Chapter

7

SECTION III: GETTING STARTED WITH STRUTS²

Working With Actions

Struts² has a single controller that handles all the user requests by invoking appropriate classes containing the required business logic. These classes are known as **Action classes**.

All the heavy lifting in the Web application is done by Actions. Actions interact with database and business rule engines, thus help transform the HTML into a rich, dynamic Web experience. After doing its job, an Action returns a control result string to indicate what the framework should do next.

Often, the next step is to render the result or display an error. In either case, the Action does not worry about generating the response. It only decides which logical result to present next.

Actions are simple Java objects. Actions are instantiated **one object per request**.

The **Struts² filter** based on the request URI decides which action to instantiate. After the action to be instantiated is chosen, an instance of that action is created and the execute() method is invoked.

Role Of Action

Actions Provide Encapsulation

One of the prime responsibilities of this role is to hold the business logic. Actions use the **execute()** method for this purpose. The code spec inside the **execute()** method should only hold the logic of the work associated with the request.

HINT

By convention, Actions are invoked by calling the execute() method. However, any other method that returns a String value can be used instead, simply by adding the appropriate configuration in the struts.xml file.

Code Spec:

```
package myApp;

public class simpleApp
{
   public String execute()
   {
       setMessage("Hi " + getName());
       return "SUCCESS";
   }
}
```

Explanation:

Here, the Action class is named **simpleApp**. The method that is invoked when this action is processed is the **execute()** method that encapsulates the business logic which in this case is a simple concatenation of two strings.

The action class does not need to**:**

- **Extend** another class
- **Implement** any interfaces

Struts2 Actions classes are simple objects very similar to a POJO [**P**lain **O**ld **J**ava **O**bject].

REMINDER

POJOs are ordinary Java objects which do not implement any interface or extend any other Java class and hence, does not depend on other APIs.

The importance of POJO as action is that there is no need to use extra objects in Struts2 framework. It is faster, simpler and easier to develop. It also shows how to organize and encapsulate the domain logic, access the database, manage transactions and handle the database concurrency.

The action class has one method named [by convention] execute. This method need not be named execute, it can be called anything as desired, **provided** that method returns a String. The only change needed would be in the configuration file [struts**.**xml]. If needed, the method **may** throw an **exception**.

The execute method does not accept any parameters, but it does returns a String object. Different return types can be used, by using the helper interfaces available in the Struts2 framework.

The helper interface provides common results such as SUCCESS, NONE, ERROR, INPUT and LOGIN.

These are string constants that can be utilized to return values to the framework that in turn help the framework decide the appropriate **View**.

From the MVC point of view, the **Action Class acts as a Model**. It executes particular business logic depending on the **Request** object and the **input parameters** it receives.

Actions Help Carry Data

Actions also carry the data around. The data is held local to the Action which makes it available during the execution of the business logic. The data can be set and retrieved using a bunch of **JavaBeans properties**. The execute() method references the data using these properties.

```
1  package myApp;
2
3  public class simpleApp
4  {
5      private String name;
6      public String getName()
7      {
8          return name;
9      }
10     public void setName(String name)
11     {
12         this.name = name;
13     }
14
15     private String message;
16     public String getMessage()
17     {
18         return message;
19     }
20     public void setMessage(String message)
21     {
22         this.message = message;
23     }
24
25     public String execute()
26     {
27         setMessage("Hi " + getName());
28         return "success";
29     }
30 }
```

JavaBeans properties
Getter/Setter methods

Explanation:

The above code spec justifies the following:

The data can be set and retrieved using a bunch of **JavaBeans properties**.

Here, for the data i.e. Name and the Message, the action class uses JavaBeans properties.

Data Entry Form And Action

In most standard Web applications, there are usually a set of data entry forms with a few form fields that allow data capture. Actions require such data for further processing as per the application's business logic, which is available in the request string or the Form data.

The **Struts²** framework follows the **JavaBean paradigm**. This means to access a form field's data, a **GETTER / SETTER** method is required.

The Struts² framework, automatically, moves the Request parameters from the **Form** to the JavaBeans properties that have matching names. In this case, the **name** parameter from the Form is automatically assigned to the **name** JavaBeans property in the Action class.

In **Struts²** providing access to the request string and form values is not very different. Here, each request string or form value is a simple **NAME-VALUE** pair. The **action class** should hold a **setter** method to assign the **VALUE** for a particular **NAME** and a **getter** method to retrieve the **VALUE** of a particular **NAME**.

In case of a call to a JSP page**:**
www.myserver.com/guestbook.action?**page**=1&**msg**=Hi

In this case, the Action would need the following SETTER methods**:**

- setPage(String page)
- setMsg(String msg)

Similarly when accessing such values in view mode, the following GETTER methods will be required**:**

- getPage()
- getMsg()

REMINDER

The setter does not always need to be a String value. Struts^2 is capable of converting a String to the required data type.

HINT

JavaBeans properties in the Action class also help expose the data received [from the Form] to the View/Result. For example, in the above code spec, using the **setMessage()** method, the message is assigned to the **msg** JavaBeans property. This, thus, exposes it to the View/Result.

Actions Return Control String

After the job is done, the action returns a control string. This string helps the **Struts² Filter** to decide the result/view that should be rendered.

Actions must return a string that map to one of the result components available for rendering the view for that action. These **mappings** are placed in the configuration file called struts**.**xml.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
        "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <package name="myApp" namespace="/" extends="struts-default"
        method="execute">
        <action name="simpleApp" class="com.book.myApp.simpleApp">
            <result name="success">/simpleView.jsp</result>
            <result name="error">/simpleError.jsp</result>
        </action>
    </package>
</struts>
```

Action Class

```
    public String execute()
    {
        setMessage("Hi " + getName());
        return "success";
    }
}
```

Explanation:

The value that is returned as the control string must match the name of the desired result in the configuration file i.e. struts.xml.

In the action class code spec, the action returns the string **success**. In the struts.xml file, **success** is the name of the one of the result components that point to a JSP page that will be rendered as the view.

The Helper Interfaces

Although, the action class does not need to:

- **Extend** another class
- **Implement** any interfaces

Sometimes it makes sense to extend helper classes or impplement interfaces provided by the Struts2 framework.

Struts2 provides two such helpers that can be used. The first being the **Action** interface which can be used to create action classes.

The Action Interface

The Action interface is an helper interface which exposes the execute() method to the action class implementing it.

Code Spec:

```
public interface Action
{
   public static final String SUCCESS = "success";
   public static final String NONE = "none";
   public static final String ERROR = "error";
   public static final String INPUT = "input";
   public static final String LOGIN = "login";

   public String execute() throws Exception;
}
```

This interface**:**

- Provides the common string based return values as CONSTANTS
- Enforces that implementing classes provide the default execute() method

The following table lists the CONSTANTS i**.**e**.** common results that can be returned by the execute() method**:**

Results	Description
SUCCESS	The action execution was successful.
NONE	The action execution was successful but do not show a view.
ERROR	The action execution was a failure.
INPUT	The action execution requires more input in order to succeed.
LOGIN	The action could not be executed as the user was not logged in.

These constants can conveniently be used as the control string values returned by the execute() method. The true benefit is that these constants are also used internally by the framework. This means that using these predefined control strings allows tapping into even more intelligent default behavior

The ActionSupport Class

The ActionSupport class is fairly simple. It adds a few useful utilities to the class that **extends** it.

The ActionSupport class implements the Action interface and some more useful interfaces.

Since ActionSupport implements the Action interface, static fields such as ERROR, INPUT, LOGIN, NONE and SUCCESS can be used in the class that extends it. There is already an implementation of the execute() method, inherited from Action, that simply returns Action**.**SUCCESS.

If a class implements the Action interface directly instead of extending ActionSupport, an implementation of the execute() method needs to be provided. Hence, it's more convenient to extend ActionSupport than to implement the Action interface.

In addition to the Action interface, ActionSupport also implements other interfaces**:**

- The **Validateable** and **ValidationAware** interfaces that provide programmatic, annotation-based and declarative XML-based validation
- The **TextProvider** and **LocaleProvide** interfaces that provide support for localization and internationalization
- **Serializable** interface used to create classes which enable the transfer of any binary object over a communication channel by transferring all the data of the object in a byte by byte manner

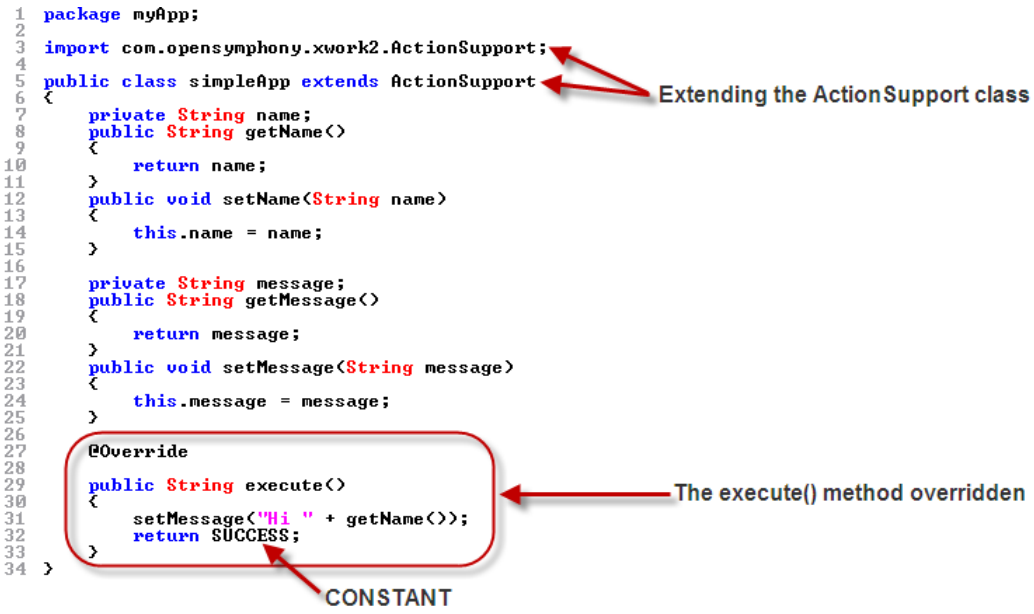
Code Spec:

```
public class ActionSupport
implements Action, Validateable, ValidationAware,
TextProvider, LocaleProvider, Serializable
{
   . . .
   public String execute() throws Exception
   {
       return SUCCESS;
   }
}
```

ActionSupport provides default implementations of several useful interfaces. If the actions extend this class, they automatically gain the use of these implementations.

Since the ActionSupport class provides default definitions of methods of all interfaces implemented by it, action class can be created by just **extending** this class and using its methods.

The simpleApp action class shown earlier, if it extends ActionSupport class looks like:




```
1  package myApp;
2
3  import com.opensymphony.xwork2.ActionSupport;
4
5  public class simpleApp extends ActionSupport
6  {
7      private String name;
8      public String getName()
9      {
10         return name;
11     }
12     public void setName(String name)
13     {
14         this.name = name;
15     }
16
17     private String message;
18     public String getMessage()
19     {
20         return message;
21     }
22     public void setMessage(String message)
23     {
24         this.message = message;
25     }
26
27     @Override
28
29     public String execute()
30     {
31         setMessage("Hi " + getName());
32         return SUCCESS;
33     }
34 }
```

Role Of The Struts² Filter

The Struts² filter:

- Instantiates the Action
- Executes the method that is specified in the configuration
- Reads the control string i.e. the return value and chooses the View/Result to present

The Struts² filter whilst handling an Action:

- Looks for a method called **execute**, if no method is specified in the configuration
- Generates the default view, if the return value is **success**

struts.xml

The **struts.xml** file holds the configuration information that is added/modified as actions are developed. This is the place where the Struts² filter looks for configurations.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
        "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <package name="myApp" namespace="/" extends="struts-default"
        method="execute">
        <action name="simpleApp" class="com.book.myApp.simpleApp">
            <result name="success">/simpleView.jsp</result>
            <result name="error">/simpleError.jsp</result>
        </action>
    </package>
</struts>
```

Being an XML file, the first element is the XML versioning and encoding information.

This is followed by the **D**ocument **T**ype **D**efinition **i.e.** DTD for the XML. The DTD provides structural information that the elements in the file should have and is used by XML parsers and editor.

<Struts>

This is the **outermost tag** that contains the Struts[2] specific configuration. All other tags are held within this tag.

<Package>

The struts.xml file is broken down into logical units called **packages**.

The package tag is used to group together configurations that share common attributes such as interceptor stacks or URL namespaces.

Packages are meant to help group the application's components based on commonality of function or domain.

Packages group the following into a logical configuration unit**:**

- Actions
- Result types
- Interceptors
- Interceptor stacks

REMINDER

Every action configured within a package inherits that package's configuration.

The Name Attribute [name="myApp"]

Indicates the name of the package.

The Namespace Attribute [namespace="/"]

The **namespace** attribute helps in separating different package into different namespace and hence, help in avoiding action mapping confliction.

The **namespace** attribute indicates the location where the action is placed. It is used to generate the URL namespace to which the actions of these packages are mapped.

In this case, the namespace specified is **/.** Hence to reach the **simpleApp** action, the Web browser will need to request **/simpleApp.action** within the Web application's context.

This means, if the web-app is called my-app and it is running on a server called www.myserver.com, the URL to access the action will be**:**
http://www.myserver.com/my-app/simpleApp.action

The Extends Attribute [extends="struts-default"]

The **Extends** attribute indicates the name of the parent package to inherit from. This attribute holds a package name whose components will be inherited by the current package that is being defined. This is very similar to the **extends** keyword in Java.

HINT

The **struts-default package** declares a huge set of commonly needed Struts[2] components ranging from complete interceptor stacks to all the common result types. These can be inherited by simply extending it.

Extending the struts-default package helps a developer avoid a lot of manual labor. This is because extending this package brings a lot of components along with it. One such component is the default Interceptor Stack.

REMINDER

The struts-default.xml [available in the distribution's main JAR file i.e. struts2-core.jar] holds the declarations of all the interceptors that the struts-default package brings in.

The Abstract Attribute [abstract="true"]

The **Abstract** attribute, if set to true, indicates that this package will only be used to define inheritable components, not actions.

In short, the Abstract attribute is used to create a base package that can omit the action configuration.

<Action>

The action maps an identifier to handle an action class. The action's name and framework use the mapping to determine how to process the request, when a request is matched.

The Name Attribute [name="simpleApp"]

The action's **name** attribute indicates the name of the action within the Web application.

The action's name is concatenated with the package's namespace to come up with the URL of the request:
http://www.myserver.com/my-app/simpleApp.action

The Class Attribute [class="com.book.myApp.simpleApp"]

The **class** attribute indicates which Java class will be instantiated for the Request.

The Method Attribute [method="execute"]

This is an optional attribute. This indicates the method to be invoked on a Request.

REMINDER

If this is un-specified, the filter assumes the **execute()** method.

<Result>

Each action element can have **one or more** result elements.

Each result is a possible view that the action can launch. The Result tag has name and type as its attributes.

The Name Attribute [name="success"]

This is an optional attribute, which indicates the result name.

REMINDER

If this is un-specified, the filter assumes **success** as the name.

The Type Attribute [type="dispatcher"]

This is an optional attribute, which indicates the kind of result.

REMINDER

If this is un-specified, the filter assumes dispatcher which forwards the Web browser to the View [JSP] specified.

<Include>

The **include** tag can be used to modularize a Struts2 application. This tag allows including other configuration files. It is always a child to the **struts** tag.

The File Attribute [file="guestbook-config.xml"]

This is the only attribute of the Include tag. It allows specifying the name of the file to be included. The file being included should have a structure identical to the **struts.xml** configuration file.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE struts PUBLIC
3          "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
4          "http://struts.apache.org/dtds/struts-2.0.dtd">
5  <struts>
6      <include file="guestbook-config.xml" />
7      <include file="bookMaster-config.xml" />
8      . . .
9  </struts>
```

Here, configurations for the guestbook as well as the bookMaster are defined using two different configuration files.

Getting Started With Actions

Now it's time to begin developing an Action. To understand Actions better, let's build a simple application [source code available on the Book's accompanying CDROM].

Application Requirements

Create a Guest Book application that provides an interface to accept visitor comments as shown in diagram 7.1. These comments should be viewable by other Web site visitors.

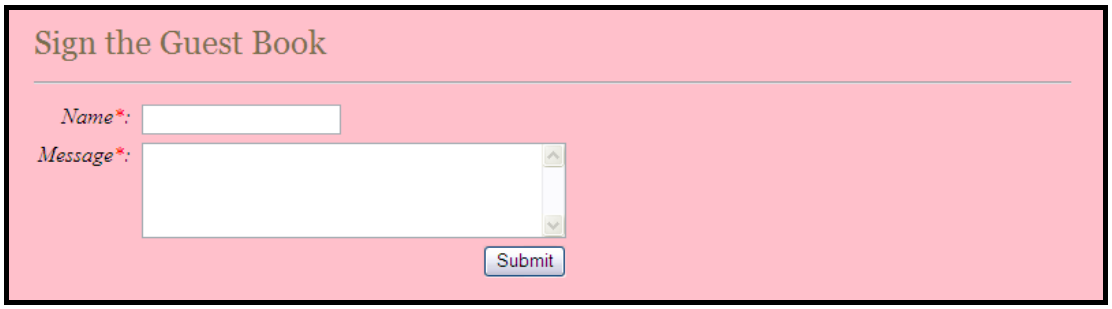


Diagram 7.1

Since this chapter focuses on the **Model** i.e. the Action class, let's build the action class to support the guestbook application. In the chapters that follow, the other components that make up the entire guestbook application will be built.

After all the components are in place, the visitors should be able to view the existing guestbook entries and from the view using a link add new entries to the guestbook as shown in diagram 7.2.

Diagram 7.2

After the visitor adds an entry in the guestbook, the entry will be added to the guestbook and the visitor will be taken back to the page where such entries can be viewed.

Based on the above requirements, the **Action class** will therefore be responsible for the following**:**

- Displaying existing entries that are available in the guestbook
- Accept new entries to the guestbook

Technically, the following two kinds of files will be created to achieve this**:**

File Type	File Name
A Bean class	GuestBook**.**java
An Action class	GuestBookAction **.**java

A Bean Class

To hold the captured data in a structured manner a bean class is required. This class should expose the following properties**:**

Property Name	To Store
guest	Visitor's Name
message	Message that the visitor enters
when	The date/time on which the message was entered

The class should have a parameterized constructor that allows setting captured values to these properties.

The primary purpose of having such a class is to hold individual guestbook entry as and when they are captured.

Open the GuestBook application in the NetBeans IDE that was created earlier in *Chapter 6: Getting Started*.

The following are the steps to create the Bean class using NetBeans IDE**:**

1. Right click **Sources Package** directory, select **New → Java Class…** as shown in diagram 7.3.1

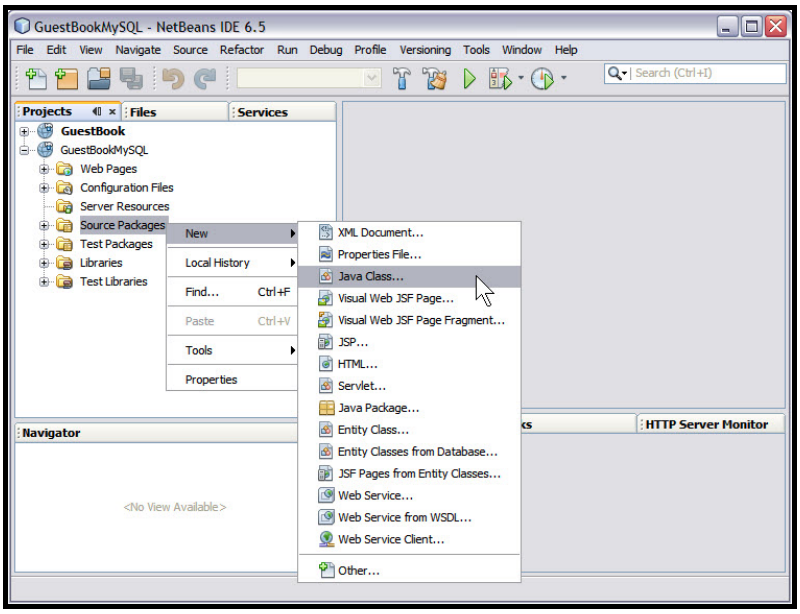


Diagram 9.3.1: Creating Java Bean Class

2. Enter **GuestBook** in the Class Name textbox and enter **myApp** in the Package textbox as shown in diagram 7.3.2

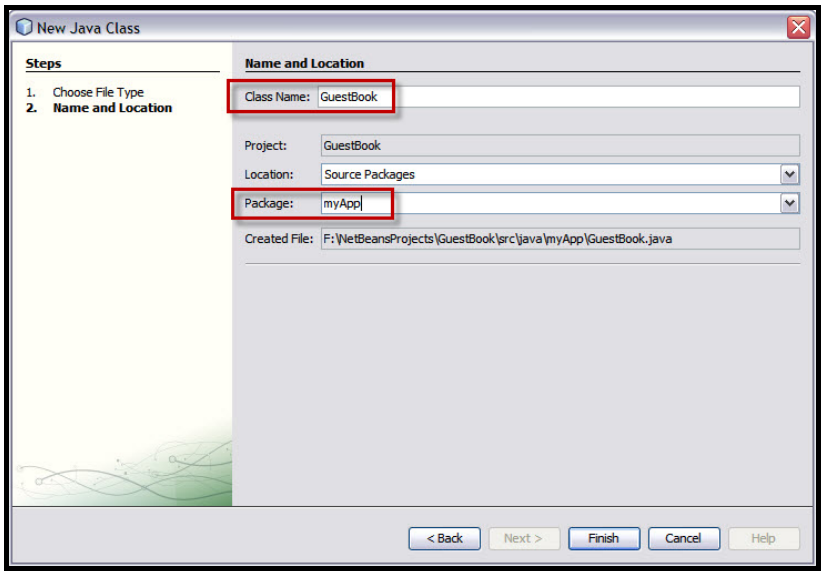


Diagram 7.3.2: Naming the Java class file

3. Click **Finish**

Now the bean class named GuestBook.java is created in the myApp package.

GuestBook.java

Edit the **GuestBook.java** file with the following contents.

```
 1  package myApp;
 2
 3  public class GuestBook
 4  {
 5      public GuestBook(String guest, String message, String when)
 6      {
 7          this.guest = guest;
 8          this.message = message;
 9          this.when = when;
10      }
11
12      public GuestBook()
13      {
14      }
15
16      private String guest;
17      private String message;
18      private String when;
19
20      public String getGuest()
21      {
22          return guest;
23      }
24
25      public void setGuest(String guest)
26      {
27          this.guest = guest;
28      }
29
30      public String getMessage()
31      {
32          return message;
33      }
34
35      public void setMessage(String message)
36      {
37          this.message = message;
38      }
39
40      public String getWhen()
41      {
42          return when;
43      }
44
45      public void setWhen(String when)
46      {
47          this.when = when;
48      }
49  }
```

Explanation:

A package named **myApp** is declared. This creates a directory named **myApp** under the <Web Application>\build\web\WEB-INF\classes\ and the **GuestBookAction.class** file is placed in the myApp directory when deployed.

The GuestBook class is a simple bean class that holds individual records. An object of this class will be created in the action class to hold an individual record and the object will then be added to the messages vector.

The Action Class

The visitor adds an entry using the data entry form.

The Action class will do the following**:**

1. **Spawn** an object of the bean class
2. Pass the captured values to its **parameterized constructor**
3. Store the data in a **Static Vector** by adding the populated object of the bean class in that vector

Using NetBeans create one more class called GuestBookAction using the same steps as shown earlier.

GuestBookAction.java

Edit the **GuestBookAction.java** file with the following contents.

```java
package myApp;

import java.util.Date;
import java.util.Vector;

import com.opensymphony.xwork2.ActionSupport;

public class GuestBookAction extends ActionSupport
{

    private String guest;
    private String message;
    private String when = new Date().toString();
    private static Vector messages = new Vector();

    @Override

    public String execute()
    {

        messages.add(new GuestBook(getGuest(),getMessage(),getWhen()));
        return SUCCESS;
    }

    public String getGuest()
    {
        return guest;
    }

    public void setGuest(String guest)
    {
        this.guest = guest;
    }

    public String getMessage()
    {
        return message;
    }

    public void setMessage(String message)
    {
        this.message = message;
    }

    public String getWhen()
    {
        return when;
    }

    public void setWhen(String when)
    {
        this.when = when;
    }

    public static Vector getMessages()
    {
        return messages;
    }

    public static void setMessages(Vector messages)
    {
        GuestBookAction.messages = messages;
    }
}
```

The static vector that holds the GuestBook entries

Spawning the object of the GuestBook class

Code spec:

```
package myApp;

import java.util.Date;
import java.util.Vector;
```

Explanation:

The following interfaces/classes are included using the import statement**:**

- java**.**util**.**Date**:** Is a wrapper for a date. This class allows manipulating dates in a system independent way
- java**.**util**.**Vector**:** Implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after Vector has been created

Code spec:

```
import com.opensymphony.xwork2.ActionSupport;
```

Explanation:

The ActionSupport class is a convenience class that provides default implementations of the Action interface and several other useful interfaces and helps add a few useful constants such as SUCCESS, INPUT, LOGIN and ERROR. These are string constants that can be utilized to return values to the framework to decide the view as**:**

```
return "success"
```

HINT

The framework does not make it mandatory to use this class, but it is a good idea to use it at least when learning the framework.

Code spec:

```
public class GuestBookAction extends ActionSupport
{
```

Explanation:

The **GuestBookAction** class extends the **ActionSupport** class.

Code spec:

```
private String guest;
private String message;
private Date when = new Date().toString();

private static Vector messages = new Vector();
```

Explanation:

To collect data from the JSP file [where the user enters data], private variables are declared.

Code spec:

```
@Override
```

Explanation:

The above code spec indicates that a method declaration is intended to override a method declaration in a superclass. If a method is annotated with '@' annotation type but does not override a superclass method, then compilers are required to generate an error message.

Code spec:

```
public String execute()
{
    messages.add(new GuestBook(getGuest(), getMessage(), getWhen()));
    return SUCCESS;
}
```

Explanation:

The execute() method is declared, which implements the logic of the action.

Inside the execute() method the name of the guest, its message and the date of capture is added to the Vector messages.

Actions must return a string that map to one of the result components available for rendering the view for that action.

The value that is returned as the control string must match the name of the desired result in the configuration file **i.e.** struts.xml.

In the action class code spec, the action returns the string **SUCCESS**. In the struts.xml file, **SUCCESS** is the name of the one of the result components.

Code spec:

```
public String getGuest()
{
   return guest;
}

public void setGuest(String guest)
{
   this.guest = guest;
}
```

```
    public String getMessage()
    {
        return message;
    }

    public void setMessage(String message)
    {
        this.message = message;
    }

    public Date getWhen()
    {
        return when;
    }

    public void setWhen(Date when)
    {
        this.when = when;
    }

    public static Vector getMessages()
    {
        return messages;
    }

    public static void setMessages(Vector messages)
    {
        GuestBookAction.messages = messages;
    }
}
```

Explanation:

Variables in a Java Bean normally have two methods associated with each variable i.e. a **get** method and a **set** method.

The **get method or getter** retrieves the value stored in the variable. The **set method or setter** sets the value for the variable. Both the set and the get methods are **public**.

When developing beans for processing form data, follow a common design pattern by matching the names of the bean properties with the names of the form input fields. Also the corresponding getter or setter method needs to be defined for each property within the bean.

For example, within the GuestBookAction [the GuestBookAction.java file], the property guest, the accessor methods getGuest() and setGuest() correspond to the form input element named guest**:**

1. **GuestBookAction.java:**

```
private String guest;
public String getGuest()
{
   return guest;
}
public void setGuest(String guest)
{
   this.guest = guest;
}
```

2. **GuestBook.jsp:**

```
<s:textfield required="true" key="Your Name" name="guest" />
```

This completes the Action component for the Guestbook application. As indicated earlier, the other components will be created in the chapters that follow.

Let's move on to the View layer and start exploring the rich options that the framework offers for rendering result pages. The next chapter heads towards the **Result** component of the framework and describes how data is pulled from the **Model** [using Struts^2 tag libraries] and rendered in the **View**.