

# Wicket: A simplified framework for building and testing dynamic Web pages

Skill Level: Intermediate

[Kumarsun M. Nadar \(kumarsun.nadar@in.ibm.com\)](mailto:kumarsun.nadar@in.ibm.com)  
Senior Staff Software Engineer  
IBM

04 Nov 2008

Wicket provides an object-oriented approach toward developing dynamic Web-based UI applications. Because Wicket is pure Java™ and HTML code, you can leverage your knowledge about Java to write applications based on Wicket, dramatically reducing your development time. This article gives you an overview of Wicket and describes how you can use Wicket to rapidly build Web-based applications in a non-intrusive and simplified way.

## Overview

Wicket is a recently launched Java Web development framework. It's an open source, lightweight, component-based framework, which puts it in an altogether different league from some of the earlier ways of developing Web-based applications. Wicket strives for a clean separation between the roles of HTML page designer and Java developer by supporting plain HTML-based templates that can be built using any WYSIWYG HTML design tools. The templates can then be made dynamic with little modification.

Like other frameworks, Wicket is built on top of Sun Microsystems' servlet API. However, unlike frameworks based on the Model-View-Controller (MVC) model (such as Struts), Wicket takes away from you the task of handling request/response objects, which is inherent with technologies such as servlets. By taking away this task, Wicket allows you to concentrate on the application's business logic.

As a Wicket developer, you should think in terms of building reusable components that are stateful, instead of building controllers that handle request/response objects

and worrying about multithreading issues. Instead of creating a controller or action class, you create a page, place components on it, and define how each component reacts to user input.

## HelloWorld example

To truly demonstrate the simplicity of developing Web-based applications using Wicket, you'll start by developing a simple "Hello World" example. Developing a dynamic page in Wicket generally involves creating the following two artifacts:

- HTML template
- Java page class

### Wicket's Java page class

The Java page class is a Wicket convention, and should not be confused with JavaServer Pages (JSP). In this context, the Java page class is merely the Java code that handles dynamic content for a Web page. In the example, it is the HelloWorld.java that corresponds with HelloWorld.html.

**Note:** You must ensure that the actual HTML file and the page class name are the same (for example, HelloWorld.html and HelloWorld.java) and that both are in the same place on the CLASSPATH. As a best practice, both can be placed in the same directory.

### HTML template (HelloWorld.html)

Listing 1 shows the template file for the HelloWorld example.

#### Listing 1. HelloWorld.html

```
<html>
  <head><script type="text/javascript" ></script></head>
  <body bgcolor="#FFCC00">
    <H1 align="center">
      <span wicket:id="message">Hello World Using Wicket!</span>
    </H1>
  </body>
</html>
```

To make a dynamic Web page, you need to identify the sections of the page that are dynamic and tell Wicket to render those sections using a component. Because in Listing 1 I want to keep the message dynamic, I have used the span element to mark the component and the `wicket:id` attribute to identify the component.

### Java page class (HelloWorld.java)

Listing 2 shows the page class for the HelloWorld.java example.

### Listing 2. HelloWorld.java

```
package myPackage;

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

public class HelloWorld extends WebPage
{
    public HelloWorld()
    {
        add(new Label("message", "Hello World using Wicket!!"));
    }
}
```

The ID given for the label component in the page class ("message") must match with the Wicket ID of the element in the template file (`wicket:id="message"`). Wicket's Java page class contains all the dynamic behavior of a Web page. There is a one-to-one relationship between an HTML template and a page class.

Finally, you need to create an `Application` object that serves as a starting point when the application is loaded by a Web container. It is also a place to do initial settings and configuration for the application. For example, you can define the home page for your application by overriding the `getHomePage()` method and returning the page class corresponding to your application's home page, as shown in Listing 3.

### Listing 3. HelloWorldApplication.java

```
package myPackage;

import org.apache.wicket.protocol.http.WebApplication;

public class HelloWorldApplication extends WebApplication {

    protected void init() {
    }

    public Class getHomePage() {
        return HelloWorld.class;
    }
}
```

You can also modify or override the default application settings by overriding the `init()` method and then calling `getXXXSettings()` to retrieve an interface to a mutable `Settings` object. The interfaces are returned by the methods shown in Table 1 and can be used to configure framework settings for your application:

Table 1 shows a sample list of settings that can be applied at an application-wide level in the `Application` class.

### Table 1. Application-wide settings

Method	Used for
<code>getApplicationSettings</code>	Application's application-wide settings
<code>getDebugSettings</code>	Application's debug-related settings
<code>getExceptionSettings</code>	Application's exception handling settings
<code>getMarkupSettings</code>	Application's markup-related settings
<code>getPageSettings</code>	Application's page-related settings
<code>getRequestCycleSettings</code>	Application's request cycle-related settings
<code>getSecuritySettings</code>	Application's security-related settings
<code>getSessionSettings</code>	Application's session-related settings

Table 2 shows some examples of how to apply an application-wide setting in the `Application` class.

**Table 2. Examples of application-wide settings.**

Example	Does this
<code>getApplicationSettings().setPageExpiredErrorPage(<code>Page</code> class&gt;)</code>	Set a generic page to be displayed in case a page expires because of a session timeout
<code>getMarkupSettings().setDefaultMarkupEncoding(<code>"UTF-8"</code>)</code>	Set the default markup format to be used for rendering
<code>getSecuritySettings().setAuthorizationStrategy(<code>AuthorizationStrategy</code> Instance&gt;)</code>	Set the authorization strategy to be used for your application

### web.xml configuration file

As a last step, to load your application and have it available, you need to define the Wicket servlet class, passing it the application class name as a parameter in the `web.xml` configuration file, as shown in Listing 4.

**Listing 4. web.xml configuration file**

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

<web-app>
  <display-name>Wicket Hello World Example</display-name>
  <servlet>
    <servlet-name>HelloWorldApplication</servlet-name>
    <servlet-class>
      wicket.protocol.http.WicketServlet
    </servlet-class>
    <init-param>
      <param-name>applicationClassName</param-name>
      <param-value>myPackage.HelloWorldApplication</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>HelloWorldApplication</servlet-name>
```

```
<url-pattern>/hello/*</url-pattern>
</servlet-mapping>
</web-app>
```

The application can now be packaged as a War/Ear file and deployed into any Java Platform, Enterprise Edition (Java EE)-based servlet container, such as Tomcat or WebSphere®, and can be invoked using the URL `http://<serverName:port>/warfileName/hello/`, substituting servername and warfilename to the appropriate values. In the example, I call `http://localhost:8090/sample/hello`, as shown in Figure 1.

**Figure 1. Sample HelloWorld Wicket application**



## Wicket life cycle

Having an in-depth knowledge of the Wicket life cycle helps you use Wicket more effectively. The life cycle consists of the following steps:

- Application loading
- Request processing
- Rendering

### Application loading

A Wicket-based application is loaded by defining a Wicket servlet in the `web.xml` file, which can be loaded into any Java EE-based application server, as shown in Listing 5.

#### Listing 5. `web.xml`

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

<web-app>
  <display-name>Sample Wicket Application</display-name>
  <servlet>
    <servlet-name>SampleWicketApplication</servlet-name>
    <servlet-class> wicket.protocol.http.WicketServlet </servlet-class>
    <init-param>
      <param-name>applicationClassName</param-name>
      <param-value>wicket.sample.SampleApplication</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>SampleApplication</servlet-name>
    <url-pattern>/sample/*</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

The servlet class specified must always be `wicket.protocol.http.WicketServlet` and the `applicationClassName` parameter value must be of type `WebApplication`. In this case, `SampleApplication` extends `WebApplication`. Whenever a client calls the application in Listing 5 using the URL `/sample/*`, the server loads the `WicketServlet`, which in turns creates a single instance of application class, that is, `SampleApplication`.

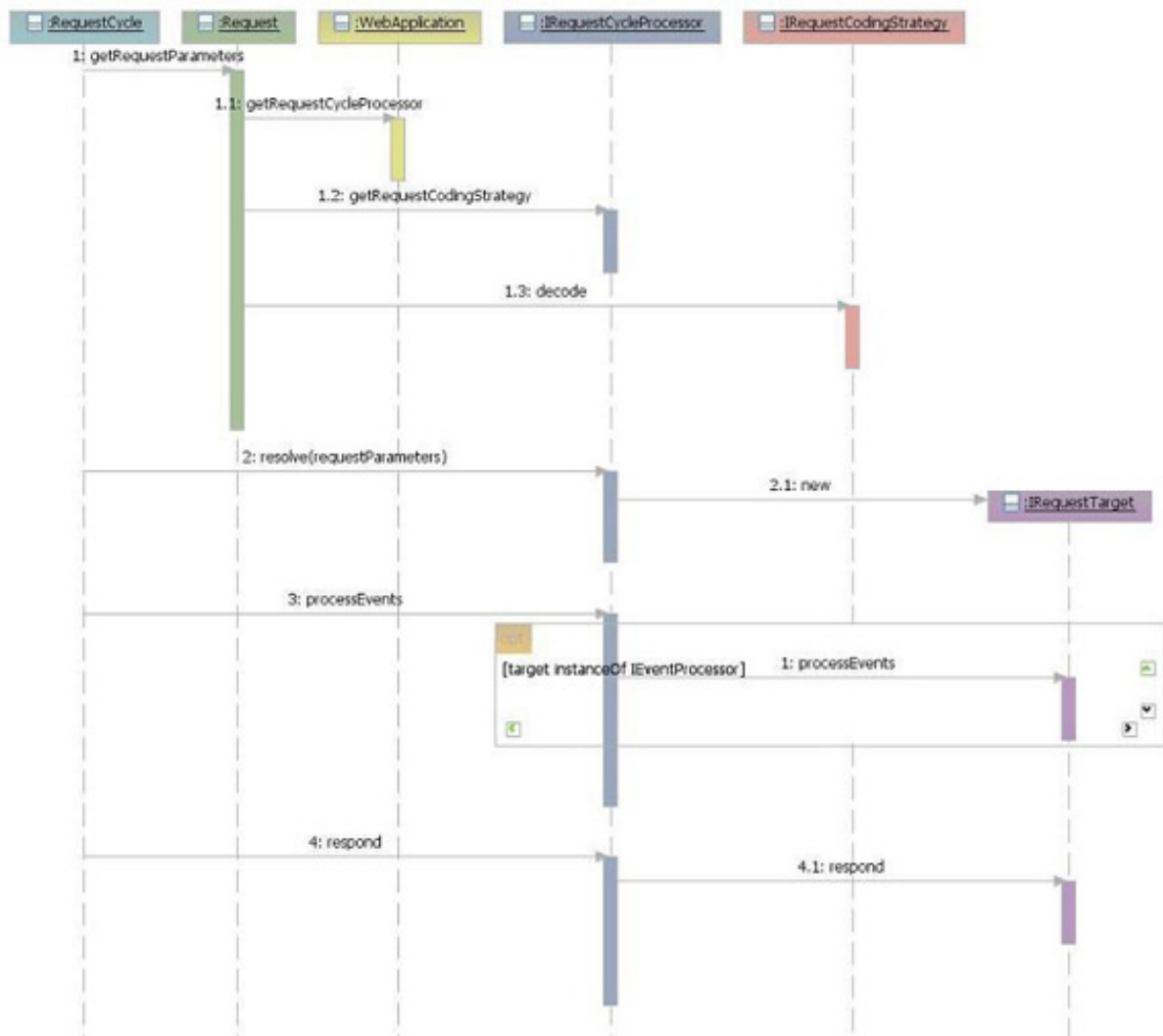
The `applicationClassName` parameter specified must be the fully qualified name of a class that extends `WebApplication`. If the application class cannot be found, does not extend `WebApplication`, or cannot be instantiated, a runtime exception of type `WicketRuntimeException` is thrown.

## Request processing

After the application class is loaded by the Wicket servlet, it creates a session for the servlet request in case no session exists, using its session factory. The application then creates a `RequestCycle` object using the session object and delegates the call to `RequestCycle` to handle the request.

The request cycle then calls the `onBeginRequest` method to let subclasses do preprocessing operations just before the request is processed. The request cycle goes through several states and, depending on the current state, sends different instructions to the request cycle processor. After sending all instructions, the request cycle reaches its final state, which denotes the end of request processing, as shown in Figure 2 (to see a larger image, click [here](#)).

**Figure 2. Wicket's request processing sequence flow**



The request cycle processor is responsible for handling instructions during the request cycle. It's used by `RequestCycle`, which calls its methods in the predefined order:

- `IRequestTarget resolve(RequestCycle, RequestParameters)` is called to get the request target. For example, a request might refer to a bookmarkable page or a component on a previously rendered page. This is one of the main responsibilities of the request cycle processor. The `RequestParameters` object consists of all possible optional parameters that can be translated from servlet request parameters and serves as a strongly typed variant of these (see `IRequestCodingStrategy`).
- `void processEvents(RequestCycle)` is meant to handle events like calls on components, for example, `onClick()` and `onSubmit()` event

handlers.

- `void respond(RequestCycle)` is called to create a response, that is, generate a Web page or do a redirect.
- `void respond(RuntimeException , RequestCycle)` is called whenever an uncaught exception occurs during the event handling or response phase so that an appropriate exception response can be generated.

## Rendering

A page renders itself by rendering its associated markup (the HTML file that sits next to the page). As `MarkupContainer` (the superclass for page) iterates through the markup stream for the associated markup, it looks up components attached to the tags in the markup by ID. Because the `MarkupContainer` (in this case, a page) is already constructed and initialized by `onBeginRequest()`, the child for each tag should be available in the container. After the Component is retrieved, its `render()` method is called.

`Component.render()` follows these steps to render a component:

1. Determine component visibility. If the component is not visible, the `RequestCycle's Response` is changed to `NullResponse.getInstance()`, which is a `Response` implementation that simply discards its output.
2. `Component.onRender()` is called to allow the Component's rendering implementation to actually render the Component.
3. Any Component models are detached to reduce the size of the Component hierarchy, in case it is replicated across a cluster.

The `renderComponent` method gets a mutable copy of the next Tag in the markup stream for the component and calls `onComponentTag(Tag)` to allow the subclass to modify the tag. After the subclass has changed the tag, it is written out to the `Response` with `renderComponentTag(Tag)`, and the markup stream is advanced to the next tag.

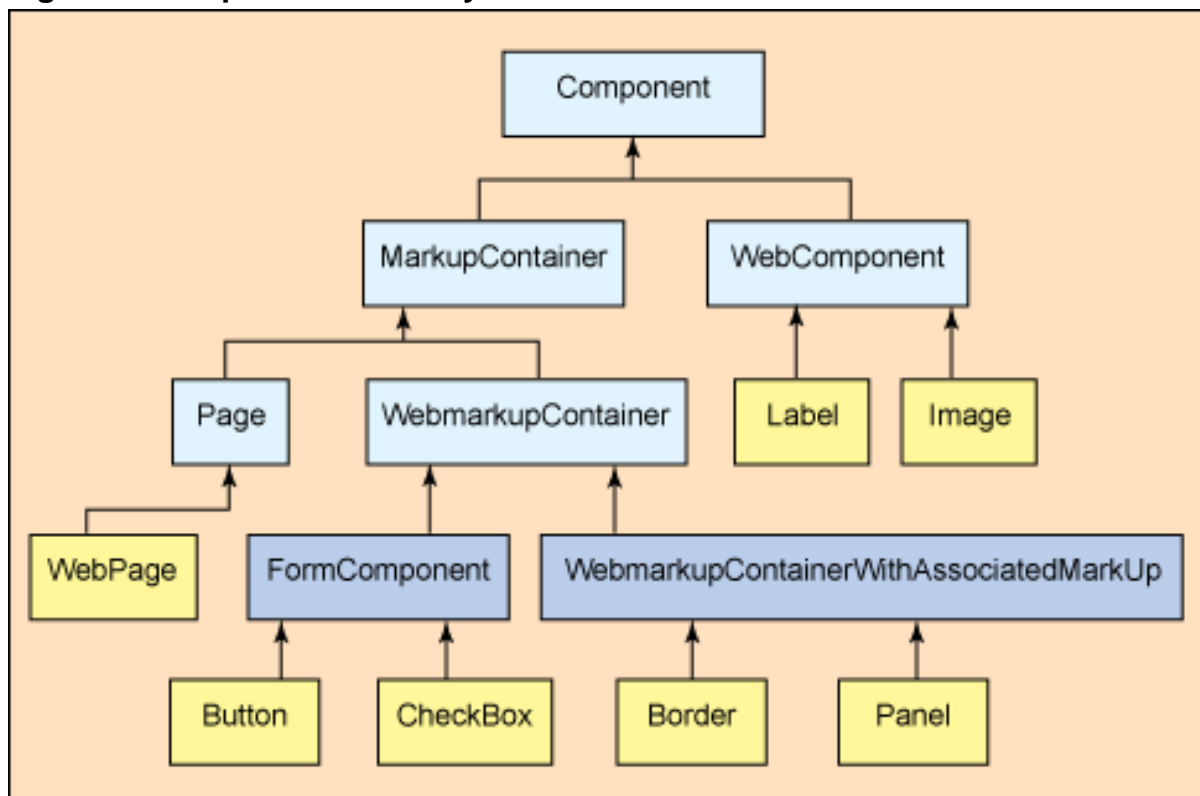
Next, `onComponentTagBody()` is called, passing in the `MarkupStream` and the `ComponentTag` that was written out as the opening tag. This allows the component to do whatever it needs to do to produce a body for the component tag. One operation the subclass can call in `onComponentTagBody()` is `Component.replaceComponentTagBody()`, which replaces the markup inside the component's body with an arbitrary `String`. Finally, the framework writes out any appropriate closing tag for the component's open tag.

## Creating custom components

One major feature that Wicket offers is the ease with which custom components can be developed. Wicket provides a very flexible Component/programming model that makes developing custom components a breeze. Custom components can be in the form of an HTML field or a panel that can be used within a page. Some of the common forms of reusable components in a Web-based application are Header, footer, Navigational bar, and so on.

First, take a look at Wicket's own component hierarchy that you can use with little effort to build a new custom component, or extend it to add new features.

**Figure 3. Component hierarchy**



### Component

`Component` sits at the top of the hierarchy and acts as an abstract base class for all components. It provides various features such as

- Identity: Non-Null ID that must be unique within its container and can be retrieved using `getID()`
- Model: Holds the data to be rendered as a response in HTML

- **Attributes:** Can be added to any component to manipulate the markup tag to which the component is attached

## WebComponent

The `WebComponent` acts as a base class for simple HTML components such as `Label` and `Image`.

## MarkupContainer

`MarkupContainer` holds all the child components and doesn't have its own markup. Child components can be added or replaced by calling `add()` and `replace()` methods, respectively, and can be retrieved by using the OGNL notation. For example, calling `get("a.b")` will return the component whose ID is "b" and whose parent component ID is "a".

## WebMarkupContainer

This serves as a container of HTML markup and components. It is very similar to the base class `MarkupContainer`, except that the markup type is defined to be HTML.

## WebMarkupContainerWithAssociatedMarkup

This extends `WebMarkupContainer` and provides the added ability to process header tags.

## Panel

A `Panel` is a reusable component that holds markup and other components. You can create reusable custom components by extending this class.

## Border

A `Border` component has its own associated markup and can be used to define the portion of its associated markup file, which is to be used in rendering the border.

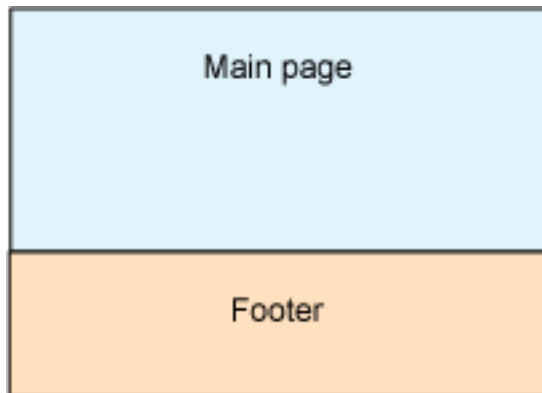
## Page

`Page` acts as an abstract base class for all pages and can contain any arbitrary tree of `Components`. All life cycle events such as `onBeginRequest`, `onEndRequest()`, and `onModelChanged()` can be overridden by subclasses of `Page`, which are interested in handling and processing those events.

Having had a brief overview of Wicket's own component hierarchy, now take a look at how to build your own custom component by extending Wicket's `Panel` component.

Using Figure 4, take a look at how to code a sample custom component like footer using Wicket.

**Figure 4. Footer component layout**



A custom Wicket component that has a visual representation typically consists of the following artifacts:

- An HTML template (Footer.html), as shown in Listing 6
- Optional JavaScript, a style sheet, or Images
- A corresponding Java Component class (Footer.java) that extends from one of the several component base classes that comes with the standard Wicket distribution, as shown in Listing 7.

**Listing 6. Footer.html**

```
<wicket:panel>
  <hr>
  Copyright <span wicket:id="year">2008</span>. IBM Inc. All rights reserved.
</wicket:panel>
```

**Listing 7. Footer.java**

```
import java.util.GregorianCalendar;
import org.apache.wicket.markup.html.basic.Label;
import org.apache.wicket.markup.html.panel.Panel;

public final class Footer extends Panel {

    public Footer(String id) {
        super(id);
        add(new Label("year", "" + new GregorianCalendar().get(GregorianCalendar.YEAR)));
    }

}
```

**Note:** The artifacts in Listing 6 and Listing 7, the HTML artifacts, and the server-side representation of the component (a Java class) typically reside in the same package

## Embedding/using components

To use the component—after it has been defined—simply instantiate it in the required `Page` class (that is, `<Page>.java`) file, for example, `Home.java` (shown in Listing 8), and call it by embedding the ID of the custom component in the corresponding template (`<Template>.html`) file, such as `Home.html`, as shown in Listing 9.

### Listing 8. Home.java

```
import org.apache.wicket.markup.html.WebPage;
public class Home extends WebPage {

    public Home() {
        add(new Footer("footer"));
    }
}
```

### Listing 9. Home.html

```
<html>
<head></head>
<body>
    Body of Home page.
    <span wicket:id="footer">Footer Info</span>
</body>
</html>
```

## Wicket validation

Wicket supports both client- and server-side form validations. Validation is supported in the form of built-in validators. Wicket ships with a number of validators such as `NumberValidator`, `StringValidator`, `PatternValidator`, `DateValidator`, and `RequiredValidator`.

You can also write a custom validator to perform validations that are not built into Wicket. In the case of server-side validation, after the form is submitted, Wicket goes through all the form components placed within the form and executes any attached validators on component input. In the process, Wicket collects any error messages, if any are thrown by the validators. The `FeedbackPanel` component then displays all the collected error messages.

### Built-in validators

Wicket provides a much simpler way of handling validation. With Wicket, you can set any validators on the field components at the time of creation in the page file. For example, a mandatory field where you expect the user to enter some input is created in the page, as shown in Listing 10.

## Listing 10. Making a TextField a mandatory input

```
TextField firstNameComp = new TextField("firstName");
    firstNameComp.setRequired(true);
```

In Wicket, the `FeedbackPanel` component renders any form-related errors in the page. Actually, `FeedbackPanel` displays all types of feedback messages attached to the components contained within the `Page`. However, you can filter only those message types (info, error, debug, warn, and so on) that need to be displayed. This component can be added to the page as shown in Listing 11.

## Listing 11. Adding FeedbackPanel in page class

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

Reference to the "feedback" component needs to be added in the HTML markup to display any validation errors if they exist, as shown in Listing 12.

## Listing 12. Embedding FeedbackPanel in template file

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

## Custom validator

In a case where form-level validations can't be handled using built-in validators you can create a custom validator by subclassing the `AbstractFormValidator` class and using a constructor that takes as parameters the form components to be validated. Custom validators need to be added to the form at the time of form instantiation.

Take an example where you need to ensure that either of the two given fields in a form is not empty. Because this can't be achieved using a built-in validator, you must write a custom validator, as shown in Listing 13 .

A custom validator can be built by extending the `AbstractFormValidator` class as shown.

## Listing 13. EitherInputValidator.java

```
import java.util.Collections;
import org.apache.wicket.markup.html.form.Form;
import org.apache.wicket.markup.html.form.FormComponent;
import org.apache.wicket.markup.html.form.validation.AbstractFormValidator;
import org.apache.wicket.util.lang.Objects;

public class EitherInputValidator extends AbstractFormValidator {
```

```

/** form components to be validated. */
private final FormComponent[] components;

public EitherInputValidator(FormComponent f1, FormComponent f2) {
    if (f1 == null) {
        throw new IllegalArgumentException(
            "FormComponent1 cannot be null");
    }
    if (f2 == null) {
        throw new IllegalArgumentException(
            "FormComponent2 cannot be null");
    }
    components = new FormComponent[] { f1, f2 };
}

public FormComponent[] getDependentFormComponents() {
    return components;
}

public void validate(Form form) {
    // we have a choice to validate the type converted values or the raw
    // input values, we validate the raw input
    final FormComponent f1 = components[0];
    final FormComponent f2 = components[1];
    String f1Value = Objects.stringValue(f1.getInput(), true);
    String f2Value = Objects.stringValue(f2.getInput(), true);
    if ("".equals(f1Value) || "".equals(f2Value)) {
        final String key = resourceKey(components);
        f2.error(Collections.singletonList(key), messageModel());
    }
}
}
}

```

**Note:** You must call `FormComponent.getInput` to get the values to be validated, rather than getting the associated models. This is because the models are not updated until after validation.

To use a custom validator you must add it to the form, passing it field elements on which the validation needs to be applied, as shown in Listing 14.

#### Listing 14. Using custom validator in page class

```

Form myForm = new Form("form");
FormComponent f1 = new TextField("firstName");
myForm.add(f1);
FormComponent f2 = new TextField("lastName");
myForm.add(f2);
myForm.add(new EitherInputValidator (f1, f2));

```

Finally, the message displayed if the validation fails needs to be added to the properties file, as shown in Listing 15.

#### Listing 15. Validation message in the properties file

```

EitherInputValidator = Please enter data for either '${input0}' from ${label0}
                        or '${input1}' from ${label1}.

```

## Using Ajax with Wicket

Ajax (Asynchronous Javascript And XML) not only allows you to build a rich UI, but also a high-performance application because an Ajax-enabled application only updates the page rather than causing an entire page refresh. This provides feedback similar to a desktop-based application. This is achieved using the browser's implementation of XMLHttpRequest, which lets the client communicate asynchronously with the server and manipulate the page content dynamically based on the response received from the server through the use of HTML DOM APIs.

Wicket works around the problem of users having to send and process the data between the server and browser. Instead of sending data to the client, Wicket renders the components on the server side and sends the rendered markup. This might not be as efficient, but it is much easier and faster to develop Ajax behaviors. For example, if you want to update a panel that shows user information on the client using Ajax, all that is necessary is to tell Wicket that the panel should be updated. There is no need to parse the response returned by Wicket on the client using JavaScript or to manipulate the DOM on the client.

In Wicket, an Ajax-based request is modeled as a behavior. Wicket comes with several implementations of the IBehaviour interface such as AbstractDefaultAjaxBehaviour and AjaxFormComponentUpdatingBehaviour, which do the internal processing and call the respond() method passing in the AjaxRequestTarget. This target object fetches the actual content that needs to be sent back to the browser in response to the Ajax request. AjaxRequestTarget renders only those components that are added to it, and built-in JavaScript artifacts re-render the added components by initializing the HTML outerHTML property.

Let's take an Ajax autoCompleteName example where, based on the user input, the application will predict a list of probables from which the user can select.

The application consists of a page with a text field element, shown in Listing 16, a list of predefined names (which can also be dynamic) in memory, and an Ajax behavior for displaying a list of probable values when the user starts entering text.

### Listing 16. HTML page template (AutoCompleteName.html)

```
<html>
  <head>
    <wicket:head>
      <style>
        div.wicket-aa {
          font-family: Verdana,"Lucida Grande","Lucida Sans Unicode",Tahoma;
          font-size: 12px;
          background-color: white;
          border-width: 1px;
          border-color: #cccccc;
          border-style: solid;
        }
      </style>
    </wicket:head>
  </head>
</html>
```

```

        padding: 2px;
        margin: 1px 0 0 0;
        text-align:left;
    }
    div.wicket-aa ul {
        list-style:none; padding: 2px; margin:0;
    }
    div.wicket-aa ul li.selected {
        background-color: #FFFF00; padding: 2px; margin:0;
    }
    </style>
</wicket:head>
</head>

<body bgcolor="#FFCC00">
  <br>
  <br>
  <form wicket:id="form">
    <b>Name :</b> <input type="text" wicket:id="name" size="60" />
  </form>
</body>
</html>

```

**Figure 5. Wicket Ajax example (on load)**



In Wicket, this auto-completion behavior is modeled in class `AutoCompleteBehaviour`. You need to extend this class and provide implementation for the methods `Iterator getChoices(String input);` to return the probable user list based on the input (Listing 17).

#### **Listing 17. Page class (AutoCompleteName.java)**

```

package myPackage;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;

```

```

import org.apache.wicket.extensions.ajax.markup.html.autocomplete.AutoCompleteTextField;
import org.apache.wicket.markup.html.WebPage;
import org.apache.wicket.markup.html.form.Form;
import org.apache.wicket.model.Model;

public class AjaxWorld extends WebPage {
    private List names = Arrays.asList(new String[] { "Kumarsun", "Ramkishore",
        "Kenneth", "Kingston", "Raju", "Rakesh", "Vijay", "Venkat", "Sachin" });

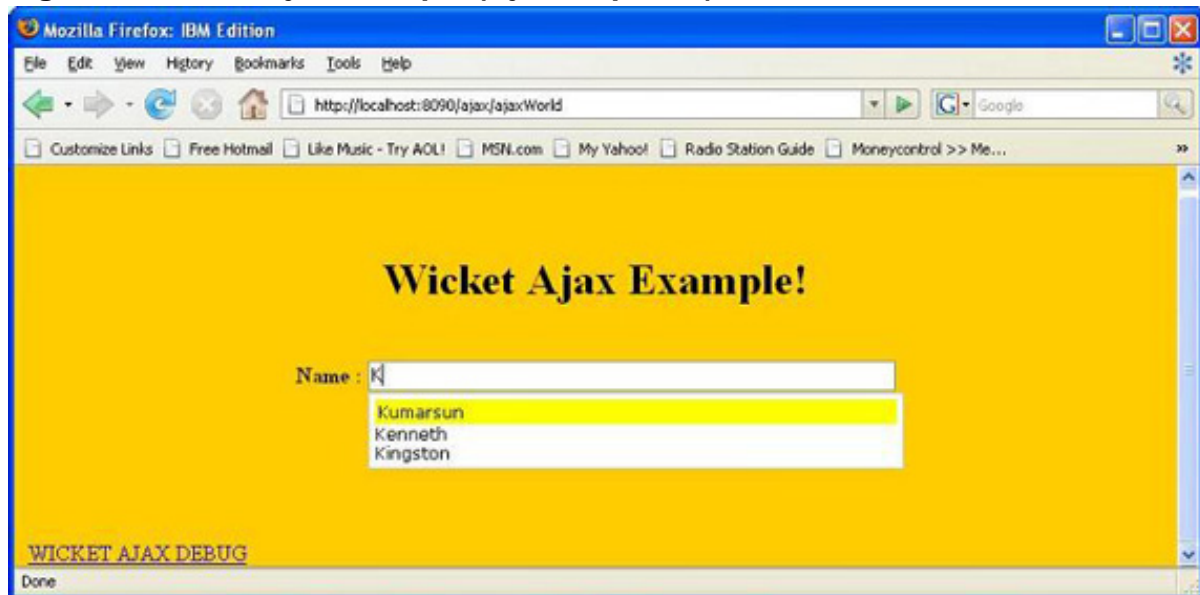
    public AjaxWorld() {

        Form form = new Form("form");
        AutoCompleteTextField txtName = new AutoCompleteTextField("name", new Model()){

            protected Iterator getChoices(String input) {
                List probables = new ArrayList();
                Iterator iter = names.iterator();
                while (iter.hasNext()) {
                    String name = (String) iter.next();
                    if (name.startsWith(input)) {
                        probables.add(name);
                    }
                }
                return probables.iterator();
            }
        };
        form.add(txtName);
        add(form);
    }
}

```

**Figure 6. Wicket Ajax example (Ajax response)**



## I18N Support

I18N in Wicket is handled by reading the messages from the locale-specific properties files. This is done through the use of the `StringResourceModel` class

in the page class or through the use of the `<wicket:message>` tag in the HTML markup file. `StringResourceModel` can also be used for formatting localized messages to be displayed.

### **I18N using `<wicket:message>` tag**

Labels and other information that needs to be displayed in pages can be localized using the `<wicket:message>` tag instead of hard-coding the label as shown in Listing 18.

#### **Listing 18. Template file (MyPage.html)**

```
<html>
<head>
<title></title>
</head>
<body>
    <form wicket:id="myForm">
        <wicket:message key="first-name">First Name</wicket:message>
    </form>
</body>
</html>
```

Whenever Wicket encounters the `<wicket:message>` tag, it reads the value of the key from the locale-specific properties based on the locale set on the HTTP request object.

### **I18N using `StringResourceModel` class**

Instead of using the `<wicket:message>` tag in the template page, you can directly use the `StringResourceModel` in the page to retrieve localized messages based on the input key, as shown in Listing 19. The advantage of the `StringResourceModel` class is that you can flexibly format the message before rendering.

#### **Listing 19. Page class (MyPage.java)**

```
public class MyPage extends WebPage {
    public MyPage() {
        Form form = new MyForm("myForm");
        String firstNameLabel = new
StringResourceModel("first-name").getString();
        form.add(new Label("firstName", new Model(firstNameLabel)));
        add(form);
    }
}
```

### **Resource bundle search order**

As usual in Java i18n systems, messages are looked up by locale. The locale is automatically extracted from the HTTP request header locale in the User Agent field

and set in the Wicket `WebRequest` object. However, it can also be explicitly set with `getSession().setLocale(Locale.US)`.

If you need to override the locale for a specific component, override `getLocale()` on that component and it will be used by that component and its children. If a client does not provide a preferred `Locale`, the Java code default `Locale` is used.

Wicket searches all resource files with the names equal to the components in your component hierarchy, with your application last. Remember that after the message is found, Wicket halts the search process. So, when Wicket wants to look up a message used in `MyPanel`, where `MyPanel` is contained within `MyForm` placed under `MyPage`, and the application is called `MyApplication`, Wicket will look in:

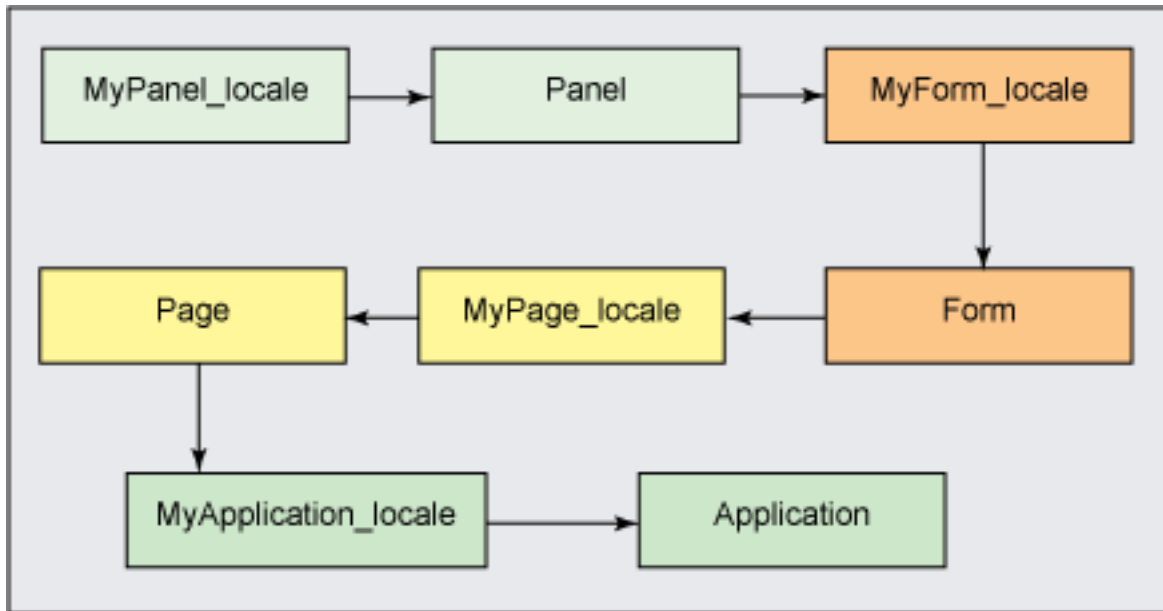
- `MyPanel_locale.properties`, ..., and then `MyPanel.properties`
- `MyForm_locale.properties`, ..., and then `MyForm.properties`
- `MyPage_locale.properties`, ..., and then `MyPage.properties`
- `MyApplication_locale.properties`, ..., and then `MyApplication.properties` (..)

Actually, it even goes two steps further. Wicket will also look at property files for the base classes of `MyPanel`, `MyForm`, `MyPage`, and `MyApplication`. When `MyPanel` inherits directly from `Panel`, `MyForm` directly from `Form`, `MyPage` directly from `Page`, and `MyApplication` directly from `Application`, Wicket will look in:

- `MyPanel_locale.properties`, ..., and then `MyPanel.properties`
- `Panel_locale.properties`, ..., and then `Panel.properties`
- `MyForm_locale.properties`, ..., and then `MyForm.properties`
- `Form_locale.properties`, ..., and then `Form.properties`
- `MyPage_locale.properties`, ..., and then `MyPage.properties`
- `Page_locale.properties`, ..., and then `Page.properties`
- `MyApplication_locale.properties`, ..., and then `MyApplication.properties` (..)
- `Application_locale.properties`, ..., and finally `Application.properties` (..)

If all of the preceding steps fail, Wicket will use the `labelId` as the `Label`, by default.

**Note:** If `MyForm` is modeled as an inner class within `MyPage`, then wicket will look for a resource file named `MyPage$MyForm.properties`. So, as a best practice you can use `MyApplication.properties` for site-wide messages and override these in any of the other properties files.

**Figure 7. Resource bundle search order**

## Unit testing Wicket pages

Wicket supports out-of-container unit testing through the use of a built-in mock object framework. This framework makes sure that the framework and the surrounding environment objects the application interacts with will behave as configured, even when running outside a Java EE servlet container. This results in productivity gains because you are not required to restart the container, allowing you to concentrate on unit testing the components you are interested in.

Mock objects are used to test a portion of your code logic in isolation from the rest of the code. They provide methods to let the tests control the behavior of all the business methods of the faked class.

Wicket support for unit testing is based on extending the JUnit framework. It's `wicket.util.test.WicketTester` class provides lot of helper methods that help you imitate various user actions, such as clicking a link or form submission, and behaviors such as page rendering or asserting the presence of error messages.

Listing 20 shows an example of a production page (`MyPage.java`), with some components and user actions such as a form submit and a link click.

### Listing 20. Page class (`MyPage.java`)

```

import org.apache.wicket.markup.html.WebPage;
import org.apache.wicket.markup.html.basic.Label;
import org.apache.wicket.markup.html.form.Button;
import org.apache.wicket.markup.html.form.Form;
import org.apache.wicket.markup.html.form.TextField;
import org.apache.wicket.markup.html.link.Link;

```

```

public class MyPage extends WebPage {
    public MyPage() {
        MyForm form = new Form("myForm");
        form.add(new Label("firstNameLabel", "First Name"));
        form.add(new Label("lastNameLabel", "Last Name"));

        form.add(new TextField("firstName"));
        form.add(new TextField("lastName"));

        form.add(new Button("Submit"));
        form.add(new Link("nextPage") {
            public void onClick() {
                setResponsePage(new NextPage("Hello!"));
            }
        });
    }
}

```

## Test case to test page render

One of the minimum tests that you must do is to ensure that each and every page renders correctly. Successful execution of this test, as shown in Listing 21, ensures that the template and the page hierarchy match.

### Listing 21. Page render test (MyPageRenderTest.java)

```

import org.apache.wicket.util.test.WicketTester;
import junit.framework.TestCase;

public class MyPageRenderTest extends TestCase {

    private WicketTester tester;

    public void setUp() {
        tester = new WicketTester();
    }

    public void testMyPageBasicRender() {
        WicketTester tester = new WicketTester();
        tester.startPage(MyPage.class);
        tester.assertRenderedPage(MyPage.class);
    }
}

```

## Test case to test page components

The `WicketTester` class has built-in methods to verify that a given page has all the required components. This is done through the use of its `assertComponent()` method, passing it the component path and the component type to expect at the given path, as shown in Listing 22. The path given needs to be relative to the page in which it is embedded.

### Listing 22. Page components test (MyPageComponentsTest.java)

```

import junit.framework.TestCase;

```

```
import org.apache.wicket.markup.html.form.TextField;
import org.apache.wicket.util.test.WicketTester;

public class MyPageComponentsTest extends TestCase {

    private WicketTester tester;

    public void setUp() {
        tester = new WicketTester();
    }

    public void testMyPageComponents() {
        WicketTester tester = new WicketTester();
        tester.startPage(MyPage.class);

        // assert rendered field components
        tester.assertComponent("myForm:firstName", TextField.class);
        tester.assertComponent("myForm:lastName", TextField.class);

        // assert rendered label components
        tester.assertLabel("myForm:firstNameLabel", "First Name");
        tester.assertLabel("myForm:lastNameLabel", "Last Name");
    }
}
```

## Test case to test OnClick user action

Testing a user action such as clicking a link can be done using WicketTester's `clickLink()` method, passing it the link component ID path, and then verifying the rendered page, as shown in Listing 23.

### Listing 23. Page OnClick action test

```
public void testOnClickAction() {
    tester.startPage(MyPage.class);

    // click link and render
    tester.clickLink("nextPage");

    tester.assertRenderedPage(NextPage.class);
    tester.assertLabel("nextPageMessage", "Hello!");
}

public void testNextPageRender() {
    // provide page instance source for WicketTester
    tester.startPage(new TestPageSource() {
        public Page getTestPage() {
            return new NextPage("Hello!");
        }
    });

    tester.assertRenderedPage(YourPage.class);
    tester.assertLabel("nextPageMessage", "Hello!");
}
```

## Test case to test the form submit user action

Form submission in Wicket can be tested using Wicket's

wicket.util.testster.FormTester class, which has APIs for setting the input values for the field components placed within a form, and subsequently submitting the form. Assuming the page that gets displayed on submitting the form is Welcome.java containing the message "Welcome to Wicket", form submission can be tested as shown in Listing 24.

#### Listing 24. Page OnSubmit action test

```
public void testFormSubmit ()
{
    // Create the FormTester object
    FormTester ft = tester.newFormTester("myForm");

    // Set the input values on the field elements
    ft.setValue("firstName", "Kumar");
    ft.setValue("lastName", "Nadar");

    // Submit the form once the form is completed
    ft.submit("Submit");

    // Check the rendered page on form submission
    tester.assertRenderedPage(Welcome.class);
    // verify the message on the rendered page
    tester.assertInfoMessage(new String[]{"Welcome to Wicket"});
}
```

## Conclusion

A Plain Old Java Object (POJO)-centric framework like Wicket can be used to rapidly build Web-based applications in a non-intrusive and simplified manner. HTML or other markups are not polluted with programming code, making it easy for UI designers to recognize and avoid framework tagging.

Wicket allows developers to create pages in a Java-Swing-like fashion to let them avoid the overuse of XML configuration files. It also provides a comprehensive way to unit test the pages developed.

## Downloads

Description	Name	Size	Download method
Sample Wicket Code for this article	wa-aj-wicketsource.zip	4.5K	<a href="#">HTTP</a>

[Information about download methods](#)

# Resources

## Learn

- Read [Control where HTML files are loaded from](#) in the Apache Wicket documentation.
- Check out the [Wicket home page](#) and the tutorials there to get a better understanding of how Wicket works.
- Visit [Wicket examples](#) for a selection of examples that best demonstrate its core functions.

## Get products and technologies

- Get [Apache Wicket](#) and try it for yourself.
- Download [Qwicket](#), a quick-start application for the Wicket framework.

## Discuss

- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

# About the author

Kumarsun M. Nadar

Kumarsun Nadar currently works as a Senior Staff Software Engineer with IBM India Software Labs (ISL) in the WebSphere Business Service Fabric product team, based out of Mumbai, India. As part of this team, he was involved in the UI development of the Fabric Web Tools module, which was based on the Wicket framework. He has attained SUN Microsystems certifications in SCJP, SCWCD and SCBCD and has experience in various client- and server-side technologies such as Wicket, EJB, Hibernate, Struts and so on, based on the Java/J2EE platform. His hobbies include watching, as well as playing, sports such as Cricket, and TT.