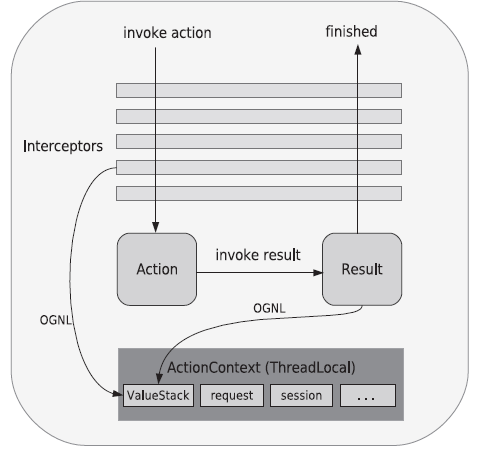
**How Struts 2 works**

We said that Struts 2 provides a cleaner implementation of MVC.

These clean lines are only possible with the help of a few other key architectural components that participate in processing every request. Chief among these are the interceptors, OGNL, and the ValueStack. We’ll learn what each of these does in the following walkthrough of Struts 2 request processing.



In the figure, the FilterDispatcher has already done its controller work by selecting the appropriate action to handle the request.

The figure demonstrates what really happens when the action is invoked by the

controller. As you can see, a few extra parts are added to the MVC basics.

We’ll explain in the next paragraphs how the interceptors and the ActionContext aid the action and result in their processing of the request.

Figure 1.4 introduces the following new Struts 2 components:

ActionContext,

interceptors,

the ValueStack,

OGNL.

This diagram goes a long way toward showing what really happens in Struts 2.

You could say that everything we’ll discuss in this book is shown in this diagram.

As interceptors come first in the request-processing cycle, we’ll start with them. The name seems obvious, but what exactly do they intercept?

**INTERCEPTORS**

You may have noticed, while studying figure 1.4, that there is a stack of interceptors in

front of the action.

The invocation of the action must travel through this stack.

This is a key part of the Struts 2 framework. We’ll devote an entire chapter to this important component later in the book.

At this time, it is enough to understand that most every action will have a stack of interceptors associated with it.

These interceptors are invoked both before and after the action, though we should note that they actually fire after the result has executed. Interceptors don’t necessarily have to do something both times they fire, but they do have the opportunity.

Some interceptors only do work before the action has been executed, and others only do work afterward.

The important thing is that the interceptor allows common, cross-cutting tasks to be defined in clean, reusable components that you can keep separate from your action code.

Interceptors are Struts 2 components that execute both before and

after the rest of the request processing.

They provide an architectural component in which to define various workflow and cross-cutting tasks so that they can be easily reused as well as separated from other architectural concerns.

What kinds of work should be done in interceptors? Logging is a good example. Logging should be done with the invocation of every action, but it probably shouldn’t be

put in the action itself. Why? Because it’s not part of the action’s own unit of work. It’s

more administrative, overhead if you will.

Earlier, we charged a framework with the responsibility of providing built-in functional solutions to common domain tasks such as data validation, type conversion, and file uploads. Struts 2 uses interceptors to do this type of work. While these tasks are important, they’re not specifically related to the action logic of the request. Struts 2 uses interceptors to both separate and reuse these cross-cutting concerns.

Interceptors play a huge role in the Struts 2 framework.

And while you probably won’t spend a large percentage of your time writing interceptors, most developers will find that many tasks are perfectly solved with custom interceptors.

***The Struts 2 framework* 17**

**THE VALUESTACK AND OGNL**

The ValueStack is a storage area that holds all of the data associated with the processing of a request.

You could think of it as a piece of scratch paper where the framework does its work

while solving the problems of request processing.

Rather than passing the data around, Struts 2 keeps it in a convenient, central location—the ValueStack.

OGNL is the tool that allows us to access the data we put in that central repository.

More specifically, it is an expression language that allows you to reference and manipulate the data on the ValueStack.

The tricky, and powerful, thing about the ValueStack and OGNL is that they don’t

belong to any of the individual framework components.

Both interceptors and results can use OGNL to target values on the ValueStack

The data in the ValueStack follows the request processing through all phases;

it slices through the whole length of the framework.

It can do this because it is stored in a ThreadLocal context called the Ac

OGNL is a powerful expression language (and more) that is used to reference

and manipulate properties on the ValueStack.

The ActionContext contains all of the data that makes up the context in which an

action occurs.

This includes the ValueStack but also includes stuff the framework itself will use internally, such as the request, session, and application maps from the

Servlet API.

You can access these objects yourself if you like;

. The use of ThreadLocal makes the ActionContext, and thus the ValueStack, accessible from anywhere in the same thread of execution.

Since Struts 2’s processing of each request occurs in a single thread, the ValueStack

is available from any point in the framework’s handling of a request.

Typically, it is considered bad form to obtain the contents of the ActionContext

yourself.

The framework provides many elegant ways to interact with that data without

actually touching the ActionContext, or the ValueStack,

Primarily, you’ll use OGNL to do this. OGNL is used in many places in the framework to reference and manipulate data in the ValueStack.

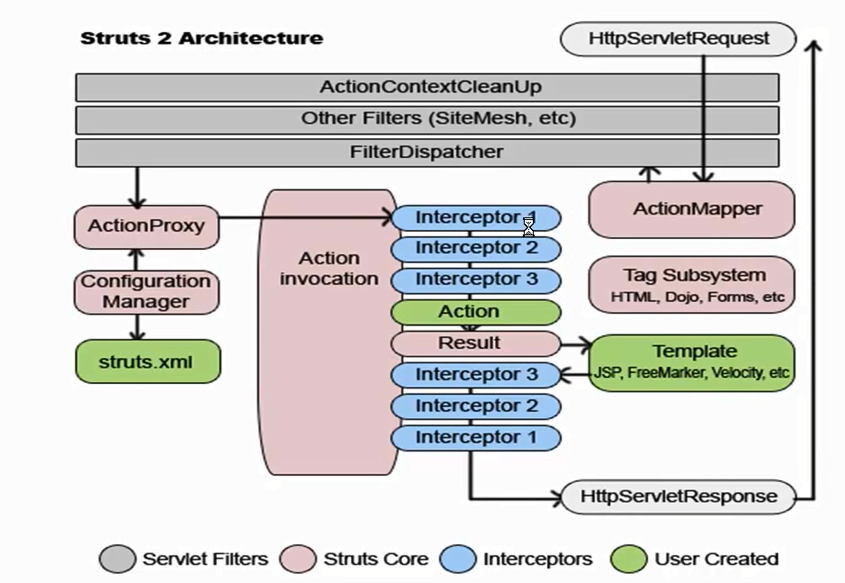
For instance, you’ll use OGNL to bind HTML form fields to data objects on the ValueStack for data transfer, and you’ll use OGNL to pull

data into the rendering of your JSPs and other result types.

At this point, you just need to understand that the ValueStack is where your data is stored while you work with it,

and that OGNL is the expression language that you, and the framework, use to target

this data from various parts of the request-processing cycle.

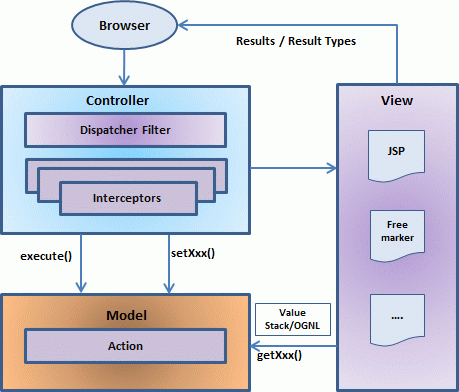


\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

From a high level, Struts2 is a pull-MVC (or MVC2) framework. The Model-View-Controller pattern in Struts2 is realized with following five core components:

* **Actions**
* **Interceptors**
* **Value Stack / OGNL**
* **Results / Result types**
* **View technologies**

Struts 2 is slightly different from a traditional MVC framework in that the action takes the role of the model rather than the controller, although there is some overlap.



The above diagram depicts the **M**odel, **V**iew and **C**ontroller to the Struts2 high level architecture. The controller is implemented with a Struts2 dispatch servlet filter as well as interceptors, the model is implemented with actions, and the view as a combination of result types and results. The value stack and OGNL provide common thread, linking and enabling integration between the other components.

Apart from the above components, there will be a lot of information that relates to configuration. Configuration for the web application, as well as configuration for actions, interceptors, results, etc.

This is the architectural overview of the Struts 2 MVC pattern. We will go through each component in more detail in the subsequent chapters.

Request life cycle:

Based on the above diagram, one can explain the user's request life cycle in Struts 2 as follows:

* User sends a request to the server for requesting for some resource (i.e pages).
* The FilterDispatcher looks at the request and then determines the appropriate Action.
* Configured interceptors functionalities applies such as validation, file upload etc.
* Selected action is executed to perform the requested operation.
* Again, configured interceptors are applied to do any post-processing if required.
* Finally the result is prepared by the view and returns the result to the user.

# Struts 2 Hello World Example

As you learnt from the Struts 2 architecture, when you click on a hyperlink or submit an HTML form in a Struts 2 web application, the input is collected by the Controller which is sent to a Java class called Actions. After the Action is executed, a Result selects a resource to render the response. The resource is generally a JSP, but it can also be a PDF file, an Excel spreadsheet, or a Java applet window.

Assume you already build-up your development environment. Now let us proceed for building our first **Hello World** struts2 project. The aim of this project is to build a web application that collects the user's name and displays "Hello World" followed by the user name. We would have to create following four components for any Struts 2 project:

|  |  |
| --- | --- |
| **SN** | **Components & Description** |
| 1 | **Action** Create an action class which will contain complete business logic and control the interaction between the user, the model, and the view. |
| 2 | **Interceptors** Create interceptors if required, or use existing interceptors. This is part of Controller. |
| 3 | **View** Create a JSPs to interact with the user to take input and to present the final messages. |
| 4 | **Configuration Files** Create configuration files to couple the Action, View and Controllers. These files are struts.xml, web.xml, struts.properties. |

I am going to use Eclipse IDE, so all the required components will be created under a Dynamic Web Project. So let us start with creating Dynamic Web Project.

## The struts.xml file:

The **struts.xml** file contains the configuration information that you will be modifying as actions are developed. This file can be used to override default settings for an application, for example *struts.devMode = false* and other settings which are defined in property file. This file can be created under the folder **WEB-INF/classes**.

Let us have a look at the struts.xml file we created in the Hello World example explained in previous chapter.

The first thing to note is the **DOCTYPE**. All struts configuration file need to have the correct doctype as shown in our little example. <struts> is the root tag element, under which we declare different packages using <package> tags. Here <package> allows separation and modularization of the configuration. This is very useful when you have a large project and project is divided into different modules.

Say, if your project has three domains - business\_applicaiton, customer\_application and staff\_application, you could create three packages and store associated actions in the appropriate package. The package tag has the following attributes:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| name (required) | The unique identifier for the package |
| extends | Which package does this package extend from? By default, we use struts-default as the base package. |
| abstract | If marked true, the package is not available for end user consumption. |
| namesapce | Unique namespace for the actions |

The **constant** tag along with name and value attributes will be used to override any of the following properties defined in **default.properties**, like we just set **struts.devMode** property. Setting **struts.devMode** property allows us to see more debug messages in the log file.

We define **action** tags corresponds to every URL we want to access and we define a class with execute() method which will be accessed whenever we will access corresponding URL.

Results determine what gets returned to the browser after an action is executed. The string returned from the action should be the name of a result. Results are configured per-action as above, or as a "global" result, available to every action in a package. Results have optional **name** and **type** attributes. The default name value is "success".

Struts.xml file can grow big over time and so breaking it by packages is one way of modularizing it, but struts offers another way to modularize the struts.xml file. You could split the file into multiple xml files and import them in the following fashion.

The other configuration file that we haven't covered is the struts-default.xml. This file contains the standard configuration settings for Struts and you would not have to touch these settings for 99.99% of your projects. For this reason, we are not going into too much detail on this file. If you are interested, take a look into the at the **default.properties** file available in struts2-core-2.2.3.jar file.

## The struts-config.xml file:

The struts-config.xml configuration file is a link between the View and Model components in the Web Client but you would not have to touch these settings for 99.99% of your projects. The configuration file basically contains following main elements:

|  |  |
| --- | --- |
| **SN** | **Interceptor & Description** |
| 1 | **struts-config** This is the root node of the configuration file. |
| 2 | **form-beans** This is where you map your ActionForm subclass to a name. You use this name as an alias for your ActionForm throughout the rest of the struts-config.xml file, and even on your JSP pages. |
| 3 | **global forwards** This section maps a page on your webapp to a name. You can use this name to refer to the actual page. This avoids hardcoding URLs on your web pages. |
| 4 | **action-mappings** This is where you declare form handlers and they are also known as **action mappings**. |
| 5 | **controller** This section configures Struts internals and rarely used in practical situations. |
| 6 | **plug-in** This section tells Struts where to find your properties files, which contain prompts and error messages |

Following is the sample struts-config.xml file:

For more detail on struts-config.xml file, kindly check your struts documentation.

## The struts.properties file

This configuration file provides a mechanism to change the default behavior of the framework. Actually all of the properties contained within the **struts.properties** configuration file can also be configured in the **web.xml** using the **init-param**, as well using the **constant** tag in the **struts.xml** configuration file. But if you like to keep the things separate and more struts specific then you can create this file under the folder **WEB-INF/classes**.

The values configured in this file will override the default values configured in **default.properties** which is contained in the struts2-core-x.y.z.jar distribution. There are a couple of properties that you might consider changing using **struts.properties** file:

Actions are the core of the Struts2 framework, as they are for any MVC (Model View Controller) framework. Each URL is mapped to a specific action, which provides the processing logic necessary to service the request from the user.

But the action also serves in two other important capacities. First, the action plays an important role in the transfer of data from the request through to the view, whether its a JSP or other type of result. Second, the action must assist the framework in determining which result should render the view that will be returned in the response to the request.

## Create Action:

The only requirement for actions in Struts2 is that there must be one no-argument method that returns either a String or Result object and must be a POJO. If the no-argument method is not specified, the default behavior is to use the execute() method.

Optionally you can extend the **ActionSupport** class which implements six interfaces including **Action** interface. The Action interface is as follows:

Let us take a look at the action method in the Hello World example:

To illustrate the point that the action method controls the view, let us make the following change to the **execute** method and extend the class ActionSupport as follows:

In this example, we have some logic in the execute method to look at the name attribute. If the attribute equals to the string "SECRET", we return SUCCESS as the result otherwise we return ERROR as the result. Because we have extended ActionSupport, so we can use String constants SUCCESS and ERROR. Now, let us modify our struts.xml file as follows:

## Create a View

Let us create the below jsp file **HelloWorld.jsp** in the WebContent folder in your eclipse project. To do this, right click on the WebContent folder in the project explorer and select **New >JSP File**. This file will be called in case return result is SUCCESS which is a String constant "success" as defined in Action interface:

Following is the file which will be invoked by the framework in case action result is ERROR which is equal to String constant "error". Following is the content of **AccessDenied.jsp**

We also need to create **index.jsp** in the WebContent folder. This file will serve as the initial action URL where the user can click to tell the Struts 2 framework to call the **execute** method of the HelloWorldAction class and render the HelloWorld.jsp view.

That's it, there is no change required web.xml file, so let us use same web.xml which we had created in Examples chapter. Now we are ready to run our Hello World application using Struts 2 framework.

## Execute the Application

Right click on the project name and click **Export > WAR File** to create a War file. Then deploy this WAR in the Tomcat's webapps directory. Finally, start Tomcat server and try to access URL http://localhost:8080/HelloWorldStruts2/index.jsp. This will give you following screen

Interceptors are conceptually the same as servlet filters or the JDKs Proxy class. Interceptors allow for crosscutting functionality to be implemented separately from the action as well as the framework. You can achieve the following using interceptors:

* Providing preprocessing logic before the action is called.
* Providing postprocessing logic after the action is called.
* Catching exceptions so that alternate processing can be performed.

Many of the features provided in the Struts2 framework are implemented using interceptors; examples include exception handling, file uploading, lifecycle callbacks and validation etc. In fact, as Struts2 bases much of its functionality on interceptors, it is not unlikely to have 7 or 8 interceptors assigned per action.

## Struts2 Framework Interceptors:

Struts 2 framework provides a good list of out-of-the-box interceptors that come preconfigured and ready to use. Few of the important interceptors are listed below:

|  |  |
| --- | --- |
| **SN** | **Interceptor & Description** |
| 1 | **alias** Allows parameters to have different name aliases across requests. |
| 2 | **checkbox** Assists in managing check boxes by adding a parameter value of false for check boxes that are not checked. |
| 3 | **conversionError** Places error information from converting strings to parameter types into the action's field errors. |
| 4 | **createSession** Automatically creates an HTTP session if one does not already exist. |
| 5 | **debugging** Provides several different debugging screens to the developer. |
| 6 | **execAndWait** Sends the user to an intermediary waiting page while the action executes in the background. |
| 7 | **exception** Maps exceptions that are thrown from an action to a result, allowing automatic exception handling via redirection. |
| 8 | **fileUpload** Facilitates easy file uploading. |
| 9 | **i18n** Keeps track of the selected locale during a user's session. |
| 10 | **logger** Provides simple logging by outputting the name of the action being executed. |
| 11 | **params** Sets the request parameters on the action. |
| 12 | **prepare** This is typically used to do pre-processing work, such as setup database connections. |
| 13 | **profile** Allows simple profiling information to be logged for actions. |
| 14 | **scope** Stores and retrieves the action's state in the session or application scope. |
| 15 | **ServletConfig** Provides the action with access to various servlet-based information. |
| 16 | **timer** Provides simple profiling information in the form of how long the action takes to execute. |
| 17 | **token** Checks the action for a valid token to prevent duplicate formsubmission. |
| 18 | **validation** Provides validation support for actions |

Please loook into Struts 2 documentation for complete detail on above mentioned interceptors. But I will show you how to use an interceptor in general in your Struts application.

## How to use Interceptors?

Let us see how to use an already existing interceptor to our "Hello World" program. We will use the **timer** interceptor whose purpose is to measure how long it took to execute an action method. Same time I'm using **params** interceptor whose purpose is to send the request parameters to the action. You can try your example without using this interceptor and you will find that **name** property is not being set because parameter is not able to reach to the action.

We will keep HelloWorldAction.java, web.xml, HelloWorld.jsp and index.jsp files as they have been created in Examples chapter but let us modify the **struts.xml** file to add an interceptor as follows

Right click on the project name and click **Export > WAR File** to create a War file. Then deploy this WAR in the Tomcat's webapps directory. Finally, start Tomcat server and try to access URL http://localhost:8080/HelloWorldStruts2/index.jsp. This will give you following screen:

Now enter any word in the given text box and click Say Hello button to execute the defined action. Now if you will check the log generated, you will find following text:

Here bottom line is being generated because of **timer** interceptor which is telling that action took total 109ms to be executed.

## Create Custom Interceptors

Using custom interceptors in your application is an elegant way to provide cross-cutting application features. Creating a custom interceptor is easy; the interface that needs to be extended is the following **Interceptor** interface:

As the names suggest, the init() method provides a way to initialize the interceptor, and the destroy() method provides a facility for interceptor cleanup. Unlike actions, interceptors are reused across requests and need to be thread-safe, especially the intercept() method.

The **ActionInvocation** object provides access to the runtime environment. It allows access to the action itself and methods to invoke the action and determine whether the action has already been invoked.

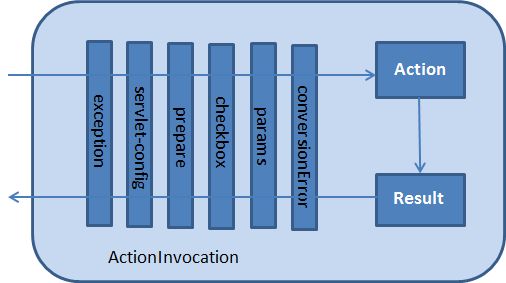
If you have no need for initialization or cleanup code, the **AbstractInterceptor** class can be extended. This provides a default no-operation implementation of the init() and destroy() methods.

## Create Interceptor Class:

Let us create following **MyInterceptor.java** in **Java Resources > src** folder:

As you notice, actual action will be executed using the interceptor by **invocation.invoke()** call. So you can do some pre-processing and some post-processing based on your requirement.

The framework itself starts the process by making the first call to the ActionInvocation object's invoke(). Each time **invoke()** is called, ActionInvocation consults its state and executes whichever interceptor comes next. When all of the configured interceptors have been invoked, the invoke() method will cause the action itself to be executed. Following digram shows the same concept through a request flow:



Interceptors are conceptually the same as servlet filters or the JDKs Proxy class. Interceptors allow for crosscutting functionality to be implemented separately from the action as well as the framework. You can achieve the following using interceptors:

* Providing preprocessing logic before the action is called.
* Providing postprocessing logic after the action is called.
* Catching exceptions so that alternate processing can be performed.

Many of the features provided in the Struts2 framework are implemented using interceptors; examples include exception handling, file uploading, lifecycle callbacks and validation etc. In fact, as Struts2 bases much of its functionality on interceptors, it is not unlikely to have 7 or 8 interceptors assigned per action.

## Struts2 Framework Interceptors:

Struts 2 framework provides a good list of out-of-the-box interceptors that come preconfigured and ready to use. Few of the important interceptors are listed below:

|  |  |
| --- | --- |
| **SN** | **Interceptor & Description** |
| 1 | **alias** Allows parameters to have different name aliases across requests. |
| 2 | **checkbox** Assists in managing check boxes by adding a parameter value of false for check boxes that are not checked. |
| 3 | **conversionError** Places error information from converting strings to parameter types into the action's field errors. |
| 4 | **createSession** Automatically creates an HTTP session if one does not already exist. |
| 5 | **debugging** Provides several different debugging screens to the developer. |
| 6 | **execAndWait** Sends the user to an intermediary waiting page while the action executes in the background. |
| 7 | **exception** Maps exceptions that are thrown from an action to a result, allowing automatic exception handling via redirection. |
| 8 | **fileUpload** Facilitates easy file uploading. |
| 9 | **i18n** Keeps track of the selected locale during a user's session. |
| 10 | **logger** Provides simple logging by outputting the name of the action being executed. |
| 11 | **params** Sets the request parameters on the action. |
| 12 | **prepare** This is typically used to do pre-processing work, such as setup database connections. |
| 13 | **profile** Allows simple profiling information to be logged for actions. |
| 14 | **scope** Stores and retrieves the action's state in the session or application scope. |
| 15 | **ServletConfig** Provides the action with access to various servlet-based information. |
| 16 | **timer** Provides simple profiling information in the form of how long the action takes to execute. |
| 17 | **token** Checks the action for a valid token to prevent duplicate formsubmission. |
| 18 | **validation** Provides validation support for actions |

Please loook into Struts 2 documentation for complete detail on above mentioned interceptors. But I will show you how to use an interceptor in general in your Struts application.

## How to use Interceptors?

Let us see how to use an already existing interceptor to our "Hello World" program. We will use the **timer** interceptor whose purpose is to measure how long it took to execute an action method. Same time I'm using **params** interceptor whose purpose is to send the request parameters to the action. You can try your example without using this interceptor and you will find that **name** property is not being set because parameter is not able to reach to the action.

## Create Custom Interceptors

Using custom interceptors in your application is an elegant way to provide cross-cutting application features. Creating a custom interceptor is easy; the interface that needs to be extended is the following **Interceptor** interface:

public interface Interceptor extends Serializable{

void destroy();

void init();

String intercept(ActionInvocation invocation)

throws Exception;

}

As the names suggest, the init() method provides a way to initialize the interceptor, and the destroy() method provides a facility for interceptor cleanup. Unlike actions, interceptors are reused across requests and need to be thread-safe, especially the intercept() method.

The **ActionInvocation** object provides access to the runtime environment. It allows access to the action itself and methods to invoke the action and determine whether the action has already been invoked.

If you have no need for initialization or cleanup code, the **AbstractInterceptor** class can be extended. This provides a default no-operation implementation of the init() and destroy() methods.

As mentioned previously, the **<results>** tag plays the role of a **view** in the Struts2 MVC framework. The action is responsible for executing the business logic. The next step after executing the business logic is to display the view using the **<results>** tag.

Often there is some navigation rules attached with the results. For example, if the action method is to authenticate a user, there are three possible outcomes. (a) Successful Login (b) Unsuccessful Login - Incorrect username or password (c) Account Locked.

In this scenario, the action method will be configured with the three possible outcome strings and three different views to render the outcome. We have already seen this in the previous examples.

But, Struts2 does not tie you up with using JSP as the view technology. Afterall the whole purpose of the MVC paradigm is to keep the layers separate and highly configurable. For example, for a Web2.0 client, you may want to return XML or JSON as the output. In this case, you could create a new result type for XML or JSON and achieve this.

Struts comes with a number of predefined **result types** and whatever we've already seen that was the default result type **dispatcher**, which is used to dispatch to JSP pages. Struts allow you to use other markup languages for the view technology to present the results and popular choices include **Velocity, Freemaker, XSLT** and **Tiles**.

## The dispatcher result type:

The **dispatcher** result type is the default type, and is used if no other result type is specified. It's used to forward to a servlet, JSP, HTML page, and so on, on the server. It uses the *RequestDispatcher.forward()* method.

We saw the "shorthand" version in our earlier examples, where we provided a JSP path as the body of the result tag.

We can also supply a **parse** parameter, which is true by default. The parse parameter determines whether or not the location parameter will be parsed for OGNL expressions.

## The FreeMaker result type:

In this example we are going to see how we can use **FreeMaker** as the view technology. Freemaker is a popular templating engine that is used to generate output using predefined templates. Let us create a Freemaker template file called **hello.fm** with the following contents:

Here above file is a template where **name** is a paramter which will be passed from outside using the defined action. You will keep this file in your CLASSPATH. Next, let us modify the **struts.xml** to specify the result as follows:

**Struts 2 Value Stack/OGNL**

## The Value Stack:

The value stack is a set of several objects which keeps the following objects in the provided order:

|  |  |
| --- | --- |
| **SN** | **Objects & Description** |
| 1 | **Temporary Objects** There are various temporary objects which are created during execution of a page. For example the current iteration value for a collection being looped over in a JSP tag. |
| 2 | **The Model Object** If you are using model objects in your struts application, the current model object is placed before the action on the value stack |
| 3 | **The Action Object** This will be the current action object which is being executed. |
| 4 | **Named Objects** These objects include #application, #session, #request, #attr and #parameters and refer to the corresponding servlet scopes |

The value stack can be accessed via the tags provided for JSP, Velocity or Freemarker. There are various tags which we will study in separate chapters, are used to get and set struts 2.0 value stack. You can get valueStack object inside your action as follows:

ActionContext.getContext().getValueStack()

Once you have a ValueStack object, you can use following methods to manipulate that object:

|  |  |
| --- | --- |
| **SN** | **ValueStack Methods & Description** |
| 1 | **Object findValue(String expr)** Find a value by evaluating the given expression against the stack in the default search order. |
| 2 | **CompoundRoot getRoot()** Get the CompoundRoot which holds the objects pushed onto the stack. |
| 3 | **Object peek()** Get the object on the top of the stack without changing the stack. |
| 4 | **Object pop()** Get the object on the top of the stack and remove it from the stack. |
| 5 | **void push(Object o)** Put this object onto the top of the stack. |
| 6 | **void set(String key, Object o)** Sets an object on the stack with the given key so it is retrievable by findValue(key,...) |
| 7 | **void setDefaultType(Class defaultType)** Sets the default type to convert to if no type is provided when getting a value. |
| 8 | **void setValue(String expr, Object value)** Attempts to set a property on a bean in the stack with the given expression using the default search order. |
| 9 | **int size()** Get the number of objects in the stack. |

## The OGNL:

**The Object-Graph Navigation Language** (OGNL) is a powerful expression language that is used to reference and manipulate data on the ValueStack. OGNL also helps in data transfer and type conversion.

The OGNL is very similar to the JSP Expression Language. OGNL is based on the idea of having a root or default object within the context. The properties of the default or root object can be referenced using the markup notation, which is the pound symbol.

As mentioned earlier, OGNL is based on a context and Struts builds an ActionContext map for use with OGNL. The ActionContext map consists of the following:

* **application** - application scoped variables
* **session** - session scoped variables
* **root / value stack** - all your action variables are stored here
* **request** - request scoped variables
* **parameters** - request parameters
* **atributes** - the attributes stored in page, request, session and application scope

It is important to understand that the Action object is always available in the value stack. So, therefore if your Action object has properties x and y there are readily available for you to use.

Objects in the ActionContext are referred using the pound symbol, however, the objects in the value stack can be directly referenced, for example if **employee** is a property of an action class then it can ge referenced as follows:

<s:property value="name"/>

instead of

<s:property value="#name"/>

If you have an attribute in session called "login" you can retrieve it as follows:

<s:property value="#session.login"/>

OGNL also supports dealing with collections - namely Map, List and Set. For example to display a dropdown list of colors, you could do:

<s:select name="color" list="{'red','yellow','green'}" />

The OGNL expression is clever to interpret the "red","yellow","green" as colours and build a list based on that.

The OGNL expressions will be used extensively in the next chapters when we will study different tags. So rather than looking at them in isolation, let us look at it using some examples in the Form Tags / Control Tags / Data Tags and Ajax Tags section.

## ValueStack/OGNL Example:

Actually, Struts 2 adds your action to the top of the valueStack when executed. So, the usual way to put stuff on the Value Stack is to add getters/setters for the values to your Action class and then use <s:property> tag to access the values. But I'm showing you how exactly ActionContext and ValueStack work in struts.

**\*\*\*\*\*\*\*\*\*\*\*\*\*Struts2 Validation\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Now we will look into how Struts's validation framework. At Struts's core, we have the validation framework that assists the application to run the rules to perform validation before the action method is executed.

Client side validation is usually achieved using Javascript. But one should not rely upon client side validation alone. Best practise suggests that the validation should be introduced at all levels of your application framework. Now let us look at two ways of adding validation to our Struts project.

Here we will take an example of **Employee** whose name and age would be captured using a simple page and we will put two validation to make sure that use always enters a name and age should be in between 28 and 65. So let us start with the main JSP page of the example.

The index.jsp makes use of Struts tag, which we have not covered yet but we will study them in tags related chapters. But for now, just assume that the s:textfield tag prints a input field, and the s:submit prints a submit button. We have used label property for each tag which creates label for each tag.

As shown in the above example, the validation method checks whether the 'Name' field has a value or not. If no value has been supplied, we add a field error for the 'Name' field with a custom error message. Secondly, we check if entered value for 'Age' field is in between 28 and 65 or not, if this condition does not meet we add an error above the validated field.

## How this validation works?

When the user presses the submit button, Struts 2 will automatically execute the validate method and if any of the if statements listed inside the method are true, Struts 2 will call its addFieldError method. If any errors have been added then Struts 2 will not proceed to call the execute method. Rather the Struts 2 framework will return **input** as the result of calling the action.

So when validation fails and Struts 2 returns **input**, the Struts 2 framework will redisplay the index.jsp file. Since we used Struts 2 form tags, Struts 2 will automatically add the error messages just above the form filed.

These error messages are the ones we specified in the addFieldError method call. The addFieldError method takes two arguments. The first is the **form** field name to which the error applies and the second is the error message to display above that form field.

addFieldError("name","The name is required");

To handle the return value of **input** we need to add the following result to our action node in **struts.xml**.

<result name="input">/index.jsp</result>

## XML Based Validation:

#### Two ways to use bundled validators

There are two ways to use bundled validators:

1. Plain-Validator (non-field validator) Syntax
2. Field-Validator Syntax

In the next page we will see the full example of bundled validators. Let's now understand the difference between plain-validator syntax and field-validator syntax.

The second method of doing validation is by placing an xml file next to the action class. Struts2 XML based validation provides more options of validation like email validation, integer range validation, form validation field, expression validation, regex validation, required validation, requiredstring validation, stringlength validation and etc.

The xml file needs to be named **'[action-class]'-validation.xml**. So, in our case we create a file called **Employee-validation.xml** with the following contents:

Above XML file would be kept in your CLASSPATH ideally along with class file. Let us have our Employee action class as follows without having **validate()** method:

Rest of the setup will remain as it is i the previous example, now if you will run the application, it will produce same result what we recieved in previous example.

The advantage of having an xml file to store the configuration allows the separation of the validation from the application code. You could get a developer to write the code and a business analyst to create the validation xml files. Another thing to note is the validator types that are available by default. There are plenty more validators that come by default with Struts. Common validators include Date Validator, Regex validator and String Length validator. Check following link for more detail

### Internationalization and Localization in struts2

**Internationalization (i18n)** is the process of planning and implementing products and services so that they can easily be adapted to specific local languages and cultures, a process called localization. The internationalization process is sometimes called translation or localization enablement. Internationalization is abbreviated **i18n** because the word starts with an i and ends with an n, and there are 18 characters between the first i and the last n.  
  
Will create a simple **employee form** screen, display the message from resource bundle via the Struts 2 UI components, and change the locale base on the selected language option.  
  
**Resource Bundles:**  
Struts2 uses resource bundles to provide multiple language and locale options to the users of the web application. You don't need to worry about writing pages in different languages. All you have to do is to create a resource bundle for each language that you want. The resource bundles will contain titles, messages, and other text in the language of your user. Resource bundles are the file that contains the key/value pairs for the default language of your application.  
  
When you reference a message element by its key, Struts framework searches for a corresponding message bundle in the following order:

* ActionClass.properties
* Interface.properties
* SuperClass.properties
* model.properties
* package.properties
* struts.properties
* global.properties

To develop your application in multiple languages, you would have to maintain multiple property files corresponding to those languages/locale and define all the content in terms of key/value pairs.   
  
For example if you are going to develop your application for **US English (Default), Hindi,Spanish, Chinese, German and Franch** the you would have to create three properties files. Here I will use global.properties file only, you can make use of different property files to segregate different type of messages.

* **global.properties:** By default English (United States) will be applied
* **global\_fr.properties:** This will be used for Franch locale.
* **global\_es.properties:** This will be used for Spanish locale.
* **global\_de.properties:** This will be used for German locale.
* **global\_hn.properties:** This will be used for Hindi locale.
* **global\_zh\_CN.properties:** This will be used for China locale.