



java/j2ee Web Application Framework

Reference Documentation

Version 2.0.2

© Copyright 2005 – pol GmbH and Contributors

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Table of Content

Table of Content	2
Preface	3
1 jWic Fundamentals.....	4
1.1 What is a jWic Application?.....	4
1.2 What is a Control?.....	4
1.2.1 Control Lifecycle	4
1.2.2 Control Container	5
1.2.3 Control Name and Id.....	5
1.3 Events and Listeners	6
1.4 The SessionContext.....	6
1.4.1 Control Stack	7
1.5 The Page.....	7
2 The Web Interface	8
2.1 Abstracting jWic from the Web.....	8
2.1.1 Field Mapping.....	8
2.1.2 Action URLs	9
2.2 Using File Upload	9
3 Renderers.....	10
3.1 Renderer Principles.....	10
3.2 Default Velocity Renderer.....	10
3.2.1 Templates.....	11
3.2.2 OuterContainerRenderer.....	12
3.3 Skeleton Renderer.....	13
4 Configuration	13
4.1 Server Configuration.....	13
4.1.1 Dispatcher Servlet	13
4.1.2 ClasspathResourceServlet.....	14
4.1.3 Runtime Configuration.....	15
4.2 Application Configuration	15
5 Integration	15
5.1 Integrate into the Springframework.....	15
6 Testing.....	15
Document History	16

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 more to 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 its 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 implement the IControl interface or extend the abstract 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. Using a control before it has been added to a container is dangerous and should be avoided.

Sample:

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

From the control perspective, the constructor is invoked in line 1 and the `init()` method is invoked in the `addControl` method in line 2, after the control has been connected to the container.

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 omitted, 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

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

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 and application programmer can register for and then react to.

The following code fragment demonstrates how a listbox control listens to the `ElementSelected` event of a list:

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

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();
    addControl(dialog);
    dialog.addElementSelectedListener(this);
    getSessionContext().pushTopControl(dialog);
}
```

And this closes the dialog:

```
public void elementSelected(ElementSelectedEvent event) {
    fieldTo.setText(dialog.getSelectedAddress().getMail());
    getSessionContext().popTopControl();
    removeControl(dialog.getName()); // 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 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 does not extend the Page class, 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 and using the session ID which is stored in the form, the dispatcher retrieves 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 control that has created the field, using the `setFieldValue(..)` method. 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

Fields are elements on a form where a user can input data. Fields are defined in HTML using the `input`, `select` or `textarea` tags. Each field must have a unique name on the form to allow the dispatcher to map it against the control that has created the field. These field names are created by the `createFieldName(String)` function implemented in the Control superclass. This function is used by the renderer to create a field. The field name is created using the following pattern:

```
fld_<control id>.<fieldname>
```

Fields that do not start with the **fld_** prefix are ignored by the framework.

This velocity template is used to display a textbox:

```
<input
  name="$control.createFieldName("TEXT")"
  value="$control.getFieldValue("TEXT")">
```

The resulting HTML code could look like this:

```
<input name="fld_app.textbox.TEXT" value="Hello">
```

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

```
public String getText() {
    return getFieldValue("TEXT");
}
public void setText(String text) {
    setFieldValue("TEXT", 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 than one instance of the same control on your page.

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);
```

Like fieldnames, 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 aparam)` method.

The control can then act upon this action, i.e. fire an event to notify listeners that it has been clicked.

Sample:

```
public void actionPerformed(String actionId, String parameter) {
    notifySelectionListeners();
}
```

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 than 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 `IFileReceiver` 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.

3 Renderers

3.1 *Renderer Principles*

Each visible control is rendered to HTML-code to become a part of the page. To decouple 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 quite 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-guide¹ to learn how templates are written.

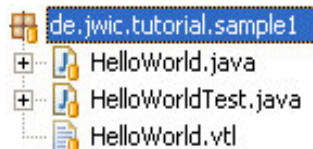
¹ Velocity User-Guide: <http://jakarta.apache.org/velocity/user-guide.html>

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 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 `.vtml` 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 `.vtml` 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.vtml
```

The default place for these files is the `WEB-INF/jwic/ctrl_vtmls` 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);
}
```

It is highly recommend that you set this property in the constructor, not in the `init` method. Otherwise users of your control who instantiate it using an IoC² container would not be able to set the `rendererId` within their container as it gets overridden at the moment the control is added to a container, calling the `init()` method.

² Inversion of Control – see <http://www.martinfowler.com/articles/injection.html> for details

3.3 Skeleton Renderer

Earlier versions of jWic used a template engine called SkeletonProcessor. To adapt those older jWic components, the SkeletonRenderer was implemented. A SkeletonRenderer can only handle controls that implement the `ISelfRenderer` interface and render themselves, writing HTML code into the `UIStream` object.

The skeleton technology is quite old now and it is not recommended using it.

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 required by jWic. 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:

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

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.dtd">
<application>

    <name>Demo Application</name>
    <serializable>true</serializable>
    <singlesession>true</singlesession>
    <requireauth>false</requireauth>
    <rootcontrol
        name="root"
        classname="de.jwic.testapp.TestApplication"
    />
    <property name="customProperty">theValue</property>

</application>
```

5 Integration

5.1 Integrate into the Springframework

The integration with the Springframework has been temporary removed in this version. It will be re-implemented in the next version.

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.

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