Struts 2

Contents

- The Introduction of Struts
- Struts architecture
- Configuration
- Action
- Result
- Result Types

The Introduction of Struts?

Struts

- Apache Struts is a free open-source framework for creating Java web applications.
  - http://struts.apache.org/
  - https://cwiki.apache.org/WW/tutorials.html
- The difference between web applications and conventional websites
  - web applications can create a dynamic response.
- A web application can interact with databases and business logic engines to customize a response.

What is Struts?

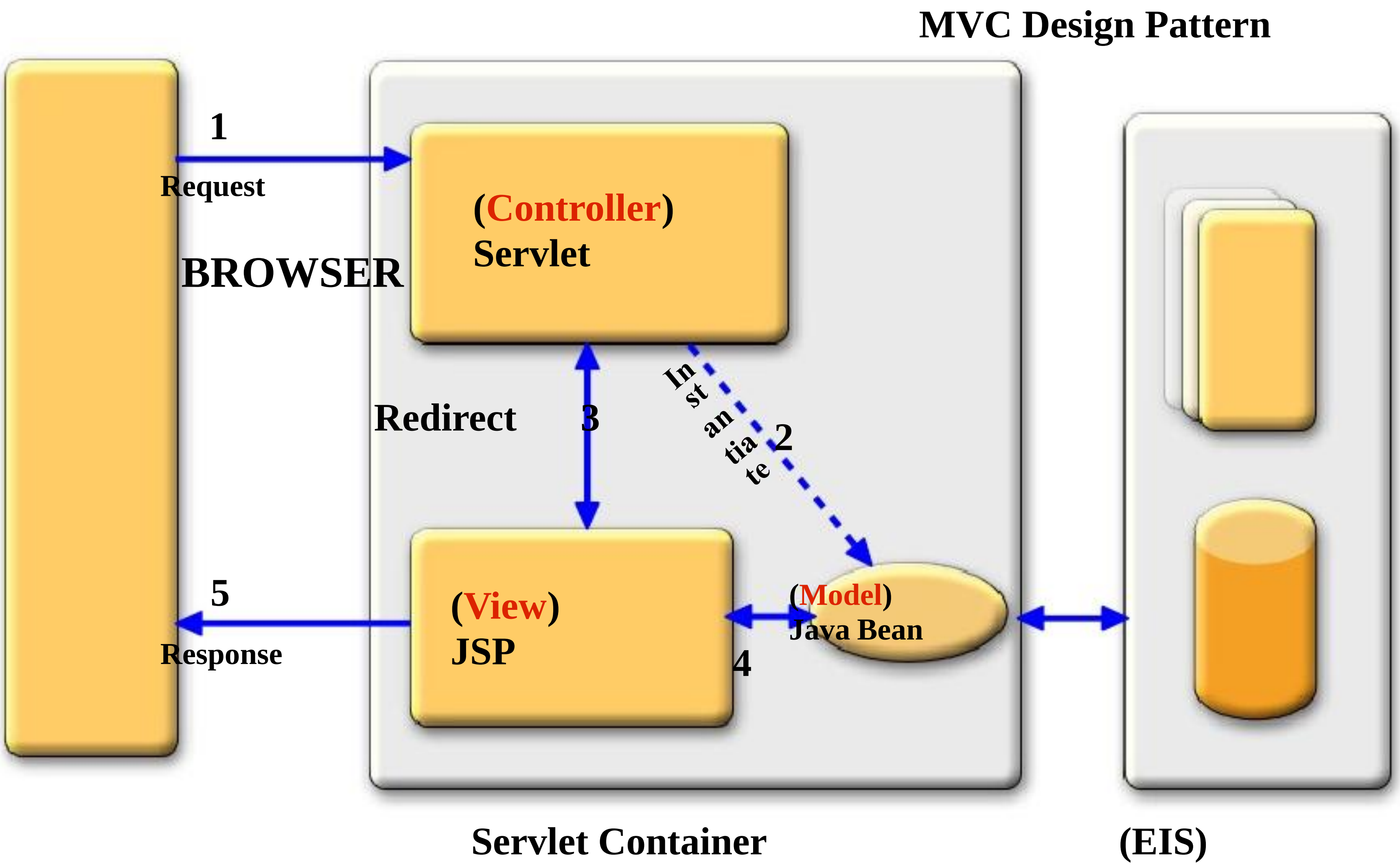
- Fashion their JSP/Servlet web applications using the Model-View-Controller (MVC) framework
- Utilize ready-to-usable framework objects through xml configuration files
- Utilize built-in design patterns in the framework
- Utilize extra features such as input validation, internationalization

Why Struts?

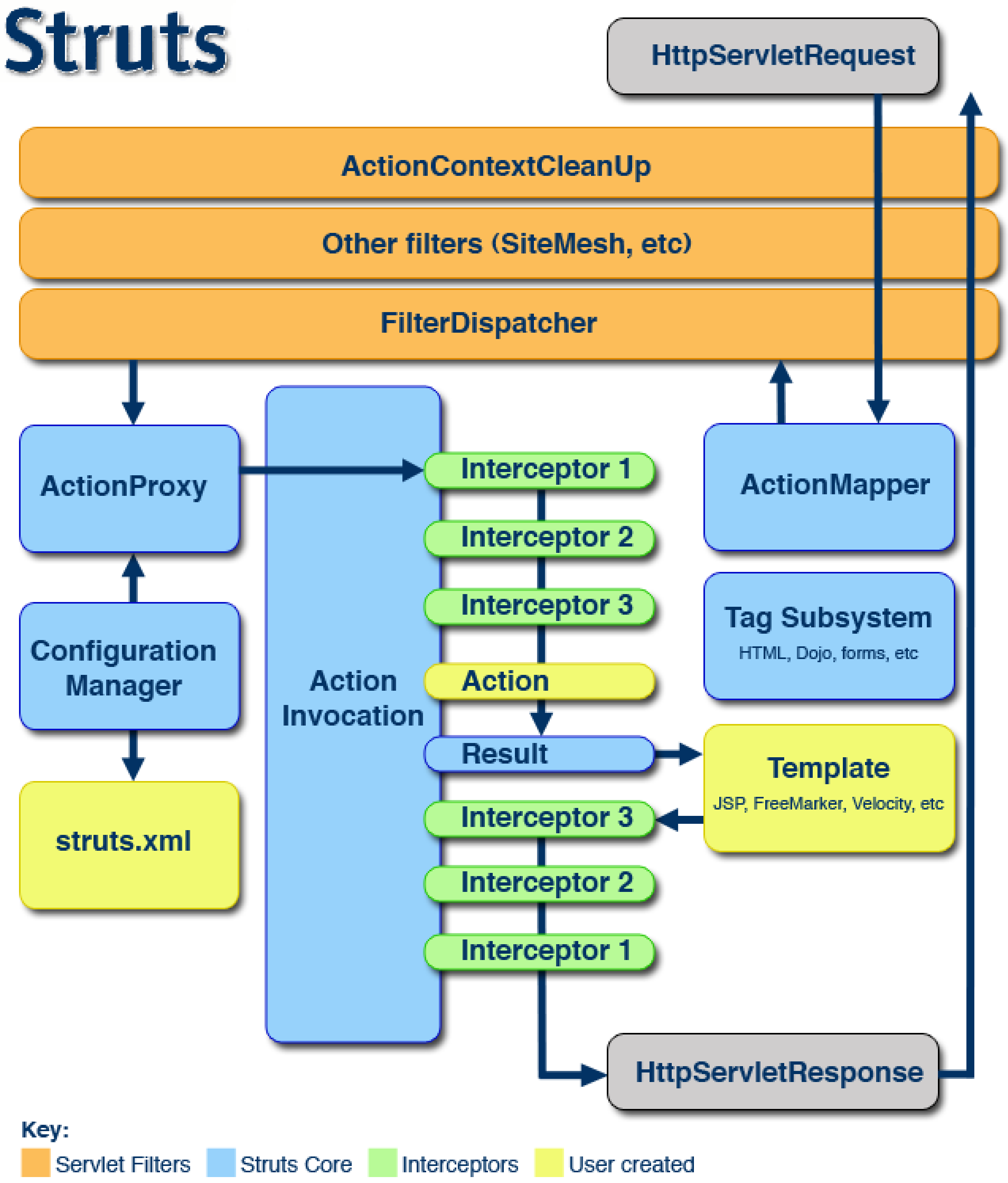
- Takes much of the complexity out of building your own MVC framework
- Encourages good design practice and modeling
- Easy to learn and use
- Feature-rich
- Many supported 3rd-party tools
- Flexible and extensible
- Large user community
- Stable and mature
- Open source

Struts Architecture

Model-View-Control (model 2)



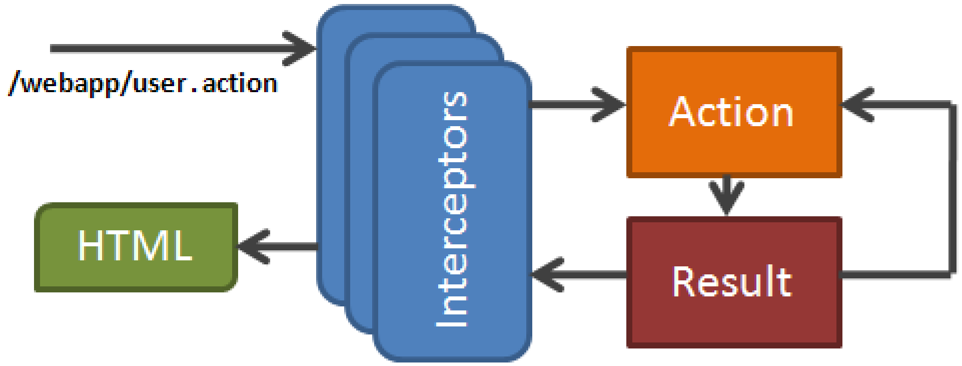
Architecture of Struts 2



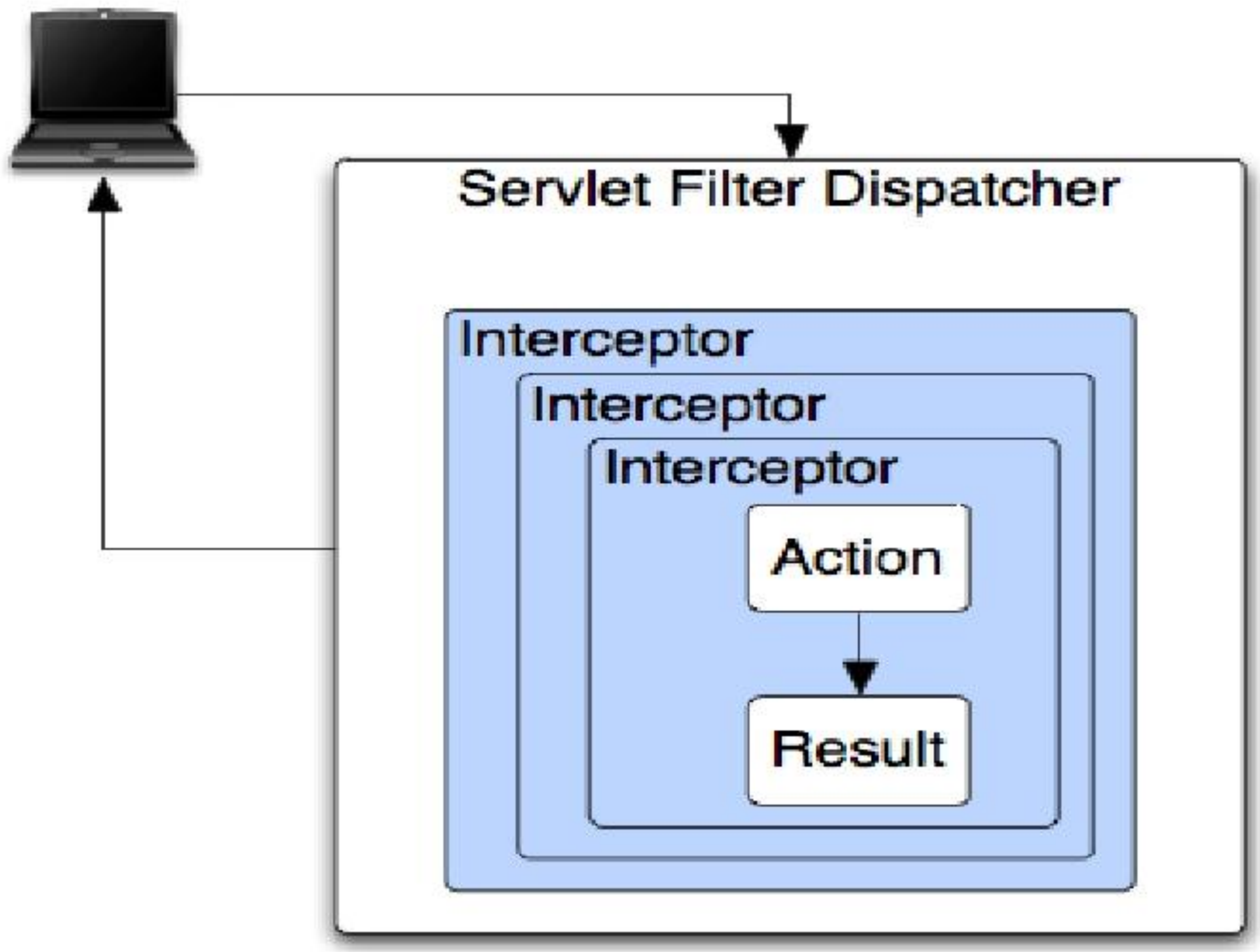
Architecture of Struts 2

- The normal lifecycle of struts begins when the request is sent from client. This results invoke the servlet container which in turn is passed through standard filter chain.
- The FilterDispatcher filter is called which consults the **ActionMapper** to determine whether an **Action** should be invoked.
- If ActionMapper finds an Action to be invoked, the FilterDispatcher delegates control to **ActionProxy**.
- ActionProxy reads the configuration file such as struts.xml. ActionProxy creates an instance of **ActionInvocation** class and delegates the control.
- ActionInvocation is responsible for command pattern implementation. It invokes the Interceptors one by one (if required) and then invoke the Action.
- Once the Action returns, the **ActionInvocation** is responsible for looking up the proper result associated with the Action result code mapped in struts.xml.
- The Interceptors are executed again in reverse order and the response is returned to the Filter (In most cases to FilterDispatcher). And the result is then sent to the servlet container which in turns send it back to client.

Request Processing Lifecycle



Request Processing Lifecycle



Request Processing Lifecycle

- Request is generated by user and sent to Servlet container.
- Servlet container invokes FilterDispatcher filter which in turn determines appropriate action.
- One by one Intercetors are applied before calling the Action.
- Interceptors apply common functionality to the request, such as Logging, Validation, File Upload, Double-submit guard etc.
- Action is executed, usually storing and/or retrieving information from a database, and the Result is generated by Action.
- The output of Action is rendered in the view (JSP,freemarker etc) and the result is returned to the user.

Struts: MVC-based Architecture

- Central controller mediates application flow and delegates to appropriate handler called Action
- Action Handlers can use model components
- Model encapsulates business logic or state
- Control forwarded back through the Controller to the appropriate View
  - The forwarding can be determined by consulting a set of mappings in configuration file, which provides a loose coupling between the View and Model

Struts: MVC-based Architecture

- 3 Major Components in Struts
  - Servlet controller (Controller)
  - Java Server Pages or any other presentation technology (View)
  - Application Business Logic in the form of whatever suits the application (Model)
- Struts is focused on Controller
  - Struts is Model and View independent
  - Struts can use any Model and View technologies

Steps to build a Struts 2 web application

1. Create development directory structure
   – If you are using NetBeans, the development directory structure is automatically created.
2. Write web.xml
3. Write struts.xml
4. Write Action classes
5. Create ApplicationResource.properties
6. Write JSP pages
7. Build, deploy, and test the application

Developer Responsibility

- Write an Action class (an extension of the class *com.opensymphony.xwork2.ActionSupport*) for each logical request that may be received
  - override execute() method
- Write the action mapping configuration file
  - struts.xml
- Update the web application deployment descriptor file to define the filter.
  - web.xml

Configuration

Configuration Files

- two important configuration files:
  - web.xml
  - Web deployment descriptor to include all necessary framework components
  - struts.xml
  - Main configuration, contains result/view types, action mappings, interceptors, and so forth

web.xml

- In the web.xml file, Struts defines its FilterDispatcher, the Servlet Filter class that initializes the Struts framework and handles all requests.

struts.xml

- Struts uses a configuration file to initialize its own resources. These resources include:
  - *Interceptors* that can preprocess and postprocess a request
  - *Action classes* that can call business logic and data access code
  - *Results* that can prepare views, like JavaServer Pages and FreeMarker templates

Action

Action Class

- Action classes act as the controller in the MVC pattern.
- Action classes respond to a user action, execute business logic (or call upon other classes to do that), and then return a result that tells Struts what view to render.
- The detination resource could be
  - JSP
  - Tile definition
  - Velocity template
  - Another Action

What is Action Class?

- POJO class which has *execute()* method
- Java class that does the “work” of your application
  - Handle request
  - Perform business logic
- Can be simple or sophisticated
- Simple action class does handle business logic by itself
- Sophisticated ones delegate business logic to Model components
  - Action class functions as a Facade pattern in this case

Coding a Struts 2 Action

- Coding a Struts 2 Action involves several parts:
  - Mapping an action to a class
  - Mapping a result to a view
  - Writing the controller logic in the Action class

Action Mapping Configuration

- Maps an identifier to a *Action* class.
- When a request matches the action's name, the framework uses the mapping to determine how to process the request.
- Also specifies
  - A set of result types
  - A set of exception handlers
  - An interceptor stack

Example: Action Mapping

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

Action Names

- Within an application, the 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.

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

Action Names

- 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"/>.
- Be careful to use 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.
- always try to use camelcase action names (eg. createUser) or underscores (eg. my_action).

Example: Plain JSP

```
<html>
  <head><title>Add Blog Entry</title></head>
  <body>
      <form action="save.action" method="post">
         Title: <input type="text" name="title" /><br/>
         Entry: <textarea rows="3" cols="25" name="entry"></textarea>
      <br/>
            <input type="submit" value="Add"/>
      </form>
    </body>
</html>
```

Example: Using Struts Tag

```
<%@ taglib prefix="s" uri="/WEB-INF/struts-tags.tld" %>
<html>
    <head><title>Add Blog Entry</title></head>
    <body>
    <s:form action="save" method="post" >
    <s:textfield label="Title" name="title" />
    <s:textarea label="Entry" name="entry" rows="3" cols="25" />
    <s:submit value="Add"/>
    </s:form>
    </body>
</html>
```

Action Interface

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

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

- Struts 2 Action classes usually extend the ActionSupport class (com.opensymphony.xwork2.ActionSupport), which is provided by the Struts 2 framework.
- Implementing the Action interface is optional.
  - If Action is not implemented, the framework will use reflection to look for an *execute* method.

Action Methods

- Sometimes, developers like to create more than one entry point to an Action.
  - For example, in the case of 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">
```

Wildcard Method

- a set of action mappings may share a common pattern
- Rather than code a separate mapping for each action class, you can write it once as a wildcard mapping.

Example: Wildcard Method

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

– *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.*

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

Action Default

- To 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>
```

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>
```

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.

Result

Result

- When an Action class method completes, it returns a String.
  - The value of the String is used to select a result element.
  - An action mapping will often have a set of results representing different possible outcomes.
- There are predefined result names (tokens)
- Applications can define other result names (tokens) to match specific cases.

Pre-defined result names (tokens)

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

Result Element

- Provides a logical name (with *name* attribute)
  - An Action can pass back a token like "success" or "error" without knowing any other implementation details.
  - If the *name* attribute is not specified, the framework will give it the name "*success*".
- Provides a Result Type (with *type* attribute)
  - Most results simply forward to a server page or template, but other Result Types can be used to do more interesting things.
  - If a *type* attribute is not specified, the framework will use the *dispatcher*

Result element

- Result element without defaults

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

- Result element using some defaults (as above)

```
<result>
    <param name="location">/ThankYou.jsp</param>
</result>
```

- Result element using default for the <param> as well

```
<result>/ThankYou.jsp</result>
```

Example: Multiple Results

```
<action name="Hello">
        <result>/hello/Result.jsp</result> <!-- name=”success” -->
        <result name="error">/hello/Error.jsp</result>
        <result name="input">/hello/Input.jsp</result>
</action>
```

Result Types

Predefined Result Types

- Dispatcher Result
- Redirect Action Result
- Chain Result
- Redirect Result
- FreeMarker Result
- Velocity Result
- PlainText Result
- Tiles Result
- HttpHeader Result
- Stream Result

Setting a default Result Type

- If a *type* attribute is not specified, the framework will use the *dispatcher*.
  - The default Result Type, dispatcher, forwards to another web resource.
- A default Result Type can be set as part of the configuration for each package.

```
<result-types>
    <result-type name="dispatcher"
    class="org.apache.struts2.dispatcher.ServletDispatcherResult" default="true"/>
</result-types>
```

Global Results

- Most often, results are nested with the action element. But some results apply to multiple actions.
  - Example: In a secure application, a client might try to access a page without being authorized, and many actions may need access to a "logon" result.
- If actions need to share results, a set of global results can be defined for each package.
- The framework will first look for a local result nested in the action. If a local match is not found, then the global results are checked.

Example: Global Results

```
<global-results>
        <result name="error">/Error.jsp</result>
        <result name="invalid.token">/Error.jsp</result>
        <result name="login" type="redirect-
    action">Logon!input</result>
</global-results>
```

Dynamic Results

- A result may not be known until execution time.
- Result values may be retrieved from its corresponding Action implementation by using EL expressions that access the Action's properties

Example: Dynamic Results

- Give the following Action fragment

```
private String nextAction;
public String getNextAction() {
                return nextAction;
}
```

- You might define a result like following

```
<action name="fragment" class="FragmentAction">
  <result name="next"
        type="redirect-action">${nextAction}</result>
</action>
```

Example: Dynamic Results

- In the code below, if it returns success, then the browser will be forwarded to
- **/<app-prefix>/myNamespace/otherAction.action?id=<value of id>**

```
<action name="myAction" class="com.project.MyAction">
    <result name="success"
          type="redirect-action">otherAction?id=${id}</result>
    <result name="back"
          type="redirect">${redirectURL}</result>
</action>
```

Action Chaining

- The framework provides the ability to chain multiple actions into a defined sequence or workflow.
- This feature works by applying a Chain Result to a given Action

Chain Result

- The Chain Result is a result type that invokes an Action with its own Interceptor Stack and Result.
- This Interceptor allows an Action to forward requests to a target Action, while propagating the state of the source Action.

```
<package name="public" extends="struts-default">
    <!-- Chain creatAccount to login, using the default parameter -->
    <action name="createAccount" class="...">
        <result type="chain">login</result>
    </action>

    <action name="login" class="...">
        <!-- Chain to another namespace -->
        <result type="chain">
            <param name="actionName">dashboard</param>
            <param name="namespace">/secure</param>
        </result>
    </action>
</package>

<package name="secure" extends="struts-default" namespace="/secure">
    <action name="dashboard" class="...">
        <result>dashboard.jsp</result>
    </action>
</package>
```