Convention Plugin

This page is DEPRECATED, please refer to the new source http://struts.apache.org/plugins/convention/

Introduction

The Convention Plugin is bundled with Struts since 2.1 and replaces the Codebehind Plugin and Zero Config plugins. It provides the following features:

- Action location by package naming conventions
- Result (JSP, FreeMarker, etc) location by naming conventions
- Class name to URL naming convention
- Package name to namespace convention
- SEO compliant URLs (i.e. my-action rather than MyAction)
- Action name overrides using annotations
- Interceptor overrides using annotations
- Namespace overrides using annotations
- XWork package overrides using annotations
- Default action and result handling (i.e. /products will try com.example.actions.Products as well as com.example.actions.products.Index)

The Convention Plugin should require no configuration to use. Many of the conventions can be controlled using configuration properties and many of the classes can be extended or overridden.

Setup

In order to use the Convention plugin, you first need to add the JAR file to the `WEB-INF/lib` directory of your application or include the dependency in your project's Maven POM file.

```
<dependency>
    <groupId>org.apache.struts</groupId>
    <artifactId>struts2-convention-plugin</artifactId>
    <version>X.X.X</version>
</dependency>
```

Where X.X.X is the current version of Struts 2. Please remember that the Convention Plugin is available from version 2.1.6.

Converting a Codebehind based application to Convention

See this page for the required changes and tips.

If you are using REST with the Convention plugin, make sure you set these constants in struts.xml:

```
<constant name="struts.convention.action.suffix" value="Controller"/>
<constant name="struts.convention.action.mapAllMatches" value="true"/>
<constant name="struts.convention.default.parent.package" value="rest-default"/>
```

Hello world

Now that the Convention plugin has been added to your application, let's start with a very simple example. This example will use an actionless result that is identified by the URL. By default, the Convention plugin assumes that all of the results are stored in **WEB-INF/content**. This can be changed by setting the property `struts.convention.result.path` in the Struts properties file to the new location. Don't worry about trailing slashes, the Convention plugin handles this for you. Here is our hello world JSP:

```
<html>
<body>
Hello world!
</body>
</html>
```

If you start Tomcat (or whichever J2EE container you are using) and type in http://localhost:8080/hello-world (assuming that your context path is "/", ie. starting application from Eclipse) into your browser you should get this result:

WEB-INF/content/hello-world.jsp

```
Hello world!
```

This illustrates that the Convention plugin will find results even when no action exists and it is all based on the URL passed to Struts.

Code behind hello world

Let's expand on this example and add a code behind class. In order to do this we need to ensure that the Convention plugin is able to find our action classes. By default, the Convention plugin will find all action classes that implement `com.opensymphony.xwork2.Action` or whose name ends with the word **Action** in specific packages.

These packages are located by the Convention plugin using a search methodology. First the Convention plugin finds packages named `struts`, `struts2`, `action` or `actions`. Any packages that match those names are considered the root packages for the Convention plugin. Next, the plugin looks at all of the classes in those packages as well as sub-packages and determines if the classes implement `com.opensymphony.xwork2.Action` or if their name ends with **Action** (i.e. FooAction). Here's an example of a few classes that the Convention plugin will find:

Classes

```
com.example.actions.MainAction
com.example.actions.products.Display (implements com.opensymphony.xwork2.Action)
com.example.struts.company.details.ShowCompanyDetailsAction
```

Each of the action classes that the plugin finds will be configured to respond to specific URLs. The URL is based on the package name that the class is defined in and the class name itself. First the plugin determines the namespace of the URL using the package names between the root package and the package the class is defined in. For our examples above, the namespaces would be:

Namespaces

```
com.example.actions.MainAction -> /
com.example.actions.products.Display -> /products
com.example.struts.company.details.ShowCompanyDetailsAction -> /company/details
```

Next, the plugin determines the URL of the resource using the class name. It first removes the word **Action** from the end of the class name and then converts camel case names to dashes. In our example the full URLs would be:

Full URLs

```
com.example.actions.MainAction -> /main
com.example.actions.products.Display -> /products/display
com.example.struts.company.details.ShowCompanyDetailsAction -> /company/details/show-company-details
```

You can tell the Convention plugin to ignore certain packages using the property `struts.convention.exclude.packages`. You can also tell the plugin to use different strings to locate root packages using the property `struts.convention.package.locators`. Finally, you can tell the plugin to search specific root packages using the property `struts.convention.action.packages`.

Here is our code behind action class:

com.example.actions.HelloWorld

```
package com.example.actions;

import com.opensymphony.xwork2.ActionSupport;

public class HelloWorld extends ActionSupport {
  private String message;

  public String getMessage() {
    return message;
  }

  public String execute() {
    message = "Hello World!";
    return SUCCESS;
  }
}
```

If you compile this class and place it into your application in the WEB-INF/classes, the Convention plugin will find the class and map the URL **/hello-world** to it. Next, we need to update our JSP to print out the message we setup in the action class. Here is the new JSP:

WEB-INF/content/hello-world.jsp

```
<html>
<body>
The message is ${message}
</body>
</html>
```

⚠ Please notice that the expression `${message}` will work without adding JSP directive `isELIgnored="false"`.

If start up the application server and open up http://localhost:8080/hello-world in our browser, we should get this result:

Result

```
The message is Hello World!
```

Results and result codes

The Convention Plugin will pre-configure all of you action classes when Struts is started. By default, this configuration will also contain results for any JSPs that it can find within the application. The JSPs have an additional feature that allows different JSPs to be used based on the result code of the action. Since action methods return Strings and these Strings are traditionally used to locate results for the action, the Convention plugin allows you to define different results based on the result code.

Building on our example from above, let's say we want to provide a different result if the result code from our action is the String `zero` rather than `success`. First, we update the action class to return different result codes:

com.example.actions.HelloWorld

```
package com.example.actions;

import com.opensymphony.xwork2.ActionSupport;

public class HelloWorld extends ActionSupport {
  private String message;

  public String getMessage() {
    return message;
  }

  public String execute() {
    if (System.currentTimeMillis() % 2 == 0) {
      message = "It's 0";
      return "zero";
    }

    message = "It's 1";
    return SUCCESS;
  }
}
```

Next, we add a new JSP to the application named `WEB-INF/content/hello-world-zero.jsp`. Notice that the first part of the file name is the same as the URL of the action and the last part of the name is the result code. This is the convention that the plugin uses to determine which results to render. Here is our new JSP:

WEB-INF/content/hello-world.jsp

```
<html>
<body>
The error message is ${message}
</body>
</html>
```

Now, if you compile the action and restart the application, based on the current time, you'll either see the result from `WEB-INF/content/hello-world.jsp` or `WEB-INF/content/hello-world-zero.jsp`.

The result type is based on the extension of the file. The supported extensions are: jsp,ftl,vm,html,html. Examples of Action and Result to Template mapping:

URL	Result	File that could match	Result Type
/hello	success	/WEB-INF/content/hello.jsp	Dispatcher
/hello	success	/WEB-INF/content/hello-success.htm	Dispatcher
/hello	success	/WEB-INF/content/hello.ftl	FreeMarker
/hello-world	input	/WEB-INF/content/hello-world-input.vm	Velocity
/test1/test2/hello	error	/WEB-INF/content/test/test2/hello-error.html	Dispatcher

Multiple names

It is possible to define multiple names for the same result:

```
@Action(results = {
    @Result(name={"error", "input"}, location="input-form.jsp"),
    @Result(name="success", location="success.jsp")
})
```

Such functionality was added in Struts 2.5

Chaining

If one action returns the name of another action in the same package, they will be chained together, if the first action doesn't have any result defined for that code. In the following example:

com.example.actions.HelloWorld

```
package com.example.actions;

import com.opensymphony.xwork2.Action;
import com.opensymphony.xwork2.ActionSupport;

public class HelloAction extends ActionSupport {
    @Action("foo")
    public String foo() {
        return "bar";
    }

    @Action("foo-bar")
    public String bar() {
        return SUCCESS;
    }
}
```

The "foo" action will be executed, because no result is found, the Convention plugin tries to find an action named "foo-bar" on the same package where "foo" is defined. If such an action is found, it will be invoked using the "chain" result.

XWork packages

Actions are placed on a custom XWork package which prevents conflicts. The name of this package is based on the Java package the action is defined in, the namespace part of the URL for the action and the parent XWork package for the action. The parent XWork package is determined based on the property named `struts.convention.default.parent.package`(defaults to `convention-default`), which is a custom XWork package that extends `struts-default`.

Therefore the naming for XWork packages used by the Convention plugin are in the form:

XWork package naming

```
<java-package>#<namespace>#<parent-package>
```

Using our example from above, the XWork package for our action would be:

XWork package naming

```
com.example.actions#/#conventionDefault
```

Annotation reference

The Convention plugin uses a number of different annotations to override the default conventions that are used to map actions to URLs and locate results. In addition, you can modify the parent XWork package that actions are configured with.

Action annotation

The Convention plugin allows action classes to change the URL that they are mapped to using the **Action** annotation. This annotation can also be used inside the **Actions** annotation to allow multiple URLs to map to a single action class. This annotation must be defined on action methods like this:

com.example.actions.HelloWorld

```
package com.example.actions;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.convention.annotation.Action;

public class HelloWorld extends ActionSupport {
  @Action("/different/url")
  public String execute() {
    return SUCCESS;
  }
}
```

Our action class will now map to the URL `/different/url` rather than `/hello-world`. If no `@Result` (see next section) is specified, then the namespace of the action will be used as the path to the result, on our last example it would be `/WEB-INF/content/different/url.jsp`.

A single method within an action class can also map to multiple URLs using the **Actions** annotation like this:

com.example.actions.HelloWorld

```
package com.example.actions;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.Actions;

public class HelloWorld extends ActionSupport {
  @Actions({
    @Action("/different/url"),
    @Action("/another/url")
  })
  public String execute() {
    return SUCCESS;
  }
}
```

Another usage of the **Action** or **Actions** annotation is to define multiple action methods within a single action class, each of which respond to a different URL. Here is an example of multiple action methods:

com.example.actions.HelloWorld

```
package com.example.actions;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.Actions;

public class HelloWorld extends ActionSupport {
  @Action("/different/url")
  public String execute() {
    return SUCCESS;


  }

  @Action("url")
  public String doSomething() {
    return SUCCESS;
  }
}
```

The previous example defines a second URL that is not fully qualified. This means that the namespace for the URL is determined using the Java package name rather than the Action annotation.

Interceptor and interceptor stacks can be specified using the `interceptorRefs` attribute. The following example applies the `validation` interceptor and the `defaultStack` interceptor stack to the action:

com.example.actions.HelloWorld

```
package com.example.actions;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.Actions;

public class HelloWorld extends ActionSupport {
  @Action(interceptorRefs={@InterceptorRef("validation"), @InterceptorRef("defaultStack")})
  public String execute() {
    return SUCCESS;
  }

  @Action("url")
  public String doSomething() {
    return SUCCESS;
  }
}
```

Parameters can be passed to results using the **params** attribute. The value of this attribute is a string array with an even number of elements in the form {"key0", "value0, "key1", "value1" ... "keyN", "valueN"}. For example:

com.example.actions.HelloWorld

```
package com.example.actions;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.Actions;

public class HelloWorld extends ActionSupport {
  @Action(interceptorRefs=@InterceptorRef(value="validation",params={"programmatic", "false", "declarative",
"true"}))
  public String execute() {
    return SUCCESS;
  }

  @Action("url")
  public String doSomething() {
    return SUCCESS;
  }
}
```

If interceptors are not specified, the default stack is applied.

You can specify className parameter which can be especially useful when Spring Framework is used to instantiate actions.

Applying @Action and @Actions at the class level

There are circumstances when this is desired, like when using Dynamic Method Invocation. If an `execute` method is defined in the class, then it will be used for the action mapping, otherwise the method to be used will be determined when a request is made (by Dynamic Method Invocation for example)

InterceptorRef annotation

Interceptors can be specified at the method level, using the **Action** annotation or at the class level using the `InterceptorRefs` annotation. Interceptors specified at the class level will be applied to all actions defined on that class. In the following example:

com.example.actions.HelloWorld

```
package com.example.actions;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.Actions;

@InterceptorRefs({
    @InterceptorRef("interceptor-1"),
    @InterceptorRef("defaultStack")
})
public class HelloWorld extends ActionSupport {
  @Action(value="action1", interceptorRefs=@InterceptorRef("validation"))
  public String execute() {
    return SUCCESS;
  }

  @Action(value="action2")
  public String doSomething() {
    return SUCCESS;
  }
}
```

The following interceptors will be applied to "action1": `interceptor-1`, all interceptors from `defaultStack`, `validation`.
All interceptors from `defaultStack` will be applied to "action2".

If you get errors like "Unable to find interceptor class referenced by ref-name XYZ". This means that the package where Convention is placing your actions, does not extend the package where the interceptor is defined. To fix this problem either 1)Use @ParentPackage annotation(or `struts.convention.default.parent.package`) passing the name of the package that defines the interceptor, or 2) Create a package in XML that extends the package that defines the interceptor, and use @ParentPackage(or `struts.convention.default.parent.package`) to point to it.

Result annotation

The Convention plugin allows action classes to define different results for an action. Results fall into two categories, global and local. Global results are shared across all actions defined within the action class. These results are defined as annotations on the action class. Local results apply only to the action method they are defined on. Here is an example of the different types of result annotations:

com.example.actions.HelloWorld

```
package com.example.actions;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.Actions;
import org.apache.struts2.convention.annotation.Result;
import org.apache.struts2.convention.annotation.Results;

@Results({
  @Result(name="failure", location="fail.jsp")
})
public class HelloWorld extends ActionSupport {
  @Action(value="/different/url",
    results={@Result(name="success", location="http://struts.apache.org", type="redirect")}
  )
  public String execute() {
    return SUCCESS;
  }

  @Action("/another/url")

  public String doSomething() {
    return SUCCESS;
  }
}
```

Parameters can be passed to results using the **params** attribute. The value of this attribute is a string array with an even number of elements in the form {"key0", "value0, "key1", "value1" ... "keyN", "valueN"}. For example:

com.example.actions.HelloWorld

```
package com.example.actions;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.Actions;
import org.apache.struts2.convention.annotation.Result;
import org.apache.struts2.convention.annotation.Results;

public class HelloWorld extends ActionSupport {
  @Action(value="/different/url",
    results={@Result(name="success", type="httpheader", params={"status", "500", "errorMessage", "Internal
Error"})}
  )
  public String execute() {
    return SUCCESS;
  }

  @Action("/another/url")
  public String doSomething() {
    return SUCCESS;
  }
}
```

From 2.1.7 on, global results (defined on the class level) defined using annotations will be inherited. Child classes can override the inherited result(s) by redefining it. Also, results defined at the method level take precedence (overwrite), over results with the same name at the action level.

Namespace annotation

The namespace annotation allows the namespace for action classes to be changed instead of using the convention of the Java package name. This annotation can be placed on an action class or within the package-info.java class that allows annotations to be placed on Java packages. When this annotation is put on an action class, it applies to all actions defined in the class, that are not fully qualified action URLs. When this annotation is place in the package-info.java file, it changes the default namespace for all actions defined in the Java package. Here is an example of the annotation on an action class:

com.example.actions.HelloWorl

```
package com.example.actions;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.Namespace;

@Namespace("/custom")
public class HelloWorld extends ActionSupport {
  @Action("/different/url")
  public String execute() {
    return SUCCESS;
  }

  @Action("url")
  public String doSomething() {
    return SUCCESS;
  }
}
```

In this example, the action will respond to two different URLs `/different/url` and `/custom/url`.

Here is an example of using this annotation in the package-info.java file:

com/example/actions/package-info.java

```
@org.apache.struts2.convention.annotation.Namespace("/custom")
package com.example.actions;
```

This changes the default namespace for all actions defined in the package `com.example.actions`. This annotation however doesn't apply to sub-packages.

ResultPath annotation

The ResultPath annotation allows applications to change the location where results are stored. This annotation can be placed on an action class and also in the package-info.java file. Here is an example of using this annotation:

com.example.actions.HelloWorl

```
package com.example.actions;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.ResultPath;

@ResultPath("/WEB-INF/jsps")
public class HelloWorld extends ActionSupport {
  public String execute() {
    return SUCCESS;
  }
}
```

The result for this class will be located in `WEB-INF/jsps` rather than the default of `WEB-INF/content`.

ParentPackage annotation

The ParentPackage annotation allows applications to define different parent Struts package for specific action classes or Java packages. Here is an example of using the annotation on an action class:

com.example.actions.HelloWorld

```
package com.example.actions;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.ParentPackage;

@ParentPackage("customXWorkPackage")
public class HelloWorld extends ActionSupport {
  public String execute() {
    return SUCCESS;
  }
}
```

To apply this annotation to all actions in a package (and subpackages), add it to package-info.java. An alternative to this annotation is to set `struts.convention.default.parent.package` in XML.

ExceptionMapping Annotation

This annotation can be used to define exception mappings to actions. See the exception mapping documentation for more details. These mappings can be applied to the class level, in which case they will be applied to all actions defined on that class:

ExceptionsActionLevelAction.java

```
@ExceptionMappings({
    @ExceptionMapping(exception = "java.lang.NullPointerException", result = "success", params = {"param1",
"val1"})
})
public class ExceptionsActionLevelAction {

    public String execute() throws Exception {
        return null;
    }
}
```

The parameters defined by `params` are passed to the result. Exception mappings can also be applied to the action level:

```
public class ExceptionsMethodLevelAction {
    @Action(value = "exception1", exceptionMappings = {
            @ExceptionMapping(exception = "java.lang.NullPointerException", result = "success", params =
{"param1", "val1"})
    })
    public String run1() throws Exception {
        return null;
    }
}
```

Actions in jar files

By default the Convention plugin will **not** scan jar files for actions. For a jar to be scanned, its URL needs to match at least one of the regular expressions in `struts.convention.action.includeJars`. In this example `myjar1.jar` and `myjar2.jar` will be scanned:

```
<constant name="struts.convention.action.includeJars" value=".*?/myjar1.*?jar(!/)?,.*?/myjar2*?jar(!/)?"
```

Note that **the regular expression will be evaluated against the URL of the jar, and not the file name**, the jar URL can contain a path to the jar file and a trailing "!/".

Automatic configuration reloading

The Convention plugin can automatically reload configuration changes, made in classes the contain actions, without restarting the container. This is a similar behavior to the automatic xml configuration reloading. To enable this feature, add this to your `struts.xml` file:

```
<constant name="struts.devMode" value="true"/>
<constant name="struts.convention.classes.reload" value="true" />
```

This feature is experimental and has not been tested on all container, and it is **strongly** advised not to use it in production environments.

JBoss

When using this plugin with JBoss, you need to set the following constants:

```
<constant name="struts.convention.exclude.parentClassLoader" value="true" />
<constant name="struts.convention.action.fileProtocols" value="jar,vfsfile,vfszip" />
```

You can also check the JBoss 5 page for more details.

Jetty (embedded)

When using this plugin with Jetty in embedded mode, you need to set the following constants:

```
<constant name="struts.convention.exclude.parentClassLoader" value="false" />
<constant name="struts.convention.action.fileProtocols" value="jar,code-source" />
```

Troubleshooting

Tips

Namespaces and Results

Make sure the namespace of the action is matched by one of the locators. The rest of the namespace after the locator, will be the namespace of the action, and will be used to find the results. For example, a class called "ViewAction" in the package "my.example.actions.orders" will be mapped to the URL `/orders/view.action`, and the results must be under `/WEB-INF/content/orders`, like `/WEB-INF/content/orders/view-success.jsp`.

Use the Configuration Browser Plugin

Add the Config Browser Plugin plugin to the lib folder or maven dependencies, and then visit: http://localhost:8080/CONTEXT/config-browser/index.action, to see the current action mappings.

Enable trace or debug mode

The Convention plugin can generate a rather verbose output when set to debug mode for logging. Use "Trace" logging level if you are using the JDK logger. If you are using Log4J, you can do something like:

log4j.logger.org.apache.struts2.convention=DEBUG

Common Errors

1. I get an error like "There is no Action mapped for namespace /orders and action name view.". This means that the URL `/orders/view.action` is not mapping to any action class. Check the namespace and the name of the action.
2. I get an error like "No result defined for action my.example.actions.orders.ViewAction and result success". This means that the action was mapped to the right URL, but the Convention plugin was unable to find a `success` result for it. Check that the result file exists, like `/WEB-INF/content/orders/view-success.jsp`.
3. I get lots of errors like "java.lang.Exception: Could not load org/apache/velocity/runtime/resource/loader/ClasspathResourceLoader.class". This happens when `struts.convention.action.includeJars` is matching jar URLs from external jars.
4. I am using a custom interceptor stack and I get an error like "Unable to find interceptor class referenced by ref-name XYZ". This means that the package where Convention is placing your actions, does not extend the package where the interceptor is defined. To fix this problem either 1)Use

@ParentPackage annotation(or `struts.convention.default.parent.package`) passing the name of the package that defines the interceptor, or 2) Create a package in XML that extends the package that defines the interceptor, and use @ParentPackage(or `struts.convention.default.parent.package`) to point to it.

Overwriting plugin classes

The Convention plugin can be extended in the same fashion that Struts does. The following beans are defined by default:

```
<bean type="org.apache.struts2.convention.ActionConfigBuilder" name="convention" class="org.apache.struts2.
convention.PackageBasedActionConfigBuilder"/>
This interface defines how the action configurations for the current web application can be constructed. This
must find all actions that are not specifically defined in the struts XML files or any plugins. Furthermore, it
must make every effort to locate all action results as well.

<bean type="org.apache.struts2.convention.ActionNameBuilder" name="convention" class="org.apache.struts2.
convention.SEOActionNameBuilder"/>
This interface defines the method that is used to create action names based on the name of a class.

<bean type="org.apache.struts2.convention.ResultMapBuilder" name="convention" class="org.apache.struts2.
convention.DefaultResultMapBuilder"/>
This interface defines how results are constructed for an Action. The action information is supplied and the
result is a mapping of ResultConfig instances to the result name.

<bean type="org.apache.struts2.convention.InterceptorMapBuilder" name="convention" class="org.apache.struts2.
convention.DefaultInterceptorMapBuilder"/>
This interface defines how interceptors are built from annotations.

<bean type="org.apache.struts2.convention.ConventionsService" name="convention" class="org.apache.struts2.
convention.ConventionsServiceImpl"/>
This interface defines the conventions that are used by the convention plugin. In most cases the methods on
this class will provide the best default for any values and also handle locating overrides of the default via
the annotations that are part of the plugin.

<constant name="struts.convention.actionConfigBuilder" value="convention"/>
<constant name="struts.convention.actionNameBuilder" value="convention"/>
<constant name="struts.convention.resultMapBuilder" value="convention"/>
<constant name="struts.convention.interceptorMapBuilder" value="convention"/>
<constant name="struts.convention.conventionsService" value="convention"/>
```

To plugin a different implementation for one of these classes, implement the interface, define a bean for it, and set the appropriate constant's value with the name of the new bean, for example:

```
<bean type="org.apache.struts2.convention.ActionNameBuilder" name="MyActionNameBuilder" class="example.
SultansOfSwingNameBuilder"/>
<constant name="struts.convention.actionNameBuilder" value="MyActionNameBuilder"/>
```

Configuration reference

Add a **constant** element to your struts config file to change the value of a configuration setting, like:

```
<constant name="struts.convention.result.path" value="/WEB-INF/mytemplates/"/>
```

Name	Default Value	Description
struts.convention.action.alwaysMapExecute	true	Set to false, to prevent Convention from creating a default mapping to "execute" when there are other methods annotated as actions in the class

struts.convention.action.includeJars		Comma separated list of regular expressions of jar URLs to be scanned. eg. ".**myJar-0\.2.**,.**thirdparty-0\.1.**"
struts.convention.action.packages		An optional list of action packages that this should create configuration for (they don't need to match a locator pattern)
struts.convention.result.path	/WEB-INF/content/	Directory where templates are located
struts.convention.result.flatLayout	true	If set to false, the result can be put in its own directory: resultsRoot/namespace/actionName/result.extension
struts.convention.action.suffix	Action	Suffix used to find actions based on class names
struts.convention.action.disableScanning	false	Scan packages for actions
struts.convention.action.mapAllMatches	false	Create action mappings, even if no @Action is found
struts.convention.action.checkImplementsAction	true	Check if an action implements com.opensymphony.xwork2.Action to create an action mapping
struts.convention.default.parent.package	convention-default	Default parent package for action mappins
struts.convention.action.name.lowercase	true	Convert action name to lowercase
struts.convention.action.name.separator	-	Separator used to build the action name, MyAction -> my-action. This character is also used as the separator between the action name and the result in templates, like action-result.jsp
struts.convention.package.locators	action, actions, struts, struts2	Packages whose name end with one of these strings will be scanned for actions
struts.convention.package.locators.disable	false	Disable the scanning of packages based on package locators

struts.convention.exclude.packages	org.apache.struts.*, org.apache.struts2.*, org.springframework.web.struts.*, org.springframework.web.struts2.*, org.hibernate.*	Packages excluded from the action scanning, packages already excluded cannot be included in other way, eg. org.demo.actions.exclude is specified as a part of the struts.convention.exclude.packages so all packages below are also excluded, eg. org.demo.actions.exclude.include even if **include** is specified as a struts.convention.package.locators or struts.convention.action.packages
struts.convention.package.locators.basePackage		If set, only packages that start with its value will be scanned for actions
struts.convention.relative.result.types	dispatcher,velocity, freemarker	The list of result types that can have locations that are relative and the result location (which is the resultPath plus the namespace) prepended to them
struts.convention.redirect.to.slash	true	A boolean parameter that controls whether or not this will handle unknown actions in the same manner as Apache, Tomcat and other web servers. This handling will send back a redirect for URLs such as /foo to /foo/ if there doesn't exist an action that responds to /foo
struts.convention.classLoader.excludeParent	true	Exclude URLs found by the parent class loader from the list of URLs scanned to find actions (needs to be set to `false` for JBoss 5)
struts.convention.action.eagerLoading	false	If set, found action classes will be instantiated by the ObjectFactory to accelerate future use, setting it up can clash with Spring managed beans