


Space Details

Key:	WW
Name:	WebWork 2
Description:	
Creator (Creation Date):	plightbo (Apr 18, 2004)
Last Modifier (Mod. Date):	plightbo (May 18, 2004)

Available Pages

- WebWork 
 - Documentation
 - Articles
 - Bug tracker, wiki
 - Comparison to Struts
 - Configuration
 - Interceptors
 - Chaining Interceptor
 - Namespaces
 - Packages
 - Reloading configuration
 - Results
 - Default results
 - Global results
 - velocity.properties
 - web.xml
 - webwork-default.xml
 - webwork.properties
 - xwork.xml
 - Dependencies
 - Deployment Notes
 - FAQ
 - Getting Started
 - Hibernate
 - Internationalization
 - Inversion of Control
 - Components
 - IoC Configuration
 - IoC Overview
 - Xwork's Component Architecture
 - JSP Tags
 - Non-UI Tags
 - UI Tags

- Templates
 - Themes
- JUnit
- Mailing Lists
- OGNL
 - OGNL Basics
- Pico
- Projects Using WebWork
- Quartz
- QuickStart
- Release Notes - 2.1
- Release Notes - 2.1.1
- Result Types
- SiteMesh
- Spring
- TutorialEnd
- TutorialExamples
- TutorialLesson01
- TutorialLesson02
- TutorialLesson03
- TutorialLesson04
 - TutorialLesson04-01
 - TutorialLesson04-01-01
 - TutorialLesson04-02
 - TutorialLesson04-03
- TutorialLesson05
- Type Conversion
- Upgrading from 1.4
- Upgrading from 2.0
- Upgrading from 2.1
- Validation
 - Client-Side Validation
 - Simple validators
 - Validation Examples
 - Visitor validation
- WebWork Community
- What is WebWork
- Examples
 - Chat Application
- IDEA Plugin
- Press Releases
 - 2.1 Press Release
 - 2.1.1 Press Release
 - About
- Testimonials

- WebWork Team

How to use the Chaining Interceptor

The following code snippet shows how interceptor stacks work for chaining. If someone wants to post xwork.xml (and more complex) examples it would be appreciated 😊

```
Interceptors-stack A before
Action A
Interceptor-stack B before
Action B
Action B result
Interceptor-stack B after
Interceptor-stack A after
```

Interceptors that wrap Chained Actions

Sometimes you may want to have an interceptor that wraps a number of chained actions (and is included in the Interceptor stack for each), but is only invoked at the start and end of the chain. For example, an Interceptor that manages a Hibernate Session / Transaction. Here is an example from my 'teach-myself-webwork-and-hibernate project named 'cash' after Johnny Cash.

```
<interceptor name="hibernate" class="cash.interceptor.HibernateInterceptor"/>
  <interceptor name="login" class="cash.interceptor.LoginInterceptor"/>

  <interceptor-stack name="cashDefaultStack">
    <interceptor-ref name="defaultStack"/>
    <interceptor-ref name="component"/>
    <interceptor-ref name="hibernate"/>
    <interceptor-ref name="login"/>
  </interceptor-stack>
  <interceptor-stack name="cashValidationWorkflowStack">
    <interceptor-ref name="cashDefaultStack"/>
    <interceptor-ref name="validation"/>
    <interceptor-ref name="workflow"/>
  </interceptor-stack>
</interceptors>

<default-interceptor-ref name="cashDefaultStack"/>

<action name="list" class="cash.action.SelectUserAction">
  <result name="success" type="dispatcher">list.vm</result>
</action>
```

```
<action name="edit" class="cash.action.EditAction">
  <result name="success" type="chain">list</result>
  <result name="input" type="dispatcher">edit.vm</result>
  <interceptor-ref name="cashValidationWorkflowStack"/>
</action>
```

In this example, after editing a user, the EditAction is chained to the ListAction to display a list of all users to the screen.

We want the following

```
EditActionInterceptorStack - before
  EditAction
  ListActionInterceptorStack (except for Hibernate) - before
    ListAction
  ListActionInterceptorStack (except for Hibernate) - end
EditActionInterceptorStack - end
```

But instead we get:

```
EditActionInterceptorStack - before
  EditAction
  ListActionInterceptorStack (including Hibernate) - before
    ListAction
  ListActionInterceptorStack (including Hibernate) - end
EditActionInterceptorStack - end ERROR!  Hibernate Session / Transaction is
already closed!!!
```

The way to get the desired behaviour is to either not use a chained action (and incorporate the ListAction logic into EditAction, or to make the HibernateInterceptor smart enough to handle chained actions.

The HibernateInterceptor can use a ThreadLocal to hold state and detect if it is inside a chain. When it detects this, it can do nothing.

```
package cash.interceptor;

import net.sf.hibernate.HibernateException;
import net.sf.hibernate.Transaction;

import org.apache.log4j.Logger;

import com.opensymphony.xwork.Action;
import com.opensymphony.xwork.ActionInvocation;
import com.opensymphony.xwork.interceptor.Interceptor;

import cash.action.HibernateAction;
import cash.util.HibernateUtil;

/**
 * @author Gavin King
 * @author Joel Hockey
 * @version $Id: $
```

```

*/
public class HibernateInterceptor implements Interceptor {
    private static final Logger LOG = Logger.getLogger(HibernateInterceptor.class);

    private static ThreadLocal s_threadLocal = new ThreadLocal();

    /** destroy */
    public void destroy() { }

    /** init */
    public void init() { }

    /** implement intercept */
    public String intercept(ActionInvocation invocation) throws Exception {
        LOG.debug("HibernateInterceptor called");

        Action action = invocation.getAction();
        if (!(action instanceof HibernateAction)) { return invocation.invoke(); }

        // continue with HibernateAction
        HibernateAction ha = (HibernateAction)action;

        // if this interceptor is being chained, then transaction will already
        exist// in that case, we should let the outer interceptor dispose of the
        sessionboolean inChainedAction = true;
        Transaction transaction = (Transaction)s_threadLocal.get();
        if (transaction == null) {
            inChainedAction = false;
            transaction = HibernateUtil.currentSession().beginTransaction();
            s_threadLocal.set(transaction);
        }

        boolean rollback = false;

        try {

            return invocation.invoke();
        } catch (Exception e) {
            // Note that all the cleanup is done// after the view is rendered, so
            we// have an open session in the view

            rollback = true;
            if (e instanceof HibernateException) {
                LOG.error("HibernateException in execute()", e);
                return HibernateAction.DBERROR;
            } else {
                LOG.error("Exception in execute()", e);
                throw e;
            }
        } finally {
            try {
                if (!inChainedAction) {
                    s_threadLocal.set(null);
                    disposeSession(transaction, ha.getRollback() || rollback);
                }
            } catch (HibernateException e) {
                LOG.error("HibernateException in dispose()", e);
                return HibernateAction.DBERROR;
            }
        }
    }
}

```

```

    /** dispose of session */
    public void disposeSession(Transaction transaction, boolean rollback) throws
    HibernateException {
        LOG.debug("disposing");

        if (!HibernateUtil.currentSession().isConnected()) {
            LOG.debug("Session has already been disposed of - this will happen in a
            chained action");
            return;
        }

        try {
            if (transaction != null) {
                if (rollback) {
                    LOG.debug("rolling back");
                    transaction.rollback();
                } else {
                    LOG.debug("committing");
                    transaction.commit();
                }
            }
        } catch (HibernateException e) {
            LOG.error("error during commit/rollback", e);
            if (!rollback && transaction != null) {
                LOG.error("rolling back affter previous attempt to commit");
                transaction.rollback();
            }
            throw e;
        } finally {
            HibernateUtil.closeSession();
        }
    }
}

```

For more information, also see OS:Webwork - Why would I use Action Chaining?

End of Tutorial

This is the end of the WebWork Tutorial. If you followed the tutorial and understood the concepts that were explained and exemplified, you're ready to start developing your own WebWork application. If you want go deeper into WebWork's details, I suggest you read the Reference Guide at the [Documentation Page](#).

[Previous Lesson](#)

Reloading configuration

This page last changed on May 14, 2004 by [mgreer](#).

Webwork allows for dynamic reloading of xml configuration file (ie, reloading actions.xml).

This allows you to reconfigure your action mapping during development. There may be a slight performance penalty, so this is not recommended for production use.

In order to enable this feature, add the following to your webwork.properties file:

```
webwork.configuration.xml.reload=true
```

webwork-default.xml

This page last changed on May 14, 2004 by [mgreer](#).

A base configuration file named `webwork-default.xml` is included in the `webwork` jar file. This file may be included at the top of your `xwork.xml` file to include the standard configuration settings without having to copy them, like so:

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork
1.0//EN" "http://www.opensymphony.com/xwork/xwork-1.0.dtd"><xwork><include
file="webwork-default.xml"/><package name="default" extends="webwork-default">
...
</package></xwork>
```

The contents of `webwork-default.xml` as of 2.1 are here:

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork
1.0//EN" "http://www.opensymphony.com/xwork/xwork-1.0.dtd"><xwork><package
name="webwork-default"><result-types><result-type name="dispatcher"
class="com.opensymphony.webwork.dispatcher.ServletDispatcherResult"
default="true"/><result-type name="redirect"
class="com.opensymphony.webwork.dispatcher.ServletRedirectResult"/><result-type
name="velocity"
class="com.opensymphony.webwork.dispatcher.VelocityResult"/><result-type
name="chain" class="com.opensymphony.xwork.ActionChainResult"/><result-type
name="xslt" class="com.opensymphony.webwork.views.xslt.XSLTResult"/><result-type
name="jasper"
class="com.opensymphony.webwork.views.jasperreports.JasperReportsResult"/><result-type
name="freemarker"
class="com.opensymphony.webwork.views.freemarker.FreemarkerResult"/></result-types><interceptors>
name="timer"
class="com.opensymphony.xwork.interceptor.TimerInterceptor"/><interceptor
name="logger"
class="com.opensymphony.xwork.interceptor.LoggingInterceptor"/><interceptor
name="chain"
class="com.opensymphony.xwork.interceptor.ChainingInterceptor"/><interceptor
name="static-params"
class="com.opensymphony.xwork.interceptor.StaticParametersInterceptor"/><interceptor
name="params"
class="com.opensymphony.xwork.interceptor.ParametersInterceptor"/><interceptor
name="model-driven"
class="com.opensymphony.xwork.interceptor.ModelDrivenInterceptor"/><interceptor
name="component"
class="com.opensymphony.xwork.interceptor.component.ComponentInterceptor"/><interceptor
name="token"
class="com.opensymphony.webwork.interceptor.TokenInterceptor"/><interceptor
name="token-session"
class="com.opensymphony.webwork.interceptor.TokenSessionStoreInterceptor"/><interceptor
name="validation"
class="com.opensymphony.xwork.validator.ValidationInterceptor"/><interceptor
name="workflow"
class="com.opensymphony.xwork.interceptor.DefaultWorkflowInterceptor"/><interceptor
name="servlet-config"
class="com.opensymphony.webwork.interceptor.ServletConfigInterceptor"/><interceptor
name="prepare"
```

```
<interceptor name="conversionError"
class="com.opensymphony.webwork.interceptor.WebWorkConversionErrorInterceptor"/><interceptor-stack
name="defaultStack"><interceptor-ref name="static-params"/><interceptor-ref
name="params"/><interceptor-ref
name="conversionError"/></interceptor-stack><interceptor-stack
name="validationWorkflowStack"><interceptor-ref
name="defaultStack"/><interceptor-ref name="validation"/><interceptor-ref
name="workflow"/></interceptor-stack></interceptors></package></xwork>
```

This file defines all of the default bundled results and interceptors and two interceptor stacks, defaultStack and validationWorkflowStack, which you can use either as-is or as a basis for your own application-specific interceptor stacks. Notice the name of the package is "webwork-default".

Mailing Lists

This page last changed on May 19, 2004 by [ehauser](#).

The Webwork mailing list is open to developers and end users. The list is hosted at <http://webwork.dev.java.net/servlets/ProjectMailingListList> in message-by-message and digest format. The archive of this list is also available from this page, so please search past postings before posting a message.

Bug tracker, wiki

This page last changed on May 19, 2004 by [ehauser](#).

The bug tracking system for both Webwork and Xwork (as well as all other Opensymphony projects) can be found at <http://jira.opensymphony.com>.

The Webwork wiki contains a wealth of knowledge about the project including example code, a cookbook, and general discussion about the framework. The wiki can be found at <http://wiki.opensymphony.com/display/WW/WebWork>.

Articles

This page last changed on May 20, 2004 by [plightbo](#).

- Book: [Java Open Source Programming : with XDoclet, JUnit, WebWork, Hibernate](#)
- A good summary of what's new in WebWork 2.0: <http://blogs.atlassian.com/rebelutionary/archives/000085.html>
- [Writeup of Mike's Talk at TSS on WebWork2](#) (7/03)
- Mike's PPT Presentation from TSS Symposium: <http://blogs.atlassian.com/rebelutionary/archives/000200.html>
- Some Japanese translated information (including the presentation above): <http://www.mobster.jp/webwork/>
- Another PPT presentation from Rick Salsa, Groove Systems: <http://www.groovesystems.com/training/java4/webwork.ppt>
- Building with WebWork on TheServerSide (11/03): <http://www.theserverside.com/resources/article.jsp?l=WebWork2>
- Jason Carreira's slides and example code (4/04): [webwork-talk.zip](#)
- A brazilian portuguese tutorial teaching the main concepts and features of Webwork (14/04): [tutorial](#)

Hibernate

This page last changed on Jun 18, 2004 by [plightbo](#).

There's nothing more that you have to do use Hibernate with WebWork than with other Web framework. Just setup Hibernate according to the <http://www.hibernate.org/5.html>. However, there're a number of good patterns that people have used successfully in the following projects:

- AdminApp <http://www.hibernate.org/159.html#a5>
- Petsoar <http://www.wiley.com/legacy/compbooks/walnes>

Overview

Result Types are classes that determine what happens after an Action executes and a Result is returned. Developers are free to create their own Result Types according to the needs of their application or environment. In WebWork 2 for example, Servlet and Velocity Result Types have been created to handle rendering views in web applications.

Note: All built in webwork result types implement the `com.opensymphony.xwork.Result` interface, which represents a generic interface for all action execution results, whether that be displaying a webpage, generating an email, sending a JMS message, etc.

Result types define classes and map them to names to be referred in the action configuration results. This serves as a shorthand name-value pair for these classes.

```
<!-- parts of xwork.xml -->
....

<result-types>
  <result-type name="dispatcher"
class="com.opensymphony.webwork.dispatcher.ServletDispatcherResult" default="true"/>
  <result-type name="redirect"
class="com.opensymphony.webwork.dispatcher.ServletRedirectResult"/>
  <result-type name="velocity"
class="com.opensymphony.webwork.dispatcher.VelocityResult"/>
  <result-type name="chain" class="com.opensymphony.xwork.ActionChainResult"/>
  <result-type name="xslt" class="com.opensymphony.webwork.views.xslt.XSLTResult"/>
  <result-type name="jasper"
class="com.opensymphony.webwork.views.jasperreports.JasperReportsResult"/>
  <result-type name="freemarker"
class="com.opensymphony.webwork.views.freemarker.FreemarkerResult"/>
</result-types>

....
<!-- this action uses the result type dispatcher which is defined above -->

<action name="bar" class="myPackage.barAction">
  <result name="success" type="dispatcher">
    <param name="location">foo.jsp</param>
  </result>
  <result name="error" type="dispatcher">
    <param name="location">error.jsp</param>
  </result>
</action>

....
```


Result Types

Webwork provides several implementations of the `com.opensymphony.xwork.Result` interface to make web-based interactions with your actions simple. These result types include:

Result Type	name	class
#Dispatcher	dispatcher	<code>com.opensymphony.webwork.dispatcher</code>
#Redirect	redirect	<code>com.opensymphony.webwork.dispatcher</code>
#Action Chaining	chain	<code>com.opensymphony.xwork.ActionChain</code>
#Velocity	velocity	<code>com.opensymphony.webwork.dispatcher</code>
#FreeMarker	freemarker	<code>com.opensymphony.webwork.views.freemarker</code>
#JasperReports	jasper	<code>com.opensymphony.webwork.views.jasper</code>
#XML/XSL	xslt	<code>com.opensymphony.webwork.views.xslt</code>
#HTTPHeader		<code>com.opensymphony.webwork.dispatcher</code>

Results are specified in a xwork xml config file(xwork.xml) nested inside `<action>`. If the `location` param is the only param being specified in the result tag, you can simplify it as follows:

```
<action name="bar" class="myPackage.barAction">
  <result name="success" type="dispatcher">
    <param name="location">foo.jsp</param>
  </result>
</action>
```

or simplified

```
<action name="bar" class="myPackage.barAction">
  <result name="success" type="dispatcher">foo.jsp</result>
</action>
```

NOTE: The Parse attribute enables the location element to be parsed for expressions. An example of how this could be useful:

```
<result name=#success# type=#redirect#>/displayCart.action?userId=${userId}</result>
```

NOTE: You can also specify global-results to use with multiple actions. This can save time from having to add the same result to many different actions. For more information on result tags and global-results, see [Results](#) section.

Dispatcher

Includes or forwards to a view (usually a jsp). Behind the scenes WebWork will use a RequestDispatcher, where the target servlet/JSP receives the same request/response objects as the original servlet/JSP. Therefore, you can pass data between them using request.setAttribute() – the WebWork action is available.

Parameters	Required	Description
location	yes	the location to go to after execution (ex. jsp)
parse	no	true by default. If set to false, the location param will not be parsed for Ognl expressions

```
<result name="success" type="dispatcher">
  <param name="location">foo.jsp</param>
</result>
```

Redirect

The response is told to redirect the browser to the specified location (a new request from the client). The consequence of doing this means that the action (action instance, action errors, field errors, etc) that was just executed is lost and no longer available. This is because actions are built on a single-thread model. The only way to pass data is through the session or with web parameters (url?name=value) which can be OGNL expressions.

Parameters	Required	Description
location	yes	the location to go to after execution
parse	no	true by default. If set to false, the location param will not be parsed for Ognl expressions

```
<result name="success" type="redirect">
  <param name="location">foo.jsp</param>
  <param name="parse">false</param>
</result>
```

Action Chaining

A special kind of view that invokes GenericDispatch (using the previously existing ActionContext) and executes another action. This is useful if you need to execute one

action immediately after another.

Parameters	Required	Description	
actionName	yes	the name of the action that will be chained to	
namespace	no	sets the namespace of the Action that we're chaining to. If namespace is null, this defaults to the current namespace.	<pre><result name="success" type="chain"> <param name="actionName">bar</param> <param name="namespace">/foo</param> </result></pre> <p>invokes this</p> <pre><action name="bar" class="myPackage.barAction"> ... </action></pre>

Velocity

This result mocks a JSP execution environment and then displays a Velocity template that will be streamed directly to the servlet output.

Parameters	Required	Description
location	yes	the location to go to after execution
parse	no	true by default. If set to false, the location param will not be parsed for Ognl expressions

```
<result name="success" type="velocity">
  <param name="location">foo.vm</param>
</result>
```

FreeMarker

Parameters	Required	Description	
location	yes	the location to go to after execution	
parse	no	true by default. If set to false, the location param will not be parsed for Ognl expressions	
contentType	no	defaults to "text/html" unless specified	<pre><result name= "success" type= "freemarker">foo.ftl</result></pre>

JasperReports

Generates a JasperReports report using the specified format or PDF if no format is specified.

Parameters	Required	Description
location	yes	the location to go to after execution
parse	no	true by default. If set to false, the location param will not be parsed for Ognl expressions
dataSource	yes	the Ognl expression used to retrieve the datasource from the value stack (usually a List)
format	no	the format in which the report should be generated, defaults to pdf

```
<result name="success" type="jasper">
  <param name="location">foo.jasper</param>
  <param name="dataSource">mySource</param>
  <param name="format">CSV</param>
</result>
```

or for pdf

```
<result name="success" type="jasper">
  <param name="location">foo.jasper</param>
  <param name="dataSource">mySource</param>
</result>
```

XML/XSL

Interfaces with xml transformations.

Parameters	Required	Description
location	yes	the location to go to after execution
parse	no	Defaults to false. If set to true, the location will be parsed for Ognl expressions

```
<result name="success" type="xslt">foo.xslt</result>
```

HttpHeader

A custom Result type for evaluating HTTP headers against the ValueStack.

Parameters	Required	Description
status	no	the http servlet response status code that should be set on a response
parse	no	true by default. If set to false, the headers param will not be parsed for Ognl expressions
headers	no	header values

Example:

```
<result name="success" type="header"><param name="status">204</param><param
name="headers.a">a custom header value</param><param name="headers.b">another custom
header value</param></result>
```

This page last changed on Jun 17, 2004 by [casey](#).

WebWork's (WW) community is made up of users, developers, and bystanders. Each one is important in making Webwork a success. Users daily use the framework and provide valuable feedback on usability, bugs, features, etc. Developers contribute their time writing a framework others may use. Bystanders look on watching the work unfold. Some of them will soon become users or developers.

Why we do what we do?

This quote taken from <http://opensource.org> best describes why we spend our time working for free.

"The basic idea behind open source is very simple: When programmers can read, redistribute, and modify the source code for a piece of software, the software evolves. People improve it, people adapt it, people fix bugs. And this can happen at a speed that, if one is used to the slow pace of conventional software development, seems astonishing."

"We in the open source community have learned that this rapid evolutionary process produces better software than the traditional closed model, in which only a very few programmers can see the source and everybody else must blindly use an opaque block of bits."

[Meet the team that contributes to Webwork and Open Symphony](#)

Configuring the web.xml file is for the most part a pretty straight forward exercise.

Configuring the ServletDispatcher

The most common entry point for calling actions in your application will be calling an URL directly or having the user submit a form to an action. The following is an example of how to configure the ServletDispatcher to process actions.

```
<!-- This entry is required to have the framework process calls to WebWork actions
--><servlet><servlet-name>webworkDispatcher</servlet-name><servlet-class>com.opensymphony.webwork
```

NOTE: The above configuration assumes that actions will use the ".action" extension. If you want your actions to use a different extension, change the url-pattern element to the desired extension and update your calling code to specify that extension.

Configuring CoolUriServletDispatcher (optional)

A custom servlet dispatcher that maps servlet paths to actions. This can be used instead of the above ServletDispatcher. The format is the following:

```
http://HOST/ACTION_NAME/PARAM_NAME1/PARAM_VALUE1/PARAM_NAME2/PARAM_VALUE2
```

You can have as many parameters you'd like to use. Alternatively the URL can be shortened to the following:

```
http://HOST/ACTION_NAME/PARAM_VALUE1/PARAM_NAME2/PARAM_VALUE2
```

This is the same as:

```
http://HOST/ACTION_NAME/ACTION_NAME/PARAM_VALUE1/PARAM_NAME2/PARAM_VALUE2
```

Suppose for example we would like to display some articles by id at using the following URL sheme:

```
http://HOST/article/ID
```

All we would have to do is to map the `/article/*` to this servlet and declare in WebWork an action named `article`. This action would set its `article` parameter ID:

```
<servlet><servlet-name>coolDispatcher</servlet-name><servlet-class>com.opensymphony.webwork.dispatcher
```

Configuring Velocity Support (optional)

WebWork uses Velocity as the underlying template system for the custom JSP tag libraries. However, it does not require the `WebWorkVelocityServlet` to be configured. You only need to do that if you want to invoke velocity templates directly or from "dispatcher" result type. If so, you need to add the following:

```
<servlet><servlet-name>velocity</servlet-name><servlet-class>com.opensymphony.webwork.views.velocity
```

Configuring tag libraries (optional)

The tag libraries are an optional part of the framework that provides a number of benefits when working with forms and actions. See the [JSP Tags](#) reference for more information. To use them, you need to specify the location of the `tld` (Tag Library Descriptor):

```
<taglib><taglib-uri>webwork</taglib-uri><taglib-location>/WEB-INF/lib/webwork-2.1.jar</taglib-location>
```

If your application server doesn't support taglib inside a jar, you will need to extract the `webwork.tld`, copy it to `/WEB-INF` and change the above to:

```
<taglib><taglib-uri>webwork</taglib-uri><taglib-location>/WEB-INF/webwork.tld</taglib-location>
```

NOTE: `webwork.tld` can be placed anywhere inside your webapp directory.

Configuring Freemarker support (optional)

Add the following:

```
<servlet><servlet-name>freemarker</servlet-name><servlet-class>com.opensymphony.webwork.views.freemarker
```


and

```
<servlet-mapping><servlet-name>freemarker</servlet-name><url-pattern>*.ftl</url-pattern></servlet-mapping>
```

Configuring IoC / LifeCycle (optional)

See [IoC Configuration](#)

Themes

This page last changed on Jun 16, 2004 by [casey](#).

A Theme is a set of [templates](#) used to customize web page development with UI tags. They provide a powerful mechanism to help web developers spice up the UI with a mixture of styles (colors, fonts, etc). For example, you may want half your form textfields to have a blue background and half a white background. A couple of notes:

- Webwork comes with 2 pre-defined themes; [#simple](#) and [#xhtml](#) (default). The default location webwork looks for themes in your web application is `/template`. The default theme is `xhtml`. Default, meaning that it will be used if you don't specify a theme attribute in your UI tag. Note: The default values can be overridden in your [webwork.properties](#) file with `webwork.ui.theme=` and `webwork.ui.templateDir=`.
- [Custom themes](#) can also be created to tailor your own needs. We recommend you consult the pre-defined templates as a starting point before you create your own.
- Every time a UI tag is used, the tag is rendered into html by referencing a [template](#). So they play a key role in how fields look and are positioned in a page. Note: webwork uses only velocity templates (*.vm) to render both velocity and webwork tags, while ww1 defined templates in both jsp and vm.

Note: Before moving forward, it is recommended that you review how the webwork template system works. (see [Templates](#))

xhtml

xhtml comes configured as the default theme for Webwork. It extends the simple theme providing built in functionality for error reporting, table positioning, and labeling. Lets look at one of the most common UI tags used, `textfield`, and show the proper way to write your views with the xhtml theme.

As you may already know, the default UI template used for the `textfield` tag is `text.vm` located under the directory `/template/xhtmll`.

```
#*
-- text.vm
*#
## notice the re-use of the simple theme template text.vm
#parse("/template/xhtmll/controlheader.vm")
#parse("/template/simple/text.vm")
#parse("/template/xhtmll/controlfooter.vm")
```

When this template is loaded, it will first parse and render the /template/xhtml/controlheader.vm. Within controlheader.vm you will notice functionality for error reporting, labeling and table positioning. If ActionSupport is returning with some errors, they are rendered into html using this this template. Also you will notice how it grabs the parameter.label value and positions it with the textfield using the table elements tr and td.

```

*#
  -- controlheader.vm
*#
## Only show message if errors are available.
## This will be done if ActionSupport is used.

#if( $fieldErrors.get($parameters.name) )
  #set ( $hasFieldErrors = $fieldErrors.get($parameters.name))
  #foreach ( $error in $fieldErrors.get($parameters.name))
    <tr>
      #if ( $parameters.labelposition == 'top')
        <td align="left" valign="top" colspan="2">
      #else
        <td align="center" valign="top" colspan="2">
      #end
      <span class="errorMessage">${!webwork.htmlEncode($error)}</span>
    </td>
  </tr>
#end
#end

## Provides alignment behavior with table tags
## if the label position is top,
## then give the label it's own row in the table
## otherwise, display the label to the left on same row

<tr>
  #if ( $parameters.labelposition == 'top')
    <td align="left" valign="top" colspan="2">
  #else
    <td align="right" valign="top">
  #end

  #if ( $hasFieldErrors)
    <span class="errorLabel">
  #else
    <span class="label">
  #end

## If you want to mark required form fields with an asterisk,
## you can set the required attribute
## Ex. <ui:textfield label="mylabel" name="myname" required="true" />
  #if ( $parameters.label)
    #if ( $parameters.required) <span class="required">*</span> #end
    ${!webwork.htmlEncode($parameters.label)}:
  #end
  </span>
</td>

## add the extra row
#if ( $parameters.labelposition == 'top')
</tr>
<tr>
```

```
#end
    <td>
```

The next template being parsed in `/template/xhtml/text.vm` is `/template/simple/text.vm`. Here you see the actual html input text tag being rendered and how the parameters are passed in.

```

##
-- text.vm
--
-- Required Parameters:
-- * label      - The description that will be used to identify the control.
-- * name       - The name of the attribute to put and pull the result from.
--               Equates to the NAME parameter of the HTML INPUT tag.
--
-- Optional Parameters:
-- * labelposition - determines where the label will be placed in relation
--                   to the control. Default is to the left of the control.
-- * size         - SIZE parameter of the HTML INPUT tag.
-- * maxlength    - MAXLENGTH parameter of the HTML INPUT tag.
-- * disabled     - DISABLED parameter of the HTML INPUT tag.
-- * readonly     - READONLY parameter of the HTML INPUT tag.
-- * onkeyup      - onkeyup parameter of the HTML INPUT tag.
-- * tabindex     - tabindex parameter of the HTML INPUT tag.
-- * onchange     - onchange parameter of the HTML INPUT tag.
--
*#
<input type="text"
                                name="$_webwork.htmlEncode($parameters.name)"
#if ($parameters.size)         size="$_webwork.htmlEncode($parameters.size)"
#end
#if ($parameters.maxlength)
maxlength="$_webwork.htmlEncode($parameters.maxlength)" #end
#if ($parameters.nameValue)
value="$_webwork.htmlEncode($parameters.nameValue)" #end
#if ($parameters.disabled == true) disabled="disabled"
#end
#if ($parameters.readonly)      readonly="readonly"
#end
#if ($parameters.onkeyup)
onkeyup="$_webwork.htmlEncode($parameters.onkeyup)" #end
#if ($parameters.tabindex)
tabindex="$_webwork.htmlEncode($parameters.tabindex)" #end
#if ($parameters.onchange)
onchange="$_webwork.htmlEncode($parameters.onchange)" #end
#if ($parameters.id)           id="$_webwork.htmlEncode($parameters.id)"
#end
#if ($parameters.cssClass)
class="$_webwork.htmlEncode($parameters.cssClass)" #end
#if ($parameters.cssStyle)
style="$_webwork.htmlEncode($parameters.cssStyle)" #end
/>

```

And finally, the controlfooter.vm is parsed to close up the td and tr tags that were previously opened in controlheader.vm

```

#*
-- controlheader.vm

```

```
*#

    </td>
</tr>
```

In our view, since the tr and td elements are already created for us, we can simply wrap them with table elements. For the sake of learning, we will just use normal html table objects, but feel free to look into how the table.vm tag gets rendered and possibly use that.

```
<%@ taglib uri="webwork" prefix="ui" %>
<link rel = "stylesheet" type="text/css" href="template/xhtml/styles.css"
title="Style">
<html>
<head><title>JSP PAGE</title></head>
<body>
<form>
  <table>
    <!-- we can set the required attribute to trueif we want to
         display and asterisk next to required form fields
    -->

    <ui:textfield label="'Username'" required="'true'" name="'user'" />
    <ui:textfield label="'Email'" name="'email'" />
  </table>
</form>
</body>
</html>
```

```
<link rel = "stylesheet" type="text/css" href="template/xhtml/styles.css"
title="Style">
<html>
<head><title>VM PAGE</title></head>
<body>
<form>
  <table>
    #tag( TextField "label='Username'" "name='user'" )<br>
    #tag( TextField "label='Email'" "name='email'" )<br>
  </table>
</form>
</body>
</html>
```

simple

The `simple` theme provides no additional functionality from HTML tags (similar to struts). This theme is considered the low end of the structure and can be re-used (extended) like `xhtml` to add additional functionality or behavior. You can easily create your own theme and extend this one to create complex pages that fit your own needs. To use the pre-defined theme `simple`

```
<%@ taglib uri="webwork" prefix="ui" %>
<link rel="stylesheet" type="text/css" href="template/xhtml/styles.css"
title="Style">
<html>
<head><title>JSP PAGE</title></head>
<body>
  <form>
    <ui:label name="'userlabel'" label="'user'" theme="'simple'"/>
      <ui:textfield name="'user'" theme="'simple'"/>

    <ui:label name="'emaillabel'" label="'user'" theme="'simple'"/>
      <ui:textfield name="'email'" theme="'simple'"/>
  </form>
</body>
</html>
```

```
<link rel="stylesheet" type="text/css" href="template/xhtml/styles.css"
title="Style">
<html>
<head><title>VM PAGE</title></head>
<body>
  <form>
    #tag( Label "name='userlabel'" "label='user'" "theme='simple'" )
    #tag( TextField "name='user'" "theme='simple'" )<br>

    #tag( Label "name='emaillabel'" "label='email'" "theme='simple'" )
    #tag( TextField "name='email'" "theme='simple'" )<br>
  </form>
</body>
</html>
```

Creating your own theme

Creating themes is quite simple and can save valuable time enabling you to minimize UI code when it comes to creating complex UI pages. It is recommended you understand webwork templates before you continue (see [Templates](#)). The steps required to use a theme.

1. Define a name for your theme by creating a subdirectory under /template directory. The name of the subdirectory you create will be the same as the value you specify in your UI tag theme attribute.

```
/template/myTheme
```

```
<ui:textfield label="'foo'" name="'bar'" theme="'myTheme'" />
```

2. Create a velocity template for every UI tag that you want to use with your theme. For example, if you have forms with only textfields and nothing else, then all you need in your subdirectory is a text.vm template. Note: if you create a text.vm and reference another template with the parse tag, then you must make sure these templates are defined as well(ex. controlheader.vm).

3. If you want your new theme to be the default so you don't have to specify the theme attribute every time in the UI tag, then modify the `webwork.ui.theme` value in your `webwork.properties` file. Otherwise, you can just specify the theme attribute in all your UI tags.

As a good starting point, a good idea is to copy the contents of `xhtml` into a new subdirectory. Therefore, you can modify the templates and still refer back to the originals as a reference point.

Deployment Notes

This page last changed on May 24, 2004 by [plightbo](#).

WebWork runs on most application servers without any problems. However, you may need to do a few modifications in order to get it running in your environment.

WebLogic 6.1

A subproject has been added to the WebWork CVS repository that vastly simplifies getting WebWork to work under BEA Weblogic Server 6.1. Documentation is included. Look for the subproject under the main folder "misc".

SunONE 7.0

You need to grant permissions to WebWork:

```
grant {
    permission java.security.AllPermission;
};
```

or more specifically,

- Give Write Permissions to java.util.PropertyPermission.
- Add java.lang.reflect.ReflectPermission "suppressAccessChecks"
- OgnlInvoke Permission

```
grant {
    permission java.util.PropertyPermission "*", "read, write";
    permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
    permission ognl.OgnlInvokePermission "*";
};
```


Example

```
<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE xwork
PUBLIC
    "-//OpenSymphony Group//XWork
1.0//EN" "http://www.opensymphony.com/xwork/xwork-1.0.dtd"><xwork><include
file="webwork-default.xml"/><package name="default"
extends="webwork-default"><interceptors><interceptor-stack
name="defaultComponentStack"><interceptor-ref name="component"/><interceptor-ref
name="defaultStack"/></interceptor-stack></interceptors><default-interceptor-ref
name="defaultStack"/><action name="SimpleCounter"
class="com.opensymphony.webwork.example.counter.SimpleCounter"><result
name="success" type="dispatcher"/>success.jsp</result><interceptor-ref
name="defaultComponentStack"/></action><!--
- Velocity implementation of the SimpleCounter. Also demonstrate a more
verbose version of result element
--><action name="VelocityCounter"
class="com.opensymphony.webwork.example.counter.SimpleCounter"><result
name="success" type="velocity"><param
name="location"/>success.vm</param></result><interceptor-ref
name="defaultComponentStack"/></action><!--
- Different method can be used (processForm).
--><action name="formTest"
class="com.opensymphony.webwork.example.FormAction" method="processForm" ><result
name="success" type="dispatcher"/>formSuccess.jsp</result><result
name="invalid.token" type="dispatcher"/>form.jsp</result><interceptor-ref
name="defaultStack"/><interceptor-ref name="token"/></action></package></xwork>
```

Actions

```
<action name="formTest" class="com.opensymphony.webwork.example.FormAction"
method="processForm">
```

Actions are the basic "unit-of-work" in WebWork, they define, well, actions. An action will usually be a request, (and usually a button click, or form submit). The main action element (tag is too synonymous with JSP) has two parts, the friendly name (referenced in the URL, i.e. saveForm.action) and the corresponding "handler" class.

The optional "**method**" parameter tells WebWork which method to call based upon this action. If you leave the method parameter blank, WebWork will call the method **execute()** by default. If there is no execute() method and no method specified in the xml file, WebWork will throw an exception.

Also, you can tell WebWork to invoke "**doSomething**" method in your action by using the pattern "**actionName!something**" in your form. For example, "**formTest!save.action**" will invoke the method "**save**" in FormAction class. The method must be public and take no arguments:

```
publicString save() throws Exception
{
    ...
    return SUCCESS;
}
```

All the configuration for "**actionName**" will be used for "actionName!something" (interceptors, result types, etc...)

Results

```
<result name="missing-data" type="dispatcher"><param
name="location">/form.jsp</param><param name="parameterA">A</param><param
name="parameterB">B</param></result>
```

Result tags tell WebWork what to do next after the action has been called. The "name" attribute maps to the result code returned from the action execute() method. The "type" attribute indicates what result type class to use (see [Result Types](#)). The "param" elements allow you to pass parameters to the view:

```
<result-types>
    ...
    <result-type name="header"
        class="com.opensymphony.webwork.dispatcher.HttpHeaderResult"/></result-types><result
name="no-content" type="header"><param name="status">204</param><param
name="headers.customHeaderA">A</param><param
name="headers.customHeaderB">B</param></result>
```

There are a standard set of result codes built-in to WebWork, (in the Action interface) they include:

```
Action.SUCCESS = "success";
Action.NONE     = "none";
Action.ERROR    = "error";
Action.INPUT    = "input";
Action.LOGIN    = "login";
```

You can extend these result codes as you see fit (i.e "missing-data"). Most of the time you will have either SUCCESS or ERROR, with SUCCESS moving on to the next page in your application.

If you only need to specify the "location" parameter, you can use the short form:

```
<result name="missing-data" type="dispatcher">/form.jsp</result>
```

See [webwork-default.xml](#) or [Result Types](#) for standard result types.

Interceptors

Interceptors allow you to define code to be executed before and/or after the execution of an action. Interceptors can be a powerful tool when writing web applications. Some of the most common implementations of an Interceptor might be:

- Security Checking (ensuring the user is logged in)
- Trace Logging (logging every action)
- Bottleneck Checking (start a timer before and after every action, to check bottlenecks in your application)

You can also chain Interceptors together to create an interceptor **stack**. If you wanted to do a login check, security check, and logging all before an Action call, this could easily be done with an interceptor package.

Interceptors must first be defined (to give name them) and can be chained together as a stack:

```
<interceptors><interceptor name="security"
class="com.mycompany.security.SecurityInterceptor"/><interceptor-stack
name="defaultComponentStack"><interceptor-ref name="component"/><interceptor-ref
name="defaultStack"/></interceptor-stack></interceptors>
```

To use them in your actions:

```
<action name="VelocityCounter"
class="com.opensymphony.webwork.example.counter.SimpleCounter"><result
name="success">...</result><interceptor-ref name="defaultComponentStack"/></action>
```

NOTE: Reference name can be either the name of the interceptor or the name of a stack

For more details, see [Interceptors](#) reference.

Views

WebWork supports JSP and Velocity for your application presentation layer. For this example we will use a JSP file. Webwork comes packaged with a tag library (taglibs). You can use these taglibs as components in your JSP file. Here is an section of our form.jsp page:

```
<%@ taglib prefix="ww" uri="webwork" %>
<html>
<head><title>Webwork Form Example</title></head>
<body>
  <ww:form name="myForm" action="'formTest'" namespace="/" method="POST">
    <table>
      <ww:textfield label="First Name" name="'formBean.firstName'"
value="formBean.firstName"/>
      <ww:textfield label="Last Name" name="'formBean.lastName'"
value="formBean.lastName"/>
      <ww:submit value="Save Form"/>
    </table>
  </ww:form>
</body>
```

The process of events will go as follows:

1. WebWork will take notice since the URI ends in **.action** (defined in **web.xml** files)
2. WebWork will look up the action **formTest** in its action hierarchy and call any Interceptors that might have been defined.
3. WebWork will translate **formTest** and decide to call the method **processForm** in the class **com.opensymphony.webwork.example.FormAction** as defined in **xwork.xml** file.
4. The method will process successfully and give WebWork the **SUCCESS** return parameter.
5. WebWork will translate the **SUCCESS** return parameter into the location **formSuccess.jsp** (as defined in **xwork.xml**) and redirect accordingly.

Include

To make it easy to manage large scale development (lots of actions + configuration), WebWork allows you to include other configuration files from xwork.xml :

```
<xwork><include file="webwork-default.xml"/><include file="user.xml"/><include  
file="shoppingcart.xml"/><include file="product.xml"/>  
....  
</xwork>
```

The included files must be the same format as xwork.xml (with the doctype and everything) and be placed on classpath (usually in /WEB-INF/classes or jar files in /WEB-INF/lib).

Most of the content here provided by Matt Dowell <matt.dowell@notiva.com>

Lesson 4: Views

There are some different technologies that you could use as the view, i.e., to construct the user interface:

Lesson 4.1 - Java Server Pages

JSP is the common choice, because most Java web developers are already familiar with the technology. This lesson assumes you already have experience with Java Server Pages and demonstrates how you can use the WebWork features in JSP, mostly by using WebWork tags.

[Go to lesson 4.1](#)

Lesson 4.2 - Velocity

Velocity is a Java-based template engine that provides a simple, but powerful, template language that replaces JSP and allows for separation of concerns. This lesson assumes that you are already familiar with Velocity and teaches you how to use WebWork features from it.

[Go to lesson 4.2](#)

Lesson 4.3 - Freemarker

Designed for MVC pattern, Freemarker is another Java-based template engine that provides a powerful template language that replaces JSP, but can remain JSP-compatible with a JSP taglib support. This lesson teaches you how to use WebWork and Freemarker together.

[Go to lesson 4.3](#)

[Previous Lesson](#) | [Next Lesson](#)

Upgrading from 2.0

This page last changed on Aug 26, 2004 by [plightbo](#).

Upgrading from Webwork 2.0 is rather trivial. This version of webwork adds enhancements and bug fixes with hardly any configuration or syntax changes. Follow these two simple steps and you should be on your way with the latest and greatest from the OS crew.

1. Update/Replace your current binaries with the new binaries located in the distribution download under the `lib/core`. You may also want to grab any related jars from the `lib/optional` folder. Don't forget the webwork binary `webwork-2.1.jar` in the base directory of the distribution download. Review the [Dependencies](#) for Webwork.
2. Check out the [Release Notes - 2.1](#) to see if any of changes need to be applied to your code base.

Overview

In WebWork, the UI tags wrap generic HTML controls while providing tight integration with the core framework. The tags have been designed to minimize the amount of logic in compiled code and delegate the actual rendering of HTML to a template system. Templates can be grouped together and separated into different [themes](#). The UI tags attempt to cover the most common scenarios, while providing a Component Tag for creating custom components. The UI tags also provide built-in support for displaying inline error messages.

Templates

WebWork uses the Velocity template system to render the actual HTML output for all UI tags (jsp and velocity). A default implementation of all templates has been included with the core distribution allowing users to use WebWork's UI tags "out of the box". Templates can be edited individually or replaced entirely allowing for complete customization of the resulting HTML output. In addition, the default template can be overridden on a per tag basis allowing for a very fine level of control.

The templates can be found in the distribution package in a directory called `template` under the `src` directory or in the `webwork-x.x.jar` file. Copy the template directory into the root directory of your application. Otherwise, `webwork` will load the templates in the jar file.

```
/myApp
  /META-INF
  /WEB-INF
  /template
```

Inside the `template` directory, you will find two template sets called [Themes](#) (`xhtml` and `simple`). The default template set that is used with UI tags is [xhtml](#) unless specified by the `theme` attribute in your UI tag or in the [webwork.properties](#) file with the `webwork.ui.theme` variable. You can modify the pre-existing templates or create your own.

The `AbstractUI` class is the base class that other UI tags extend. It provides a set of attributes that are common across UI components. The `AbstractUI` class defines an abstract method:


```
protectedabstractString getTemplateName();
```

The AbstractUI class will load the template specified by the subclass or optionally, a template specified by the user using the template attribute. The following will load myOwnTextTemplate.vm for the textfield UI tag instead of the built in template text.vm

NOTE: You have to create a template file called myOwnTextTemplate.vm and store it in xhtml for this to work.

```
<!-- loads /template/xhtml/myOwnTextTemplate.vm -->  
  
<ww:ui textfield label=##mylabel## name=##myname## template=##myOwnTextTemplate.vm##  
>
```

otherwise

```
<!-- loads default /template/xhtml/text.vm -->  
<ww:ui textfield label=##mylabel## name=##myname## >
```

Built in templates

The default templates that correspond to each UI tag are as follows:

UI tag	default template
checkboxList	checkboxlist.vm
checkbox	checkbox.vm
combobox	combobox.vm
component	empty.vm
doubleSelect	doubleselect.vm
file	file.vm
form	form.vm(to open) form-close.vm(to close)
hidden	hidden.vm
label	label.vm
password	password.vm
radio	radiomap.vm
select	select.vm
submit	submit.vm
tabbedpane	tabbedpane.vm

textarea	textarea.vm
textfield	text.vm
token	token.vm

Accessing variables

A VelocityContext object is created and used by all WW velocity views with the following context parameters:

- tag - a reference to the tag object
- stack - the current OgnlValueStack
- ognl - a reference to the utility class OgnlTool
- req - a reference to the HttpServletRequest object
- res - a reference to the HttpServletResponse
- webwork - instance of WebWorkUtil
- action - the current Webwork action
- parameters - map of the current parameters

These variables can be accessed in the template by using \$TAG_NAME where TAG_NAME is one of tag, stack, ognl, req, ...). The template file is then processed. A few examples:

NOTE: The bang (!) will print the value if its defined and "" if its not

```
$!req.requestURI
$!req.method
```

```
$!tag.templateDir
$!tag.theme
```

```
$!parameters.name
$!parameters
```

Understanding the Webwork Template System

Look at how the template is found and loaded. A peek into AbstractUITag shows us the string used to build the template:

```
protectedString buildTemplateName(String myTemplate, String myDefaultTemplate) {
    ...
    return "/" + getTemplateDir() + "/" + getTheme() + "/" + template;
}
```

With the defaults, this will return the string for the textfield UI tag

```
/template/xhtml/text.vm
```

Webwork will attempt to find these values before it uses the default ones. You don't have to override any values and can modify the built in templates if you so desire(your choice). Webwork searches for these values in the order they are listed:

- getTemplateDir()
 - webwork.ui.templateDir value in webwork.properties
 - otherwise, "template" is returned
- getTheme()
 - in UI tag theme attribute
 - webwork.ui.theme value in webwork.properties
 - otherwise, "xhtml" is returned
- template
 - in UI tag template attribute
 - otherwise, defaults to specified template(see [defaults](#))

Templates with CSS

The default templates define several properties for use with CSS when HTML is generated from webwork tags. These properties can be found in a stylesheet located in the /template/xhtml directory called `styles.css`. You can use this stylesheet as a skeleton for your application and build on it or create your own, but remember you must include a link to the stylesheet within your jsp or velocity page.

`styles.css`:

```
.label {font-style:italic; }
.errorLabel {font-style:italic; color:red; }
.errorMessage {font-weight:bold; text-align: center; color:red; }
.checkboxLabel {}
.checkboxErrorLabel {color:red; }
.required {color:red;}
.requiredLabel {font-weight:bold; font-style:italic; }
```

referencing the stylesheet with a link inside your webpage (relative path or you can specify it from the root of your container):

```
<link rel ="stylesheet" type="text/css"
href="/webwork-example/template/xhtml/styles.css" title="Style">
```

Note: Webwork now has new attributes in the UI tags for more generic support of HTML styles and classes to make the look and feel even more flexible to implement. These are defined respectively as `cssStyle` and `cssClass`.

```
<ui:textfield label="'lebal'" name="'nmae'" cssStyle="'float:left; color:red'"
```

```
cssClass=" 'myclass' " />
```

Creating Custom Components

At first glance the component tag doesn't look that impressive. The ability to specify a single template and use a number of predetermined attributes looks rather lacking. But the supplied tag offers a number of benefits to developers.

Before diving right into the custom component, first I will identify some advantages to using the component tag to create your components. Then I will detail the two types of error messages in WebWork 2 and how our custom component (for displaying one of these types) fits into the equation. Finally, I will present a sample Action class, Jsp file and template file for our component. When we are finished, you will be able to incorporate the new component into your application.

Why use the component tag?

- removes the need to develop your own Jsp tag library
- provides integrated support for accessing the ValueStack
- leverages XWork's support for internationalization, localization and error handling
- faster prototyping using templates (editable text files) instead of compiled code
- re-use and combine existing templates

More on error message support:

In WebWork 2, there are two types of error messages: field error messages and action error messages. Field error messages are used to indicate a problem with a specific control and are displayed inline with the control. A number of tags provide built-in support for displaying these types of messages. Action error messages on the other hand, indicate a problem with executing an action. Many things can go wrong in a web application, especially an application that relies on external resources such as a database, remote web service, or other resource that might not be accessible during the execution of an action. Handling an error gracefully and presenting the user with a useful message can often be the difference in a positive user/customer experience and a bad one.

When these types of errors occur, it is more appropriate to display these messages separate from individual controls on the form. In the example below, we will create a custom component that can be used to display action error messages in a bulleted list. This component can then be used on all your forms to display these error messages.

The action class below was created to handle a promotion on the website: a free

e-certificate. It will try to email the certificate, but an exception will be thrown.

Action class:

```
package example;

import com.opensymphony.xwork.ActionSupport;

public class AddUser extends ActionSupport {

    privateString fullname;
    privateString email;

    publicString execute() throws Exception {
        // we are ignoring field validation in this example
        try {
            MailUtil.sendCertificate(email, fullname);
        } catch (Exception ex) {
            // there was a problem sending the email// in a real application, we
            // would also// log the exception
            addActionError("We are experiencing a technical problem and have
            contacted our support staff. " +
                "Please try again later.");
        }

        if (hasErrors()) {
            return ERROR;
        } else {
            return SUCCESS;
        }
    }

    publicString getFullname() {
        return fullname;
    }

    public void setFullname(String fullname) {
        this.fullname = fullname;
    }

    publicString getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

Jsp page:

```
<%@ taglib uri="webwork" prefix="ui" %>

<html>
<head><title>custom component example</title></head>

<!-- don't forget this -->
```

```

<link rel = "stylesheet" type="text/css"
href="/webwork-example/template/xhtmll/styles.css" title="Style">

<body>

<ui:form action="AddUser.action" method="POST">
<table>
    <ui:component template="action-errors.vm" />
    <ui:textfield label="Full Name" name="fullname" />
    <ui:textfield label="Email" name="email" />
    <ui:submit name="submit" value="Send me a free E-Certificate!" />
</table>
</ui:form>

</body>
</html>

```

HTML output (before submitting):

```

<html>
<head><title>custom component example</title></head>
<link rel = "stylesheet" type="text/css"
href="/webwork-example/template/xhtmll/styles.css" title="Style">

<body>

<form action="AddUser.action" method="POST" />

<table>

    <tr>
        <td align="right" valign="top"><span class="label">Full Name:</span></td>
        <td>
<input type="text" name="fullname" value="" />
        </td>
    </tr>
    <tr>
        <td align="right" valign="top"><span class="label">Email:</span></td>
        <td>
<input type="text" name="email" value="" />
        </td>
    </tr>
    <tr>
        <td colspan="2">
            <div align="right">
                <input type="submit" name="submit" value="Send me a free E-Certificate!" />
            </div>
        </td>
    </tr>

</table>
</form>

</body>
</html>

```

The template below will loop through any action errors and display them to the user in a bulleted list.

Template (action-errors.vm)

```
#set ($actionErrors = $stack.findValue("actionErrors"))

#if ($actionErrors)
<tr>
  <td colspan="2">
    <span class="errorMessage">The following errors occurred:</span>
    <ul>
      #foreach ($actionError in $actionErrors)
        <li><span class="errorMessage">$actionError</span></li>
      #end
    </ul>
  </td>
</tr>
#end
```

HTML output (after submitting):

```
<html>
<head><title>custom component example</title></head>
<link rel = "stylesheet" type="text/css"
href="/webwork-example/template/xhtmll/styles.css" title="Style">

<body>

<form action="AddUser.action" method="POST" />

<table>

<tr>
  <td colspan="2">
    <span class="errorMessage">The following errors occurred:</span>
    <ul>
      <li class="errorMessage">
        We are experiencing a technical problem and have contacted our
        support staff. Please try again later.
      </li>
    </ul>
  </td>
</tr>

  <tr>
    <td align="right" valign="top"><span class="label">Full Name:</span></td>
    <td>
      <input type="text" name="fullname" value="Sample User" />
    </td>
  </tr>

  <tr>
    <td align="right" valign="top"><span class="label">Email:</span></td>
    <td>
      <input type="text" name="email" value="user@example.com" />
    </td>
  </tr>
</table>
```

```
</tr>

<tr>
  <td colspan="2">
    <div align="right">
      <input type="submit" name="submit" value="Send me a free E-Certificate!"/>
    </div>
  </td>
</tr>

</table>
</form>

</body>
</html>
```


WebWork Tutorial's Examples

The examples supplied in this tutorial are available as a Web Application for you to deploy and run. Download `wwtutorialexamples.war` below and install it in your Servlet Container. You can also extract the files from the archive using your favorite Zip software.

[wwtutorialexamples.war](#)

Lesson 4.1: Using JSP as the View

When using JSP to render the views, you can choose to access the action's data using scriptlets or tags. Tags are the recommended approach.

Accessing Action Data through Scriptlets:

Action data can be accessed through an object called *Value Stack*. The example below does the same thing as the result page of [lesson 3](#)'s second example (*Supplying Data to the Action*), but using scriptlets:

```
<%@ page import="com.opensymphony.xwork.util.OgnlValueStack" %>
<html>
<head>
<title>WebWork Tutorial - Lesson 4.1 - Lesson 3's example modified</title>
</head>
<body>

<%
OgnlValueStack stack = (OgnlValueStack)request.getAttribute("webwork.valueStack");
out.write("Hello, " + stack.findValue("person"));
%>

</body>
</html>
```

WebWork tags, however, are recommended over scriptlets. For instance, `<ww:property />` tags do exactly what the scriptlet above does, with a cleaner syntax and also handles the case where the Value Stack doesn't exist.

WebWork Tag Library:

We've already showed in [lesson 3](#)'s example how to access an action's property using tags. This section describes and exemplifies the use of the WebWork Tag Library, which can be divided in seven categories:

- **Common tags:** the most frequently used, basic tags;
- **Componentisation tags:** foster componentisation within your views;

- **Flow control tags:** govern the flow of control within the JSP page;
- **Iteration tags:** iterate over elements and manipulate iterable objects;
- **UI tags:** generate HTML form fields and controls;
- **VUI tags:** *volunteers needed to write this part*;
- **Internationalisation tags:** internationalise your views.

Common tags

<code><ww:property /></code>	Gets the value of a result attribute. If the value isn't given, the top of the stack will be returned.
<code><ww:push /></code>	Pushes a value onto the Value Stack.
<code><ww:param /></code>	Sets a parent tag's parameter. This tag is used only inside another tag to set the value of some property of the parent tag.
<code><ww:set /></code>	Sets the value of an object in the Value Stack to a scope (page, stack, application, session). If the value is not given, the top of the stack is used. If the scope is not given, the default scope of "webwork" is used.
<code><ww:url /></code>	Builds an encoded URL.

EXAMPLE NEEDED.

Read more: [Non-UI Tags](#)

Componentisation tags

<code><ww:action /></code>	Executes an Action from within the context of a taglib. The body of the tag is used to display the results of the action invocation.
<code><ww:bean /></code>	Creates a JavaBean, instantiate its properties and place it in the ActionContext for later use.
<code><ww:include /></code>	Includes another page or action.

EXAMPLE NEEDED.

Read more: [Non-UI Tags](#)

Flow control tags

This if-else set of tags works just like if-else scriptlets.

<code><ww:if /></code>	Conditional execution path. That is, evaluates the tag body if a boolean expression is true.
<code><ww:else /></code>	Negative execution path for the if tag. That is, if the preceeding conditional tag's boolean expression evaluated to false, then evaluate this tag's body.
<code><ww:elseif /></code>	Negative conditional execution path for the if tag. That is, if the preceeding conditional tag's boolean expression evaluated to false and if this tag's boolean expression evaluates to true, then evaluate this tag's body.

EXAMPLE NEEDED.

Read more: [Non-UI Tags](#)

Iteration tags

<code><ww:iterator /></code>	Iterates over a collection.
<code><ww:generator /></code>	Generates iterators.
<code><ww:append /></code>	Appends several iterators.
<code><ww:subset /></code>	Gets a subset of an iterator.
<code><ww:merge /></code>	Merges several iterators into one.
<code><ww:sort /></code>	Sorts an iterator.

EXAMPLE NEEDED.

| Read more: [Non-UI Tags](#)

UI tags

The UI tags wrap generic HTML controls while providing tight integration with the core framework. The tags have been designed to minimize the amount of logic in compiled code and delegate the actual rendering of HTML to a template system. The UI tags attempt to cover the most common scenarios, while providing a Component Tag for creating custom components. The UI tags also provide built-in support for displaying inline error messages.

There is a separate lesson about WebWork UI Tags which explains in detail how they work, how you could customize their appearance through the use of templates, how to create custom components, etc.

[Go to WebWork UI Tags Lesson.](#)

VUI tags

Volunteers needed to write this part.

Internationalisation tags

<code><ww:text /></code>	Prints out an internationalized string.
<code><ww:i18n /></code>	Places a resource bundle on the Value Stack, for access by the text tag.

| Read more: [UI Tags](#)

[Previous Lesson](#) | [Next Lesson](#)

velocity.properties

This page last changed on Jun 18, 2004 by [plightbo](#).

This file if provided (/WEB-INF/classes) will be loaded by Velocity. It can be used to load custom macros:

```
# Velocity Macro libraries.  
velocimacro.library = webwork.vm, tigris-macros.vm, myapp.vm
```

Check Velocity documentation for other parameters.

Visitor validation

This page last changed on Jul 10, 2004 by [unkyaku](#).

The VisitorFieldValidator allows you to forward validation to object properties of your action using the objects own validation files. This allows you to use the ModelDriven development pattern and manage your validations for your models in one place, where they belong, next to your model classes. The VisitorFieldValidator can handle either simple Object properties, Collections of Objects, or Arrays. See [XW:Standard Validators#VisitorFieldValidator](#)

There's a number of approaches you can take to unit-test your WebWork actions.

The simplest is to instantiate your actions, call setters then execute(). This allows you to bypass all the complicated container setup.

Taken from Petsoar:

```
package org.petsoar.actions.inventory;

import com.mockobjects.constraint.IsEqual;
import com.mockobjects.dynamic.C;
import com.mockobjects.dynamic.Mock;
import com.opensymphony.xwork.Action;
import junit.framework.TestCase;
import org.petsoar.pets.Pet;
import org.petsoar.pets.PetStore;

public class TestViewPet extends TestCase {
    private Mock mockPetStore;
    private ViewPet action;

    protected void setUp() throws Exception {
        mockPetStore = new Mock(PetStore.class);
        PetStore petStore = (PetStore) mockPetStore.proxy();

        action = new ViewPet();
        action.setPetStore(petStore);
    }

    public void testViewPet() throws Exception {
        Pet existingPet = new Pet();
        existingPet.setName("harry");
        existingPet.setId(1);

        Pet expectedPet = new Pet();
        expectedPet.setName("harry");
        expectedPet.setId(1);

        mockPetStore.expectAndReturn("getPet", C.args(new IsEqual(new Long(1))),
existingPet);
        action.setId(1);

        String result = action.execute();

        assertEquals(Action.SUCCESS, result);
        assertEquals(expectedPet, existingPet);
    }
}
```



```

        mockPetStore.verify();
    }

    public void testViewPetNoId() throws Exception {
        mockPetStore.expectAndReturn("getPet", C.ANY_ARGS, null);

        String result = action.execute();

        assertEquals(Action.ERROR, result);
        assertEquals(1, action.getActionErrors().size());
        assertEquals("Invalid pet selected.",
            action.getActionErrors().iterator().next());
        assertNull(action.getPet());
        mockPetStore.verify();
    }

    public void testViewPetInvalidId() throws Exception {
        action.setId(-1);
        testViewPetNoId();
    }
}

```

Test interceptors and/or result types

Check out the test suites in XWork/WebWork. These are pretty comprehensive and provide a good starting point. For example, this is how the **ParametersInterceptor** is tested:

```

public void testDoesNotAllowMethodInvocations() {
    Map params = new HashMap();
    params.put("@java.lang.System@exit(1).dummy", "dumb value");

    HashMap extraContext = new HashMap();
    extraContext.put(ActionContext.PARAMETERS, params);

    try {
        ActionProxy proxy = ActionProxyFactory.getFactory().
            createActionProxy("",
                MockConfigurationProvider.MODEL_DRIVEN_PARAM_TEST, extraContext);
        assertEquals(Action.SUCCESS, proxy.execute());

        ModelDrivenAction action = (ModelDrivenAction) proxy.getAction();
        TestBean model = (TestBean) action.getModel();

        String property = System.getProperty("webwork.security.test");
        assertNull(property);
    } catch (Exception e) {
        e.printStackTrace();
        fail();
    }
}

```

Note: these are not the ONLY ways so make your own judgement.

What is Spring and Why do you want to use it with WebWork?

Check out <http://www.springframework.org> for more details about Spring. To summarize, Spring provides several different layers. Spring's IoC container, for instance, provides a nice transparent way to wire together objects with their dependencies, such as services they use. It can also, with the help of its AOP framework, provide transactional behavior to plain Java beans. Spring also provides an MVC framework, which is what could be compared to WebWork. There are those who believe WebWork is a better MVC framework, and so would choose WebWork for this part and integrate the rest of the Spring stack.

There are a number of ways to integrate Spring into WebWork.

Use SpringObjectFactory

The [xwork-optional](#) package from dev.java.net contains a module **xwork-spring** that contains all the necessary code to use Spring in WebWork. It contains primarily a SpringObjectFactory to wire up the dependencies for an Action before passing it to WebWork. Each action should be configured within a Spring application context as a prototype (because WebWork assumes a new instance of a class for every action invocation). Specify something like this in applicationContext.xml:

```
<bean name="some-action" class="fully.qualified.class.name"
singleton="false"><property name="someProperty"><ref
bean="someOtherBean" /></property></bean>
```

and in xwork.xml:

```
<action name="myAction" class="some-action"><result
name="success">view.jsp</result></action>
```

Notice that the WebWork Action's class name **some-action** is the bean name defined in the Spring application context.

Another advantage of the SpringObjectFactory approach is that it can also be used to load interceptors using the same sort of logic. If the interceptor is stateless, then it's possible to create the interceptor as a singleton instance, but otherwise it's best to

create it as a Spring prototype.

In order to be used, the default ObjectFactory that WebWork uses should be replaced with an instance of the SpringObjectFactory. The xwork-optional package ships with a ContextListener that does this, assuming that the Spring application context has already been configured. Add the following to web.xml:

```
<!-- This needs to be after Spring ContextLoaderListener  
--><listener><listener-class>com.opensymphony.xwork.spring.SpringObjectFactoryListener</listener>
```

Note: this is actually a XWork configuration but for simplicity, I just assume WebWork.

The following class performs the glue between Quartz and Webwork:

```
package com.trantek.sit.action;

import com.opensymphony.xwork.ActionProxy;
import com.opensymphony.xwork.ActionProxyFactory;
import org.quartz.Job;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;

public class WebWorkJob implements Job
{
    public void execute(JobExecutionContext context) throws JobExecutionException
    {
        try
        {
            ActionProxy proxy = ActionProxyFactory.getFactory().
                createActionProxy("", context.getJobDetail().getName(),
context.getJobDetail().getJobDataMap());

            proxy.execute();
        }
        catch (Exception e)
        {
            throw new JobExecutionException(e);
        }
    }
}
```

To schedule webwork actions you simply create a job where

- the name of your job is the name of the WW action to execute (no ".action" suffix).
- all the parameters you want to send to the WW action is contained in the JobDataMap of the JobDetail

(the Quartz scheduler is setup as a servlet according to the javadocs of `org.quartz.ee.servlet.QuartzInitializerServlet`.)

The following code schedules an e-mail action:

```
Scheduler scheduler = StdSchedulerFactory.getDefaultScheduler();

JobDetail jobDetail = new JobDetail("email.send",
                                     scheduler.DEFAULT_GROUP, WebworkJob.class);

Map m = jobDetail.getJobDataMap();
m.put("to", "me@bogusdomain.com");
```

```
m.put("subject", "quartz test");
m.put("body", "This is a quartz test, Hey ho");
m.put("smtpServer", "smtp.bogusdomain.com");
m.put("from", "quartz@bogusdomain.com");

SimpleTrigger trigger = new SimpleTrigger("myTrigger",
                                           scheduler.DEFAULT_GROUP,
                                           new Date(), null, 0, 0L);

scheduler.deleteJob("email.send", scheduler.DEFAULT_GROUP);
scheduler.scheduleJob(jobDetail, trigger);
```

This example is based on [WW1: Integrating Webwork and Quartz](#)

Namespaces

The namespace attribute allows you to segregate action configurations into namespaces, so that you may use the same action alias in more than one namespace with different classes, parameters, etc. This is in contrast to Webwork 1.x, where all action names and aliases were global and could not be re-used in an application. The default namespace, which is "" (an empty string) is used as a "catch-all" namespace, so if an action configuration is not found in a specified namespace, the default namespace will also be searched. This allows you to have global action configurations outside of the "extends" hierarchy, as well as to allow the previous Webwork 1.x behavior by not specifying namespaces. It is also intended that the namespace functionality can be used for security, for instance by having the path before the action name be used as the namespace by the Webwork 2.0 ServletDispatcher, thus allowing the use of J2EE declarative security on paths to be easily implemented and maintained.

Namespace example

```
<package name="default">

  <action name="foo" class="mypackage.simpleAction">
    <result name="success" type="dispatcher">greeting.jsp</result>
  </action>
  <action name="bar" class="mypackage.simpleAction">
    <result name="success" type="dispatcher">bar1.jsp</result>
  </action>

</package>

<package name="mypackage" namespace="/barspace">

  <action name="bar" class="mypackage.simpleAction">
    <result name="success" type="dispatcher">bar2.jsp</result>
  </action>

</package>
```

If a request for /barspace/bar.action is made, then the package named mypackage is searched and the bar action is executed. If success is returned, then bar2.jsp is displayed.

Note: If a request is made to /barspace/foo.action, the action foo will be searched for in a namespace of /barspace. If the action is not found, the action will then be searched for in the default namespace. Unless specified, the default namespace will be

"". In our example above, there is no action foo in the namespace /barspace, therefore the default will be searched and /foo.action will be executed.

Validation Examples

This page last changed on Jun 11, 2004 by [ctran](#).

Included in the [WW:WebWork](#) example war file is an example of using the [XW:Validation Framework](#) in WebWork2. This example consists of three links which all use the same Action Class and view pages (Velocity).

The sources

First, I had to add the **validators.xml** file to the root of the source tree for the example app

```
<validators><validator name="required"
class="com.opensymphony.xwork.validator.validators.RequiredFieldValidator"/><validator
name="requiredstring"
class="com.opensymphony.xwork.validator.validators.RequiredStringValidator"/><validator
name="int"
class="com.opensymphony.xwork.validator.validators.IntRangeFieldValidator"/><validator
name="date"
class="com.opensymphony.xwork.validator.validators.DateRangeFieldValidator"/><validator
name="expression"
class="com.opensymphony.xwork.validator.validators.ExpressionValidator"/><validator
name="fieldexpression"
class="com.opensymphony.xwork.validator.validators.FieldExpressionValidator"/><validator
name="email"
class="com.opensymphony.xwork.validator.validators.EmailValidator"/><validator
name="url"
class="com.opensymphony.xwork.validator.validators.URLValidator"/><validator
name="visitor"
class="com.opensymphony.xwork.validator.validators.VisitorFieldValidator"/></validators>
```

The Action class used by all of the validation examples is **ValidatedAction**

```
package com.opensymphony.webwork.example;

import com.opensymphony.xwork.ActionSupport;

/**
 * ValidatedAction
 * @author Jason Carreira
 * Created Sep 12, 2003 9:23:38 PM
 */
public class ValidatedAction extends ActionSupport {
    private ValidatedBean bean = new ValidatedBean();
    private String name;
    private String validationAction = "basicValidation.action";

    public ValidatedBean getBean() {
        return bean;
    }
}
```



```

public void setBean(ValidatedBean bean) {
    this.bean = bean;
}

publicString getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

publicString getValidationAction() {
    return validationAction;
}

public void setValidationAction(String validationAction) {
    this.validationAction = validationAction;
}
}

```

The base validation file for the ValidatedAction is **ValidatedAction-validation.xml**

```

<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator
1.0//EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.dtd"><validators><field
name="name"><field-validator type="requiredstring"><message>You must enter a
name.</message></field-validator></field></validators>

```

This and all other validation files are placed in the same package as the classes to which they apply.

The form for this Action is **validationForm.vm**

```

<html>
<head><title>Webwork Validation Example</title></head>
<body>
#if( $actionErrors.size() > 0 )
<p>
<font color="red">
<b>ERRORS:</b><br>
<ul>
#foreach( $error in $actionErrors )
<li>$error</li>
#end
</ul>
</font>
</p>
#end
<p>
<form name="myForm" action="{validationAction}" method="POST">
<input type="hidden" name="validationAction" value="{validationAction}"/>
Action Properties:
<br>
<table>
#tag( TextField "label=Name" "name=name" "value=name" )
</table>
Bean Properties:
#if( $stack.findValue("fieldErrors") )

```

```

    #set( $beanErrors = $stack.findValue("fieldErrors.get('bean')") )
    #if( $beanErrors.size() > 0 )
    <br>
    <font color="red">
    <b>Bean Errors:</b><br>
    <ul>
    #foreach( $beanError in $beanErrors )
    <li>$beanError</li>
    #end
    </ul>
    </font>
    #end
#end
<table>
#tag( TextField "label=Bean.Text" "name=bean.text" "value=bean.text" )<br>
#tag( TextField "label=Bean.Date" "name=bean.date" "value=bean.date" )<br>
#tag( TextField "label=Bean.Number" "name=bean.number" "value=bean.number" )<br>
#tag( TextField "label=Bean.Number2" "name=bean.number2" "value=bean.number2" )<br>
</table>
<input type="submit" value="Test Validation"/>
</form>
</body>

```

The success page for these examples is a very simple page, **valid.vm**

```

<html>
<head><title>WebWork Validation Test: Valid</title></head>
<body>
Input was valid!
</body>
</html>

```

We'll look at any other example-specific configuration files as we get to them.

Basic Validation

The BasicValidation example is defined in the example **xwork.xml** file like this

```

<action name="basicValidation"
class="com.opensymphony.webwork.example.ValidatedAction"><interceptor-ref
name="validationWorkflowStack"/><result name="success"
type="dispatcher">valid.vm</result><result name="input"
type="dispatcher">validationForm.vm</result><result name="error"
type="dispatcher">validationForm.vm</result></action>

```

The **interceptor-ref** here, to **"validationWorkflowStack"**, is defined in **webwork-default.xml** and provides the parameter interceptors as well as the **ValidationInterceptor** (see [XW:Validation Framework](#) and the **DefaultWorkflowInterceptor** (see [XW:Interceptors#DefaultWorkflow](#)). All of the parameters from the configuration file (there are none in this case) followed by the parameters from the request will be set onto the Action. Next, the validations will be

run, and finally the DefaultWorkflow will be applied (see [XW:Interceptors#DefaultWorkflow](#)).

This example is very simple, and the ValidatedAction-validation.xml file is the only set of Validations which will be applied. This means that the only validation done is that you enter some text for the name field.

Visitor Validation Example

The **ValidatedAction** holds a reference to a plain Java bean, **ValidatedBean**:

```
package com.opensymphony.webwork.example;

import java.util.Date;

/**
 * ValidatedBean
 * @author Jason Carreira
 * Created Sep 12, 2003 9:24:18 PM
 */
public class ValidatedBean {
    privateString text;
    private Date date = new Date(System.currentTimeMillis());
    privateint number;
    privateint number2;
    publicstaticfinalint MAX_TOTAL = 12;

    publicString getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    publicint getNumber() {
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
    }

    publicint getNumber2() {
        return number2;
    }
}
```

```

    public void setNumber2(int number2) {
        this.number2 = number2;
    }
}

```

The base validation file for the ValidatedBean is **ValidatedBean-validation.xml**

```

<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator
1.0//EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.dtd"><validators><field
name="text"><field-validator type="requiredstring"><message key="invalid.text">Empty
Text!</message></field-validator></field><field name="date"><field-validator
type="date"><param name="min">01/01/1970</param><message key="invalid.date">Invalid
Date!</message></field-validator></field><field name="number"><field-validator
type="int"><param name="min">1</param><param name="max">10</param><message
key="invalid.number">Invalid
Number!</message></field-validator></field></validators>

```

In the Visitor Validation Example, we add a **VisitorFieldValidator** to apply these validations to our **ValidatedBean**. The Action is defined in our **xwork.xml** file like this:

```

<action name="visitorValidation"
class="com.opensymphony.webwork.example.ValidatedAction"><interceptor-ref
name="validationWorkflowStack"/><param
name="validationAction">visitorValidation.action</param><result
name="success">valid.vm</result><result
name="input">validationForm.vm</result><result
name="error">validationForm.vm</result></action>

```

Here we see a slight difference from the basic validation example above. I've added a static param to the Action which will be applied to the Action by the static-param interceptor. This parameter only sets the value for the action to post the form to for validation (see **validationForm.vm**).

The Action name above, *visitorValidation* is mapped to another set of validations defined in the file **ValidatedAction-visitorValidation-validation.xml**

```

<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator
1.0//EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.dtd"><validators><field
name="bean"><field-validator type="required"><message>The bean must not be
null.</message></field-validator><field-validator type="visitor"><message>bean:
</message></field-validator></field></validators>

```

This file is found automatically by the validation framework based on the class name (ValidatedAction), the action alias, which is used as the validation context (visitorValidation) and the standard suffix (-validation.xml) to form the filename **ValidatedAction-visitorValidation-validation.xml**.

This file defines two validators for the "bean" field, a required validator which makes sure the bean is not null, and a VisitorFieldValidator. The VisitorFieldValidator will

apply the validators for the **ValidatedBean** using the same validation context as is used in validating **ValidatedAction**, *visitorValidation*. It therefore looks for the validation files **ValidatedBean-validation.xml** (the default validations for the ValidatedBean) and **ValidatedBean-visitorValidation-validation.xml** (the validations specific to this validation context) , in that order.

The **ValidatedBean-validation.xml** looks like this:

```
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator
1.0//EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.dtd"><validators><field
name="text"><field-validator type="requiredstring"><message key="invalid.text">Empty
Text!</message></field-validator></field><field name="date"><field-validator
type="date"><param name="min">01/01/1970</param><message key="invalid.date">Invalid
Date!</message></field-validator></field><field name="number"><field-validator
type="int"><param name="min">1</param><param name="max">10</param><message
key="invalid.number">Invalid
Number!</message></field-validator></field></validators>
```

This file applies validations for three fields (text, date, and number) and gives message keys and default messages for each of them. These message keys will be used to look up messages from a resource bundle specific to the **ValidatedBean** class. In the same package as the **ValidatedBean** is a file named **ValidatedBean.properties**

```
invalid.date=You must enter a date after ${min}.
invalid.text=You must enter some text.
invalid.number=You must enter a number between ${min} and ${max}.
invalid.total=The total of number and number2 must be less than
${@com.opensymphony.webwork.example.ValidatedBean@MAX_TOTAL}.
```

These messages will be used for any errors added for the **ValidatedBean** using a message key. As you can see from the body of the messages, they can be parameterized with properties from the Bean, the Interceptor, and the Action (and they will be searched in that order). There is also an example of using a Static field **`${@com.opensymphony.webwork.example.ValidatedBean@MAX_TOTAL}`**.

The **ValidatedBean-visitorValidation-validation.xml** file would define validations specific for the *visitorValidation* validation context, but it is not there, so it is ignored.

Visitor Validation with the Expression Validator

The final example shows a similar setup to the previous visitor validation example.

The **xwork.xml** configuration for this example is very similar to the **visitorValidation** example:

```
<action name="expressionValidation"
class="com.opensymphony.webwork.example.ValidatedAction"><interceptor-ref
name="validationWorkflowStack"/><param
name="validationAction">expressionValidation.action</param><result
name="success">valid.vm</result><result
name="input">validationForm.vm</result><result
name="error">validationForm.vm</result></action>
```

The **ValidatedAction-expressionValidation-validation.xml** file defines the validations specific to this validation context:

```
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator
1.0//EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.dtd"><validators><field
name="bean"><field-validator type="required"><message>The bean must not be
null.</message></field-validator><field-validator type="visitor"><param
name="context">expression</param><message>bean:
</message></field-validator></field></validators>
```

This is almost identical to the **ValidatedAction-visitorValidation-validation.xml** file, but shows an example of passing a context param to the **VisitorFieldValidator**. In this case, rather than using the same validation context as is used for the **ValidatedAction** (*expressionValidation*), it passes another context (*expression*) to be used instead.

In this case, the validation context specific validations for the **ValidatedBean** is present, and it's named **ValidatedBean-expression-validation.xml**

```
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator
1.0//EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.dtd"><validators><validator
type="expression"><param
name="expression">@com.opensymphony.webwork.example.ValidatedBean@MAX_TOTAL >
(number + number2)</param><message key="invalid.total">Invalid
total!</message></validator></validators>
```

This adds an object-level (as opposed to field-level) ExpressionValidator which checks the total of the number and number2 fields against a static constant, and adds an error message if the total is more than the constant.

A note about error messages with the VisitorFieldValidator

With the VisitorFieldValidator, message field names are appended with the field name of the field in the Action. In this case, the fields *text*, *date*, and *number* in the **ValidatedBean** would be added as field error messages to the Action with field

names "*bean.text*", "*bean.date*", and "*bean.number*". The error messages added for the object-level ExpressionValidator applied in the last example will be added as field-level errors to the Action with the name "*bean*".

Lesson 1: Downloading and Installing WebWork

The first step to use WebWork is installing it. The current version can be found at webwork.dev.java.net. This tutorial is based on version 2.1, which was the latest version at the time it was written.

After downloading *webwork-2.1.zip*, unpack it anywhere you like. The libraries that are needed to use WebWork are *webwork-2.1.jar* and all the JAR files inside the *lib/core* folder.

The next lesson will use these libraries to set up the web application. [Next Lesson](#)

Lesson 2: Setting up the Web Application

It is assumed that you have a Servlet container set up and you know how to create a web application. If you don't, we suggest you learn about [Apache Tomcat](#), which is a free Servlet container from the Apache Jakarta Project, or Resin, from [Caucho Technology](#), which is free for non-commercial use.

To use WebWork, copy the required libraries, described in the previous lesson (`webwork-2.1.jar` and `lib/core/*.jar`), to the directory `WEB-INF/lib` of the Web Application. Then, configure `web.xml` and create other two XML files: `xwork.xml` and `validators.xml`. These three files will be explained below.

web.xml:

Change your web application's `web.xml` file to look somewhat like this:

```
<?xml version="1.0"?><!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>My WebWork
Application</display-name><servlet><servlet-name>webwork</servlet-name><servlet-class>com.opensy
```

To use WebWork you must register `ServletDispatcher` and its mapping to `*.action`. It will be explained why those lines are important in the section about Actions in the [next lesson](#). In the example above WebWork's taglib descriptor is also declared to allow the usage of WebWork tags (more about WebWork tags on [lesson 4.1](#)) – it is safe to remove it if WebWork tags are not being used.

| Read more: [web.xml](#)

xwork.xml:

At the root of the classpath (namely, `WEB-INF/classes`) create a file called

, which WebWork uses to configure itself. For now, place the following content in it:

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork
1.0//EN" "http://www.opensymphony.com/xwork/xwork-1.0.dtd"><xwork><!-- Include
webwork defaults (from WebWork-2.1 JAR). --><include file="webwork-default.xml"
/><!-- Configuration for the default package. --><package name="default"
extends="webwork-default"></package></xwork>
```

This is just a skeleton of a configuration file, which will be incremented as we go through the lessons. As of now, it is only doing two things:

- informing WebWork that it should import the configuration information from `webwork-default.xml` (which is located at the root of `webwork-2.1.jar` and thus available for use) – this file defines the `webwork-default` package, which contains the *default* configuration for WebWork applications;
- defining a default package, which is where the actions, results and interceptors are registered. This package extends `webwork-default`, i.e., the default package will inherit all the configuration defined in `webwork-default`.

Read more: [xwork.xml](#)

validators.xml:

Again, at the root of the classpath, create a file called `validators.xml`, with the following content:

```
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator
1.0//EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.dtd"><validators><validator
name="required"
    class="com.opensymphony.xwork.validator.validators.RequiredFieldValidator"/><validator
name="requiredstring"
    class="com.opensymphony.xwork.validator.validators.RequiredStringValidator"/><validator
name="int"
    class="com.opensymphony.xwork.validator.validators.IntRangeFieldValidator"/><validator
name="date"
    class="com.opensymphony.xwork.validator.validators.DateRangeFieldValidator"/><validator
name="expression"
    class="com.opensymphony.xwork.validator.validators.ExpressionValidator"/><validator
name="fieldexpression"
    class="com.opensymphony.xwork.validator.validators.FieldExpressionValidator"/><validator
name="email"
    class="com.opensymphony.xwork.validator.validators.EmailValidator"/><validator
name="url"
    class="com.opensymphony.xwork.validator.validators.URLValidator"/><validator
name="visitor"
    class="com.opensymphony.xwork.validator.validators.VisitorFieldValidator"/><validator
name="conversion"
    class="com.opensymphony.xwork.validator.validators.ConversionErrorFieldValidator"/></valida
```

This file defines the validators that are available for use.

| Read more: [Validation](#)

All Set Up!

Now there is a skeleton of the WebWork application already set up. Next lessons will teach how to use WebWork's actions, views and interceptors.[Previous Lesson](#) | [Next Lesson](#)

Lesson 3: Actions and Results

Actions are the basic unit of execution. An action is a class that is registered under WebWork's configuration to respond to a specific request. In a Model-View-Controller approach, the Action is part of the Controller, leaving to JSP pages what they do best: being the View.

The following steps are a possible way of creating an action in WebWork:

1. Create a JSP page that will call the action;
2. Create the action class;
3. Create a JSP page that will process the result;
4. Register the action in `xwork.xml`.

The first example of this tutorial could be no other than *"Hello, WebWorld!"*. The code below displays WebWork's configuration file `xwork.xml` with configuration for an action under the `default` package.

xwork.xml:

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork
1.0//EN" "http://www.opensymphony.com/xwork/xwork-1.0.dtd"><xwork><!-- Include
webwork defaults (from WebWork-2.1 JAR). --><include file="webwork-default.xml"
/><!-- Configuration for the default package. --><package name="default"
extends="webwork-default"><!-- Default interceptor stack.
--><default-interceptor-ref name="defaultStack" /><!-- Action: Lesson 03:
HelloWebWorldAction. --><action name="helloWebWorld"
class="lesson03.HelloWebWorldAction"><result name="success"
type="dispatcher">ex01-success.jsp</result></action></package></xwork>
```

Don't worry about the `default-interceptor-ref` yet. For now we are interested in the action only. This configuration file is telling WebWork that there is an action called `helloWebWorld` which is implemented by the class `lesson03.HelloWebWorldAction`. For this action, we define a result under the name `success` which points to the web page `ex01-success.jsp`.

| Read more: [xwork.xml](#)

Things should get clearer once we take a look at the code that calls the action:

ex01-index.jsp:

```
<html>
<head>
<title>WebWork Tutorial - Lesson 3 - Example 1</title>
</head>

<body>

<p>Click the button below to activate HelloWebWorldAction.</p>

<form action="helloWebWorld.action" method="post">
<p><input type="submit" value="Hello!" /></p>
</form>

</body>
</html>
```

This is how web pages can integrate with WebWork: by accessing *.action URLs. Recalling the previous lesson, that's why we registered WebWork's ServletDispatcher with mapping to *.action.

When we click the button in our example page, the browser will send the form data to the URL helloWebWorld.action. Since this URL fits under the mapping *.action, the Servlet container will activate WebWork's ServletDispatcher, which will read xwork.xml and look for an action called helloWebWorld. If the action is found, a **new** instance of the action class is created and the method execute() is called.

Let's see the action class, then:

HelloWebWorldAction.java:

```
package lesson03;

import com.opensymphony.xwork.ActionSupport;

public class HelloWebWorldAction extends ActionSupport {
    String hello;
    publicString getHello() {
        return hello;
    }
    publicString execute() throws Exception {
        hello = "Hello, WebWorld!";
        return SUCCESS;
    }
}
```

First, notice that this class extends com.opensymphony.xwork.ActionSupport and

implements the method `public String execute()`. For starters, all your action classes should do that. Second, notice that the method `execute()` only sets the value of the `hello` property and returns the constant `SUCCESS`, which is a `String` whose value is `"success"`.

When the `ServletDispatcher` gets `"success"` as return from `execute()`, it looks again at `xwork.xml` for a result with that name, finds it, and moves on to `ex01-success.jsp`, as specified in the configuration.

Let's see what that page does:

ex01-success.jsp:

```
<%@ taglib uri="webwork" prefix="ww" %>
<html>
<head>
<title>WebWork Tutorial - Lesson 3 - Example 1</title>
</head>
<body>

<ww:property value="hello" />

</body>
</html>
```

If you run the example, you will see that this page will display `"Hello, WebWorld!"`. That happens because what the `<ww:property value="hello" />` tag does is look for the property `hello` in the action class that just executed. Actually, it looks for an accessor method, so it ends up calling `getHello()`. Since `HelloWebWorldAction` was called and set the value of the `hello` property to `"Hello, WebWorld!"`, `getHello()` will return that value and it will be displayed by the resulting JSP page.

[Try the example!](#)

Supplying Data to the Action

The previous example demonstrated how WebWork's actions work, but we can't do much if we're not able to supply data to our action. Let's see an example that does just that:

xwork.xml:

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork
1.0//EN" "http://www.opensymphony.com/xwork/xwork-1.0.dtd"><xwork><!-- Include
webwork defaults (from WebWork-2.1 JAR). --><include file="webwork-default.xml"
/><!-- Configuration for the default package. --><package name="default"
extends="webwork-default"><!-- Default interceptor stack.
--><default-interceptor-ref name="defaultStack" /><!-- Action: Lesson 03:
HelloAction. --><action name="hello" class="lesson03.HelloAction"><result
name="error" type="dispatcher">ex02-index.jsp</result><result name="success"
type="dispatcher">ex02-success.jsp</result></action></package></xwork>
```

HelloAction.java:

```
package lesson03;

import com.opensymphony.xwork.ActionSupport;

public class HelloAction extends ActionSupport {
    String person;
    public String getPerson() {
        return person;
    }
    public void setPerson(String person) {
        this.person = person;
    }
    public String execute() throws Exception {
        if ((person == null) || (person.length() == 0)) return ERROR;
        else return SUCCESS;
    }
}
```

ex02-index.jsp:

```
<html>
<head>
<title>WebWork Tutorial - Lesson 3 - Example 2</title>
</head>

<body>

<p>What's your name?</p>

<form action="hello.action" method="post">
<p><input type="text" name="person" /><input type="submit" /></p>
</form>

</body>
</html>
```

ex02-success.jsp:

```
<%@ taglib uri="webwork" prefix="ww" %>
<html>
<head>
<title>WebWork Tutorial - Lesson 3 - Example 2</title>
</head>
<body>

Hello, <ww:property value="person" />

</body>
</html>
```

Let's analyse the differences: in this example we are sending form data via POST, under the name of `person`. Because of that, after a new instance of `HelloAction` is created, WebWork's `ServletDispatcher` will try to fill the action's property `person` with the data that was sent, therefore calling the mutator method `setPerson()` (actually, that is done by the `ParametersInterceptor`, which we will only learn about later, in [lesson 5](#) – for now, know that when the action is executed, the property is already set).

If you look at the action class' code, you will see that during `execute()`, it checks if the property was filled (i.e. if data was supplied in the form field). If it was, it returns `SUCCESS`, therefore dispatching the request to `ex02-success.jsp`, otherwise, it returns `ERROR`, moving back to `ex02-index.jsp`.

| [Try the example!](#)

Types of Result

On the examples above, when we say that a result is of type `"dispatcher"` it means that WebWork looks for a `result-type` called `dispatcher` and finds out which result class implements it. An instance of that class is activated whenever the action returns that type of result.

Result classes implement the `com.opensymphony.xwork.Result` interface. WebWork comes with some result classes already implemented, but you can build your own, if you want. WebWork's result types are already configured in `webwork-default.xml`:

- **dispatcher**

(`com.opensymphony.webwork.dispatcher.ServletDispatcherResult`): forwards the result to the specified location;

- **redirect** (`com.opensymphony.webwork.dispatcher.ServletRedirectResult`): redirects the result to the specified location. Unlike the dispatcher, the redirection does not send form data (via POST or GET) to the specified page;
- **velocity** (`com.opensymphony.webwork.dispatcher.VelocityResult`): uses a Velocity template as the result. You could use the dispatcher to forward results to Velocity pages if you have `VelocityServlet` configured in `web.xml`, but using the Velocity result is a better approach. More on Velocity on [lesson 4.2](#);
- **chain** (`com.opensymphony.xwork.ActionChainResult`): chains an action on another, i.e., the result of an action is another action;
- **xslt** (`com.opensymphony.webwork.views.xslt.XSLTResult`): uses an XML document transformed by an XSLT style sheet as the result.

Read more: [Result Types](#)

Moving on

These examples illustrate the main concept of actions and results: they are units of execution that respond to requests, do some processing and, depending on the result, dispatch the request to some other location. Although that might seem very simple, it's just the core of WebWork's functionality.

Here are some other things that can be done with actions, just so you know what WebWork can do before we move on to other lessons:

- With interceptors, it can log every action execution and even clock the execution time ([lesson 5](#));
- The action could work directly with your business object by implementing the `ModelDriven` interface;
- With XML files, configure the validation of the data that is sent to the action. If some data is not valid according to your specification, the action is not executed and error messages are attached to the action (reference: [Validation](#));
- By using WebWork UI Tags in association with the validation framework, forms will automatically display error messages due to invalid data input ([lesson 4.1.1](#));

- WebWork can provide to some actions instances of components (in different scopes) automatically, allowing Inversion of Control (reference: [Inversion of Control & Components]);

The next lessons will talk about the different types of views that can be used by WebWork. [Previous Lesson](#) | [Next Lesson](#)

Lesson 4.2: Using Velocity with WebWork

There are two ways of using Velocity as the view.

- Using the `velocity` result-type to render velocity templates;
- Registering `WebWorkVelocityServlet` in your `web.xml` file to render Velocity templates accessed directly through browser requests.

To use the second approach, we have to modify `web.xml` and add a servlet and a servlet mapping for `WebWorkVelocityServlet`, as demonstrated below:

```
<servlet><servlet-name>velocity</servlet-name><servlet-class>com.opensymphony.webwork.views.velocity.servlet.WebWorkVelocityServlet</servlet-class></servlet>
```

| Read more: [xwork.xml](#)

Using `velocity` result-type means that Velocity templates can only be rendered through an action, i.e., request to `.vm` pages will not render the file and it will be returned as plain text. If you choose this approach, it's recommended that you place your Velocity files under `WEB-INF` so they become inaccessible.

Using `WebWorkVelocityServlet` means that Velocity templates can be rendered through requests to `.vm` pages. That also means that you should implement security checks in your templates so an user doesn't access it directly without going through an action first (if that is required).

No matter which approach you choose (and you can choose to use both at the same time), not only all the features from Velocity are available to you when you're writing templates, but also some other functionalities, specific of WebWork, are available. It is supposed that you are already familiar with Velocity, so we will focus only in the WebWork-specific features. If that's not the case, please [get started with Velocity](#) before continuing.

The main feature of it is to provide easy access to objects that are on the Value Stack, which contains some things that WebWork provides to you automatically, because you may find them useful at some point. These are some of the things that are available in the value stack:

- The current `HttpServletRequest`;
- The current `HttpServletResponse`;
- The current `OgnlValueStack`;
- An instance of `OgnlTool`;
- All the properties of the current action class.

To access the objects in the value stack, all you have to do is use appropriate Velocity references:

- `$req` = `HttpServletRequest`;
- `$res` = `HttpServletResponse`;
- `$stack` = `OgnlValueStack`;
- `$ognl` = `OgnlTool`;
- `$name-of-property` = property of the current action class.

The example below does the same thing as the Hello example from [lesson 3](#), but now, using a Velocity template as the result. Notice that the `<property value="person" />` tag was replaced by the `$person` reference, which returns the same thing: a property from the action class. In this example we chose to use the *velocity result-type* approach.

xwork.xml:

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork
1.0//EN" "http://www.opensymphony.com/xwork/xwork-1.0.dtd"><xwork><!-- Include
webwork defaults (from WebWork-2.1 JAR). --><include file="webwork-default.xml"
/><!-- Configuration for the default package. --><package name="default"
extends="webwork-default"><!-- Default interceptor stack.
--><default-interceptor-ref name="defaultStack" /><!-- Action: Lesson 4.2:
HelloAction using Velocity as result. --><action name="helloVelocity"
class="lesson03.HelloAction"><result name="error"
type="dispatcher">ex01-index.jsp</result><result name="success"
type="velocity">ex01-success.vm</result></action></package></xwork>
```

HelloAction.java (same as lesson 3):

```
package lesson03;

import com.opensymphony.xwork.ActionSupport;

public class HelloAction extends ActionSupport {
    String person;
    public String getPerson() {
        return person;
    }
    public void setPerson(String person) {
        this.person = person;
    }
}
```

```

publicString execute() throws Exception {
    if ((person == null) || (person.length() == 0)) return ERROR;
    elsereturn SUCCESS;
}
}

```

ex01-index.jsp (same as lesson 3):

```

<html>
<head>
<title>WebWork Tutorial - Lesson 3 - Example 2</title>
</head>

<body>

<p>What's your name?</p>

<form action="hello.action" method="post">
<p><input type="text" name="person" /><input type="submit" /></p>
</form>

</body>
</html>

```

ex01-success.vm:

```

<html>
<head>
<title>WebWork Tutorial - Lesson 4.2 - Example 1</title>
</head>
<body>

Hello, $person

</body>
</html>

```

[Try the example!](#)

Using WebWork Tags from Velocity:

As you already know, when you switch from JSP to Velocity you lose the ability of using JSP Tags. But WebWork's Velocity Servlet provides a way of doing this through the use of #tag, #bodytag and #param velocimacros. The general syntax is:

```
#tag (name-of-tag list-of-attributes)
```

– or –

```
#bodytag (name-of-tag list-of-attributes)
#param (key value)
#param (key value)
...
#end
```

Let's revisit [lesson 4.1.1](#)'s form example to demonstrate the usage of the UI tags from velocity:

xwork.xml:

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork
1.0//EN" "http://www.opensymphony.com/xwork/xwork-1.0.dtd"><xwork><!-- Include
webwork defaults (from WebWork-2.1 JAR). --><include file="webwork-default.xml"
/><!-- Configuration for the default package. --><package name="default"
extends="webwork-default"><!-- Default interceptor stack.
--><default-interceptor-ref name="defaultStack" /><!-- Actions: Lesson 4.2:
FormProcessingAction using Velocity. --><action name="formProcessingVelocityIndex"
class="lesson04_02.FormProcessingIndexAction"><result name="success"
type="velocity">ex02-index.vm</result></action><action name="formProcessingVelocity"
class="lesson04_01_01.FormProcessingAction"><result name="input"
type="velocity">ex02-index.vm</result><result name="success"
type="velocity">ex02-success.vm</result><interceptor-ref
name="validationWorkflowStack" /></action></package></xwork>
```

ex02-index.vm:

```
<html>
<head>
<title>WebWork Tutorial - Lesson 4.2 - Example 2</title>
<style type="text/css">
.errorMessage { color: red; }
</style>
</head>

<body>

<p>UI Form Tags Example using Velocity:</p>

#bodytag (Form "action='formProcessingVelocity.action'" "method='post'")
#tag (Checkbox "name='checkbox'" "label='A
checkbox'" "fieldValue='checkbox_value'")
#tag (File "name='file'" "label='A file field'")
#tag (Hidden "name='hidden'" "value='hidden_value'")
#tag (Label "label='A label'")
#tag (Password "name='password'" "label='A password field'")
#tag (Radio "name='radio'" "label='Radio buttons'" "list={'One', 'Two', 'Three'}")
```

```

    #tag (Select "name='select'" "label='A select list'" "list={'One', 'Two',
'Three'}" "emptyOption=true")
    #tag (Textarea "name='textarea'" "label='A text area'" "rows='3'" "cols='40'")
    #tag (TextField "name='textfield'" "label='A text field'")
    #tag (Submit "value='Send Form'")
#end

</body>
</html>

```

ex02-success.vm:

```

<html>
<head>
<title>WebWork Tutorial Lesson 4.2 - Example 2</title>
</head>

<body>

<p>UI Form Tags Example result using Velocity:</p>

<ul>
  <li>checkbox: $!checkbox</li>
  <li>file: $!file</li>
  <li>hidden: $!hidden</li>
  <li>password: $!password</li>
  <li>radio: $!radio</li>
  <li>select: $!select</li>
  <li>textarea: $!textarea</li>
  <li>textfield: $!textfield</li>
</ul>

</body>
</html>

```

FormProcessingAction.java (same as lesson 4.1.1):

```

package lesson04_01_01;

import com.opensymphony.xwork.ActionSupport;

public class FormProcessingAction extends ActionSupport {
    privateString checkbox;
    privateString file;
    privateString hidden;
    privateString password;
    privateString radio;
    privateString select;
    privateString textarea;
    privateString textfield;

    publicString getCheckbox() { return checkbox; }
    publicString getFile() { return file; }
    publicString getHidden() { return hidden; }

```

```

    publicString getPassword() { return password; }
    publicString getRadio() { return radio; }
    publicString getSelect() { return select; }
    publicString getTextArea() { return textarea; }
    publicString getTextfield() { return textfield; }

    public void setCheckbox(String checkbox) { this.checkbox = checkbox; }
    public void setFile(String file) { this.file = file; }
    public void setHidden(String hidden) { this.hidden = hidden; }
    public void setPassword(String password) { this.password = password; }
    public void setRadio(String radio) { this.radio = radio; }
    public void setSelect(String select) { this.select = select; }
    public void setTextArea(String textarea) { this.textarea = textarea; }
    public void setTextfield(String textfield) { this.textfield = textfield; }

    publicString execute() throws Exception {
        return SUCCESS;
    }
}

```

FormProcessingAction-validation.xml (same as lesson 4.1.1):

```

<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator
1.0//EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.dtd"><validators><field
name="checkbox"><field-validator type="requiredstring"><message>Please, check the
checkbox.</message></field-validator></field><field name="file"><field-validator
type="requiredstring"><message>Please select a
file.</message></field-validator></field><field name="password"><field-validator
type="requiredstring"><message>Please type something in the password
field.</message></field-validator></field><field name="radio"><field-validator
type="requiredstring"><message>Please select a radio
button.</message></field-validator></field><field name="select"><field-validator
type="requiredstring"><message>Please select an option from the
list.</message></field-validator></field><field name="textarea"><field-validator
type="requiredstring"><message>Please type something in the text
area.</message></field-validator></field><field name="textfield"><field-validator
type="requiredstring"><message>Please type something in the text
field.</message></field-validator></field></validators>

```

[Try the example!](#)

The example above does not use the `#param` tag. So, let's revisit another example from [lesson 4.1.1](#) - custom components:

ex03.vm:

```
<html>
```



```

<head>
<title>WebWork Tutorial - Lesson 4.2 - Example 3</title>
</head>

<body>

<p>Custom Component Example:</p>

<p>
#bodytag (Component "template=/files/templates/components/datefield.vm")
  #param ("label" "Date")
  #param ("name" "mydatefield")
  #param ("size" "3")
#end
</p>

</body>
</html>

```

/files/templates/components/datefield.vm (same as lesson 4.1.1):

```

#set ($name = $tag.params.get('name'))
#set ($size = $tag.params.get('size'))
#set ($yearSize = $size * 2)

$tag.params.get('label'):
<input type="text" name="{name}.day" size="$size" /> /
<input type="text" name="{name}.month" size="$size" /> /
<input type="text" name="{name}.year" size="$yearSize" /> (dd/mm/yyyy)

```

Notice that, this time, we did not enclose `Date` and `mydatefield` with single quotes, as we had to do when we used the JSP tag.

| [Try the example!](#)

[Previous Lesson](#) | [Next Lesson](#)

Lesson 4.1.1: WebWork UI Tags

In WebWork, the UI tags wrap generic HTML controls while providing tight integration with the core framework. The tags have been designed to minimize the amount of logic in compiled code and delegate the actual rendering of HTML to a template system. The UI tags attempt to cover the most common scenarios, while providing a Component Tag for creating custom components. The UI tags also provide built-in support for displaying inline error messages.

This lesson tries to explain how to take advantage of the UI tags to build forms and other graphical controls and, by explaining how the template system works, teaches you how to change the look of existing components and create your own UI components.

Building forms:

WebWork comes with ready-to-use tags to construct forms. Some of these tags relate directly to HTML tags that are used to make forms and you probably can figure them out by their names: `<ww:checkbox />`, `<ww:file />`, `<ww:form />`, `<ww:hidden />`, `<ww:label />`, `<ww:password />`, `<ww:radio />`, `<ww:select />`, `<ww:submit />`, `<ww:textarea />` and `<ww:textfield />`.

To build forms with these tags, place them in your page as you would do with the HTML tags. The only difference is that the parameters should be enclosed in double quotes and single quotes (`key="'value'"`). That's because names that are not single-quoted are evaluated against the Value Stack.

Let's check out an example:

ex01-index.jsp:

```
<%@ taglib uri="webwork" prefix="ww" %>
<html>
<head>
<title>WebWork Tutorial - Lesson 4.1.1 - Example 1</title>
```

```

<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<style type="text/css">
  .errorMessage { color: red; }
</style>
</head>

<body>

<p>UI Form Tags Example:</p>

<ww:form action="'formProcessing.action'" method="'post'">
  <ww:checkbox name="'checkbox'" label="'A checkbox'" fieldValue="'checkbox_value'"
/>
  <ww:file name="'file'" label="'A file field'" />
  <ww:hidden name="'hidden'" value="'hidden_value'" />
  <ww:label label="'A label'" />
  <ww:password name="'password'" label="'A password field'" />
  <ww:radio name="'radio'" label="'Radio buttons'" list="{ 'One', 'Two', 'Three' }" />
  <ww:select name="'select'" label="'A select list'" list="{ 'One', 'Two', 'Three' }"
    emptyOption="true" />
  <ww:textarea name="'textarea'" label="'A text area'" rows="3" cols="40" />
  <ww:textfield name="'textfield'" label="'A text field'" />
  <ww:submit value="'Send Form'" />
</ww:form>

</body>
</html>

```

HTML result after processing ex01-index.jsp:

```

<html>
<head>
<title>WebWork Tutorial - Lesson 4.1.1 - Example 1</title>
<style type="text/css">
  .errorMessage { color: red; }
</style>
</head>

<body>

<p>UI Form Tags Example:</p>

<table>
<form
action="formProcessing.action" method="post" >

<tr>
<td valign="top" colspan="2">

<table width="100%" border="0" cellpadding="0" cellspacing="0">
<tr><td valign="top">
<input type="checkbox"
name="checkbox"
value="checkbox_value"
/>

```

```

</td>
<td width="100%" valign="top">
<span class="checkboxLabel">
A checkbox
</span>
</td>
</tr>
</table>
</td>
</tr>

<tr>
<td align="right" valign="top">

<span class="label">

A file field:
</span>
</td>

<td>

<input type="file"
name="file"
/>

</td>
</tr>

<input
type="hidden"
name="hidden" value="hidden_value" />

<tr>
<td align="right" valign="top">

<span class="label">

A label:
</span>
</td>

<td>
<label> </label>
</td>
</tr>

<tr>
<td align="right" valign="top">

```

```

<span class="label">

A password field:
</span>
</td>

<td>

<input type="password"
name="password"

/>

</td>
</tr>


<tr>
<td align="right" valign="top">

<span class="label">

Radio buttons:
</span>
</td>

<td>

<input
type="radio"
name="radio"
id="radioOne"
value="One" />
<label for="radioOne">One</label>


<input
type="radio"
name="radio"
id="radioTwo"
value="Two" />
<label for="radioTwo">Two</label>


<input
type="radio"
name="radio"
id="radioThree"

```

```
value="Three" />
<label for="radioThree">Three</label>
```

```
</td>
</tr>
```

```
<tr>
<td align="right" valign="top">

<span class="label">
```

```
A select list:
</span>
</td>
```

```
<td>

<select name="select"
>
```

```
<option value=""></option>
```

```
<option value="One"
>One</option>
```

```
<option value="Two"
>Two</option>
```

```
<option value="Three"
>Three</option>
```

```
</select>
```

```
</td>
</tr>
```

```
<tr>
<td align="right" valign="top">
```

```

<span class="label">

A text area:
</span>
</td>

<td>

<textarea name="textarea"
cols="40"
rows="3"
></textarea>

</td>
</tr>


<tr>
<td align="right" valign="top">

<span class="label">

A text field:
</span>
</td>

<td>

<input type="text"
name="textfield"
/>

</td>
</tr>

<tr>
<td colspan="2"><div
align="right" ><input
type="submit"
value="Send Form" /></div>
</td>
</tr>

</form>
</table>

</body>
</html>

```

xwork.xml:

```

<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork
1.0//EN" "http://www.opensymphony.com/xwork/xwork-1.0.dtd"><xwork><!-- Include
webwork defaults (from WebWork-2.1 JAR). --><include file="webwork-default.xml"

```

```
<!-- Configuration for the default package. --><package name="default"
extends="webwork-default"><action name="formProcessing"
class="lesson04_01_01.FormProcessingAction"><result name="input"
type="dispatcher">ex01-index.jsp</result><result name="success"
type="dispatcher">ex01-success.jsp</result><interceptor-ref
name="validationWorkflowStack" /></action></package></xwork>
```

FormProcessingAction.java:

```
package lesson04_01_01;

import com.opensymphony.xwork.ActionSupport;

public class FormProcessingAction extends ActionSupport {
    privateString checkbox;
    privateString file;
    privateString hidden;
    privateString password;
    privateString radio;
    privateString select;
    privateString textarea;
    privateString textfield;

    publicString getCheckbox() { return checkbox; }
    publicString getFile() { return file; }
    publicString getHidden() { return hidden; }
    publicString getPassword() { return password; }
    publicString getRadio() { return radio; }
    publicString getSelect() { return select; }
    publicString getTextArea() { return textarea; }
    publicString getTextField() { return textfield; }

    public void setCheckbox(String checkbox) { this.checkbox = checkbox; }
    public void setFile(String file) { this.file = file; }
    public void setHidden(String hidden) { this.hidden = hidden; }
    public void setPassword(String password) { this.password = password; }
    public void setRadio(String radio) { this.radio = radio; }
    public void setSelect(String select) { this.select = select; }
    public void setTextArea(String textarea) { this.textarea = textarea; }
    public void setTextField(String textfield) { this.textfield = textfield; }

    publicString execute() throws Exception {
        return SUCCESS;
    }
}
```

FormProcessingAction-validation.xml:

```
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator
1.0//EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.dtd"><validators><field
name="checkbox"><field-validator type="requiredstring"><message>Please, check the
checkbox.</message></field-validator></field><field name="file"><field-validator
type="requiredstring"><message>Please select a
file.</message></field-validator></field><field name="password"><field-validator
```



```

type="requiredstring"><message>Please type something in the password
field.</message></field-validator></field><field name="radio"><field-validator
type="requiredstring"><message>Please select a radio
button.</message></field-validator></field><field name="select"><field-validator
type="requiredstring"><message>Please select an option from the
list.</message></field-validator></field><field name="textarea"><field-validator
type="requiredstring"><message>Please type something in the text
area.</message></field-validator></field><field name="textfield"><field-validator
type="requiredstring"><message>Please type something in the text
field.</message></field-validator></field></validators>

```

ex01-success.jsp:

```

<%@ taglib uri="webwork" prefix="ww" %>
<html>
<head>
<title>WebWork Tutorial - Lesson 4.1.1 - Example 1</title>
</head>

<body>

<p>UI Form Tags Example result:</p>

<ul>
  <li>checkbox: <ww:property value="checkbox" /></li>
  <li>file: <ww:property value="file" /></li>
  <li>hidden: <ww:property value="hidden" /></li>
  <li>password: <ww:property value="password" /></li>
  <li>radio: <ww:property value="radio" /></li>
  <li>select: <ww:property value="select" /></li>
  <li>textarea: <ww:property value="textarea" /></li>
  <li>textfield: <ww:property value="textfield" /></li>
</ul>

</body>
</html>

```

Notice how much cleaner ex01-index.jsp is, compared to its HTML result. The default layout of the form components is a table layout, with the label on the left column and the field to the right. You can learn how to create your own layouts when we explain the template system, below.

Another thing to notice is the reference to the `validationWorkflowStack` in the action's configuration. This makes WebWork validate the parameters that are sent to our actions according to a configuration file we place in the same location as the action class – in our case, `FormProcessingAction-validation.xml` (see [Validation](#)). In case something is not valid, it prevents the action from executing and dispatches the request to the input result with error messages attached to each field (using the method `addFieldError(String fieldName, String errorMessage)`).

But don't worry about how the validation framework works for now. Run the example

and try leaving some fields blank. You will see that the UI tags provide error messages that integrate with the validation framework and that's what we want to demonstrate here. This separation of concerns can help programmers and designers concentrate more on their part of the work.

| Read more: [UI Tags](#)

| [Try the example!](#)

Other UI Controls:

Besides the standard form controls that HTML designers are already familiar with, WebWork provides some other controls and also the ability to create a custom control. Let's take a look at the custom controls that are already provided by WebWork:

<code><ww:checkboxlist /></code>	Works just like the <code><ww:radio /></code> tag, but with check boxes instead of radio buttons. It gets the keys and values from a collection and creates a list of checkboxes, all with the same name.
<code><ww:combobox /></code>	Simulates a combo box, which is a control that mixes a selection list with a text field. It does this by placing a text field with a <code><select /></code> list right below it and a JavaScript code that fills the text field with the selection of the list every time it changes.
<code><ww:tabbedpane /></code>	<i>Help needed here.</i>
<code><ww:token /></code>	<i>Help needed here.</i>

| Read more: [UI Tags](#)

The Template System:

WebWork uses the Velocity template system to render the actual HTML output for all UI tags. A default implementation of all templates has been included with the core distribution allowing users to use WebWork's UI tags "out of the box". Templates can be edited individually or replaced entirely allowing for complete customization of the resulting HTML output. In addition, the default template can be overridden on a per

tag basis allowing for a very fine level of control. The default templates are located in the `webwork-2.1.jar` file under `/template/xhtmll`.

If you unpack `webwork-2.1.jar` and look under the `/template/xhtmll` directory you will see a bunch of velocity templates. Most of them correspond to a specific UI Tag, and those have the name of the tag they render. If you're familiar with Velocity, I recommend you analyse the template files to see what you're capable of doing with them. Since version 2.1, there's also a `/template/simple` directory, which is a simpler version of the HTML form controls (just the control, no table or label).

If you want to display your UI components in a different layout than the one that comes with WebWork, you can:

- Edit and replace the files in `/template/xhtmll` (repack the JAR or create the same directory structure somewhere else and make sure your container looks that path before the JAR);
- Change the location of the templates by editing the `webwork.ui.theme` property in `webwork.properties` (file that should be placed in the root of your classpath);
- Specifying the location of the templates for each tag individually using the theme or the template property. The former allows you to specify the directory where all templates are (thus, WebWork looks for templates with the same name as the ones in `/template/xhtmll`), while the latter allows you to indicate the exact template to be used for that component.

| Read more: [Templates](#), [Themes](#)

The third approach is demonstrated in the example below.

ex02.jsp:

```
<%@ taglib uri="webwork" prefix="ww" %>
<html>
<head>
<title>WebWork Tutorial - Lesson 4.1.1 - Example 2</title>
</head>

<body>

<p>Template Change Example:</p>

<p><ww:checkbox name="'checkbox'" label="'A checkbox'"
fieldValue="'checkbox_value'" theme="/files/templates/components" /></p>
```

```
<p><ww:textfield name="'textfield'" label="'A text field'"
template="/files/templates/components/mytextfield.vm" /></p>

</body>
</html>
```

/files/templates/components/checkbox.vm:

```
<div align="center">
  <input type="checkbox"
    name="`${webwork.htmlEncode($parameters.name)}`"
    value="`${webwork.htmlEncode($parameters.fieldValue)}`"
    #if ($parameters.nameValue) checked="checked" #end
    #if ($parameters.disabled == true) disabled="disabled" #end
    #if ($parameters.tabindex) tabindex="`${webwork.htmlEncode($parameters.tabindex)}`"
  #end
  #if ($parameters.onchange) onchange="`${webwork.htmlEncode($parameters.onchange)}`"
  #end
  #if ($parameters.id) id="`${webwork.htmlEncode($parameters.id)}`" #end
  /><br />
  `${webwork.htmlEncode($parameters.label)}`
</div>
```

/files/templates/components/mytextfield.vm:

```
<div align="center">
  <input type="text"
    name="`${webwork.htmlEncode($parameters.name)}`"
    #if ($parameters.size) size="`${webwork.htmlEncode($parameters.size)}`" #end
    #if ($parameters.maxlength) maxlength="`${webwork.htmlEncode($parameters.maxlength)}`"
  #end
    #if ($parameters.nameValue) value="`${webwork.htmlEncode($parameters.nameValue)}`"
  #end
    #if ($parameters.disabled == true) disabled="disabled" #end
    #if ($parameters.readonly) readonly="readonly" #end
    #if ($parameters.onkeyup) onkeyup="`${webwork.htmlEncode($parameters.onkeyup)}`"
  #end
    #if ($parameters.tabindex) tabindex="`${webwork.htmlEncode($parameters.tabindex)}`"
  #end
    #if ($parameters.onchange) onchange="`${webwork.htmlEncode($parameters.onchange)}`"
  #end
    #if ($parameters.id) id="`${webwork.htmlEncode($parameters.id)}`" #end
  /><br />
  `${webwork.htmlEncode($parameters.label)}`
</div>
```

HTML result after processing ex02.jsp:

```
<html>
```

```

<head>
<title>WebWork Tutorial - Lesson 4.1.1 - Example 2</title>
</head>

<body>

<p>Template Change Example:</p>

<p><div align="center">
  <input type="checkbox"
    name="checkbox"
    value="checkbox_value"
  /><br />
  A checkbox
</div></p>

<p><div align="center">
  <input type="text"
                                name="textfield"
  /><br />
  A text field
</div></p>

</body>
</html>

```

[Try the example!](#)

Building Customized UI Components:

There are some situations in which none of the UI Components that come bundled with WebWork fit your requirements. In this case, the recommended approach would be to create your own custom component. In this way, you keep your web page clean of layout and error-checking issues and also promotes component reuse.

To create a custom component, just create a Velocity template for it, just like the ones that already exist. To place it in a web page, use the `<ww:component />` tag and specify the location of the template in its `template` parameter.

To pass parameters to be used by your template, use the `<ww:param />` tag (see [lesson 4.1](#)). The example below demonstrates the creation of a custom date field.

Read more: [UI Tags](#)

ex03.jsp:

```

<%@ taglib uri="webwork" prefix="ww" %>

```

```

<html>
<head>
<title>WebWork Tutorial - Lesson 4.1.1 - Example 3</title>
</head>

<body>
<p>Custom Component Example:</p>

<p>
<ww:component template="/files/templates/components/datefield.vm">
  <ww:param name="label" value="'Date'" />
  <ww:param name="name" value="'mydatefield'" />
  <ww:param name="size" value="3" />
</ww:component>
</p>

</body>
</html>

```

/files/templates/components/datefield.vm:

```

#set ($name = $tag.params.get('name'))
#set ($size = $tag.params.get('size'))
#set ($yearSize = $size * 2)

$tag.params.get('label'):
<input type="text" name="${name}.day" size="$size" /> /
<input type="text" name="${name}.month" size="$size" /> /
<input type="text" name="${name}.year" size="$yearSize" /> (dd/mm/yyyy)

```

HTML result after processing ex03.jsp:

```

<html>
<head>
<title>WebWork Tutorial - Lesson 4.1.1 - Example 3</title>
</head>

<body>

<p>Custom Component Example:</p>

<p>
Date:
<input type="text" name="mydatefield.day" size="3" /> /
<input type="text" name="mydatefield.month" size="3" /> /
<input type="text" name="mydatefield.year" size="6" /> (dd/mm/yyyy)

</p>

</body>
</html>

```

[Try the example!](#)

[Previous Lesson](#) | [Next Lesson](#)

Overview

OGNL is the Object Graph Navigation Language - see <http://www ognl.org> for the full documentation of OGNL. In this document we will only show a few examples of OGNL features that co-exist with Webwork.

- To review basic concepts, refer to [OGNL Basics](#)
- [#Collections](#)
- [Lambda Expressions](#)

Collections (Maps, Lists, Sets)

Dealing with collections (maps, lists, and sets) in webwork comes often, so here are a few examples using the select tag:

Syntax for list: {e1,e2}. This creates a List containing the String "name1" and "name2".

```
<webwork:select label="'lebal'" name="'nmae'" list="{ 'name1','name2' }" />
```

Syntax for map: #{key1:value1,key2:value2}. This creates a map that maps the string "foo" to the string "foovalue" and "bar" to the string "barvalue":

```
<webwork:select label="'lebal'" name="'nmae'" list="#{ 'foo':'foovalue',  
'bar':'barvalue' }" />
```

You may need to determine if an element exists in a collection. You can accomplish this with the operations `in` and `not in`

```
<ui:if test="'foo' in { 'foo','bar' }">  
  muhahaha  
</ui:if>  
<ui:else>  
  boo  
</ui:else>  
  
<ui:if test="'foo' not in { 'foo','bar' }">  
  muhahaha  
</ui:if>  
<ui:else>  
  boo  
</ui:else>
```


To select a subset of a collection (called projection), you can use a wildcard within the collection.

- ? - All elements matching the selection logic
- ^ - Only the first element matching the selection logic
- \$ - Only the last element matching the selection logic

To obtain a subset of just male relatives from the object person:

```
person.relatives.{? #this.gender == 'male'}
```

Lambda Expressions

OGNL supports basic lambda expression syntax enabling you to write simple functions.

For example:

For all you math majors who didn't think you would ever see this one again.

Fibonacci: if n==0 return 0; elseif n==1 return 1; else return fib(n-2)+fib(n-1);

fib(0) = 0

fib(1) = 1

fib(11) = 89

The lambda expression is everything inside the brackets. The #this variable holds the argument to the expression, which is initially starts at 11.

```
<ww:property value="#fib =:[#this==0 ? 0 : #this==1 ? 1 :  
#fib(#this-2)+#fib(#this-1)], #fib(11)" />
```

Global results

This page last changed on Jun 16, 2004 by [casey](#).

Global results allows you to define result mappings which will be used as defaults for all action configurations and will be automatically inherited by all action configurations in this package and all packages which extend this package. In other words, if you have the same result specified within multiple actions, then you can define it as a global result.

global results example

```
<package name="default">
....
<global-results>
  <result name="login" type="dispatcher">
    <param name="location">login.jsp</param>
  </result>
</global-results>
<action name="foo" class="mypackage.fooAction">
  <result name="success" type="dispatcher">bar.jsp</result>
</action>
<action name="submitForm" class="mypackage.submitFormAction">
  <result name="success" type="dispatcher">submitSuccess.jsp</result>
</action>
...
</package>
```

Same thing

```
<package name="default">
....
<action name="foo" class="mypackage.fooAction">
  <result name="success" type="dispatcher">bar.jsp</result>
  <result name="login" type="dispatcher">login.jsp</result>
</action>
<action name="submitForm" class="mypackage.submitFormAction">
  <result name="success" type="dispatcher">submitSuccess.jsp</result>
  <result name="login" type="dispatcher">login.jsp</result>
</action>
...
</package>
```

Results

This page last changed on Jun 16, 2004 by [casey](#).

Overview

Results are string constants that Actions return to indicate the status of an Action execution. A standard set of Results are defined by default: error, input, login, none and success. Developers are, of course, free to create their own Results to indicate more application specific cases. Results are mapped to defined [Result Types](#) using a name-value pair structure.

- [Global results](#)
- [Default results](#)

Result tags

Result tags tell WebWork what to do next after the action has been called. There are a standard set of result codes built-in to WebWork, (in the Action interface) they include:

```
String SUCCESS = "success";
String NONE    = "none";
String ERROR   = "error";
String INPUT   = "input";
String LOGIN   = "login";
```

You can extend these as you see fit. Most of the time you will have either **SUCCESS** or **ERROR**, with **SUCCESS** moving on to the next page in your application;

```
<result name="success" type="dispatcher"><param
name="location">/thank_you.jsp</param></result>
```

...and **ERROR** moving on to an error page, or the preceding page;

```
<result name="error" type="dispatcher"><param
name="location">/error.jsp</param></result>
```

Results are specified in a xwork xml config file(xwork.xml) nested inside <action>. If the location param is the only param being specified in the result tag, you can simplify it as follows:

```
<action name="bar" class="myPackage.barAction">
```

```
<result name="success" type="dispatcher">
  <param name="location">foo.jsp</param>
</result>
</action>
```

or simplified

```
<action name="bar" class="myPackage.barAction">
  <result name="success" type="dispatcher">foo.jsp</result>
</action>
```

Default results

This page last changed on Jun 16, 2004 by [casey](#).

Webwork has the ability to define a default result type for your actions. Thus, you don't have to specify the result-type for results using the default. If a package extends another package and you don't specify a new default result type for the child package, then the parent package default type will be used when the type attribute is not specified in the result tag.

```
<!-- parts of xwork.xml -->
....

<result-types>
  <result-type name="dispatcher"
class="com.opensymphony.webwork.dispatcher.ServletDispatcherResult" default="true"/>
  <result-type name="redirect"
class="com.opensymphony.webwork.dispatcher.ServletRedirectResult"/>
  <result-type name="velocity"
class="com.opensymphony.webwork.dispatcher.VelocityResult"/>
</result-types>

....

<action name="bar" class="myPackage.barAction">

<!-- this result uses dispatcher, so you can omit the type="dispatcher" if you want
-->
  <result name="success">foo.jsp</result>

<!-- this result uses velocity result, so the type needs to be specified -->
  <result name="error" type="velocity">error.vm</result>

</action>

....
```

Overview

In many applications you have component objects that are required by a given class to use. In a nutshell, the IoC pattern allows a parent object (in the case of Webwork, XWork's `ComponentManager` instance) to give a resource Object to the action Object that needs it (usually an action, but it could be any object that implements the appropriate *enabler*) rather than said Object's needing to obtain the resource itself.

There are two ways of implementing IoC: Instantiation and using an enabler interface. With instantiation, a given action Object is instantiated with the resource Object as a constructor parameter. With enablers interfaces, the action will have an interface with a method, say `"setComponent(ComponentObject r);"` that will allow the resource to be passed to said action Object after it is instantiated. The `ComponentObject` is passed, because the Object implements the given interface. XWork uses *enablers* to pass components.

Why IoC?

So why is IoC useful? It means that you can develop components (generally services of some sort) in a top-down fashion, without the need to build a registry class that the client must then call to obtain the component instance.

Traditionally when implementing services you are probably used to following steps similar to these:

1. Write the component (eg an `ExchangeRateService`)
2. Write the client class (eg an XWork action)
3. Write a registry class that holds the component object (eg `Registry`)
4. Write code that gives the component object to the registry (eg `Registry.registerService(new MyExchangeRateService())`)
5. Use the registry to obtain the service from your client class (eg `ExchangeRateService ers = Registry.getExchangeRateService()`)
6. Make calls to the component from the client class (eg `String baseCurrencyCode = ers.getBaseCurrency()`)

Using IoC, the process is reduced to the following:

1. Write the component class (eg an `ExchangeRateService`)
2. Register the component class with `XWork` (eg `componentManager.addEnabler(MyExchangeRateService, ExchangeRateAware)`)
3. Write the client class, making sure it implements the enabling interface (eg an `XWork` action that implements `ExchangeRateAware`)
4. Access the component instance directly from your client action (eg `String baseCurrencyCode = ers.getBaseCurrency()`)

More advantages of Inversion of Control are the following:

1. Testability - You can more easily test your objects by passing mock objects using the enabler method rather than needing to create full containers that allow your objects to get the components they need.
2. A component describes itself. When you instantiate a component, you can easily determine what dependencies it requires without looking at the source or using trial and error.
3. Dependencies can be discovered easily using reflection. This has many benefits ranging from diagram generation to runtime optimization (by determining in advance which components will be needed to fulfill a request and preparing them asynchronously, for example).
4. Avoids the super-uber-mega-factory pattern where all the components of the app are held together by a single class that is directly tied back to other domain specific classes, making it hard to 'just use that one class'.
5. Adheres to Law of Demeter. Some people think this is silly, but in practise I've found it works much better. Each class is coupled to only what it actually uses (and it should never use too much) and no more. This encourages smaller responsibility specific classes which leads to cleaner design.
6. Allows context to be isolated and explicitly passed around. `ThreadLocals` may be ok in a web-app, but they aren't well suited for high concurrency async applications (such as message driven applications).

Writing Component Classes

In [XW:XWork](#) the actual component class can be virtually anything you like. The only constraints on it are that it must be a concrete class with a default constructor so that XWork can create instances of it as required. Optionally, a component may implement the Initializable and/or Disposable interfaces so it will receive lifecycle events just after it is created or before it is destroyed. Simply:

```
public class MyComponent implements Initializable, Disposable {
    public void init () {
        //do initialization here
    }

    public void dispose() {
        //do any clean up necessary before garbage collection of this component
    }
}
```

Component Dependencies

One feature that is not immediately obvious is that it is possible for components to depend on other components. For example if the ExchangeRateService described above depended on a Configuration component, XWork will pass the Configuration component through to the ExchangeRateService instance after ExchangeRateService is instantiated. Note that XWork automatically takes care of initializing the components in the correct order, so if A is an action or component that depends on B and C, and B depends on C and if A, B, and C have not been previously instantiated, the ComponentManager will in the following order:

1. Instantiate C and call it's init() method if it implements Initializable.
2. Instantiate B, then using the enabler method, set C to be used by B
3. Call B's init() method, if it implements Initializable.
4. Set B using B's enabler method to be used by A.

And so on and so forth. Of course, if there are instances of B or C that would be reused in this case, those instances would be passed using the enabler method rather than a new instance.

Writing Enablers

An enabler should consist of just a single method that accepts a single parameter. The parameter class should either be the component class that is to be enabled, or one of the component's superclasses. XWork does not care what the name of the enabler's method is.

Here is an example of what the ExchangeRateAware enabler might look like:

```
publicinterface ExchangeRateAware {  
    public void setExchangeRateService(ExchangeRateService exchangeRateService);  
}
```

Note that typically an enabler would be an interface, however there is nothing to prevent you from using a class instead if you so choose.

Writing "Enabler-aware" Actions

All an action needs to do is implement the relevant enabler interface. XWork will then call the action's enabler method just prior to the action's execution. As a simple example:

```
public class MyAction extends ActionSupport implements ExchangeRateAware {  
    ExchangeRateService ers;  
  
    public void setExchangeRateService(ExchangeRateService exchangeRateService) {  
        ers = exchangeRateService;  
    }  
  
    publicString execute() throws Exception {  
        System.out.println("The base currency is " + ers.getBaseCurrency());  
    }  
}
```

If you have an object that is not an action or another component, you must explicitly tell XWork to supply any enabled components to your object by calling `componentManager.initializeObject(enabledObject);`

Using an external reference resolver

You can also use an external reference resolver in XWork, i.e., references that will be resolved not by XWork itself. One such example is using an external resolver to

integrate XWork with the [Spring Framework](#)

You just need to write an external reference resolver and then tell XWork to use it in the package declaration:

```
<package
  name="default"
  externalReferenceResolver="com.atlassian.xwork.ext.SpringServletContextReferenceResolver">
```

Now, to use external references you do something like this:

```
<external-ref name="foo">Foo</external-ref>
```

Where the name attribute is the setter method name and Foo is the reference to lookup.

For more details and sample code about this integration, take a look at the javadocs to the `com.opensymphony.xwork.config.ExternalReferenceResolver` class (unfortunately unavailable online) and at [XW-122](#)

-Chris

XWork-specific language features

The biggest addition that XWork provides on top of OGNL is the support for the ValueStack. While OGNL operates under the assumption there is only one "root", XWork's ValueStack concept requires there be many "roots".

For example, suppose we are using standard OGNL (not using XWork) and there are two objects in the OgnlContext map: "foo" -> foo and "bar" -> bar and that the foo object is also configured to be the single **root** object. The following code illustrates how OGNL deals with these three situations:

```
#foo.blah // returns foo.getBlah()
#bar.blah // returns bar.getBlah()
blah      // returns foo.getBlah() because foo is the root
```

What this means is that OGNL allows many objects in the context, but unless the object you are trying to access is the root, it must be prepended with a namespaces such as @bar. Now let's talk about how XWork is a little different...

In XWork, the entire ValueStack is the root object in the context. But rather than having your expressions get the object you want from the stack and then get properties from that (ie: peek().blah), XWork has a special OGNL PropertyAccessor that will automatically look at the all entries in the stack (from the top down) until it finds an object with the property you are looking for.

For example, suppose the stack contains two objects: Animal and Person. Both objects have a "name" property, Animal has a "species" property, and Person has a "salary" property. Animal is on the top of the stack, and Person is below it. The follow code fragments help you get an idea of what is going on here:

```
species    // call to animal.getSpecies()
salary     // call to person.getSalary()
name       // call to animal.getName() because animal is on the top
```

In the last example, there was a tie and so the animal's name was returned. Usually this is the desired effect, but sometimes you want the property of a lower-level object. To do this, XWork has added support for indexes on the ValueStack. All you have to do is:

```
[0].name    // call to animal.getName()  
[1].name    // call to person.getName()
```

Accessing static properties

OGNL supports accessing static properties as well as static methods. As the OGNL docs point out, you can explicitly call statics by doing the following:

```
@some.package.ClassName@FOO_PROPERTY  
@some.package.ClassName@someMethod()
```

However, XWork allows you to avoid having to specify the full package name and call static properties and methods of your action classes using the "vs" prefix:

```
@vs@FOO_PROPERTY  
@vs@someMethod()  
  
@vs1@FOO_PROPERTY  
@vs1@someMethod()  
  
@vs2@BAR_PROPERTY  
@vs2@someOtherMethod()
```

"vs" stands for "value stack". The important thing to note here is that if the class name you specify is just "vs", the class for the object on the top of the stack is used. If you specify a number after the "vs" string, an object's class deeper in the stack is used instead.

Differences from the WebWork 1.x EL

Besides the examples and descriptions given above, there are a few major changes in the EL since WebWork 1.x. The biggest one is that properties are no longer accessed with a forward slash (/) but with a dot (.). Also, rather than using ".." to traverse down the stack, we now use "[n]" where n is some positive number. Lastly, in WebWork 1.x one could access special named objects (the request scope attributes to be exact) by using "@foo", but now special variables are accessed using "#foo". However, it is important to note that "#foo" does NOT access the request attributes. Because XWork is not built only for the web, there is no concept of "request attributes", and thus "#foo" is merely a request to another object in the OgnlContext other than the root.

Old Expression	New Expression
foo/blah	foo.blah
foo/someMethod()	foo.someMethod()
../bar/blah	[1].bar.blah
@baz	not directly supported, but #baz is similar
.	top or [0]

WebWork-specific named objects

name	value
parameters['foo']	request parameter ['foo'] (request.getParameter())
request['foo']	request attribute ['foo'] (request.getAttribute())
session['foo']	session attribute 'foo'
application['foo']	ServletContext attributes 'foo'

Getting Started

This page last changed on Jun 17, 2004 by [casey](#).

This site is geared towards developers that have an understanding towards certain technologies. Before diving into how Webwork works and running demos, it is recommended that you review the concepts below:

- Java
- Servlets, JSP, and Tag Libraries
- JavaBeans
- HTML and HTTP
- Web Containers (ex. Tomcat)
- XML

Website & downloads

This site is set up with many features. Here are links to help you around:

- [Download Webwork](#) - download Webwork Distribution
- [Webwork Mailing List](#) - [Browse mail archive](#) or [post a question](#). The list is full of active developers, contributors, and power users. This is the best and quickest way to get a question answered.
- [CVS](#) - Browse CVS and source at java.net
- [Webwork Wiki](#) - Powered by [Confluence](#), the professional J2EE wiki
- [Webwork Bugs & Issues](#) - Powered by [JIRA:Bug & Issue Traking System](#)
- [OpenSymphony Home](#)

What's included in the distro

The distribution contains the following directory layout:

```
docs/  
lib/  
src/  
src/template/  
webwork-(VERSION).jar  
webwork-example.war  
webwork-migration.jar
```

The docs directory contains the current Javadocs, the document you are reading, as well as JUnit reports for the build. The lib directory contains the required as well as

the optional dependencies for Webwork:

```
lib/  
  core/  
  migration/  
  optional/
```

Note that none of the optional packages are required to use Webwork. If you wish to use certain features such as JasperReports and FreeMarker results, you must include the optional packages.

Webwork also comes packaged with all the source files and the templates for the JSP tags.

Installing

The following illustrates how your web application should be set up. Copy the webwork-(VERSION).jar, all the *.jar files in /lib/core and any necessary optional *.jar files in /lib/optional to your webapp/lib directory. Also, copy the /src/template directory into your webapp/ directory. Your webapp should look similar to this:

```
/mywebapp/  
/mywebapp/template/  
/mywebapp/META-INF/  
/mywebapp/WEB-INF/  
/mywebapp/WEB-INF/classes/  
/mywebapp/WEB-INF/lib/  
/mywebapp/WEB-INF/lib/CORE&OPTIONAL *.jar  
/mywebapp/WEB-INF/web.xml
```

Onward to [Configuration](#) or the [Webwork Tutorial](#)

Running demos

In order to run webwork applications and demos, you need to have a servlet/jsp engine. If you don't, we suggest you learn about [Apache Tomcat](#), which is a free Servlet container from the Apache Jakarta Project, or Resin, from [Caucho Technology](#), which is free for non-commercial use. Once you have a Servlet container setup, you can install the webwork example applications (*.war) and any other demos by placing the .war file inside the containers webapp directory. Example of location with tomcat:

```
<TOMCAT_HOME>/webapps/webwork-example.war
```

After the war file is in the correct location, start your web container and access your application through a web browser with the following url.

`http://HOST:PORT/webwork-example`

Packages

This page last changed on Jun 17, 2004 by [casey](#).

Overview

Packages are a way to group Actions, Results, Result Types, Interceptors and Stacks into a logical unit that shares a common configuration. Packages are similar to objects in that they can be extended and have individual parts overridden by "sub" packages.

Packages

The package element has one required attribute, "name", which acts as the key for later reference to this package. The "extends" attribute is optional and allows one package to inherit the configuration of one or more previous packages including all interceptor, interceptor-stack, and action configurations. Note that the configuration file is processed sequentially down the document, so the package referenced by an "extends" should be defined above the package which extends it. The "abstract" optional attribute allows you to make a package abstract, which will allow you to extend from it without the action configurations defined in the abstract package actually being available at runtime.

Attribute	Required	Description
name	yes	key to for other packages to reference
extends	no	inherits package behavior of the package it extends
namespace	no	see Namespaces
abstract	no	declares package to be abstract (no action configurations required in package)

Sample usage of packages in xwork.xml

```
<package name="bar" extends="webwork-default" namespace="/foo/bar">
  <interceptors>
    <interceptor-stack name="barDefaultStack">
      <interceptor-ref name="debugStack"/>
      <interceptor-ref name="defaultStack"/>
    </interceptor-stack>
  </interceptors>
</package>
```

```

    <action name="Bar" class="com.opensymphony.xwork.SimpleAction">
      <interceptor-ref name="barDefaultStack"/>
    </action>

    <action name="TestInterceptorParamInheritance"
class="com.opensymphony.xwork.SimpleAction">
      <interceptor-ref name="test">
        <param name="expectedFoo">expectedFoo</param>
      </interceptor-ref>
    </action>

    <action name="TestInterceptorParamInheritanceOverride"
class="com.opensymphony.xwork.SimpleAction">
      <interceptor-ref name="test">
        <param name="foo">foo123</param>
        <param name="expectedFoo">foo123</param>
      </interceptor-ref>
    </action>
  </package>

  <package name="abstractPackage" namespace="/abstract" abstract="true">
    <action name="test" class="com.opensymphony.xwork.SimpleAction"/>
  </package>

  <package name="nonAbstractPackage" extends="abstractPackage"
namespace="/nonAbstract"/>

  <package name="baz" extends="default" namespace="baz">
    <action name="commandTest" class="com.opensymphony.xwork.SimpleAction">
      <param name="foo">123</param>
      <result name="error" type="chain">
        <param name="actionName">bar</param>
      </result>
      <interceptor-ref name="static-params"/>
    </action>
    <action name="myCommand" class="com.opensymphony.xwork.SimpleAction"
method="commandMethod">
      <param name="bar">456</param>
      <result name="success" type="chain">
        <param name="actionName">foo</param>
      </result>
      <interceptor-ref name="logger"/>
    </action>
  </package>

  <package name="multipleInheritance" extends="default,abstractPackage,bar"
namespace="multipleInheritance">
    <action name="testMultipleInheritance"
class="com.opensymphony.xwork.SimpleAction">
      <result name="success" type="chain">
        <param name="actionName">foo</param>
      </result>
      <interceptor-ref name="barDefaultStack"/>
    </action>
  </package>

```

Inversion of Control

This page last changed on Jun 18, 2004 by [plightbo](#).

Inversion of control is a way to handle dependencies of objects. In WebWork, objects that have their dependencies managed are called "components". For an overview of Inversion of Control (also referred to now as Dependency Injection), please read Martin Fowler's article on IoC at <http://www.martinfowler.com/articles/injection.html>. Besides WebWork's IoC container, there are numerous other containers available for you to use, including [Spring](#) and [Pico](#).

- [IoC Overview](#)
- [Xwork's Component Architecture](#)
- [How Webwork Uses Components](#)
- [Configuration of Components in Webwork and XWork](#)

Pico

This page last changed on Jun 18, 2004 by [plightbo](#).

Pico is an Inversion of Control container available at <http://picocontainer.codehaus.org>. There have been several reports of integration between WebWork and Pico. We recommend that you build your own ObjectFactory to do this integration.

Configuration

This page last changed on Jun 18, 2004 by [plightbo](#).

WebWork has two main configuration files you need to be aware of: web.xml and xwork.xml. Here you will find out all the information you need for both WebWork's required and optional configuration files.

1. [web.xml](#)
2. [xwork.xml](#)
3. [webwork-default.xml](#)
4. [Results](#)
 - a. [Global results](#)
 - b. [Default results](#)
5. [Interceptors](#)
6. [Packages](#)
7. [Namespaces](#)
8. [Reloading configuration](#)
9. [webwork.properties](#)
10. [velocity.properties](#)

Below are all the files that you may need to be aware of.

File	Optional	Location (relative to webapp)	Purpose
web.xml	no	/WEB-INF/	Web deployment descriptor to include all necessary WebWork components
xwork.xml	no	/WEB-INF/classes/	Main configuration, contains result/view types, action mappings, interceptors, etc
webwork.properties	yes	/WEB-INF/classes/	WebWork properties
webwork-default.xml	yes	/WEB-INF/lib/webwork-default.xml	Default configuration that should be included in xwork.xml
velocity.properties	yes	/WEB-INF/classes/	Override the default

			velocity configuration
validators.xml	yes	/WEB-INF/classes/	Define input validators to be used later
components.xml		/WEB-INF/classes/	Define IOC components
taglib.tld	no	/WEB-INF/lib/webwork- 1.2.3 1.2.3.jar	Webwork tag library descriptor

WebWork's Supported Views - JSP

JavaServer Pages (JSP) is an important view technology that WebWork (WW) supports. This section provides you the building blocks you need to use JSP as a view technology.

Reference Pages

1. [Non-UI Tags](#)
2. [UI Tags](#)
 - a. [Templates](#) (how UI tags render html)
 - b. [Themes](#)

Background

So what is JSP? From a technical perspective, JSP is "a page created by the web developer that includes JSP technology-specific tags, declarations, and possibly scriptlets, in combination with other static (HTML or XML) tags. A JSP page has the extension .jsp; this signals to the web server that the JSP engine will process elements on this page." See JSP for more info.

In addition, JSP "provides a simplified, fast way to create web pages that display dynamically-generated content. The JSP specification, developed through an industry-wide initiative led by Sun Microsystems, defines the interaction between the server and the JSP page, and describes the format and syntax of the page." See JSP for more info.

So you create your first JSP and deploy it to your web server. The first time a user requests your page, your servlet runner such as Tomcat will compile the JSP into a servlet class and use this class to render your web page to the user. So, if the end result of this work is a servlet, why not just write a servlet? Because servlets don't offer you an easy means of abstracting web presentation from web content. In the early days, people developed web sites exclusively with servlets. A developer would use the servlet to dynamically output HTML code. The typical way of doing this was to write out HTML code - `out.write("<HTML>..");`. This approach became a nightmare to maintain and costly to develop since it required a Java developer.

When JSP technology arrived, developers began using this new technology for their presentation. The early JSP specification had limited support for custom tags so early adopters would write Java scriptlets and JavaScript to create dynamic sites. Mixing Java and HTML was less than ideal because the Java code itself is within tags. This makes the JSP difficult to read and comprehend. This approach does work and may be fine for simple sites but it is less than adequate for complex sites. Complex sites require the ability to abstract the design and its logic to a form that is manageable and maintainable.

JSP has since matured and now provides rich support for custom tags. Developers are now writing libraries of custom tags to perform simple and complex tasks. These custom tags abstract functionality in a simple semantic form that is consistent with HTML or XML. With well-written JSP tags, web developers can develop and maintain complex web sites with NO Java code in their JSP.

WebWork's JSP tags

Writing custom tags can at first be a daunting task but no worries, WW provides an extensive JSP tag library to help you develop web sites from simple to complex. This tag library is grouped into [Non-UI Tags](#) and [UI Tags](#). The major difference between the two is that UI tags have an implied JSP template associated with them that will render HTML form controls or a set of HTML tags to produce a composite output such as a table. These templates are just defaults and will probably be all you need for your development needs. But if you need more, WW gives you the ability to override the default and insert your own template. In addition, you can group templates together under a directory and logically refer to the directory as a theme.

Themes give you the ability to skin your web site. A skin doesn't just mean different graphics, colors, etc. but it can mean different views for different browsing technologies. For instance, you might have a theme for WAP, HTML, and DHTML.

Non-UI tags provide support for control flow, internationalization, WW's Actions, iterators, text, JavaBeans and more.

A more exhaustive description of the tags can be found in the appendix on the WW taglib: [Non-UI Tags](#), [UI Tags](#).

An example

The best way to introduce how to use JSP as a view technology with WW is to walk through a simple but complete example. The example we will examine in detail is Webshop app which you can find on the index.jsp page if you deployed WW's examples. WW provides many examples to give you a better understanding of its features and capabilities. Webshop app is a good example to review because it utilizes an array of WW features.

Before we jump into the details of the example, let's look at the application's use case. Listed below is a simple normal flow of a shopper purchasing a CD.

1. Webshop app use case flow User selects desired language.
2. User is presented with a list of CDs. This list is an aggregate of album, artist, and price for each CD.
3. User adds CD to their shopping cart by selecting desired CD and quantity.
4. User continues to add or remove CDs from their shopping cart.
5. User checks out when they want to make a purchase and the shopping cart contents are displayed to the user.
6. User can shop again if desired.

Step 1 - User selects desired language

So let's begin walking through this application. When you select this application from the example index page, its link points to `webwork/i18n/index.jsp`. Upon examining this page, we learn that it forwards the user to an action - `i18n.Language.action`. By default, WW maps URIs with `*.action` pattern to servlet `ServletDispatcher`. This class is responsible to retrieve an appropriate Action class and execute it. If the result of the execution is assigned a view (JSP), then the user will be forwarded to this view.

`ServletDispatcher`'s task of determining an appropriate action begins with the URI of the request. In our case, the dispatcher is passed the URI `i18n.Language.action`. This URI maps directly to an action class named `Language` in the `webwork.action.test.i18n` package. Notice that our URI was not `webwork.action.test.i18n.Language.action` as it actually is packaged but just `i18n.Language.action`. The reason is that we redefined `webwork.action.packages` property in our `webwork.properties` file with the line shown below. This override allows us to drop `webwork.action.test` prefix from any action reference.

```
webwork.action.packages=webwork.action.test, webwork.action.standard
```

Now let's examine action `Language`. The first thing we discover is that it extends `Shop`

which extends `ActionSupport`. This class is a base class that you will extend most of the time. This class gives you access to a set of useful functionality that we will see used later. In addition, it provides the default flow of execution. Actions by default will have their `execute` method called when an URI or alias maps to it. When `execute` is called on `ActionSupport`, it determines if the URI was suffixed with a command such as `!foo`. `foo` in this example is the command you want executed. This command will be translated into a `doFoo()` method call that `ActionSupport` will call on your action. This allows you to call specific methods on your action.

Now let's examine action `Language`. The first thing we discover is that it extends `Shop` which extends `ActionSupport`. This class is a base class that you will extend most of the time. This class gives you access to a set of useful functionality that we will see used later. In addition, it provides the default flow of execution. Actions by default will have their `execute` method called when an URI or alias maps to it. When `execute` is called on `ActionSupport`, it determines if the URI was suffixed with a command such as `!foo`. `foo` in this example is the command you want executed. This command will be translated into a `doFoo()` method call that `ActionSupport` will call on your action. This allows you to call specific methods on your action.

Our URL was not a command, so `ActionSupport` will first call `doValidate()` method if you provide one. This allows you to perform validation before execution begins. Since we don't have this method, `ActionSupport` then calls our `doExecute()`. For our example, `Language` checks to see if you have set a desired language. Since we have not, it returns `ERROR`. The servlet dispatcher will now check the view mappings for `i18n.Language.error`. Looking at `views.properties` located in `WEB-INF/classes`, we find a section of mappings for our application shown below. For our error, we see that the alias maps to `language.jsp`. So the dispatcher will forward the user to that page.

Webshop view mapping:

```
# Webshop (I18N example adaptation)
i18n.Shop.success=shop.jsp
i18n.Add.success=shop.jsp
i18n.Delete.success=shop.jsp
i18n.Checkout.success=checkout.jsp
i18n.Language.success=shop.jsp
i18n.Language.error=language.jsp
i18n.Restart.success=shop.jsp
i18n.Cart.success=cart.jsp
i18n.CDList.success=cdlist.jsp
```

So what really happens when my action is fetched? By default, WW is setup with `DefaultActionFactory` as your `ActionFactory`. This factory chains together other factories that get called trying to fulfill the request of retrieving the desired action.

Take for example our Language action. The servlet dispatcher will call DefaultActionFactory and ask it to return the appropriate action. The factory then places this request on the chain. Once on the chain, each factory will either ignore the request and pass it up the chain, do some processing on the action and pass it up the chain, and/or return the action. Listed below is the chain of factory proxies that comprises DefaultActionFactory.

1. ParametersActionFactoryProxy - This proxy will call action setters for matching parameters.
2. PrepareActionFactoryProxy - This proxy will call prepare() method on action. This will allow actions to do any preparation work needed before validation or execution.
3. ContextActionFactoryProxy - This proxy will set action's context. Actions can implement *Aware interfaces that will tell this proxy what to set on your action.
4. CommandActionFactoryProxy - This proxy will strip the command from the URI and set the command on the action.
5. AliasingActionFactoryProxy - This proxy will retrieve the appropriate action name if the URI was an alias.
6. JspActionFactoryProxy - This proxy will wrap a JSP and make it an action.
7. PrefixActionFactoryProxy - This proxy will use the prefixes setup in the properties file to find the appropriate action class.
8. XMLActionFactoryProxy - This proxy will retrieve the an XML action if required.
9. ScriptActionFactoryProxy - This proxy will wrap a JavaScript and execute it.
10. JavaActionFactory - This proxy will return the appropriate Java action object.

As you can see, a lot is happening under the hood of DefaultActionFactory. So lets return back to our example beginning on language.jsp where we are to select our desired language. On this page, the user is presented with a radio list of languages. Lets take a closer look at the code to render this control.

```
<webwork:action name="'il8n.LanguageList'">
  <ui:radio label="'Language'" name="'language'" list="languages"/>
</webwork:action>
```

First notice the non-UI action tag. This tag will execute the action LanguageList. On execution, this class will load a locale-to-language mapping and save it to its attribute languages. The tag will then place this action object on the ValueStack so its body can reference it.

Now look at the UI tag radio. This tag will create an HTML INPUT control of type radio. The name of the control will be language, the label for the control will be Language. The values to display in the control will come from a method getLanguages() called on LanguageList action. The tag then iterates over the map using the keys for the VALUE parameter of the INPUT tag and the values for the displayed text.

The user now selects their desired language and submits the form. The form is submitted to Language again. This time a request parameter language is sent with the request. When the appropriate action is fetched again, the ParametersActionFactoryProxy will call all setter methods on the action for the request parameters passed in. This means action Language will have its setLanguage() method called. Now, when doExecute() is called we discover that the user has selected a language, which we will use to create the appropriate locale and place it in the user's session.

```
session.put("locale", locale);
```

Notice how easy it was to put information into the session. But where did this attribute come from? Remember that our action extends Shop and this class implements SessionAware. Because it is SessionAware it will have its setSession() method called providing the action with a map of the user's session. Now that we placed the locale in the session, we return SUCCESS. The dispatcher will look up the alias i18n.Language.success and see it maps to shop.jsp. The user will now be forwarded to this page to begin shopping.

Step 2 - User is presented with a list of CDs

In this step the user has already selected their language and their locale is in their session. They are now on shop.jsp. Examining the code on this page we see several uses of a non UI tag text. This tag provides the ability to fetch strings from resource bundles. The tag will recognize our locale and use the appropriate resource bundle to retrieve the correct language text.

```
<webwork:text name="'main.title'"/>
```

We also notice that this page includes another page i18n.CDList.action. The non UI tag include will invoke action CDList and upon success will include view cdlist.jsp in the shop.jsp. Action CDList will load a list of CDs from a text file into a list. After the CDs are loaded we now turn our attention to cdlist.jsp

```
CD:<webwork:include page="i18n.CDList.action" />
```

cdlist.jsp will provide a HTML SELECT form control. In this example, we build an HTML control instead of using WW's UI tag select. We use WW's non UI tag iterator to build the select control. This tag allows us to iterate over the cd list. So where did the cd list come from? Remember, that we called action CDList beforehand. After the action was called, WW placed this action on the ValueStack for reference by the view. So, WW will look for a method named getCDList() which it will find on action CDList. This method returns a list that the iterator tag will iterate over. Each iteration will output a HTML OPTION tag. Also notice the non UI tag property. For each member in the list, the property tag is retrieving album, artist, and country from the CD. Also notice that the last property is using action ComputePrice to determine the appropriate price for the CD. The semantic @pricer/computePrice(price) means use the object defined by parameter pricer and call its method getComputePrice passing in the CD's price as a parameter. Pretty cool. 😊

```
<webwork:action name="'webwork.action.test.il8n.ComputePrice'" id="pricer"/>

<select name="album">
  <webwork:iterator value="CDList">
    <option value="<webwork:property value="album"/>">
      <webwork:property value="album"/>,
      <webwork:property value="artist"/>, <webwork:property value="country"/>,
      <webwork:property value="@pricer/computePrice(price)"/>
    </option>
  </webwork:iterator>
</select>
```

Now back to shop.jsp. Examining the code further reveals another interesting aspect. Look at the UI tag textfield's label attribute shown below. Notice the attribute's value is a method call text('main.qtyLabel'). This method will retrieve an appropriate string from our resource bundle. But what action has a getText() method? Remember that action Shop extends ActionSupport. This action provides this method.

```
<ui:textfield label="text('main.qtyLabel')" name="'quantity'" value="1" size="3"/>
```

The last thing of interest about JSP shop.jsp is it includes the users shopping cart at the end.

```
<webwork:include value="'cart.jsp'" />
```

When cart.jsp (shown below) is included it retrieves user's cart items shown in the

code below. The attribute value of value translates into a couple of method calls getCart()/getItems(). The first method is called on action Shop which returns a Cart and then the next method is called on Cart. If the first method call does not retrieve a cart, Shop will create one with no items and place it in the user's session.

```
<webwork:property value="cart/items">
<webwork:if test=".">

    <center>
    <table border="0" cellpadding="0" width="100%" bgcolor='<webwork:text
name=" 'cart.bgcolor' "/>'>
    <tr>
        <td><b><webwork:text name=" 'cd.albumLabel' "/></b></td>
        <td><b><webwork:text name=" 'cd.artistLabel' "/></b></td>
        <td><b><webwork:text name=" 'cd.countryLabel' "/></b></td>
        <td><b><webwork:text name=" 'cd.priceLabel' "/></b></td>
        <td><b><webwork:text name=" 'cd.quantityLabel' "/></b></td>
    </tr>

<webwork:action name=" 'il8n.ComputePrice' " id="pricer"/>
<webwork:iterator>
    <tr>
        <webwork:property value="cd">
            <td><b><webwork:property value="album"/></b></td>
            <td><b><webwork:property value="artist"/></b></td>
            <td><b><webwork:property value="country"/></b></td>
            <td><b><webwork:property value="@pricer/computePrice(price)"/></b></td>
            </webwork:property>

            <td><b><webwork:property value="quantity"/></b></td>
            <td>
                <form action="il8n.Delete.action" method="post">
                    <input type=submit value='<webwork:text name=" 'cart.delLabel' "/>'>
                    <input type=hidden name="album" value='<webwork:property
value="cd/album"/>'>
                </form>
            </td>
        </tr>
    </webwork:iterator>

</table>
<p>
<p>
<form action="il8n.Checkout.action" method="post">
    <input type="submit" value='<webwork:text name=" 'cart.checkoutLabel' "/>'>
</form>
</center>

</webwork:if>
</webwork:property>
```

Step 3 - User adds CD to their shopping cart

Now the user sees a list of CDs specific to their locale. The user selects a CD and

enters a quantity and submits the form. The form is submitted to `i18n.Add.action` which maps to action `Add`. Again, parameters posted will be set on this action. For our scenario, this will be quantity and album. Then the action's `doExecute()` method is called and it retrieve's the user's shopping cart and adds the cd to it. The action then returns `SUCCESS` which maps to view `shop.jsp` again. This time the user's shopping cart has contents so it will be displayed below the cd list.

Step 4 - User continues to add or remove CDs

At this point, the user repeats the procedure of adding to and removing from their shopping cart. When they are done shopping, they then checkout.

Step 5 - User checks out

At this stage, the user is done shopping and has decided to checkout. The post of this form is to `i18n.Checkout.action`. This maps to action `Checkout`. This action will determine the total cost of the contents in the user's shopping cart and sets its attribute `totalPrice`. The action returns `SUCCESS` which maps to view `checkout.jsp`. This view will display the user's shopping cart contents along with its total price.

Step 6 - User can shop again if desired.

At this stage, the user has checked out and if they want they can begin shopping again. If they choose to start again, they select the `href` which points to `i18n.Restart.action`. This action will remove the user's shopping cart and return `SUCCESS` which will map to view `shop.jsp` to start this process all over again.

Summary

Hopefully, this example gave you a good start at understanding what a JSP might look like using WW's features. This example was not meant to be an exhaustive discovery of WW but just an introduction. There are other examples you can walk through on your own to provide additional insight into WW. In addition, you will want to read the sections referenced below for more specific documentation covering WW's features.

This page last changed on Jun 18, 2004 by [plightbo](#).

WebWork supports internationalization (in short, i18n) in two different places: the UI tags and the action/field error messages.

- [UI Tags](#)
- [Validation Examples](#)

Resource bundles are searched in the following order:

- ActionClass.properties
- BaseClass.properties (all the way to Object.properties)
- Interface.properties (every interface and sub-interface)
- package.properties (every of every base class, all the way to java/lang/package.properties)

To display i18n text, you can use a call to `getText()` in the property tag, or any other tag such as the UI tags (this is especially useful for labels of UI tags):

```
<ww:property value="getText('some.key')"/>
```

You may also use the text tag:

```
<ww:text name="'some.key'"/>
```

Also, note that there is an i18n tag that will push a resource bundle on to the stack, allowing you to display text that would otherwise not be part of the resource bundle search hierarchy mentioned previously.

```
<ww:i18n name="some/package/bundle"><ww:text name="'some.key'"/></ww:i18n>
```

Using a master application catalog

Struts users should be familiar with the `application.properties` resource bundle, where you can put all the messages in the application that are going to be translated. WebWork2, though, splits the resource bundles per action or model class, and you may end up with duplicated messages in those resource bundles. A quick fix for that is to create a file called `ActionSupport.properties` in `com/opensymphony/xwork` and put it on your classpath. This will only work well if all your actions subclass `ActionSupport`.

If this is not the case, another solution is to create a `ServletContextListener` and have it call `LocalizedTextUtil.addDefaultResourceBundle(String resourceName)`. You can also use a Servlet that's initialized on startup to do that.

Overview

WebWork builds on XWork's component implementation by providing lifecycle management of component objects and then making these components available to your action classes (or any other user code for that matter) as required.

Two types of classes in WebWork can use an enabler interface for inversion of control: Actions and Components. In order for an Action class to have its components set, the ComponentInterceptor must be made available for the Action to set those resources. In turn, if those components require other components to be initialized and set for their own use, those initializations take place at the time the ComponentInterceptor intercepts the action as well.

Scopes and Lifecycle

Components can be configured to exist across three different scopes in WebWork:

1. for the duration of a single request,
2. across a user session, or
3. for the entire lifetime of the web application.

WW:WebWork lazy loads components, meaning that components, no matter what scope, are initialized at the time they are used and disposed of at the end of the given lifecycle of that scope. Thus, an application scoped component, for example, will be initialized the first time a user makes a request to an action that implements the enabler interface of that component and will be disposed of at the time the application closes.

While components are allowed to have dependencies on other components they must not depend on another component that is of a narrower scope. So, for example, a session component cannot depend on a component that is only of request scope.

All components must be registered in the components.xml file, which is discussed in the Configuration section.

Obtaining a ComponentManager

During any request there are three component managers in existence, one for each scope. They are stored as an attribute called "DefaultComponentManager" in their respective scope objects. So if for example you need to retrieve the ComponentManager object for the request scope, the following code will do the trick:

```
ComponentManager cm = (ComponentManager)
request.getAttribute("DefaultComponentManager");
```

Feature Comparison

Feature	Struts	WebWork 1.x	WebWork 2.x
Action classes	Struts requires Action classes to extend an Abstract base class. This shows a common problem in Struts of programming to abstract classes instead of interfaces.	Action classes must implement the <code>webwork.Action</code> Interface. There are other Interfaces which can be implemented for other services, such as storing error messages, getting localized texts, etc. The <code>ActionSupport</code> class implements many of these Interfaces and can act as a base class. WebWork is all written to Interfaces, which allows for plugging in your own implementations.	An Action must implement the <code>com.opensymphony.xwork.Action</code> Interface, with a series of other Interfaces for other services, like in WebWork 1.x. WebWork2 has its own <code>ActionSupport</code> to implement these Interfaces.
Threading Model	Struts Actions must be thread-safe because there will only be one instance to handle all requests. This places restrictions on what can be done with Struts Actions as any resources held must	WebWork Actions are instantiated for each request, so there are no thread-safety issues. In practice, Servlet containers generate many throw-away objects per request, and one more Object does	ditto

	be thread-safe or access to them must be synchronized.	not prove to be a problem for performance or garbage collection.	
Servlet Dependency	Struts Actions have dependencies on Servlets because they get the ServletRequest and ServletResponse (not HttpServletRequest and HttpServletResponse, I've been told) when they are executed. This tie to Servlets (although not Http*) is a defacto tie to a Servlet container, which is an unneeded dependency. Servlets may be used outside a Web context, but it's not a good fit for JMS, for instance.	WebWork Actions are not tied to the web or any container. WebWork actions CAN choose to access the request and response from the ActionContext, but it is not required and should be done only when ABSOLUTELY necessary to avoid tying code to the Web.	ditto
Testability	Many strategies have sprung up around testing Struts applications, but the major hurdle is the fact that Struts Actions are so tightly tied to the web (receiving a Request and Response object). This often leads people to test Struts Actions inside	WebWork actions can be tested by instantiating your action, setting the properties, and executing them	ditto, but the emphasis on Inversion of Control makes testing even simpler, as you can just set a Mock implementation of your services into your Action for testing, instead of having to set up service registries or static singletons

	<p>a container, which is both slow and NOT UNIT TESTING.</p> <p>There is a Junit extension : Struts TestCase</p> <p>(http://strutstestcase.sourceforge.net/)</p>		
FormBeans	<p>Struts requires the use of FormBeans for every form, necessitating either a lot of extra classes or the use of DynaBeans, which are really just a workaround for the limitation of requiring FormBeans</p>	<p>WebWork 1.x allows you to have all of your properties directly accessible on your Action as regular JavaBeans properties, including rich Object types which can have their own properties which can be accessed from the web page.</p> <p>WebWork also allows the FormBean pattern, as discussed in "WW1:Populate Form Bean and access its value"</p>	<p>WebWork 2 allows the same features as WebWork 1, but adds ModelDriven Actions, which allow you to have a rich Object type or domain object as your form bean, with its properties directly accessible to the web page, rather than accessing them as sub-properties of a property of the Action.</p>
Expression Language	<p>Struts 1.1 integrates with JSTL, so it uses the JSTL EL. This EL has basic object graph traversal, but relatively weak collection and indexed property support.</p>	<p>WebWork 1.x has its own Expression language which is built for accessing the ValueStack. Collection and indexed property support are basic but good. WebWork can also be made to work directly with JSTL using the Filter described in</p>	<p>WebWork 2 uses XW:Ognl which is a VERY powerful expression language, with additions for accessing the value stack. Ognl supports very powerful collection and indexed property support. Ognl also supports powerful features like</p>

		WW1:Using JSTL seamlessly with WebWork	projections (calling the same method on each member of a collection and building a new collection of the results), selections (filtering a collection with a selector expression to return a subset), list construction, and lambda expressions (simple functions which can be reused). Ognl also allows access to static methods, static fields, and constructors of classes. WebWork2 may also use JSTL as mentioned in WW1:Using JSTL seamlessly with WebWork
Binding values into views	Struts uses the standard JSP mechanism for binding objects into the page context for access, which tightly couples your view to the form beans being rendered	WebWork sets up a ValueStack which the WebWork taglibs access to dynamically find values very flexibly without tightly coupling your view to the types it is rendering. This allows you to reuse views across a range	ditto

		of types which have the same properties.	
Type Conversion	<p>Struts FormBeans properties are usually all Strings. Struts uses Commons-Beanutils for type conversion. Converters are per-class, and not configurable per instance. Getting a meaningful type conversion error out and displaying it to the user can be difficult.</p>	<p>WebWork 1.x uses PropertyEditors for type conversion. PropertyEditors are per type and not settable per Action, but field error messages are added to the field error map in the Action to be automatically displayed to the user with the field.</p>	<p>WebWork2 uses Ognl for type conversion with added converters provided for all basic types. Type converters default to these converters, but type conversion can be specified per field per class. Type conversion errors also have a default error message but can be set per field per class using the localization mechanism in WW2 and will be set into the field error messages of the Action.</p>
Modular Before & After Processing	<p>Class hierarchies of base Actions must be built up to do processing before and after delegating to the Action classes, which can lead deep class hierarchies and limitations due to the inability to have multiple inheritance</p> <p>WW:Comparison to Struts#1</p>	Class hierarchies	<p>WebWork 2 allows you to modularize before and after processing in Interceptors. Interceptors can be applied dynamically via the configuration without any coupling between the Action classes and the Interceptors.</p>
Validation	Struts calls validate()	WebWork1.x calls	WebWork2 can use

	<p>on the FormBean. Struts users often use Commons Validation for validation. I don't know a lot about this, so I'll put some questions here:</p> <p>Because FormBean properties are usually Strings, some types of validations must either be duplicated (checking type conversion) or cannot be done?</p> <p>Can Commons Validation have different validation contexts for the same class? (I've been told yes, so that's a good thing)</p> <p>Can Commons Validation chain to validations on sub-objects, using the validations defined for that object properties class?</p>	<p>the validate() method on Actions, which can either do programmatic validations or call an outside validation framework (this is apparently the same as Struts)</p>	<p>the validate() method of WebWork and Struts and / or use the XW:Validation Framework, which is activated using an XWork Interceptor. The Xwork Validation Framework allows you to define validations in an XML format with default validations for a class and custom validations added for different validation contexts. The Xwork Validation Framework is enabled via an Interceptor and is therefore completely decoupled from your Action class. The Xwork Validation Framework also allows you to chain the validation process down into sub-properties using the VisitorFieldValidator which will use the validations defined for the properties class type and the validation context.</p>
--	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Control Of Action Execution	<p>As far as I know Struts sets up the Action object for you, and you have very little control over the order of operations. To change them I think (?) you need to write your own Servlet to handle dispatching as you want</p>	<p>The ActionFactory chain controls the order in which an Action is constructed and initialised, but this requires writing a class</p>	<p>The interceptor stacks in WebWork 2 are hugely powerful in this regard. All aspects of Action setup have been moved into Interceptor implementations (ie setting paramters from the web, validation etc), so you can control on a per action basis the order in which they are performed. For example you might want your IOC framework to setup the action before the parameters are set from the request or vice versa - you can thusly control this on a per package or per action basis with interceptor stacks.</p>
------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

References

- <http://www.mail-archive.com/opensymphony-webwork@lists.sourceforge.net/msg00995.htm>
- compares Struts development to WebWork 1.x development from the point of view of a Stuts developer who switched to WebWork
- <http://www.mail-archive.com/opensymphony-webwork@lists.sourceforge.net/msg04700.htm>
- Kind of the first draft of this comparison

Footnotes

1. Some Struts users have built the beginnings of an Interceptor framework for Struts (<http://struts.sourceforge.net/saif/>). It currently has some serious limitations (no "around" processing, just before and after) and is not part of the main Struts project

Configuration - web.xml

To configure WebWork's component management, the following lines must be added in the appropriate places to web.xml:

```
<filter><filter-name>container</filter-name><filter-class>com.opensymphony.webwork.lifecycle.Rec  
modify appropriately  
--></filter-mapping><listener><listener-class>com.opensymphony.webwork.lifecycle.SessionLifecycl
```

These settings allow WebWork to manage components across the application, session and request scopes. Note that even if one or more of the scopes are not required by your application, all three scopes need to be specified in web.xml for WebWork's component management to function correctly.

Configuration - xwork.xml

The ComponentInterceptor class is used to apply the IoC pattern to XWork actions (ie, to supply components to actions). The ComponentInterceptor should be declared in the <interceptors> block of xwork.xml as follows:

```
<interceptor name="component"  
    class="com.opensymphony.xwork.interceptor.component.ComponentInterceptor" />
```

You should ensure that any actions that are to be supplied with components have this interceptor applied. (See OS:XWork Interceptors for information on how to apply interceptors to actions.)

If you want to apply IoC to objects other than actions or other components, you will need to use the ComponentManager object directly.

Note too, that the ComponentInterceptor is applied as part of the webwork defaultStack. Thus, if you are applying the defaultStack to the action, you would include the ComponentInterceptor.

Configuration - components.xml

The components.xml file is used to specify the components that are to be available. The components specified here are loaded into XWork's ComponentManager and are then made available to any actions that are an instance of the specified enabler. The components.xml file must be placed in the root of the classpath (ie, in the WEB-INF/classes directory).

Here is an example components.xml file that configures a Counter component. The Counter object will live in session scope, and will be passed to any objects that are enabled due to their implementing the CounterAware interface:

```
<components><component><scope>session</scope><class>com.opensymphony.webwork.example.counter.Cou
```

Each component must have the following three attributes:

- *scope*: Valid values are *application*, *session* and *request*. This determines the component's lifetime. Application scope components will be created when the webapp starts up, and they will survive for the whole lifetime of the webapp. Session scoped components exist for the duration of a user session, while components in request scope only last for the duration of a single client request.
- *class*: This specifies the component's class. An instance of this object will live for the duration of the specified scope, and will be made available to any actions (or other code) as required. Note that components are lazy-loaded, so if nothing makes use of the component during its lifetime, the component will never actually be instantiated. At the moment components must have a zero argument constructor.
- *enabler*: Any actions that are an instance of the enabler class or interface will be passed an instance of the component.

Note that while components are allowed to have dependencies on other components they must not depend on another component that is of a narrower scope. So for example, a session component cannot depend on a component that is only of request scope.

Welcome to WebWork!

WebWork is a powerful web-based MVC framework built on top of a command pattern framework API called XWork. WebWork's specific features include dispatchers that handle and delegate requests, result types that support several view technologies (JSP, Velocity, JasperReports, XML, FreeMarker), and a small but powerful library of JSP tags and Velocity macros. Dispatchers invoke specified XWork actions that access and manipulate the model and provide easy access for the view to display model data. WebWork's true power is its underlying concept of simplicity and interoperability. Using WebWork will help minimize code and allow developers to concentrate more on business logic and modeling, rather than things like building Servlets.

Features

Some of WebWork's key features include:

- A flexible **Validation** framework allowing you to define validation with XML files that are automatically applied at runtime via an Interceptor and therefore completely decoupled from the Action class. New support for client side validation.
- **Type conversion** allowing you to easily convert objects from one class to another.
- A powerful **Expression Language** (EL) with OGNL allowing dynamic object graph traversal and method execution and transparent access to properties from multiple beans using a ValueStack. Webwork also has the ability to use JSTL.
- **Inversion of Control** (IoC) that manages component lifecycle and dependencies without the need to build registry classes that clients must call to obtain a component instance.
- **Velocity Templates** which are reusable UI components allowing the developer to easily customize the look & feel of web pages.
- **Interceptors** that dynamically intercept actions enabling before and/or after processing which simplify the action code itself and increase opportunities for code reuse.
- Support for **I18n**.
- Easy integration with third party software including **Hibernate, Spring, Pico, Sitemesh**.
- Support for many view technologies such as **JSP, Velocity, FreeMarker, JasperReports, XML**.
- **Packages** and **Namespaces** to manage hundreds of actions.

Background and Purpose

WebWork is a community project conducted using the Open Source process, aimed at providing tools and a framework for building complex websites in a short amount of time, that are easy to understand and easy to maintain. Java is the platform and language upon which it is based, although it supports many others as the language in which systems are built, such as JavaScript and XML.

WebWork is architecturally based upon best practices and design patterns that have proven themselves to be useful in this context. It is also based on a strong motivation to keep things as simple as possible, while maintaining flexibility (which is a difficult balancing act).

It also encourages you, as a user, to do things the way you seem fit for your needs. WebWork can be configured and used in a wide range of ways, many of which are useful depending on the context. As an example of this, WebWork supports many different ways of providing the HTML generation technology, such as JSP, the Velocity template engine, and XSLT. They are all widely different, both philosophically and technologically, but can all be used with WebWork, and different users do indeed use all of these ways. "You can't do that" is a statement that we try to avoid as much as possible, and when we can't it is often because another tool would be better suited for the task.

WebWork's Model-1 and Model-2 Support

One of the most important tasks of a web application framework is to support the concept of separation of logic, content, and presentation. If this is not done one typically gets problems with maintenance, and it also makes the construction of the application more difficult if teams are involved, since each team member usually has responsibility of a particular aspect of the application. A popular way to accomplish this separation is to use the design pattern known as Model-View-Controller . This pattern encourages the separation of code into pieces that each handles the model (aka "business logic"), the controller (aka "application logic"), and the view. With this separation in place, the next question is how the controller code and the presentation should interact. There are two popular models for how to do this, which are called Model-1 and Model-2 respectively. These two are described below.

Model-1

The basic idea in the Model-1 approach is to invoke the controller code from within your presentation layer, e.g. the JSP's or templates. If you are using JSP's this would mean that your WebWork actions are executed by using the "webwork:action" custom tag, or by invoking regular JavaBeans using the "webwork:bean" tag.

Model-2

In the Model-2 approach, the decision of what code to call and what view to present is determined by a third party, normally a servlet dispatcher. The dispatcher will decode the URL of the HTTP request, and determine what code to execute. A Java object representing the controller code is retrieved and executed, thus performing some custom application logic and business logic processing. After the execution is done, the dispatcher forwards the request to a view handler (for example a JSP), which then renders the result view using the data from the previous processing.

When to use what?

Since the controller logic and presentation generation is completely decoupled, it is possible to show different result pages depending on how the execution went. For example, if the processing went wrong an error page might be shown instead of the usual result page.

The benefits of the Model-1 approach are as follows.

- No need to create a mapping between code and presentation.
- Easy to see in the JSP or template what code is being executed for that page.
- If a part of the page requires some custom processing that can only result in success (or system failure), then that code invocation and presentation code (e.g. JSP taglib and HTML) does not have to be separated out into a new action mapping and JSP page. This will improve performance and readability.

The benefits of the Model-2 approach are as follows.

- Very clean separation between code and presentation. The same presentation page can be reused with many different actions, each of which may access data differently but present them in the same way.
- If an action processing can result in many different states, such as "success",

"need more input", or "error occurred", then using a Model-2 approach will make it trivial to map these states to different pages.

A general rule of thumb of when to use what is to use Model-1 for read-type code that can only result in the retrieved data being showed, and use Model-2 whenever the model is updated by the action or a process flow is being done.

- `$Session = HttpServletResponse;`
- `$Application = OgnlValueStack.`

The example below does the same thing as example 2 from [lesson 3](#), but now, using Freemarker templates.

xwork.xml:

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork
1.0//EN" "http://www.opensymphony.com/xwork/xwork-1.0.dtd"><xwork><!-- Include
webwork defaults (from WebWork-2.1 JAR). --><include file="webwork-default.xml"
/><!-- Configuration for the default package. --><package name="default"
extends="webwork-default"><!-- Default interceptor stack.
--><default-interceptor-ref name="defaultStack" /><!-- Action: Lesson 4.3:
HelloAction. --><action name="indexFreemarker"
class="com.opensymphony.xwork.ActionSupport"><result name="success"
type="dispatcher">/WEB-INF/ftl/lesson3/index.ftl</result></action><action
name="helloFreemarker" class="lesson03.HelloAction"><result name="error"
type="dispatcher">/WEB-INF/ftl/lesson3/index.ftl</result><result name="success"
type="dispatcher">/WEB-INF/ftl/lesson3/success.ftl</result></action></package></xwork>
```

HelloAction.java (same as lesson 3):

```
package lesson03;

import com.opensymphony.xwork.ActionSupport;

public class HelloAction extends ActionSupport {
    String person;
    public String getPerson() {
        return person;
    }
    public void setPerson(String person) {
        this.person = person;
    }
    public String execute() throws Exception {
        if ((person == null) || (person.length() == 0)) return ERROR;
        else return SUCCESS;
    }
}
```

ex02-index.ftl

```
<#assign ww=JspTaglibs["/WEB-INF/lib/webwork.tld"] />

<html>
<head>
<title>WebWork Tutorial - Lesson 4.3 - Example 1</title>
</head>
```

```

<body>

<p>Click <a href="${wwUtil.buildUrl('indexFreemarker.action')}">here</a> to reload
this page.</p>

<@ww.form name="'nameForm'" action="'helloFreemarker.action'" method="'POST'">
    <@ww.textfield label="'What is your name ?'" name="'person'" value="person"
size="20"/>
    <@ww.submit name="'submit'" value="'Submit'"/>
</@ww.form>

</body>
</html>

```

If you don't want to use WebWork's UI Tags, you could do it like this:

ex02-index-notags.ftl

```

<html>
<head>
<title>WebWork Tutorial - Lesson 4.3 - Example 1</title>
</head>

<body>

<p>Click <a href="${wwUtil.buildUrl('indexFreemarker.action')}">here</a> to reload
this page.</p>

<form name="nameForm" action="${wwUtil.buildUrl('helloFreemarker.action')}"
method="POST">
    What is your name ?
    <input type="text" name="person" value="${person}" size="20">
    <input type="submit" name="submit" value="Submit">
</form>
</body>
</html>

```

However, if you choose not to use tags, it's recommended that you use Freemarker Macros to write the form elements.

ex02-success.ftl:

```

<#assign ww=JspTaglibs["/WEB-INF/lib/webwork.tld"] />

<html>
<head>
<title>WebWork Tutorial - Lesson 4.3 - Example 1</title>
</head>
<body>

Come from the property WW tag (taglibs support) : <@ww.property value="person"/>
<br>
Come from the Freemarker lookup in the WW stack : ${person}

</body>

```

```
</html>
```

You can use either WebWork `property` tag or the Freemarker `$person` reference. Both of them return the same thing: a property from the action class.[Previous Lesson](#) | [Next Lesson](#)

Webwork is a popular, easy-to-use MVC framework, for more information on the WebWork project, please visit [WW:WebWork](#). This guide should be helpful to seasoned Java developers with previous experience in MVC frameworks. We will briefly cover the three main components on a WebWork2 based application, the configuration the action classes, and the views.

GETTING STARTED

To use WebWork as your framework for writing Java-based web applications, you need to start by installing the various libraries (these are all available in the [webwork 2.0 distribution](#)):

Core JAR files

- [webwork-2.1.jar](#)
- [xwork-1.0.1.jar](#)
- ognl-2.6.5.jar
- commons-logging.jar
- oscore-2.2.4.jar
- velocity-dep-1.3.1.jar

Optional JAR files

- jstl.jar (needed for Standard Tag Libraries)
- cos-multipart.jar
- pell-multipart.jar
- standard.jar
- mail.jar

CONFIGURATION

WebWork is built upon the Xwork framework. Xwork handles the translation requests to commands execution, but let's not worry about that right now. You need to know this information in case you were curious about the xwork JAR file, and if you want to

learn some of the more advanced features of the WebWork command structure, you can visit the [XW:XWork](#) site.

Example web.xml file

```
<?xml version="1.0" encoding="ISO-8859-1"?><!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>WebWork 2.0
Quick
Start</display-name><servlet><servlet-name>webwork</servlet-name><servlet-class>com.opensymphony
```

Servlet mappings

```
<servlet-mapping><servlet-name>webwork</servlet-name><url-pattern>*.action</url-pattern></servlet-mapping>
```

The above section will map ANY servlet called with an extension of **.action** to the WebWork base servlet, and assume it is a WebWork action class.

```
<servlet><servlet-name>velocity</servlet-name><servlet-class>com.opensymphony.webwork.views.velocity
```

The above section will startup the WebWorkVelocityServlet which is used to render Velocity pages. It also initializes the underlying Velocity templating subsystem which is required for using the WebWork UI tag libraries (see below).

Taglibs

```
<taglib><taglib-uri>webwork</taglib-uri><taglib-location>/WEB-INF/lib/webwork-2.0.jar</taglib-location>
```

The above section will load the standard WebWork tag libraries. To load more or different libraries, add more **<taglib>** calls.

Example Xwork config file (xwork.xml)

```
<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE xwork
PUBLIC
    "-//OpenSymphony Group//XWork
1.0//EN" "http://www.opensymphony.com/xwork/xwork-1.0.dtd"><xwork><include
file="webwork-default.xml"/><package name="default"
extends="webwork-default"><interceptors><interceptor name="security"
class="com.acme.LoginCheck"/><interceptor-ref
name="defaultStack"/></interceptors><default-interceptor-ref
name="security"/><action name="showForm" class="com.acme.FormAction"><result
name="success" type="dispatcher"><param
name="location">/form.jsp</param></result></action><action name="saveForm"
```

```
"com.acme.FormAction" method="processForm"><result name="success"
type="dispatcher"><param name="location">/form.jsp</param></result><result
name="missing-data" type="dispatcher"><param
name="location">/form.jsp</param></result><interceptor-ref
name="security"/></action></package></xwork>
```

ACTIONS

```
<action name="saveForm" class="com.acme.FormAction" method="processForm">
```

Actions are the basic "unit-of-work" in WebWork, they define, well, actions. An action will usually be a request, (and usually a button click, or form submit). The main action element (tag is too synonymous with JSP) has two parts, the friendly name (referenced in the URL, i.e. **saveForm.action**) and the corresponding "handler" class.

The method parameter tells WebWork which method to call based upon this action. If you leave the method parameter blank, WebWork will call the method **execute()** from the Action Interface by default. Since every Action must implement the Action Interface, this method will always be available.

RESULTS

```
<result name="missing-data" type="dispatcher"><param
name="location">/form.jsp</param></result>
```

Result tags tell WebWork what to do next after the action has been called. There are a standard set of result codes built-in to WebWork, (in the Action interface) they include:

```
String SUCCESS = "success";
String NONE    = "none";
String ERROR   = "error";
String INPUT   = "input";
String LOGIN   = "login";
```

You can extend these as you see fit. Most of the time you will have either **SUCCESS** or **ERROR**, with **SUCCESS** moving on to the next page in your application;

```
<result name="success" type="dispatcher"><param
name="location">/thank_you.jsp</param></result>
```


...and **ERROR** moving on to an error page, or the preceding page;

```
<result name="error" type="dispatcher"><param
name="location">/error.jsp</param></result>
```

You can stack as many result tags within a single action tag as you wish.

INTERCEPTORS

Interceptors allow you to define code to be executed before and/or after the execution of an action. Interceptors can be a powerful tool when writing web applications. Some of the most common implementations of an Interceptor might be:

- Security Checking (ensuring the user is logged in)
- Trace Logging (logging every action)
- Bottleneck Checking (start a timer before and after every action, to check bottlenecks in your application)

You can also group Interceptors together to create "stacks". If you wanted to do a login check, security check, and logging all before an Action call, this could easily be done with an interceptor stack.

For further Reading on Interceptors, see [XW:Interceptors](#)

For further Reading on Xwork configuration files, see [XW:Configuration](#)

ACTION CLASSES

The action classes do what they say, they handle the action. They are called by the actions specified in the **xwork.xml** file and initiated by the user in the "views". To turn your class into an action class, you simply need to extend the class **ActionSupport** or implement the **Action** interface.

Here is what our **saveForm** action looks like in its Java form (NOTE: If you look in the **xwork.xml** file above, we've overridden the action to call the **processForm** method):

```
package com.acme;
import com.opensymphony.xwork.*;

public class FormAction extends ActionSupport {

    private FormBean myFormBean = new FormBean();

    public void setFormBean(FormBean inBean) {
```

```

        myFormBean = inBean;
    }

    public FormBean getFormBean() {
        return myFormBean;
    }

    public String processForm() {

        FormParameters formParams = this.getFormBean();
        checkBizRules(formParams);
        this.saveParamsToDb(formParams);

        return SUCCESS;
    }
    ...
}

```

VIEWS

WebWork supports JSP and Velocity for your application presentation layer. For this example we will use a JSP file. Webwork comes packaged with a tag library (taglibs). You can use these taglibs as components in your JSP file. Here is an section of our **form.jsp** page:

```

<%@ taglib prefix="ww" uri="webwork" %>
<html>
<head><title>Webwork Form Example</title></head>
<body>

<ww:form name="'myForm'" action="'saveForm.action'" method="'POST'">
    <table>

        <ww:textfield label="'First Name'" name="'formBean.firstName'"
value="'formBean.firstName'"/>
        <ww:textfield label="'Last Name'" name="'formBean.lastName'"
value="'formBean.lastName'"/>

    </table>
    <input type="submit" value="Save Form"/>
</ww:form>
</body>

```

The process of events will go as follows:

1. WebWork will take notice since the URI ends in **.action** (defined in our **web.xml** files)
2. WebWork will look up the action **saveForm** in its action hierarchy and call any Interceptors that we might have defined.
3. WebWork will translate **saveForm** and decide we would like to call the method **processForm** in our class **com.acme.FormAction** as defined in our **xwork.xml** file.
4. Our method will process successfully and give WebWork the **SUCCESS** return

parameter.

5. WebWork will translate the **SUCCESS** return parameter into the location **thank_you.jsp** (as defined in **xwork.xml**) and redirect us accordingly.

SUMMARY

The purpose of this guide is to provide the user with a quick-and-dirty understanding of WebWork2. I hope we successfully briefed you on the three most important components of any WebWork based application, the configuration (including **web.xml** and **xwork.xml**), the action classes, and the views. This information should give you a starting point to experiment and become more familiar with the rising star of the open source, Model2, web frameworks.

Matt Dowell

matt.dowell@notiva.com

September 15, 2003

Non-UI Tags

This page last changed on Jul 22, 2004 by [plightbo](#).

These are tags that interact with the value stack, and control the logic of the page.

Tag Name	Description
Common Tags	
<ww:param />	Add parameters to tags that support it
<ww:property />	Fetches a value and prints it
<ww:push />	Add an object of your choice to the top of the value stack
<ww:set />	Create your own named variables
<ww:url />	Builds an encoded URL
Componentisation Tags	
<ww:action />	Provides another method to call Actions
<ww:bean />	Instantiate a bean that can be used to access functionality
<ww:include />	Used to include another page or action
Flow-Control Tags	
<ww:if />	Used to determine if a statement is true or false
<ww:elseif />	Used to determine if a statement is true or false after a previous test.
<ww:else />	Used to determine if the preceding statement was false
Iteration Tags	
<ww:iterator />	Iterate over a value
<ww:generator />	Create Iterator
<ww:append />	Append a list of iterators
<ww:subset />	Iterate over a portion of an iterable object
<ww:merge />	Merge several iterators into one
<ww:sort />	Sort an iterator

Common Tags

<ww:param />

Allows you to add parameters to tags that support adding parametric tags.

attribute	required	description
value	no	This attribute is used to pass data to the tag.
name	no	The name of the action to invoke.

You can place param tags within the body of parametric supporting tags and param will add its parameter to its parent. It evaluates the body as the value if no value is given.

In this example, each param will add its parameter to Counter. This means param will call Counter's appropriate setter method.

```
<ww:bean name="'webwork.util.Counter'" id="year">
  <ww:param name="'first'" value="text('firstBirthYear')"/>
  <ww:param name="'last'" value="2000"/>

  <ui:combobox label="'Birth year'" size="6" maxlength="4" name="'birthYear'"
list="#year"/>
</ww:bean>
```

[return to top](#)

<ww:property />

The property tag fetches a value and prints it

attribute	required	description
id	no	This attribute assigns a unique name to an element (note).
value	no	This attribute is used to pass data to the tag.
escape	no	Determines if the contents should be escaped appropriately for valid HTML characters

Some examples will illustrate these different uses:

```
Print getX().getY()  
<ww:property value="x.y"/>
```

HTML characters will be escaped by default, whereas the contents of property tags with bodies will not be escaped. This behavior can be overridden by explicitly setting the escape attribute. Quoted text that is escaped will have its outer quotes stripped.

Note also that if the property tag has an empty body, it behaves the same as having no body and prints the value, though both spaces and carriage returns constitute nonempty content.

[return to top](#)

<ww:push />

Using ww:push, you can add an object of your choice to the top of the value stack.

attribute	required	description
value	yes	This attribute is used to pass data to the tag.

This is similar to what you can do with ww:set (see below), so read both before deciding which to use.

```
<ww:push value="counter">  
  <ww:property value="count"/>  
</ww:push>
```

To make an action available on the stack:

```
<ww:action name="'SomeAction'" id="sa"/>  
<ww:push value="#sa">  
  foo = <ww:property value="foo"/>  
</ww:push>
```

[return to top](#)

<ww:set />

You can create your own named variables from within a JSP using the ww:set tag. Reference your variable later using the # variableName notation.

attribute	required	description
name	yes	Unique name for the variable, accessed as "#name".
value	no	This attribute is used to pass data to the tag.
scope	no	Scope of the variable: page, request, session, application

Sets the value of an object in the VS to a scope. If the value is not given, the top of the stack is used. If the scope is not given, the default scope is the action context which is only available in the PageContext if no action has been executed on the same request.

[return to top](#)

<ww:url />

Url builds an encoded URL. If you do not include a value, then the tag will point to the current page.

attribute	required	description
value	no	This attribute is used to pass data to the tag.
scheme	no	can be "http" or "https"
includeContext	no	Determines whether the context path should be prepended to absolute urls or not. Default is true
encode	no	Determines if the contents should be escaped appropriately for valid HTML characters
includeParams	no	The includeParams attribute may have the value 'none' (no params), 'get'(only GET params) or 'all'(GET and POST params). It is used when the url tag is used

		without a value or page attribute. Its value is looked up on the ValueStack. If no includeParams is specified then 'get' is used.
--	--	-----------------------------------------------------------------------------------------------------------------------------------

In this example, the form action value will be an url hiturl.action that is encoded.

```
<form action="<ww:url value='hiturl.action' />" method="POST">
  ...
</form>
```

In this example, we are adding name/value pairs to the URL. The URL tag will build up the URL appropriately. You can also place them in the normal way with "?"; i.e., - 'hiturl.action?user=john'.

```
<form action="<ww:url value='hiturl.action'>
  <ww:param name='user' value='john' />
</ww:url>" method="POST">
  ...
</form>
```

By default, port 80 is assumed to be the "http" port and port 443 is assumed to be the "https" port. However, some servers, such as Tomcat, use different default ports, such as 8080 and 8443. You can change these values by setting the configuration elements in webwork.properties:

- webwork.url.http.port
- webwork.url.https.port

Componentisation Tags

[return to top](#)

<ww:action />

Action tag provides another method to call Actions. This is an alternative way to invoke an action besides calling an url; i.e. - *.action that would be sent to the ServletDispatcher.

attribute	required	description
id	no	This attribute assigns a unique name to an element (note).
name	yes	The name of the action to invoke.
namespace	no	Namespace of this action
executeResult	no	Whether to execute result

If the **id** attribute is given, the executed action is assigned a name reference that can be later retrieved from the context "**#id**".

In this example, the ClientInfo action will be executed and its methods will be used to retrieve information and perform a conditional test.

```
<ww:action name="'ClientInfo'" id="cinfo"><ww:param name="detailedMode"
value="false"/></ww:action>
Browser:<ww:property value="#cinfo.browser"/><br>
Version:<ww:property value="#cinfo.version"/><br>
Supports GIF:<ww:if test="#cinfo.supportsType('image/gif') ==
true">Yes</ww:if><ww:else>No</ww:else><br>
```

[return to top](#)

<ww:bean />

Create a JavaBean and instantiate its properties. It is then placed in the ActionContext for later use.

attribute	required	description
id	no	This attribute assigns a unique name to an element (note).
name	yes	The name of the action to invoke.

In this example, Counter is used as a bean. We can now call the methods we desire. In this case, we setFirst() to first birth year which is 1975 and we setLast() to 2000. We then display a combo box using Counter as an Iterator.

```
<ww:bean name="'webwork.util.Counter'" id="year">
  <ww:param name="'first'" value="text('firstBirthYear')"/>
```

```
<ww:param name="'last'" value="2000"/>

<ui:combobox label="'Birth year'" size="6" maxlength="4" name="'birthYear'"
list="#year"/>
</ww:bean>
```

[return to top](#)

<ww:include />

Used to include another page or action.

attribute	required	description
page	no	Name of page or action.
value	no	This attribute is used to pass data to the tag.

In this example, beaninfo.jsp will introspec on people[0] which is a Person. Take a look at beaninfo.jsp example and notice how it retrieves the parent value off the ValueStack with "..".

```
<ww:property value="people[0]">
  <ww:include value="'beaninfo.jsp'"/>
</ww:property>
```

In this example, an Action is invoked.

```
<h1>RSS viewer</h1>
<ww:include value="'rss.viewer.action'"/>
```

Flow Control Tags

[return to top](#)

<ww:if />

Used to determine if a statement is true or false.

attribute	required	description
id	no	This attribute assigns a unique name to an element (note).
test	yes	This attribute is the conditional expression evaluated by WW's parser. It returns boolean true or false.

In this example, if will evaluate its body since the test condition is true. elseif and else will not evaluate.

```
<ww:if test="true == true">
  <b>if: Success</b>
</ww:if>

<ww:elseif test="true == true">
  <b>elseif: Failure</b>
</ww:elseif>

<ww:else>
  <b>else: Failure</b>
</ww:else>
```

[return to top](#)

<ww:elseif />

Used to determine if a statement is true or false after a previous test.

attribute	required	description
id	no	This attribute assigns a unique name to an element (note).
test	yes	This attribute is the conditional expression evaluated by WW's parser. It returns boolean true or false.

In this example, elseif will evaluate its body since its test condition is true and if is false.

```

<ww:if test="true == false">
  <b>if: Failures</b>
</ww:if>

<ww:elseif test="true == true">
  <b>elseif: Success</b>
</ww:elseif>

<ww:else>
  <b>else: Failure</b>
</ww:else>

```

[return to top](#)

<ww:else />

Used to determine if the preceding statement was false.

attribute	required	description
id	no	This attribute assigns a unique name to an element (note).

In this example, else will evaluate its body since both if and elseif conditions are false.

```

<ww:if test="true == false">
  <b>if: Failures</b>
</ww:if>

<ww:elseif test="true == false">
  <b>elseif: Failure</b>
</ww:elseif>

<ww:else>
  <b>else: Success</b>
</ww:else>

```

Iteration Tags

[return to top](#)

<ww:iterator />

Iterator will iterate over a value. An iterable value can be either of: java.util.Collection, java.util.Iterator, java.util.Enumeration, java.util.Map, array, XML

Node, or XML NodeList.

attribute	required	description
id	no	This attribute assigns a unique name to an element (note).
status	no	This attribute indicates the name of the IteratorStatus object to be exposed. An IteratorStatus allows one to get information about the status of the iteration: getCount(), getIndex(), isFirst(), isLast(), isEven(), isOdd().
value	no	This attribute is used to pass data to the tag.

In this example, iterator will iterate over Counter. property will output the current value which is 1 through 10.

```
<ww:bean name="'webwork.util.Counter'">
  <ww:param name="'last'" value="10"/>

  <ww:iterator>
    <ww:property/><br />
  </ww:iterator>
</ww:bean>
```

In this example, we use a couple of IteratorStatus to see where we are within iterations.

```
<h1>Testing iterator status</h1>

<ww:bean name="'webwork.util.Counter'" id="rowcounter">
  <ww:param name="'first'" value="0"/>
  <ww:param name="'last'" value="5"/>
</ww:bean>

<table border="1">
  <ww:iterator value="#rowcounter" status="rowstatus">
    <tr>
      <ww:bean name="'webwork.util.Counter'" id="colcounter">
        <ww:param name="'first'" value="0"/>
        <ww:param name="'last'" value="5"/>
      </ww:bean>
    </tr>
  </ww:iterator>
</table>
```

```

<ww:iterator value="#colcounter" status="colstatus">
  <!--
    if it is (first row) or (first column) or (last row) then
    output the column number.
  -->
  <ww:if test="#rowstatus.first==true || #colstatus.first==true ||
#rowstatus.last==true">
    <th><ww:property value="#colstatus.count"/></th>
  </ww:if>

  <ww:else>
    <td><ww:property/></td>
  </ww:else>

</ww:iterator>

</tr>
</ww:iterator>
</table>

```

Here we use the IteratorStatus determine every other row to insert an extra line break. This is very useful for shading alternate rows in an HTML table. Both even and odd attributes are available.

```

<ww:iterator status="'status'">
  <ww:if test="#status.odd == true"> <br /> </ww:if>
  <br />
</ww:iterator>

Here we use the IteratorStatus determine every fourth row to insert an extra line
break.
<ww:iterator status="'status'">
  <ww:if test="#status.modulus(4) == 0"> <br /> </ww:if>
  <br />
</ww:iterator>

```

Following are the list of operations available on the status object:

- even : boolean - returns true if the current iteration is even
- odd : boolean - returns true if the current iteration is odd
- count : int - returns the count (1 based) of the current iteration
- index : int - returns the index (0 based) of the current iteration
- first : boolean - returns true if the iterator is on the first iteration
- last : boolean - returns true if the iteration is on the last iteration
- modulus(operand : int) : int - returns the current count (1 based) modulo the given operand

[return to top](#)

<ww:generator />

Generate will create Iterators from val.

attribute	required	description
id	no	This attribute assigns a unique name to an element (note).
count	no	This attribute indicates how many items there are.
separator	no	This attribute is the character the StringTokenizer will use to create tokens.
val	yes	This attribute is the list of values the generator should use to create tokens.

In this example, two Iterators are created. One for val="foo,bar,xyzzzy" and the other for val=" ".

```
<h1>Testing append, subset, and value generators</h1>

<table border="1">
  <ww:bean name="'webwork.util.Counter'">
    <ww:param name="'last'" value="5"/>
    <ww:iterator id="colcount">
      <tr>

        <!--
          Generator will create an Iterator that has 5 items.
          The first 3 are "foo,bar,xyzzzy". Item 4 and 5 will be
          foo and bar respectively. If the count is more than
          the items, you start over.
        -->

        <ww:generator val="'foo,bar,xyzzzy'" separator="','" count="#colcount"
id="values"/>


        <!--
          Generator will create an Iterator that has infinite
          . Count=-1 means indefinite.
        -->
        <ww:generator val="' '" count="-1" id="space"/>
        <ww:append>
          <ww:param name="'source'" value="#values"/>
          <ww:param name="'source'" value="#space"/>

          <ww:subset count="6">
            <ww:iterator>
```

```

        <td width="40"><ww:property/></td>
    </ww:iterator>
</iterator:subset>
</iterator:append>
</tr>
</ww:iterator>
</ww:bean>
</table>

```

 This tag is mostly superfluous, now that we can do this in OGNL:

```

<ww:iterator value="{1, 2, 3, 4}">
</ww:iterator>

```

[return to top](#)

<ww:append />

Append will append a list of iterators. The values of the iterators will be appended and treated as one iterator. The outputs from the iterator will be in the sequence the sources were added.

attribute	required	description
id	no	This attribute assigns a unique name to an element (note).

In this example, the two iterators #values and #spaces are appended. This means #spaces values are after #values.

```

<h1>Testing append, subset, and value generators</h1>

<table border="1">
  <ww:bean name="'webwork.util.Counter'">
    <ww:param name="'last'" value="5"/>
    <ww:iterator id="colcount">
      <tr>
        <ww:generator val="'foo,bar,xyzzzy'" separator="','" count="#colcount"
id="values"/>
        <ww:generator val="' '" count="-1" id="space"/>
        <ww:append>
          <ww:param name="'source'" value="#values"/>
          <ww:param name="'source'" value="#space"/>

          <ww:subset count="6">
            <ww:iterator>
              <td width="40"><ww:property/></td>
            </ww:iterator>

```



```

        </iterator:subset>
    </iterator:append>
</tr>
</ww:iterator>
</ww:bean>
</table>

```

[return to top](#)

<ww:subset />

Subset will iterate over a portion of its source. It will start at start and continue for count. You can set count to -1 if you want to iterate until the end of source. If you do not supply a source, the current object on the ValueStack- "." will be used.

attribute	required	description
id	no	This attribute assigns a unique name to an element (note).
count	no	This attribute indicates how many items there are.
source	no	This attribute is the source the tag will use to perform work on. It may be Enumeration, Iterator, or a Collection.
start	no	This attribute indicates the index to start reading.

In this example, subset will iterate over 6 items for the current object in the ValueStack.

```

<h1>Testing append, subset, and value generators</h1>

<table border="1">
  <ww:bean name="'webwork.util.Counter'">
    <ww:param name="'last'" value="5"/>
    <ww:iterator id="colcount">
      <tr>
        <ww:generator val="'foo,bar,xyzy'" separator="','" count="#colcount"
id="values"/>
        <ww:generator val="' ' " count="-1" id="space"/>
      <ww:append>
        <ww:param name="'source'" value="#values"/>
        <ww:param name="'source'" value="#space"/>

```

```

        <ww:subset count="6">
            <ww:iterator>
                <td width="40"><ww:property/></td>
            </ww:iterator>
        </iterator:subset>
    </iterator:append>
</tr>
</ww:iterator>
</ww:bean>
</table>

```

[return to top](#)

<ww:merge />

Merge several iterators into one. It weaves them together. If one iterator runs out, it will drop off and the others will continue weaving until there are no more values.

attribute	required	description
id	no	This attribute assigns a unique name to an element (note).

In this example, #foo, #bar, and #xyzzzy iterators are merged together. So, the output will be foo, bar, xyzzzy until #foo and #xyzzzy iterators run out in which case #bar will finish.

```

Three value generators with merge and subset limits:<br>
<ww:generator val="'foo'" count="5" id="foo"/>
<ww:generator val="'bar'" count="10" id="bar"/>
<ww:generator val="'xyzzzy'" count="5" id="xyzzzy"/>
<ww:merge>
    <ww:param name="'source'" value="#foo"/>
    <ww:param name="'source'" value="#bar"/>
    <ww:param name="'source'" value="#xyzzzy"/>

    <ww:subset count="30">
        <ww:iterator status="'status'">
            <ww:property value="#status.count"/><ww:property/><br>
        </ww:iterator>
    </iterator:subset>
</iterator:merge>

```

[return to top](#)

<ww:sort />

Sort allows you to sort an iterator. It uses `Collections.sort()` given the comparator you supply.

attribute	required	description
id	no	This attribute assigns a unique name to an element (note).
comparator	yes	This attribute will be the Comparator used to sort the Collection.
source	no	This attribute is the source the tag will use to perform work on. It may be Enumeration, Iterator, or a Collection.

In this example, we sort ascending.

```
<ww:bean name="'webwork.util.Counter'" id="counter">
  <ww:param name="'first'" value="0"/>
  <ww:param name="'last'" value="5"/>
</ww:bean>

<ww:bean name="'webwork.util.Sorter'" id="sorter"/>

Ascending:<br />
<ww:sort source="#counter" comparator="#sorter.ascending">
  <ww:iterator>
    <ww:property/><br />
  </ww:iterator>
</iterator:sort>
```

In this example, we sort descending.

```
<ww:bean name="'webwork.util.Sorter'" id="sorter"/>

<ww:bean name="'webwork.util.Counter'" id="counter">
  <ww:param name="'first'" value="0"/>
  <ww:param name="'last'" value="5"/>
</ww:bean>


Descending:<br>
<ww:sort source="#counter" comparator="#sorter.descending">
  <ww:iterator>
    <ww:property/><br>
  </ww:iterator>
</iterator:sort>
```


In this example, we sort ascending over strings.

```
<ww:bean name="'webwork.util.Sorter'" id="sorter"/>

Sorting strings:<br>
<ww:generator val="'Rickard,Maurice,Hristo'" separator="','" id="names"/>
<ww:sort source="#names" comparator="#sorter.ascending">
  <ww:iterator>
    <ww:property/><br>
  </ww:iterator>
</iterator:sort>
```

Notes

 **Id** The "id" attribute assigns a name to an element. This name must be unique in a document. This attribute is the standard id supported by JSP TagSupport and is therefore always a string. You do not need to indicate a string literal as you would for the rest of WW attributes; i.e. - id="age". Instead you should use id=age".

 It's very important to note that all tags that insert something into the valuestack (like i18n or bean tags) will remove those objects from the stack on its end tag. So, if you instantiate a bean with the bean tag (<ww:bean name="br.univap.fcc.sgpw.util.FormatterHelper">) that bean will be available on the valuestack only until the </ww:bean> tag.

SiteMesh can be found at <http://www.opensymphony.com/sitemesh>

Integrating WebWork with SiteMesh is amazingly simple: you don't have to do anything in fact. WebWork stores all its value stack information in the request attributes, meaning that if you wish to display data that is in the stack (or even the ActionContext) you can do so by using the normal tag libraries that come with WebWork. That's it!

One thing to note is when you want to pass a value from a decorated page to a decorator using the **<ww:set>** tag, you need to specify a scope (request, session, application) if the decorated page is invoked directly (not a result of an action). By default if no action has been executed and no scope was specified, the set value will only be available from the same PageContext.

Projects Using WebWork

This page last changed on Jun 28, 2004 by [botah](#).

- [Atlassian Confluence](#) - Commercial Wiki and knowledge management system using WebWork 2.0, Hibernate, Spring, and Velocity
- [Chemist Australia](#) - Online Pharmacy
- [DriveNow](#) - last minute Australian car rentals
- [OpenReports](#) - an open source web based reporting application that uses WebWork 2.0, Velocity, and Hibernate
- [eSage Group](#) is a consulting company that uses it for all their client engagements. Additionally, its used for their internal systems
- [Filmweb](#) - Polish Film Portal
- [TeraMEDICA](#) - WebWork is used in TeraMEDICA's commercial TI2m product, which performs intelligent image management for the healthcare enterprise. Specifically, WebWork is a key component of the system's management interface.
- [EBIA COBRA and 401K Benefits Site](#) provides law reviews for employee benefits like COBRA and 401K. The site moved from all struts to a current architecture of about 50% WebWork and 50% Struts. We are trying to move it all over to WebWork. This site is also a great example of porting from Tiles to SiteMesh.

Simple validators

This page last changed on Jul 10, 2004 by [unkyaku](#).

The following validators are included in the default validators.xml:

Name	JavaScript aware	Description
#required		Field value must have a value (non-null)
#requiredstring	x	Field value is non-null and has a length > 0
#int	x	Field value must be an integer and within a range
#date		Field value must be a date (the format is based on locale) and within a range
#expression		A given OGNL expression is evaluated against the value stack and must return true. This is mostly usefully for cross-field validation. Errors are added as action errors
#fieldexpression		A given OGNL expression is evaluated against the value stack and must return true. This is similar to expression but errors are added as field errors
#email	x	Field value must be a valid e-mail address
#url	x	Field value must be a valid url
visitor		Allows you to forward validation to object properties of your action using the objects own validation files
conversion		Add conversion errors from

		ActionContext to field errors of the action. This does the same thing as WebWorkConversionErrorInterceptor
--	--	----------------------------------------------------------------------------------------------------------------------------

Note: the above name can be changed if you supply your own validators.xml.

required

In SimpleAction-validation.xml:

```
<validators><field name="bar"><field-validator type="required"><message>You must enter a value for bar.</message></field-validator></field></validators>
```

[#top](#)

requiredstring

In LoginAction-validation.xml:

```
<validators><field name="userName"><field-validator type="requiredstring"><message>You must enter an username.</message></field-validator></field></validators>
```

The error is shown if request parameter **userName** is missing or an empty string

[#top](#)

int

```
<validators><field name="foo"><field-validator type="int"><param name="min">0</param><param name="max">100</param><message key="foo.range">Could not find foo.range!</message></field-validator></field></validators>
```

[#top](#)

date


```
<validators><field name="startDate"><field-validator type="date"><param
name="min">12/22/2002</param><param name="max">12/25/2002</param><message>The date
must be between 12-22-2002 and
12-25-2002.</message></field-validator></field></validators>
```

[#top](#)

expression

```
<validators><validator type="expression"><param name="expression">foo >
bar</param><message>Foo must be greater than Bar. Foo = ${foo}, Bar =
${bar}.</message></validator></validators>
```

The validator is not associated with a single field. You may need to place your expression within a CDATA if it contains bad xml characters.

[#top](#)

fieldexpression

```
<validators><field name="productCode"><field-validator type="fieldexpression"><param
name="expression">name.length() == 5</param><message>Product code must be 5
characters, it is currently
'${productCode}'</message></field-validator></field></validators>
```

[#top](#)

email

```
<validators><field name="email"><field-validator type="email"><message>You must
enter a valid email address.</message></field-validator></field></validators>
```

The address must be in the format xxx@yyy.com|net|gov|org|edu|info|mil|biz|tv|...

[#top](#)

url

```
<validators><field name="companyUrl"><field-validator type="url"><message>You must  
enter a valid URL.</message></field-validator></field></validators>
```

[#top](#)

Validation

This page last changed on Jul 16, 2004 by [unkyaku](#).

WebWork relies on [XWork's](#) validation framework to enable the application of input validation rules to your Actions before they are executed. This section only provides the bare minimum to get you started and focuses on WebWork's extension of the XWork validators to support client-side validation. Please consult XWork's [validation framework documentation](#) for complete details.

Reference pages

1. [Simple validators](#)
2. [Visitor validation](#)
3. [Client-Side Validation](#)
4. [Validation Examples](#)

Registering Validators

Validation rules are handled by validators, which must be registered with the ValidatorFactory. The simplest way to do so is to add a file name **validators.xml** in the root of the classpath (/WEB-INF/classes) that declares all the validators you intend to use. The syntax of the file is as follows:

```
<validators><validator name="required"
    class="com.opensymphony.webwork.validators.JavaScriptRequiredFieldValidator"/><validator
name="requiredstring"
    class="com.opensymphony.webwork.validators.JavaScriptRequiredStringValidator"/><validator
name="stringlength"
    class="com.opensymphony.xwork.validator.validators.StringLengthFieldValidator"/><validator
name="int"
    class="com.opensymphony.webwork.validators.JavaScriptIntRangeFieldValidator"/><validator
name="date"
    class="com.opensymphony.webwork.validators.JavaScriptDateRangeFieldValidator"/><validator
name="expression"
    class="com.opensymphony.xwork.validator.validators.ExpressionValidator"/><validator
name="fieldexpression"
    class="com.opensymphony.xwork.validator.validators.FieldExpressionValidator"/><validator
name="email"
    class="com.opensymphony.webwork.validators.JavaScriptEmailValidator"/><validator
name="url"
    class="com.opensymphony.webwork.validators.JavaScriptURLValidator"/><validator
name="visitor"
    class="com.opensymphony.webwork.validators.VisitorFieldValidator"/><validator
name="conversion"
    class="com.opensymphony.xwork.validator.validators.ConversionErrorFieldValidator"/></val
```

This list declares all the validators that comes with WebWork.

Turning on Validation

All that is required to enable validation for an Action is to put the ValidationInterceptor in the interceptor refs of the action (see [xwork.xml](#)) like so:

```
<interceptor name="validator"
class="com.opensymphony.xwork.validator.ValidationInterceptor"/>
```

Note: The default **validationWorkflowStack** already includes this.

Defining Validation Rules

Validation rules can be specified:

1. Per Action class: in a file named ActionName-validation.xml
2. Per Action alias: in a file named ActionName-alias-validation.xml
3. Inheritance hierarchy and interfaces implemented by Action class: WebWork searches up the inheritance tree of the action to find default validations for parent classes of the Action and interfaces implemented

Here is an example for SimpleAction-validation.xml:

```
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator
1.0//EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.dtd"><validators><field
name="bar"><field-validator type="required"><message>You must enter a value for
bar.</message></field-validator><field-validator type="int"><param
name="min">6</param><param name="max">10</param><message>bar must be between ${min}
and ${max}, current value is ${bar}.</message></field-validator></field><field
name="date"><field-validator type="date"><param name="min">12/22/2002</param><param
name="max">12/25/2002</param><message>The date must be between 12-22-2002 and
12-25-2002.</message></field-validator></field><field name="foo"><field-validator
type="int"><param name="min">0</param><param name="max">100</param><message
key="foo.range">Could not find
foo.range!</message></field-validator></field><validator type="expression"><param
name="expression">foo > bar</param><message>Foo must be greater than Bar. Foo =
${foo}, Bar = ${bar}.</message></validator></validators>
```

Here we can see the configuration of validators for the SimpleAction class. Validators (and field-validators) must have a **type** attribute, which refers to a name of an Validator registered with the ValidatorFactory as above. Validator elements may also have <param> elements with name and value attributes to set arbitrary parameters into the Validator instance. See below for discussion of the message element.

Each Validator or Field-Validator element must define one message element inside the validator element body. The message element has 1 attributes, key which is not required. The body of the message tag is taken as the default message which should be added to the Action if the validator fails.

Key gives a message key to look up in the Action's ResourceBundles using getText() from LocaleAware if the Action implements that interface (as ActionSupport does). This provides for Localized messages based on the Locale of the user making the request (or whatever Locale you've set into the LocaleAware Action).

After either retrieving the message from the ResourceBundle using the Key value, or using the Default message, the current Validator is pushed onto the ValueStack, then the message is parsed for `${...}` sections which are replaced with the evaluated value of the string between the `${` and `}`. This allows you to parameterize your messages with values from the Validator, the Action, or both. Here is an example of a parameterized message:

```
bar must be between ${min} and ${max}, current value is ${bar}.
```

This will pull the min and max parameters from the IntRangeFieldValidator and the value of bar from the Action.

Package changes

Webwork1.x was separated into two projects, XWork and Webwork. From this, several classes have been moved to different package names.

- ActionSupport has moved from **webwork.ActionSupport** to **com.opensymphony.xwork.ActionSupport**
 - doExecute() no longer exists, override execute()
 - the methods addError and addErrorMessage are now addFieldError and addActionError respectively

Configuration changes

- **actions.xml/views.properties needs to be converted to xwork.xml**

If you're using an actions.xml file to configure your webwork 1, you can use the attached XSLT to convert the actions.xml file to a vanilla xwork.xml file.

To apply this XSLT, you'll need to do the following:

Get a copy of the XSLT. You can find the latest version in CVS in webwork/src/etc/actions.xsl . Next, find yourself an XSLT rendering engine. Xalan is a good choice and can be found at <http://xml.apache.org/xalan-j/index.html>

Finally, do the conversion.

```
java org.apache.xalan.xslt.Process -IN actions.xml -XSL actions.xsl -OUT xwork.xml
```

Remember that you'll need to Xalan libraries in your classpath to run the above command.

If you want to look at these pages directly in your browser, I recommend user Internet Explorer as it automagically formats XML documents reasonably. There one caveat though. WW1 had a way to shorten the declaration of actions by allowing you to specify a package prefix in webwork.properties file. Since this information is outside the actions.xml file, the XSLT is unable to take advantage of it. Consequently, you might need to edit the xwork.xml file to update the class names.

WebWork 1.x configuration used a pull paradigm to load action configurations when they are asked for, whereas WebWork2 builds the configuration up-front to make the configuration queryable. The webwork.MigrationConfiguration must therefore act as an adapter between these two paradigms. It does this by returning a custom

RuntimeConfiguration which first tries the default XWork Configuration (which, by default, loads configuration information from a file named "xwork.xml" in the root of the classpath) and then attempts to load action configuration using the Configuration classes from WebWork 1.x. In this way, an application can be slowly converted over to WebWork2 while reusing the configuration and Actions from a WebWork 1.x application. One caveat in this is that your migrated application MUST be rebuilt against the WebWork2 and migration jar files, as the classloader will rightly recognize that the webwork.Action and webwork.ActionSupport in WebWork 1.x are not the same as those provided by the migration jar files. Other than that, it should be seamless (and let us know if it isn't).

If the webwork.MigrationRuntimeConfiguration does not find the action configuration using the RuntimeConfiguration from the supplied RuntimeConfiguration (which is acquired from the Xwork DefaultConfiguration and will load configurations from all of the sources configured for XWork / WebWork2), it will build an ActionConfiguration by instantiating an Action using the ActionFactories from WebWork 1.x. The ActionFactory stack used is a subset of the default ActionFactory stack used in WebWork 1.x:

```
factory = new JavaActionFactory();
factory = new ScriptActionFactoryProxy(factory);
factory = new XMLActionFactoryProxy(factory);
factory = new PrefixActionFactoryProxy(factory);
factory = new JspActionFactoryProxy(factory);
factory = new CommandActionFactoryProxy(factory);
factory = new AliasingActionFactoryProxy(factory);
factory = new CommandActionFactoryProxy(factory);
factory = new ContextActionFactoryProxy(factory);
```

Some of the ActionFactory classes have been left out as they are handled by Interceptors in WebWork2. If the Action instance is created (meaning that the configuration has been found in the webwork.properties or actions.xml files used by the WebWork 1.x configuration classes) a parameter Map is created by introspecting the Action instance. A Map is needed for results and, again, WebWork 1.x used a pull paradigm to find results when they were needed, so a LazyResultMap is created which extends HashMap and overrides get() to look up the Result configuration if it has not previously been loaded. If the result ends in the Action suffix (defaulting to ".action"), then a ChainingResult is created, otherwise a ServletDispatcherResult is created. Using the Action class of the instantiated Action, the Map of parameters introspected from the Action instance, and the LazyResultMap, a new ActionConfig is created. The ActionConfig is saved into a special Package, "webwork-migration", so that it will pick up the default Interceptor stack defined for that package. The "webwork-migration" package is defined in a webwork-migration.xml file which is included in the migration jar file and which is automatically added to the Xwork configuration providers when

the MigrationConfiguration is used:

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork
1.0//EN" "http://www.opensymphony.com/xwork/xwork-1.0.dtd"><xwork><include
file="webwork-default.xml"/><package name="webwork-migration" abstract="true"
extends="webwork-default"><interceptors><interceptor-stack
name="migrationStack"><interceptor-ref name="timer"/><interceptor-ref
name="logger"/><interceptor-ref name="chain"/><interceptor-ref
name="static-params"/><interceptor-ref name="prepare"/><interceptor-ref
name="params"/><interceptor-ref
name="workflow"/></interceptor-stack></interceptors><default-interceptor-ref
name="migrationStack"/></package></xwork>
```

Here we can see that a number of the functions previously performed by ActionFactories in WebWork 1.x are now replaced by Interceptors in WebWork2, including the parameters, the chaining, calling prepare(), and the workflow (which was formerly implemented in ActionSupport in WebWork 1.x).

By creating and saving the ActionConfig in the PackageConfig for the "webwork-migration" package, the ActionConfig for the migrated Action is available for future calls, obviating the need to re-parse the old configuration files and making the configuration for the Action available for querying via the configuration API for tools such as the configuration browser.

Tag Changes

The biggest change is the use of OGNL for accessing object properties. Properties are no longer accessed with a forward slash "/" but with a dot "." Also, rather than using ".." to traverse down the stack, we now use "[n]" where n is some positive number. Lastly, in WebWork 1.x one could access special named objects (the request scope attributes to be exact) by using "@foo", but now special variables are accessed using "#foo". However, it is important to note that "#foo" does NOT access the request attributes. "#foo" is merely a request to another object in the OgnlContext other than the root. See [OGNL](#) reference for more details.

[property](#)

The property tag is now only used to print out values from the stack. In WW1, it was also used to set a variable in the scope, and to push properties to the top of the stack. These functions are now performed by the [set](#) and [push](#) tags.

[action tag](#)

The action tag does not evaluate the body section any more and does not push the executed action onto the ValueStack. Instead, use the "**id**" attribute to assign a name to the action and reference it as "**#id**".

Examples

Lets enumerate some examples of differences between code snips using [WW:WebWork](#) and [WW:WebWork](#).

- *New JSP syntax*

There are numerous changes in syntax. First of all there are new tags and secondly there is a new expression language. Here's a small example:

Webwork 1

```
<ww:property value="a/b"><ww:property value="foo" /></ww:property>
```

Webwork 2

```
<ww:push value="a.b"><ww:property value="foo" /></ww:push>
```

One can note that the "push" tag doesn't just push it pops too at the end of the tag. Surprise! Also note the "." instead of the "/" for traversing object properties.

- *List errors posted by an Action*

Webwork 1

```
<webwork:if test="hasErrorMessages == true">
  ERROR:<br /><font color="red"><webwork:iterator
value="errorMessages"><webwork:property/><br
/></webwork:iterator></font></webwork:if>
```

Webwork 2

```
<webwork:if test="hasErrors()">
  ERROR:<br /><font color="red"><webwork:iterator
value="actionErrors"><webwork:property/><br
/></webwork:iterator></font></webwork:if>
```

Update your web.xml file

- If you're using Velocity for views, you'll need to make sure you have the following snippet. Specifically note that the `<load-on-startup>` tag is now required so that the servlet can initialize some important Velocity properties.

```
<servlet>
  <servlet-name>velocity</servlet-name>
  <servlet-class>com.opensymphony.webwork.views.velocity.WebWorkVelocityServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

- Set the property **webwork.velocity.configfile** in your `_webwork.properties_`. For example:

```
webwork.velocity.configfile=velocity.properties
```

WebWork will use this file to initialize the Velocity engine. The search path for the file is:

1. context root (web root)
2. WEB-INF/
3. classpath

- Additional Steps:
 1. If you used the `<ww:action taglib` in 1.3... you used to reference the java Action classname. In 2.x this reference is now the action name not the class. you will need to change all your old references in your view.

ResultException doesn't exist anymore

It might be possible to copy WW1's ResultException, and write an Interceptor that catches the ResultExceptions and add the result of `getMessage()` to the actionErrors of the executed Action and return `ResultException.getResult()`.

Maybe it would be possible to include ResultException in WW2 too to make migration easier?!

DateFormatter doesn't exist anymore

It can be replaced by directly using **java.text.DateFormat**

addError(String, String) in webwork.action.ActionSupport

has been removed

The new method to use is **addFieldError(String, String)**.

addErrorMessage(String) in webwork.action.ActionSupport has been removed

The new method is now **addActionError(String)**.

webwork.util.ValueStack has been removed

The ValueStack is **com.opensymphony.xwork.util.OgnlValueStack**

The old methods **pushValue** and **popValue** are renamed to simply **push** and **pop**.

An instance of the ValueStack can be obtained by using **ActionContext.getContext().getValueStack** instead of the old **ValueStack.getStack()**.

*Aware-Interfaces have been removed

Instead of implementing **ServletRequestAware** etc the **[Servlet]ActionContext.getXXX**-methods can be used to obtain application-map, request, response etc.

CommandDriven interface removed

The **CommandDriven** interface is removed. It is not necessary to implement a special interface when working with commands anymore. Use the **method** attribute in your **action**-Element in xwork.xml to tell xwork which method to invoke on your action.

isCommand(String) method has been removed

You can see which alias you're accessing by doing this:

```
ActionContext.getContext().getActionInvocation().getProxy().getActionName()
```

WebWork has very advanced type conversion support thanks to XWork. For more details please consult [XW:Type Conversion](#).

A Simple Example

Type conversion is great for situations where you need to turn a String in to a more complex object. Because the web is type-agnostic (everything is a String in HTTP), WebWork's type conversion features are very useful. For instance, if you were prompting a user to enter in coordinates in the form of a String ("3, 22"), you could have WebWork do the conversion both from String to Point and from Point to String.

Using this "point" example, if your action (or another compound object in which you are setting properties on) has a corresponding className-conversion.properties file, WebWork will use the configured type converters for conversion to and from strings. So turning "3, 22" in to new Point(3, 22) is done by merely adding the following entry to ClassName-conversion.properties:

```
point = com.acme.PointConverter
# note: PointerConverter must implement ognl.TypeConverter
#       or extend ognl.DefaultTypeConverter
```

Your type converter should be sure to check what class type it is being requested to convert. Because it is used for both to and from strings, you will need to split the conversion method in to two parts: one that turns Strings in to Points, and one that turns Points in to Strings.

After this is done, you can now reference your point (<ww:property value="point"/>) and it will be printed as "3, 22" again. As such, if you submit this back to an action, it will be converted back to a Point once again.

NOTE: Type conversion should not be used as a substitute for `i18n`. It is not recommended to use this feature to print out properly formatted dates. Rather, you should use the `i18n` features of WebWork (and consult the JavaDocs for JDK's `MessageFormat` object) to see how a properly formatted date should be displayed.

Overview

Interceptors are objects that dynamically intercept Action invocations. They provide the developer with the opportunity to define code that can be executed before and/or after the execution of an action. They also have the ability to prevent an action from executing. Interceptors provide developers a way to encapsulate common functionality in a re-usable form that can be applied to one or more Actions. See [XW:Interceptors](#) for further details. Below describes built in Webwork interceptors.

Webwork & XWork Interceptors

Interceptor classes are also defined using a key-value pair specified in the xwork configuration file. The names specified below come specified in [webwork-default.xml](#). If you extend the webwork-default package, then you can use the names below. Otherwise they must be defined in your package with a name-class pair specified in the <interceptors> tag.

Name	Description
timer	Outputs how long the action (including nested interceptors and view) takes to execute
logger	Outputs the name of the action
chain	Makes the previous action's properties available to the current action. Commonly used together with <result type="chain"> (in the previous action.)
static-params	Sets the xwork.xml defined parameters onto the action. These are the <param> tags that are direct children of the <action> tag.
params	Sets the request parameters onto the action.
model-driven	If the action implements ModelDriven, pushes the getModel() result onto the valuestack.
component	Enables and makes the components

	available to the Actions. Refer to components.xml
token	Checks for valid token presence in action, prevents duplicate form submission
token-session	Same as above, but storing the submitted data in session when handed an invalid token
validation	Performs validation using the validators defined in {Action}-validation.xml
workflow	Calls the validate method in your action class. If action errors created then it returns the INPUT view.
servlet-config	Give access to HttpServletRequest and HttpServletResponse (think twice before using this since this ties you to the Servlet api)
prepare	If the action implements Preparable, calls its prepare() method.
conversionError	adds conversion errors from the ActionContext to the Action's field errors
fileUpload	an interceptor that adds easy access to file upload support; read the javadocs for more info
execAndWait	an interceptor that executes the action in the background and then sends the user off to an intermediate waiting page; read the javadocs for more info

For more information about the chain interceptor, see [WW:Chaining Interceptor](#).

Client-Side Validation

This page last changed on Jul 22, 2004 by [unkyaku](#).

WebWork adds support for client-side validation on top of XWork's standard validation framework. You can enable it on a per-form basis by specifying **validate="true"** in the `<ww:form>` tag:

```
<ww:form name="'test'" action="'javascriptValidation'" validate="true">
  ...
</ww:form>
```

You must specify a *name* for the form in order for client-side validation.


You should also make sure you provide the correct *action* and *namespace* attributes to the `<ww:form>` tag. For example, if you have an Action named "submitProfile" in the "/user" namespace, you must use


```
<ww:form namespace="'/user'" action="'submitProfile'" validate="true">
  ...
</ww:form>
```


While the following will "work" in the sense that the form will function correctly, client-side validation will not:

```
<ww:form action="'/user/submitProfile.action'" validate="true">
  ...
</ww:form>
```

Of course, all the standard [validation configuration](#) steps still apply. Client-side validation uses the same validation rules as server-side validation. If server-side validation doesn't work, then client-side validation won't work either.

 Not all validators support client-side validation. Only validators that implement `ScriptValidationAware` support this feature. Refer to the list of WebWork validators to see which ones do so.

 Note that the *required* attribute on many WebWork [UI tags](#) has nothing to do with client-side validation.

 **Upgrade Alert:** This feature was introduced in WebWork 2.1. If upgrading from a previous version, make sure you are using the correct validators in [validators.xml](#). You must be using the

com.opensymphony.webwork.validators.JavaScriptRequired*Validator version of the standard XWork validators.

Building a Validator that supports client-side validation

Any validator can be extended to support client-side validation by implementing the **com.opensymphony.webwork.validators.ScriptValidationAware** interface:

```
public interface ScriptValidationAware extends FieldValidator {  
    public String validationScript(Map parameters);  
}
```

The value returned by **validationScript** will be executed on the client-side before the form is submitted if client-side validation is enabled. For example, the **requiredstring** validator has the following code:

```
public String validationScript(Map parameters) {  
    String field = (String) parameters.get("name");  
    StringBuffer js = new StringBuffer();  
  
    js.append("value = form.elements['" + field + "'].value;\n");  
    js.append("if (value == \"\") {\n");  
    js.append("\talert('\" + getMessage(null) + "\");\n");  
    js.append("\treturn '\" + field + "';\n");  
    js.append("}\n");  
    js.append("\n");  
  
    return js.toString();  
}
```

Only JavaScript is supported at this time.

webwork.properties

This page last changed on Aug 04, 2004 by [jcarreira](#).

WebWork uses a number of properties that can be changed to fit your needs. To change them, specify your values in webwork.properties in /WEB-INF/classes. The list of properties can be found in default.properties (inside webwork.jar):

```
### These can be used to set the default HTTP and HTTPS ports
webwork.url.http.port = 80
webwork.url.https.port = 443

### This can be used to set your locale and encoding scheme
webwork.locale=en_US
webwork.il8n.encoding=ISO-8859-1

### Parser to handle HTTP POST requests, encoded using the MIME-type
multipart/form-data
#webwork.multipart.parser=cos
webwork.multipart.parser=pell

# uses javax.servlet.context.tempdir by default
webwork.multipart.saveDir=
webwork.multipart.maxSize=2097152

### Load custom property files (does not override webwork.properties!)
webwork.custom.properties=application,com/webwork/extension/custom

# extension for actions. This uses by the webwork:form tag
webwork.action.extension=action

### Standard UI theme
# Change this to reflect which path should be used for JSP control tag templates by
default
webwork.ui.theme=xhtml
webwork.ui.templateDir=template

### Configuration reloading
# This will cause the configuration to reload xwork.xml when it is changed
webwork.configuration.xml.reload=false

### Location of velocity.properties file. defaults to velocity.properties
webwork.velocity.configfile = velocity.properties

### Comma separated list of VelocityContext classnames to chain to the
WebWorkVelocityContext
webwork.velocity.contexts =

### Override the default VelocityManager with a custom implementation
webwork.velocity.manager.classname=path.and.classname

### Load custom default resource bundles
#webwork.custom.il8n.resources=testmessages,testmessages2
```

WebWork 2 UI Tags

Click on a tag to find more information on the tag. Note that all UI tags are now evaluated against the value stack so you need to (single) quote your literal string values.

The actual rendering of these tags can be customized. The location of the tag templates is defined by the `webwork.ui.theme` property in `webwork.properties`. See the [Themes](#) reference for more details.

Tag Name	Description
<ww:checkbox />	render a checkbox input field
<ww:checkboxlist />	render a list of checkboxes
<ww:combobox />	Widget that fills a text box from a select
<ww:component />	render a custom ui widget
<ww:file />	renders a file select input field
<ww:form />	defines an input form
<ww:hidden />	render a hidden field
<ww:label />	render a label that displays read-only information
<ww:password />	render a password input field
<ww:radio />	renders a radio button input field
<ww:select />	renders a select element
<ww:submit />	renders a submit button
<ww:table />	renders a table
<ww:tabbedpane />	renders a tabbedpane
<ww:textarea />	renders a text area input field
<ww:textfield />	renders an input field of type text
<ww:token />	Stop double-submission of forms
Internationalization Tags	
<ww:i18n />	put resource bundle in stack
<ww:text />	renders string from bundle

([back to the top](#))

<ww:checkbox />

Renders an HTML <input> element of type checkbox, populated by the specified property from the OgnlValueStack.

Sample Usages

```
JSP
<ww:checkbox label="'checkbox test'" name="'checkboxField1'" value="aBoolean"
fieldValue="'true'"/>

Velocity
#tag( Checkbox "label='checkbox test'" "name='checkboxField1'" value="aBoolean"
fieldValue="'true'" )

HTML (simple template, aBoolean == true)
<input type="checkbox" name="checkboxField1" value="true" checked="true" />
```

Attributes

Name	Required	Description
id	no	HTML id attribute
name	yes	HTML name attribute
label	no	Text used as label in template
labelposition	no	Alignment of label (left,right,center)
required	no	Is field required for form submission
value	no	Boolean which if true adds "checked=true" to tag
fieldValue	yes	the actual HTML value attribute of the checkbox
tabindex	no	HTML tabindex attribute
onchange	no	HTML onchange attribute
cssClass	no	HTML class attribute
cssStyle	no	HTML style attribute
theme	no	Theme to use

template	no	Name of template to use
----------	----	-------------------------

([back to the top](#))

<ww:checkboxlist />

Creates a series of checkboxes from a list. Setup is like <ww:select /> or <ww:radio />, but creates checkbox tags.

Attributes

Name	Required	Description
id	no	HTML id attribute
name	yes	HTML name attribute
value	no	Data to pass as field value
label	no	Text used as label in template
labelposition	no	Alignment of label (left,right,center)
required	no	Is field required for form submission
list	no	Iterable source to populate from
listKey	no	Property of list objects to get field value from
listValue	no	Property of list objects to get field content from
cssClass	no	HTML class attribute
cssStyle	no	HTML style attribute
theme	no	Theme to use
template	no	Name of template to use

([back to the top](#))

<ww:combobox />

The combo box is basically an HTML INPUT of type text and HTML SELECT grouped together to give you a combo box functionality. You can place text in the INPUT control by using the SELECT control or type it in directly in the text field.

In this example, the SELECT will be populated from id=year attribute. Counter is itself an Iterator. It will span from first to last.

```
<ww:bean name="'webwork.util.Counter'" id="year">
  <ww:param name="'first'" value="text('firstBirthYear')"/>
  <ww:param name="'last'" value="2000"/>

  <ww:combobox label="'Birth year'" size="6" maxlength="4" name="'birthYear'"
list="#year"/>
</ww:bean>
```

Attributes

Name	Required	Description
id	no	HTML id attribute
name	yes	HTML name attribute
value	no	Data to pass as field value
list	no	Iterable source to populate from
size	no	HTML size attribute
maxlength	no	HTML maxlength attribute
disabled	no	HTML disabled attribute
tabindex	no	HTML tabindex attribute
onkeyup	no	HTML onkeyup attribute
onchange	no	HTML onchange attribute
cssClass	no	HTML class attribute
cssStyle	no	HTML style attribute
label	no	Text used as label in template
labelposition	no	Alignment of label (left,right,center)
theme	no	Theme to use
template	no	Name of template to use

([back to the top](#))

<ww:component />

Renders an custom UI widget using the specified templates. Additional objects can be

passed in to the template using the param tags. Objects provided can be retrieve from within the template via `$parameters._paramname_`.

In the bottom JSP and Velocity samples, two parameters are being passed in to the component. From within the component, they can be accessed as `$parameters.get('key1')` and `$parameters.get('key2')`. Velocity also allows you reference them as `$parameters.key1` and `$parameters.key2`.

Currently, your custom UI components **must** be written in Velocity.

Remember: the value params will always be resolved against the OgnlValueStack so if you mean to pass a string literal to your component, make sure to wrap it in quotes i.e. `value="value1"` otherwise, the the value stack will search for an Object on the stack with a method of `getValue1()`. (now that i've written this, i'm not entirely sure this is the case. i should verify this manana)

Sample Usages

```
JSP
<ww:component template="/my/custom/component.vm"/>
  or

<ww:component template="/my/custom/component.vm">
  <ww:param name="key1" value="value1"/>
  <ww:param name="key2" value="value2"/>
</ww:component>

Velocity
#tag( Component "template=/my/custom/component.vm" )

  or

#bodytag( Component "template=/my/custom/component.vm" )
  #param( "key1" "value1" )
  #param( "key2" "value2" )
#end
```

Attributes

Name	Required	Description
id	no	HTML id attribute
name	yes	HTML name attribute
value	no	Data to pass as field value
label	no	Text used as label in template
labelposition	no	Alignment of label (left,right,center)

required	no	Is field required for form submission
cssClass	no	HTML class attribute
cssStyle	no	HTML style attribute
theme	no	Theme to use
template	yes	Name of template to use

([back to the top](#))

<ww:file />

File upload form tag.

Sample Usages

```
<ww:file name="'uploadedFile'" />
```

Attributes

Name	Required	Description
id	no	HTML id attribute
name	yes	HTML name attribute
value	no	Data to pass as field value
label	no	Text used as label in template
labelposition	no	Alignment of label (left,right,center)
required	no	Is field required for form submission
accept	no	HTML accept attribute: list of file types to accept
onchange	no	HTML onchange attribute
cssClass	no	HTML class attribute
cssStyle	no	HTML style attribute
theme	no	Theme to use
template	no	Name of template to use

([back to the top](#))

<ww:form />

An HTML form.

Sample Usages

```
<ww:form name="myForm" action="submit.action">
...
</ww:form>
```

```
<ww:form name="myForm" action="submit" namespace="/foo">
...
</ww:form>
```

Attributes

Name	Required	Description
name	yes	HTML name attribute
value	no	Data to pass as field value
openTemplate	no	Template to use for the open form tag
action	yes	HTML action attribute
namespace	no	Namespace of action
method	no	'GET' or 'POST'
enctype	no	HTML enctype attribute
validate	no	Whether to use client-side validation
cssClass	no	HTML class attribute
cssStyle	no	HTML style attribute
theme	no	Theme to use
template	no	Name of template to use for the closing form tag

Note: The **action** attribute works in two different ways. The original (and deprecated) format is shown in the first sample usage. The newer format allows you to specify just the action name (as well as a namespace, if there is one) as defined in xwork.xml. This is **required** if you wish to do any form of [Client-Side Validation](#) ([back to the top](#))

<ww:hidden />

An HTML input tag of type "hidden".

Sample Usages

```
<ww:hidden name=" 'someName' " value="value"/>
```

Name	Required	Description
id	no	HTML id attribute
name	yes	HTML name attribute
value	no	Data to pass as text
cssClass	no	HTML class attribute
cssStyle	no	HTML style attribute
theme	no	Theme to use
template	no	Name of template to use

([back to the top](#))

<ww:label />

An HTML LABEL that will allow you to output label:name combination that has the same format treatment as the rest of your UI controls.

Attributes

Name	Required	Description
id	no	HTML id attribute
name	no	HTML name attribute
value	no	Data to pass as text
label	no	Text used as label in template
labelposition	no	Alignment of label (left,right,center)
required	no	Is field required for form submission
cssClass	no	HTML class attribute
cssStyle	no	HTML style attribute

theme	no	Theme to use
template	no	Name of template to use

In this example, a label is rendered. The label is retrieved from a ResourceBundle by calling ActionSupport's getText() method giving you an output of User name: a label.

```
<ww:label label="text('user_name')" name="'a label'"/>
```

([back to the top](#))

<ww:password />

An HTML input tag of type password.

In this example, a password control is displayed. For the label, we are calling ActionSupport's getText() to retrieve password label from a resource bundle.

```
<ww:password label="text('password')" name="'password'" size="10" maxlength="15"/>
```

Name	Required	Description
id	no	HTML id attribute
name	yes	HTML name attribute
value	no	Data to pass as text
size	no	HTML size attribute
maxlength	no	HTML maxlength attribute
disabled	no	HTML disabled attribute
readonly	no	HTML readonly attribute
onkeyup	no	HTML onkeyup attribute
tabindex	no	HTML tabindex attribute
onchange	no	HTML onchange attribute
show	no	Redisplay value (security concerns)
label	no	Text used as label in template
labelposition	no	Alignment of label (left,right,center)
required	no	Is field required for form submission

cssClass	no	HTML class attribute
cssStyle	no	HTML style attribute
theme	no	Theme to use
template	no	Name of template to use

([back to the top](#))

<ww:radio />

An HTML Radiobox UI widget

In this example, a radio control is displayed with a list of genders. The gender list is built from attribute id=genders. WW calls getGenders() which will return a Map. For examples using listKey and listValue attributes, see the section select tag.

```
<ww:action name=" 'GenderMap' " id="genders"/>
<ww:radio label=" 'Gender' " name=" 'male' " list="#genders.genders"/>
```

Attributes

Name	Required	Description
id	no	HTML id attribute
name	yes	HTML name attribute
value	no	Data to pass as field value
list	no	Iterable source to populate from
listKey	no	Property of list objects to get field value from
listValue	no	Property of list objects to get field content from
disabled	no	HTML disabled attribute
tabindex	no	HTML tabindex attribute
onchange	no	HTML onchange attribute
label	no	Text used as label in template
labelposition	no	Alignment of label (left,right,center)
theme	no	Theme to use

template	no	Name of template to use
required	no	Is field required for form submission

([back to the top](#))

<ww:select />

Generates a select list filled with a specified list. The "listKey" attribute is the property to pull from each item in the list to generate the value of the <option> tag for that item. The "listValue" attribute fills the label of the option (the display name). One great feature is that it will auto-select the appropriate option based on the "value" attribute. If the value matches the current listKey, that option will be selected.

```
<ww:select label=" 'Users' "
  name=" 'userId' "
  listKey="id"
  listValue="name"
  list="app.users"
  value="app.user.id"
  onchange=" 'chooseUser(this)' "
/>
```

will create the following (if getApp().getUser().getId() == 2):

```
<tr>
  <td>Users</td>
  <td>
    <select name="userId" onchange="chooseUser(this)">
      <option value="1">
        User Number One
      </option>
      <option value="2" selected>
        User Number Two
      </option>
    </select>
  </td>
</tr>
```

Of course, the <td> formatting and such depends on the [template](#) you are using.

Sample Usages

```
<ww:select label=" 'Pets' "
  name=" 'petIds' "
  list="petDao.pets"
  listKey="id"
  listValue="name"
```

```
multiple="true"  
size="3"  
required="true"  
</>
```

Attributes

Name	Required	Description
id	no	HTML id attribute
name	yes	HTML name attribute
value	no	Data to pass as field value
required	no	Is field required for form submission
list	no	Iterable source to populate from. If the list is a Map (key, value), the Map key will become the option "value" parameter and the Map value will become the option body.
listKey	no	Property of list objects to get field value from
listValue	no	Property of list objects to get field content from
emptyOption	no	Whether or not to add an empty (--) option after the header option
multiple	no	Is this a multiple select?
size	no	Character width
disabled	no	HTML disabled attribute
tabindex	no	HTML tabindex attribute
onchange	no	HTML onchange attribute
headerKey	no	Key for first item in list
headerValue	no	Value for first item in list
label	no	Text used as label in template
labelposition	no	Alignment of label (left,right,center)

cssClass	no	HTML class attribute
cssStyle	no	HTML style attribute
theme	no	Theme to use
template	no	Name of template to use

([back to the top](#))

<ww:submit />

Submit button.

Sample Usages

```
<ww:submit value="'Submit'"/>
```

Name	Required	Description
id	no	HTML id attribute
name	no	HTML name attribute
value	yes	Data to pass as text
align	no	HTML align attribute
label	no	Text used as label in template
labelposition	no	Alignment of label (left,right,center)
cssClass	no	HTML class attribute
cssStyle	no	HTML style attribute
theme	no	Theme to use
template	no	Name of template to use

([back to the top](#))

<ww:tabbedpane />

Tabbed pane allows you to associated tabs with different views. When the user clicks the tab, that view will render and become current. Tabbed pane is basically a table which includes the selected tab's page in its bottom row.

In this example, a tabbed pane is rendered with tabs aligned right. See WW's example

for a more complete picture.

```
<p><ww:tabbedPane id="tp1" contentName="'tabs1'" tabAlign="'RIGHT'"/></p>
```

Name	Required	Description
id	no	HTML id attribute
theme	no	Theme to use
contentName	yes	???
tabAlign	no	Tab horizontal alignment: RIGHT,LEFT,CENTER

([back to the top](#))

<ww:table />

HTML table using the displaytag library.

Sample Usages

```
<ww:table modelName="'/result'" sortable="true" sortColumn="0" sortOrder="ASC">
  <ww:param name="'columnHidden(1)'" value="true"/>
  <ww:param name="'columnDisplayName(2)'" value="'New Display Name'"/>
  <ww:param name="'columnRenderer(0)'" value="#dateRenderer"/>
  <ww:param name="'columnRenderer(2)'" value="#linkRenderer"/>
  <ww:param name="'columnRenderer(4)'" value="#intRenderer"/>
</ww:table>
```

Name	Required	Description
theme	no	Theme to use
modelName	yes	???
sortable	no	Whether the columns are sortable
sortColumn	no	Index of initial sorted column
sortOrder	no	ASC,DESC,NONE

([back to the top](#))

<ww:textarea />

Renders a <textarea></textarea> tag.

Sample Usages


```
<ww:textarea label="'Comments'" name="'comments'" cols="30" rows="8"/>
```

Attributes

Name	Required	Description
id	no	HTML id attribute
name	yes	HTML name attribute
value	no	Data to pass as field value
required	no	Is field required for form submission
rows	no	HTML rows attribute
cols	no	HTML cols attribute
readonly	no	HTML readonly attribute
disabled	no	HTML disabled attribute
tabindex	no	HTML tabindex attribute
onkeyup	no	HTML onkeyup attribute
label	no	Text used as label in template
labelposition	no	Alignment of label (left,right,center)
cssClass	no	HTML class attribute
cssStyle	no	HTML style attribute
theme	no	Theme to use
template	no	Name of template to use

([back to the top](#))

<ww:textfield />

In this example, a text control is rendered. The label is retrieved from a ResourceBundle by calling ActionSupport's getText() method.

```
<ww:textfield label="text('user_name')" name="'user'"/>
```

Attributes

Name	Required	Description
id	no	HTML id attribute
name	yes	HTML name attribute
value	no	Data to pass as field value
required	no	Indicates if this field is required for form submission
size	no	HTML size attribute
maxlength	no	HTML maxlength attribute
readonly	no	HTML readonly attribute
disabled	no	HTML disabled attribute
tabindex	no	HTML tabindex attribute
onkeyup	no	HTML onkeyup attribute
onchange	no	HTML onchange attribute
label	no	Text used as label in template
labelposition	no	Alignment of label (left,right,center)
cssClass	no	HTML class attribute
cssStyle	no	HTML style attribute
theme	no	Theme to use
template	no	Name of template to use

Note that the default template name for this tag is `text.vm` and not `textfield.vm`. ([back to the top](#))

<ww:token />

The token tag is used to help with the "double click" submission problem. It is needed if you are using the `TokenInterceptor` or the `TokenSessionInterceptor`. They are documented somewhere in the wiki. The `ww:token` tag merely places a hidden element that contains the unique token.

Sample Usages

```
<ww:token />
```

Attributes

Name	Required	Description
name	no	Name of token
theme	no	Theme to use
template	no	Name of template to use

Internationalization Tags

([back to the top](#))

<ww:i18n />

Place a resource bundle on the value stack, for access by the text tag.

Attributes

Name	Required	Description
name	yes	Name of bundle

Sample Usages

-see text tag below

([back to the top](#))

<ww:text />

Print out an internationalized string. It is used in conjunction with the i18n tag. The text tag gets a specific message from the bundle specified in the surrounding i18n tag. Values can be passed into the message for parsing, for instance to format a date or currency.

Attributes

Name	Required	Description
name	yes	Name of property to fetch
value0	no	Pass data to param 0 in message
value1	no	Pass data to param 1 in message

value2	no	Pass data to param 2 in message
value3	no	Pass data to param 3 in message

Sample Usages

```
Accessing messages from a given bundle (the i18n Shop example bundle in
thiscase)<br>
<ww:i18n name="'webwork.action.test.i18n.Shop'">
  <ww:text name="'main.title'"/>
</ww:i18n>
```

Note that instead of using value0..value4, you may also:

```
<ww:text name="'someKey'">
  <ww:param value="'Hello'"/>
</ww:text>
```


OR


```
<ww:text name="'someKey'">
  <ww:param>Hello</ww:param>
</ww:text>
```

As per Patrick Lightbody:

The last format is particularly good when you are embedding HTML in to the message, since you don't need to worry about escaping the various quotes that might be there.

Notes

 The "required" attribute on many WebWork UI tags has nothing to do with client-side validation. It serves as a flag that allows the template to display some kind of indication to the user that a particular field is required for form submission.

 It's very important to note that all tags that insert something into the valuestack (like i18n or bean tags) will remove those objects from the stack on its end tag. This means that if you instantiate a bean with the bean tag (<ww:bean name="br.univap.fcc.sgpw.util.FormatterHelper"/>) that bean will be available on the valuestack only until the </ww:bean> tag.

FAQ

This page last changed on Aug 14, 2004 by [sutter2k](#).

How do I get the latest version of Webwork and XWork from CVS?

```
cvs -d :pserver:guest@cvs.dev.java.net:/cvs login
```

(Use an empty password, just hit enter..)

```
cvs -d :pserver:guest@cvs.dev.java.net:/cvs checkout webwork
```

```
cvs -d :pserver:guest@cvs.dev.java.net:/cvs checkout xwork
```

Note: WebWork from the CVS does not compile with the latest 1.5 J2sdk. Use the stable J2sdk 1.4.2.

How do I build the latest versions XWork and Webwork?

Just go into the XWork or WebWork directories and run 'ant' (you must have ant installed and have the jars of junit and clover inside \$ANT_HOME/lib)

Once you have built the xwork.jar copy it into the webwork/lib/core folder, and delete the old one.

How do I use messages from within the validator?

```
<validators><field name="name"><field-validator type="requiredstring"><message
key="template.name.errors.required">A default message in case the key is not
found</message></field-validator></field></validators>
```

How do I set a global resource bundle?

In webwork.properties(as of Webwork 2.1.1),
you can now use:

```
webwork.custom.i18n.resources=global-messages
```

Several resource bundles can be specified by comma separating them.
for example see webwork.properties :

<http://wiki.opensymphony.com/display/WW/webwork.properties>

Java class (thanks Drew McAuliffe):

```
public class WebworkGlobalMessagesListener implements ServletContextListener {
    private static Logger log =
    Logger.getLogger(WebworkGlobalMessagesListener.class);
    private static final String DEFAULT_RESOURCE = "global-messages";

    /**
     * Uses the LocalizedTextUtil to load messages from the global
     * message bundle.
     * @see
```

```

        javax.servlet.ServletContextListener#contextInitialized( javax.servlet.Servle
        tContextEvent)
        */
    public void contextInitialized(ServletContextEvent arg0) {
        log.info("Loading global messages from " + DEFAULT_RESOURCE);
        LocalizedTextUtil.addDefaultResourceBundle(DEFAULT_RESOURCE);
        log.info("Global messages loaded.");
    }

    /**
     * @see
     javax.servlet.ServletContextListener#contextDestroyed( javax.servlet.ServletContextEvent)
     */
    public void contextDestroyed(ServletContextEvent arg0) {

        // do nothing
    }
}

```

web.xml:

(under listeners section)

```

<listener><listener-class>mypackagename.WebworkGlobalMessagesListener</listener-class></listener>

```

How do I change the error message for invalid inputted fields?

You need to create a message for that field, for example if you have a user.dob field you would use this in your messages file (see above for example on setting a global messages file):

invalid.fieldvalue.user.dob=Please enter Date of Birth in the correct format.

How do I get access to the Session?

ActionContext.getContext().getSession() (returns Map, works internally using a ThreadLocal)

How can I see all parameters passed into the Action?

ActionContext.getParameters() (returns Map, works internally using a ThreadLocal)

How can I get the HttpServletRequest?

ServletActionContext.getRequest() (works internally using a ThreadLocal)

How can I use the IOC container to initialize a component in another object that isnt an action?

Obtain the ComponentManager from the request: ComponentManager cm = (ComponentManager)

ServletActionContext.getRequest().getAttribute("DefaultComponentManager");

then you need to initialize it using: cm.initializeObject(Object)

How do I decouple XWork LocalizedTextUtil global resource bundle loading

from serlvets (ServletContextListener)?

If you're using XWork outside a Web context, then use whatever startup hooks you have in that context (i.e. application start for a desktop app) to add the global resource bundle. This is a startup activity, so use whatever mechanisms are provided in the context you're running in.

What i need to do to put values in a combobox. If I am using webwork2?

If i have :

```
#tag(Select "label='xxx '" "name='xxx'" "list=?")
or
#tag(combobox "label='Prioridade'" "name='inavis.avisTpPrioridade'" "list=?")
```

the values in this combobox, what i need to do?

Example:

```
html tag i use to do:

<select..>
  <otpion value="" selected>XXX</option>
</selct>
```

so...i need to do this using Webwork tags from Velocity...how can i do this??

How do I add I18N to a UI tag, like ww:textfield?

```
<ww:textfield label="'i18n.label'" name="'label1'" value="''">
```

This will get the localized text message for the key "i18n.label" and put it in the label.

```
<ww:textfield label="getText('i18n.label')" name="'label1'" value="''">
```

Alternatively, you could modify controlheader.vm and copy it to /template/xhtmll. There you could make it so that it automatically does a call to `$stack.findValue("getText($parameters.label)")`, making the first example actually work for i18n.

Can I add I18N outside the Action's context? i.e. adding i18n to some JSP using the ww taglib?

Yes, use the `<ww:i18n>` tag to push a resource bundle on to the stack. Now calls with `<ww:text/>` or `<ww:property value="getText(...)"/>` will read from that resource bundle.

Can I break up my large XWork.xml file into smaller pieces?

Sure, that's what the `<include>` element is for. Most xwork.xml files already have one:

```
<xwork>
  <include file="webwork-default.xml"/>
  <include file="config-browser.xml"/>
  <package name="default" extends="webwork-default">
    ....
  </package>
  <include file="other.xml"/>
</xwork>
```

This tells it to load the webwork-default.xml from the webwork jar file to get all of those interceptor and result definitions.

You can put your own <include> in your xwork.xml interchangeably with <package> elements... They will be loaded in the same order as it reads from top to bottom and adds things as it reads them.

How can I put a String literal in a Javascript call, for instance in an onChange attribute?

The problem is in escaping quotes and getting the double quotes around the final value, like we expect in HTML attributes. Here's an example of the right way to do this (thanks to John Brad):

```
onChange=' "someFunc(this.form, \'abc\') "'
```

Notice here that there are single quotes surrounding the double quotes, and then the single quotes inline in the Javascript are escaped. This produces this result:

```
onChange="someFunc(this.form, 'abc')"
```


Lesson 5: Interceptors

Interceptors allow arbitrary code to be included in the call stack for your action before and/or after processing the action, which can vastly simplify your code itself and provide excellent opportunities for code reuse. Many of the features of XWork and WebWork are implemented as interceptors and can be applied via external configuration along with your own Interceptors in whatever order you specify for any set of actions you define.

In other words, when you access a *.action URL, WebWork's `ServletDispatcher` proceeds to the invocation of the an action object. Before it is executed, however, the invocation can be intercepted by another object, that is hence called interceptor. To have an interceptor executed before (or after) a given action, just configure `xwork.xml` properly, like the example below, taken from [lesson 4.1.1](#):

Interceptor configuration from lesson 4.1.1:

```
<action name="formProcessing" class="lesson04_01_01.FormProcessingAction"><result
name="input" type="dispatcher">ex01-index.jsp</result><result name="success"
type="dispatcher">ex01-success.jsp</result><interceptor-ref
name="validationWorkflowStack" /></action>
```

As you can see, lesson 4.1.1's `formProcessing` Action uses the `validationWorkflowStack`. That is an interceptor stack, which organizes a bunch of interceptors in the order in which they are to be executed. That stack is configured in `webwork-default.xml`, so all we have to do to use it is declare a `<interceptor-ref />` under the action configuration or a `<default-interceptor-ref />`, under package configuration, as seen in [lesson 3](#)'s first example:

Interceptor configuration from lesson 3.1:

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork
1.0//EN" "http://www.opensymphony.com/xwork/xwork-1.0.dtd"><xwork><!-- Include
webwork defaults (from WebWork-2.1 JAR). --><include file="webwork-default.xml"
/><!-- Configuration for the default package. --><package name="default"
```

```
"webwork-default"><!-- Default interceptor stack. --><default-interceptor-ref
name="defaultStack" /><!-- Action: Lesson 03: HelloWebWorldAction. --><action
name="helloWebWorld" class="lesson03.HelloWebWorldAction"><result name="success"
type="dispatcher">ex01-success.jsp</result></action></package></xwork>
```

But let's see how it works from scratch:

1. Create an interceptor class, which is a class that implements the `com.opensymphony.xwork.interceptor.Interceptor` interface (bundled in `xwork-1.0.jar`);
2. Declare the class in your XML configuration file (`xwork.xml`) using the element `<interceptor />` nested within `<interceptors />`;
3. Create stacks of interceptors, using the `<interceptor-stack />` element (*optional*);
4. Determine which interceptors are used by which action, using `<interceptor-ref />` or `<default-interceptor-ref />`. The former defines the interceptors to be used in a specific action, while the latter determines the default interceptor stack to be used by all actions that do not specify their own `<interceptor-ref />`.

Looking inside `webwork-default.xml` we can see how it's done:

webwork-default.xml:

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork
1.0//EN" "http://www.opensymphony.com/xwork/xwork-1.0.dtd"><xwork><package
name="webwork-default"><result-types><result-type name="dispatcher" default="true"
class="com.opensymphony.webwork.dispatcher.ServletDispatcherResult"/><result-type
name="redirect"
class="com.opensymphony.webwork.dispatcher.ServletRedirectResult"/><result-type
name="velocity"
class="com.opensymphony.webwork.dispatcher.VelocityResult"/><result-type
name="chain"
class="com.opensymphony.xwork.ActionChainResult"/><result-type
name="xslt"
class="com.opensymphony.webwork.views.xslt.XSLTResult"/></result-types><interceptor
name="timer"
class="com.opensymphony.xwork.interceptor.TimerInterceptor"/><interceptor
name="logger"
class="com.opensymphony.xwork.interceptor.LoggingInterceptor"/><interceptor
name="chain"
class="com.opensymphony.xwork.interceptor.ChainingInterceptor"/><interceptor
name="static-params"
class="com.opensymphony.xwork.interceptor.StaticParametersInterceptor"/><interceptor
name="params"
class="com.opensymphony.xwork.interceptor.ParametersInterceptor"/><interceptor
name="model-driven"
class="com.opensymphony.xwork.interceptor.ModelDrivenInterceptor"/><interceptor
name="component"
class="com.opensymphony.xwork.interceptor.component.ComponentInterceptor"/><interceptor
name="token"
class="com.opensymphony.webwork.interceptor.TokenInterceptor"/><interceptor
```

```

name="token-session"
    class="com.opensymphony.webwork.interceptor.TokenSessionStoreInterceptor"/><interceptor
name="validation"
    class="com.opensymphony.xwork.validator.ValidationInterceptor"/><interceptor
name="workflow"
    class="com.opensymphony.xwork.interceptor.DefaultWorkflowInterceptor"/><interceptor
name="servlet-config"
    class="com.opensymphony.webwork.interceptor.ServletConfigInterceptor"/><interceptor
name="prepare"
    class="com.opensymphony.xwork.interceptor.PrepareInterceptor"/><interceptor
name="conversionError"
    class="com.opensymphony.webwork.interceptor.WebWorkConversionErrorInterceptor"/><interceptor
name="defaultStack"><interceptor-ref name="static-params"/><interceptor-ref
name="params"/><interceptor-ref
name="conversionError"/></interceptor-stack><interceptor-stack
name="validationWorkflowStack"><interceptor-ref
name="defaultStack"/><interceptor-ref name="validation"/><interceptor-ref
name="workflow"/></interceptor-stack></interceptors></package></xwork>

```

Since we included `webwork-default.xml` in our `xwork.xml`, all the interceptors and stacks above are available for us to use in our actions. Here's what these interceptors do:

- **timer**: clocks how long the action (including nested interceptors and view) takes to execute;
- **logger**: logs the action being executed;
- **chain**: makes the previous action's properties available to the current action. Used to make action chaining (reference: [Result Types](#));
- **static-params**: sets the parameters defined in `xwork.xml` onto the action. These are the `<param />` tags that are direct children of the `<action />` tag;
- **params**: sets the request (POST and GET) parameters onto the action class. We have seen an example of this in [lesson 3](#);
- **model-driven**: if the action implements `ModelDriven`, pushes the `getModel()` result onto the Value Stack;
- **component**: enables and makes registered components available to the actions. (reference: [\[IoC & Components\]](#));
- **token**: checks for valid token presence in action, prevents duplicate form submission;
- **token-session**: same as above, but storing the submitted data in session when handed an invalid token;
- **validation**: performs validation using the validators defined in `{Action}-validation.xml` (reference: [Validation](#)). We've seen an example of this in [lesson 4.1.1](#);
- **workflow**: calls the `validate` method in your action class. If action errors created then it returns the `INPUT` view. Good to use together with the validation interceptor (reference: [Validation](#));
- **servlet-config**: give access to `HttpServletRequest` and `HttpServletResponse` (think twice before using this since this ties you to the Servlet API);
- **prepare**: allows you to programmatic access to your Action class before the parameters are set on it.;

- **conversionError:** *help needed here.*

Building your own Interceptor

If none of the above interceptors suit your particular need, you will have to implement your own interceptor. Fortunately, this is an easy task to accomplish. Suppose we need an interceptor that places a greeting in the Session according to the time of the day (morning, afternoon or evening). Here's how we could implement it:

GreetingInterceptor.java:

```
package lesson05;

import java.util.Calendar;
import com.opensymphony.xwork.interceptor.Interceptor;
import com.opensymphony.xwork.ActionInvocation;

public class GreetingInterceptor implements Interceptor {
    public void init() { }
    public void destroy() { }
    public String intercept(ActionInvocation invocation) throws Exception {
        Calendar calendar = Calendar.getInstance();
        int hour = calendar.get(Calendar.HOUR_OF_DAY);
        String greeting = (hour < 6) ? "Good evening" :
            ((hour < 12) ? "Good morning":
            ((hour < 18) ? "Good afternoon": "Good evening"));

        invocation.getInvocationContext().getSession().put("greeting", greeting);

        String result = invocation.invoke();

        return result;
    }
}
```

xwork.xml:

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork
1.0//EN" "http://www.opensymphony.com/xwork/xwork-1.0.dtd"><xwork><!-- Include
webwork defaults (from WebWork-2.1 JAR). --><include file="webwork-default.xml"
/><!-- Configuration for the default package. --><package name="default"
extends="webwork-default"><interceptors><interceptor name="greeting"
class="section02.lesson05.GreetingInterceptor" /></interceptors><!-- Action: Lesson
5: GreetingInterceptor. --><action name="greetingAction"
class="lesson05.GreetingAction"><result name="success"
type="velocity">ex01-result.vm</result><interceptor-ref name="greeting"
/></action></package></xwork>
```

GreetingAction.java:

```
package lesson05;

import com.opensymphony.xwork.ActionSupport;

public class GreetingAction extends ActionSupport {
    publicString execute() throws Exception {
        return SUCCESS;
    }
}
```

ex01-result.vm:

```
<html>
<head>
<title>WebWork Tutorial - Lesson 5 - Example 1</title>
</head>
<body>

#set ($ses = $req.getSession())
<p><b>${ses.getAttribute('greeting')}!</b></p>

</body>
</html>
```

Let's take a look at our interceptor class first. As explained before, the interceptor must implement `com.opensymphony.xwork.interceptor.Interceptor`'s methods: `init()`, called during interceptor initialization, `destroy()`, called during destruction, and most importantly, `intercept(ActionInvocation invocation)`, which is where we place the code that does the work.

Notice that our interceptor returns the result from `invocation.invoke()` which is the method responsible for executing the next interceptor in the stack or, if this is the last one, the action. This means that the interceptor has the power of short-circuiting the action invocation and return a result string without executing the action at all! Use this with caution, though.

One other thing that interceptors can do is execute code after the action has executed. To do that, just place code after the `invocation.invoke()` call. WebWork provides an abstract class that already implements this kind of behaviour: `com.opensymphony.xwork.interceptor.AroundInterceptor`. Just extend it and implement the methods `before(ActionInvocation invocation)` and `after(ActionInvocation dispatcher, String result)`.

The `xwork.xml` configuration, the action class and the result page are pretty straightforward and require no further explanation.

| [Try the example!](#)

[Previous Lesson](#) | [End of Tutorial](#)

WebWork

This page last changed on Aug 26, 2004 by [plightbo](#).

Welcome to the WebWork wiki. WebWork's official homepage is <http://www.opensymphony.com/webwork/>. There you can find documentation for the latest released version of WebWork. This wiki is used for additional information as well as documentation for the latest developing version, currently 2.1.1.

- [Documentation](#)
 - [API JavaDocs](#)
- [Press Releases](#)
- [Download Binaries](#)
- [CVS](#)
- [QuickStart](#)
- Cookbook - Patterns and tips
- [Examples](#)
- [Testimonials](#)
- [WebWork Team](#)
- [IDEA Plugin](#)

Upgrading from 2.1

This page last changed on Aug 27, 2004 by [plightbo](#).

Upgrading from 2.1 to 2.1.1 is very easy. Simply copy over the new webwork.jar file and make sure you are running all the correct [Dependencies](#).

1. Overview

1. [What is WebWork](#)
2. [Getting Started](#)
3. [FAQ](#)
4. [Deployment Notes](#)
5. [WebWork Community](#)
 - a. [Mailing lists](#)
 - b. [Bug tracker, wiki](#)
6. [Articles and press about WebWork](#)
7. [Projects Using WebWork](#)
8. [Comparison to Struts](#)

2. WebWork versions

- Current Release
 - Release Notes
 - [Release Notes - 2.1.1](#)
 - [Release Notes - 2.1](#)
 - [Dependencies](#)
- Upgrading from previous versions
 - [Upgrading from 2.1](#)
 - [Upgrading from 2.0](#)
 - [Upgrading from 1.4](#)

3. Tutorial

1. [Lesson 1: Downloading and Installing WebWork](#)
2. [Lesson 2: Setting up the Web Application](#)
3. [Lesson 3: Actions and Results](#)
4. [Lesson 4: Views](#) (JSP, Velocity, Freemarker)
5. [Lesson 5: Interceptors](#)

4. Reference Guide

1. [Configuration](#)
2. [Inversion of Control](#)
3. [JSP Tags](#)
4. [Result Types](#)
5. [Type Conversion](#)
6. [Validation](#)
7. [OGNL](#)
8. [Internationalization](#)

5. Third-party integration

1. [SiteMesh](#)
2. [Spring](#)
3. [Pico](#)
4. [Hibernate](#)
5. [JUnit](#)
6. [Quartz](#)

Dependencies

This page last changed on Aug 26, 2004 by [plightbo](#).

Dependencies are split into runtime and compile time dependencies.

- Runtime dependencies are stored in CVS in the lib/core directory.
- Compile time dependencies are stored in CVS in the lib/build directory.

You can see all the dependencies as well as their version numbers by looking at the libraries.txt file located in each directory.

Key Changes



- JavaScript client validation support - not totally complete, but basic validators work well. Look at the validators.xml file include in src/example to see how you can configure your validators to do client side validation on top of their normal duties
- The label attribute in UI tags are no longer required
- The themes and templates in UI tags behave like they did in 1.x
- A new theme, in addition to the existing "xhtml" one, called "simple" is included that doesn't have any of the labels, error reporting, or table rows that the "xhtml" template has. This is more in line with the tags included with Struts.
- New UI tags for CSS styles and classes added: cssStyle and cssClass
- Old action!command URL support works again. This means you can invoke a doCommand() method like in 1.x
- ww:param tag no longer requires the name attribute (for ordered params, like with ww:text). It also evaluates the the body as the value if no value is given.
- UI tags now have access to the FormTag parameter map using the "form" key. This means \$parameters.form.name would return the form name, for example. The result is that complex JavaScript-based components can be built.



















Migration Notes

Version	Description	Old Code	New Code
2.0	WebWorkUtil has been refactored into a number of classes, and the constructor has changed. If you were using it for Velocity support before, look at VelocityWebWorkUtil now		
2.0	The <i>webwork.ui.templateDir</i> configuration	<code>webwork.ui.templateDir = /webwork/mytheme</code>	<code>webwork.ui.templateDir = /webwork/webwork.ui.theme</code>

	property has been broken into <i>webwork.ui.templateDir</i> and <i>webwork.ui.theme</i>		<div>= mytheme</div>
2.0	"namespace" attribute of the <code>ww:action</code> tag is now evaluated; those upgrading from 2.0 will need to place single quotes around the attribute value	<code><ww:action namespace="/foo" .../></code>	<code><ww:action namespace=""/foo" .../></code>
2.0, but not 1.x	theme and template attributes in UI tags have changed are now evaluated; those upgrading from 2.0 will need to place single quotes around the attribute value	<code><ww:xxxx theme="/template/foo" template="bar.vm"/></code>	<code><ww:xxxx theme=""foo" template=""bar.vm"/></code>
1.x, 2.0	label UI tag evaluates the value attribute now instead of the name attribute	<code><ww:label name=""Foo"/></code>	<code><ww:label value=""Foo"/></code>

Changelog

OpenSymphony JIRA (25 issues)		
T	Key	Summary
	WW-592	Upgrade commons-logging
	WW-560	SessionMap holds on to requests when it doesn't need to

	WW-546	Make the config-browser show validators applied via the XML validation files
	WW-544	Velocity result hardcodes contenttype and encoding
	WW-541	Webpage link for download
	WW-537	Velocity tag outputs to the response, not the velocity writer
	WW-530	Config Browser doesn't work after lates ActionConfig refactoring
	WW-519	ActionTag should evaluate namespace attribute
	WW-518	Label attribute shouldn't be required
	WW-517	Themes and templates should behave like 1.x
	WW-516	Simple theme that has no tables and xhtml extends from
	WW-515	Class attribute is illegal
	WW-514	Form tag double evaluates name attribute
	WW-503	Fix tag libraries
	WW-502	foo!default.action should work
	WW-501	JavaScript-based client side validation
	WW-500	ww:param tag fixes
	WW-499	UI tags should have access to form
	WW-488	Check QuickStart Guide to make sure it works
	WW-487	WebWorkConversionErrorInterceptorTe

		in wrong branch
	WW-484	label tag problems
	WW-478	URLTag tld entry does not correspond with actual property
	WW-476	WebWork needs a simple changelog for each release
	WW-475	Multipart encoding still not fixed
	WW-474	Ability to dynamically create array of Objects from a given request

WebWork 2.1.1 Release Notes

Key Changes

- Improved integration with Sitemesh
 - WebWork taglibs can be used in Sitemesh decorators to access Action properties
- Validator short-circuiting to allow validation to stop on first invalid data
- Improved class hierarchy resource bundle searching
- File upload support has been rebuilt to allow for multiple files with the same HTTP parameter name. Besides "cos" and "pell" support, "jakarta" support has been added, utilizing the Commons-FileUpload library. Only "jakarta" supports multiple files with the same HTTP parameter name. In future versions "jakarta" may become the default upload library, replacing "pell".









Migration Notes



Version	Description	Old Code	New Code
2.1	There is a new validator DTD: xwork-validator-1.0.2.dtd. You aren't required to use this, but you will need to if you wish to use the new short-circuiting validation	N/A	N/A
2.1	File upload support has been rebuilt, although we don't see any compatibility problems with 2.1. However, many of	N/A	N/A




	the methods in MultiPartRequest have become deprecated in favor of new ones. Please switch to these as soon as possible.		
--	--------------------------------------------------------------------------------------------------------------------------	--	--

Changelog








WebWork 2.1.1

OpenSymphony JIRA (25 issues)		
T	Key	Summary
	WW-596	IllegalArgumentException when setting an indexed property
	WW-591	Wrong output when using nesting #bodytags (with a custom tag)
	WW-589	Printer-friendly of all documentation
	WW-588	Unexpected behaviour of SessionMap.clear()
	WW-585	ComboBoxTag should subclass TextFieldTag
	WW-581	JSP Tags should support better syntax
	WW-574	ActionButton and ActionLink UI tags
	WW-572	Lesson 2 incorrectly references the xwork-validator-1.0.dtd for validators.xml

	WW-570	Table tag parameters and sorting
	WW-558	Exception Handler Interceptor and Exception Action
	WW-557	Generator Tag doesn't work properly
	WW-556	Allow ComponentManager scopes to be independant
	WW-555	Disable logging by default
	WW-554	Bad value for IMAGES_URI in JasperReportsResult.java
	WW-540	Add a ServletFilter to clean up the ActionContext, allowing Sitemesh decorators to use the ActionContext with the taglib
	WW-536	Lost the session on OC4J 9.0.4 using WebWork2
	WW-533	Add support for Velocity 1.4 - Changes WebWork directives
	WW-528	Switch to using VelocityViewServlet from VelocityTools
	WW-527	Multiple components with same <class> but different <enabler> throws exception
	WW-526	Velocity using include ignores character encoding
	WW-520	Wrong & management in XML based output (WML and XHTML)
	WW-513	IOC Application and Session scopes do not work in

		Tomcat 5
	WW-511	Support a list of resource bundles on i18n tag
	WW-507	"command" parameter isn't honored for command driven actions
	WW-506	Param tag should try to get parameters from the original request as well

Xwork 1.0.2

OpenSymphony JIRA (15 issues)		
T	Key	Summary
	XW-210	Make default type conversion message a localized text that can be overridden
	XW-205	missing xwork 1.0.2 dtd in jar and website and typo in ValidationInterceptor
	XW-204	TextProvider.getText() should look in child property files
	XW-203	Add "trim" parameter to string validators
	XW-202	Integer and Float conversion dont work in CVS HEAD
	XW-200	i18n broken when the name of the text to find starts with a property exposed by the action
	XW-195	Add interface XWorkStatics which contains

		XWork-related constants from WebWorkStatics
	XW-194	Patch to help LocalizedTextUtil deal with messages for indexed fields (collections)
	XW-193	InstantiatingNullHandler and Typeconversion fails
	XW-192	Create a version 1.0.2 of the XWork validation DTD with short circuit
	XW-191	Type conversion improvement.
	XW-190	Provide a xwork-default.xml.
	XW-189	Improve ActionValidationManager's short circuit behaviour
	XW-179	Optimise OgnlUtil.copy method
	XW-172	XWorkBasicConverter doesn't care about the current locale