java/j2ee Web Application Framework

Reference Documentation

Version 3.1

Table of Content

Preface

For a long time, we have developed medium to large scale web applications using common patterns in the java web world. Compared to the development of a rich client, it was a fall back to older ages. Therefore, we developed jWic, which has enabled us to develop web applications as easy as we develop a Java rich client. As we believe that this style represents an attractive alternative to the classic style, we have released it as an open source project.

jWic is a small framework that does not aim to be "the-one-and-only". It's design aim is to either run stand-alone or integrate into other frameworks without wiring into them, creating unnecessary dependencies. The core functionality of jWic is to manage the UI of your application so that the programmer can concentrate upon their business case.

1 jWic Fundamentals

1.1 What is a jWic Application?

A jWic application is a collection of controls that provide a (web-)interface for a user. An application is started when a user requests a resource from the web-server that is linked to the application they wish to start. The application is then loaded into the server's memory until the user exits the application or their session times out.

Each user session has it's own instance of the application and a user may have multiple instances of an application on the server within the same session. This could be the case when you are using generic applications, like an object editor app. A user may reference (link) from one editor instance to another which works with a different object. They are then able to return to their previous instance, which is still in the same state as when they left it.

1.2 What is a Control?

A Control is a JavaBean that represents one or more elements on a web-page. It maintains its own state, which can be changed by the user or the application code. These changes are reflected to the user the next time the application (or parts of it) is rendered into a HTML page.

Controls are rendered to HTML using a ControlRenderer. While the control specifies the type of renderer required, the framework chooses the implementation. Read more about renderers in Chapter 3 (Renderers).

Controls must extend the abstract `de.jwic.base.Control` class.

1.2.1 Control Lifecycle

Controls are created using their constructor, just like any other java object. Earlier versions of jWic have used the factory pattern to create controls, but this has been removed for simplicity.

A control lives in a hierarchy of controls and containers. Therefore, the control must be added to a container, which results in the initialisation of the control. Since version 3.0 of jWic, the container must be provided in the constructor of the control. This way it is impossible to create control instances that are not fully initialised.

Sample:

```
1:    ButtonControl button = new ButtonControl(container, "abortButton");
2:    button.setTitle("Abort");
```

A control must implement a constructor with the arguments for the container the control should be added to and the name of the control. After the super(..) constructor has been invoked, the control may then initialise itself.

Sample:

```
1:    public MyControl(IControlContainer container, String name) {
2:          super(container, name);
3:          // setup the control
4:    }
```

Controls are destroyed when they are removed from their container. This happens if the control is removed by the application code or when the application is destroyed (i.e. because the user exited the application).

Each control implements the `Serializable` interface and should adhere to the rules for serialization. This enables the JWicRuntime to serialize a running application to disk to avoid excessive memory usage. It is also a must for clustering.

Convenience Constructors – Just Say No

"Some programmers demand convenience constructors using arguments such as, "Every time a button is created, I always set the text so there should be a button constructor that takes a string". Although it is tempting to add convenience constructors, there is just no end to them. Buttons can have images. They can be checked, disabled and hidden. It is tempting to provide convenience constructors for these properties as well. When a new API is defined, even more convenience constructors are needed."
Quote: SWT: The Standard Widget Toolkit chapter 1.1

This design principle used is to minimize the size of the control library and to provide consistency, and therefore jWic does not normally provide convenience constructors.

1.2.2 Control Container

Containment (sometimes called nesting) is a valuable technique that allows one control to be placed inside another. ControlContainer support nesting of controls, allowing you to build complex containment hierarchies.

ControlContainer in jWic are controls themselves, except the SessionContext which acts as the root element in the application hierarchy. You can navigate from a control to it's parent using the `getContainer()` method, which returns either a `ControlContainer` or the `SessionContext`. Both of these classes implement the `IControlContainer` interface.

1.2.3 Control Name and Id

Each control must have a name to be identified within its container. The name must be unique within the container the control lives in. If the name is null or omitted[1], the container will generate a unique name for the control.

To identify a control within your application, a unique control id is composed when a control is added to a container. The control id represents the path to the control, using the name of each control in the hierarchy separated with a dot.

The name and id of a control cannot be changed once it is set.

Sample Hierachy

Control	Parent	Name	Id
MyApp	SessionContext	app	app
+ TextBoxControl	MyApp	txt1	app.txt1
+ ButtonControl	MyApp	btOk	app.btOk
+ ActionBar	MyApp	abar	app.abar
+ ButtonControl	ActionBar	save	app.abar.save

1.3 Events and Listeners

An event is simply an indication that something interesting has happened. Events, such as "element selected" are issued when the user interacts with controls. Event classes, used to represent the event, contain detailed information about what has happened. For example, when the user selects an item in a dropdown list, an event is created, capturing the fact that a "selection" occurred. The event is delivered to the application via a listener.

A listener is an instance of a class that implements one or more agreed-upon methods whose signatures are captured by an interface. Listener methods always take an instance of an event class as an argument. When something interesting occurs in a control, the listener method is invoked with an appropriate event.

The event model in jWic follows the JavaBeans listener pattern. Common events and their corresponding listeners are found in the *de.jwic.events* package. Events in jWic are usually defined by controls. The controls are notified by the framework that the user has triggered an action that is related to the control. The control examines the user action and fires an event, when appropriate, that an application programmer can register for and react to.

The following code fragment demonstrates how a to add a listener to a listbox to get notified when the selection has changed:

```
listbox.addElementSelectedListener( new ElementSelectedListener() {
      public void elementSelected(ElementSelectedEvent event) {
            // element selected
      }
});
```

[1] Some controls offer an alternative constructor without the name parameter.

1.4 The SessionContext

The SessionContext represents one instance of your application. It acts as the container for your root control(s) and provides methods to control your application, such as `exit()`.

A SessionContext object is created by the JWicRuntime when a user starts a new application. As the session context is not present in the UI, a control must be added to the SessionContext which is the **root**-control. According to the `IControlContainer` interface, the SessionContext may have more than one child-control, but the framework will render only one root control and its childs at a time.

1.4.1 Control Stack

The SessionContext maintains a stack of controls that are rendered as top-control. A top-control is the only object being rendered, including its child. Any control may be pushed on top of this stack, hiding all other controls in the stack. When the control is done, it may pop the control, returning to the previous one.

This mechanism is used to bring up dialog or wizard pages that should hide the previous controls. Imagine a mail-client application with a textfield to input the receiver mail address. When the user hits the "Lookup" button, you want to display an address lookup page and hide the mail form. This is done using the control stack.

The following code opens a dialog:

```
public void lookupNamesAction() {
      dialog = new AddrDialog(parent, "addr");
      dialog.addElementSelectedListener(this);
      getSessionContext().pushTopControl(dialog);
}
```

And this closes the dialog:

```
public void elementSelected(ElementSelectedEvent event) {
      fieldTo.setText(dialog.getSelectedAdress().getMail());
      getSessionContext().popTopControl();
      dialog.destroy(); // remove the control
      dialog = null;
}
```

Control Stack vs. Control Visibility

Another way of handling this issue is to place all possible controls into one page (container) and "hiding" all controls using the `visible` property, except the one on top. We discourage using this pattern as the complexity rises with the number of top-controls to handle. In this case, the controller must know all possible controls on top and manage them.
One more advantage of the control stack is that **any** control can push a control on top, without their parent control knowing it.

1.5 The Application

An Application object is used to manage the lifecycle of your application. The application class is responsible to create the root controls. It receives events from the framework when the application gets initialised and when it is destroyed.

Every jWic application you create should start with a subclass of Application. If you extend the abstract Application class, you will only have to implement the `createRootControl(..)` method where you create your application controls.

Sample:

```
public class MyApplication extends Application {
  public Control createRootControl(IControlContainer container) {
    // create a page
    Page page = new Page(container);
    page.setTitle("My-Application");

    Calculator calc = new Calculator(page);
    //You must return the control that is the root of the application!
    return page;
  }
}
```

1.6 The Page

The abstract class Page is extending the ControlContainer, adding a few special properties relevant for top-controls. Controls that act as a top- (or sometimes called root-) control are supposed to extend Page instead of the standard ControlContainer.

The Page control can specify the title of the html-page and receives client specific information about the visible width and height of the frame the application runs in as well as the top and left scrolling position of the page. The width and height of the frame enables controls to adjust their layout. The scrolling position is used to reposition the html page to the place it was before the submit.

If the top control is not the Page class or a subclass of it, this information is lost and after each submit, the html page is positioned at the top left position. Therefore, it is highly recommend that your top-controls inherit from the Page class. Imagine how annoying it is for the user when digging down a big tree, scrolling down over and over again.

2 The Web Interface

2.1 Abstracting jWic from the Web

One of the goals for jWic is to abstract the application from the surrounding environment. To reach this goal, jWic packs an application into kind of a nutshell, that has a clear interface to the outer world. But in the end, jWic applications still run in a browser, so how is this abstraction made?

The most important difference to a common web application is that the application instance is kept alive in the server's memory during it's lifetime (unless it is serialized out to disk in the meantime). When the user does some action in your application, eg. clicks a link, the page is submitted to the DispatcherServlet. The content of the page is actually contained within a form tag with the session ID stored in the form, so that the dispatcher can retrieve the matching SessionContext to update the state of the controls.

Each field that was submitted within the form is examined and the value is stored in the corresponding Field element of the control that has created it. The action that has lead to the submit is examined by the control and events fired to the application that contains the control to allow the application to react to the event eg. start a wizard when a link is clicked.

2.1.1 Field Mapping

A HTML form often contain elements that allow the user to enter or select data. They are defined by an `input`, `select` or `textarea` tag. The data that is inside those fields is send to the server when the page is submitted.
When a control uses fields, it must create a `Field` object that acts as a container for the data. A `Field` contains an String or an array of String objects.
Field-objects are very similar to Controls, except that they do not get rendered themselves. When a `Field` is created, it must get bound to a control and get a unique name. If no name is given, a unique ID is calculated for that field. Based upon the name and the control-id of the parent control, a unique id is generated for that field. This allows the framework to map submitted field values to the right `Field` object.

Fields are created like controls:

```
public class TextField extends Control {
      private Field field;
      public TextField(IControlContainer container, String name) {
            super(container, name);
            field = new Field(this, "name");
            field.setValue("Default Value");
      }
}
```

The velocity template that renders the HTML code must use the Field objects properties. For the above sample, it looks like this:

```
#set($field = $control.getField("name"))
<input
      name="$field.id"
      value="$jwic.formatInp($field.value)">
```

The resulting HTML code could look like this:

```
<input name="fld_controlid.name" value="Default Value">
```

The control could access the field using this piece of code:

```
public String getText() {
      return field.getValue();
}
public void setText(String text) {
```

```
        field.setValue(text);
}
```

This makes a control independent from other controls on the form. It is very important not to create static field names which would create conflicting fields if you have more then one instance of the same control on your page.

A Field object provides a ValueChangedEvent, that is fired when the value has changed.

Note that there is a standard jWic control for every standard HTML field like input, select and textarea. As long as those standard controls fulfil your requirements, you wont have to deal with the `Field` object.

2.1.2 Action URLs

A page usually contains one or more elements that define some kind of action. Such an element might be a button or a link that can be clicked by the user. Whenever a user (or script) performs an action, the page must be submitted to the server, including the information which control is responsible and what kind of action has happened.

The submit is performed by a JavaScript function defined by the framework. The syntax looks like this:

```
jWic().fireAction(controlId, action, parameter);
```

Action URLs are created using the `createActionURL(..)` function. A velocity template for a simple anchor link looks like this:

```
<a href="$control.createActionURL("clicked", "")">$control.title</a>
```

The resulting HTML code would look like this:

```
<a href="javascript:jWic().fireAction('app.mylink', 'clicked', '')">lnk</a>
```

The JavaScript function saves these parameters in hidden fields and submits the form. The dispatcher will then get the specified control and call the controls `actionPerformed(String action, String acpara)` method. The default implementation of this methods will then try to find a method that starts with 'action' followed by the actionId and invokes it. The code for the above action would look like the following code:

```
public void actionClicked(String parameter) {
        notifySelectionListeners();
}
```

If the method does not expect parameter values, the String argument can be omitted:

```
public void actionClicked() {
        notifySelectionListeners();
}
```

If both methods exist, the method that takes a String argument is preferred.

If you do not wish to make use of this action delegation, you can override the actionPerformed method and handle the several actions on your own.

Sample:

```
public void actionPerformed(String actionId, String parameter) {
      if (actionId.equals("clicked")) {
            notifySelectionListeners();
      } else if (actionId.startsWith("..")) {
            ...
      }
}
```

2.2 Using File Upload

jWic offers the possibility to upload files to the server. Unlike normal text fields, file-fields only function if the data in the form is send by the client using a special MIME encoding, which is `multipart/form-data`. As this encoding type is more expensive to handle then the basic type, it is **not** used as default encoding type.

If you place a file-field on a form, the file is not submitted until you tell jWic to use multipart encoding. This is managed by the top-control, which should extend the Page class. The property `multipart` must be set to true, to be recognized when the page is rendered.

When a file is submitted, jWic will lookup the control that has created the field using the field mapping mechanism and call the controls handleFile(..) method. The control must implement the `IFileReciever` interface which defines this method.

Take a look at the `FileUploadControl` in the `de.jwic.controls` package. This will help you in most of the cases.

Note that if your application is using AJAX based rendering/updates, you don't have to take care about the multipart-property in the top control. When the client script library detects that a file-upload field contains data, the submit type is automatically changed to multipart and the data is submitted.

2.3 Double Post Protection

Communication between the server and the client (browser) is an asynchronous process, which has always been a problem in web application development. When the server's answer is delayed because of a slow connection or because he is busy, most end-users simply "do it again", submitting a request twice (or even more often). This can lead to wired results, like a message posted twice on a discussion board.

In a jWic application, it can be even more critical when the state of the application has changed during two submits. A good sample is an exit button. The first submit will exit the application, removing the button itself. The second

request will then result in an error, cause there is no button who could receive the "clicked" event.

To avoid these problems, jWic uses a ticket system called double post protection (DPP). The DPP system is based upon a number identifying each request. The number is stored in the `SessionContext` and is increased by one every time a request has been processed. The new number is returned to the client, send back by the client with the next request. Each ticket number is processed only once. If the ticket number does not equal the expected one, the request is ignored.

3 Renderers

3.1 Renderer Principles

Each visible control is rendered to HTML-code to become a part of the page. To decuple the controls from the rendering, jWic uses renderers that create HTML code based on the controls properties. A control defines the type of renderer that should be used by the `rendererId` property. The runtime will then map this `rendererId` to a configured renderer implementation.

The default rendererId is `jwic.renderer.default`, which is mapped by default to the basic velocity renderer. See 4.1 Server Configuration for details.

As a renderer is quit generic, each control comes with a `templateName` property so that the renderer can choose a template and merge it with the control. If the templateName property is not explicitly set, the classname is returned. In the end, it is up to the renderer implementation how to use the template name. Each renderer may have its own strategy how to load the right template.

3.2 Default Velocity Renderer

The default renderer implementation uses the Velocity engine to merge controls with templates. Velocity is an open source template engine from the Apache Software Foundation (See http://jakarta.apache.org/velocity/). It is used to reference java objects within a template and access properties or call methods. If you are new to velocity, we highly recommend reading the Velocity user-guide[2] to learn how templates are written.

3.2.1 Templates

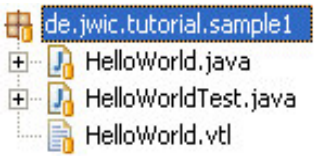
Each template used by a velocity renderer is accessed using a set of ResourceLoaders, which can load a template from various sources. These loaders are configured within a VelocityEngine. The default jWic setup configures two ResourceLoaders to load templates from the file system or the classpath. When a

[2] Velocity User-Guide: http://jakarta.apache.org/velocity/user-guide.html

template is not found in the file-system, the classpath is searched for the template.

The templateName property is therefore mapped to a template file by adding a `.vtl` extension to the name. If no template was found using that name, all dot (.) characters are replaced by a slash (/).

This mechanism makes it easy to assign a template to a control. Just create a .vtl file inside the same package, named like the class name of your control.



Storing templates in the same package as the class is the recommended way. It simplifies deploying your controls as they are bundled with your classes. When a user wants to override a template, they can extract the template from the jar-file and put it into the file system, using a filename like this:

```
de.jwic.tutorial.sample1.HelloWorld.vtl
```

The default place for these files is the `WEB-INF/jwic/ctrl_vtls` directory.

3.2.1.1 Velocity Context Objects

The Velocity context is a kind of map that holds the java objects that may be referenced from within the templates. The Velocity renderers offer the following objects to be used:

Id	Description
jwic	An instance of the JWicTools class, which provides general features like formatting.
control	The control being rendered.
insert	If the control being rendered is a ControlContainer, this object is used to insert a child at the specified position.

3.2.1.2 Samples

Read the property `userName` from the control being rendered:

```
Hello $control.userName, how are you?
```

Insert two child controls:

```
<table width="100%">
      <tr>
            <td valign="top">$insert.control("tree")</td>
            <td valign="top">$insert.control("table")</td>
      </tr>
</table>
```

Format the property `description` to be placed in an input field:

```
<textarea>$jwic.formatInp($control.description)</textarea>
```

3.2.2 OuterContainerRenderer

Most container controls use templates to arrange their child controls. To do this, containers are usually overridden and a new template is provided or the templateName property is changed. To give controls the possibility to provide a kind of border around the custom template, the OuterContainerRenderer can be used.

Controls using this renderer must implement the `IOuterLayout` interface, which defines a `getOuterTemplateName()` method. This method must return a valid template name to be used by the OuterContainerRenderer.

The velocity context for the outer template contains an object named `content`, that provides the method `render()` to render the inner content.

Sample:

```
<table border="1">
      <tr>
            <th>$control.title</th>
      </tr>
      <tr>
            <td>$content.render()</td>
      </tr>
</table>
```

To tell the runtime to use the OuterContainerRenderer instead of the default renderer, the property rendererId must be set to "jwic.renderer.OuterContainer". This value is also available as constant defined in the Control class.

Sample:

```
      public MyControl() {
            setRendererId(DEFAULT_OUTER_RENDERER);
      }
```

4 Configuration

jWic comes in a stand-alone distribution, requiring a minimal set of external libraries. This chapter describes how to configure this stand-alone version. In most real life projects, jWic would be integrated with other frameworks like Hibernate or the Springframework. Details on the integration into other frameworks are described in chapter 5: Integration.

4.1 Server Configuration

4.1.1 Dispatcher Servlet

jWic uses a servlet to dispatch incoming requests to the appropriate control(s) and force the rendering. This servlet is configured in the `web.xml` file of your web application, located in the WEB-INF directory. The servlet must be mapped to files with the extension `.xwic`.

Minimal configuration:

```
<servlet>
      <servlet-name>jwic</servlet-name>
      <servlet-class>de.jwic.web.DispatcherServlet</servlet-class>
</servlet>
<servlet-mapping>
      <servlet-name>jwic</servlet-name>
      <url-pattern>*.xwic</url-pattern>
</servlet-mapping>
```

The DispatcherServlet may be configured by certain init parameters as described below.

4.1.1.1 Log4j initialisation

The DispatcherServlet is able to initialize the Log4j system, which is used by jWic through the commons-logging API. To let the DispatcherServlet do so, you must specify the path to a property file that configures Log4j. The location is relative to your web application root directory. It is specified with the init-parameter `log4j-init-file`.

Sample:

```
<servlet>
      ..
      <init-param>
            <param-name>log4j-init-file</param-name>
            <param-value>WEB-INF/log4j.properties</param-value>
      </init-param>
</servlet>
```

As the Log4j system does not know the root path of your web application, a relative path to the log-file might end up in a location you didn't wanted. Due to this problem, Log4j offers variables within the configuration file that are replaced with a system property. The DispatcherServlet can store the root path of your web application in a system property for you, if you specify the name of the system property using the `setRootDir` init-parameter.

Sample (web.xml):

```
      <init-param>
            <param-name>setRootDir</param-name>
            <param-value>jwicweb.root</param-value>
      </init-param>
```

Sample (log4j.properties):

```
Log4j.appender.logfile.File=${jwicweb.root}/WEB-INF/jwic.log
```

Warning: System properties can be global over all your webapps. If you deploy multiple webapps on a server, each webapp must use its own property name.

4.1.1.2 Upload Limit

You can specify a limit in bytes for files uploaded to the server with the init-parameter `uploadlimit`. Files that exceed this limit are refused.

4.1.1.3 Interceptors

You can add interceptors to the DispatcherServlet which are invoked before and after an `HttpServletRequest` was handled. Interceptors must implement the `ServletInterceptor` interface. They are registered with the init-parameter servlet.interceptors, with the classnames of your interceptors separated by semicolon.

Sample:

```
      <init-param>
            <param-name>servlet.interceptors</param-name>
            <param-value>de.demo.Interceptor1;...</param-value>
      </init-param>
```

4.1.2 ClasspathResourceServlet

The ClasspathResourceServlet is used to reference resource files like images from the classpath. This is useful when you want to distribute a component as a single jar file that contains both templates and resources.

The servlet must be mapped to the path /cp/* to work.

4.1.3 Runtime Configuration

The jWic runtime and the available renderer are configured in the file `jwic-setup.xml` placed in the `WEB-INF` directory.

4.2 Application Configuration

Applications are configured in XML files with an .xwic extension.

Sample:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application SYSTEM "http://www.jwic.de/dtd/xwic-3.0.dtd">
<application>

      <name>Demo Application</name>
      <class>de.jwic.testapp.TestApplication</class>

</application>
```

For backward compatibility reasons, you can also specify the name of a control instead of an IApplication class:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application SYSTEM "http://www.jwic.de/dtd/xwic-3.0.dtd">
<application>

      <name>Demo Application</name>
      <rootcontrol
            name="root"
            classname="de.jwic.myapp.MyRootControl"
      />
</application>
```

There are additional configuration options available:

<serializable> values: true|false default: true

Specifies if the JWicRuntime is allowed to serialize the application to disk. If an application is mored to be not serializeable, the session remains loaded until the application is terminated or the session times out.

<singlesession> values: true|false default: false

Specifies if the application should be loaded only once per clientsession. If set to true, each try to launch another instance of the application will only reactivate an existing session if exists.

<useAjaxRendering> values: true|false default: true

Specifies if the application should be rendered using AJAX (Asynchronous JavaScript and XML) and DHTML. Usually jWic automatically tries to use ajax if it is supported by the browser. But it is possible to prevent it from being used at all, i.e. for testing purpose.

<requireauth> values: true|false default: false

Specifies if the user must be authenticated to launch the application. To use this feature, an `IAuthenticator` must be specified for the DispatcherServlet.

<property name="name">

Specifies a property. Properties are stored within the `IApplicationSetup` object and can be accessed like this:

```
String s = sessionContext.getApplicationSetup().getProperty("name");
```

5 Integration

5.1 Integrate into the Springframework

Users of the Springframework can integrate

6 Testing

A great thing about jWic is that you can write tests easily for your components and applications. You can even run JUnit tests without a requiring a framework that simulates a web server. All you need is to use the TestJWicRuntimeProvider to get your JWicRuntime instance and create a SessionContext based on your application setup.

6.1 Setup the Test-Environment

Before you can run tests, you must add a few configuration files that are used to setup the JWicRuntime and the loggers. Start by creating a folder "test" in your project and copy the files jwic-setup.xml and log4j.properties from your WEB-INF folder into it. You may now customize the logging settings if required.

Second step is to create a file named ''testenv.properties'' in the classpath of your tests. We think that it is a good practice to create a seperated source folder for the tests , i.e. named src_tests. There you can create the file with the following content:

```
# Test-Environment setup for the jWic Base Tests
rootpath=test
# The log4j.properties file
log4j.properties=test/log4j.properties
# The path where the runtime stores the serialized sessions
jwic-setup.xml=test/jwic-setup.xml
```

Now you are ready to write unit tests.

6.2 Create a ControlTest

Simply create a new class that extends de.jwic.test.ControlTestCase. This class is a TestCase object that can be run with JUnit. It does create a new application with a Page as root object. Then it calls the abstract method createControl(IContainer) where the control that is under test must be created. Furthermore it offers methods to manipulate the control to simulate a browser.

Here is the sample code for a test:

```
public class MyControlTest extends ControlTestCase {
 private MyControl myControl;
 public Control createControl(IControlContainer container) {
   myControl = new MyControl(container);
   return myControl;
 }

 public void testAlert() {
   myControl.alert();
   assertTrue(myControl.isAlerted());
 }
}
```

You can already run this test. In the above sample, the method under test is called directly as a controller would do. But it is also possible to simulate the access as it happens through the browser. With the method sendAction(..), you can simulate the click of an user on a link generated by the createActionURL method. The test code looks like this:

```
 public void testUserAction() {
   myControl.addElement("A");
   myControl.addElement("B");
```

```
    // simulate user action
    sendAction(myControl, "selectElement", "A");

    assertEquals("A", myControl.getSelectedKey());
  }
```

Controls that work with ''Field'' objects can also be tested using the updateField(..) methods provided by the ControlTestCase. Important in this case is that after all fields have been updated, the method updateDone() must be called. The code to test a calculator control could looks like this:

```
  public void testUserAction() {

    updateField(myControl, "value1", "2");
    updateField(myControl, "value2", "3");
    updateField(myControl, "op", "+");
    updateDone(); // process value changed listeners

    // simulate user action
    sendAction(myControl, "calc", "");

    assertEquals(5, myControl.getResult());
  }
```

You can test simple controls as well as complex control containers.

Document History

Date	Author	Changes
16-Apr-2005	Florian Lippisch	Document created
25-Apr-2005	Florian Lippisch	History added, File Upload, Renderer
26-Apr-2005	Florian Lippisch	Renderer Principles, Configuration
27-Apr-2005	Florian Lippisch	Testing
27-Apr-2005	Mark Frewin	Proof-read
27-May-2005	Florian Lippisch	Added Double Post Protection
29-Sep-2005	Florian Lippisch	Modified to reflect 3.0 changes
17-Nov-2005	Florian Lippisch	Updated Field Mapping to reflect 3.0 changes, Application Setup
14-Mar-2006	Florian Lippisch	Notes about FileUpload & AJAX update mechanism
10-May-2007	Aron Cotrau	Updated 2.1.4
11-May-2007	Florian Lippisch	Added ControlTesting guide