Published on TheServerSide.com
November 4, 2002

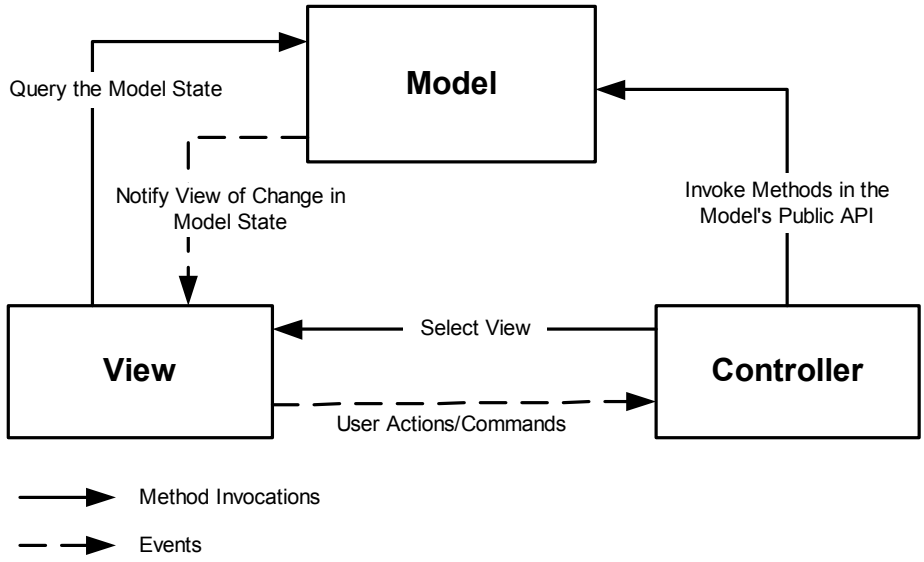
Introduction

The objective of this article is to introduce prospective Struts users to the key benefits of using Struts, and at the same time illustrate its configuration and usage semantics. We will define the requirements of a robust presentation framework and simultaneously discuss how these requirements are implemented in the Struts framework. We will also explore the design patterns implemented by Struts, the semantics of the controller, and the semantics of the associated helper components; this knowledge will be useful when designing components that will interact with the framework, and when there is a need for extending the framework for accommodating special needs of a project. This article complements the information available at *http://jakarta.apache.org/struts*.

MVC Architecture

The MVC (Model-View-Controller) architecture is a way of decomposing an application into three parts: the model, the view and the controller. It was originally applied in the graphical user interaction model of input, processing and output.

Adapted material from Chapter 4 of Practical J2EE Application Architecture by Nadir Gulzar, to be published by McGraw-Hill/Osborne Media(www.osborne.com) in March 2003.



Model A model represents an application's data and contains the logic for accessing and manipulating that data. Any data that is part of the persistent state of the application should reside in the model objects. The services that a model exposes must be generic enough to support a variety of clients. By glancing at the model's public method list, it should be easy to understand how to control the model's behavior. A model groups related data and operations for providing a specific service; these group of operations wrap and abstract the functionality of the business process being modeled. A model's interface exposes methods for accessing and updating the state of the model and for executing complex processes encapsulated inside the model. Model services are accessed by the controller for either querying or effecting a change in the model state. The model notifies the view when a state change occurs in the model.

View The view is responsible for rendering the state of the model. The presentation semantics are encapsulated within the view, therefore model data can be adapted for several different kinds of clients. The view modifies itself when a change in the model is communicated to the view. A view forwards user input to the controller.

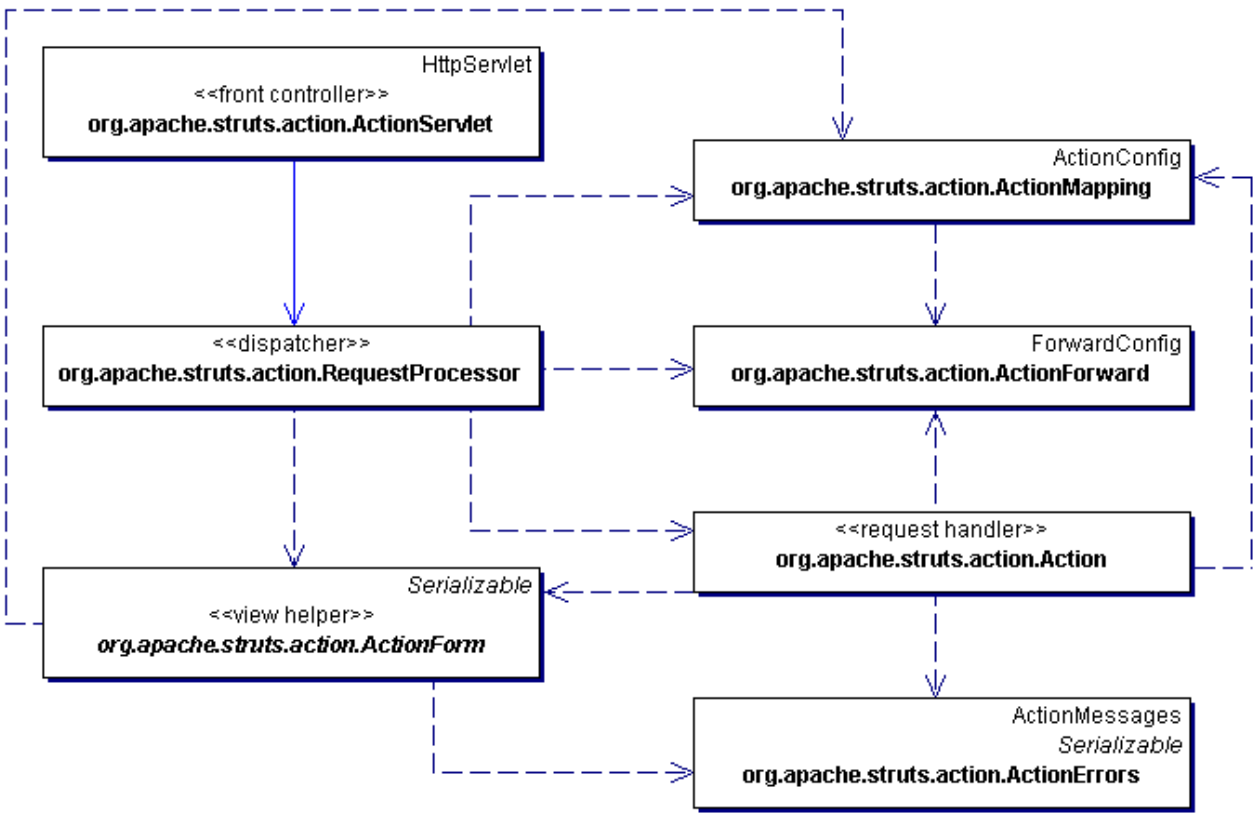
Controller The controller is responsible for intercepting and translating user input into actions to be performed by the model. The controller is responsible for selecting the next view based on user input and the outcome of model operations.

In a J2EE based application, MVC architecture is used for separating business layer functionality represented by JavaBeans or EJBs (the model) from the presentation layer functionality

represented by JSPs (the view) using an intermediate servlet based controller. However, a controller design must accommodate input from various types of clients including HTTP requests from web clients, WML from wireless clients, and XML-based documents from suppliers and business partners. For HTTP Request/Response paradigm, incoming HTTP requests are routed to a central controller, which in turn interprets and delegates the request to the appropriate request handlers. This is also referred to as MVC Type-II (Model 2) Architecture. Request handlers are hooks into the framework provided to the developers for implementing request specific logic that interacts with the model. Depending on the outcome of this interaction, the controller can decided the next view for generating the correct response.

Struts MVC Semantics

Let's begin by looking at the key Struts abstractions that is at the core of its MVC framework. Struts implements the MVC pattern using the *Service to Worker* pattern [Core].



Adapted material from Chapter 4 of Practical J2EE Application Architecture by Nadir Gulzar, to be published by McGraw-Hill/Osborne Media(www.osborne.com) in March 2003.

The Controller Object

The controller is implemented by the ActionServlet class. It provides a central place for handling all client requests. This promotes a cleaner division of labor for the controller layer that typically deals with view and navigation management, leaving the model access and manipulation to request handlers (Command objects [Gof]) that are typically request specific. All incoming requests are mapped to the central controller in the deployment descriptor as follows.

```
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
</servlet>
```

All request URIs with the pattern *.do are mapped to this servlet in the deployment descriptor as follows.

```
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

A request URI that matches this pattern will have the following form.

```
http://www.my_site_name.com/mycontext/actionName.do
```

The preceding mapping is called extension mapping, however, you can also specify path mapping where a pattern ends with /* as shown below.

```
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>/do/*</url-pattern>
</servlet-mapping>
```

A request URI that matches this pattern will have the following form.

```
http://www.my_site_name.com/mycontext/do/action_Name
```

The logical mapping of resources depicted above permits modification of resource mappings within the configuration file without the need to change any application code; this mapping scheme is also referred to as *Mulitplexed Resource Mapping*. The controller provides a centralized access point for all presentation-tier requests. The controller delegates each incoming request to the RequestProcessor

which in turn dispatches the request to the associated form bean for form validation, and to a request handler for accessing the model. The combination of controller and RequestProcessor forms the core controller process. The abstraction provided by the controller alleviates a developer from creating common application services like managing views, sessions, and form data; a developer leverages standardized mechanisms such as error and exception handling, navigation, internalization, data validation, data conversion etc.

In Struts 1.1, the Struts required configurations are loaded by the controller in its init() method. The configurations control the behavior of the framework; this includes mapping of URIs to request handlers using ActionMapping configuration objects, configuring message resources, providing access to external resources via plugins etc. In fact, processing of incoming requests actually occur in the RequestProcessor to which ActionServlet delegates all the input requests.

The Dispatcher Object

The RequestProcessor functions as a dispatcher and handles client requests by instantiating (or reusing) a request handler, and a corresponding form bean. The errors created, or exceptions thrown by the form beans and the request handlers are processed by the RequestProcessor, which influences the view management function of the RequestProcessor. Form beans assist RequestProcessor for storing the form data and/or staging intermediate model data required by the view. The RequestProcessor uses the `<action>` declarations in the struts-config.xml file, as shown below, for instantiating request specific request handlers.

```
<action-mappings>
    <action  path="/editCustomerProfile"
             type="packageName.EditCustomerProfileAction"
             name="customerProfileForm"
             scope="request"/>
</action-mappings>
<form-bean  name="customerProfileForm"
            type="packageName.customerProfileForm"/>
```

All incoming requests are delegated by the controller to the dispatcher, which is the RequestProcessor object. The RequestProcessor examines the request URI for an action identifier, creates a request handler instance using the information in the ActionMapping configuration object (explained in the next section), and calls the *requesthandler*.execute(…) method. The execute(…)

method of the request handler is responsible for interacting with the application model. Depending on the outcome, the request handler will return an ActionForward configuration object (ActionForward is the runtime representation of `<forward>` element and is explained in section *Navigation using ActionForward*) to the RequestProcessor. The RequestProcessor will use the ActionForward object for invoking the next view by calling either the RequestDispatcher.forward(…) or response.sendRedirect(…).

Command Pattern using ActionMapping

Struts provides a declarative way of specify the mapping between the servlet path in the request URI and an appropriate request handler using XML syntax. This implementation is very similar to the command pattern [Gof]. The following snippet is from struts-config.xml file; these declarations are used for creating an ActionMapping configuration object, which is the runtime representation of the `<action>` element.

```
<action-mappings>
    <action  path="/editCustomerProfile"
             type="packageName.EditCustomerProfileAction"
             name="customerProfileForm"
             scope="request"/>
</action-mappings>
```

The following briefly explains the attributes used in the preceding declaration.

path The context relative path in the HTTP request that is used for identifying this action mapping.

type Class name that will be used for creating an instance of the request handler for handling this request.

name The logical name of a JavaBean, also called a form bean, that will be used to hold form data. The form bean will be saved in the specified scope using this name.

scope Request or session scope for saving the form bean.

The path attribute shown in the preceding snippet maps to the `action` attribute of the HTML <form> element. The declarative specifications shown above prevents hard coding of mappings in the code base and enables convenient visualization of how servlet path specifications in HTML forms are mapped to instances of request handlers; in addition, application behavior and navigation semantics

can be changed by simply altering the action mappings. A request handler is a subclass of the Struts provided Action class..

For an HTML <form> tag, dynamic URL generation for the `action` attribute using the custom org.apache.struts.taglib.html.FormTag will protect the HTML documents from being adversely impacted as a result of change of context path or `<url-pattern>`. For a **.do* url pattern, the following custom FormTag
<html:form action="/editCustomerProfile?customerType=preferred"> will dynamically generate an HTML `<form>` tag with the action attribute containing the following server-relative URL:
<form action="/*contextPath*/editCustomerProfile.do?customerType=preferred"/>

Using the `name` attribute, an action mapping can declaratively specify a JavaBean whose properties will hold the parameters from the HTTP request; this JavaBean is subclassed from the ActionForm class. The `name` in the action mapping declaration is a unique identifier using which the instances of ActionForm classes are stored in the specified scope. The ActionForm subclass is declared in the struts-config.xml file using the <form-beans> tag as follows.

```
<form-bean  name="customerProfileForm"
            type="packageName.customerProfileForm"/>
```

Refer to section *Capturing Form Data* for more information on `<form-bean>` element and ActionForm class.

Model Interaction with Request Handlers

A subclass of Action is used as an adaptor between incoming requests and the model. The Action subclass, also called the request handler, is created specific to every request. An action is interpreted initially by the RequestProcessor, which in turn instantiates a corresponding request handler. This Action class derived object is created specific to every request as explained in the preceding section *The Dispatcher Object*. The request handler implements the *Command* pattern [Gof]. A client request encapsulates the desired action in the request URI as servlet path, the path information is subsequently extracted by the dispatcher (RequestProcessor) for creating an instance of the corresponding request handler. The command pattern decouples the UI from request handlers.

The base Action class provides common functions for accessing framework related resources and methods for saving errors detected by the execute(…) method of its subclass. The errors are

subsequently extracted and displayed in the HTML form using the custom org.apache.struts.taglib.html.ErrorsTag as explained in the section *Displaying Errors with ErrorsTag*. The execute(…) method of a request handler should contain control flow for dealing with request parameters and the associated ActionForm, it should encapsulate model interaction semantics, and it should provide the next view based on the outcome of model operations. Request handlers are cached by the RequestProcessor when first created, and subsequently made available to other incoming requests; as such, request handlers must not contain user specific state information; also, request handlers must synchronize access to resources that require serialized access.

For distributed applications, an action class houses the control logic required for interacting with business logic in EJB components and will typically use a *Business Delegate*[Core] object for this purpose. Business delegate shields the request handlers from having to deal with the complexity of accessing distributed components. The business delegate design pattern promotes loose coupling between the request handlers and the server side components since the logic for accessing server side components is embedded in the business delegate. A request handler is written by a developer working in the presentation tier, while a business delegate is usually written by a developer responsible for creating the business tier services. For smaller non-distributed applications, the action class may contain business logic. When distributed processing in not required and business logic is embedded in request handlers, a *Data Access Object* [Core] can be used to abstract the underlying data access implementation; this provides a loose coupling between the request handlers and the data access layer, thus protecting the presentation tier from implementation changes in the integration tier. The base Action class of request handlers provides several convenience methods, please refer to the API documentation at *http://jakarta.apache.org/struts/api/index.html*.

Navigation using ActionForward

ActionForward objects are configuration objects. These configuration objects have a unique identifier to enable their lookup based on meaningful names like ‘success’, ‘failure’ etc. ActionForward objects encapsulate the forwarding URL path and are used by request handlers for identifying the target view. ActionForward objects are created from the `<forward>` elements in struts-config.xml. Following is an example of a `<forward>` element in Struts that is in the local scope of an `<action>` element.

```
<action  path="/editCustomerProfile"
         type="packageName.EditCustomerProfileAction"
         name="customerProfileForm"
         scope="request">
    <forward name="success"  path="/MainMenu.jsp"/>
    <forward name="failure"  path="/CustomerService.jsp"/>
</action>
```

Global <forward> elements are typically specified for common destinations within the application as illustrated by the following example.

```
<global-forwards>
    <forward   name="success"  path="/MainMenu.jsp"/>
    <forward name="failure"  path="/CustomerService.jsp"/>
</global-forwards>
```

Based on the outcome of processing in the request handler's execute(…) method, the next view can be selected by a developer in the execute(…) method by using the convenience method org.apache.struts.action.ActionMapping.findForward(…) while passing a value that matches the value specified in the name attribute of <forward> element. The ActionMapping.findForward(…) method will provide an ActionForward object either from its local scope, or from the global scope, and the ActionForward object is returned to the RequestProcessor for invoking the next view using RequestDispatcher.forward(…) method or response.sendRedirect(…). RequestDispatcher.forward(…) method is called when the <forward> element has an attribute of redirect="false" or the redirect attribute is absent; redirect="true" will invoke the sendRedirect(…) method. Following snippet illustrates the redirect attribute usage.

```
<forward name="success"  path="/Catalog.jsp"  redirect="true"/>
```

The <controller> element in the struts-config.xml file provides yet another feature for controlling how <forward> element's name attribute is interpreted; the <controller> element is used in conjunction with the input attribute on the <action> element as shown below.

```
<action  path="/editCustomerProfile"
         type="packageName.EditCustomerProfileAction"
         name="customerProfileForm"
         scope="request"
         input="profile">
```

```
    <forward name="profile"  path="/CustomerProfile.jsp"/>
    <forward name="success"  path="/MainMenu.jsp"/>
</action>
<controller>
    <set-property property="inputForward" value="true"/>
</controller>
```

The preceding `<action>` element has an `input` attribute with a forward name; this forward name is identical to the one used in the `<forward>` element. With the preceding `<controller>` configuration, when the ActionForm.validate(…) returns a non-empty or non-null ActionErrors object, the RequestProcessor will select the `<forward>` element whose `name` attribute has the same value as the `input` attribute of the `<action>` element; unless overridden by a subclass of RequestProcessor, this behavior is standard when validation errors are encountered. With the following `<controller>` element declaration, when the ActionForm.validate(…) returns a non-empty or non-null ActionErrors object, the `input` attribute provides a forwarding URL instead of a ActionForward name to which the forward occurs. In the absence of `inputForward` property, this is the default configuration.

```
<controller>
    <set-property property="inputForward" value="false"/>
</controller>
```

The forward is done to the specified path, with a '/' prepended if not already included in the `path` specification. For forward or redirect, URLs in Struts are created internally by the RequestProcessor with the following structure.

- If redirect=true, URL is created as /*contextPath*/*path* because for HttpServletResponse.sendRedirect(…) the container interprets a URL with leading '/' as relative to the servlet container root.
- If redirect=false, URI is created as /*path* because ServletContext.getRequestDisptacher(…) uses context relative URL.

Internationalization and Localization Support

Internationalization or I18N is the process of engineering an application such that it can be adapted to various languages and regions without requiring any change to the application logic. For internationalization support, an application must consider the following.

- Textual content, error messages, exception messages, and labels on GUI components must be externalized into resource files. These resource files will contain Locale specific information as discussed below.
- Date, time, currency, numbers, measurements and phone numbers must be formatted based on local preferences and culture.

In today's global market place it is important to design the applications with internationalization; doing this upfront takes relatively less time and effort than incorporating I18N after the application has been developed. The JDK provides the Locale class that is used by internationalized classes to behave in a locale-sensitive way. A Locale object represents a specific geographical, political, or cultural region. Following is a discussion on how Struts implements I18N and localization.

The Locale Object

Struts classes providing I18N support retrieve the locale specific information from the HttpSession using getAttribute(Action.LOCALE_KEY). The Locale object is saved in the session in several different ways as explained below.

Using HtmlTag

The custom tag org.apache.struts.taglib.html.HtmlTag is inserted in a JSP as `<html:html locale="true">`. This is a declarative way of populating locale in the session. When locale=true is specified, the tag logic will retrieve the Locale using the HttpServletRequest.getLocale() method. The getLocale() method returns the preferred Locale that a client browser will accept content in based on the Accept-Language header. A default locale for the server is returned when the client does not provide an Accept-Language header. A session object is created if it does not exist and the Locale object is then stored in the session object using Action.LOCALE_KEY. The HTML tag is subsequently written to the output stream with the `lang` attribute set to the language specified in the

locale. The locale object is stored only once in this manner; subsequent locale=true specification will not be able to replace the locale object in the session. This method of setting locale works best when the users have their browser set with the preferred locale list.

Using the Action object

For programmatically changing the locale object, the Action class provides the setLocale(…) method for saving the locale object in the session using Action.LOCALE_KEY. This method of setting locale works best when a user has the option of choosing locale in the HTML form by clicking a UI component. However, using this method can sometimes cause problems if locale specific resources are preloaded and a user is allowed to switch locale in the middle of a process flow. It is best to allow this functionality in a controlled manner and reset all locale specific resources when a locale change is requested.

Using <controller> Element

Under this scheme, the `<controller>` tag from the *struts-config.xml* file is used to flag the RequestProcessor to get the locale from the HttpServletRequest object and put it in the session using Action.LOCALE_KEY. This is illustrated below.

```
<controller>
    <set-property property="locale" value="true"/>
</controller>
```

If `value=true`, then the Locale object obtained from request. getLocale() is saved in the session if not previously saved.

Internationalized Messaging and Labeling

For I18N support, all error messages, instructional messages, informational messages, titles, labels for GUI components, and labels for input fields must be stored externally and accessed in a locale-specific way. The Struts framework provides the MessageResources class that mimics the ResourceBundle class provided by the JDK. Locale-specific resource bundles provide a way of isolating locale-specific information. Resource bundles belong to families whose members share a common base name, but whose names also have additional components that identify their locales. The default resource bundle has the same name as the base name of a family of resource bundles and

is the bundle of last resort when locale-specific bundles are not found. Locale specific bundles extend the base bundle name with locale specific identifiers like the language, country and variant of a locale. Consider the following example.

If base ResourceBundle name is MyApplicationResources then resource bundles belonging to this family may be identified as follows.

- MyApplicationResources_en identifies the bundle for the English language.
- MyApplicationResources_fr identifies the bundle for the French language.
- MyApplicationResources_fr_FR identifies the bundle for the French language for France.
- MyApplicationResources_fr_CA identifies the bundle for the French language for Canada.

Say the desired locale is fr_FR and the default locale is en_US, The search order for accessing resource bundles can be summarized as follows. The search goes from being more specific to less specific.

1. MyApplicationResources_fr_FR – the desired resource bundle
2. MyApplicationResources_fr – less specific bundle if the desired bundle not found
3. MyApplicationResources_en_US – the default bundle if no matching bundles found this far
4. MyApplicationResources_en – less specific bundle if the default bundle not found
5. MyApplicationResources – the base bundle

Struts provides a similar facility for accomplishing the above mechanism using MessageResources objects. MessageResources objects are initialized from the key/value pairs specified in underlying properties files. You have to specify only the base name for MessageResources properties file in the struts-config.xml file to access all the locale-specific properties files using the same search order as specified for the ResourceBundle(s). Following depicts how message resources are declared in the struts-config.xml file.

```
<message-resources parameter="packageName.MyApplicationResources"
key="MyResources"/>
```

The value of the `parameter` attribute declares the base non-locale specific properties file. This base resource file will have the name MyApplicationResources.properties, while locale-specific files will have the name MyApplicationResoures_*localeSpecificExtension*.properties. For each

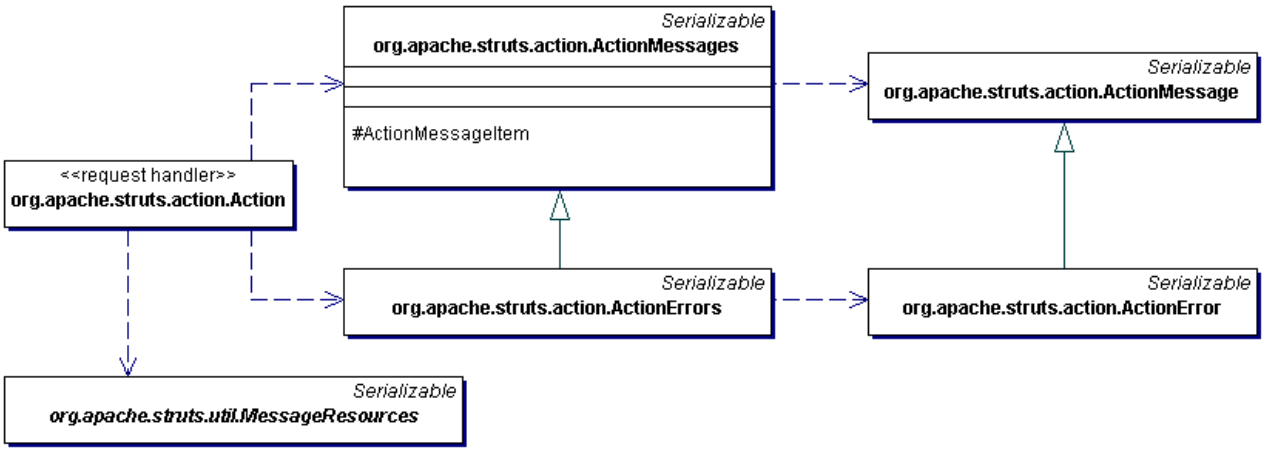
application, we can specify one or more base bundle names. MessageResources objects are created by the controller i.e. ActionServlet and saved in the ServletContext using either a generic key Globals.MESSAGES_KEY (same as Action.MESSAGES_KEY) or in case of multiple MessageResources, each MessageResources object is saved using the `key` attribute provided in the `<message-resources>` element.

For accessing message resources objects in request handlers, the Action class provides a convenience method Action.getResources(…) for retrieving a message resource from the ServletContext using the key (i.e. unique identifier) associated with the MessageResources object.. Each MessageResources object will be responsible for getting locale-specific messages by accessing the underlying set of locale-specific properties files; the properties files are identified by the base MessageResources name specified by the `parameter` attribute in the `<message-resources>` tag.

To retrieve a locale-specific message, use MessageResources.getMessage(…) while passing locale and message key as arguments. The locale can be retrieved from the session using Action.LOCALE_KEY. When an Object[] is provided as an argument for MessageResources.getMessage(…), the message retrieved is treated as a message format pattern and is converted to a MessageFormat object. The MessageFormat object is subsequently used for calling MessageFormat.format(…) method while passing the object[] to be appropriately formatted and inserted into the pattern at appropriate places. The MessageFormat class is not locale specific, therefore the corresponding message format pattern and the Object[] must take localization into account. MessageResources API provides several convenience methods for retrieving messages; the corresponding Javadoc is available at *http://jakarta.apache.org/struts/api/index.html*.

Error Handling

Most form interactions require that the user be informed of the possible outcome of the form submission. Displaying error and informational messages in a consistent manner is a desirable feature of a framework. In the preceding section *Internationalized Messaging and Labeling*, we discussed Locale-specific messaging using the MessageResources objects. The set of properties files associated with each MessageResources object has key/value pairs. A Struts based application will accumulate, for message lookup, the keys associated with validation and informational messages in an ActionErrors object as a precursor to accessing resource bundles. Following static model illustrates the classes involved in the error handling mechanism provided by Struts.



We will briefly discuss the interactions depicted in the preceding illustration. This discussion will provide us with insight on how message keys are captured in Struts to get locale-specific messages, and how the messages are rendered in a consistent manner in the view. For this discussion the view component is a JSP.

Identifying Errors with ActionError

Implementations of Action.execute(…) or ActionForm.validate(…) should capture validation or application specific errors in ActionErrors objects, which aggregates ActionError objects. An ActionError object consists of a message key and optionally an object[] to be used for parametric replacement in the retrieved message. Refer to the preceding section *Internationalized Messaging and Labeling* for relevant information. ActionError objects must be created without worrying about the locale or the associated resource bundles. We will deal with I18N when the ActionError objects are used for retrieving messages. Refer to the ActionError API for a complete list of available convenience methods for creating ActionError objects. Once an ActionError object is created, it should be added to the ActionErrors object using ActionErrors.add(…) method while passing as argument the ActionError and the property name for which a validation error was detected. For saving error messages not related with a property, a convenience instance member ActionErrors.GLOBAL_ERROR is available for use in place of property argument in ActionErrors.add(…). Usage of property name in creating ActionErrors object is clarified in the section *ActionErrors*. The ActionError class extends the ActionMessage class; as of Struts 1.1, the ActionError class simply delegates the calls to ActionMessage class.

Compiling Errors with ActionErrors

ActionErrors object hold all ActionError objects in a HashMap whose key is the name of the property for which messages have been accumulated, and the value is an ActionMessageItem object. ActionMessageItem is declared as inner class of ActionMessages. Each ActionMessageItem object consists of a unique sequence number and an ArrayList object representing all possible validation errors for a given property. The sequence number is used for sorting the ActionMessageItem collection such that validation errors are reported according to the property that was first flagged as invalid. ActionErrors.get() returns an Iterator on an ArrayList containing ActionError objects. This Iterator object is referenced by the custom tag ErrorsTag and will be discussed in the next section *ErrorsTag*.

In request handlers, i.e. in Action.execute(…), ActionErrors should be saved in the HttpServletRequest using the attribute name Action.ERROR_KEY; this is done by calling a convenience method saveErrors(…) on the base Action class while passing as arguments the request object and the ActionErrors object The ActionErrors generated as a result of ActionForm.validate(…) are saved by RequestProcessor in the request object using Action.ERROR_KEY. The next view can use the ErrorsTag for retrieving the ActionErrors object; the ErrorsTag can be used in a JSP as follows.

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<html:errors/>
```

The ActionErrors class extends the ActionMessages class. ActionErrors provides the static member GLOBAL_ERROR and ActionMessages provides the static member GLOBAL_MESSAGE; these static members can be used as keys when the messages are not property specific. For saving ActionMessages object in a request handler, the convenience method Action.saveMessages(…) can be used while passing the request object and the ActionMessages object; the ActionMessages object is saved in the request using the Action.MESSAGE_KEY.

For simply capturing message keys, without the property name and the substitution parameters, a convenience method org.apache.struts.util.RequestUtils.getActionErrors(…) is available for converting a String object, a String array, or an ErrorMessages object (a Vector of message keys) into an ActionErrors object. For these implementations, the getActionErrors(…) method will use the ActionErrors.GLOBAL_ERROR in place of property argument.

Displaying Errors with ErrorsTag

This custom tag renders the messages in an HTML document. It retrieves the ActionErrors from HttpServletRequest using Action.ERROR_KEY and then using the ActionErrors.get() method retrieves an Iterator on an ArrayList containing ActionError objects. For each ActionError object in the ArrayList, a locale-specific message is retrieved and send to the response stream. By default, the locale object in the session is used; but an alternate `locale` attribute can be specified for the tag. By default, the resource bundle saved in the context with the key Action.MESSAGES_KEY will be used unless overridden by the `bundle` attribute on the tag. You will need to override the resource bundle if more than one base resource files are being used for manageability. As of Struts 1.1, an ErrorsTag can only use one resource bundle family (i.e. the bundles have the same base name), therefore all errors in the ActionErrors object must be available in this resource bundle family. Because all ActionError objects within the ActionErrors object are logged by a property name, the messages can be restricted to a single property by specifying a `property` attribute specification on the ErrorsTag. The ErrorsTag uses message keys 'errors.header' and 'errors.footer' for providing caption and formatting around error messages as shown below.

```
errors.header=<h3><font color="red">Please review following message(s)
before proceeding:</font></h3><ul>
errors.footer=</ul>
```

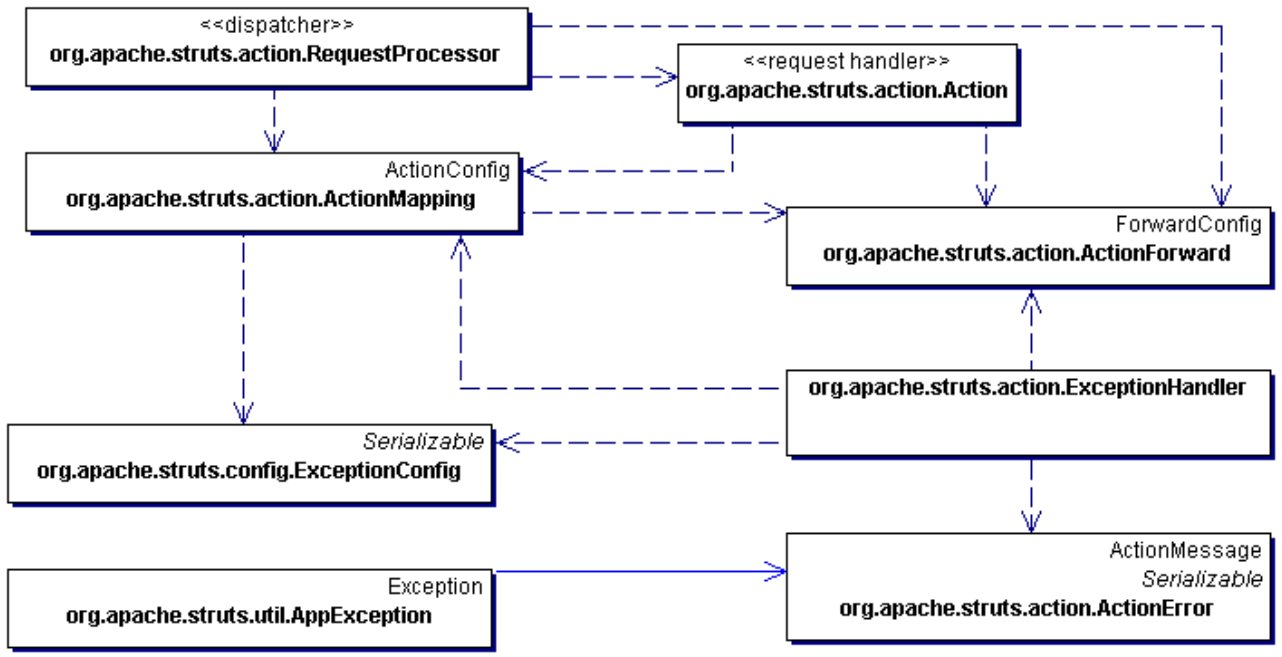
Exception Handling

In addition to error handling mechanism, a presentation framework must provide a mechanism for showing locale-specific exceptions of meaning and relevance to the user. A recommended way to do this is to capture the actual exception and its context in a log file and then send a meaningful informational message for assisting the user in determining a suitable course of action. Uncaught exceptions in JSPs are handled the by the `errorPage` mechanism as specified in JSP 1.2 specification. Similarly, uncaught exceptions in servlets are handled using the `<error-page>` specification in the web.xml deployment descriptor. Struts provides a simple mechanism that is somewhat similar to the error page mechanism provided by JSP and servlet containers. This is illustrated by the following declaration in the struts-config.xml file.

```
<action  path="/editCustomerProfile"
         type="packageName.EditCustomerProfileAction"
```

```
        name="customerProfileForm"
        scope="request"
        input="profile">
    <forward name="profile"  path="/CustomerProfile.jsp"/>
    <forward name="success"  path="/MainMenu.jsp"/>
    <exception
        key="profile.inaccessible"
        type=" packageName.ProfileAccessException"
        path="/logon.jsp"/>
</action>
```

Exception handling mechanism builds on top of error handling mechanism and therefore uses the MessageResources for providing locale-specific messages. Following static model illustrates the classes involved in the exception handling mechanism provided by Struts.



Role of the Dispatcher

As discussed in the section *RequestProcessor*, the request processor calls the execute(…) method of the request handler. Any exception thrown by the request handler is caught by the RequestProcessor and interrogated for a possible match with the <exception> elements in the struts-config.xml file. The RequestProcessor will call the ActionMapping.findException(…) method to find

an ExceptionConfig configuration object (ExceptionConfig objects are runtime representations of `<exception>` elements) whose `type` attribute matches the type of the exception. If an attempt to find an `<exception>` configuration for the original exception fails then the findException(…) method will lookup the exception superclass chain for a suitable match until it reaches the top of the chain. ActionMapping.findException(…) will search for the `<exception>` element both in the local scope of the ActionMapping object, as well as in the global scope.

Global `<exception>` elements are typically specified for common exceptions within the application as illustrated by the following example.

```
<global-exceptions>
    <exception
         key="profile.inaccessible"
         type=" packageName.ProfileAccessException"
         path="/logon.jsp"/>
</global-exceptions>
```

If an ExceptionConfig object is found for a given exception type then the RequestProcessor will create an exception handler and call its execute(…) method; this is explained further in section *Converting an Exception into ActionErrors*. The RequestProcessor will forward to the URL specified in the ActionForward object returned by the exception handler.

Exception Handling with AppException

This is a convenience base class for creating exceptions within the request handlers. It encapsulates both, the attribute causing an exception (optional) , and associated ActionError object. A subclass of AppException will be responsible for providing the appropriate constructors for correctly instantiating this object by using a message key, and optionally the attribute name and an object[] for parametric substitution. The message key can be extracted from the ExceptionConfig object that corresponds to this exception. Refer to AppException API for available list of constructors that can be called from the constructors of its subclass. The AppException is passed as an argument in the ExceptionHandler.execute(…) method.

Converting an Exception into ActionErrors

The RequestProcessor queries the ExceptionConfig for an exception handler specification. The RequestProcessor creates the specified ExceptionHandler and calls its execute(…) method while passing the AppException as one of the arguments. A default exception handler specification of org.apache.struts.action.ExceptionHandler is pre-configured in the ExceptionConfig object. The ExceptionHandler retrieves the ActionError from the AppException object and creates an ActionErrors object for consumption by ErrorsTag. If the exception is not of type AppException or one of its derived classes then the ExceptionHandler will create the ActionErrors object using the key specified in the <exception> element; this alleviates the request handler developer from writing extra code for exception handling; however this limits the ability of the framework to call only a single constructor of ActionError that only accepts a key value. Use the `handler` attribute on the <exception> element to override the default exception handler if desired. The ExceptionHandler or a subclass of ExceptionHandler will create an ActionForward object using the `path` property of the ExceptionConfig; if this path is not specified then it will use the path specified in the `input` attribute of the ActionMapping configuration object. The ExceptionHandler will also save the original exception in the request object using Action.EXCEPTION_KEY. A view is free to access this information in any way desired. The Action.EXCEPTION_KEY can be also be used to retrieve an re-throw the original exception for using the error-page mechanism provided by the servlet container.

Once-Only Form Submission

A problem always encountered in developing browser based clients is the possibility of a form getting submitted more than once. It is apparent that such submissions are undesirable in any eCommerce application. Struts provides a mechanism to protect the model layer from the adverse effect of multiple form submissions by using a token generated by the base Action class generateToken(…) method. To control transactional integrity and atomicity, simply call the saveToken(…) in a request handler before selecting the next view with an ActionForward. The saveToken(…) method calls the generateToken(…) method to create a unique identifier and then saves it in the session with the key Action.TRANSACTION_TOKEN_KEY. The FormTag retrieves the token from the session and saves it as a hidden field with the name Constants.TOKEN_KEY.

On a subsequent request, the request handler can check for token validity by calling the convenience method isTokenValid(…) on the base Action class. Should this method return false then

the request handler must implement suitable logic to account for the problem. An example of this is illustrated below.

```
if (!isTokenValid(request)) {
    errors.add(ActionErrors.GLOBAL_ERROR,
    new ActionError("transaction.token.invalid"));
//errors object is of type ActionErrors
}
```

The isTokenValid(…) method synchronizes the session object to prevent multiple requests from accessing the token. In the request handlers, method isTokenValid(…) must be followed by a resetToken(…) to remove the token from the session; this will ensure that any subsequent request will result in isTokenValid(…) returning false, thus preventing a form from multiple submissions. The saveToken(…) should be called in the request handler to recreate a new transaction token for the next request.

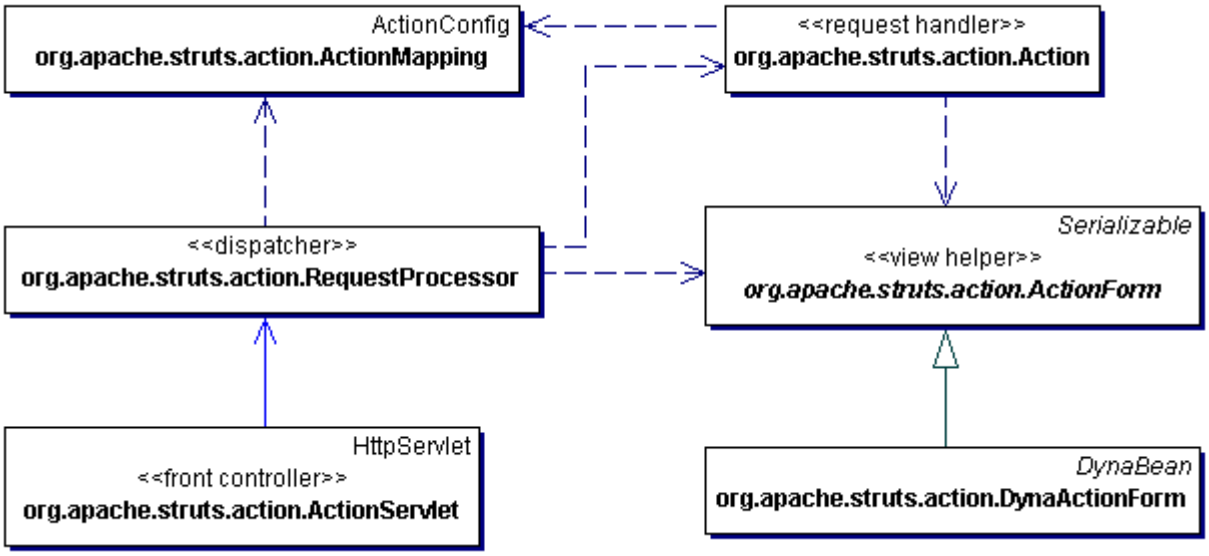
Capturing Form Data

The JSP specification provides a standard way for extracting and storing form data at request time in JavaBeans using `<jsp:useBean>` and `<jsp:setProperty>`. However, this solution creates a strong coupling between the presentation layer and the JavaBeans; furthermore the HTML document creator has to be aware of such components and their correct usage in the context of a page. Because the JavaBeans can be created and placed in a specified scope by the `<jsp:useBean>` tag or by another server component, there could be problems with bean lifecycle management between different components sharing the JavaBean. Struts provides a mechanism for extracting, storing, and validating form data; at the same time it overcomes the shortcomings of the `<jsp:useBean>` and `<jsp:setProperty>`.

Following is a recap of the `<action>` and `<form-bean>` elements.

```
<form-bean  name="customerProfileForm"
            type="packageName.customerProfileForm"/>
<action  path="/editCustomerProfile"
         type="packageName.EditCustomerProfileAction"
         name="customerProfileForm"
         scope="request"/>
```

The preceding snippet maps a JavaBean of type *packageName*.customerProfileForm with name=customerProfileForm (unique identifier) to an <action> element with name=customerProfileForm; the request handler is uniquely identified by the path /editCustomerProfile in the incoming request. The semantics of the form creation and usage are illustrated with the following static model.



First, we will explore the semantics of forms processing while employing simple JavaBeans objects. These objects are implemented as ActionForm objects. We will then discuss forms processing using the DynaActionForm object that can support dynamic set of properties at request time.

Initializing ActionForm objects with FormTag

An mentioned earlier in this section, the action URL in the HTML form is mapped to an <action> configuration, which in turn is mapped to a <form-bean> configuration. The URL specified in the action property of the FormTag is translated by the FormTag into a URL whose path structure confirms to the <url-pattern> specified in the deployment descriptor. For extension mapping, this implies that the resource extension is same as that specified for the <url-pattern>. Therefore, a URL of the form /editCustomerProfile?customerType=preferred, is translated into /editCustomerProfile.do?customerType=preferred.

The FormTag calls the RequestUtils.createActionForm(…) method, which will search for an ActionFormBean configuration object (ActionFormBean is the runtime representaion of `<form-bean>` element) with a name that matches the name specified on the corresponding `<action>` element. A new instance of the ActionForm is created using the `type` attribute of the `<form-bean>` element; a new instance is created when the ActionForm instance in not found in the specified scope, otherwise the FormTag calls the ActionForm.reset(…) method on the existing form bean to clear it in preparation for the form data from the next request. The scope is specified by the `scope` attribute in the `<action>` element; the new ActionForm instance or the existing reinitialized instance is saved in the specified scope using the `name` attribute.

Storing Form Data using ActionForm

The ActionForm derived objects are used for storing the parameters from a request object, and therefore they are tightly coupled to a user. An ActionForm subclass is a JavaBean with accessor methods for properties corresponding to parameters in the HttpServletRequest object. If an ActionForm object is created by the FormTag (discussed in the preceding section *FormTag*), then in the request subsequent to form rendering by the FormTag, the RequestProcessor will access the form from the specified scope; the form to be retrieved is identified by the related action mapping. The RequestProcessor will then reset the form properties, populate the form with request time parameters, and then call the validate(…) method on the form object to perform sever side validation of user input. The validate(…) method is called only when `validate` property in the ActionMapping object is set to true; this is the default behavior. The request.getParameterValues(parameterName) is used for getting a String[] object which is used for form population; this is explained further in section *Request Parameter Type-Conversion*. The result of validation could be an ActionErrors object, explained in the section *Error Handling*, which is used by org.apache.struts.taglib.html.ErrorsTag to display the validation errors to the user. The ActionForm can also be used for storing intermediate model state which is subsequently referenced by a view (a JSP) for presenting to the user.

An ActionForm class can also be created by the RequestProcessor. This happens when a forward is done to a URL that maps to the controller servlet rather than a JSP and the corresponding action mapping specifies the form property. In this case an attempt by the RequestProcessor to lookup the form bean may result in the creation of a new ActionForm object if not found in the specified scope. The ActionForm objects are found in the specified scope using the `name` attribute

specified in the <action> element; when a form object is found by the RequestProcessor, it is passed to the request handler's execute(…) method. An ActionForm object can also be created in a request handler.

Form objects created for the purpose of providing intermediate model state to the JSP should use request scope; this will ensure that the objects do not hang around after their usefulness expires. By default, all forms are saved in the session scope. The existence of form objects in the session beyond their usefulness could result in wasted memory, as such, the request handlers must track the lifecycle of form objects stored in the session. A good practice for capturing form data is to use a single form bean for related forms that span several user interactions. Form beans can also be used to store intermediate model state which can be adapted by custom tags for use in a view at response time. Tag usage prevents incorporation of Java code (scriptlets) in the view, thus achieving a good division of responsibility between web production team that primarily deals with markup, and application development team that primarily deals with writing Java code. The tags factor out logic for accessing intermediate model state; this logic could be quite complex when accessing nested objects or when iterating through a collection.

Creating ActionForm with Dynamic Properties

A DynaActionForm object is an object with dynamic set of properties. DynaActionForm extends the ActionForm; its usage permits creation of a form object through declarations made in the struts-config.xml as follows.

```
<form-bean  name="logonForm"
            type="org.apache.struts.action.DynaActionForm">
    <form-property name="username" type="java.lang.String"/>
    <form-property name="password" type="java.lang.String"/>
</form-bean>
```

The RequestProcessor creates, populates, and validates the DynaActionForm in the same way it does ActionForm, i.e. the parameters in the request object are populated in the DynaActionForm for the dynamic set of properties specified in the <form-bean> element; other parameters are simply skipped.

Request Parameter Type-Conversion

This discussion focuses on how String[] type retrieved by request.getParameterValues(*parameterName*) is converted to the target property type of the form object. Following is a list of supported target types.

```
java.lang.BigDecimal
java.lang.BigInteger
boolean and java.lang.Boolean
byte and java.lang.Byte
char and java.lang.Character
double and java.lang.Double
float and java.lang.Float
int and java.lang.Integer
long and java.lang.Long
short and java.lang.Short
java.lang.String
java.sql.Date
java.sql.Time
java.sql.Timestamp
```

The target types, i.e. the type associated with form object properties, are found using introspection mechanism; a Struts-specific custom introspection mechanism is used for DynaActionForm objects. Struts also supports indexed parameter names of the form parameterName[n]; where the index n is zero-based. The form bean methods corresponding to this naming convention are created according to the indexed property design patterns prescribed by the JavaBeans specification as shown below.

Following methods are used to access all array elements of an indexed property

```
public <PropertyType>[] get<PropertyName>();
public void set<PropertyName>(<PropertyType>[] value);
```

Following methods are used to access individual array elements

```
public <PropertyType> get<PropertyName>(int index)
public void set<PropertyName>(int index, <PropertyType> value)
```

The following describes the usage scenarios for indexed properties and simple properties.

1. When the bean property is an array, and the parameter name in the request does not use the indexing notation parameterName[n], then the String[] returned by request.getParameterValues(parameterName) is converted to an array of target component type. The method signatures implemented for DynaActionForm are internal to the Struts framework. The ActionForm subclass should be defined with following method signatures.

   ```
   public void set<PropertyName>(<PropertyType>[] value)
   public <PropertyType>[] get<PropertyName>();
   ```

2. When the bean property is of type array, and the parameter name in the request uses the indexing notation parameterName[n], then the String[] returned by request.getParameterValues(parameterName) is assumed to be containing only a single value; as such only String[0] is converted to the component type of the array. The method signatures implemented by DynaActionForm are internal to the Struts framework. The ActionForm subclass should be defined with following method signatures that accept an index argument.

   ```
   public void set<PropertyName>(int index, <PropertyType> value)
   public <PropertyType> get<PropertyName>(int index)
   ```

   The above method signatures follow the design patterns of indexed properties as stated in the JavaBeans specification. In the absence of these methods, indexed access using the indexing notation is also possible by implementing the following method.

   ```
   public <PropertyType>[] get<PropertyName>();
   ```

   In this scenario, the required array element to `get` or `set` is accessed by the Struts framework by first getting the underlying array object and then accessing the element for the given index. This pattern can also support a List based implementation for request parameters that use the indexing notation parameterName[n].

3. For simple property types, the String[] returned by request.getParameterValues(parameterName) is assumed to be containing only a single value; as such only String[0] is converted to the target type. The method signatures implemented by DynaActionForm are internal to the Struts framework. For simple properties, the ActionForm subclass should be defined with the following method signatures.

   ```
   public void set<PropertyName>(<PropertyType> value)
   public <PropertyType> get<PropertyName>();
   ```

The tag libraries bundled with Struts provide access to simple and indexed properties; the org.apache.struts.taglib.nested package contains tags that access simple and indexed properties in a nested combination. For a complete list of all tags available with Struts, refer to the *http://jakarta.apache.org/struts/userGuide/index.html*; more resources on Struts are available at *http://jakarta.apache.org/struts/resources/index.html*. The future direction of Struts is to transition over to JavaServer Pages Standard Tag Library (JSTL) and JavaServer Faces tags.

Custom Extensions with Plug-ins

A framework must provide a facility for creating custom extensions by allowing a mechanism for plugging external services seamlessly into the framework. This implies that the framework must provide extension points using which the life cycle management (i.e. init() and destroy()) of the pluggable component is possible. By providing such extension points, a developer can write a service that confirms to the interface supported by the extension mechanism, in this case the PlugIn interface, for controlling the creation, usage, and cleanup of the service and its corresponding resources within the context of the framework.

The Struts Validator is an example of a plug-in that enables declarative form validation. The corresponding entry in struts-config.xml is depicted below.

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
    <set-property property="pathnames"
        value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
</plug-in>
```

The ValidatorPlugIn class, and all other plug-in classes, are instantiated by the controller during its initialization. Each plug-in object is instantiated using the `className` attribute in the <plug-in> element. This plug-in object adheres to the design patterns of JavaBeans specification by providing the property accessor methods for each property specified in the `<plug-in>` element. Once a plug-in is instantiated, its init(…) method is called to enable the developer to perform plug-in specific initialization. For example, the ValidatorPlugIn.init(…) method will initialize its resources and save the resources in the context using ValidatorPlugIn.VALIDATOR_KEY; these resources are subsequently used for creating an instance of the class org.apache.commons.validator.Validator in the context of the framework. The plug-in(s) instantiated by the controller are saved in the context as an array of org.apache.struts.action.PlugIn objects using the key Action.PLUG_INS_KEY. This array is

subsequently used by the controller's destroy() method to call the destroy() method on each plug-in for releasing acquired resources.

Summary

Implementing MVC semantics for a request/response based HTTP protocol demands significant investment of time and effort. Selecting a suitable framework for solving this problem provides a head start for a project while allowing the architects and developers to focus on business semantics rather than integration semantics. Struts also provides complementary modules like the Struts Validator for declarative form validation, and Tiles for assembling composite views; these modules augment the framework and greatly simplify the task of designing and developing an application. More information on Struts and the related configuration and installation instructions can be found at *http://jakarta.apache.org/struts/userGuide/index.html*. Because Struts development is an ongoing endeavor, it is likely that by the time you read this article, some of the implementation may change, therefore it is best to complement the information provided in this chapter with release notes and updates posted at *http://jakarta.apache.org/struts*.

References

[Core] Core J2EE Patterns by Deepak Alur et. al. (Prentice Hall, 2001)

[Gof] Design Patterns by Erich Gamma et. al. (Addison-Wesley, 1995)

About the Author and his Upcoming Book

Nadir Gulzar is a Senior Architect with Enterpulse (http://www.enterpulse.com/). Nadir has been an evagelist and mentor for J2EE for several years. This article has been adapted from Chapter 4 of his upcoming book 'Practical J2EE Application Architecture' (ISBN 0072227117, Price $49.99),to be published by McGraw-Hill/Osborne Media (http://www.osborne.com/) in March 2003.

This book is written primarily for people interested in learning about the complete life cycle management of a J2EE application; this includes requirements analysis with use cases, information architecture for use case elaboration, project planning with a use case driven approach, application security, architecture and designing with patterns, implementation using

J2EE technologies, and implementing Web services. The book provides more details on Struts and its configuration semantics and shows a practical implementation of a Struts based application. By following the 'What you learn is what you use', readers are offered practical advise while going through the project lifecycle. The emphasis is on 'how to' rather than 'what is', which saves the readers from revisiting the basics that they are already familiar with. The book teaches the varied disciplines involved in creating enterprise solutions and at the same time synthesizes this information in a way that binds together the various artifacts of a project from inception to delivery.

Adapted material from Chapter 4 of Practical J2EE Application Architecture by Nadir Gulzar, to be published by McGraw-Hill/Osborne Media(www.osborne.com) in March 2003.