementation of a form, the markup and internationalization options available.

PROJECT LAYOUT

The project layout most typical of Apache Wicket applications is based on the default Maven directories. Any Wicket component that requires view markup in the form of HTML needs to be side-by-side with the Java file. Using Maven however, we can separate the source directories into java/ and resources/ to give some distinction. To get started, download either the wicket-quickstart project and modify it to your needs, or use the maven archetype here:

```
mvn archetype:create \
-DarchetypeGroupId=org.apache.wicket \
-DarchetypeArtifactId=wicket-archetype-quickstart \
-DarchetypeVersion=1.3.5 \
-DgroupId=com.mysticcoders.refcardmaker \
-DartifactId=refcardmaker
```

Either way, if using Maven, you'll need the wicket jar, and the latest slf4j jar.

```
<dependency>
    <groupId>org.apache.wicket</groupId>
    <artifactId>wicket</artifactId>
    <version>1.3.6</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.4.2</version>
</dependency>
```

CONFIGURING THE WEB APPLICATION

I mentioned that Wicket has no XML, and that's mostly true, but J2EE requires a web.xml file to do anything. We set up the WicketFilter and point it to our implementation of WebApplication along with the URL mapping.

```
<web-app>
    <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.mysticcoders.refcardmaker.
RefcardApplication</param-value>
        </init-param>
        <init-param>
            <param-name>filterPath</param-name>
            <param-value>/*</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>wicketFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

Apache Wicket offers a development and deployment mode that can be configured in the web.xml file:

```
<context-param>
    <param-name>configuration</param-name>
    <param-value>development</param-value>
</context-param>
```

Hot Tip

Depending on your configuration needs, you can set this parameter in the web.xml as either:
- a context-param or init-param to the filter
- a command line parameter wicket.configuration
- by overriding Application.getConfigurationType()

MODELS

Apache Wicket uses models to separate the domain layer from the view layer in your application and to bind them together. Components can retrieve data from their model, and convert and store data in the model upon receiving an event. There are a variety of implementations of Model, and they all extend from the interface IModel.

IModel
There are only two methods that a class would have to implement to be a Model, and that is getObject and setObject. getObject returns the value from the model, and setObject sets the value of the model. Your particular implementation of IModel can get data from wherever you'd like; the Component in the end only requires the ability to get and set the value. Every component in Wicket has a Model: some use it, some don't, but it's always there.

PropertyModel
A model contains a domain object, and it's common practice to follow JavaBean conventions. The PropertyModel allows you to use a property expression to access a property in your domain object. For instance if you had a model containing Person and it had a getter/setter for firstName to access this property, you would pass the String "firstName" to that Model.

CompoundPropertyModel
An even fancier way of using models is the CompoundPropertyModel. Since most of the time, the property identifier you would give Wicket mimics that of the JavaBean property, this Model takes that implied association and makes it work for you.

```
...
setModel(new CompoundPropertyModel(person));
add(new Label("firstName"));
add(new Label("lastName"));
add(new Label("address.address1"));
...
```

We can see from the example above, that if we set the model with the person object using a CompoundPropertyModel, the corresponding components added to this parent Component will use the component identifiers as the property expression.

IDetachable
In order to keep domain objects around, you're either going to need a lot of memory / disk space, or devise a method to minimize what gets serialized in session. The detachable design helps you do this with minimal effort. Simply store as little as needed to reconstruct the domain object, and within the detach method that your Model overrides, null out the rest.

LoadableDetachableModel
In order to make things easier, LoadableDetachableModel implements a very common use case for detachable models. It gives you the ability to override a few constructors and a load method, and provides automatic reattach and detach within Wicket's lifecycle. Let's look at an example:

```
public class LoadableRefcardModel extends LoadableDetachableModel
{
    private Long id;
    public LoadableRefcardModel(Refcard refcard) {
        super(refcard);
        id = refcard.getId();
    }
    public LoadableRefcardModel(Long id) {
        super();
        this.id = id;
    }
    protected Object load() {
        if(id == null) return new Refcard();
        RefcardDao dao = ...
        return dao.get(id);
    }
}
```

Here we have two constructors, each grabbing the identifier and storing it with the Model. We also override the load method so that we can either return a newly created Object, or use an access object to return the Object associated with the Model's stored identifier. LoadableDetachableModel handles the process of attaching and detaching the Object properly giving us as little overhead as possible.

COMPONENTS

In Wicket, Components display data and can react to events from the end user. In terms of the Model-View-Controller pattern, a Component is the View and the Controller. The following three distinct items make up every Wicket Component in some way:

- **Java Class implementation** – defines the behavior and responsibilities
- **HTML Markup** – defines the Components using their identifiers within view markup to determine where it shows to the end user
- **The Model** – provides data to the Component

Now that we have an idea about what makes up a Component, let's look at a few of the building blocks that make up the majority of our Pages. Forms and their Components are so important they have their own section.

Label
When developing your application, if you'd like to show text on the frontend chances are pretty good that you'll be using the Label Component. A Label contains a Model object which it will convert to a String for display on the frontend.

```
<span wicket:id="message">[message]</span>

...

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

The first portion is an HTML template, which gives a component identifier of "message" which must be matched in the Java code. The Java code passes the component identifier as the first parameter.

Link
Below is a list of the different types of links, bookmarkable and non-bookmarkable, and how they are used to navigate from page-to-page.

Name	Description

`Link`	If linking to another Page, it is best to use a Link in most instances:
	`add(new Link("myLink") {`
	`    public void onClick() {`
	`        setResponsePage(MyNewPage.class);`
	`    }`
	`}`

`BookmarkablePageLink`	A Bookmarkable page gives you a human readable URL that can be linked to directly from outside of the application. The default look of this URL can be overridden as we'll see in the next section. `add(new BookmarkablePageLink("myLink", MyNewPage.class);`
`ExternalLink`	If linking to an external website that is not within your application, here's the Link component you'll need and an example: `add(new ExternalLink("myLink", http://www.mysticcoders.com, "Mystic");`

Repeaters
Due to the lack of any executable code inside of Wicket's HTML templates, the method of showing lists may seem a

little counterintuitive at first. One of the simplest methods for showing a list of data is the RepeatingView. Here's an example of how to use it:

```
<ul>
    <li wicket:id="list"></li>
</ul>

...

RepeatingView list = new RepeatingView("list");
add(list);
for(int i = 1; i <= 10; i++) {
    list.add(new Label(list.newChildId(), "Item " + i));
}
```

This will simply print out a list from 1 to 10 into HTML. RepeatingView provides a method .newChildId() which should be used to ensure the Component identifier is unique. As your needs get more complex, this method quickly turns stale as there is a lot of setup that has to be done. Using a ListView is a great approach for managing possibly complex markup and business logic, and is more akin to other ways we're asked to interact with Apache Wicket:

```
<ul>
    <li wicket:id="list"><span wicket:id="description">[descripti
on]</span></li>
</ul>
...
ListView list = new ListView("list", Arrays.asList("1", "2", "3",
"4", "5", "6", "7", "8", "9", "10") {
    @Override
    protected void populateItem(ListItem item) {
        String text = (String)item.getModelObject();
        item.add(new Label("description", text));
    }
};
add(list);
```

This method, while it looks more complex, allows us a lot more flexibility in building our lists to show to the user. The two list approaches described above each suffer from some drawbacks, one of which is that the entirety of the list must be held in memory. This doesn't work well for large data sets, so if you need finer grain control on how much data is kept in memory, paging, etc., DataTable or DataView is something to look into.

Custom

The beauty of Wicket is that reuse is as simple as putting together a Panel of Components and adding it to any number of pages – this could be a login module, a cart, or whatever you think needs to be reused. For more great examples of reusable components check out the wicket-extensions (http://cwiki.apache.org/Wicket/wicket-extensions.html) and wicket-stuff (http:..wicketstuff.org) projects.

Hot Tip Since Wicket always needs a tag to bind to, even for a label, a <span> tag is sometimes easier to place into your markup; however, this can throw your CSS design off. .setRenderBodyOnly(true) can be used so the span never shows on the frontend but be careful using this with any AJAX enabled components, since it requires the tag to stick around.

PAGE AND NAVIGATION

A Wicket Page is a component that allows you to group components together that make up your view. All Components will be related in a tree hierarchy to your page, and if the page is bookmarkable you can navigate directly to it. To create a new page, simply extend WebPage and start adding components.

Most webapps will share common areas that you don't want to duplicate on every page -- this is where markup inheritance comes into play. Because every page is just a component, you can extend from a base page and inherit things like header, navigation, footer, whatever fits your requirements. Here's an example:

```
public class BasePage extends WebPage {
   ... header, footer, navigation, etc ...
}
public class HomePage extends BasePage {
... everything else, the content of your pages...
}
```

Everything is done similarly to how you would do it in Java, without the need for messy configuration files. If we need to offer up pages that can be referenced and copied, we're going to need to utilize bookmarkable pages. The default Wicket implementation of a BookmarkablePage is not exactly easy to memorize, so in your custom Application class you can define several mount points for your pages:

```
// when a user goes to /about they will get directly to this page
mountBookmarkablePage("/about", AboutPage.class);

// this mount makes page available at /blog/param0/param1/param2
and fills PageParameters with 0-indexed numbers as the key
mount(new IndexedParamUrlCodingStrategy("/blog", BlogPage.class);

// this mount makes page available at /blog?paramKey=paramValue&pa
ramKey2=paramValue2
mount(new QueryStringUrlCodingStrategy("/blog", BlogPage.class);
```

In your code, you'll need several ways of navigating to pages, including within Link implementations, in Form onSubmits, and for an innumerable number of reasons. Here are a few of the more useful:

```
// Redirect to MyPage.class
setResponsePage(MyPage.class);

// Useful to immediately interrupt request processing to perform a
redirect
throw new RestartResponseException(MyPage.class);

// Redirect to an interim page such as a login page, keep the URL
in memory so page can call continueToOriginalDestination()
redirectToInterceptPage(LoginPage.class);

// Useful to immediately interrupt request processing to perform a
redirectToInterceptPage call
throw new RestartResponseAtInterceptPageException(MyPage.class);
```

MARKUP

Apache Wicket does require adding some attributes and tags to otherwise pristine X/HTML pages to achieve binding with Component code. The following table illustrates the attributes available to use in your X/HTML templates, the most important and often used being wicket:id.

Attribute Name	Description
`wicket:id`	Used on any X/HTML element you want to bind a compoent to
`wicket:message`	Used on any tag we want to fill an attribute with a resource bundle value. To use, prefix with te [attribute name]:[resource name]

The following table lists out all of the most commonly used tags in X/HTML templates with Wicket.

Tag Name	Description
`wicket:panel`	This tag is used in your template to define the area associatedf with the component. Anything outside of this tag's hierarchy will be ignored. It is sometimes useful to wrap each of your templates with html and body tags like so: `<html xmlns:wicket="http://wicket.apache.org">` `<body>` `<wicket:panel> ... </wicket:panel>` `</body>` `</html>` In this example, you can avoid errors showing in your IDE, and it won't affect the resulting HTML.
`wicket:child`	Used in conjunction with markup inheritance. The subclassing Page will replace the tag with the output of its component
`wicket:extend`	Defining a page that inherits from a parent Page requires a mirroring of the relationship in your X/HTML template. As with wicket:panel, everything outside of the tag's hierarchy will be ignored, and the component's result will end up in the wrapping template
`wicket:link`	Using this tag enables autolinking to another page without having to add BookmarkablePageLink's to the component hierarchy as this is done automatically for you. To link to the homepage from one of its subpages: `<wicket:link><a href="Homepage.html">Homepage</a></wicket:link>`
`wicket:head`	Adding this to the root-level hierarchy of the template will give you access to inject code into the X/HTML <head></head> section.
`wicket:message`	This tag will look for the given key in the resource bundle component hierarchy and replace the tag with the String retrieved from that bundle property. To pull the resource property page.label: `<wicket:message key="page.label">[page label]</wicket:message>`
`wicket:remove`	The entire contents of this tag will be removed upon running this code in the container. Its use is to ensure that the template can show design intentions such as repeated content without interfering with the resulting markup.
`wicket:fragment`	A fragment is an inline Panel. Using a Panel requires a separate markup file, and with a fragment this block can be contained within the parent component.
`wicket:enclosure`	A convenience tag added in 1.3 that defines a block of code surrounding your component which derives its entire visibility from the enclosing component. This is useful in situations when showing multiple fields some of which may be empty or null where you don't want to add WebMarkupContainers to every field just to mimic this behavior. For example if we were printing out phone and fax: `<wicket:enclosure>` `<tr><td class="label">Fax:</td><td><span wicket:id="fax">[fax number]</span></td></tr>` `</wicket:enclosure>` `...` `add(new Label("fax") { public boolean isVisible() { return getModelObjectAsString()!=null; } } );`
`wicket:container`	This tag is useful when you don't want to render any tags into the markup because it may cause invalid markup. Consider the following: `<table>` `<wicket:container wicket:id="repeater">` `<tr><td>1</td></tr>` `<tr><td>2</td></tr>` `</wicket:container>` `</table>` In this instance, if we were to add any code in between the table and tr tags, it would be invalid. Wicket:container fixes that.

FORM

A Form in Wicket is a component that takes user input and processes it upon submission. This component is a logical holder of one or more input fields that get processed together.

The Form component, like all others, must be bound to an HTML equivalent, in this case the <form> tag.

```
<form wicket:id="form">
    Name: <input type="text" wicket:id="name" />
    <input type="submit" value="Send" />
</form>
...
Form form = new Form("form") {
   @Override
    protected void onSubmit() {
```

```
        System.out.println("form submit");
    }
};
add(form);
form.add(new TextField("name", new Model("")));
```

Form input controls can each have their own Models attached to them, or can inherit from their parent, the Form. This is usually a good place to use CompoundPropertyModel as it gets rid of a lot of duplicate code. As you can see, each input component should be added to the Form element.

Wicket uses a POST to submit your form, which can be changed by overriding the Form's getMethod and returning Form.METHOD_GET. Wicket also uses a redirect to buffer implementation details of form posts which gets around the form repost popup. The following behavior settings can be changed:

Name	Setting	Description
No redirect	`IRequestCycleSettings.ONE_PASS_RENDER`	Renders the response directly
Redirect to buffer	`IRequestCycleSettings.REDIRECT_BUFFER`	Renders the response directly to a buffer, redirects the browser and prevents reposting the form
Redirect to render	`IRequestCycleSettings.REDIRECT_TO_RENDER`	Redirects the browser directly; renders in a separate request

Components of a Form

The following table lists all the different form components available, and how to use them with Models.

Name	Example
TextField	`<input type="text" wicket:id="firstName" />` `...` `add(new TextField("firstName", new PropertyModel(person, "firstName"));`
TextArea	`<textarea wicket:id="comment"></textarea>` `...` `add(new TextArea("comment", new PropertyModel(feedback, "comment"));`
Button	`<form wicket:id="form">` `<input type="submit" value="Submit" wicket:id="submit" />` `</form>` `...` `Form form = new Form("form") {` `@Override` `protected void onSubmit() {` `System.out.println("onSubmit called");` `}` `};` `add(form);` `form.add(new Button("submit"));`
CheckBoxMultipleChoice	`<span wicket:id="operatingSystems">` `<input type="checkbox" /> Windows<br />` `<input type="checkbox" /> OS/2 Warp<br />` `</span>` `...` `add(new CheckBoxMultipleChoice("operatingSystems", new PropertyModel(system, "operatingSystems"), Arrays.asList("Windows", "OS X", "Linux", "Solaris", "HP/UX", "DOS")));`
DropDownChoice	`<select wicket:id="states">` `<option>[state]</option>` `</select>` `...` `add(new DropDownChoice("states", new PropertyModel(address, "state"), listOfStates));`

`PasswordTextField`	`<input type="password" wicket:id="password" />`
	`...`
	`add(new PasswordTextField("password", new PropertyModel(user, "password"));`

`RadioChoice`	`<span wicket:id="gender">`
	`    <input type="radio" /> Male<br />`
	`    <input type="radio" /> Female</br />`
	`</span>`
	`...`
	`add(new RadioChoice("sex", new PropertyModel(person, "gender"), Arrays.asList("Male", "Female"));`

`SubmitLink`	`<form wicket:id="form">`
	`    <a href="#" wicket:id="submitLink">Submit</a>`
	`</form>`
	`...`
	`form.add(new SubmitLink("submitLink") {`
	`    @Override`
	`    public void onSubmit() {`
	`        System.out.println("submitLink called");`
	`    }`
	`});`

Validation

When dealing with user input, we need to validate it against what we're expecting, and guide the user in the right direction if they stray. Any user input is processed through this flow:

- Check that the required input is supplied
- Convert input values from String to the expected type
- Validate input using registered validators
- Push converted and validated input to models
- Call onSubmit or onError depending on the result

Wicket provides the following set of validators:

Resource Key	Example
`Required`	`textField.setRequired(true)`
`RangeValidator.range`	`numField.add(RangeValidator.range(0,10))`
`MinimumValidator.minimum`	`numField.add(MinimumValidator.minimum(0))`
`MaximumValidator.maximum`	`numField.add(MaximumValidator.maximum(0))`
`StringValidator.exact`	`textField.add(StringValidator.exact(8))`
`StringValidator.range`	`textField.add(StringValidator.range(6, 18))`
`StringValidator.maximum`	`textField.add(StringValidator.maximum(8))`
`StringValidator.minimum`	`textField.add(StringValidator.minimum(2))`
`DateValidator.range`	`dateField.add(DateValidator.range(startDate, endDate))`
`DateValidator.minimum`	`dateField.add(DateValidator.minimum(minDate))`
`DateValidator.maximum`	`dateField.add(DateValidator.maximum(maxDate))`
`CreditCardValidator`	`ccField.add(new CreditCardValidator())`
`PatternValidator`	`textFIeld.add(new PatternValidator("\d+")`
`EmailAddressValidator`	`emailField.add(EmailAddressValidator.getInstance())`
`UrlValidator`	`urlField.add(new UrlValidator())`
`EqualInputValidator`	`add(new EqualInputValidator(formComp1, formComp2))`
`EqualPasswordInputValidator`	`Add(new EqualPasswordInputValidator(passFld1, passFld2))`

More than one validator can be added to a component if needed. For instance, if you have a password that needs to be within the range of 6 – 20 characters, must be alphanumeric and is required, simply chain the needed validators above to your component. If the validators listed above don't fit your needs, Wicket lets you create your own and apply them to your components.

```
public class PostalCodeValidator extends AbstractValidator {
    public PostalCodeValidator() {
    }

    @Override
    protected void onValidate(IValidatable validatable) {
        String value = (String)validatable.getValue();
        if(!postalCodeService.isValid(value)) {
            error(validatable);
        }
    }
    @Override
    protected String resourceKey() {
        return "PostalCodeValidator";
    }
    @Override
    protected Map variablesMap(IValidatable validatable) {
        Map map = super.variablesMap(validatable);
        map.put("postalCode", n);
        return map;
    }
}
```

When Wicket has completed processing all input it will either pass control to the Form's onSubmit, or the Form's onError.. If you don't choose to override onError, you'll need a way to customize the error messages that show up.

Feedback Messages

Apache Wicket offers a facility to send back messages for failed validations or flash messages to provide notification of status after submitting a form or performing some action. Wicket's validators come with a default set of feedback messages in a variety of languages, which you can override in your own properties files. Here's the order Wicket uses to grab messages out of resource bundles:

Location	Order	Description	Example
Page class	1	Messages Specific to a page	`Index.properties` `Index_es.properties`
Component class	2	Messages specific to a component	`AddressPanel_es.properties` `CheckOutForm.properties`

Custom Application class	3	Default application-wide message bundle	`RefcardApplication_es_MX.properties`
			`RefcardApplication_es.properties`
			`RefcardApplication.properties`

During a Form submission, if you'd like to pass back messages to the end user, Wicket has a message queue that you can access with any component:

```
info("Info message");
warn("Warn message");
error("Error message");
```

With that added to the queue, the most basic method of showing these to users is to use the FeedbackPanel component which you can add to your Page as follows:

```
<div wicket:id="feedback"></div>

…

add(new FeedbackPanel("feedback"));
```

When you'd like to get them back out again, it will give you an Iterator to cycle through on the subsequent page:

```
getSession().getFeedbackMessages().iterator();
```

INTERNATIONALIZATION

Earlier sections touched on the order of resource bundles importance from the Page down to Wicket's default application. Apache Wicket uses the same resource bundles standard in the Java platform, including the naming convention, properties file or XML file.

Using ResourceBundles, you can pull out messages in your markup using <wicket:message>, or use a ResourceModel with the component to pull out the localized text.

Another available option is to directly localize the filename of the markup files, i.e. HomePage_es_MX.html, HomePage.html. Your default locale will be used for HomePage.html, and if you were from Mexico, Wicket would dutifully grab HomePage_es_MX.html.

Hot Tip Wicket's Label component overrides the getModelObjectAsString of Component to offer you Localaware String's output to the client, so you don't have to create your own custom converter.

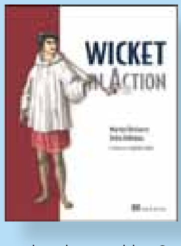
Resources	
Wicket 1.3 Homepage	http://wicket.apache.org/
Component Reference	http://wicketstuff.org/wicket13/compref/
Wicket Wiki	http://cwiki.apache.org/WICKET/
Wicket by Example	http://wicketbyexample.com/

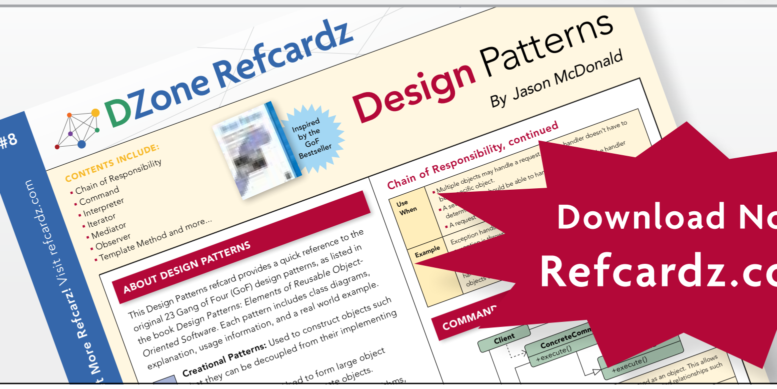
ABOUT THE AUTHOR

Andrew Lombardi is one of a new breed of business-men: the enlightened entrepreneur. He has been writing code since he was a 5-year old, sitting at his dad's knee at their Apple II computer. Having such a deep affinity for the computer model, it is no surprise that at the age of 17 he began to delve deeply into the inner workings of the human mind. He became a student of Neuro Linguistic Programming and other mind technologies, and then went on to study metaphysics. He is certified as an NLP Trainer, Master Hypnotherapist and Time Line Therapy practitioner.

Using all of his accumulated skills, at the age of 24, Andrew began his consulting business, Mystic Coders, LLC. Since the inception of Mystic in 2000, Andrew has been building the business and studying finance and economics as he stays on the cutting edge of computer technology.

RECOMMENDED BOOK





DZone, Inc.
1251 NW Maynard
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-66-0
ISBN-10: 1-934238-66-X
50795
9 781934 238660
$7.95

Version 1.0