Action Configuration

The action mappings are the basic "unit-of-work" in the framework. Essentially, the action maps an identifier to a handler class. When a request matches the action's name, the framework uses the mapping to determine how to process the request.

- Action Mappings
- Action Names
    - Action Names With Slashes
    - Action Names with Dots and Dashes
    - Allowed action names
- Action Methods
- Wildcard Method
- Dynamic Method Invocation
    - Strict DMI
    - Strict Method Invocation
- ActionSupport Default
- Post-Back Default
- Action Default
    - Wildcard Default
- Next: Wildcard Mappings

Action Mappings

The action mapping can specify a set of result types, a set of exception handlers, and an interceptor stack. Only the `name` attribute is required. The other attributes can also be provided at package scope.

A Logon Action

```
<action name="Logon" class="tutorial.Logon">
  <result type="redirectAction">Menu</result>
  <result name="input">/Logon.jsp</result>
</action>
```

When using Convention Plugin the action mapping can be configured with annotations:

A Logon Action with annotations

```
package tutorial

@Action("Logon") // actually that is not necessary as it is added by convention
@Results(
        @Result(type="redirectAction", location="Menu"),
        @Result(name="input", location="/Logon.jsp")
)
public class Logon {
```

Action Names

In a web application, the `name` attribute is matched as part of the location requested by a browser (or other HTTP client). The framework will drop the host and application name and the extension and match what's in the middle: the action name. So, a request for `http://www.planetstruts.org/struts2-mailreader/Welcome.action` will map to the `Welcome` action.

Within an application a link to an action is usually generated by a Struts Tag. The tag can specify the action by name, and the framework will render the default extension and anything else that is needed. Forms may also submit directly to a Struts Action name (rather than a "raw" URI).

A Hello Form

```
<s:form action="Hello">
    <s:textfield label="Please enter your name" name="name"/>
    <s:submit/>
</s:form>
```

Action Names With Slashes

If your action names have slashes in them (for example, `<action name="admin/home" class="tutorial.Admin"/>`) you need to specifically allow slashes in your action names via a constant in the `struts.xml` file by specifying `<constant name="struts.enable.SlashesInActionNames" value="true"/>`. See JIRA Issue WW-1383 for discussion as there are side effects to setting this property to `true`.

Action Names with Dots and Dashes

Although action naming is pretty flexible, one should pay attention when using dots (eg. create.user) and/or dashes (eg. my-action). While the dot notation has no known side effects at this time, the dash notation will cause problems with the generated JavaScript for certain tags and themes. Use with caution, and always try to use camelcase action names (eg. createUser) or underscores (eg. my_action).

Allowed action names

`DefaultActionMapper` is using pre-defined RegEx to check if action name matches allowed names. The default RegEx is defined as follow: `[a-zA-Z0-9._!/\-]*` - if at some point this doesn't match your action naming schema you can define your own RegEx and override the default using constant named `struts.allowed.action.names`, e.g.:

```
<struts>
  <constant name="struts.allowed.action.names" value="[a-z{}]"*/>
  ...
</struts>
```

NOTE: Please be aware that action names not matching the RegEx will rise an exception.

Action Methods

The default entry method to the handler class is defined by the Action interface.

Action interface

```
public interface Action {
    public String execute() throws Exception;
}
```

Implementing the Action interface is optional. If Action is not implemented, the framework will use reflection to look for an `execute` method.

Sometimes, developers like to create more than one entry point to an Action. For example, in the case of a data-access Action, a developer might want separate entry-points for `create`, `retrieve`, `update`, and `delete`. A different entry point can be specified by the `method` attribute.

```
<action name="delete" class="example.CrudAction" method="delete">
    ...
```

If there is no `execute` method and no other method specified in the configuration the framework will throw an exception.

Convention Plugin allows that by annotating methods:

Annotated action method

```
@Action("crud")
public class CrudAction {
        @Action("delete")
        public String delete() {
                ...
```

Wildcard Method

Many times, a set of action mappings will share a common pattern. For example, all your `edit` actions might start with the word "edit", and call the `edit` method on the Action class. The `delete` actions might use the same pattern, but call the `delete` method instead.

Rather than code a separate mapping for each action class that uses this pattern, you can write it once as a wildcard mapping.

```
<action name="*Crud" class="example.Crud" method="{1}">
    ...
```

Here, a reference to "editCrud" will call the `edit` method on an instance of the Crud Action class. Likewise, a reference to "deleteCrud" will call the `delete` method instead.

Another common approach is to postfix the method name and set it off with an exclamation point (aka "bang"), underscore, or other special character.

- "action=Crud_input"
- "action=Crud_delete"

To use a postfix wildcard, just move the asterisk and add an underscore.

```
<action name="Crud_*" class="example.Crud" method="{1}">
```

From the framework's perspective, a wildcard mapping creates a new "virtual" mapping with all the same attributes as a conventional, static mapping. As a result, you can use the expanded wildcard name as the name of validation, type conversion, and message resource files, just as if it were an Action name (which it is!).

- `Crud_input-validation.xml`
- `Crud_delete-conversion.xml`

If Wildcard Method mapping uses a "!" in the action name, the Wildcard Method will overlap with another flexible approach to mapping, Dynamic Method Invocation. To use action names that include the "!" character, set `struts.enable.DynamicMethodInvocation` to FALSE in the application configuration.

Dynamic Method Invocation

There's a feature embedded in Struts 2 that lets the "!" (bang) character invoke a method other than `execute`. It is called "Dynamic Method Invocation" aka DMI.

DMI will use the string following a "!" character in an action name as the name of a method to invoke (instead of `execute`). A reference to "`Category!create.action`", says to use the "Category" action mapping, but call the `create` method instead.

Another way to use DMI is to provide HTTP parameters prefixed with "`method:`". For example in the URL it could be "`Category.action?method:create=foo`", the parameter value is ignored. In POST-Requests that can be used e.g. with a hidden parameter (`<s:hidden name="method:create" value="foo" />`) or along with a button (`<s:submit method="create" />`).

For Struts 2, we added a switch to disable DMI for two reasons. First, DMI can cause security issues if POJO actions are used. Second, DMI overlaps with the Wildcard Method feature that we brought over from Struts 1 (and from Cocoon before that). If you have security concerns, or would like to use the "!" character with Wildcard Method actions, then set `struts.enable.DynamicMethodInvocation` to FALSE in the application configuration.

The framework does support DMI, but there are problems with way DMI is implemented. Essentially, the code scans the action name for a "!" character, and finding one, tricks the framework into invoking the other method instead of `execute`. The other method is invoked, but it uses the same configuration as the `execute` method, including validations. The framework "believes" it is invoking the `Category` action with the `execute` method.

The Wildcard Method feature is implemented differently. When a Wildcard Method action is invoked, the framework acts as if the matching action had been hardcoded in the configuration. The framework "believes" it's executing the action `Category!create` and "knows" it is executing the `create` method of the corresponding Action class. Accordingly, we can add for a Wildcard Method action mapping its own validations, message resources, and type converters, just like a conventional action mapping. For this reason, the Wildcard Method is preferred.

Strict DMI

In Struts 2.3, an option was added to restrict the methods that DMI can invoke. First, set the attribute `strict-method-invocation="true"` on your `<package>` element. This tells Struts to reject any method that is not explicitly allowed via either the `method` attribute (including wildcards) or the `<allowed-methods>` tag. Then specify `<allowed-methods>` as a comma-separated list of method names in your `<action>`. (If you specify a `method` attribute for your action, you do not need to list it in `<allowed-methods>`.)

Note that you can specify `<allowed-methods>` even without `strict-method-invocation`. This restricts access only for the specific actions that have `<allowed-methods>`.

Example struts.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
        "http://struts.apache.org/dtds/struts-2.3.dtd">
<struts>

  <constant name="struts.enable.DynamicMethodInvocation" value="true"/>

  <package name="default" extends="struts-default" strict-method-invocation="true">

        <action name="index" class="org.apache.struts2.examples.actions.Index">
                <result name="success" type="redirectAction">hello</result>
        </action>

        <action name="hello" class="org.apache.struts2.examples.actions.HelloAction">
                <result name="success">/WEB-INF/content/hello.jsp</result>
                <result name="redisplay" type="redirectAction">hello</result>
                <allowed-methods>add</allowed-methods>
        </action>

  </package>
</struts>
```

Strict Method Invocation

In Struts 2.5 the Strict DMI was extended and it's called **Strict Method Invocation** aka SMI. You can imagine that the DMI is a "border police", where SMI is a "tax police" and keeps eye on internals. With this version, SMI is enabled by default (`strict-method-invocation` attribute is set to `true` by default in `struts-default` package), you have option to disable it per package - there is no global switch to disable SMI for the whole application. To gain advantage of new configuration option please use the latest DTD definition:

Struts 2.5 DTD

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 2.5//EN"
        "http://struts.apache.org/dtds/struts-2.5.dtd">
<struts>
...
</struts>
```

SMI works in the following way:

- `<allowed-methods>` / `@AllowedMethods` is defined per action - SMI works without switching it on but just for those actions (plus adding `<global-allowed-methods/>`)
- SMI is enabled but no `<allowed-methods>` / `@AllowedMethods` are defined - SMI works but only with `<global-allowed-methods/>`
- SMI is disabled - call to any action method is allowed that matches the default RegEx - `([A-Za-z0-9_$]*)`

You can redefine the default RegEx by using a constant as follow `<constant name="struts.strictMethodInvocation.methodRegex" value="([a-zA-Z]*)"/>`

When using wildcard mapping in actions' definitions SMI works in two ways:

- SMI is disabled - any wildcard will be substituted with the default RegEx, ie.: `<action name="Person*" method="perform*">` will be translated into `allowedMethod = "regex:perform([A-Za-z0-9_$]*)".`
- SMI is enabled - no wildcard substitution will happen, you must strictly define which methods can be accessed by annotations or `<allowed-method/>` tag.

You can configure SMI per `<action/>` using `<allowed-methods/>` tag or via `@AllowedMethod` annotation plus using per `<package/>` `<global-allowed-methods/>`, see the examples below:

SMI via struts.xml

```
<package ...>
  ...
  <global-allowed-methods>execute,input,back,cancel,browse</global-allowed-methods>
  ...

  <action name="Bar">
    <allowed-methods>foo,bar</allowed-methods>
  </action>

  ...
</package>
```

SMI via annotation on action class level

```
@AllowedMethods("end")
public class ClassLevelAllowedMethodsAction {
    public String execute() {
        return ...
    }
}
```

SMI via annotation on package level (in package-info.java)

```
@org.apache.struts2.convention.annotation.AllowedMethods({"home", "start"})
package org.apache.struts2.convention.actions.allowedmethods;
```

Allowed methods can be defined as:

- literals ie. in xml: `execute,cancel` or in annotation: `{"execute", "cancel"}`
- patterns when using with wildcard mapping, i.e `<action ... method="do{2}"/>`
- RegExs using `regex:` prefix, ie: `<global-allowed-methods>execute,input,cancel,regex:user([A-Z]*)</global-allowed-methods>`

Please be aware when using your own `Configurationprovider` that the logic to set allowed methods is defined in built-in providers - `XmlConfigurationProvider` and `PackageBasedActionConfigBuilder` - and you must replicate such logic in your code as by default only `execute` method is allowed, even when SMI is disabled.

ActionSupport Default

If the class attribute in an action mapping is left blank, the `com.opensymphony.xwork2.ActionSupport` class is used as a default.

```
<action name="Hello">
   // ...
</action>
```

The ActionSupport class has an `execute` method that returns "success" and an `input` method that returns "input".
To specify a different class as the default Action class, set the `default-class-ref` package attribute.

For more about using wildcards, see Wildcard Mappings.

Post-Back Default

A good practice is to link to actions rather than pages. Linking to actions encapsulates which server page renders, and ensures that an Action class can fire before a page renders.

Another common workflow stategy is to first render a page using an alternate method, like `input` and then have it submit back to the default `execute` method.

Using these two strategies together creates an opportunity to use a "post-back" form that doesn't specify an action. The form simply submits back to the action that created it.

Posting Back

```
<s:form>
    <s:textfield label="Please enter your name" name="name"/>
    <s:submit/>
</s:form>
```

Action Default

Usually, if an action is requested, and the framework can't map the request to an action name, the result will be the usual "404 - Page not found" error. But, if you would prefer that an omnibus action handle any unmatched requests, you can specify a default action. If no other action matches, the default action is used instead.

```
<package name="Hello" extends="action-default">

    <default-action-ref name="UnderConstruction"/>

    <action name="UnderConstruction">
        <result>/UnderConstruction.jsp</result>
    </action>

    ...
```

There are no special requirements for the default action. Each package can have its own default action, but there should only be one default action per namespace.

One to a Namespace

The default action features should be set up so that there is only one default action per namespace. If you have multiple packages declaring a default action with the same namespace, there is no guarantee which action will be the default.

Wildcard Default

Using wildcards is another approach to default actions. A wildcard action at the end of the configuration can be used to catch unmatched references.

```
<action name="*">
  <result>/{1}.jsp</result>
</action>
```

When a new action is needed, just add a stub page.

⚠️ It's important to put a "catchall" wildcard mapping like this at the end of your configuration so it won't attempt to map every request!

Next: Wildcard Mappings