JavaServer™ Faces Specification

Version 20040513 Expert Draft

Craig McClanahan, Ed Burns, Roger Kitain, editors

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

February 2004, Revision 01

Submit comments about this document to `jsr-127-comments@jcp.org`

Specification: JSR-127, JavaServer(TM) Faces Specification ("Specification")

Version: 1.1

Status: Maintenance Release

Release: MAY 28, 2004

Contents

Preface

This is the JavaServer Faces 1.0 (JSF 1.0) specification, developed by the JSR-127 expert group under the Java Community Process (see <http://www.jcp.org> for more information about the JCP).

What's Changed Since the Last Release

Major changes/features in this release

There have been a few changes since the initial release of JavaServer technology. Here is a summary of the most important ones. Many thanks to Hans Bergsten and Adam Winer of the JSR127 Expert Group for these changes. Thanks also to Ryan Lubke of the TCK team for several changes.

General changes

- New 1.1 version of the DTD, backwards compatible with the 1.0 version. The only difference is that components and renderers can declare what facets they support. Please See *Section 10.3.3 "Application Configuration Resource Format"*.
- Introduce the concept of "no value" for SelectOne and SelectMany. class `com.sun.faces.component.UIInput`:
  - modify `isEmpty()` method to consider values that are zero length array or `List` instances to be empty.

- Refactor validation implementation in class `com.sun.faces.component.UIInput` to prevent spurious `ValueChangeEvent` instances from being fired from `UISelectOne` and `UISelectMany` classes. See the javadocs for `UIInput.validate()`.
- Method `com.sun.faces.component.UIViewRoot.getRenderKitId()` now returns `null` unless the setter has been explicitly called. See the javadocs for that method.
- `DoubleRangeValidator`, `Lengthvalidator`, and `LongRangeValidator` now require that any validation parameters passed to the validation error message be converted by the `javax.faces.Number` converter.
- The JavaDocs description `ResultSetDataModel.getRowData()` specifies that the returned Map must use a case-insensitive Comparator.
- `DataModelEvent.getRowIndex()` now returns -1 to indicate that no row is selected.
- Fix the JavaDoc description of the defaults for `showDetail` and `showSummary` for `UIMessage` to match the code.
- Fix JavaDoc description of `EditableValueHoldergetSubmittedValue()` to correctly say when this method is called.
- Fix JavaDoc for `UIComponentTag.setProperties()` to correctly describe which parameters are set.
- The implementation now allows nesting `<h:dataTable>` tags. Previously this didn't work.
- Fix bug where multiple action events could be generated in the case of multiple `<h:commandLink>` tags on page that is visited as a result of going “back” in the browser history.

Standard HTML RenderKit changes

- Made the "for" attribute no longer required for the outputLabel tag. This is necessary when tools want to allow the user to stick the label on the page before associating the component with it.
- RenderKit changes for SelectManyMenu, SelectManyList, SelectOneRadio, SelectManyCheckboxlist
  - Remove span around "select" tags in SelectManyMenu, SelectManyList, SelectOneMenu and SelectOneList.
  - Remove span around SelectOne radio buttons and SelectMany checkboxes. Render "id", "style", "styleclass" as part of outer table.
- The SelectManyCheckbox and SelectOneRadio renderers now do not render a “for” attribute on their nested `<label>` elements.
- The SelectOneRadio renderer description is more explicit about the use of the `<label>` element.

- The description of the “size” attribute in the SelectMany renderers is more correct with respect to the actual attributes exposed.
- The OutputLabel renderer is now able to handle the case where the compnent to which this label points hasn’t been created yet, as long as the component and the label are both in the same form.
- The “enabledClass” and “disabledClass” attributes are now specified for all select* renderers.

Spec document changes

- 2.5.2.4 LIMIT messages not used, remove LIMIT messages.
- 5.2 Table 5.1, modify action method signature to return String, not void.
- 5.3.1.3
  - In the section describing how to set a list-entries property, added a step describing what to do if the property is an array, yet the property getter had returned null.
  - Assign scopes to the implicit variables, so we can determine if a bean is able to refer to an implicit variable, dependeng on its scope. For example, a session scoped bean cannot refer to something in request scope.
  - Add a rule dealing with the net scope of mixed expressions: The net scope of mixed expressions is considered to be the scope of the narrowest expression in the mixed expression, excluding expressions with the none scope.
- 5.3.1.13 clarify that errors described in this section occurr at runtime, not deploytime.
- 9.4.3 Data type for "timeZone".
  - The "timeZone" attribute for <f:convertDateTime> in 9.4.3 is described to only accept a TimeZone instance, but must also accepts a String.
  - The "locale" attribute for <f:convertDateTime> and <f:convertNumber> in 9.4.3 and 9.4.4 is described to only accept a Locale instance, but must also accepts a String.
- 9.4.12 - 9.4.14 Correct validator and converter IDs
- 9.4.9 Incorrect data type for "itemValue"
  - The attributes table for <f:selectItem> in 9.4.9 states that the "itemValue" attribute takes a String but it should be Object to match the type of the UISelectItem property.
  - The syntax section in 9.4.9 for <f:selectItem> is missing a couple of right square brackets to mark the end for optional attributes.
- 9.4.10 contains a number of errors: The description of the getComponentType() return value omits the "javax.faces" prefix. The list of acceptable data types for the "value" attribute doesn't match the data type for UISelectItems.

- 9.4.8 <f:param> syntax section missing "binding"
- 10.2.6.1 Correct classnames for LifecycleFactory and RenderKitFactory.

Other Java™ Platform Specifications

JSF is based on the following Java API specifications:

- JavaServer Pages™ Specification, version 1.2 (JSP™) <http://java.sun.com/products/jsp/>
- Java™ Servlet Specification, version 2.3 (Servlet) <http://java.sun.com/products/servlet/>
- Java™2 Platform, Standard Edition, version 1.3 <http://java.sun.com/j2se/>
- JavaBeans™ Specification, version 1.0.1 <http://java.sun.com/products/javabeans/docs/spec.html>
- JavaServer Pages™ Standard Tag Library, version 1.0 (JSTL) <http://java.sun.com/products/jsp/jstl/>

Therefore, a JSF container must support all of the above specifications. This requirement allows faces applications to be portable across a variety of JSF implementations.

In addition, JSF is designed to work synergistically with other web-related Java APIs, including:

- Portlet Specification, under development in JSR-168 <http://www.jcp.org/jsr/detail/168.jsp>

Related Documents and Specifications

The following documents and specifications of the World Wide Web Consortium will be of interest to JSF implementors, as well as developers of applications and components based on JavaServer Faces.

- Hypertext Markup Language (HTML), version 4.01 <http://www.w3.org/TR/html4/>
- Extensible HyperText Markup Language (XHTML), version 1.0 <http://www.w3.org/TR/xhtml1>
- Extensible Markup Language (XML), version 1.0 (Second Edition) <http://www.w3.org/TR/REC-xml>

The class and method Javadoc documentation for the classes and interfaces in `javax.faces` (and its subpackages) are incorporated by reference as requirements of this Specification.

Terminology

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this document are to be interpreted as described in

- Key words for use in RFCs to Indicate Requirement Levels (RFC 2119) <http://www.rfc-editor.org/rfc/rfc2119.txt>

Providing Feedback

We welcome any and all feedback about this specification. Please email your comments to <jsr127-comments@sun.com>.

Please note that, due to the volume of feedback that we receive, you will not normally receive a reply from an engineer. However, each and every comment is read, evaluated, and archived by the specification team.

Acknowledgements

The JavaServer Faces Specification (version 1.0) is the result of the diligent efforts of the JSR-127 Expert Group, working under the auspices of the Java Community Process. We would like to thank all of the members of the Expert Group: Peter Abraham, Shawn Bayern, Hans Bergsten, Joseph Berkovitz, Mathias Bogaert, David Bosshaert, Pete Carapetyan, Renaud Demeur, Karl Ewald, Mike Frisino, David Geary, Antonio Hill, Kevin Jones, Amit Kishnani, Tom Lane, Eric Lazarus, Bart Leeten, Takahide Matsutsaka, Kumara Swamy Reddy Mettu, Kris Meukens, Steve Meyfroidt, Brendan Murray, Michael Nash, Daryl Olander, Steve Reiner, Brian Robinson, Michael Stapp, James Strachan, Kai Toedter, Ana Von Klopp, Adam Winer, Johanna Voolich Wright, John Zukowski, and Jason van Zyl.

Hans Bergsten and Adam Winer deserve special recognition for not only being actively involved in every detail of the development of the specification, and the corresponding APIs, but also for tirelessly contributing time to test, and patch bugs in, the reference implementation. Joe Berkovitz, David Geary, Brendan Murray, and Ana Von Klopp also made significant contributions.

Our thanks also go to Amy Fowler and Hans Muller, who were the original specification leads when JSR-127 was originally submitted to the JCP, and developed some of the key architectural ideas, and to Graham Hamilton, who had the idea to have this JSR in the first place.

CHAPTER 1

Overview

JavaServer Faces (JSF) is a *user interface* (UI) framework for Java web applications. It is designed to significantly ease the burden of writing and maintaining applications that run on a Java application server and render their UIs back to a target client. JSF provides ease-of-use in the following ways:

- Makes it easy to construct a UI from a set of reusable UI components
- Simplifies migration of application data to and from the UI
- Helps manage UI state across server requests
- Provides a simple model for wiring client-generated events to server-side application code
- Allows custom UI components to be easily built and re-used

Most importantly, JSF establishes standards which are designed to be leveraged by tools to provide a developer experience which is accessible to a wide variety of developer types, ranging from corporate developers to systems programmers. A "corporate developer" is characterized as an individual who is proficient in writing procedural code and business logic, but is not necessarily skilled in object-oriented programming. A "systems programmer" understands object-oriented fundamentals, including abstraction and designing for re-use. A corporate developer typically relies on tools for development, while a system programmer may define his or her tool as a text editor for writing code.

Therefore, JSF is designed to be tooled, but also exposes the framework and programming model as APIs so that it can be used outside of tools, as is sometimes required by systems programmers.

1.1 Solving Practical Problems of the Web

JSF's core architecture is designed to be independent of specific protocols and markup. However it is also aimed directly at solving many of the common problems encountered when writing applications for HTML clients that communicate via

HTTP to a Java application server that supports servlets and JavaServer Pages (JSP) based applications. These applications are typically form-based, and are comprised of one or more HTML pages with which the user interacts to complete a task or set of tasks. JSF tackles the following challenges associated with these applications:

- Managing UI component state across requests
- Supporting encapsulation of the differences in markup across different browsers and clients
- Supporting form processing (multi-page, more than one per page, and so on)
- Providing a strongly typed event model that allows the application to write server-side handlers (independent of HTTP) for client generated events
- Validating request data and providing appropriate error reporting
- Enabling type conversion when migrating markup values (Strings) to and from application data objects (which are often not Strings)
- Handling error and exceptions, and reporting errors in human-readable form back to the application user
- Handling page-to-page navigation in response to UI events and model interactions.

1.2 Specification Audience

The *JavaServer Faces Specification*, and the technology that it defines, is addressed to several audiences that will use this information in different ways. The following sections describe these audiences, the roles that they play with respect to JSF, and how they will use the information contained in this document. As is the case with many technologies, the same person may play more than one of these roles in a particular development scenario; however, it is still useful to understand the individual viewpoints separately.

1.2.1 Page Authors

A *page author* is primarily responsible for creating the user interface of a web application. He or she must be familiar with the markup and scripting languages (such as HTML and JavaScript) that are understood by the target client devices, as well as the rendering technology (such as JavaServer Pages) used to create dynamic content. Page authors are often focused on graphical design and human factors engineering, and are generally not familiar with programming languages such as Java or Visual Basic (although many page authors will have a basic understanding of client side scripting languages such as JavaScript).

Page authors will generally assemble the content of the pages being created from libraries of prebuilt user interface components that are provided by component writers, tool providers, and JSF implementors. The components themselves will be represented as configurable objects that utilize the dynamic markup capabilities of the underlying rendering technology. When JavaServer Pages are in use, for example, components will be represented as JSP custom actions, which will support configuring the attributes of those components as custom action attributes in the JSP page. In addition, the pages produced by a page author will be the used by the JSF framework to create component tree hierarchies, called "views", that represent the components on those pages.

Page authors will generally utilize development tools, such as HTML editors, that allow them to deal directly with the visual representation of the page being created. However, it is still feasible for a page author that is familiar with the underlying rendering technology to construct pages "by hand" using a text editor.

1.2.2 Component Writers

Component writers are responsible for creating libraries of reusable user interface objects. Such components support the following functionality:

- Convert the internal representation of the component's properties and attributes into the appropriate markup language for pages being rendered (encoding).
- Convert the properties of an incoming request—parameters, headers, and cookies—into the corresponding properties and attributes of the component (decoding)
- Utilize request-time events to initiate visual changes in one or more components, followed by redisplay of the current page.
- Support validation checks on the syntax and semantics of the representation of this component on an incoming request, as well as conversion into the internal form that is appropriate for this component.
- Saving and restoring component state across requests

As discussed in Chapter 8 "Rendering Model," the encoding and decoding functionality may optionally be delegated to one or more *Render Kits*, which are responsible for customizing these operations to the precise requirements of the client that is initiating a particular request (for example, adapting to the differences between JavaScript handling in different browsers, or variations in the WML markup supported by different wireless clients).

The component writer role is sometimes separate from other JSF roles, but is often combined. For example, reusable components, component libraries, and render kits might be created by:

- A page author creating a custom "widget" for use on a particular page
- An application developer providing components that correspond to specific data objects in the application's business domain

- A specialized team within a larger development group responsible for creating standardized components for reuse across applications
- Third party library and framework providers creating component libraries that are portable across JSF implementations
- Tool providers whose tools can leverage the specific capabilities of those libraries in development of JSF-based applications
- JSF implementors who provide implementation-specific component libraries as part of their JSF product suite

Within JSF, user interface components are represented as Java classes that follow the design patterns outlined in the JavaBeans Specification. Therefore, new and existing tools that facilitate JavaBean development can be leveraged to create new JSF components. In addition, the fundamental component APIs are simple enough for developers with basic Java programming skills to program by hand.

1.2.3 Application Developers

Application Developers are responsible for providing the server-side functionality of a web application that is not directly related to the user interface. This encompasses the following general areas of responsibility:

- Define mechanisms for persistent storage of the information required by JSF-based web applications (such as creating schemas in a relational database management system)
- Create a Java object representation of the persistent information, such as Entity Enterprise JavaBeans (Entity EJBs), and call the corresponding beans as necessary to perform persistence of the application's data.
- Encapsulate the application's functionality, or business logic, in Java objects that are reusable in web and non-web applications, such as Session EJBs.
- Expose the data representation and functional logic objects for use via JSF, as would be done for any servlet- or JSP-based application.

Only the latter responsibility is directly related to JavaServer Faces APIs. In particular, the following steps are required to fulfill this responsibility:

- Expose the underlying data required by the user interface layer as objects that are accessible from the web tier (such as via request or session attributes in the Servlet API), via *value reference expressions*, as described in Chapter 4 "Standard User Interface Components."
- Provide application-level event handlers for the events that are enqueued by JSF components during the request processing lifecycle, as described in Section 2.2.5 "Invoke Application".

Application modules interact with JSF through standard APIs, and can therefore be created using new and existing tools that facilitate general Java development. In addition, application modules can be written (either by hand, or by being generated) in conformance to an application framework created by a tool provider.

1.2.4 Tool Providers

Tool providers, as their name implies, are responsible for creating tools that assist in the development of JSF-based applications, rather than creating such applications directly. JSF APIs support the creation of a rich variety of development tools, which can create applications that are portable across multiple JSF implementations. Examples of possible tools include:

- GUI-oriented page development tools that assist page authors in creating the user interface for a web application
- IDEs that facilitate the creation of components (either for a particular page, or for a reusable component library)
- Page generators that work from a high level description of the desired user interface to create the corresponding page and component objects
- IDEs that support the development of general web applications, adapted to provide specialized support (such as configuration management) for JSF
- Web application frameworks (such as MVC-based and workflow management systems) that facilitate the use of JSF components for user interface design, in conjunction with higher level navigation management and other services
- Application generators that convert high level descriptions of an entire application into the set of pages, UI components, and application modules needed to provide the required application functionality

Tool providers will generally leverage the JSF APIs for introspection of the features of component libraries and render kit frameworks, as well as the application portability implied by the use of standard APIs in the code generated for an application.

1.2.5 JSF Implementors

Finally, *JSF implementors* will provide runtime environments that implement all of the requirements described in this specification. Typically, a JSF implementor will be the provider of a Java 2 Platform, Enterprise Edition (J2EE) application server, although it is also possible to provide a JSF implementation that is portable across J2EE servers.

Advanced features of the JSF APIs allow JSF implementors, as well as application developers, to customize and extend the basic functionality of JSF in a portable way. These features provide a rich environment for server vendors to compete on features and quality of service aspects of their implementations, while maximizing the portability of JSF-based applications across different JSF implementations.

1.3 Introduction to JSF APIs

This section briefly describes major functional subdivisions of the APIs defined by JavaServer Faces. Each subdivision is described in its own chapter, later in this specification.

1.3.1 package `javax.faces`

This package contains top level classes for the JavaServer(tm) Faces API. The most important class in the package is `FactoryFinder`, which is the mechanism by which users can override many of the key pieces of the implementation with their own.

Please see *Section 10.2.6.1 "FactoryFinder"*.

1.3.2 package `javax.faces.application`

This package contains APIs that are used to link an application's business logic objects to JavaServer Faces, as well as convenient pluggable mechanisms to manage the execution of an application that is based on JavaServer Faces. The main class in this package is `Application`.

Please see *Section 7.1 "Application"*.

1.3.3 package `javax.faces.component`

This package contains fundamental APIs for user interface components.

Please see *Chapter 3 "User Interface Component Model.*

1.3.4 package `javax.faces.component.html`

This package contains concrete base classes for each valid combination of component + renderer.

1.3.5 package `javax.faces.context`

This package contains classes and interfaces defining per-request state information. The main class in this package is `FacesContext`, which is the access point for all per-request information, as well as the gateway to several other helper classes.

Please see *Section 6.1 "FacesContext"*.

1.3.6 package `javax.faces.convert`

This package contains classes and interfaces defining converters. The main class in this package is `Converter`.

Please see *Section 3.3 "Conversion Model"*.

1.3.7 package `javax.faces.el`

This package contains classes and interfaces for evaluating and processing reference expressions.

Please see *Chapter 5 "Value Binding and Method Binding Expression Evaluation.*

1.3.8 package `javax.faces.lifecycle`

This package contains classes and interfaces defining lifecycle management for the JavaServer Faces implementation. The main class in this package is `Lifecycle`. `Lifecycle` is the gateway to executing the request processing lifecycle.

Please see *Chapter 2 "Request Processing Lifecycle.*

1.3.9 package `javax.faces.event`

This package contains interfaces describing events and event listeners, and concrete event implementation classes. All component-level events extend from `FacesEvent` and all component-level listeners extend from `FacesListener`.

Please see *Section 3.4 "Event and Listener Model"*.

1.3.10 package `javax.faces.render`

This package contains classes and interfaces defining the rendering model. The main class in this package is `RenderKit`. `RenderKit` vends a set of `Renderer` instances which provide rendering capability for a specific client device type.

Please see *Chapter 8 "Rendering Model.*

1.3.11 package `javax.faces.validator`

Interface defining the validator model, and concrete validator implementation classes.

Please see *Section 3.5 "Validation Model"*

1.3.12 package `javax.faces.webapp`

Classes required for integration of JavaServer Faces into web applications, including a standard servlet, base classes for JSP custom component tags, and concrete tag implementations for core tags.

Please see *Chapter 10 "Using JSF in Web Applications.*

CHAPTER **2**

Request Processing Lifecycle

Each request that involves a JSF component tree (also called a “view”) goes through a well-defined *request processing lifecycle* made up of *phases*. There are three different scenarios that must be considered, each with its own combination of phases and activities:

- Non-Faces Request generates Faces Response
- Faces Request generates Faces Response
- Faces Request generates Non-Faces Response

Where the terms being used are defined as follows:

- *Faces Response*—A response that was created by the execution of the *Render Response* phase of the request processing lifecycle.
- *Non-Faces Response*—A response that was not created by the execution of the *render response* phase of the request processing lifecycle. Examples would be a servlet-generated or JSP-rendered response that does not incorporate JSF components, or a response that sets an HTTP status code other than the usual 200 (such as a redirect).
- *Faces Request*—A request that was sent from a previously generated *Faces response*. Examples would be a hyperlink or form submit from a rendered user interface component, where the request URI was crafted (by the component or renderer that created it) to identify the view to use for processing the request.
- *Non-Faces Request*—A request that was sent to an application component (e.g. a servlet or JSP page), rather than directed to a Faces view.

In addition, of course, your web application may receive non-Faces requests that generate non-Faces responses. Because such requests do not involve JavaServer Faces at all, their processing is outside the scope of this specification, and will not be considered further.

READER NOTE: The dynamic behavior descriptions in this Chapter make forward references to the sections that describe the individual classes and interfaces. You will probably find it useful to follow the reference and skim the definition of each new class or interface as you encounter them, then come back and finish the behavior description. Later, you can study the characteristics of each JSF API in the subsequent chapters.

2.1 Request Processing Lifecycle Scenarios

Each of the scenarios described above has a lifecycle that is composed of a particular set of phases, executed in a particular order. The scenarios are described individually in the following subsections.

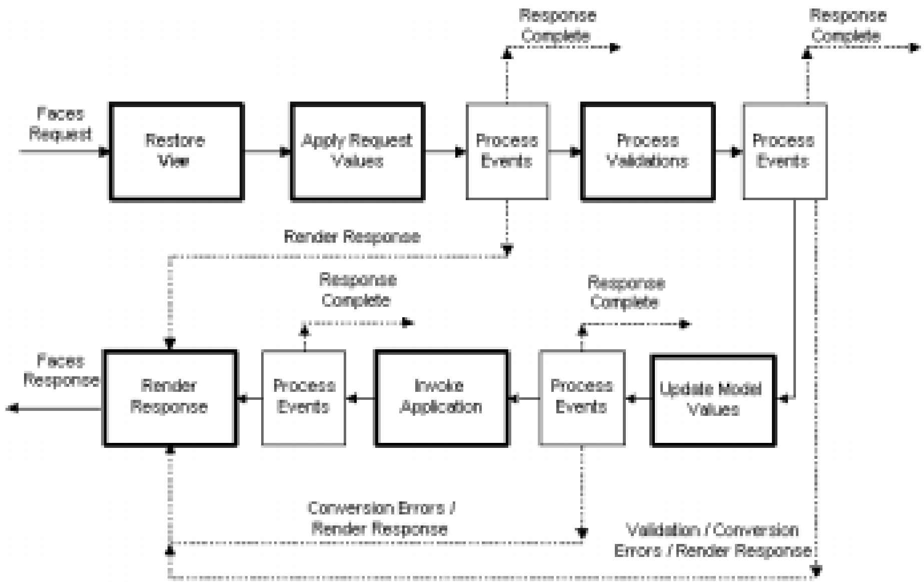
2.1.1 Non-Faces Request Generates Faces Response

An application that is processing a non-Faces request may use JSF to render a Faces response to that request. In order to accomplish this, the application must perform the common activities that are described in the following sections:

- Acquire Faces object references, as described in Section 2.4.1 “Acquire Faces Object References”, below.
- Create a new view, as described in Section 2.4.2 “Create And Configure A New View”, below.
- Store the view into the `FacesContext` by calling the `setViewRoot()` method on the `FacesContext`.
- Call the `render()` method on the `Lifecycle` instance that was acquired. This signals the JSF implementation to begin processing at the *Render Response* phase of the request processing lifecycle.

2.1.2 Faces Request Generates Faces Response

The most common lifecycle will be the case where a previous Faces response includes user interface controls that will submit a subsequent request to this web application, utilizing a request URI that is mapped to the JSF implementation’s controller, as described in Section 10.1.2 “Servlet Mapping”. Because such a request will be initially handled by the JSF implementation, the application need not take any special steps—its event listeners, validators, and application actions will be invoked at appropriate times as the standard request processing lifecycle, described in the following diagram, is invoked.



The behavior of the individual phases of the request processing lifecycle are described in individual subsections of Section 2.2 "Standard Request Processing Lifecycle Phases". Note that, at the conclusion of several phases of the request processing lifecycle, common event processing logic (as described in Section 2.3 "Common Event Processing") is performed to broadcast any `FacesEvents` generated by components in the component tree to interested event listeners.

2.1.3 Faces Request Generates Non-Faces Response

Normally, a JSF-based application will utilize the *Render Response* phase of the request processing lifecycle to actually create the response that is sent back to the client. In some circumstances, however, this behavior might not be desirable. For example:

- A Faces Request needs to be redirected to a different web application resource (via a call to `HttpServletResponse.sendRedirect`).
- A Faces Request causes the generation of a response using some other technology (such as a servlet, or a JSP page not containing JSF components).

In any of these scenarios, the application will have used the standard mechanisms of the servlet or portlet API to create the response headers and content. It is then necessary to tell the JSF implementation that the response has already been created, so that the *Render Response* phase of the request processing lifecycle should be skipped. This is accomplished by calling the `responseComplete()` method on the `FacesContext` instance for the current request, prior to returning from event handlers or application actions.

2.2 Standard Request Processing Lifecycle Phases

The standard phases of the request processing lifecycle are described in the following subsections.

2.2.1 Restore View

The JSF implementation must perform the following tasks during the *Restore View* phase of the request processing lifecycle:

- Examine the `FacesContext` instance for the current request. If it already contains a `UIViewRoot`:
  - Set the `locale` on this `UIViewRoot` to the value returned by the `getRequestLocale()` method on the `ExternalContext` for this request.
  - For each component in the component tree, determine if a `ValueBinding` for "binding" is present. If so, call the `setValue()` method on this `ValueBinding`, passing the component instance on which it was found.
  - Take no further action during this phase.
- Derive the *view identifier* that corresponds to this request, as follows:
  - If prefix mapping (such as "/faces/*") is used for `FacesServlet`, the `viewId` is set from the extra path information of the request URI.
  - If suffix mapping (such as "*.faces") is used for `FacesServlet`, the `viewId` is set from the servlet path information of the request URI, after replacing the suffix with the value of the context initialization parameter named by the symbolic constant `ViewHandler.DEFAULT_SUFFIX_NAME` (if no such context initialization parameter is present, use the value of the symbolic constant `ViewHandler.DEFAULT_SUFFIX` as the replacement suffix).
  - If no view identifier can be derived, throw an exception.
- Call `ViewHandler.restoreView()`, passing the `FacesContext` instance for the current request and the derived view identifier, and returning a `UIViewRoot` for the restored view (if any).

- If `restoreView()` returns null, call `ViewHandler.createView()` and `FacesContext.renderResponse()`.
- If the incoming request contains no POST data or query parameters, call `renderResponse()` on the `FacesContext` instance for this request.

- Store the restored or created `UIViewRoot` in the `FacesContext`.
- For each component in the component tree, determine if a `ValueBinding` for “binding” is present. If so, call the `setValue()` method on this `ValueBinding`, passing the component instance on which it was found.

At the end of this phase, the `viewRoot` property of the `FacesContext` instance for the current request will reflect the saved configuration of the view generated by the previous Faces Response (if any), or a new view returned by `ViewHandler.createView()` for the derived view identifier.

2.2.2 Apply Request Values

The purpose of the *Apply Request Values* phase of the request processing lifecycle is to give each component the opportunity to update its current state from the information included in the current request (parameters, headers, cookies, and so on).

During the *Apply Request Values* phase, the JSF implementation must call the `processDecodes()` method of the `UIViewRoot` of the component tree. This will normally cause the `processDecodes()` method of each component in the tree to be called recursively, as described in the Javadocs for the `UIComponent.processDecodes()` method. For `UIInput` components, data conversion must occur as described in the `UIInput` Javadocs.

During the decoding of request values, some components perform special processing, including:

- Components that implement `ActionSource` (such as `UICommand`), which recognize that they were activated, will queue an `ActionEvent`. The event will be delivered at the end of *Apply Request Values* phase, or at the end of *Invoke Application* phase, depending upon the state of the immediate property on the activated component.
- Components that implement `EditableValueHolder` (such as `UIInput`), and whose `immediate` property is set to `true`, will cause the conversion and validation processing (including the potential to fire `ValueChangeEvent` events) that normally happens during *Process Validations* phase to occur during *Apply Request Values* phase instead.

As described in Section 2.3 “Common Event Processing”, the `processDecodes()` method on the `UIViewRoot` component at the root of the component tree will have caused any queued events to be broadcast to interested listeners.

At the end of this phase, all `EditableValueHolder` components in the component tree will have been updated with new submitted values included in this request (or enough data to reproduce incorrect input will have been stored, if there were conversion errors). In addition, conversion and validation will have been performed on `EditableValueHolder` components whose `immediate` property is set to `true`. Conversions and validations that failed will have caused messages to be enqueued via calls to the `addMessage()` method of the `FacesContext` instance for the current request, and the `valid` property on the corresponding component(s) will be set to `false`.

If any of the `decode()` methods that were invoked, or an event listener that processed a queued event, called `responseComplete()` on the `FacesContext` instance for the current request, lifecycle processing of the current request must be immediately terminated. If any of the `decode()` methods that were invoked, or an event listener that processed a queued event, called `renderResponse()` on the `FacesContext` instance for the current request, control must be transferred to the *Render Response* phase of the request processing lifecycle. Otherwise, control must proceed to the *Process Validations* phase.

2.2.3 Process Validations

As part of the creation of the view for this request, zero or more `Validator` instances may have been registered for each component. In addition, component classes themselves may implement validation logic in their `validate()` methods.

During the *Process Validations* phase of the request processing lifecycle, the JSF implementation must call the `processValidators()` method of the `UIViewRoot` of the tree. This will normally cause the `processValidators()` method of each component in the tree to be called recursively, as described in the API reference for the `UIComponent.processValidators()` method. Note that `EditableValueHolder` components whose `immediate` property is set to `true` will have had their conversion and validation processing performed during *Apply Request Values* phase.

During the processing of validations, events may have been queued by the components and/or `Validator`s whose `validate()` method was invoked. As described in Section 2.3 “Common Event Processing”, the `processValidators()` method on the `UIViewRoot` component at the root of the component tree will have caused any queued events to be broadcast to interested listeners.

At the end of this phase, all conversions and configured validations will have been completed. Conversions and Validations that failed will have caused messages to be enqueued via calls to the `addMessage()` method of the `FacesContext` instance for the current request, and the `valid` property on the corresponding components will have been set to `false`.

If any of the `validate()` methods that were invoked, or an event listener that processed a queued event, called `responseComplete()` on the `FacesContext` instance for the current request, lifecycle processing of the current request must be immediately terminated. If any of the `validate()` methods that were invoked, or an event listener that processed a queued event, called `renderResponse()` on the `FacesContext` instance for the current request, control must be transferred to the *Render Response* phase of the request processing lifecycle. Otherwise, control must proceed to the *Update Model Values* phase.

2.2.4 Update Model Values

If this phase of the request processing lifecycle is reached, it is assumed that the incoming request is syntactically and semantically valid (according to the validations that were performed), that the local value of every component in the component tree has been updated, and that it is now appropriate to update the application's model data in preparation for performing any application events that have been enqueued.

During the *Update Model Values* phase, the JSF implementation must call the `processUpdates()` method of the `UIViewRoot` component of the tree. This will normally cause the `processUpdates()` method of each component in the tree to be called recursively, as described in the API reference for the `UIComponent.processUpdates()` method. The actual model update for a particular component is done in the `updateModel()` method for that component.

During the processing of model updates, events may have been queued by the components whose `updateModel()` method was invoked. As described in Section 2.3 "Common Event Processing", the `processUpdates()` method on the UIViewRoot component at the root of the component tree will have caused any queued events to be broadcast to interested listeners.

At the end of this phase, all appropriate model data objects will have had their values updated to match the local value of the corresponding component, and the component local values will have been cleared.

If any of the `updateModel()` methods that were invoked, or an event listener that processed a queued event, called `responseComplete()` on the `FacesContext` instance for the current request, lifecycle processing of the current request must be immediately terminated. If any of the `updateModel()` methods that was invoked, or an event listener that processed a queued event, called `renderResponse()` on the `FacesContext` instance for the current request, control must be transferred to the *Render Response* phase of the request processing lifecycle. Otherwise, control must proceed to the *Invoke Application* phase.

2.2.5 Invoke Application

If this phase of the request processing lifecycle is reached, it is assumed that all model updates have been completed, and any remaining event broadcast to the application needs to be performed. The implementation must ensure that the `processApplication()` method of the `UIViewRoot` instance is called. The default behavior of this method will be to broadcast any queued events that specify a phase identifier of `PhaseId.INVOKE_APPLICATION`.

Advanced applications (or application frameworks) may replace the default `ActionListener` instance by calling the `setActionListener()` method on the `Application` instance for this application. However, the JSF implementation must provide a default `ActionListener` instance that behaves as described in Section 7.1.1 "ActionListener Property".

2.2.6 Render Response

This phase accomplishes two things:

1. Causes the response to be rendered to the client

2. Causes the state of the response to be saved for processing on subsequent requests.

The reason for bundling both of these responsibilities into this phase is that in JSP applications, the act of rendering the response may cause the view to be built, as the page renders. Thus, we can't save the state until the view is built, and we have to save the state before sending the response to the client to enable saving the state in the client.

JSF supports a range of approaches that JSF implementations may utilize in creating the response text that corresponds to the contents of the response view, including:

- Deriving all of the response content directly from the results of the encoding methods (on either the components or the corresponding renderers) that are called.
- Interleaving the results of component encoding with content that is dynamically generated by application programming logic.
- Interleaving the results of component encoding with content that is copied from a static "template" resource.
- Interleaving the results of component encoding by embedding calls to the encoding methods into a dynamic resource (such as representing the components as custom tags in a JSP page).

Because of the number of possible options, the mechanism for implementing the *Render Response* phase cannot be specified precisely. However, all JSF implementations of this phase must conform to the following requirements:

- JSF implementations must provide a default ViewHandler implementation that performs a RequestDispatcher.forward() call to a web application resource whose context-relative path is equal to the view identifier of the component tree.
- If all of the response content is being derived from the encoding methods of the component or associated Renderers, the component tree should be walked in the same depth-first manner as was used in earlier phases to process the component tree, but subject to the additional constraints listed here.
- If the response content is being interleaved from additional sources and the encoding methods, the components may be selected for rendering in any desired order[1].
- During the rendering process, additional components may be added to the component tree based on information available to the ViewHandler implementation[2]. However, before adding a new component, the ViewHandler implementation must first check for the existence of the corresponding component in the component tree. If the component already exists (perhaps because a previous phase has pre-created one or more components), the existing component's properties and attributes must be utilized.
- Under no circumstances should a component be selected for rendering when its parent component, or any of its ancestors in the component tree, has its rendersChildren property set to true. In such cases, the parent or ancestor component must render the content of this child component when the parent or ancestor was selected.
- If the isRendered() method of a component returns false, the renderer for that component must not generate any markup, and none of its facets or children (if any) should be rendered.

When each particular component in the component tree is selected for rendering, calls to its encodeXxx() methods must be performed in the manner described in Section 3.1.12 "Component Specialization Methods". For components that implement ValueHolder (such as UIInput and UIOutput), data conversion must occur as described in the UIOutput Javadocs.

Upon completion of rendering, the completed state of the view must have been saved using the methods of the class StateManager. This state information must be made accessible on a subsequent request, so that the *Restore View* can access it. For more on StateManager, see Section 7.6.3 "State Saving Methods."

1. Typically, component selection will be driven by the occurrence of special markup (such as the existence of a JSP custom tag) in the template text associated with the component tree.
2. For example, this technique is used when custom tags in JSP pages are utilized as the rendering technology, as described in *Chapter 9 "Integration with JSP*.

2.3 Common Event Processing

For a complete description of the event processing model for JavaServer Faces components, see Section 3.4 “Event and Listener Model”.

During several phases of the request processing lifecycle, as described in Section 2.2 “Standard Request Processing Lifecycle Phases”, the possibility exists for events to be queued (via a call to the `queueEvent()` method on the source `UIComponent` instance, or a call to the `queue()` method on the `FacesEvent` instance), which must now be broadcast to interested event listeners. The broadcast is performed as a side effect of calling the appropriate lifecycle management method (`processDecodes()`, `processValidators()`, `processUpdates()`, or `processApplication()`) on the `UIViewRoot` instance at the root of the current component tree.

For each queued event, the `broadcast()` method of the source `UIComponent` must be called to broadcast the event to all event listeners who have registered an interest, on this source component for events of the specified type, after which the event is removed from the event queue. See the API reference for the `UIComponent.broadcast()` method for the detailed functional requirements.

It is also possible for event listeners to cause additional events to be enqueued for processing during the current phase of the request processing lifecycle. Such events must be broadcast in the order they were enqueued, after all originally queued events have been broadcast, before the lifecycle management method returns.

2.4 Common Application Activities

The following subsections describe common activities that may be undertaken by an application that is using JSF to process an incoming request and/or create an outgoing response. Their use is described in Section 2.1 “Request Processing Lifecycle Scenarios”, for each request processing lifecycle scenario in which the activity is relevant.

2.4.1 Acquire Faces Object References

This phase is only required when the request being processed was not submitted from a previous response, and therefore did not initiate the *Faces Request Generates Faces Response* lifecycle. In order to generate a Faces Response, the application must first acquire references to several objects provided by the JSF implementation, as described below.

2.4.1.1 Acquire and Configure Lifecycle Reference

As described in Section 11.1 "Lifecycle", the JSF implementation must provide an instance of `javax.faces.lifecycle.Lifecycle` that may be utilized to manage the remainder of the request processing lifecycle. An application may acquire a reference to this instance in a portable manner, as follows:

```
LifecycleFactory lFactory = (LifecycleFactory)
  FactoryFinder.getFactory(FactoryFinder.LIFECYCLE_FACTORY);
Lifecycle lifecycle =
  lFactory.getLifecycle(LifecycleFactory.DEFAULT_LIFECYCLE);
```

It is also legal to specify a different lifecycle identifier as a parameter to the `getLifecycle()` method, as long as this identifier is recognized and supported by the JSF implementation you are using. However, using a non-default lifecycle identifier will generally not be portable to any other JSF implementation.

2.4.1.2 Acquire and Configure FacesContext Reference

As described in Section 6.1 "FacesContext", the JSF implementation must provide an instance of `javax.faces.context.FacesContext` to contain all of the per-request state information for a Faces Request or a Faces Response. An application that is processing a Non-Faces Request, but wants to create a Faces Response, must acquire a reference to a `FacesContext` instance as follows

```
FacesContextFactory fcFactory = (FacesContextFactory)
  FactoryFinder.getFactory(FactoryFinder.FACES_CONTEXT_FACTORY);
FacesContext facesContext =
  fcFactory.getFacesContext(context, request, response,
                            lifecycle);
```

where the `context`, `request`, and `response` objects represent the corresponding instances for the application environment. For example, in a servlet-based application, these would be the `ServletContext`, `HttpServletRequest`, and `HttpServletResponse` instances for the current request.

2.4.2 Create And Configure A New View

When a Faces response is being intially created, or when the application decides it wants to create and configure a new view that will ultimately be rendered, it may follow the steps described below in order to set up the view that will be used. You must start with a reference to a `FacesContext` instance for the current request.

2.4.2.1 Create A New View

Views are represented by a data structure rooted in an instance of `javax.faces.component.UIViewRoot`, and identified by a view identifier whose meaning depends on the `ViewHandler` implementation to be used during the *Render Response* phase of the request processing lifecycle[3]. The `ViewHandler` provides a factory method that may be utilized to construct new component trees, as follows:

```
String viewId = ...identifier of the desired Tree...;
ViewHandler viewHandler = application.getViewHandler();
UIViewRoot view = viewHandler.createView(facesContext, viewId);
```

The `UIViewRoot` instance returned by the `createView()` method must minimally contain a single `UIViewRoot` provided by the JSF implementation, which must encapsulate any implementation-specific component management that is required. Optionally, a JSF implementation's `ViewHandler` may support the automatic population of the returned `UIViewRoot` with additional components, perhaps based on some external metadata description.

The caller of `ViewHandler.createView()` must cause the `FacesContext` to be populated with the new `UIViewRoot.` Applications must make sure that it is safe to discard any state saved in the view rooted at the `UIViewRoot` currently stored in the `FacesContext`.

3. The default `ViewHandler` implementation performs a `RequestDispatcher.forward` call to the web application resource that will actually perform the rendering, so it expects the tree identifier to be the context-relative path (starting with a `/` character) of the web application resource

2.4.2.2 Configure the Desired RenderKit

The `UIViewRoot` instance provided by the `ViewHandler`, as described in the previous subsection, must automatically be configured to utilize the default `javax.faces.render.RenderKit` implementation provided by the JSF implementation, as described in Section 8.1 “RenderKit”. This `RenderKit` must support the standard components and `Renderer`s described later in this specification, to maximize the portability of your application.

However, a different `RenderKit` instance provided by your JSF implementation (or as an add-on library) may be utilized instead, if desired. A reference to this `RenderKit` instance can be obtained from the standard `RenderKitFactory`, and then assigned to the `UIViewRoot` instance created previously, as follows:

```
String renderKitId = ... identifier of desired RenderKit ...;
RenderKitFactory rkFactory = (RenderKitFactory)
  FactoryFinder.getFactory(FactoryFinder.RENDER_KIT_FACTORY);
RenderKit renderKit = rkFactory.getRenderKit(renderKitId,
  facesContext);
view.setRenderKitId(renderKitId);
```

As described in Chapter 8, changing the `RenderKit` being used changes the set of `Renderer`s that will actually perform decoding and encoding activities. Because the components themselves store only a `rendererType` property (a logical identifier of a particular `Renderer`), it is thus very easy to switch between `RenderKit`s, as long as they support renderers with the same renderer types.

In the current version of this specification, the default `ViewHandler` implementation does not support using `RenderKit`s other than the default one (configured by the `<default-render-kit-id>` configuration element), because the render kit identifier is not exposed separately in the `StateManager` APIs. This restriction may be lifted in a future version of the specification. In the mean time, it is possible to support this feature by implementing a custom `ViewHandler` that handles saving and restoring the render kit identifier in a custom manner.

2.4.2.3 Configure The View’s Components

At any time, the application can add new components to the view, remove them, or modify the attributes and properties of existing components. For example, a new `FooComponent` (an implementation of `UIComponent`) can be added as a child to the root `UIViewRoot` in the component tree as follows:

```
FooComponent component = ...create a FooComponent instance...;
facesContext.getViewRoot().getChildren().add(component);
```

2.4.2.4 Store the new View in the FacesContext

Once the view has been created and configured, the `FacesContext` instance for this request must be made aware of it by calling `setViewRoot()`.

2.5 Concepts that impact several lifecycle phases

This section is intended to give the reader a “big picture” perspective on several complex concepts that impact several request processing lifecycle phases.

2.5.1 Value Handling

At a fundamental level, JavaServer Faces is a way to get values from the user, into your model tier for processing. The process by which values flow from the user to the model has been documented elsewhere in this spec, but a brief holistic survey comes in handy. The following description assumes the JSP / HTTP case, and that all components have Renderers.

2.5.1.1 Apply Request Values Phase

The user presses a button that causes a form submit to occur. This causes the state of the form to be sent as `name=value` pairs in the `POST` data of the HTTP request. The JSF request processing lifecycle is entered, and eventually we come to the *Apply Request Values Phase*. In this phase, the `decode()` method for each `Renderer` for each `UIComponent` in the view is called. The `Renderer` takes the value from the request and passes it to the `setSubmittedValue()` method of the component, which is, of course, an instance of `EditableValueHolder`. If the component has the "`immediate`" property set to `true`, we execute validation immediately after decoding. See below for what happens when we execute validation.

2.5.1.2 Process Validators Phase

`processValidators()` is called on the root of the view. For each `EditableValueHolder` in the view, If the “`immediate`” property is not set, we execute validation for each `UIInput` in the view. Otherwise, validation has already occurred and this phase is a no-op.

2.5.1.3 Executing Validation

Please see the javadocs for `UIInput.validate()` for more details, but basically, this method gets the submitted value from the component (set during *Apply Request Values*), gets the `Renderer` for the component and calls its `getConvertedValue()`, passing the submitted value. If a conversion error occurs, it is dealt with as described in the javadocs for that method. Otherwise, all validators attached to the component are asked to validate the converted value. If any validation errors occur, they are dealt with as described in the javadocs for `Validator.validate()`. The converted value is pushed into the component's `setValue()` method, and a `ValueChangeEvent` is fired if the value has changed.

2.5.1.4 Update Model Values Phase

For each `UIInput` component in the view, its `updateModel()` method is called. This method only takes action if a local value was set when validation executed and if the page author configured this component to push its value to the model tier. This phase simply causes the converted local value of the `UIInput` component to be pushed to the model in the way specified by the page author. Any errors that occur as a result of the attempt to push the value to the model tier are dealt with as described in the javadocs for `UIInput.updateModel()`.

2.5.2 Localization and Internationalization (L10N / I18N)

JavaServer Faces is fully internationalized. The I18N capability in JavaServer Faces builds on the I18N concepts offered in the Servlet, JSP and JSTL specifications. I18N happens at several points in the request processing lifecycle, but it is easiest to explain what goes on by breaking the task down by function.

2.5.2.1 Determining the active `Locale`

JSF has the concept of an active `Locale` which is used to look up all localized resources. Converters must use this `Locale` when performing their conversion. This `Locale` is stored as the value of the `locale` JavaBeans property on the `UIViewRoot` of the current `FacesContext`. The application developer can tell JSF what locales the application supports in the applications' `WEB-INF/faces-config.xml` file. For example:

```
<faces-config>
  <application>
    <locale-config>
```

```
    <default-locale>en</default-locale>
    <supported-locale>de</supported-locale>
    <supported-locale>fr</supported-locale>
    <supported-locale>es</supported-locale>
  </locale-config>
</application>
```

This application's default locale is en, but it also supports de, fr, and es locales. These elements cause the Application instance to be populated with Locale data. Please see the javadocs for details.

The UIViewRoot's Locale is determined and set by the ViewHandler during the execution of the ViewHandler's createView() method. This method must cause the active Locale to be determined by looking at the user's preferences combined with the application's stated supported locales. Please see the javadocs for details.

The application can call UIViewRoot.setLocale() directly, but it is also possible for the page author to override the UIViewRoot's locale by using the locale attribute on the <f:view> tag. The value of this attribute must be specified as language[{-|_}country[{-|_}variant]] without the colons, for example "ja_JP_SJIS". The separators between the segments may be '-' or '_'.

In all cases where JSP is utilized, the active Locale is set under "request scope" into the JSTL class javax.servlet.jsp.jstl.core.Config, under the key Config.FMT_LOCALE.

2.5.2.2 Determining the Character Encoding

The request and response character encoding are set and interpreted as follows.

On an initial request to a Faces webapp, the request character encoding is left unmodified, relying on the underlying request object (e.g., the servlet or portlet request) to parse request parameter correctly.

At the beginning of the render-response phase, the ViewHandler must ensure that the response Locale is set to be that of the UIViewRoot, for example by calling ServletResponse.setLocale() when running in the servlet environment. Setting the response Locale may affect the response character encoding, see the Servlet and Portlet specifications for details.

At the end of the render-response phase, the ViewHandler must store the response character encoding used by the underlying response object (e.g., the servlet or portlet response) in the session (if and only if a session already exists) under a well known, implementation-dependent key.

On a subsequent postback, before any of the ExternalContext methods for accessing request parameters are invoked, the ViewHandler must examine the Content-Type header to read the charset attribute and use its value to set it as the request encoding for the underlying request object. If the Content-Type header doesn't contain a charset attribute, the encoding previously stored in the session (if and only if a session already exists), must be used to set the encoding for the underlying request object. If no character encoding is found, the request encoding must be left unmodified.

The above algorithm allows an application to use the mechanisms of the underlying technologies to adjust both the request and response encoding in an application-specific manner, for instance using the page directive with a fixed character encoding defined in the contentType attribute in a JSP page, see the Servlet, Portlet and JSP specifications for details. Note, though, that the character encoding rules prior to Servlet 2.4 and JSP 2.0 are imprecise and special care must be taken for portability between containers.

2.5.2.3 Localized Text

There is no direct support for this in the API, but the JSP layer provides a convenience tag that converts a `ResourceBundle` into a `java.util.Map` and stores it in the scoped namespace so all may get to it. This section describes how resources displayed to the end user may be localized. This includes images, labels, button text, tooltips, alt text, etc.

Since most JSF components allow pulling their display value from the model tier, it is easy to do the localization at the model tier level. As a convenience, JSF provides the `<f:loadBundle>` tag, which takes a `ResourceBundle` and loads it into a `Map`, which is then stored in the scoped namespace in request scope, thus making its messages available using the same mechanism for accessing data in the model tier. For example:

```
<f:loadBundle basename=”com.foo.industryMessages.chemical”
              var=”messages” />
<h:outputText value=”#{messages.benzene}” />
```

This must cause the `ResourceBundle` named `com.foo.industryMessages.chemical` to be loaded as a Map into the request scope under the key `messages`. Localized content can then be pulled out of it using the normal value binding syntax.

2.5.2.4 Localized Application Messages

This section describes how JSF handles localized error and informational messages that occur as a result of conversion, validation, or other application actions during the request processing lifecycle. The JSF class `javax.faces.application.FacesMessage` is provided to encapsulate summary, detail, and severity information for a message. A JSF implementation must provide a `javax.faces.Messages ResourceBundle` containing all of the necessary keys for the standard messages. The required keys (and a non-normative indication of the intended message text) are as follows:

- javax.faces.component.UIInput.CONVERSION -- Conversion error occurred
- javax.faces.component.UIInput.REQUIRED -- Value is required
- javax.faces.component.UISelectOne.INVALID -- Value is not a a valid option
- javax.faces.component.UISelectMany.INVALID -- Value is not a valid option
- javax.faces.validator.NOT_IN_RANGE -- Specified attribute is not between the expected values of {0} and {1}
- javax.faces.validator.DoubleRangeValidator.MAXIMUM -- Value is greater than allowable maximum of ''{0}''
- javax.faces.validator.DoubleRangeValidator.MINIMUM -- Value is less than allowable minimum of ''{0}''
- javax.faces.validator.DoubleRangeValidator.TYPE -- Value is not of the correct type
- javax.faces.validator.LengthValidator.MAXIMUM -- Value is greater than allowable maximum of ''{0}''
- javax.faces.validator.LengthValidator.MINIMUM -- Value is less than allowable minimum of ''{0}''
- javax.faces.validator.LongRangeValidator.MAXIMUM -- Value is greater than allowable maximum of ''{0}''
- javax.faces.validator.LongRangeValidator.MINIMUM -- Value is less than allowable minimum of ''{0}''
- javax.faces.validator.LongRangeValidator.TYPE -- Value is not of the correct type

A JSF application may provide its own messages, or overrides to the standard messages by supplying a `<message-bundle>` element to in the application configuration resources. Since the `ResourceBundle` provided in the Java platform has no notion of summary or detail, JSF adopts the policy that `ResourceBundle` key for the message looks up the message summary. The detail is stored under the same key as the summary, with `_detail` appended. These `ResourceBundle` keys must be used to look up the necessary values to create a localized `FacesMessage` instance. Note that the value of the summary and detail keys in the `ResourceBundle` may contain parameter substitution tokens, which must be substituted with the appropriate values using `java.text.MessageFormat`.

These messages can be displayed in the page using the `UIMessage` and `UIMessages` components and their corresponding tags, `<h:message>` and `<h:messages>`.

The following algorithm must be used to create a `FacesMessage` instance given a message key.

- Call `getMessageBundle()` on the `Application` instance for this web application, to determine if the application has defined a resource bundle name. If so, load that ResourceBundle and look for the message there.
- If not there, look in the `javax.faces.Messages` resource bundle.
- In either case, if a message is found, use the above conventions to create a `FacesMessage` instance.

2.5.3 State Management

JavaServer Faces introduces a powerful and flexible system for saving and restoring the state of the view between requests to the server. It is useful to describe state management from several viewpoints. For the page author, state management happens transparently. For the app assembler, state management can be configured to save the state in the client or on the server by setting the ServletContext InitParameter named `javax.faces.STATE_SAVING_METHOD` to either `client` or `server`. The value of this parameter directs the state management decisions made by the implementation.

2.5.3.1 State Management Considerations for the Custom Component Author

Since the component developer cannot know what the state saving method will be at runtime, they must be aware of state management. As shown in *Section FIGURE 4-1 "The* javax.faces.component package", all JSF components implement the `StateHolder` interface. As a consequence the standard components provide implementations of `StateHolder` to suit their needs. A custom component that extends `UIComponent` directly, and does not extend any of the standard components must implement `StateHolder` manually.Please see *Section 3.2.3 "StateHolder"* for details.

A custom component that **does** extend from one of the standard components and maintains its own state, in addition to the state maintained by the superclass must take special care to implement `StateHolder` correctly. Notably, calls to `saveState()` must not alter the state in any way. The subclass is responsible for saving and restoring the state of the superclass. Consider this example. My custom component represents a "slider" ui widget. As such, it needs to keep track of the maximum value, minimum value, and current values as part of its state.

```
public class Slider extends UISelectOne {
  protected Integer min = null;
  protected Integer max = null;
  protected Integer cur = null;

// ... details omitted
public Object saveState(FacesContext context) {
  Object values[] = new Object[4];
  values[0] = super.saveState(context);
  values[1] = min;
  values[2] = max;
  values[3] = cur;
}

public void restoreState(FacesContext context, Object state) {
  Object values[] = (Object {}) state; // guaranteed to succeed
  super.restoreState(context, values[0]);
  min = (Integer) values[1];
  max = (Integer) values[2];
  cur = (Integer) values[3];
}
```

Note that we call `super.saveState()` and `super.restoreState()` as appropriate. This is absolutely vital! Failing to do this will prevent the component from working.

2.5.3.2 State Management Considerations for the JSF Implementor

The intent of the state management facility is to make life easier for the page author, app assembler, and component author. However, the complexity has to live somewhere, and the JSF implementor is the lucky role. Here is an overview of the key players. Please see the javadocs for each individual class for more information.

Key Players in State Management

- `ViewHandler` the entry point to the state management system. Uses a helper class, `StateManager`, to do the actual work. In the JSP case, delegates to the tag handler for the `<f:view>` tag for some functionality.

- `StateManager` abstraction for the hard work of state saving. Uses a helper class, `ResponseStateManager`, for the rendering technology specific decisions.
- `ResponseStateManager` abstraction for rendering technology specific state management decisions.
- `UIComponent` directs process of saving and restoring individual component state.

CHAPTER **3**

User Interface Component Model

A JSF *user interface component* is the basic building block for creating a JSF user interface. A particular component represents a configurable and reusable element in the user interface, which may range in complexity from simple (such as a button or text field) to compound (such as a tree control or table). Components can optionally be associated with corresponding objects in the data model of an application, via *value binding expressions*.

JSF also supports user interface components with several additional helper APIs:

- *Converters*—Pluggable support class to convert the markup value of a component to and from the corresponding type in the model tier.
- *Events and Listeners*—An event broadcast and listener registration model based on the design patterns of the JavaBeans Specification, version 1.0.1.
- *Validators*—Pluggable support classes that can examine the local value of a component (as received in an incoming request) and ensure that it conforms to the business rules enforced by each Validator. Error messages for validation failures can be generated and sent back to the user during rendering.

The user interface for a particular page of a JSF-based web application is created by assembling the user interface components for a particular request or response into a *view*. The view is a tree of classes that implement `UIComponent`. The components in the tree have parent-child relationships with other components, starting at the *root element* of the tree, which must be an instance of `UIViewRoot`. Components in the tree can be anonymous or they can be given a *component identifier* by the framework user. Components in the tree can be located based on *component identifiers*, which must be unique within the scope of the nearest ancestor to the component that is a *naming container*. For complex rendering scenarios, components can also be attached to other components as *facets*.

This chapter describes the basic architecture and APIs for user interface components and the supporting APIs.

3.1 UIComponent and UIComponentBase

The base abstract class for all user interface components is `javax.faces.component.UIComponent`. This class defines the state information and behavioral contracts for all components through a Java programming language API, which means that components are independent of a rendering technology such as JavaServer Pages (JSP). A standard set of components (described in Chapter 4 "Standard User Interface Components") that add specialized properties, attributes, and behavior, is also provided as a set of concrete subclasses.

Component writers, tool providers, application developers, and JSF implementors can also create additional `UIComponent` implementations for use within a particular application. To assist such developers, a convenience subclass, `javax.faces.component.UIComponentBase`, is provided as part of JSF. This class provides useful default implementations of nearly every `UIComponent` method, allowing the component writer to focus on the unique characteristics of a particular `UIComponent` implementation.

The following subsections define the key functional capabilities of JSF user interface components.

3.1.1 Component Identifiers

```
public String getId();

public void setId(String componentId);
```

Every component may be named by a *component identifier*, which (if utilized) must be unique among the components that share a common *naming container* parent in a component tree. Component identifiers must conform to the following rules:

- They must start with a letter (as defined by the `Character.isLetter()` method) or underscore ('_').
- Subsequent characters may be letters (as defined by the `Character.isLetter()` method), digits as defined by the `Character.isDigit()` method, dashes ('-'), and underscores ('_').

To minimize the size of responses generated by JavaServer Faces, it is recommended that component identifiers be as short as possible.

If a component has been given an identifier, it must be unique in the namespace of the closest ancestor to that component that is a `NamingContainer` (if any).

3.1.2 Component Type

While not a property of `UIComponent`, the `component-type` is an important piece of data related to each `UIComponent` subclass that allows the `Application` instance to create new instances of `UIComponent` subclasses with that type. Please see *Section 7.1.10 "Object Factories"* for more on `component-type`.

Component types starting with "javax.faces." are reserved for use by the JSF specification.

3.1.3 Component Family

```
public String getFamily();
```

Each standard user interface component class has a standard value for the component family, which is used to look up renderers associated with this component. Subclasses of a generic UIComponent class will generally inherit this property from its superclass, so that renderers who only expect the superclass will still be able to process specialized subclasses.

Component families starting with "javax.faces." are reserved for use by the JSF specification.

3.1.4 Value Binding Expressions

Properties and attributes of standard concrete component classes may be *value binding enabled*. This means that, rather than specifying a literal value as the parameter to a property or attribute setter, the caller instead associates a ValueBinding (see Section 5.3.3 "ValueBinding") whose `getValue()` method must be called (by the property getter) to return the actual property value to be returned if no value has been set via the corresponding property setter. If a property or attribute value has been set, that value must be returned by the property getter (shadowing any associated value binding expression for this property).

Value binding expressions are managed with the following method calls:

```
public ValueBinding getValueBinding(String name);

public void setValueBinding(String name, ValueBinding binding);
```

where `name` is the name of the attribute or property for which to establish the value binding. For the standard component classes defined by this specification, all attributes, and all properties other than `id` and `parent`, are value binding enabled.

3.1.5 Component Bindings

A *component binding* is a special value binding expression that can be used to facilitate "wiring up" a component instance to a corresponding property of a JavaBean that is associated with the page, and wants to manipulate component instances programatically. It is established by calling `setValueBinding()` (see Section 3.1.4 "Value Binding Expressions") with the special property name `binding`.

The specified `ValueBinding` must point to a read-write JavaBeans property of type `UIComponent` (or appropriate subclass). Such a component binding is used at two different times during the processing of a Faces Request:

- When a component instance is first created (typically by virtue of being referenced by a `UIComponentTag` in a JSP page), the JSF implementation will retrieve the `ValueBinding for the name binding,` and call `getValue()` on it. If this call returns a non-null `UIComponent` value (because the JavaBean programmatically instantiated and configured a component already), that instance will be added to the component tree that is being created. If the call returns `null`, a new component instance will be created, added to the component tree, and `setValue()` will be called on the `ValueBinding` (which will cause the property on the JavaBean to be set to the newly created component instance).
- When a component tree is recreated during the *Restore View* phase of the request processing lifecycle, for each component that has a `ValueBinding` associated with the name binding, `setValue()` will be called on it, passing the recreated component instance.

Component bindings are often used in conjunction with JavaBeans that are dynamically instantiated via the Managed Bean Creation facility (see Section 5.3.1.2 "Default VariableResolver Implementation"). It is strongly recommend that application developers place managed beans that are pointed at by component binding expressions in "request" scope. This is because placing it in session or application scope would require thread-safety, since `UIComponent` instances depend on running inside of a single thread.

3.1.6 Client Identifiers

Client identifiers are used by JSF implementations, as they decode and encode components, for any occasion when the component must have a client side name. Some examples of such an occasion are:

- to name request parameters for a subsequent request from the JSF-generated page.
- to serve as anchors for client side scripting code.
- to serve as anchors for client side accessibility labels.

```
public String getClientId(FacesContext context);
```

The client identifier is derived from the component identifier (or the result of calling `UIViewRoot.createUniqueId()` if there is not one), and the client identifier of the closest parent component that is a `NamingContainer`. The `Renderer` associated with this component, if any, will then be asked to convert this client identifier to a form appropriate for sending to the client. The value returned from this method must be the same throughout the lifetime of the component instance unless `setId()` is called, in which case it will be recalculated by the next call to `getClientId()`.

3.1.7 Component Tree Manipulation

```
public UIComponent getParent();

public void setParent(UIComponent parent);
```

Components that have been added as children of another component can identify the parent by calling the `getParent` method. For the root node component of a component tree, or any component that is not part of a component tree, `getParent` will return `null`. The `setParent()` method should only be called by the `List` instance returned by calling the `getChildren()` method, or the `Map` instance returned by calling the `getFacets()` method, when child components or facets are being added, removed, or replaced.

```
public List getChildren();
```

Return a mutable `List` that contains all of the child `UIComponent`s for this component instance. The returned `List` implementation must support all of the required and optional methods of the `List` interface, as well as update the parent property of children that are added and removed, as described in the Javadocs for this method.

```
public int getChildCount();
```

A convenience method to return the number of child components for this component. If there are no children, this method must return 0. The method must not cause the creation of a child component list, so it is preferred over calling `getChildren().size()` when there are no children.

3.1.8 Component Tree Navigation

```
public UIComponent findComponent(String expr);
```

Search for and return the UIComponent with an `id` that matches the specified search expression (if any), according to the algorithm described in the Javadocs for this method.

```
public Iterator getFacetsAndChildren();
```

Return an immutable `Iterator` over all of the facets associated with this component (in an undetermined order), followed by all the child components associated with this component (in the order they would be returned by `getChildren()`).

3.1.9 Facet Management

JavaServer Faces supports the traditional model of composing complex components out of simple components via parent-child relationships that organize the entire set of components into a tree, as described in Section 3.1.7 “Component Tree Manipulation”. However, an additional useful facility is the ability to define particular subordinate components that have a specific *role* with respect to the owning component, which is typically independent of the parent-child relationship. An example might be a “data grid” control, where the children represent the columns to be rendered in the grid. It is useful to be able to identify a component that represents the column header and/or footer, separate from the usual child collection that represents the column data.

To meet this requirement, JavaServer Faces components offer support for *facets*, which represent a named collection of subordinate (but non-child) components that are related to the current component by virtue of a unique *facet name* that represents the role that particular component plays. Although facets are not part of the parent-child tree, they participate in request processing lifecycle methods, as described in Section 3.1.13 “Lifecycle Management Methods”.

```
public Map getFacets();
```

Return a mutable Map representing the facets of this UIComponent, keyed by the facet name.

```
public UIComponent getFacet(String name);
```

A convenience method to return a facet value, if it exists, or `null` otherwise. If the requested facet does not exist, no facets `Map` must not be created, so it is preferred over calling `getFacets().get()` when there are no `Facets`.

For easy use of components that use facets, component authors may include type-safe getter and setter methods that correspond to each named facet that is supported by that component class. For example, a component that supports a `header` facet of type `UIHeader` should have methods with signatures and functionality as follows:

```
public UIHeader getHeader() {
   return ((UIHeader) getFacet(“header”);
}

public void setHeader(UIHeader header) {
   getFacets().put(“header”, header);
}
```

3.1.10 Generic Attributes

```
public Map getAttributes();
```

The render-independent characteristics of components are generally represented as JavaBean component properties with getter and setter methods (see Section 3.1.11 “Render-Independent Properties”). In addition, components may also be associated with generic attributes that are defined outside the component implementation class. Typical uses of generic attributes include:

- Specification of render-dependent characteristics, for use by specific `Renderer`s.
- General purpose association of application-specific objects with components.

The attributes for a component may be of any Java programming language object type, and are keyed by attribute name (a String). However, see Section 7.6.2 "State Saving Alternatives and Implications" for implications of your application's choice of state saving method on the classes used to implement attribute values.

Attribute names that begin with `javax.faces` are reserved for use by the JSF specification. Names that begin with `javax` are reserved for definition through the Java Community Process. Implementations are not allowed to define names that begin with `javax`.

The `Map` returned by `getAttributes()` must also support attribute-property transparency, which operates as follows:

- When the `get()` method is called, if the specified attribute name matches the name of a readable JavaBeans property on the component implementation class, the value returned will be acquired by calling the appropriate property getter method, and wrapping Java primitive values (such as int) in their corresponding wrapper classes (such as `java.lang.Integer`) if necessary.
- When the `put()` method is called, if the specified attribute name matches the name of a writable JavaBeans property on the component implementation class, the appropriate property setter method will be called.

3.1.11 Render-Independent Properties

The render-independent characteristics of a user interface component are represented as JavaBean component properties, following JavaBeans naming conventions. Specifically, the method names of the getter and/or setter methods are determined using standard JavaBeans component introspection rules, as defined by `java.beans.Introspector`. The render-independent properties supported by all `UIComponent`s are described in the following table:

Name	Access	Type	Description
`id`	RW	String	The component identifier, as described in Section 3.1.1 "Component Identifiers".
`parent`	RW	`UIComponent`	The parent component for which this component is a child or a facet.
`rendered`	RW	`boolean`	A flag that, if set to `true`, indicates that this component should be processed during all phases of the request processing lifecycle. The default value is "true".

Name	Access	Type	Description
rendererType	RW	String	Identifier of the Renderer instance (from the set of Renderer instances supported by the RenderKit associated with the component tree we are processing. If this property is set, several operations during the request processing lifecycle (such as decode and the encodeXxx family of methods) will be delegated to a Renderer instance of this type. If this property is not set, the component must implement these methods directly.
rendersChildren	RO	boolean	A flag that, if set to true, indicates that this component manages the rendering of all of its children components (so the JSF implementation should not attempt to render them). The default implementation in UIComponentBase delegates this setting to the associated Renderer, if any, and returns false otherwise.
transient	RW	boolean	A flag that, if set to true, indicates that this component must not be included in the state of the component tree. The default implementation in UIComponentBase returns false for this property.

The method names for the render-independent property getters and setters must conform to the design patterns in the JavaBeans specification. See Section 7.6.2 “State Saving Alternatives and Implications” for implications of your application’s choice of state saving method on the classes used to implement property values.

3.1.12 Component Specialization Methods

The methods described in this section are called by the JSF implementation during the various phases of the request processing lifecycle, and may be overridden in a concrete subclass to implement specialized behavior for this component.

```
public boolean broadcast(FacesEvent event) throws
AbortProcessingException;
```

The broadcast() method is called during the common event processing (see Section 2.3 “Common Event Processing”) at the end of several request processing lifecycle phases. For more information about the event and listener model, see Section 3.4 “Event and Listener Model”. Note that it is not necessary to override this method to support additional event types.

```
public void decode(FacesContext context);
```

This method is called during the *Apply Request Values* phase of the request processing lifecycle, and has the responsibility of extracting a new local value for this component from an incoming request. The default implementation in `UIComponentBase` delegates to a corresponding `Renderer`, if the `rendererType` property is set, and does nothing otherwise.

Generally, component writers will choose to delegate decoding and encoding to a corresponding `Renderer` by setting the `rendererType` property (which means the default behavior described above is adequate).

```
public void encodeBegin(FacesContext context) throws IOException;

public void encodeChildren(FacesContext context) throws
IOException;

public void encodeEnd(FacesContext context) throws IOException;
```

These methods are called during the *Render Response* phase of the request processing lifecycle, and have the responsibility of creating the response data for the beginning of this component, this component's children (only called if the `rendersChildren` property of this component is `true`), and the ending of this component, respectively. Typically, this will involve generating markup for the output technology being supported, such as creating an HTML `<input>` element for a `UIInput` component. For clients that support it, the encode methods might also generate client-side scripting code (such as JavaScript), and/or stylesheets (such as CSS). The default implementations in `UIComponentBase` delegate to a corresponding `Renderer`, if the `rendererType` property is `true`, and do nothing otherwise.

Generally, component writers will choose to delegate encoding to a corresponding `Renderer`, by setting the `rendererType` property (which means the default behavior described above is adequate).

```
public void queueEvent(FacesEvent event);
```

Enqueue the specified event for broadcast at the end of the current request processing lifecycle phase. Default behavior is to delegate this to the `queueEvent()` of the parent component, normally resulting in broadcast via the default behavior in the `UIViewRoot` lifecycle methods.

The component author can override any of the above methods to customize the behavior of their component.

3.1.13 Lifecycle Management Methods

The following methods are called by the various phases of the request processing lifecycle, and implement a recursive tree walk of the components in a component tree, calling the component specialization methods described above for each component. These methods are not generally overridden by component writers, but doing so may be useful for some advanced component implementations. See the javadocs for detailed information on these methods.

```
public void processRestoreState(FacesContext context, Object
state);
```

Perform the component tree processing required by the *Restore View* phase of the request processing lifecycle for all facets of this component, all children of this component, and this component itself.

```
public void processDecodes(FacesContext context);
```

Perform the component tree processing required by the *Apply Request Values* phase of the request processing lifecycle for all facets of this component, all children of this component, and this component itself

```
public void processValidators(FacesContext context);
```

Perform the component tree processing required by the *Process Validations* phase of the request processing lifecycle for all facets of this component, all children of this component, and this component itself.

```
public void processUpdates(FacesContext context);
```

Perform the component tree processing required by the Update Model Values phase of the request processing lifecycle for all facets of this component, all children of this component, and this component itself.

```
public void processSaveState(FacesContext context);
```

Perform the component tree processing required by the state saving portion of the *Render Response* phase of the request processing lifecycle for all facets of this component, all children of this component, and this component itself.

3.1.14 Utility Methods

```
protected FacesContext getFacesContext();
```

Return the FacesContext instance for the current request.

```
protected Renderer getRenderer(FacesContext context);
```

Return the `Renderer` that is associated with this `UIComponent`, if any, based on the values of the `family` and `rendererType` properties.

```
protected void addFacesListener(FacesListener listener);

protected void removeFacesListener(FacesListener listener);
```

These methods are used to register and deregister an event listener. They should be called only by a public addXxxListener() method on the component implementation class, which provides typesafe listener registration.

3.2 Component Behavioral Interfaces

In addition to extending `UIComponent`, component classes may also implement one or more of the *behavioral interfaces* described below. Components that implement these interfaces must provide the corresponding method signatures and implement the described functionality.

3.2.1 ActionSource

The `ActionSource` interface defines a way for a component to indicate that wishes to be a source of `ActionEvent` events, including the ability invoke application actions (see Section 7.3 “Application Actions”) via the default `ActionListener` facility (see Section 7.1.1 “ActionListener Property”).

3.2.1.1 Properties

The following render-independent properties are added by the `ActionSource` interface:

Name	Access	Type	Description
`action`	RW	`MethodBinding`	A `MethodBinding` (see Section 5.3.4 "MethodBinding") that must (if non-`null`) point at an action method (see Section 7.3 "Application Actions"). The specified method will be called during the *Apply Request Values* or *Invoke Application* phase of the request processing lifecycle, as described in Section 2.2.5 "Invoke Application".
actionListener	RW	MethodBinding	A `MethodBinding` (see Section 5.3.4 "MethodBinding") that (if non-`null`) must point at a method accepting an `ActionEvent`, with a return type of `void`. Any `ActionEvent` that is sent by this `ActionSource` will be passed to this method along with the `processAction()` method of any registered `ActionListener`s, in either Apply Request Values or Invoke Application phase, depending upon the state of the `immediate` property.
immediate	RW	boolean	A flag indicating that the default `ActionListener` should execute immediately (that is, during the *Apply Request Values* phase of the request processing lifecycle, instead of waiting for *Invoke Application* phase). The default value of this property must be `false`.

3.2.1.2 Methods

`ActionSource` adds no new processing methods.

3.2.1.3 Events

A component implementing `ActionSource` is a source of `ActionEvent` events. There are three important moments in the lifetime of an `ActionEvent`:

- when an the event is *created*
- when the event is *queued* for later processing
- when the listeners for the event are *notified*

`ActionEvent` creation occurs when the system detects that the component implementing `ActionSource` has been activated. For example, a button has been pressed. This happens when the `decode()` processing of the *Apply Request Values* phase of the request processing lifecycle detects that the corresponding user interface control was activated.

`ActionEvent` queueing occurs immediately after the event is created.

Event listeners that have registered an interest in `ActionEvent`s fired by this component (see below) are notified at the end of the *Apply Request Values* or *Invoke Application* phase, depending upon the immediate property of the originating `UICommand`.

`ActionSource` includes the following methods to register and deregister `ActionListener` instances interested in these events. See Section 3.4 “Event and Listener Model” for more details on the event and listener model provided by JSF.

```
public void addActionListener(ActionListener listener);

public void removeActionListener(ActionListener listener);
```

In addition to manually registered listeners, the JSF implementation provides a default `ActionListener` that will process `ActionEvent` events during the *Apply Request Values* or *Invoke Application* phases of the request processing lifecycle. See Section 2.2.5 “Invoke Application” for more information.

3.2.2 NamingContainer

`NamingContainer` is a marker interface. Components that implement `NamingContainer` have the property that, for all of their children that have non-`null` component identifiers, all of those identifiers are unique. This property is enforced by the `renderView()` method on `ViewHandler`. In JSP based applications, it is also enforced by the `UIComponentTag`. Since this is just a marker interface, there are no properties, methods, or events.

NamingContainer defines a public static final character constant, SEPARATOR_CHAR, that is used to separate components of client identifiers, as well as the components of search expressions used by the findComponent() method see (Section 3.1.8 “Component Tree Navigation”). The value of this constant must be a colon character (“:”).

Use of this separator character in client identifiers rendered by Renderers can cause problems with CSS stylesheets that attach styles to a particular client identifier. For the Standard HTML RenderKit, this issue can be worked around by using the style attribute to specify CSS style values directly, or the styleClass attribute to select CSS styles by class rather than by identifier.

3.2.3 StateHolder

The StateHolder interface is implemented by UIComponent, Converter, FacesListener, and Validator classes that need to save their state between requests. UIComponent implements this interface to denote that components have state that must be saved and restored between requests.

3.2.3.1 Properties

The following render-independent properties are added by the StateHolder interface:

Name	Access	Type	Description
transient	RW	boolean	A flag indicating whether this instance has decided to opt out of having its state information saved and restored. The default value for all standard component, converter, and validator classes that implement StateHolder must be false.

3.2.3.2 Methods

Any class implementing `StateHolder` must implement both the `saveState()` and `restoreState()` methods, since these two methods have a tightly coupled contract between themselves. In other words, if there is an inheritance hierarchy, it is not permissible to have the `saveState()` and `restoreState()` methods reside at different levels of the hierarchy.

```
public Object saveState(FacesContext context);
public void restoreState(FacesContext context, Object state)
throws IOException;
```

Gets or restores the state of the instance as a `Serializable Object`.

If the class that implements this interface has references to Objects which also implement `StateHolder` (such as a `UIComponent` with a converter, event listeners, and/or validators) these methods must call the `saveState()` or `restoreState()` method on all those instances as well.

Any class implementing `StateHolder` must have a public no-args constructor.

If the state saving method is server, these methods may not be called.

If the class that implements this interface has references to Objects which do not implement `StateHolder`, these methods must ensure that the references are preserved. For example, consider class `MySpecialComponent`, which implements `StateHolder`, and keeps a reference to a helper class, `MySpecialComponentHelper`, which does not implement `StateHolder`. `MySpecialComponent.saveState()` must save enough information about `MySpecialComponentHelper`, so that when `MySpecialComponent.restoreState()` is called, the reference to `MySpecialComponentHelper` can be restored. The return from `saveState()` must be `Serializable`.

Since all of the standard user interface components listed in Chapter 4 “Standard User Interface Components” extend from `UIComponent`, they all implement the `StateHolder` interface. In addition, the standard `Converter` and `Validator` classes that require state to be saved and restored also implement `StateHolder.`

3.2.3.3 Events

`StateHolder` does not originate any standard events.

3.2.4 ValueHolder

`ValueHolder` is an interface that may be implemented by any concrete `UIComponent` that wishes to support a local value, as well as access data in the model tier via a *`value binding expression`*, and support conversion between `String` and the model tier data's native data type.

3.2.4.1 Properties

The following render-independent properties are added by the `ValueHolder` interface:

Name	Access	Type	Description
converter	RW	Converter	The `Converter` (if any) that is registered for this UIComponent.
`value`	RW	`Object`	First consult the local value property of this component. If non-`null` return it. If the local value property is `null`, see if we have a `ValueBinding` for the value property. If so, return the result of evaluating the property, otherwise return `null`.
localValue	RO	Object	allows any value set by calling `setValue()` to be returned, without potentially evaluating a `ValueBinding` the way that `getValue()` will do

Like nearly all component properties, the `value` property may have a value binding expression (see Section 3.1.4 "Value Binding Expressions") associated with it. If present (and if there is no `value` set directly on this component), such an expression is utilized to retrieve a value dynamically from a model tier object during *Render Response Phase* of the request processing lifecycle. In addition, for input components, the value binding is used during *Update Model Values* phase (on the subsequent request) to push the possibly updated component value back to the model tier object.

The `Converter` property is used to allow the component to know how to convert the model type from the `String` format provided by the Servlet API to the proper type in the model tier.

3.2.4.2 Methods

ValueHolder adds no methods.

3.2.4.3 Events

`ValueHolder` does not originate any standard events.

3.2.5 EditableValueHolder

The `EditableValueHolder` interface (extends `ValueHolder`, see *Section 3.2.4 "ValueHolder"*) describes additional features supported by editable components, including `ValueChangeEvents` and `Validators`.

3.2.5.1 Properties

The following render-independent properties are added by the `EditableValueHolder` interface:

Name	Access	Type	Description
immediate	RW	boolean	Flag indicating that conversion and validation of this component's value should occur during *Apply Request Values* phase instead of *Process Validations* phase.
localValueSet	RW	boolean	Flag indicating whether the `value` property has been set.
required	RW	boolean	Is the user required to provide a non-empty value for this component? Default value must be `false`.
submittedValue	RW	`Object`	The submitted, unconverted, value of this component. This property should only be set by the decode() method of this component, or its corresponding Renderer, or by the validate method of this component. This property should only be read by the validate() method of this component.

Name	Access	Type	Description
valid	RW	boolean	A flag indicating whether the local value of this component is valid (that is, no conversion error or validation error has occurred).
validator	RW	MethodBinding	A `MethodBinding` that (if not null) must point at a method accepting a `FacesContext` and a `UIInput`, with a return type of `void`. This method will be called during *Process Validations* phase, after any validators that are externally registered.
valueChangeListener	RW	MethodBinding	A MethodBinding that (if not null) must point at a method that accepts a `ValueChangeEvent`, with a return type of `void`. The specified method will be called during the *Process Validations* phase of the request processing lifecycle, after any externally registered `ValueChangeListener`s.

3.2.5.2 Methods

The following methods support the validation functionality performed during the *Process Validations* phase of the request processing lifecycle:

```
public void addValidator(Validator validator);

public void removeValidator(Validator validator);
```

The `addValidator()` and `removeValidator()` methods are used to register and deregister additional external `Validator` instances that will be used to perform correctness checks on the local value of this component.

If the `validator` property is not null, the method it points at must be called by the `processValidations()` method, after the `validate()` method of all registered `Validator`s is called.

3.2.5.3 Events

`EditableValueHolder` is a source of `ValueChangeEvent` events, which are emitted when the `validate()` processing of the *Process Validations* phase of the request processing lifecycle determines that the previous value of this component differs from the current value, and all validation checks have passed (i.e. the `valid` property of this component is still true). It includes the following methods to register and deregister `ValueChangeListener` instances interested in these events. See Section 3.4 “Event and Listener Model” for more details on the event and listener model provided by JSF.

```
public void addValueChangeListener(ValueChangeListener listener);

public void removeValueChangeListener(ValueChangeListener
listener);
```

In addition to the above listener registration methods, If the `valueChangeListener` property is not `null`, the method it points at must be called by the `broadcast()` method, after the `processValueChange()` method of all registered `ValueChangeListener`s is called.

3.3 Conversion Model

This section describes the facilities provided by JavaServer Faces to support type conversion between server-side Java objects and their (typically String-based) representation in presentation markup.

3.3.1 Overview

A typical web application must constantly deal with two fundamentally different viewpoints of the underlying data being manipulated through the user interface:

- The *model* view—Data is typically represented as Java programming language objects (often JavaBeans components), with data represented in some native Java programming language datatype. For example, date and time values might be represented in the model view as instances of `java.util.Date`.
- The *presentation* view—Data is typically represented in some form that can be perceived or modified by the user of the application. For example, a date or type value might be represented as a text string, as three text strings (one each for month/date/year or one each for hour/minute/second), as a calendar control, associated with a spin control that lets you increment or decrement individual elements of the date or time with a single mouse click, or in a variety of other ways. Some presentation views may depend on the preferred language or locale of the user (such as the commonly used mm/dd/yy and dd/mm/yy date formats, or the variety of punctuation characters in monetary amount presentations for various currencies).

To transform data formats between these views, JavaServer Faces provides an ability to plug-in an optional `Converter` for each `ValueHolder`, which has the responsibility of converting the internal data representation between the two views. The application developer attaches a particular `Converter` to a particular `ValueHolder` by calling `setConverter`, passing an instance of the particular converter. A `Converter` implementation may be acquired from the `Application` instance (see Section 7.1.10 "Object Factories") for your application.

3.3.2 Converter

JSF provides the `javax.faces.convert.Converter` interface to define the behavioral characteristics of a `Converter`. Instances of implementations of this interface are either identified by a *converter identifier*, or by a class for which the

Converter class asserts that it can perform successful conversions, which can be registered with, and later retrieved from, an Application, as described in Section 7.1.10 “Object Factories”.

Often, a Converter will be an object that requires no extra configuration information to perform its responsibilities. However, in some cases, it is useful to provide configuration parameters to the Converter (such as a java.text.DateFormat pattern for a Converter that supports java.util.Date model objects). Such configuration information will generally may be provided via JavaBeans properties on the Converter instance.

Converter implementations should be programmed so that the conversions they perform are symmetric. In other words, if a model data object is converted to a String (via a call to the getAsString method), it should be possible to call getAsObject and pass it the converted String as the value parameter, and return a model data object that is semantically equal to the original one. In some cases, this is not possible. For example, a converter that uses the formatting facilities provided by the java.text.Format class might create two adjacent integer numbers with no separator in between, and in this case the Converter could not tell which digits belong to which number.

For UIInput and UIOutput components that wish to explicitly select a Converter to be used, a new Converter instance of the appropriate type must be created, optionally configured, and registered on the component by calling setConverter()[1]. Otherwise, the JSF implementation will automatically create new instances based on the data type being converted, if such Converter classes have been registered. In either case, Converter implementations need not be threadsafe, because they will be used only in the context of a single request processing thread.

The following two method signatures are defined by the Converter interface:

```
public Object getAsObject(FacesContext context, UIComponent
component, String value) throws ConverterException;
```

This method is used to convert the presentation view of a component’s value (typically a String that was received as a request parameter) into the corresponding model view. It is called during the *Apply Request Values* phase of the request processing lifecycle.

```
public String getAsString(FacesContext context, UIComponent
component, Object value) throws ConverterException;
```

1. In a JSP environment, these steps are performed by a custom tag extending ConverterTag.

This method is used to convert the model view of a component's value (typically some native Java programming language class) into the presentation view (typically a String that will be rendered in some markup language. It is called during the *Render Response* phase of the request processing lifecycle.

3.3.3 Standard Converter Implementations

JSF provides a set of standard `Converter` implementations. A JSF implementation must register the `DateTime` and `Number` converters by name with the `Application` instance for this web application, as described in the table below. This ensures that the converters are available for subsequent calls to `Application.createConverter()`. Each concrete implementation class must define a static final String constant `CONVERTER_ID` whose value is the standard converter id under which this Converter is registered.

The following converter id values must be registered to create instances of the specified Converter implementation classes:

- `javax.faces.BigDecimal` -- An instance of `javax.faces.convert.BigDecimalConverter` (or a subclass of this class).
- `javax.faces.BigInteger` -- An instance of `javax.faces.convert.BigIntegerConverter` (or a subclass of this class).
- `javax.faces.Boolean` -- An instance of `javax.faces.convert.BooleanConverter` (or a subclass of this class).
- `javax.faces.Byte` -- An instance of `javax.faces.convert.ByteConverter` (or a subclass of this class).
- `javax.faces.Character` -- An instance of `javax.faces.convert.CharacterConverter` (or a subclass of this class).
- `javax.faces.DateTime` -- An instance of `javax.faces.convert.DateTimeConverter` (or a subclass of this class).
- `javax.faces.Double` -- An instance of `javax.faces.convert.DoubleConverter` (or a subclass of this class).
- `javax.faces.Float` -- An instance of `javax.faces.convert.FloatConverter` (or a subclass of this class).
- `javax.faces.Integer` -- An instance of `javax.faces.convert.IntegerConverter` (or a subclass of this class).
- `javax.faces.Long` -- An instance of `javax.faces.convert.LongConverter` (or a subclass of this class).
- `javax.faces.Number` -- An instance of `javax.faces.convert.NumberConverter` (or a subclass of this class).
- `javax.faces.Short` -- An instance of `javax.faces.convert.ShortConverter` (or a subclass of this class).

See the Javadocs for these classes for a detailed description of the conversion operations they perform, and the configuration properties that they support.

A JSF implementation must register converters for all of the following classes using the by-type registration mechanism:

- `java.lang.Boolean`, and `java.lang.Boolean.TYPE` -- An instance of `javax.faces.convert.BooleanConverter` (or a subclass of this class).
- `java.lang.Byte`, and `java.lang.Byte.TYPE` -- An instance of `javax.faces.convert.ByteConverter` (or a subclass of this class).
- `java.lang.Character`, and `java.lang.Character.TYPE` -- An instance of `javax.faces.convert.CharacterConverter` (or a subclass of this class).
- `java.lang.Double`, and `java.lang.Double.TYPE` -- An instance of `javax.faces.convert.DoubleConverter` (or a subclass of this class).
- `java.lang.Float`, and `java.lang.Float.TYPE` -- An instance of `javax.faces.convert.FloatConverter` (or a subclass of this class).
- `java.lang.Integer`, and `java.lang.Integer.TYPE` -- An instance of `javax.faces.convert.IntegerConverter` (or a subclass of this class).
- `java.lang.Long`, and `java.lang.Long.TYPE` -- An instance of `javax.faces.convert.LongConverter` (or a subclass of this class).
- `java.lang.Short`, and `java.lang.Short.TYPE` -- An instance of `javax.faces.convert.ShortConverter` (or a subclass of this class).

See the Javadocs for these classes for a detailed description of the conversion operations they perform, and the configuration properties that they support.

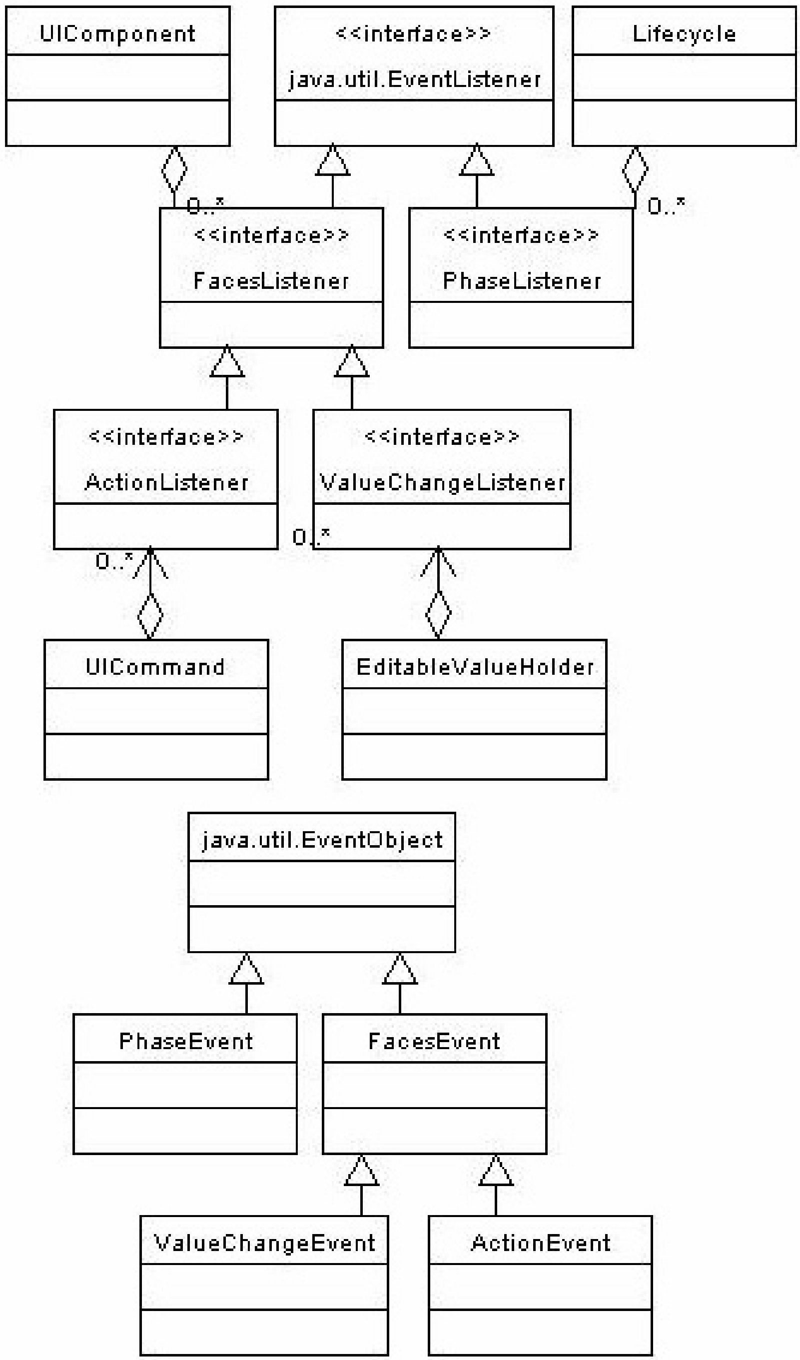
3.4 Event and Listener Model

This section describes how JavaServer Faces provides support for generating and handling user interface events.

3.4.1 Overview

JSF implements a model for event notification and listener registration based on the design patterns in the *JavaBeans Specification*, version 1.0.1. This is similar to the approach taken in other user interface toolkits, such as the Swing Framework included in the JDK.

A `UIComponent` subclass may choose to emit *events* that signify significant state changes, and broadcast them to *listeners* that have registered an interest in receiving events of the type indicated by the event's implementation class. At the end of several phases of the request processing lifecycle, the JSF implementation will broadcast all of the events that have been queued to interested listeners.The following UML class diagram illustrates the key players in the event model.


UIComponent
<<interface>>
java.util.EventListener
Lifecycle
0..*
<<interface>>
FacesListener
<<interface>>
PhaseListener
0..*
<<interface>>
ActionListener
<<interface>>
ValueChangeListener
0..*
0..*
UICommand
EditableValueHolder
java.util.EventObject
PhaseEvent
FacesEvent
ValueChangeEvent
ActionEvent

3.4.2 Event Classes

All events that are broadcast by JSF user interface components must extend the `javax.faces.event.FacesEvent` abstract base class. The parameter list for the constructor(s) of this event class must include a `UIComponent`, which identifies the component from which the event will be broadcast to interested listeners. The source component can be retrieved from the event object itself by calling `getComponent`. Additional constructor parameters and/or properties on the event class can be used to relay additional information about the event.

In conformance to the naming patterns defined in the *JavaBeans Specification*, event classes typically have a class name that ends with `Event`. It is recommended that application event classes follow this naming pattern as well.

The component that is the source of a FacesEvent can be retrieved via this method:

```
public UIComponent getComponent();
```

`FacesEvent` has a `phaseId` property (of type `PhaseId`, see Section 3.4.4 "Phase Identifiers") used to identify the request processing lifecycle phase after which the event will be delivered to interested listeners.

```
public PhaseId getPhaseId();

public void setPhaseId(PhaseId phaseId);
```

If this property is set to PhaseId.ANY_PHASE (which is the default), the event will be delivered at the end of the phase in which it was enqueued.

To facilitate general management of event listeners in JSF components, a `FacesEvent` implementation class must support the following methods:

```
public abstract boolean isAppropriateListener(FacesListener
listener);

public abstract void processListener(FacesListener listener);
```

The `isAppropriateListener()` method returns true if the specified `FacesListener` is a relevant receiver of this type of event. Typically, this will be implemented as a simple "instanceof" check to ensure that the listener class implements the `FacesListener` subinterface that corresponds to this event class

The `processListener()` method must call the appropriate event processing method on the specified listener. Typically, this will be implemented by casting the listener to the corresponding `FacesListener` subinterface and calling the appropriate event processing method, passing this event instance as a parameter.

```
public void queue();
```

The above convenience method calls the `queueEvent()` method of the source `UIComponent` for this event, passing this event as a parameter.

JSF includes two standard `FacesEvent` subclasses, which are emitted by the corresponding standard `UIComponent` subclasses described in the following chapter.

- `ActionEvent`—Emitted by a `UICommand` component when the user activates the corresponding user interface control (such as a clicking a button or a hyperlink).
- `ValueChangeEvent`—Emitted by a `UIInput` component (or appropriate subclass) when a new local value has been created, and has passed all validations.

3.4.3 Listener Classes

For each event type that may be emitted, a corresponding listener interface must be created, which extends the `javax.faces.event.FacesListener` interface. The method signature(s) defined by the listener interface must take a single parameter, an instance of the event class for which this listener is being created. A listener implementation class will implement one or more of these listener interfaces, along with the event handling method(s) specified by those interfaces. The event handling methods will be called during event broadcast, one per event.

In conformance to the naming patterns defined in the *JavaBeans Specification*, listener interfaces have a class name based on the class name of the event being listened to, but with the word `Listener` replacing the trailing `Event` of the event class name (thus, the listener for a `FooEvent` would be a `FooListener`). It is recommended that application event listener interfaces follow this naming pattern as well.

Corresponding to the two standard event classes described in the previous section, JSF defines two standard event listener interfaces that may be implemented by application classes:

- `ActionListener`—a listener that is interested in receiving `ActionEvent` events.
- `ValueChangeListener`—a listener that is interested in receiving `ValueChangeEvent` events.

3.4.4 Phase Identifiers

As described in Section 2.3 "Common Event Processing", event handling occurs at the end of several phases of the request processing lifecycle. In addition, a particular event must indicate, through the value it returns from the `getPhaseId()` method, the phase in which it wishes to be delivered. This indication is done by returning an instance of `javax.faces.event.PhaseId`. The class defines a typesafe enumeration of all the legal values that may be returned by `getPhaseId()`. In addition, a special value (`PhaseId.ANY_PHASE`) may be returned to indicate that this event wants to be delivered at the end of the phase in which it was queued.

3.4.5 Listener Registration

A concrete `UIComponent` subclass that emits events of a particular type must include public methods to register and deregister a listener implementation. In order to be recognized by development tools, these listener methods must follow the naming patterns defined in the *JavaBeans Specification*. For example, for a component that emits `FooEvent` events, to be received by listeners that implement the `FooListener` interface, the method signatures (on the component class) must be:

```
public void addFooListener(FooListener listener);

public FooListener[] getFooListeners();

public void removeFooListener(FooListener listener);
```

The application (or other components) may register listener instances at any time, by calling the appropriate add method. The set of listeners associated with a component is part of the state information that JSF saves and restores. Therefore, listener implementation classes must have a public zero-argument constructor, and may implement `StateHolder` (see Section 3.2.3 "StateHolder") if they have internal state information that needs to be saved and restored.

The `UICommand` and `UIInput` standard component classes include listener registration and deregistration methods for event listeners associated with the event types that they emit. The `UIInput` methods are also inherited by `UIInput` subclasses, including `UISelectBoolean`, `UISelectMany`, and `UISelectOne`.

3.4.6 Event Queueing

During the processing being performed by any phase of the request processing lifecycle, events may be created and queued by calling the `queueEvent()` method on the source `UIComponent` instance, or by calling the `queue()` method on the `FacesEvent` instance itself. As described in Section 2.3 “Common Event Processing”, at the end of certain phases of the request processing lifecycle, any queued events will be broadcast to interested listeners in the order that the events were originally queued.

Deferring event broadcast until the end of a request processing lifecycle phase ensures that the entire component tree has been processed by that state, and that event listeners all see the same consistent state of the entire tree, no matter when the event was actually queued.

3.4.7 Event Broadcasting

As described in Section 2.3 “Common Event Processing”, at the end of each request processing lifecycle phase that may cause events to be queued, the lifecycle management method of the `UIViewRoot` component at the root of the component tree will iterate over the queued events and call the `broadcast()` method on the source component instance to actually notify the registered listeners. See the Javadocs of the `broadcast()` method for detailed functional requirements.

During event broadcasting, a listener processing an event may:

- Examine or modify the state of any component in the component tree.
- Add or remove components from the component tree.
- Add messages to be returned to the user, by calling `addMessage` on the `FacesContext` instance for the current request.
- Queue one or more additional events, from the same source component or a different one, for processing during the current lifecycle phase.
- Throw an `AbortProcessingException`, to tell the JSF implementation that no further broadcast of this event, or any further events, should take place.
- Call `renderResponse()` on the `FacesContext` instance for the current request. This tells the JSF implementation that, when the current phase of the request processing lifecycle has been completed, control should be transferred to the *Render Response* phase.
- Call `responseComplete()` on the `FacesContext` instance for the current request. This tells the JSF implementation that, when the current phase of the request processing lifecycle has been completed, processing for this request should be terminated (because the actual response content has been generated by some other means).

3.5 Validation Model

This section describes the facilities provided by JavaServer Faces for validating user input.

3.5.1 Overview

JSF supports a mechanism for registering zero or more *validators* on each `EditableValueHolder` component in the component tree. A validator's purpose is to perform checks on the local value of the component, during the *Process Validations* phase of the request processing lifecycle. In addition, a component may implement internal checking in a `validate` method that is part of the component class.

3.5.2 Validator Classes

A validator must implement the `javax.faces.validator.Validator` interface, which contains a `validate` method signature. General purpose validators may require configuration values in order to define the precise check to be performed. For example, a validator that enforces a maximum length might wish to support a configurable length limit. Such configuration values are typically implemented as JavaBeans component properties, and/or constructor arguments, on the `Validator` implementation class. In addition, a validator may elect to use generic attributes of the component being validated for configuration information.

JSF includes implementations of several standard validators, as described in Section 3.5.5 "Standard Validator Implementations".

3.5.3 Validation Registration

The `EditableValueHolder` interface (implemented by `UIInput`) includes an `addValidator` method to register an additional validator for this component, and a `removeValidator` method to remove an existing registration, as well as the ability to add a `MethodBinding` that points to a method that adheres to the `validate` signature in the `Validator` interface.

The application (or other components) may register validator instances at any time, by calling the `addValidator` method. The set of validators associated with a component is part of the state information that JSF saves and restores. Validators that wish to have configuration properties saved and restored must also implement `StateHolder` (see Section 3.2.3 “StateHolder”).

3.5.4 Validation Processing

During the *Process Validations* phase of the request processing lifecycle (as described in Section 2.2.3 “Process Validations”), the JSF implementation will ensure that the `validate()` method of each registered `Validator`, the method referenced by the `validator` property (if any), and the `validate()` method of the component itself, is called for each `EditableValueHolder` component in the component tree, regardless of the validity state of any of the components in the tree. The responsibilities of each `validate()` method include:

- Perform the check for which this validator was registered.
- If violation(s) of the correctness rules are found, create a `FacesMessage` instance describing the problem, and create a `ValidatorException` around it, and throw the `ValidatorException`. The `EditableValueHolder` on which this validation is being performed will catch this exception, set `valid` to `false` for that instance, and cause the message to be added to the `FacesContext`.

In addition, a `validate()` method may:

- Examine or modify the state of any component in the component tree.
- Add or remove components from the component tree.
- Queue one or more events, from the same component or a different one, for processing during the current lifecycle phase.

The render-independent property `required` is a shorthand for the function of a “required” validator. If the value of this property is true and the component has no value, the component is marked invalid and a message is added to the `FacesContext` instance. See Section 2.5.2.4 “Localized Application Messages” for details on the message.

3.5.5 Standard Validator Implementations

JavaServer Faces defines a standard suite of `Validator` implementations that perform a variety of commonly required checks. In addition, component writers, application developers, and tool providers will often define additional `Validator` implementations that may be used to support component-type-specific or application-specific constraints. These implementations share the following common characteristics:

- Standard `Validators` accept configuration information as either parameters to the constructor that creates a new instance of that `Validator`, or as JavaBeans component properties on the `Validator` implementation class.
- To support internationalization, `FacesMessage` instances should be created. The message identifiers for such standard messages are also defined by manifest String constants in the implementation classes. It is the user's responsibility to ensure the content of a `FacesMessage` instance is properly localized, and appropriate parameter substitution is performed, perhaps using `java.text.MessageFormat`.
- Unless otherwise specified, components with a `null` local `value` cause the validation checking by this `Validator` to be skipped. If a component should be required to have a non-`null` value, a component attribute with the name `required` and the value `true` must be added to the component in order to enforce this rule.
- Concrete Validator implementations must define a public static final String constant VALIDATOR_ID, whose value is the standard identifier under which the JSF implementation must register this instance (see below).

Please see Section 2.5.2.4 "Localized Application Messages" for the list of message identifiers.

The following standard `Validator` implementations (in the `javax.faces.validator` package) are provided:

- `DoubleRangeValidator`—Checks the local value of a component, which must be of any numeric type, against specified maximum and/or minimum values. Standard identifier is "javax.faces.DoubleRange".
- `LengthValidator`—Checks the length (i.e. number of characters) of the local value of a component, which must be of type `String`, against maximum and/or minimum values. Standard identifier is "javax.faces.Length".
- `LongRangeValidator`—Checks the local value of a component, which must be of any numeric type convertible to `long`, against maximum and/or minimum values. Standard identifier is "javax.faces.LongRange".

CHAPTER 4

Standard User Interface Components

In addition to the abstract base class UIComponent and the abstract base class UIComponentBase, described in the previous chapter, JSF provides a number of concrete user interface component implementation classes that cover the most common requirements. In addition, component writers will typically create new components by subclassing one of the standard component classes (or the UIComponentBase class). It is anticipated that the number of standard component classes will grow in future versions of the JavaServer Faces specification.

Each of these classes defines the render-independent characteristics of the corresponding component as JavaBeans component properties. Some of these properties may be *value binding expressions* that indirectly point to values related to the current request, or to the properties of model data objects that are accessible through request-scope, session-scope, or application-scope attributes. In addition, the rendererType property of each concrete implementation class is set to a defined value, indicating that decoding and encoding for this component will (by default) be delegated to the corresponding Renderer.

4.1 Standard User Interface Components

This section documents the features and functionality of the standard UIComponent classes and implementations that are included in JavaServer Faces.

The implementation for each standard UIComponent class must specify two public static final String constant values:

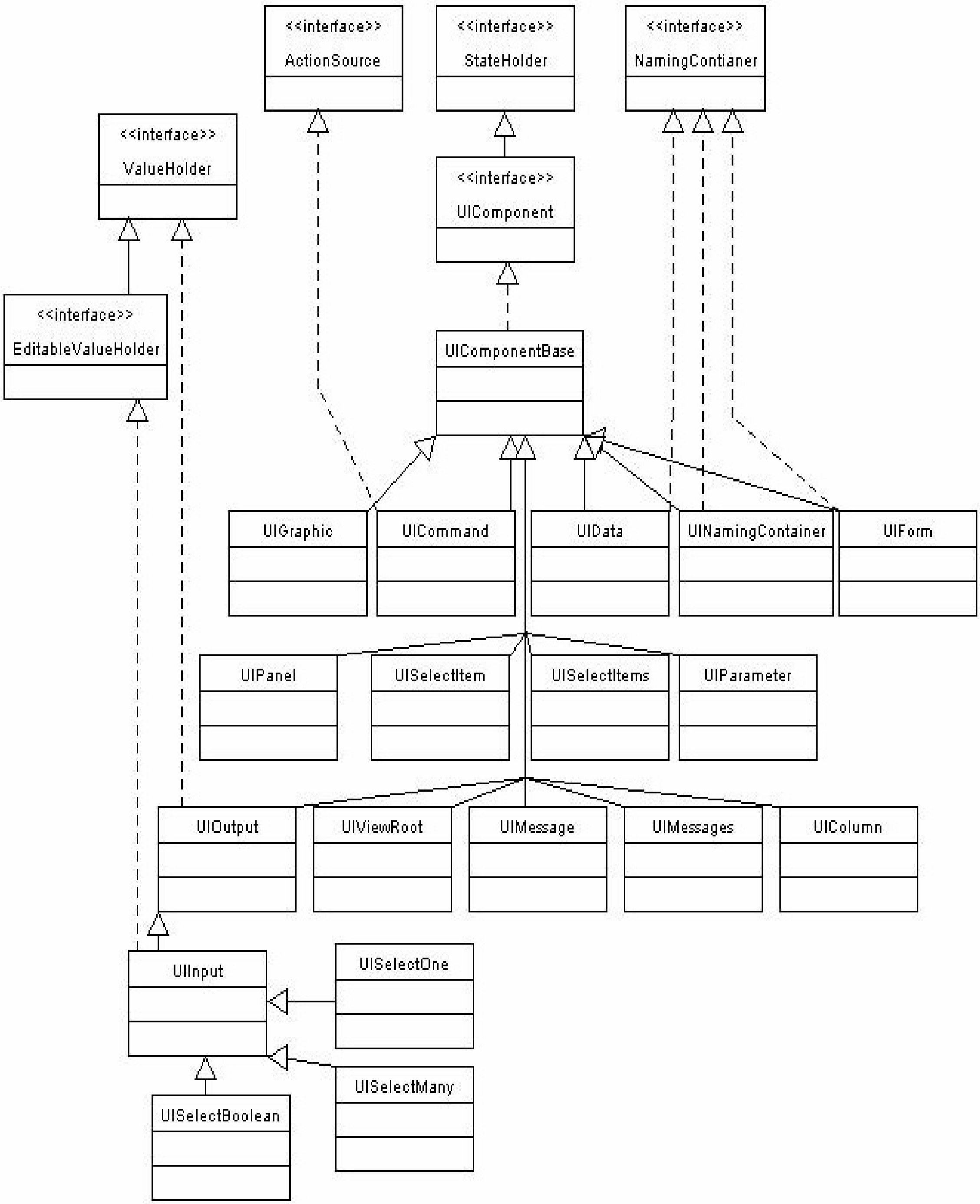
- COMPONENT_TYPE -- The standard component type identifier under which the corresponding component class is registered with the Application object for this application. This value may be used as a parameter to the createComponent() method.

- COMPONENT_FAMILY -- The standard component family identifier used to select an appropriate Renderer for this component.

For all render-independent properties in the following sections (except for id, scope, and var) the value may either be a literal, or it may come from a value binding expression. Please see Chapter 5 “Value Binding Expressions” for more information.

The following UML class diagram shows the classes and interfaces in the package `javax.faces.component.`

FIGURE 4-1 The `javax.faces.component` package



4.1.1 UIColumn

`UIColumn` (extends `UIComponentBase`) is a component that represents a single column of data with a parent `UIData` component. The child components of a `UIColumn` will be processed once for each row in the data managed by the parent `UIData`.

4.1.1.1 Component Type

The standard component type for `UIColumn` components is "javax.faces.Column".

4.1.1.2 Properties

`UIColumn` adds the following render-independent properties:

Name	Access	Type	Description
`footer`	RW	`UIComponent`	Convenience methods to get and set the "footer" facet for this component.
`header`	RW	UIComponent	Convenience methods to get and set the "header" facet for this component.

`UIColumn` specializes the behavior of render-independent properties inherited from the parent class as follows:

- The default value of the `family` property must be set to "javax.faces.Column".
- The default value of the `rendererType` property must be set to `null`.

4.1.1.3 Methods

`UIColumn` adds no new processing methods.

4.1.1.4 Events

`UIColumn` adds no new event handling methods.

4.1.2 UICommand

`UICommand` (extends `UIComponentBase;` implements `ActionSource`) is a control which, when activated by the user, triggers an application-specific "command" or "action." Such a component is typically rendered as a push button, a menu item, or a hyperlink.

4.1.2.1 Component Type

The standard component type for `UICommand` components is "javax.faces.`Command`".

4.1.2.2 Properties

`UICommand` adds the following render-independent properties.

Name	Access	Type	Description
`value`	RW	`Object`	The value of this component, normally used as a label.

See Section 3.2.1 "ActionSource" for information about properties introduced by the implemented classes.

`UICommand` components specialize the behavior of render-independent properties inherited from the parent class as follows:

- The default value of the `family` property must be set to "javax.faces.Command".
- The default value of the `rendererType` property must be set to "javax.faces.Button".

4.1.2.3 Methods

`UICommand` adds no new processing methods. See Section 3.2.1 "ActionSource" for information about methods introduced by the implemented classes.

4.1.2.4 Events

`UICommand adds no new event processing methods.` See Section 3.2.1 "ActionSource" for information about event handling introduced by the implemented classes.

4.1.3 UIData

UIData (extends UIComponentBase; implements NamingContaier) is a component that represents a data binding to a collection of data objects represented by a DataModel instance (see Section 4.2.1 “DataModel”). Only children of type UIColumn should be processed by renderers associated with this component.

4.1.3.1 Component Type

The standard component type for UIData components is “javax.faces.Data”

4.1.3.2 Properties

UIData adds the following render-independent properties.

Name	Access	Type	Description
first	RW	int	One-relative row number of the first row in the underlying data model to be displayed, or zero to start at the beginning of the data model.
footer	RW	UIComponent	Convenience methods to get and set the “footer” facet for this component.
header	RW	UIComponent	Convenience methods to get and set the “header” facet for this component.
rowCount	RO	int	The number of rows in the underlying DataModel, which can be -1 if the number of rows is unknown.
rowAvailable	RO	boolean	Return true if there is row data available for the currently specified rowIndex; else return false.
rowData	RO	Object	The data object representing the data for the currently selected rowIndex value.
rowIndex	RW	int	Zero-relative index of the row currently being accessed in the underlying DataModel, or -1 for no current row. See below for further information.

Name	Access	Type	Description
rows	RW	int	The number of rows (starting with the one identified by the `first` property) to be displayed, or zero to display the entire set of available rows.
value	RW	Object	The `DataModel` instance representing the data to which this component is bound, or a collection of data for which a `DataModel` instance is synthesized. See below for more information.
var	RW	String	The request-scope attribute (if any) under which the data object for the current row will be exposed when iterating.

See Section 3.2.2 “NamingContainer” for information about properties introduced by the implemented classes.

`UIData` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to “javax.faces.Data”.
- The default value of the `rendererType` property must be set to “`javax.faces.Table`”.

The current value identified by the `value` property is normally of type `DataModel`. However, a `DataModel` wrapper instance must automatically be provided by the JSF implementation if the current value is of one of the following types:

- `java.util.List`
- Array of `java.util.Object`
- `java.sql.ResultSet` (which therefore also supports `javax.sql.RowSet`)
- `javax.servlet.jsp.jstl.sql.Result`
- Any other Java object is wrapped by a `DataModel` instance with a single row.

Convenience implementations of `DataModel` are provided in the `javax.faces.model` package for each of the above (see Section 4.2.1.4 “Concrete Implementations”), and must be used by the `UIData` component to create the required `DataModel` wrapper.

4.1.3.3 Methods

`UIData` adds no new processing methods. See Section 3.2.2 “NamingContainer” for information about methods introduced by the implemented classes.

UIData specializes the behavior of the `getClientId()` method inherited from its parent, in order to create a client identifier that includes the current rowIndex value (if it is not -1). Because `UIData` is a `NamingContainer`, this makes it possible for rendered client identifiers of child components to be row-specific.

`UIData` specializes the behavior of the `queueEvent()` method inherited from its parent, to wrap the specified event (bubbled up from a child component) in a private wrapper containing the current rowIndex value, so that this rowIndex can be reset when the event is later broadcast.

`UIData` specializes the behavior of the `broadcast()` method to unwrap the private wrapper (if this event was wrapped), and call `setRowIndex()` to re-establish the context in which the event was queued, followed by delivery of the event.

`UIData` specializes the behavior of the `processDecodes()`, `processValidators()`, and `processUpdates()` methods inherited from its parent as follows:

- For each of these methods, the `UIData` implementation must iterate over each row in the underlying data model, starting with the row identified by the `first` property, for the number of rows indicated by the `rows` property, by calling the `setRowIndex()` method.
- When iteration is complete, set the `rowIndex` property of this component, and of the underlying `DataModel`, to zero, and remove any request attribute exposed via the `var` property.

4.1.3.4 Events

`UIData` adds no new event handling methods. SeeSection 3.2.2 “NamingContainer” for information about event handling introduced by the implemented classes.

4.1.4 UIForm

UIForm (extends UIComponentBase; implements NamingContainer) is a component that represents an input form to be presented to the user, and whose child components (among other things) represent the input fields to be included when the form is submitted.

The encodeEnd() method of the renderer for UIForm must call ViewHandler.writeState() *before* writing out the markup for the closing tag of the form.This allows the state for multiple forms to be saved.

4.1.4.1 Component Type

The standard component type for UIForm components is "javax.faces.Form".

4.1.4.2 Properties

UIForm adds no new render-independent properties.

UIForm specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the family property must be set to "javax.faces.Form".
- The default value of the rendererType property must be set to "javax.faces.Form".

4.1.4.3 Methods.

```
public boolean isSubmitted();
public void setSubmitted(boolean submitted)
```

The setSubmitted() method of each UIForm instance in the view must be called during the *Apply Request Values* phase of the request processing lifecycle, during the processing performed by the UIComponent.decode() method. If this UIForm instance represents the form actually being submitted on this request, the parameter must be set to true; otherwise, it must be set to false. The standard implementation of UIForm delegates the responsibility for calling this method to the Renderer associated with this instance.

The value of a `UIForm's` `submitted` property must not be saved as part of its state.

```
public void processDecodes(FacesContext context);
```

Override `UIComponent.processDecodes()` to ensure that the `submitted` property is set for this component. If the `submitted` property decodes to false, do not process the children and return immediately.

```
public void processValidators(FacesContext context);
public void processUpdates(FacesContext context);
```

Override `processValidators()` and `processUpdates()` to ensure that the children of this `UIForm` instance are only processed if `isSubmitted()` returns true.

```
public void saveState(FacesContext context);
```

The `saveState()` method of UIForm must call `setSubmitted(false)` before calling `super.saveState()`.

4.1.4.4 Events

`UIForm` adds no new event handling methods.

4.1.5 UIGraphic

`UIGraphic` (extends `UIComponentBase`) is a component that displays a graphical image to the user. The user cannot manipulate this component; it is for display purposes only.

4.1.5.1 Component Type

The standard component type for `UIGraphic` components is “javax.faces.`Graphic`”.

4.1.5.2 Properties

The following render-independent properties are added by the UIGraphic component:

Name	Access	Type	Description
`url`	RW	`String`	The URL of the image to be displayed. If this URL begins with a `/` character, it is assumed to be relative to the context path of the current web application. This property is a typesafe alias for the `value` property, so that the actual URL to be used can be acquired via a value binding expression.
`value`	RW	`Object`	The value of this component, normally used as a URL.

`UIGraphic` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to “javax.faces.Graphic”.
- The default value of the `rendererType` property must be set to “`javax.faces.Image`”.

4.1.5.3 Methods

`UIGraphic` adds no new processing methods.

4.1.5.4 Events

`UIGraphic` does not originate any standard events.

4.1.6 UIInput

`UIInput` (extends `UIOutput`, implements `EditableValueHolder`) is a component that both displays the current value of the component to the user (as `UIOutput` components do), and processes request parameters on the subsequent request that need to be decoded.

4.1.6.1 Component Type

The standard component type for `UIInput` components is "`javax.faces.Input`".

4.1.6.2 Properties

`UIInput` adds no new render-independent properties. See Section 3.2.5 "EditableValueHolder" for information about properties introduced by the implemented interfaces.

UIInput specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "`javax.faces.Input`".
- The default value of the `rendererType` property must be set to "`javax.faces.Text`".
- The `Converter` specified by the `converter` property (if any) must also be used to perform String->Object conversions during decoding.
- If the `value` property has an associated `ValueBinding`, the `setValue()` method of that `ValueBinding` will be called during the *Update Model Values* phase of the request processing lifecycle to push the local value of the component back to the corresponding model bean property.

4.1.6.3 Methods

The following method is used during the *Update Model Values* phase of the request processing lifecycle, to push the converted (if necessary) and validated (if necessary) local value of this component back to the corresponding model bean property.

```
public void updateModel(FacesContext context);
```

The following method is over-ridden from `UIComponent`:

```
public void broadcast(FacesEvent event);
```

In addition to the default `UIComponent.broadcast(javax.faces.event.FacesEvent)` processing, pass the `ValueChangeEvent` being broadcast to the method referenced by the `valueChangeListener` property (if any).

```
public void validate(FacesContext context);
```

Perform the algorithm described in the javadoc to validate the local value of this `UIInput`.

4.1.6.4 Events

All events are described in *Section 3.2.5 "EditableValueHolder"*.

4.1.7 UIMessage

`UIMessage` (extends `UIComponentBase`) encapsulates the rendering of error message(s) related to a specified input component.

4.1.7.1 Component Type

The standard component type for `UIMessage` components is "`javax.faces.Message`".

4.1.7.2 Properties

The following render-independent properties are added by the UIMessage component:

Name	**Access**	**Type**	**Description**
`for`	RW	`String`	Identifier of the component for which to render error messages. If this component is within the same NamingContainer as the target component, this must be the component identifier. Otherwise, it must be an absolute component identifier (starting with ":"). See the UIComponent.findComponent() Javadocs for more information.
showDetail	RW	boolean	Flag indicating whether the "detail" property of messages for the specified component should be rendered. Default value is "true".
showSummary	RW	boolean	Flag indicating whether the "summary" property of messages for the specified component should be rendered. Default value is "false".

`UIMessage` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "`javax.faces.Message`".
- The default value of the `rendererType` property must be set to "`javax.faces.Message`".

4.1.7.3 Methods.

`UIMessage` adds no new processing methods.

4.1.7.4 Events

`UIMessage` adds no new event handling methods.

4.1.8 UIMessages

`UIMessage` (extends `UIComponentBase`) encapsulates the rendering of error message(s) not related to a specified input component, or all enqueued messages.

4.1.8.1 Component Type

The standard component type for `UIMessage` components is "`javax.faces.Messages`".

4.1.8.2 Properties

The following render-independent properties are added by the UIMessages component:

Name	Access	Type	Description
`globalOnly`	RW	`boolean`	Flag indicating whether only messages **not** associated with any specific component should be rendered. If not set, all messages will be rendered. Default value is "false".
showDetail	RW	boolean	Flag indicating whether the "detail" property of messages for the specified component should be rendered. Default value is "false".
showSummary	RW	boolean	Flag indicating whether the "summary" property of messages for the specified component should be rendered. Default value is "true".

`UIMessages` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "`javax.faces.Messages`".
- The default value of the `rendererType` property must be set to "`javax.faces.Messages`".

4.1.8.3 Methods.

`UIMessages` adds no new processing methods.

4.1.8.4 Events

`UIMessages` adds no new event handling methods.

4.1.9 UIOutput

`UIOutput` (extends `UIComponentBase;` implements `ValueHolder`) is a component that has a value, optionally retrieved from a model tier bean via a value binding expression (see Section 5.1 "Value Binding Expressions"), that is displayed to the user. The user cannot directly modify the rendered value; it is for display purposes only:

4.1.9.1 Component Type

The standard component type for `UIOutput` components is "javax.faces.`Output`".

4.1.9.2 Properties

`UIOutput` adds no new render-independent properties. See Section 3.2.4 "ValueHolder" for information about properties introduced by the implemented classes.

`UIOutput` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "javax.faces.Output".
- The default value of the `rendererType` property must be set to "javax.faces.Text".

4.1.9.3 Methods

`UIOutput` adds no new processing methods. See Section 3.2.4 "ValueHolder" for information about methods introduced by the implemented interfaces.

4.1.9.4 Events

UIOutput does not originate any standard events. See Section 3.2.4 "ValueHolder" for information about events introduced by the implemented interfaces.

4.1.10 UIPanel

`UIPanel` (extends `UIComponentBase`) is a component that manages the layout of its child components.

4.1.10.1 Component Type

The standard component type for `UIPanel` components is "`javax.faces.Panel`".

4.1.10.2 Properties

`UIPanel` adds no new render-independent properties.

`UIPanel` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "`javax.faces.Panel`".
- The default value of the `rendererType` property must be set to `null`.

4.1.10.3 Methods

`UIPanel` adds no new processing methods.

4.1.10.4 Events

`UIPanel` does not originate any standard events

4.1.11 UIParameter

`UIParameter` (extends `UIComponentBase` is a component that represents an optionally named configuration parameter that affects the rendering of its parent component. `UIParameter` components do not generally have rendering behavior of their own.

4.1.11.1 Component Type

The standard component type for `UIParameter` components is "`javax.faces.Parameter`".

4.1.11.2 Properties

The following render-independent properties are added by the `UIParameter` component:

Name	Access	Type	Description
name	RW	String	The optional name for this parameter.
value	RW	Object	The value for this parameter.

`UIParameter` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "javax.faces.Parameter".
- The default value of the `rendererType` property must be set to `null`.

4.1.11.3 Methods

`UIParameter` adds no new processing methods.

4.1.11.4 Events

`UIParameter` does not originate any standard events

4.1.12 UISelectBoolean

UISelectBoolean (extends UIInput) is a component that represents a single boolean (true or false) value. It is most commonly rendered as a checkbox.

4.1.12.1 Component Type

The standard component type for UISelectBoolean components is "javax.faces.SelectBoolean".

4.1.12.2 Properties

The following render-independent properties are added by the UISelectBoolean component:

Name	Access	Type	Description
selected	RW	boolean	The selected state of this component. This property is a typesafe alias for the value property, so that the actual state to be used can be acquired via a value binding expression.

UISelectBoolean specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the family property must be set to "javax.faces.SelectBoolean".
- The default value of the rendererType property must be set to "javax.faces.Checkbox".

4.1.12.3 Methods

UISelectBoolean adds no new processing methods.

4.1.12.4 Events

UISelectBoolean inherits the ability to send ValueChangeEvent events from its parent UIInput component.

4.1.13 UISelectItem

`UISelectItem` (extends `UIComponentBase`) is a component that may be nested inside a `UISelectMany` or `UISelectOne` component, and represents exactly one `SelectItem` instance in the list of available options for that parent component.

4.1.13.1 Component Type

The standard component type for `UISelectItem` components is "`javax.faces.SelectItem`".

4.1.13.2 Properties

The following render-independent properties are added by the `UISelectItem` component:

Name	Access	Type	Description
`itemDescription`	RW	`String`	The optional description of this available selection item. This may be useful for tools.
itemDisabled	RW	boolean	Flag indicating that any synthesized `SelectItem` object should have its `disabled` property set to `true`.
itemLabel	RW	String	The localized label that will be presented to the user for this selection item.
itemValue	RW	Object	The server-side value of this item, of the same basic data type as the parent component's value. If the parent component type's value is a value binding expression that points at a primitive, this value must be of the corresponding wrapper type.
value	RW	javax.faces.model.SelectItem	The `SelectItem` instance associated with this component.

`UISelectItem` specializes the behavior of render-independent properties inherited

- The default value of the `family` property must be set to "javax.faces.SelectItem".
- The default value of the `rendererType` property must be set to `null`.
- If the `value` property is non-`null`, it must contain a `SelectItem` instance used to configure the selection item specified by this component.

- If the `value` property is a value binding expression, it must point at a `SelectItem` instance used to configure the selection item specified by this component.
- If the `value` property is `null`, and there is no corresponding value binding expression, the `itemDescription`, `itemDisabled`, `itemLabel` and `itemValue` properties must be used to construct a new `SelectItem` representing the selection item specified by this component.

4.1.13.3 Methods

`UISelectItem` adds no new processing methods.

4.1.13.4 Events

`UISelectItem` does not originate any standard events.

4.1.14 UISelectItems

UISelectItems (extends UIComponentBase) is a component that may be nested inside a UISelectMany or UISelectOne component, and represents zero or more SelectItem instances for adding selection items to the list of available options for that parent component.

4.1.14.1 Component Type

The standard component type for UISelectItems components is "javax.faces.SelectItems".

4.1.14.2 Properties

The following render-independent properties are added by the UISelectItems component:

Name	Access	Type	Description
value	RW	See below	The SelectItem instances associated with this component.

UISelectItems specializes the behavior of render-independent properties inherited

- The default value of the family property must be set to "javax.faces.SelectItems".
- The default value of the rendererType property must be set to null.
- If the value property (or the value returned by a value binding expression associated with the value property) is non-null, it must contain a SelectItem bean, an array of SelectItem beans, a Collection of SelectItem beans, or a Map, where each map entry is used to construct a SelectItem bean with the key as the label property of the bean, and the value as the value property of the bean (which must be of the same basic type as the value of the parent component's value).

4.1.14.3 Methods

UISelectItems adds no new processing methods.

4.1.14.4 Events

`UISelectItems` does not originate any standard events.

4.1.15 UISelectMany

`UISelectMany` (extends `UIInput`) is a component that represents one or more selections from a list of available options. It is most commonly rendered as a combobox or a series of checkboxes.

4.1.15.1 Component Type

The standard component type for `UISelectMany` components is "`javax.faces.SelectMany`".

4.1.15.2 Properties

The following render-independent properties are added by the `UISelectMany` component:

Name	Access	Type	Description
`selectedValues`	RW	`Object[]` or array of primitives	The selected item values of this component. This property is a typesafe alias for the `value` property, so that the actual state to be used can be acquired via a value binding expression.

`UISelectMany` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "`javax.faces.SelectMany`".
- The default value of the `rendererType` property must be set to "`javax.faces.Listbox`".
- See the class Javadocs for `UISelectMany` for additional requirements related to implicit conversions for the `value` property.

4.1.15.3 Methods

`UISelectMany` must provide a specialized `validate()` method which ensures that any decoded values are valid options (from the nested `UISelectItem` and `UISelectItems` children).

4.1.15.4 Events

`UISelectMany` inherits the ability to send `ValueChangeEvent` events from its parent `UIInput` component.

4.1.16 UISelectOne

`UISelectOne` (extends `UIInput`) is a component that represents zero or one selections from a list of available options. It is most commonly rendered as a combobox or a series of radio buttons.

4.1.16.1 Component Type

The standard component type for `UISelectOne` components is "`javax.faces.SelectOne`".

4.1.16.2 Properties

`UISelectOne` adds no new render-independent properties.

`UISelectOne` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "`javax.faces.SelectOne`".
- The default value of the `rendererType` property must be set to "`javax.faces.Menu`".

4.1.16.3 Methods

`UISelectOne` must provide a specialized `validate()` method which ensures that any decoded value is a valid option (from the nested `UISelectItem` and `UISelectItems` children).

4.1.16.4 Events

`UISelectOne` inherits the ability to send `ValueChangeEvent` events from its parent `UIInput` component.

4.1.17 UIViewRoot

`UIViewRoot` (extends `UIComponentBase;`) represents the root of the component tree.

In JSP applications, the tag handler for this component is involved in the state saving process. The tag handler for `UIViewRoot` must indicate that the body content must be buffered. In the `doAfterBody()` method of the tag handler, the `StateManager.getSerializedView()` and `StateManager.restoreView()` methods must be called.

4.1.17.1 Component Type

The standard component type for `UIViewRoot` components is "`javax.faces.ViewRoot`"

4.1.17.2 Properties

The following render-independent properties are added by the `UIViewRoot` component:

Name	Access	Type	Description
locale	RW	java.util.Locale	The Locale to be used in localizing the response for this view.
renderKitId	RW	String	The id of the `RenderKit` used to render this page.
`viewId`	RW	`String`	The view identifier for this view.

For an existing view, the `locale` property may be modified only from the event handling portion of *Process Validations* phase through *Invoke Application* phase, unless it is modified by an *Apply Request Values* event handler for an `ActionSource` or `EditableValueHolder` component that has its `immediate` property set to true (which therefore causes *Process Validations*, *Update Model Values*, and *Invoke Application* phases to be skipped).

`UIViewRoot` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "`javax.faces.ViewRoot`".
- The default value of the `rendererType` property must be set to `null`.

4.1.17.3 Methods

`UIViewRoot` adds no new processing methods.

`UIViewRoot` specializes the behavior of the `UIComponent.queueEvent()` method to maintain a list of queued events that can be transmitted later. It also specializes the behavior of the `processDecodes()`, `processValidators()`, `processUpdates()`, and `processApplication()` methods to broadcast queued events to registered listeners.

4.1.17.4 Events

`UIViewRoot` does not originate any standard events.

4.2 Standard UIComponent Model Beans

Several of the standard `UIComponent` subclasses described in the previous section reference JavaBean components to represent the underlying model data that is rendered by those components. The following subsections define the standard `UIComponent` model bean classes.

4.2.1 DataModel

`DataModel` is an abstract base class for creating wrappers around arbitrary data binding technologies. It can be used to adapt a wide variety of data sources for use by JavaServer Faces components that want to support access to an underlying data set that can be modelled as multiple rows. The data underlying a DataModel instance is modelled as a collection of row objects that can be accessed randomly via a zero-relative index

4.2.1.1 Properties

An instance of `DataModel` supports the following properties:

Name	Access	Type	Description
rowAvailable	RO	boolean	Flag indicating whether the current `rowIndex` value points at an actual row in the underlying data.
`rowCount`	RO	int	The number of rows of data objects represented by this DataModel instance, or -1 if the number of rows is unknown.
`rowData`	RO	`Object`	An object representing the data for the currently selected row. `DataModel` implementations must return an object that be successfully processed as the “base” parameter for the `PropertyResolver` in use by this application. If the current rowIndex value is -1, `null` is returned.
`rowIndex`	RW	`int`	Zero-relative index of the currently selected row, or -1 if no row is currently selected. When first created, a `DataModel` instance must return -1 for this property.
wrappedData	RW	Object	Opaque property representing the data object wrapped by this DataModel. Each individual implementation will restrict the types of Object(s) that it supports.

4.2.1.2 Methods

An instance of `DataModel` supports no additional public processing methods.

4.2.1.3 Events

No events are generated for this component.

4.2.1.4 Concrete Implementations

The JSF implementation must provide concrete implementations of DataModel (in the javax.faces.model package) for the following data wrapping scenarios:

- `ArrayDataModel` -- Wrap an array of Java objects.
- `ListDataModel` -- Wrap a `java.util.List` of Java objects.
- `ResultDataModel` -- Wrap an object of type `javax.servlet.jsp.jstl.sql.Result` (the query results from JSTL's SQL tag library)
- `ResultSetDataModel` -- Wrap an object of type `java.sql.ResultSet` (which therefore means that `javax.sql.RowSet` instances are also supported).
- `ScalarDataModel` -- Wrap a single Java object in what appears to be a one-row data set.

Each concrete `DataModel` implementation must extend the `DataModel` abstract base class, and must provide a constructor that accepts a single parameter of the object type being wrapped by that implementation (in addition to a zero-args constructor). See the JavaDocs for specific implementation requirements on `DataModel` defined methods, for each of the concrete implementation classes.

4.2.2 SelectItem

`SelectItem` is a utility class representing a single choice, from among those made available to the user, for a `UISelectMany` or `UISelectOne` component. It is not itself a `UIComponent` subclass.

4.2.2.1 Properties

An instance of `SelectItem` supports the following properties:

Name	Access	Type	Description
description	RW	String	A description of this selection item, for use in development tools.
`disabled`	RW	boolean	Flag indicating that this option should be rendered in a fashion that disables selection by the user. Default value is `false`.
`label`	RW	`String`	Label of this selection item that should be rendered to the user.
`value`	RW	`Object`	The server-side value of this item, of the same basic data type as the parent component's value. If the parent component type's value is a value binding expression that points at a primitive, this value must be of the corresponding wrapper type.

4.2.2.2 Methods

An instance of `SelectItem` supports no additional public processing methods.

4.2.2.3 Events

An instance of SelectItem supports no events.

4.2.3 SelectItemGroup

`SelectItemGroup` is a utility class extending `SelectItem`, that represents a group of subordinate `SelectItem` instances that can be rendered as a "sub-menu" or "option group". `Renderer`s will typically ignore the `value` property of this instance, but will use the `label` property to render a heading for the sub-menu.

4.2.3.1 Properties

An instance of `SelectItemGroup` supports the following additional properties:

Name	Access	Type	Description
selectItems	RW	SelectItem[]	Array of SelectItem instances representing the subordinate selection items that are members of the group represented by this SelectItemGroup instance.

Note that, since `SelectItemGroup` is a subclass of `SelectItem`, `SelectItemGroup` instances can be included in the `selectItems` property in order to create hierarchies of subordinate menus. However, some rendering environments may limit the depth to which such nesting is supported; for example, HTML/4.01 does not allow an `<optgroup>` to be nested inside another `<optgroup>` within a `<select>` control.

4.2.3.2 Methods

An instance of `SelectItemGroup` supports no additional public processing methods.

4.2.3.3 Events

An instance of `SelectItemGroup` supports no events.

CHAPTER 5

Value Binding and Method Binding Expression Evaluation

In the descriptions of the standard user interface component model, it was noted that all attributes, and nearly all properties can have a *value binding expression* associated with them (see Section 3.1.4 “Value Binding Expressions”). In addition, the `action`, `actionListener`, `validator`, and `valueChangeListener` properties can be defined by a *method binding expression* pointing at a public method in some class to be executed. This chapter describes the mechanisms and APIs that JavaServer Faces utilizes in order to evaluate value binding expressions and method binding expressions.

5.1 Value Binding Expressions

5.1.1 Overview

To support binding of attribute and property of values to dynamically calculated results, the name of the attribute or property can be associated with a value binding expression using the `setValueBinding()` method. Whenever the dynamically calculated result of evaluating the expression is required, the `getValue()` method of the `ValueBinding` is called, which returns the evaluated result. Such expressions can be used, for example, to dynamically calculate a component value to be displayed:

```
<h:outputText value=”#{customer.name}”/>
```

which, when this page is rendered, will retrieve the bean stored under the "customer" key, then acquire the name property from that bean and render it.

Besides the component value itself, value binding expressions can be used to dynamically compute attributes and properties. The following example checks a `boolean` property `manager` on the current `user` bean (presumably representing the logged-in user) to determine whether the `salary` property of an employee should be displayed or not:

```
<h:outputText rendered="#{user.manager}" value=
"#{employee.salary}"/>
```

which sets the `rendered` property of the component to `false` if the user is not a manager, and therefore causes this component to render nothing.

Value binding expressions also have special semantics (with restrictions on the available syntax) when a component that implements EditableValueHolder establishes a binding for the value property. See Section 5.1.4 "Set Value Semantics"for more information.

5.1.2 Value Binding Expression Syntax

The syntax of a *value binding expression* is identical to the syntax of an expression language expression defined in the *JavaServer Pages Specification* (version 2.0), sections 2.3 through 2.9, with the following exceptions:

- The expression delimiters for a value binding expression are "#{" and "}" instead of "${" and "}".
- Value binding expressions do not support EL functions.

This difference in delimiters points out the semantic differences between the two expression types:

- During rendering, value binding expressions are evaluated by the JSF implementation (via calls to the getValue() method) rather than by the compiled code for a JSP page.
- Value binding expressions may be evaluated programmatically, even when a JSP page is not present.
- Value binding expression evaluation leverages the facilities of the configured VariableResolver and PropertyResolver objects available via the Application object for the current web application, for which applications may provide plug in replacement classes that provide additional capabilities.

- A value binding expression is used for the value property of an `EditableValueHolder` component is used during the *Update Model Values* phase of the request processing lifecycle to **modify** the referenced value, rather than to **retrieve** it.

Examples of valid value binding expressions include:

- `#{foo}`
- `#{foo.bar}`
- `#{foo.bar.baz}`
- `#{foo[bar]}`
- `#{foo[“bar”]}`
- `#{foo[3]}`
- `#[foo[3].bar}`
- `#{foo.bar[3]}`
- #{customer.status == ‘VIP’}
- #((city.farenheitTemp - 32) * 5 / 9}
- Reporting Period: #{report.fromDate} to #{report.toDate}

For value binding expressions where the setValue() method is going to be called (such as during Update Model Values), the syntax of a value binding expression is limited to one of the following forms:

- #{expr-a.value-b}
- #{expr-a[value-b]]
- #{value-b}

where “expr-a” is a general expression (as described above) that evaluates to some object, and “value-b” is an identifier.

5.1.3 Get Value Semantics

When the `getValue()` method of a `ValueBinding` instance is called, the expression is evaluated (and the result of that evaluation is returned), evaluation takes place exactly as described in the *JavaServer Pages Specification* (version 2.0), sections 2.3 through 2.9, with the following exceptions:

- The left-most identifier in an expression is evaluated by the VariableResolver instance that is acquired from the Application instance for this web application. See Section 5.3.1 “VariableResolver” for more information.
- Each occurrence of the “.” or “[...]” operators in an expression is evaluated by the PropertyResolver instance that is acquired from the Application instance for this web application. See Section 5.3.2 “PropertyResolver” for more information.

Thus, page authors familiar with JSP EL expressions will be able to immediately understand how value binding expressions work in JSF.

5.1.4 Set Value Semantics

When the `setValue()` method on a `ValueBinding` is called, the syntax of the value binding restriction is restricted as described above. The implementation must perform the following processing to evaluate an expression of the form "#{expr-a.value-b}" or "#{expr-a[value-b]}":

- Evaluate `expr-a` into `value-a`.
- If `value-a` is `null`, throw `PropertyNotFoundException`.
- If `value-b` is `null`, throw `PropertyNotFoundException`.
- If `value-a` is a `Map`, call `value-a.put(value-b, new-value)`.
- If `value-a` is a `List` or an array:
  - Coerce `value-b` to `int`, throwing `ReferenceSyntaxException` on an error.
  - Attempt to execute `value-a.set(value-b, new-value)` or `Array.set(value-b, new-value)` as appropriate.
  - If `IndexOutOfBoundsException` or `ArrayIndexOutOfBoundsException` is thrown, throw `PropertyNotFoundException`.
  - If a different exception was thrown, throw `EvaluationException`.
- Otherwise (`value-a` is a JavaBean object):
  - Coerce `value-b` to `String`.
  - If `value-b` is a writeable property of `value-a` (as per the *JavaBeans Specification*), call the setter method (passing `new-value`); throwing `ReferenceSyntaxException` if an exception is thrown.
  - Otherwise, throw `PropertyNotFoundException`.

If the entire expression consists of a single identifier, the following rules apply:

- If the identifier matches the name of one of the implicit objects described below, throw `ReferenceSyntaxException`.
- Otherwise, if the identifier matches the key of an attribute in request scope, session scope, or application scope, the corresponding attribute value will be replaced by `new-value`.
- Otherwise, a new request scope attribute will be created, whose key is the identifier and whose value is `new-value`.

5.2 Method Binding Expressions

Method binding expressions are a specialized variant of value binding expressions. Rather than supporting the dynamic retrieval and setting of properties, method binding expressions support the invocation (i.e. execution) of an arbitrary public method of an arbitrary object, passing a specified set of parameters, and returning the result from the called method (if any). They may be used in any phase of the request processing lifecycle; the standard JSF components and framework employ them (encapsulated in a `MethodBinding` object) at the following times:

- During *Apply Request Values* or *Invoke Application* phase (depending upon the state of the `immediate` property), components that implement the `ActionSource` behavioral interface (see Section 3.2.1 “ActionSource”) utilize MethodBindings as follows:
  - If the `action` property is specified, it must be a `MethodBinding` expression that identifies an Application Action method (see Section 7.3 “Application Actions”) that takes no parameters and returns a String.
  - If the `actionListener` property is specified, it must be a `MethodBinding` that identifies a public method that accepts an `ActionEvent` (see Section 3.4.2 “Event Classes”) instance, and has a return type of `void`. The called method has exactly the same responsibilities as the `processAction()` method of an `ActionListener` instance (see Section 3.4.3 “Listener Classes”) that was built in to a separate Java class.
- During the *Apply Request Values* or *Process Validations* phase (depending upon the state of the `immediate` property), components that implement `EditableValueHolder` (such as `UIInput` and its subclasses) components (see Section 3.2.5 “EditableValueHolder”) utilize method binding expressions as follows:
  - If the `validator` property is specified, it must be a `MethodBinding` that identifies a public method that accepts a `FacesContext` instance and a `UIComponent` instance, and an `Object` containing the value to be validated, and has a return type of `void`. The called method has exactly the same responsibilities as the `validate()` method of a `Validator` instance (see Section 3.5.2 “Validator Classes”) that was built in to a separate Java class.
  - If the `valueListenerChange` property is specified, it must be a `MethodBinding` that identifies a public method that accepts a `ValueChangeEvent` (see Section 3.4.2 “Event Classes”) instance, and has a return type of `void`. The called method has exactly the same responsibilities as the `processValueChange()` method of a `ValueChangeListener` instance (see Section 3.4.3 “Listener Classes”) that was built in to a separate Java class.

Here is the set of component attributes that currently support `MethodBinding`s, and the method signatures to which they must point:

TABLE 5-1

component property	method signature
`action`	`public String <methodName>();`
`actionListener`	`public void <methodName>(javax.faces.event.ActionEvent);`
`validator`	`public void <methodName>(javax.faces.context.FacesContext, javax.faces.component.UIComponent, java.lang.Object);`
`valueChangeListener`	`public void <methodName>(javax.faces.event.ValueChangeEvent);`

Note that any of the method arguments may also be a subclass of what is listed above.

5.2.1 Method Binding Expression Syntax

The syntax of a *method binding expression* must conform to one of the following patterns:

- #{expr-a.value-b}
- #{expr-a[value-b]}

where "expr-a" is a value binding expression (see Section 5.1.2 "Value Binding Expression Syntax") and "value-b" is an identifier whose syntax matches that of a Java method name.

5.2.2 Method Binding Expression Semantics

Method binding expressions are evaluated via the use of a `MethodBinding` instance (see Section 5.3.4 "MethodBinding"), which supports two methods:

- If the `invoke()` method is executed:
  - The "expr-a" portion of the expression is used to construct a ValueBinding instance, and the getValue() method is called.

- The underlying class of the object returned by this evaluation is examined for the presence of a public `Method` whose parameter signature is compatible with the signature specified when the MethodBinding was created. The Method object may represent a Java method implemented by the underlying class, or by one of its super-classes.
- The identified method is called on the referenced Java object, passing the parameters specified on the invoke() call, and any returned value is returned.
- If the `getType()` method is executed:
  - The "expr-a" portion of the expression is used to construct a ValueBinding instance, and the getValue() method is called.
  - The underlying class of the object returned by this evaluation is examined for the presence of a public method whose parameter signature is compatible with the signature specified when the MethodBinding was created.
  - The `Class` representing the return type of the identified method is returned.

5.3 Expression Evaluation APIs

The description of expression evaluation in Section 5.1 "Value Binding Expressions" describes the default behavior provided by the JSF implementation. For advanced use cases, the application developer can modify the behavior of expression evaluation by implementing one or both of the following APIs, and configuring their use as described in Section 7.1 "Application".

5.3.1 VariableResolver

5.3.1.1 Overview

A `VariableResolver` is used by a `ValueBinding` (see Section 5.3.3 "ValueBinding") to support retrieval of the object associated with the left most identifier in a value binding expression.

The JSF implementation must provide a default `VariableResolver` implementation that provides the functionality described in Section 5.3.1.2 "Default VariableResolver Implementation". It is accessible via the `getVariableResolver()` method on the `Application` instance for this application (see Section 7.1 "Application").

An application (or framework) can provide an implementation with more features (such as support for additional implicit object names). This is accomplished by calling the `setVariableResolver()` method on the `Application` instance for this application. Typically, such an enhanced implementation will employ the Decorator Pattern, providing the additional support for implicit object names that it recognizes, and delegating responsibility for variable resolution to the standard implementation when the implicit object name is not recognized.

The following method signatures are supported:

```
public Object resolveVariable(FacesContext context, String name);
```

This method resolves the specified variable name, and returns the corresponding object instance, or `null` if no such instance can be identified.

5.3.1.2 Default VariableResolver Implementation

The JSF implementation must provide a default `VariableResolver` implementation, which may be acquired by calling `getVariableResolver()` on the `Application` instance for this application. This implementation's `resolveVariable()` method must support the following behavior:

The implementation must first compare the `name` parameter passed to the `resolveVariable()` method against the following values, returning the corresponding object on a match:

- `applicationScope`—A `Map` of the application scope attribute values, keyed by attribute name.
- `cookie`—An immutable `Map` of the cookie values for the current request, keyed by cookie name.
- `facesContext`—The `FacesContext` instance for the current request.
- `header`—An immutable `Map` of HTTP header values for the current request, keyed by header name. Only the first value for each header name is included.
- `headerValues`—An immutable `Map` of `String` arrays containing all of the header values for HTTP headers in the current request, keyed by header name.
- `initParam`—An immutable `Map` of the context initialization parameters for this web application.
- `param`—An immutable `Map` of the request parameters for this request, keyed by parameter name. Only the first value for each parameter name is included.
- `paramValues`—An immutable `Map` of `String` arrays containing all of the parameter values for request parameters in the current request, keyed by parameter name.

- requestScope—A Map of the request attributes for this request, keyed by attribute name.
- sessionScope—A Map of the session attributes for this request, keyed by attribute name.
- view—The UIViewRoot in the current component tree stored in the FacesContext for this request.

Next, the implementation must search for an attribute in request scope, then session scope (if it exists), then application scope with a matching key. If a match is found, the corresponding attribute value is returned.

Next, the implementation must examine the configuration information for the Managed Bean Facility, to determine if there is an entry with a matching <managed-bean-name>. If a match is found, a new bean will be created, optionally stored in some scope, and returned. See Section 5.3.1.3 "The Managed Bean Facility" for more information.

If no match is found based on any of the above rules, resolveVariable() must return null.

5.3.1.3 The Managed Bean Facility

The Managed Bean Creation facility is configured by the existence of <managed-bean> elements in one or more application configuration resources (see Section 10.3 "Application Configuration Resources"). Such elements describe the characteristics of a bean to be created, and properties to be initialized, with the following nested elements:

- <managed-bean-name> -- The key under which the created bean can be retrieved; also the key in the scope under which the created bean will be stored, unless the value of <managed-bean-scope> is set to none.
- <managed-bean-class> -- The fully qualified class name of the application class used to instantiate a new instance. This class must conform to JavaBeans design patterns -- in particular, it must have a public zero-args constructor, and must have public property setters for any properties referenced with nested <managed-property> elements -- or it must be a class that implements java.util.Map or java.util.List.
- <managed-bean-scope> -- The scope (request, session, or application) under which the newly instantiated bean will be stored after creation (under the key specified by the <managed-bean-name> element), or none for a bean that should be instantiated and returned, but not stored in any scope. The latter option is useful when dynamically constructing trees of related objects, as illustrated in the following example.

- <list-entries> or <map-entries> -- Used to configure managed beans that are themselves instances of java.util.List or java.util.Map, respectively. See below for details on the contents of these elements.
- <managed-property> -- Zero or more elements used to initialize the properties of the newly instantiated bean (see below).

After the new managed bean instance is instantiated, but before it is placed into the specified scope (if any), each nested <managed-property> element must be processed and a call to the corresponding property setter must be made to initialize the value of the corresponding property. If the managed bean has properties not referenced by <managed-property> elements, the values of such properties will not be affected by the creation of this managed bean; they will retain whatever default values are established by the constructor.

Each <managed-property> element contains the following elements used to configure the execution of the corresponding property setter call:

- <property-name> -- The property name of the property to be configured. The actual property setter method to be called will be determined as described in the JavaBeans Specification.
- Exactly one of the following sub-elements that can be used to initialize the property value in a number of different ways:
  - <map-entries> -- A set of key/value pairs used to initialize the contents of a property of type java.util.Map (see below for more details).
  - <null-value/> -- An empty element indicating that this property must be explicitly initialized to null. This element is not allowed if the underlying property is of a Java primitive type.
  - <value> -- A String value that will have any leading and trailing spaces stripped, and then be converted (according to the rules described in the JSP Specification for the <jsp:setProperty> action) to the corresponding data type of the property, prior to setting it to this value.
  - <list-entries> -- A set of values used to initialize the contents of a property of type array or java.util.List. See below for more information.

As described above, the <map-entries> element is used to initialize the key-value pairs of a property of type java.util.Map. This element may contain the following nested elements:

- <key-class> -- Optional element specifying the fully qualified class name for keys in the map to be created. If not specified, java.lang.String is used.
- <value-class> -- Optional element specifying the fully qualified class name for values in the map to be created. If not specified, java.lang.String is used.
- <map-entry> -- Zero or more elements that define the actual key-value pairs for a single entry in the map. Nested inside is a <key> element to define the key, and then exactly one of <null-value>, <value> to define the value. These elements

have the same meaning as when nested in a `<managed-property>` element, except that they refer to an individual map entry's value instead of the entire property value.

As described above, the `<list-entries>` element is used to initialize a set of values for a property of type array or `java.util.List`. This element may contain the following nested elements:

- `<value-class>` -- Optional element specifying the fully qualified class name for values in the map to be created. If not specified, `java.lang.String` is used.
- Zero or more elements of type `<null-value>`, `<value>` to define the individual values to be initialized. These elements have the same meaning as when nested in a `<managed-property>` element, except that they refer to an individual list element instead of the entire property value.

The following general rules apply to the operation of the Managed Bean Creation facility:

- Properties are assigned in the order that their `<managed-property>` elements are listed in the application configuration resource.
- If a managed bean has writeable properties that are not mentioned in `<managed-property>` elements, the values of those properties are not assigned any values.
- The bean instantiation and population with properties must be done lazily, when `Variable.resolveVariable()` is called. For example, this is the case when a `ValueBinding` or `MethodBinding` has its `getValue()` or `setValue()` method called.
- Due to the above mentioned laziness constraint, any error conditions that occur below are only required to be manifested at runtime. However, it is conceivable that tools may want to detect these errors earlier; this is perfectly acceptable. The presense of any of the errors described below, until the end of this section, must not prevent the application from deploying and being made available to service requests.
- It is an error to specify a managed bean class that does not exist, or that cannot be instantiated with a public, zero-args constructor.
- It is an error to specify a `<property-name>` for a property that does not exist, or does not have a public setter method, on the specified managed bean class.
- It is an error to specify a `<value>` element that cannot be converted to the type required by a managed property, or that, when evaluated, results in a value that cannot be converted to the type required by a managed property.
- It is an error for a managed bean created through this facility to have a property that points at an object stored in a scope with a (potentially) shorter life span. Specifically, this means, for an object created with the specified `<managed-bean-scope>`, then `<value>` evaluations can only point at created objects with the specified managed bean scope:
  - none -- none

- application -- none, application
- session -- none, application, session
- request -- none, application, session, request

- If a bean points to a property whose value is a mixed expression containing literal strings and expressions, the net scope of the mixed expression is considered to be the scope of the narrowest sub-expression, excluding expressions in the none scope.
- Data accessed via an implicit object is also defined to be in a scope. The following implicit objects are considered to be in request scope:
  - `cookie`
  - `facesContext`
  - `header`
  - `headerValues`
  - `param`
  - `paramValues`
  - `requestScope`
  - `view`
- The only implicit object in session scope is `sessionScope`
- The following implicit objects are considered to be in application scope:
  - `applicationScope`
  - `initParam`
- It is an error to configure cyclic references between managed beans.
- Managed bean names must conform to the syntax of a Java language identifier.

The initialization bean properties from `<map-entries>` and `<list-entries>` elements must adhere to the following algorithm, though any confirming implementation may be used.

For `<map-entries>`:

1. Call the property getter, if it exists.
2. If the getter returns `null` or doesn't exist, create a `java.util.HashMap`, otherwise use the returned `java.util.Map`.
3. Add all entries defined by nested `<map-entry>` elements in the order they are listed, converting key values defined by nested `<key>` elements to the type defined by `<key-class>` and entry values defined by nested `<value>` elements to the type defined by `<value-class>`. If a value is given as a value binding expression, evaluate the reference and store the result, converting to `<value-class>` if necessary. If `<key-class>` and/or `<value-class>` are not defined, use `java.lang.String`. Add `null` for each `<null-value>` element.

4. If a new `java.util.Map` was created in step 2), set the property by calling the setter method, or log an error if there is no setter method.

For `<list-entries>`:

1. Call the property getter, if it exists.
2. If the getter returns `null` or doesn't exist, create a `java.util.ArrayList`, otherwise use the returned `Object` (an array or a `java.util.List`).
3. If a `List` was returned or created in step 2), add all elements defined by nested `<value>` elements in the order they are listed, converting values defined by nested `<value>` elements to the type defined by `<value-class>`. If a value is given as a value binding expression, evaluate the reference and store the result, converting to `<value-class>` if necessary. If a `<value-class>` is not defined, use the value as-is (i.e., as a `java.lang.String`). Add null for each `<null-value>` element.
4. If an array was returned in step 2), create a `java.util.ArrayList` and copy all elements from the returned array to the new `List`, wrapping elements of a primitive type. Add all elements defined by nested `<value>` elements as described in step 3).
5. If a new `java.util.List` was created in step 2) and the property is of type `List`, set the property by calling the setter method, or log an error if there is no setter method.
6. If a new `java.util.List` was created in step 2) and the property is a java array, convert the `List` into an array of the property type, and set it by calling the setter method, or log an error if there is no setter method.
7. If a new `java.util.List` was created in step 4), convert the `List` to an array of the proper type for the property and set the property by calling the setter method, or log an error if there is no setter method.

5.3.1.4 Managed Bean Configuration Example

The following <managed-bean> elements might appear in one or more application configuration resources (see Section 10.3 “Application Configuration Resources”) to configure the behavior of the Managed Bean Creation facility.

Assume that your application includes `CustomerBean` with properties `mailingAddress` and `shippingAddress` of type `Address` (along with additional properties that are not shown), and `AddressBean` implementation classes with String properties of type `street`, `city`, `state`, `country`, and `postalCode`.

```
<managed-bean>
  <description>
    A customer bean will be created as needed, and stored in
    request scope. Its “mailingAddress” and “streetAddress”
    properties will be initialized by virtue of the fact that the
    “value” expressions will not encounter any object under
    key “addressBean” in any scope.
  </description>
  <managed-bean-name>customer</managed-bean-name>
  <managed-bean-class>
    com.mycompany.mybeans.CustomerBean
  </managed-bean-class>
  <managed-bean-scope> request </managed-bean-scope>
  <managed-property>
    <property-name>mailingAddress</property-name>
    <value>#{addressBean}</value>
  </managed-property>
  <managed-property>
    <property-name>shippingAddress</property-name>
    <value>#{addressBean}</value>
  </managed-property>
  <managed-property>
    <property-name>customerType</property-name>
    <value>New</value> <!-- Set to literal value -->
  </managed-property>
</managed-bean>
```

```
<managed-bean>
  <description>
    A new AddressBean will not be added to any scope, because we
    only want to create instances when a CustomerBean creation asks
    for them. Therefore, we set the scope to “none”.
  </description>
  <managed-bean-name>addressBean</managed-bean-name>
  <managed-bean-class>
    com.mycompany.mybeans.AddressBean
  </managed-bean-class>
  <managed-bean-scope> none </managed-bean-scope>
</managed-bean>
```

If a value binding expression “`#{customer.mailingAddress.city}`” were to be evaluated by the JSF implementation, and there was no object stored under key “`customer`” in request, session, or application scope, a new `CustomerBean` instance will be created and stored in request scope, with its `mailingAddress` and `shippingAddress` properties being initialized to instances of `AddressBean` as defined by the configuration elements shown above. Then, the evaluation of the remainder of the expression can proceed as usual.

Although not used by the JSF implementation at application runtime, it is also convenient to be able to indicate to JSF tools (at design time) that objects of particular types will be created and made available (at runtime) by some other means. For example, an application configuration resource could include the following information to declare that a JDBC data source instance will have been created, and stored in application scope, as part of the application’s own startup processing.

```
<referenced-bean>
  <description>
    A JDBC data source will be initialized and made available in
    some scope (presumably application) for use by the JSF based
    application when it is actually run. This information is not
    used by the JSF implementation itself; only by tools.
  </description>
  <referenced-bean-name> dataSource </referenced-bean-name>
  <referenced-bean-class>
    javax.sql.DataSource
  </referenced-bean-class>
</referenced-bean>
```

This information can be utilized by the tool to construct user interfaces based on the properties of the referenced beans.

5.3.2 PropertyResolver

A `PropertyResolver` is used by a `ValueBinding` (see Section 5.3.3 "ValueBinding") to resolve an `.` or `[]` operator during the evaluation of a value binding expression.

The JSF implementation must provide a default `PropertyResolver` implementation that provides the functionality described in Section 5.1.3 "Get Value Semantics". It is accessible via the `getPropertyResolver` method on the `Application` instance for this application (see Section 7.1 "Application").

An application (or framework) can provide an implementation with more features (such as support for non-JavaBeans-based property resolution on additional supported base classes). This is accomplished by calling the `setPropertyResolver` method on the `Application` instance for this application. Typically, such an enhanced implementation will employ the Decorator Pattern, providing the additional support for additional base classes that it recognizes, and delegating responsibility for property resolution to the standard implementation when the implicit object name is not recognized.

The following method signatures are supported:

```
public Object getValue(Object base, Object property) throws
EvaluationException, PropertyNotFoundException;

public Object getValue(Object base, int index) throws
EvaluationException, PropertyNotFoundException;
```

Retrieve and return the specified property value from the specified base object. The `int` variant is used for accessing elements of a property that is based on a `List` or array, while the `String` variant is used in all other cases.

```
public void setValue(Object base, Object property, Object
newValue) throws EvaluationException, PropertyNotFoundException;

public void setValue(Object base, int index, Object newValue)
throws EvaluationException, PropertyNotFoundException;
```

Modify the value of the specified property on the specified base object. The `int` variant is used for accessing elements of a property that is based on a `List` or array, while the `String` variant is used in all other cases.

```
public boolean isReadOnly(Object base, Object property) throws
EvaluationException, PropertyNotFoundException;

public boolean isReadOnly(Object base, int index) throws
EvaluationException, PropertyNotFoundException;
```

Return `true` if the specified property on the specified base object is known to be immutable; otherwise, return `false`. The `int` variant is used for accessing elements of a property that is based on a `List` or array, while the `String` variant is used in all other cases.

```
public Class getType(Object base, Object property) throws
EvaluationException, PropertyNotFoundException;

public Class getType(Object base, int index) throws
EvaluationException, PropertyNotFoundException;
```

Return the `Class` that defines the property type of the specified property on the specified base object, if it can be determined; otherwise, return `null`. The `int` variant is used for accessing elements of a property that is based on a `List` or array, while the `String` variant is used in all other cases.

5.3.3 ValueBinding

The `ValueBinding` class encapsulates the actual evaluation of a value binding expression. Instances of `ValueBinding` for specific references are acquired from the `Application` instance by calling the `createValueBinding` method (see Section 7.1 “Application”).

```
public Object getValue(FacesContext context) throws
EvaluationException, PropertyNotFoundException;
```

Evaluate the value binding expression used to create this `ValueBinding` instance, relative to the specified `FacesContext`, and return the referenced value.

```
public void setValue(FacesContext context, Object value) throws
EvaluationException, PropertyNotFoundException;
```

Evaluate the value binding expression used to create this `ValueBinding` instance, relative to the specified `FacesContext`, and update the referenced value to the specified new value.

```
public boolean isReadOnly(FacesContext context) throws
EvaluationException, PropertyNotFoundException;
```

Evaluate the value binding expression used to create this `ValueBinding` instance, relative to the specified `FacesContext`, and return `true` if the corresponding property is known to be immutable. Otherwise, return `false`.

```
public Class getType(FacesContext context) throws
EvaluationException, PropertyNotFoundException;
```

Evaluate the value binding expression used to create this `ValueBinding` instance, relative to the specified `FacesContext`, and return the `Class` that represents the data type of the referenced value, if it can be determined. Otherwise, return `null`.

5.3.4 MethodBinding

The `MethodBinding` class encapsulates the actual evaluation of a method binding expression. Instances of `MethodBinding` for specific references are acquired from the `Application` instance by calling the `createMethodBinding()` method (see Section 7.1.9 “Acquiring MethodBinding Instances”). Note that instances of `MethodBinding` are immutable, and contain no references to a `FacesContext` (which is passed in as a parameter when the reference expression is evaluated).

```
public Object invoke(FacesContext context, Object params[]) throws
EvaluationException, MethodNotFoundException;
```

Evaluate the method binding expression (see Section 5.2.2 “Method Binding Expression Semantics”) and call the identified method, passing the specified parameters. Return any value returned by the invoked method, or return `null` if the invoked method is of type `void`.

```
public Class getType(FacesContext context) throws
MethodNotFoundException;
```

Evaluate the method binding expression (see Section 5.2.2 “Method Binding Expression Semantics”) and return the `Class` representing the return type of the identified method. If this method is of type `void`, return `null` instead.

5.3.5 Expression Evaluation Exceptions

Three exception classes are defined to report errors related to the evaluation of value binding exceptions:

- `EvaluationException` (which extends `FacesException`)—used to report a problem evaluating a value binding exception dynamically.
- `MethodNotFoundException` (which extends `EvaluationException`)—used to report that a requested public method does not exist in the context of evaluation of a method binding expression.
- `PropertyNotFoundException` (which extends `EvaluationException`)—used to report that a requested property does not exist in the context of evaluation of a value binding expression.
- `ReferenceSyntaxException` (which extends `EvaluationException`)—used to report a syntax error in a value binding exception.

CHAPTER 6

Per-Request State Information

During request processing for a JSF page, a context object is used to represent request-specific information, as well as provide access to services for the application. This chapter describes the classes which encapsulate this contextual information.

6.1 FacesContext

JSF defines the `javax.faces.context.FacesContext` abstract base class for representing all of the contextual information associated with processing an incoming request, and creating the corresponding response. A `FacesContext` instance is created by the JSF implementation, prior to beginning the request processing lifecycle, by a call to the `getFacesContext` method of `FacesContextFactory`, as described in Section 6.5 "FacesContextFactory". When the request processing lifecycle has been completed, the JSF implementation will call the `release` method, which gives JSF implementations the opportunity to release any acquired resources, as well as to pool and recycle `FacesContext` instances rather than creating new ones for each request.

6.1.1 Application

```
public Application getApplication();
```

The JSF implementation must ensure that the `Application` instance for the current web application is available via this method, as a convenient alternative to lookup via an `ApplicationFactory`.

6.1.2 ExternalContext

It is sometimes necessary to interact with APIs provided by the containing environment in which the JavaServer Faces application is running. In most cases this is the servlet API, but it is also possible for a JavaServer Faces application to run inside of a portlet. JavaServer Faces provides the `ExternalContext` abstract class for this purpose. This class must be implemented along with the `FacesContext` class, and must be accessible via the `getExternalContext` method in `FacesContext`.

```
public ExternalContext getExternalContext();
```

The `ExternalContext` instance provides immediate access to all of the components defined by the containing environment (servlet or portlet) within which a JSF-based web application is deployed. The following table lists the container objects available from `ExternalContext`. Note that the **Access** column refers to whether the returned object is mutable. None of the properties may be set through `ExternalContext`. itself.

Name	Access	Type	Description
`applicationMap`	RW	`java.util.Map`	The application context attributes for this application.
`authType`	RO	`String`	The method used to authenticate the currently logged on user (if any).
`context`	RW	`Object`	The application context object for this application.
`initParameterMap`	RO	`java.util.Map`	The context initialization parameters for this application
`remoteUser`	RO	`String`	The login name of the currently logged in user (if any).
`request`	RW	`Object`	The request object for this request.
`requestContextPath`	RO	`String`	The context path for this application.
`requestCookieMap`	RO	`java.util.Map`	The cookies included with this request.

Name	Access	Type	Description
requestHeaderMap	RO	java.util.Map	The HTTP headers included with this request (value is a String).
requestHeaderValuesMap	RO	java.util.Map	.The HTTP headers included with this request (value is a String array).
requestLocale	RW	java.util.Locale	The preferred Locale for this request.
requestLocales	RW	java.util.Iterator	The preferred Locales for this request, in descending order of preference.
requestMap	RW	java.util.Map	The request scope attributes for this request.
requestParameterMap	RO	java.util.Map	The request parameters included in this request (value is a String).
requestParameterNames	RO	Iterator	The set of request parameter names included in this request.
requestParameterValuesMap	RO	java.util.Map	The request parameters included in this request (value is a String array).
requestPathInfo	RO	String	The extra path information from the request URI for this request.
requestServletPath	RO	String	The servlet path information from the request URI for this request.
response	RW	Object	The response object for the current request.
sessionMap	RW	java.util.Map	The session scope attributes for this request*.
userPrincipal	RO	java.security.Principal	The Principal object containing the name of the currently logged on user (if any).

* Accessing attributes via this Map will cause the creation of a session associated with this request, if none currently exists.

In addition to the above properties of `ExternalContext`, the following methods must be exposed. See the JavaDocs for more details.

```
public void dispatch(String path) throws IOException;

public void redirect(String url) throws IOException;
```

The `dispatch()` must use a RequestDispatcher provided by the application context object to incorporate content from a specified context-relative resource. The `redirect()` method must cause an HTTP Redirect to be sent to the client.

```
public String encodeActionURL(String url);

public String encodeResourceURL(String url);
```

Return the specified URLs, after performing any necessary encoding or rewriting to ensure that the URL correctly identifies an addressable action or resource, respectively, in the current application.

```
public String encodeNamespace(String value);
```

Return the specified name, prefixed as needed to ensure that it will be unique within the scope of the current page.

```
public void log(String message);

public void log(String message, Throwable throwable);
```

Log the message (and a stack trace of the exception) to the underlying context.

```
public String getInitParameter(String name);
```

Return the value of the specified context initialization parameter (if any).

```
public URL getResource(String path);

public InputStream getResourceAsStream(String path);
```

Return a URL or an InputStream, respectively, for the specified web application resource.

```
public Set getResourcePaths(String path);
```

Return the context-relative paths of web application resources matching the specified path.

```
public Object getSession(boolean create);
```

Return the session option associated with the current request, if any. If the `create` flag is set to `true`, a new session must be created if none is currently associated with this request.

```
public boolean isUserInRole(String role);
```

Return true if the currently logged in user is included in the specified role.

6.1.3 ViewRoot

```
public UIViewRoot getViewRoot();

public void setViewRoot(UIViewRoot root);
```

During the *Restore View* phase of the request processing lifecycle, the state management subsystem of the JSF implementation will identify the component tree (if any) to be used during the inbound processing phases of the lifecycle, and call `setViewRoot()` to establish it.

6.1.4 Message Queue

```
public void addMessage(String clientId, FacesMessage message);
```

During the *Apply Request Values*, *Process Validations*, *Update Model Values*, and *Invoke Application* phases of the request processing lifecycle, messages can be queued to either the component tree as a whole (if `clientId` is `null`), or related to a specific component based on its client identifier.

```
public Interator getClientIdsWithMessages();

public Severity getMaximumSeverity();

public Iterator getMessages(String clientId);

public Iterator getMessages();
```

The `getClientIdsWithMessages()` method must return an `Iterator` over the client identifiers for which at least one `Message` has been queued. The `getMaximumSeverity()` method returns the highest severity level on any `Message` that has been queued, regardless of whether or not the message is associated with a specific client identifier or not. The `getMessages(String)` method returns an `Iterator` over queued `Messages`, either those associated with the specified client identifier, or those associated with no client identifier if the parameter is `null`. The `getMessages()` method returns an `Iterator` over all queued `Messages`, whether or not they are associated with a particular client identifier.

For more information about the `Message` class, see Section 6.2 “FacesMessage”.

6.1.5 RenderKit

```
public RenderKit getRenderKit();
```

Return the `RenderKit` associated with the render kit identifier in the current `UIViewRoot` (if any).

6.1.6 ResponseStream and ResponseWriter

```
public ResponseStream getResponseStream();

public void setResponseStream(ResponseStream responseStream);

public ResponseWriter getResponseWriter();

public void setResponseWriter(ResponseWriter responseWriter);
```

JSF supports output that is generated as either a byte stream or a character stream. `UIComponent`s or `Renderer`s that wish to create output in a binary format should call `getResponseStream()` to acquire a stream capable of binary output. Correspondingly, `UIComponent`s or `Renderer`s that wish to create output in a character format should call `getResponseWriter()` to acquire a writer capable of character output.

Due to restrictions of the underlying servlet APIs, either binary or character output can be utilized for a particular response—they may not be mixed.

Please see Section 7.5 “ViewHandler” to learn when `setResponseWriter()` and `setResponseStream()` are called.

6.1.7 Flow Control Methods

```
public void renderResponse();

public void responseComplete();

public boolean getRenderResponse();

public boolean getResponseComplete();
```

Normally, the phases of the request processing lifecycle are executed sequentially, as described in Chapter 2 “Request Processing Lifecycle.” However, it is possible for components, event listeners, and validators to affect this flow by calling one of these methods.

The `renderResponse()` method signals the JSF implementation that, at the end of the current phase (in other words, after all of the processing and event handling normally performed for this phase is completed), control should be transferred immediately to the *Render Response* phase, bypassing any intervening phases that have not yet been performed. For example, an event listener for a tree control that was designed to process user interface state changes (such as expanding or contracting a node) on the server would typically call this method to cause the current page to be redisplayed, rather than being processed by the application.

The `responseComplete()` method, on the other hand, signals the JSF implementation that the HTTP response for this request has been completed by some means other than rendering the component tree, and that the request processing lifecycle for this request should be terminated when the current phase is complete. For example, an event listener that decided an HTTP redirect was required would perform the appropriate actions on the response object (i.e. calling `ExternalContext.redirect()`) and then call this method.

In some circumstances, it is possible that both `renderResponse()` and `responseComplete()` might have been called for the request. In this case, the JSF implementation must respect the `responseComplete()` call (if it was made) before checking to see if `renderResponse()` was called.

The `getRenderResponse()` and `getResponseComplete()` methods allow a JSF-based application to determine whether the renderResponse() or responseComplete() methods, respectively, have been called already for the current request.

6.1.8 Access To The Current FacesContext Instance

```
public static FacesContext getCurrentInstance();

public static void setCurrentInstance(FacesContext context);
```

Under most circumstances, JSF components, and application objects that access them, are passed a reference to the `FacesContext` instance for the current request. However, in some cases, no such reference is available. The `getCurrentInstance()` method may be called by any Java class in the current web application to retrieve an instance of the `FacesContext` for this request. The JSF implementation must ensure that this value is set correctly before `FacesContextFactory` returns a `FacesContext` instance, and that the value is maintained in a thread-safe manner.

6.2 FacesMessage

Each message queued within a `FacesContext` is an instance of the `javax.faces.application.FacesMessage` class. It offers the following constructors:

```
public FacesMessage();

public FacesMessage(String summary, String detail);

public FacesMessage(Severity severity, String summary, String
detail);
```

The following method signatures are supported to retrieve and set the properties of the completed message:

```
public String getDetail();
public void setDetail(String detail);

public Severity getSeverity();
public void setSeverity(Severity severity);

public String getSummary();
public void setSummary(String summary);
```

The message properties are defined as follows:

- `detail`—Localized detail text for this `FacesMessage` (if any). This will generally be additional text that can help the user understand the context of the problem being reported by this `FacesMessage`, and offer suggestions for correcting it.
- `severity`—A value defining how serious the problem being reported by this `FacesMessage` instance should be considered. Four standard severity values (SEVERITY_INFO, SEVERITY_WARN, SEVERITY_ERROR, and SEVERITY_FATAL) are defined as a typesafe enum in the `FacesMessage` class.
- `summary`—Localized summary text for this `FacesMessage`. This is normally a relatively short message that concisely describes the nature of the problem being reported by this `FacesMessage`.

6.3 ResponseStream

`ResponseStream` is an abstract class representing a binary output stream for the current response. It has exactly the same method signatures as the `java.io.OutputStream` class.

6.4 ResponseWriter

`ResponseWriter` is an abstract class representing a character output stream for the current response. A `ResponseWriter` instance is obtained via a factory method on `RenderKit`. Please see *Chapter 8 "RenderKit"*. It supports both low-level and high level APIs for writing character based information

```
public void close() throws IOException;

public void flush() throws IOException;

public void write(char c[]) throws IOException;

public void write(char c[], int off, int len) throws IOException;

public void write(int c) throws IOException;

public void write(String s) throws IOException;

public void write(String s, int off, int len) throws IOException;
```

The `ResponseWriter` class extends `java.io.Writer`, and therefore inherits these method signatures for low-level output. The `close()` method flushes the underlying output writer, and causes any further attempts to output characters to throw an `IOException`. The `flush` method flushes any buffered information to the underlying output writer, and commits the response. The `write` methods write raw characters directly to the output writer.

```
public abstract String getContentType();
public abstract String getCharacterEncoding();
```

Return the content type or character encoding used to create this ResponseWriter.

```
public void startDocument() throws IOException;
public void endDocument() throws IOException;
```

Write appropriate characters at the beginning (`startDocument`) or end (`endDocument`) of the current response.

```
public void startElement(String name, UIComponent
componentForElement) throws IOException;
```

Write the beginning of a markup element (the < character followed by the element name), which causes the `ResponseWriter` implementation to note internally that the element is open. This can be followed by zero or more calls to `writeAttribute` or `writeURIAttribute` to append an attribute name and value to the currently open element. The element will be closed (i.e. the trailing > added) on any subsequent call to `startElement`(), `writeComment`(), `writeText`(), `endDocument`(), `close()`, `flush()`, or `write()`. The `componentForElement` parameter tells the `ResponseWriter` which `UIComponent` this element corresponds to, if any. This parameter may be null to indicate that the element has no corresponding component. The presence of this parameter allows tools to provide their own implementation of `ResponseWriter` to allow the design time environment to know which component corresponds to which piece of markup.

```
public void endElement(String name) throws IOException;
```

Write a closing for the specified element, closing any currently opened element first if necessary.

```
public void writeComment(Object comment) throws IOException;
```

Write a comment string wrapped in appropriate comment delimiters, after converting the comment object to a `String` first. Any currently opened element is closed first.

```
public void writeAttribute(String name, Object value, String
componentPropertyName) throws IOException;

public void writeURIAttribute(String name, Object value, String
componentPropertyName) throws IOException;
```

These methods add an attribute name/value pair to an element that was opened with a previous call to `startElement()`, throwing an exception if there is no currently open element. The `writeAttribute()` method causes character encoding to be performed in the same manner as that performed by the `writeText()` methods. The `writeURIAttribute()` method assumes that the attribute value is a URI, and performs URI encoding (such as `%` encoding for HTML). The `componentPropertyName`, if present, denotes the property on the associated `UIComponent` for this element, to which this attribute corresponds. The `componentPropertyName` parameter may be null to indicate that this attribute has no corresponding property.

```
public void writeText(Object text, String property) throws
IOException;

public void writeText(char text[], int off, int len) throws
IOException;
```

Write text (converting from `Object` to `String` first, if necessary), performing appropriate character encoding and escaping. Any currently open element created by a call to `startElement` is closed first.

```
public abstract ResponseWriter cloneWithWriter(Writer writer);
```

Creates a new instance of this `ResponseWriter`, using a different `Writer`.

6.5 FacesContextFactory

A single instance of `javax.faces.context.FacesContextFactory` must be made available to each JSF-based web application running in a servlet or portlet container. This class is primarily of use by JSF implementors—applications will not generally call it directly. The factory instance can be acquired, by JSF implementations or by application code, by executing:

```
FacesContextFactory factory =
(FacesContextFactory)
FactoryFinder.getFactory(FactoryFinder.FACES_CONTEXT_FACTORY);
```

The `FacesContextFactory` implementation class provides the following method signature to create (or recycle from a pool) a `FacesContext` instance:

```
public FacesContext getFacesContext(Object context, Object
request, Object response, Lifecycle lifecycle);
```

Create (if necessary) and return a `FacesContext` instance that has been configured based on the specified parameters. In a servlet environment, the first argument is a `ServletContext`, the second a `ServletRequest` and the third a `ServletResponse`.

CHAPTER 7

Application Integration

Previous chapters of this specification have described the component model, request state information, and the next chapter describes the rendering model for JavaServer Faces user interface components. This chapter describes APIs that are used to link an application's business logic objects, as well as convenient pluggable mechanisms to manage the execution of an application that is based on JavaServer Faces. These classes are in the `javax.faces.application` package.

Access to application related information is centralized in an instance of the `Application` class, of which there is a single instance per application based on JavaServer Faces. Applications will typically provide one or more implementations of `ActionListener` (or a method that can be referenced by an `action` expression) in order to respond to `ActionEvent` events during the *Apply Request Values* or *Invoke Application* phases of the request processing lifecycle. Finally, a standard implementation of `NavigationHandler` (replaceable by the application or framework) is provided to manage the selection of the next view to be rendered.

7.1 Application

There must be a single instance of `Application` per web application that is utilizing JavaServer Faces. It can be acquired by calling the `getApplication()` method on the `FacesContext` instance for the current request, or the `getApplication()` method of the `ApplicationFactory` (see Section 7.2 "ApplicationFactory"), and provides default implementations of features that determine how application logic interacts with the JSF implementation. Advanced applications (or application frameworks) can install replacements for these default implementations, which will be used from that point on. Access to several integration objects is available via JavaBeans property getters and setters, as described in the following subsections.

7.1.1 ActionListener Property

```
public ActionListener getActionListener();

public void setActionListener(ActionListener listener);
```

Return or replace an `ActionListener` instance that will be utilized to process `ActionEvent` events during the *Apply Request Values* or *Invoke Application* phase of the request processing lifecycle. The JSF implementation must provide a default implementation `ActionListener` that performs the following functions:

- The `processAction()` method must call `FacesContext.renderResponse()` in order to bypass any intervening lifecycle phases, once the method returns.
- The `processAction()` method must next determine the logical outcome of this event, as follows:
  - If the originating component has a non-null `action` property, retrieve the `MethodBinding` and call `invoke()` to perform the application-specified processing in this action method, and use the value returned as the logical outcome.
  - Otherwise, the logical outcome is `null`.
- The `processAction()` method must finally retrieve the `NavigationHandler` instance for this application, and pass the logical outcome value (determined above) as a parameter to the `handleNavigation()` method of the `NavigationHandler` instance.

7.1.2 DefaultRenderKitId Property

```
public String getDefaultRenderKitId();

public void setDefaultRenderKitId(String defaultRenderKitId);
```

An application may specify the render kit identifier of the `RenderKit` to be used by the `ViewHandler` to render views for this application. If not specified, the default render kit identifier specified by `RenderKitFactory.HTML_BASIC_RENDER_KIT` will be used by the default `ViewHandler` implementation.

Unless the application has provided a custom `ViewHandler` that supports the use of multiple RenderKit instances in the same application, this method may only be called at application startup, before any Faces requests have been processed. This is a limitation of the current Specification, and may be lifted in a future release.

7.1.3 NavigationHandler Property

```
public NavigationHandler getNavigationHandler();

public void setNavigationHandler(NavigationHandler handler);
```

Return or replace the `NavigationHandler` instance (see Section 7.4 "NavigationHandler") that will be passed the logical outcome of the application `ActionListener` as described in the previous subsection. A default implementation must be provided, with functionality described in Section 7.4.2 "Default NavigationHandler Implementation":

7.1.4 PropertyResolver Property

```
public PropertyResolver getPropertyResolver();

public void setPropertyResolver(PropertyResolver resolver);
```

Return or replace the `PropertyResolver` instance that will be utilized to evaluate each `.` or `[]` operator when processing a value binding expression. A default implementation must be provided, which operates as described in Section 5.3.2 "PropertyResolver".

7.1.5 StateManager Property

```
public StateManager getStateManager();

public void setStateManager(StateManager manager);
```

Return or replace the `StateManager` instance that will be utilized during the *Restore View* and *Render Response* phases of the request processing lifecycle to manage state persistence for the components belonging to the current view. A default implementation must be provided, which operates as described in Section 7.6 "StateManager".

7.1.6 VariableResolver Property

```
public VariableResolver getVariableResolver();

public void setVariableResolver(VariableResolver resolver);
```

Return or replace the `VariableResolver` instance that will be utilized to convert the first name in a value binding expression into a corresponding object. A default implementation must be provided, which operates as described in Section 5.3.1 “VariableResolver”.

7.1.7 ViewHandler Property

```
public ViewHandler getViewHandler();

public void setViewHandler(ViewHandler handler);
```

See Section 7.5 “ViewHandler” for the description of the ViewHandler. The JSF implementation must provide a default `ViewHandler` implementation. This implementation may be replaced by calling `setViewHandler()` before the first time the *Render Response* phase has executed. If a call is made to `setViewHandler()` after the first time the *Render Response* phase has executed, the call must be ignored by the implementation.

7.1.8 Acquiring ValueBinding Instances

```
public ValueBinding createValueBinding(String ref);
```

Create and return a `ValueBinding` (see Section 5.3.3 “ValueBinding”) that can be used to evaluate the specified value binding expression. To avoid nondeterministic behavior, it is recommended that applications (or frameworks) wishing to plug in their own resolver implementations do so before `createValueBinding()` is called for the first time.

7.1.9 Acquiring MethodBinding Instances

```
public MethodBinding createMethodBinding(String ref, Class
params[]);
```

Create and return a `MethodBinding` (see Section 5.3.4 “MethodBinding”) that can be used to evaluate the specified method binding expression, and invoke the specified method. This method must have parameter signatures that are compatible with the classes in the `params` parameter[1] (which may be `null` or a zero-length array if the method to be called takes no parameters). The actual parameters to be passed when the method is executed are specified on the `invoke()` call of the returned `MethodBinding` instance.

To avoid nondeterministic behavior, it is recommended that applications (or frameworks) wishing to plug in their own resolver implementations do so before calling `createMethodBinding()` for the first time.

7.1.10 Object Factories

The `Application` instance for a web application also acts as an object factory for the creation of new JSF objects such as components, converters, and validators.

```
public UIComponent createComponent(String componentType);

public Converter createConverter(Class targetClass);

public Converter createConverter(String converterId);

public Validator createValidator(String validatorId);
```

Each of these methods creates a new instance of an object of the requested type[2], based on the requested identifier. The names of the implementation class used for each identifier is normally provided by the JSF implementation automatically (for standard classes described in this Specification), or in one or more application

1. The actual `Method` selected for execution must be selected as if by calling Class.getMethod() and passing the method name and the parameters signature specified in the createMethodBinding() call.

2. Converters can also be requested based on the object class of the value to be converted.

configuration resources (see Section 10.3 “Application Configuration Resources”) included with a JSF web application, or embedded in a JAR file containing the corresponding implementation classes.

```
public UIComponent createComponent(ValueBinding componentRef,
FacesContext context, String componentType);
```

Special version of the factory for UIComponent instances that is used when evaluating component reference expression properties. This method has the following behavior:

- Call the `getValue()` method on the specified `ValueBinding`, in the context of the specified `FacesContext`. If this results in a non-null `UIComponent` instance, return that as the value of the `getComponent()` call.
- If the getValue() call did not return a component instance, create a new component instance of the specified component type.

```
public void addComponent(String componentType, String
componentClass);

public void addConverter(Class targetClass, String
converterClass);

public void addConverter(String converterId, String
converterClass);

public void addValidator(String validatorId, String
validatorClass);
```

JSF-based applications can register additional mappings of identifiers to a corresponding fully qualified class name, or replace mappings provided by the JSF implementation in order to customize the behavior of standard JSF features. These methods are also used by the JSF implementation to register mappings based on `<component>`, `<converter>`, and `<validator>` elements discovered in an application configuration resource.

```
public Iterator getComponentTypes();

public Iterator getConverterIds();

public Iterator getConverterTypes();

public Iterator getValidatorIds();
```

JSF-based applications can ask the `Application` instance for a list of the registered identifiers for components, converters, and validators that are known to the instance.

7.1.11 Internationalization Support

The following methods and properties allow an application to describe its supported locales, and to provide replacement text for standard messages created by JSF objects.

```
public Iterator getSupportedLocales();
public void setSupportedLocales(Collection newLocales);
public Locale getDefaultLocale();
public void setDefaultLocale(Locale newLocale);
```

JSF applications may state the `Locale`s they support (and the default `Locale` within the set of supported `Locale`s) in the application configuration resources file. The setters for the following methods must be called when the configuration resources are parsed. Each time the setter is called, the previous value is overwritten.

```
public String getMessageBundle();

public void setMessageBundle(String messageBundle);
```

Specify the fully qualified name of the ResourceBundle from which the JSF implementation will acquire message strings that correspond to standard message keys See Section 2.5.2.4 “Localized Application Messages” for a list of the standard message keys recognized by JSF.

7.2 ApplicationFactory

A single instance of `javax.faces.application.ApplicationFactory` must be made available to each JSF-based web application running in a servlet or portlet container. The factory instance can be acquired by JSF implementations or by application code, by executing:

```
ApplicationFactory factory = (ApplicationFactory)
  FactoryFinder.getFactory(FactoryFinder.APPLICATION_FACTORY);
```

The `ApplicationFactory` implementation class supports the following methods:

```
public Application getApplication();

public void setApplication(Application application);
```

Return or replace the `Application` instance for the current web application. The JSF implementation must provide a default `Application` instance whose behavior is described in Section 7.1 "Application".

Note that applications will generally find it more convenient to access the `Application` instance for this application by calling the `getApplication()` method on the `FacesContext` instance for the current request.

7.3 Application Actions

An *application action* is an application-provided method on some Java class that performs some application-specified processing when an `ActionEvent` occurs, during either the *Apply Request Values* or the *Invoke Application* phase of the request processing lifecycle (depending upon the `immediate` property of the `ActionSource` instance initiating the event).

Application action is not a formal JSF API; instead any method that meets the following requirements may be used as an Action by virtue of evaluating a method binding expression:

- The method must be public.
- The method must take no parameters.
- The method must return `String`.

The action method will be called by the default `ActionListener` implementation, as described in Section 7.1.1 "ActionListener Property" above. Its responsibility is to perform the desired application actions, and then return a logical "outcome" (represented as a `String`) that can be used by a `NavigationHandler` in order to determine which view should be rendered next. The action method to be invoked is defined by a `MethodBinding` that is specified in the `action` property of a component that implements `ActionSource`. Thus, a component tree with more than one such `ActionSource` component can specify individual action methods to be invoked for each activated component, either in the same Java class or in different Java classes.

7.4 NavigationHandler

7.4.1 Overview

A single `NavigationHandler` instance is responsible for consuming the logical outcome returned by an application action that was invoked, along with additional state information that is available from the `FacesContext` instance for the current request, and (optionally) selecting a new view to be rendered. As mentioned below, if the outcome returned by the application action is `null`, the same view must be re-displayed. This is the only case where the same view (and component tree) is re-used..

```
public void handleNavigation(FacesContext context, String
fromAction, String outcome);
```

The `handleNavigation` method may select a new view by calling `createView()` on the `ViewHandler` instance for this application, optionally customizing the created view, and then selecting it by calling the `setViewRoot()` method on the `FacesContext` instance that is passed. Alternatively, the `NavigationHandler` can complete the actual response (for example, by issuing an HTTP redirect), and call `responseComplete()` on the `FacesContext` instance.

After a return from the `NavigationHandler`, control will normally proceed to the *Render Response* phase of the request processing lifecycle (see Section 2.2.6 “Render Response”), which will cause the newly selected view to be rendered. If the `NavigationHandler` called the `responseComplete()` method on the `FacesContext` instance, however, the *Render Response* phase will be bypassed.

7.4.2 Default NavigationHandler Implementation

JSF implementations must provide a default `NavigationHandler` implementation that maps the action reference that was utilized (by the default `ActionListener` implementation) to invoke an application action, the logical outcome value returned by that application action, as well as other state information, into the view identifier for the new view to be selected. The remainder of this section describes the functionality provided by this default implementation.

The behavior of the default NavigationHandler implementation is configured, at web application startup time, from the contents of zero or more *application configuration resources* (see Section 10.3 “Application Configuration Resources”). The configuration information is represented as zero or more <navigation-rule> elements, each keyed to a matching pattern for the *view identifier* of the current view expressed in a <from-view-id> element. This matching pattern must be either an exact match for a view identifier (such as “/index.jsp” if you are using the default ViewHandler), or the prefix of a component view id, followed by an asterisk (“*”) character. A matching pattern of “*”, or the lack of a <from-view-id> element inside a <navigation-rule> rule, indicates that this rule matches any possible component view identifier.

Nested within each <navigation-rule> element are zero or more <navigation-case> elements that contain additional matching criteria based on the action reference expression value used to select an application action to be invoked (if any), and the logical outcome returned by calling the invoke() method of that application action[3]. Finally, the <navigation-case> element contains a <to-view-id> element whose content is the view identifier that will be selected and stored in the FacesContext for the current request. See below for an example of the configuration information for the default NavigationHandler might be configured.

It is permissible for the application configuration resource(s) used to configure the default NavigationHandler to include more than one <navigation-rule> element with the same <from-view-id> matching pattern. For the purposes of the algorithm described below, all of the nested <navigation-case> elements for all of these rules shall be treated as if they had been nested inside a single <navigation-rule> element.

The default NavigationHandler implementation must behave as if it were performing the following algorithm (although optimized implementation techniques may be utilized):

- If the logical outcome value passed to the handleNavigation() method is null, do not scan for matching rules. This is an indication that the current view should be redisplayed.
- Find a <navigation-rule> element for which the view identifier (of the view in the FacesContext instance for the current request) matches the <from-view-id> matching pattern of the <navigation-rule>. Rule instances are considered in the following order:
  - An exact match of the view identifier against a <from-view-id> pattern that does not end with an asterisk (“*”) character.

3. It is an error to specify more than one <navigation-case>, nested within one or more <navigation-rule> elements with the same <from-view-id> matching pattern, that have exactly the same combination of <from-xxx> element values.

- For `<from-view-id>` patterns that end with an asterisk, an exact match on characters preceding the asterisk against the prefix of the view id. If the patterns for multiple navigation rules match, pick the longest matching prefix first.
- If there is a `<navigation-rule>` with a `<from-view-id>` pattern of only an asterisk[4], it matches any view identifier.

- From the `<navigation-case>` elements nested within the matching `<navigation-rule>` element, locate a matching navigation case by matching the `<from-action>` and `<from-outcome>` values against the corresponding parameter values passed in to the handleNavigation() method. Navigation cases are checked in the following order:
  - Cases specifying both a `<from-action>` value and a `<from-outcome>` value are matched against the `action` expression and `outcome` parameters passed to the `handleNavigation()` method (both parameters must be not null, and both must be equal to the corresponding condition values, in order to match).
  - Cases that specify only a `<from-outcome>` value are matched against the `outcome` parameter passed to the `handleNavigation()` method (which must be not null, and equal to the corresponding condition value, to match).
  - Cases that specify only a `<from-action>` value are matched against the `action` expression parameter passed to the `handleNavigation()` method (which must be not null, and equal to the corresponding condition value, to match).
  - Any remaining case is assumed to match.
- If a matching `<navigation-case>` element was located, and the <redirect/> element was *not* specified in this <navigation-case> (or the application is running in a Portlet environment, where redirects are not possible), use the `<to-view-id>` element of the matching case to request a new `UIViewRoot` instance from the `ViewHandler` instance for this application, and pass it to the `setViewRoot()` method of the `FacesContext` instance for the current request. Then, exit the algorithm.
- If a matching `<navigation-case>` element was located, the `<redirect/>` element *was* specified in this `<navigation-case>`, and the application is not running in a Portlet environment, use the `<to-view-id>` element of the matching case to construct a context-relative path that corresponds to that view id, cause the current response to perform an HTTP redirect to this path, and call `responseComplete()` on the `FacesContext` instance for the current request.
- If no matching `<navigation-case>` element was located, return to Step 1 and find the next matching `<navigation-rule>` element (if any). If there are no more matching rule elements, return without changing the current view.

A rule match always causes a new view to be created, losing the state of the old view.

4. Or, equivalently, with no `<from-view-id>` element at all.

7.4.3 Example NavigationHandler Configuration

The following `<navigation-rule>` elements might appear in one or more application configuration resources (see Section 10.3 “Application Configuration Resources”) to configure the behavior of the default `NavigationHandler` implementation:

```
<navigation-rule>

  <description>
    APPLICATION WIDE NAVIGATION HANDLING
  </description>
  <from-view-id> * </from-view-id>

  <navigation-case>
    <description>
      Assume there is a “Logout” button on every page that
      invokes the logout Action.
    </description>
    <display-name>Generic Logout Button</display-name>
    <from-action>#{userBean.logout}</from-action>
    <to-view-id>/logout.jsp</to-view-id>
  </navigation-case>

  <navigation-case>
    <description>
      Handle a generic error outcome that might be returned
      by any application Action.
    </description>
    <display-name>Generic Error Outcome</display-name>
    <from-outcome>loginRequired</from-outcome>
    <to-view-id>/must-login-first.jsp</to-view-id>
  </navigation-case>

</navigation-rule>
```

```
<navigation-rule>

  <description>
    LOGIN PAGE NAVIGATION HANDLING
  </description>
  <from-view-id> /login.jsp </from-view-id>

  <navigation-case>
    <description>
      Handle case where login succeeded.
    </description>
    <display-name>Successful Login</display-name>
    <from-action>#{userBean.login}</from-action>
    <from-outcome>success</from-outcome>
    <to-view-id>/home.jsp</to-view-id>
  </navigation-case>

  <navigation-case>
    <description>
      User registration for a new user succeeded.
    </description>
    <display-name>Successful New User Registration</display-name>
    <from-action>#{userBean.register}</from-action>
    <from-outcome>success</from-outcome>
    <to-view-id>/welcome.jsp</to-view-id>
  </navigation-case>

  <navigation-case>
    <description>
      User registration for a new user failed because of a
      duplicate username.
    </description>
    <display-name>Failed New User Registration</display-name>
    <from-action>#{userBean.register}</from-action>
    <from-outcome>duplicateUserName</from-outcome>
    <to-view-id>/try-another-name.jsp</to-view-id>
  </navigation-case>

</navigation-rule>
```

```
<navigation-rule>

  <description>
    Assume there is a search form on every page. These navigation
    cases get merged with the application-wide rules above because
    they use the same “from-view-id” pattern. The same thing would
    also happen if “from-view-id” was omitted here, because that is
    equivalent to a matching pattern of “*”.
  </description>
  <from-view-id> * </from-view-id>

  <navigation-case>
    <display-name>Search Form Success</display-name>
    <from-action>#{searchForm.go}</from-action>
    <from-outcome>success</from-outcome>
    <to-view-id>/search-results.jsp</to-view-id>
  </navigation-case>

  <navigation-case>
    <display-name>Search Form Failure</display-name>
    <from-action>#{searchForm.go}</from-action>
    <to-view-id>/search-problem.jsp</to-view-id>
  </navigation-case>

</navigation-rule>
```

```
<navigation-rule>

  <description>
    Searching works slightly differently in part of the site.
  </description>
  <from-view-id> /movies/* </from-view-id>

  <navigation-case>
    <display-name>Search Form Success</display-name>
    <from-action>#{searchForm.go}</from-action>
    <from-outcome>success</from-outcome>
    <to-view-id>/movie-search-results.jsp</to-view-id>
  </navigation-case>

  <navigation-case>
    <display-name>Search Form Failure</display-name>
    <from-action>#{searchForm.go}</from-action>
    <to-view-id>/search-problem.jsp</to-view-id>
  </navigation-case>

</navigation-rule>
```

7.5 ViewHandler

`ViewHandler` is the pluggability mechanism for allowing implementations of or applications using the JavaServer Faces specification to provide their own handling of the activities in the *Render Response* and *Restore View* phases of the request processing lifecycle. This allows for implementations to support different response generation technologies, as well as different state saving/restoring approaches.

A JSF implementation must provide a default implementation of the `ViewHandler` interface. See Section 7.1.7 "ViewHandler Property" for information on replacing this default implementation with another implementation.

7.5.1 Overview

ViewHandler defines the public APIs described in the following paragraphs

```
public Locale calculateLocale(FacesContext context);
public String calculateRenderKitId(FacesContext context);
```

These methods are called from `createView()` to allow the new view to determine the `Locale` to be used for all subsequent requests, and to find out which `renderKitId` should be used for rendering the view.

```
public UIViewRoot createView(FacesContext context, String viewId);
```

Create and return a new `UIViewRoot` instance, initialized with information from the specified `FacesContext` and view identifier parameters. It is the callers responsibility to ensure that `setViewId()` is called on the returned view, passing the same `viewId` value.

```
public String getActionURL(FacesContext context, String viewId);
```

Returns a URL, suitable for encoding and rendering, that (if activated) will cause the JSF request processing lifecycle for the specified `viewId` to be executed

```
public String getResourceURL(FacesContext context, String path);
```

Returns a URL, suitable for encoding and rendering, that (if activated) will retrieve the specified web application resource.

```
public void renderView(FacesContext context, UIViewRoot
viewToRender) throws IOException, FacesException;
```

This method must be called during the *Render Response* phase of the request processing lifecycle. It must provide a valid `ResponseWriter` or `ResponseStream` instance, storing it in the `FacesContext` instance for the current request (see Section 6.1.6 "ResponseStream and ResponseWriter"), and then perform whatever actions are required to cause the view currently stored in the `viewRoot` of the `FacesContext` instance for the current request to be rendered to the corresponding writer or stream. It must also interact with the associated `StateManager` (see Section 7.6 "StateManager"), by calling the `getSerializedView()` and `saveView()` methods, to ensure that state information for current view is saved between requests.

```
public UIViewRoot restoreView(FacesContext context, String viewId)
throws IOException;
```

This method must be called from the *Restore View* phase of the request processing lifecycle. It must perform whatever actions are required to restore the view associated with the specified `FacesContext` and `viewId`.

It is the caller's responsibility to ensure that the returned `UIViewRoot` instance is stored in the `FacesContext` as the new `viewRoot` property. In addition, if `restoreView()` returns `null` (because there is no saved state for this view identifier), the caller must call `createView()`, and call `renderResponse()` on the `FacesContext` instance for this request.

```
public void writeState(FacesContext context) throws IOException;
```

Take any appropriate action to either immediately write out the current view's state information (by calling `StateManager.writeState()`), or noting where state information may later be written. This method must be called once per call to the `encodeEnd()` method of any renderer for a `UIForm` component, in order to provide the `ViewHandler` an opportunity to cause saved state to be included with each submitted form

7.5.2 Default ViewHandler Implementation

The terms *view identifier* and `viewId` are used interchangeably below and mean the context relative path to the web application resource that produces the view, such as a JSP page. In the JSP case, this is a context relative path to the jsp page representing the view, such as `/foo.jsp`.

JSF implementations must provide a default `ViewHandler` implementation, designed to support the rendering of JSP pages containing JSF components, that must behave as described in the remainder of this section:

The `calculateLocale()` method must fulfill the following responsibilities:

- Attempt to match one of the locales returned by the `getLocales()` method of the `ExternalContext` instance for this request, against the supported locales for this application as defined in the application configuration resources. Matching is performed by the algorithm described in Section JSTL.8.3.2 of the JSTL Specification. If a match is found, return the corresponding `Locale` object.
- Otherwise, if the application has specified a default locale in the application configuration resources, return the corresponding `Locale` object.
- Otherwise, return the value returned by calling `Locale.getDefault()`.

The `calculateRenderKitId()` method must fulfill the following responsibilities:

- Return the value returned by `Application.getDefaultRenderKitId()` if it is not `null`.

- Otherwise, return the value specified by the symbolic constant `RenderKitFactory.HTML_BASIC_RENDER_KIT`.

The `createView()` method must fulfill the following responsibilities:

- Create a new `UIViewRoot` object instance
- Conditionally copy the `renderKitId` and `locale` from any current view for the current request (as described in the Javadocs for `createView()`).
- Return the newly created `UIViewRoot`.

The `getActionURL()` method must fulfill the following responsibilities:

- If the specified `viewId` does not start with a "/", throw `IllegalArgumentException`.
- If prefix mapping (such as "/faces/*") is used for `FacesServlet`, prepend the context path of the current application, and the specified prefix, to the specified viewId and return the completed value. For example "`/cardemo/faces/chooseLocale.jsp`".
- If suffix mapping (such as "*.faces") is used for `FacesServlet`, and the specified viewId ends with the specified suffix, replacing the suffix with the value specified by the context initialization parameter named by the symbolic constant `ViewHandler.DEFAULT_SUFFIX_NAME` (if no such context initialization parameter is present, use the value of the symbolic constant ViewHandler.DEFAULT_SUFFIX as the replacement suffix), prefix this value with the context path for the current web application, and return the result. For example "`/cardemo/chooseLocale.faces`"

The `getResourceURL()` method must fulfill the following responsibilities:

- If the specified path starts with a "/", prefix it with the context path for the current web application, and return the result.
- Otherwise, return the specified `path` value unchanged.

The `renderView()` method must fulfill the following responsibilities:

- If the current request is a `ServletRequest`, call the `set()` method of the `javax.servlet.jsp.jstl.core.Config` class, passing the current `ServletRequest`, the symbolic constant `Config.FMT_LOCALE`, and the `locale` property of the specfied `UIViewRoot`. This configures JSTL with the application's preferred locale for rendering this response.
- If suffix mapping (such as "*.faces") is used for `FacesServlet`, examine the `viewId` property of the specfied `UIViewRoot`. If it ends with a matching suffix, modify the viewId property by replacing the suffix with the value specified by the context initialization parameter named by the symbolic constant `ViewHandler.DEFAULT_SUFFIX_NAME` (if no such context initialization parameter is present, use the value of the symbolic constant ViewHandler.DEFAULT_SUFFIX as the replacement suffix).

- Treat the (possibly modified) viewId as a context-relative path (starting with a slash character), by passing it to the dispatch() method of the ExternalContext associated with this request.

The restoreView() method must fulfill the following responsibilities:

- If the current request is a servlet request, set the character encoding to be used for processing this request, either from a "charset" attribute included on the incoming Content-Type header, or from a value previously saved in the session under the key specified by the symbolic constant ViewHandler.CHARACTER_ENCODING_KEY (if the request is part of a session).
- Calculate the viewId that corresponds to this request, as follows:
  - If prefix mapping (such as "/faces/*") is used for FacesServlet, the viewId is set from the extra path information of the request URI.
  - If suffix mapping (such as "*.faces") is used for FacesServlet, the viewId is set from the servlet path information of the request URI, after replacing the suffix with the value of the context initialization parameter named by the symbolic constant ViewHandler.DEFAULT_SUFFIX_NAME (if no such context initialization parameter is present, use the value of the symbolic constant ViewHandler.DEFAULT_SUFFIX as the replacement suffix).
- If no viewId could be identified, call the redirect() method of the ExternalContext instance for this request, passing the context path of this web application.
- Otherwise, call the restoreView() method of the associated StateManager, passing the FacesContext instance for the current request and the calculated viewId, and return the returned UIViewRoot.

In JSP applications, the default ViewHandler must delegate certain of its responsibilities, as follows:

- The responsibility to configure and install an appropriate ResponseWriter is delegated to the doStartTag() method of UIComponentTag.
- The renderView() responsibility to interact with the StateManager for ensuring that state is saved between requests (by calling saveSerializedView() and writeState()) is delegated to the doAfterBody() method of the tag handler corresponding to the <f:view> custom action.

In non-JSP applications, these responsibilities must be performed by a custom ViewHandler implementation.

7.6 StateManager

`StateManager` directs the process of saving and restoring the view between requests. The `StateManager` instance for an application is retrieved from the `Application` instance, and therefore cannot know any details of the markup language created by the `RenderKit` being used to render a view. Therefore, the `StateManager` utilizes a helper object (see Section 8.3 "ResponseStateManager"), that is provided by the `RenderKit` implementation, and is therefore aware of the markup language details. The JSF implementation must provide a default `StateManager` implementation that supports the behavior described below.

7.6.1 Overview

The state of a view is divided into two pieces:

- *Tree Structure*. This includes component parent-child relationships, including facets.
- *Component State*. This includes:
  - Component attributes and properties, and
  - `Validator`s, `Converter`s, `FacesListener`s, and other objects attached to a component. The manner in which these *attached objects* are saved is up to the component implementation. For attached objects that may have state, the `StateHolder` interface (see Section 3.2.3 "StateHolder") is provided to allow these objects to preserve their own attributes and properties. If an attached object does not implement `StateHolder`, but does implement `Serializable`, it is saved using standard serialization. Attached objects that do not implement either `StateHolder` or `Serializable` must have a public, zero-arg constructor, and will be restored only to their initial, default object state[5].

The separation between tree structure and tree state has been explicitly called out to make it clear that implementations can use a different mechanism for persisting the structure than is used to persist the state. For example, in a system where the tree structure is stored statically, as an XML file, for example, the system could keep a DOM representation of the trees representing the webapp UI in memory, to be used by all requests to the application.

5. The implementation classes for attached object must include a public zero-arguments constructor.

7.6.2 State Saving Alternatives and Implications

JSF implementations support two primary mechanisms for saving state, based on the value of the `javax.faces.STATE_SAVING_METHOD` initialization parameter (see Section 10.1.3 “Application Configuration Parameters”). The possible values for this parameter give a general indication of the approach to be used, while allowing JSF implementations to innovate on the technical details:

- *client* -- Cause the saved state to be included in the rendered markup that is sent to the client (such as in a hidden input field for HTML). The state information must be included in the subsequent request, making it possible for JSF to restore the view without having saved information on the server side.
- *server* -- Cause the saved state to be stored on the server (perhaps by being stored in a servlet or portlet session) in between requests.

If your application uses *client* state saving, the values of all component attributes and properties (as well as the saved state of attached objects) must implement `java.io.Serializable`.

7.6.3 State Saving Methods.

```
public StateManager.SerializedView
saveSerializedView(FacesContext context);
```

This method causes the tree structure and component state of the view contained in the argument `FacesContext` to be collected, stored, and returned in a `StateManager.SerializedView` instance. If `null` is returned from this method, there is no state to save.

This method must also enforce the rule that component ids within a `NamingContainer` must be unique

```
public void writeState(FacesContext context,
StateManager.SerializedView state) throws IOException;
```

Save the state represented in the specified `SerializedView` instance, in an implementation dependent manner.

```
protected Object getTreeStructureToSave(FacesContext context);
```

This method must create a `Serializable` object that represents the tree structure of the component tree for this view. Tree structure is comprised of parent-child relationships, including facets. The `id` of each component and facet must also be saved to allow the naming containers in the tree to be correctly restored when this view is restored.

```
protected Object getComponentStateToSave(FacesContext context);
```

This method must create a `Serializable` object representing the component state (attributes, properties, and attached objects) of the component tree for this view. Attached objects that wish to save and restore their own state must implement `StateHolder`.

7.6.4 State Restoring Methods

```
public UIViewRoot restoreView(FacesContext context, String
viewId);
```

Restore the tree structure and the component state of the view for this `viewId` to be restored, in an implementation dependent manner. If there is no saved state information available for this `viewId`, this method returns `null`.

The default implementation of this method calls through to `restoreTreeStructure()` and, if necessary `restoreComponentState()`.

```
protected UIViewRoot restoreTreeStructure(FacesContext context,
String viewId);
```

Convenience method to construct a new UIViewRoot and populate it with the child and facet descendants represented in the saved tree structure information.

```
protected void restoreComponentState(FacesContext context,
UIViewRoot viewRoot);
```

Convenience method to restore the attributes, properties, and attached objects of all components in the restored component tree. This method must be called only if `restoreTreeStructure()` returned a non-`null` `UIViewRoot` instance.

CHAPTER 8

Rendering Model

JavaServer Faces supports two programming models for decoding component values from incoming requests, and encoding component values into outgoing responses - the *direct implementation* and *delegated implementation* models. When the *direct implementation* model is utilized, components must decode and encode themselves. When the *delegated implementation* programming model is utilized, these operations are delegated to a `Renderer` instance associated (via the `rendererType` property) with the component. This allows applications to deal with components in a manner that is predominantly independent of how the component will appear to the user, while allowing a simple operation (selection of a particular `RenderKit`) to customize the decoding and encoding for a particular client device or localized application user.

Component writers, application developers, tool providers, and JSF implementations will often provide one or more `RenderKit` implementations (along with a corresponding library of `Renderer` instances). In many cases, these classes will be provided along with the `UIComponent` classes for the components supported by the `RenderKit`. Page authors will generally deal with `RenderKit`s indirectly, because they are only responsible for selecting a render kit identifier to be associated with a particular page, and a `rendererType` property for each `UIComponent` that is used to select the corresponding `Renderer`.

8.1 RenderKit

A `RenderKit` instance is optionally associated with a view, and supports components using the *delegated implementation* programming model for the decoding and encoding of component values. Each JSF implementation must provide a default

RenderKit instance (named by the render kit identifier associated with the String constant RenderKitFactory.HTML_BASIC_RENDER_KIT as described below) that is utilized if no other RenderKit is selected.

```
public Renderer getRenderer(String family, String rendererType);
```

Return the Renderer instance corresponding to the specified component family and rendererType (if any), which will typically be the value of the rendererType property of a UIComponent about to be decoded or encoded.

```
public void addRenderer(String family, String rendererType,
Renderer renderer);
```

Applications that wish to go beyond the capabilities of the standard RenderKit that is provided by every JSF implementation may either choose to create their own RenderKit instances and register them with the RenderKitFactory instance (see Section 8.4 “RenderKitFactory”), or integrate additional (or replacement) supported Renderer instances into an existing RenderKit instance. For example, it will be common to for an application that requires custom component classes and Renderers to register them with the standard RenderKit provided by the JSF implementation, at application startup time See Section 10.3.6 “Example Application Configuration Resource”for an example of a faces-config.xml configuration resource that defines two additional Renderer instances to be registered in the default RenderKit.

```
public ResponseWriter createResponseWriter(Writer writer, String
contentTypeList, String characterEncoding);
```

Use the provided Writer to create a new ResponseWriter instance for the specified character encoding.

The contentTypeList parameter is an "Accept header style" list of content types for this response, or null if the RenderKit should choose the best fit. The RenderKit must support a value for the contentTypeList argument that comes straight from the Accept HTTP header, and therefore requires parsing according to the specification of the Accept header. Please see Section 14.1 of RFC 2616 (he HTTP 1.1 RFC) for the specification of the Accept header.

Implementors are advised to consult the getCharacterEncoding() method of class javax.faces.servlet.ServletResponse to get the required value for the characterEncoding parameter for this method. Since the Writer for this response will already have been obtained (due to it ultimately being passed to this method), we know that the character encoding cannot change during the rendering of the response. Please see Section 6.4 “ResponseWriter”

```
public ResponseStream createResponseStream(OuputStream out);
```

Use the provided `OutputStream` to create a new `ResponseStream` instance.

```
public ResponseStateManager getResponseStateManager();
```

Return an instance of `ResponseStateManager` to handle rendering technology specific state management decisions.

8.2 Renderer

A `Renderer` instance implements the decoding and encoding functionality of components, during the *Apply Request Values* and *Render Response* phases of the request processing lifecycle, when the component has a non-`null` value for the `rendererType` property.

```
public void decode(FacesContext context, UIComponent component);
```

For components utilizing the *delegated implementation* programming model, this method will be called during the *apply request values* phase of the request processing lifecycle, for the purpose of converting the incoming request information for this component back into a new local value. See the API reference for the `Renderer.decode()` method for details on its responsibilities.

```
public void encodeBegin(FacesContext context, UIComponent
component) throws IOException;

public void encodeChildren(FacesContext context, UIComponent
component) throws IOException;

public void encodeEnd(FacesContext context, UIComponent component)
throws IOException;
```

For components utilizing the *delegated implementation* programming model, these methods will be called during the *Render Response* phase of the request processing lifecycle. These methods have the same responsibilities as the corresponding `encodeBegin()`, `encodeChildren()`, and `encodeEnd()` methods of `UIComponent` (described in Section 3.1.12 “Component Specialization Methods” and the corresponding Javadocs) when the component implements the *direct implementation* programming model.

```
public String convertClientId(FacesContext context, String
clientId);
```

Converts a component-generated client identifier into one suitable for transmission to the client.

```
public boolean getRendersChildren();
```

Return a flag indicating whether this Renderer is responsible for rendering the children of the component it is asked to render.

```
 public Object getConvertedValue(FacesContext context,
UIComponent  component, Object submittedValue) throws
ConverterException;
```

Attempt to convert previously stored state information into an object of the type required for this component (optionally using the registered `Converter` for this component, if there is one). If conversion is successful, the new value should be returned from this method; if not, a `ConverterException` should be thrown.

8.3 ResponseStateManager

`ResponseStateManager` is the helper class to `javax.faces.application.StateManager` that knows the specific rendering technology being used to generate the response. It is a singleton abstract class. This class knows the mechanics of saving state, whether it be in hidden fields, session, or some combination of the two.

```
public Object getComponentStateToRestore(FacesContext context);
```

The implementation must inspect the current request and return the component tree state Object passed to it on a previous invocation of writeState().

```
public Object getTreeStructureToRestore(FacesContext context,
String viewId);
```

The implementation must inspect the current request and return the tree structure Object passed to it on a previous invocation of writeState().

```
public void writeState(FacesContext context, SerializedView state)
throws IOException;
```

Take the argument content buffer and replace the state markers that we've written using `writeStateMarker()` with the appropriate representation of the structure and state, writing the output to the output writer.

If the structure and state are to be written out to hidden fields, the implementation must take care to make all necessary character replacements to make the Strings suitable for inclusion as an HTTP request paramater.

8.4 RenderKitFactory

A single instance of `javax.faces.render.RenderKitFactory` must be made available to each JSF-based web application running in a servlet or portlet container. The factory instance can be acquired by JSF implementations, or by application code, by executing

```
RenderKitFactory factory = (RenderKitFactory)
  FactoryFinder.getFactory(FactoryFinder.RENDER_KIT_FACTORY);
```

The `RenderKitFactory` implementation class supports the following methods:

```
public RenderKit getRenderKit(FacesContext context, String
renderKitId);
```

Return a `RenderKit` instance for the specified render kit identifier, possibly customized based on the dynamic characteristics of the specified, (yet possibly null) `FacesContext`. For example, an implementation might choose a different

RenderKit based on the "User-Agent" header included in the request, or the Locale that has been established for the response view. Note that applications which depend on this feature are not guaranteed to be portable across JSF implementations.

Every JSF implementation must provide a RenderKit instance for a default render kit identifier that is designated by the String constant RenderKitFactory.HTML_BASIC_RENDER_KIT. Additional render kit identifiers, and corresponding instances, can also be made available.

```
public Iterator getRenderKitIds();
```

This method returns an Iterator over the set of render kit identifiers supported by this factory. This set must include the value specified by RenderKitFactory.HTML_BASIC_RENDER_KIT.

```
public void addRenderKit(String renderKitId, RenderKit renderKit);
```

Register a RenderKit instance for the specified render kit identifier, replacing any previous RenderKit registered for that identifier.

8.5 Standard HTML RenderKit Implementation

To ensure application portability, all JSF implementations are required to include support for a RenderKit, and the associated Renderers, that meet the requirements defined in this section, to generate textual markup that is compatible with HTML 4.01. JSF implementors, and other parties, may also provide additional RenderKit libraries, or additional Renderers that are added to the standard RenderKit at application startup time, but applications must ensure that the standard Renderers are made available for the web application to utilize them.

The required behavior of the standard HTML RenderKit is specified in a set of external HTML pages that accompany this specification, entitled "The Standard HTML RenderKit". The behavior described in these pages is normative, and are required to be fulfilled by all implementations of JSF.

8.6 The Concrete HTML Component Classes

For each valid combination of `UIComponent` subclass and standard renderer given in the previous section, there is a concrete class in the package `javax.faces.component.html` package. Each class in this package is a subclass of an corresponding class in the `javax.faces.component` package, and adds strongly typed JavaBeans properties for all of the renderer-dependent properties.

TABLE 8-1 Concrete HTML Component Classes

javax.faces.component class	renderer-type	javax.faces.component.html class
UICommand	javax.faces.Button	HtmlCommandButton
UICommand	javax.faces.Link	HtmlCommandLink
UIData	javax.faces.Table	HtmlDataTable
UIForm	javax.faces.Form	HtmlForm
UIGraphic	javax.faces.Image	HtmlGraphicImage
UIInput	javax.faces.Hidden	HtmlInputHidden
UIInput	javax.faces.Secret	HtmlInputSecret
UIInput	javax.faces.Text	HtmlInputText
UIInjput	javax.faces.Textarea	HtmlInputTextarea
UIMessage	javax.faces.Message	HtmlMessage
UIMessages	javax.faces.Messages	HtmlMessages
UIOutput	javax.faces.Format	HtmlOutputFormat
UIOutput	javax.faces.Label	HtmlOutputLabel
UIOutput	javax.faces.Link	HtmlOutputLink
UIOutput	javax.faces.Text	HtmlOutputText
UIPanel	javax.faces.Grid	HtmlPanelGrid
UIPanel	javax.faces.Group	HtmlPanelGroup
UISelectBoolean	javax.faces.Checkbox	HtmlSelectBooleanCheck box
UISelectMany	javax.faces.Checkbox	HtmlSelectManyCheckb ox
UISelectMany	javax.faces.Listbox	HtmlSelectManyListbox
UISelectMany	javax.faces.Menu	HtmlSelectManyMenu

TABLE 8-1 Concrete HTML Component Classes

javax.faces.component class	renderer-type	javax.faces.component.html class
UISelectOne	javax.faces.Listbox	HtmlSelectOneListbox
UISelectOne	javax.faces.Menu	HtmlSelectOneMenu
UISelectOne	javax.faces.Radio	HtmlSelectOneRadio

As with the standard components in the `javax.faces.component` package, each HTML component implementation class must define a static public final String constant named `COMPONENT_TYPE`, whose value is "`javax.faces.`" concatenated with the class name. HTML components, however, must not define a `COMPONENT_FAMILY` constant, or override the `getFamily()` method they inherit from their superclass.

CHAPTER 9

Integration with JSP

JavaServer Faces implementations must support (although JSF-based applications need not utilize) using JavaServer Pages (JSP) as the page description language for JSF pages. This JSP support is provided by providing custom actions so that a JSF user interface can be easy defined in a JSP page by adding custom actions corresponding to JSF UI components. Custom actions provided by a JSF implementation may be mixed with standard JSP actions and custom actions from other libraries, as well as template text for layout, in the same JSP page.

For JSP version 2.0 and onward, the file extension "`.jsf`" is reserved, and may optionally be used (typically by authoring tools) to represent JSP pages containing JSF content[1]. When running in a JSP 1.2 environment, JSP authors must give their JSP pages that contain JSF content a filename ending in "`.jsp`".

1. If this extension is used, it must be declared in the web application deployment descriptor, as described in the JSP 2.0 (or later) specification.

9.1 UIComponent Custom Actions

A JSP custom action for a JSF `UIComponent` is constructed by combining properties and attributes of a Java UI component class with the rendering attributes supported by a specific `Renderer` from a concrete `RenderKit`. For example, assume the existence of a concrete `RenderKit`, `HTMLRenderKit`, which supports three `Renderer` types for the `UIInput` component:

TABLE 9-1 Example Renderer Types

RendererType	**Render-Dependent Attributes**
"Text"	"size"
"Secret"	"size", "secretChar"
"Textarea"	"size", "rows"

The tag library descriptor (TLD) file for the corresponding tag library, then, would define three custom actions—one per `Renderer`. Below is an example of a portion of the custom action definition for the `inputText` tag[2]:

```
<tag>
   <name>inputText</name>
   <tag-class>acme.html.tags.InputTag</tag-class>
   <bodycontent>JSP</bodycontent>
   <attribute>
      <name>id</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
   </attribute>
   <attribute>
      <name>value</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
   </attribute>
   <attribute>
      <name>size</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
   </attribute>
   ...
</tag>
```

Note that the `size` attribute is derived from the `Renderer` of type “Text”, while the `id` and `value` attributes are derived from the `UIInput` component class itself. `RenderKit` implementors will generally provide a JSP tag library which includes component custom actions corresponding to each of the component classes (or types) supported by each of the `RenderKit`’s `Renderers`. See Section 8.1 “RenderKit” and Section 8.2 “Renderer” for details on the `RenderKit` and `Renderer` APIs. JSF implementations must provide such a tag library for the standard HTML RenderKit (see Section 9.5 “Standard HTML RenderKit Tag Library”).

9.2 Using UIComponent Custom Actions in JSP Pages

The following subsections define how a page author utilizes the custom actions provided by the `RenderKit` implementor in the JSP pages that create the user interface of a JSF-based web application.

9.2.1 Declaring the Tag Libraries

This specification hereby reserves the following Uniform Resource Identifier (URI) values to refer to the standard tag libraries for the custom actions defined by JavaServer Faces:

- **http://java.sun.com/jsf/core** -- URI for the *JavaServer Faces Core Tag Library*
- **http://java.sun.com/jsf/html** -- URI for the *JavaServer Faces Standard HTML RenderKit Tag Library*

The page author must use the standard JSP `taglib` directive to declare the URI of each tag library to be utilized, as well as the prefix used (within this page) to identify custom actions from this library. For example,

```
<%@ taglib uri=”http://java.sun.com/jsf/core” prefix=”f” %>
<%@ taglib uri=”http://java.sun.com/jsf/html” prefix=”h” %>
```

2. This example illustrates a non-normative convention for naming custom actions based on a combination of the component name and the renderer type. This convention is useful, but not required; custom actions may be given any desired custom action name; however the convention is rigorously followed in the Standard HTML RenderKit Tag Library.

declares the unique resource identifiers of the tag libraries being used, as well as the prefixes to be used within the current page for referencing actions from these libraries[3].

9.2.2 Including Components in a Page

A JSF `UIComponent` custom action can be placed at any desired position in a JSP page that contains the `taglib` directive for the corresponding tag library, subject to the following restrictions:

- When using a single JSP page to create the entire view, JSF component custom actions must be nested inside the `<f:view>` custom action from the JSF Core Tag Library.
- When using the `<jsp:include>` standard action (or the JSTL `<c:import>` action) to compose a single view from multiple JSP pages, all JSF component custom actions in included pages must be nested inside the `<f:subview>` custom action from the JSF Core Tag Library (which is itself nested inside the `<f:view>` custom action). The `<f:subview>` action itself may be present in the including page (i.e. with the `<jsp:include>` or `<c:import>` action nested inside it), or in the included page.
- For the current version of this specification, any template text (or non-JSF custom actions) present in a page that is included with the `<jsp:include>` or `<c:import>` action, or any other mechanism that uses `RequestDispatcher.include()`, must be enclosed in an `<f:verbatim>` custom action (see Section 9.4.17 “<f:verbatim>”). This restriction may be lifted in future versions of this specification.

The following example illustrates the general use of a UIComponent custom action in a JSP page. In this scenario:

```
<h:inputText id=”username” value=”#{logonBean.username}”/>
```

represents a `UIInput` field, to be rendered with the “Text” renderer type, and points to the username property of a backing bean for the actual value. The `id` attribute specifies the *component id* of a `UIComponent` instance, from within the component tree, to which this custom action corresponds. If no `id` is specified, one will be automatically generated by the custom action implementation.

3. Consistent with the way that namespace prefixes work in XML, the actual prefix used is totally up to the page author, and has no semantic meaning. However, the values shown above are the suggested defaults, which are used consistently in tag library examples throughout this specification.

Custom actions that correspond to JSF `UIComponent` instances must subclass either `javax.faces.webapp.UIComponentTag` (see Section 10.2.6.3 "UIComponentTag") or `javax.faces.webapp.UIComponentBodyTag` (see Section 10.2.6.4 "UIComponentBodyTag"), depending on whether the custom action needs to support `javax.servlet.jsp.tagext.BodyTag` functionality or not.

During the *Render Response* phase of the request processing lifecycle, the appropriate encoding methods of the component (or its associated `Renderer`) will be utilized to generate the representation of this component in the response page. In addition, the first time a particular page is rendered, the component tree may also be dynamically constructed.

All markup other than `UIComponent` custom actions is processed by the JSP container, in the usual way. Therefore, you can use such markup to perform layout control, or include non-JSF content, in conjunction with the actions that represent UI components.

9.2.3 Creating Components and Overriding Attributes

As `UIComponent` custom actions are encountered during the processing of a JSP page, the custom action implementation must check the component tree for the existence of a corresponding `UIComponent`, and (if not found) create and configure a new component instance corresponding to this custom action. The details of this process (as implemented in the findComponent() method of UIComponentTag, for easy reuse) are as follows:

- If the component associated with this component custom action has been identified already, return it unchanged.
- Identify the *component identifier* for the component related to this UIComponent custom action, as follows:
  - If the page author has specified a value for the `id` attribute, use that value.
  - Otherwise, call the `createUniqueId()` method of the `UIViewRoot` at the root of the component tree for this view, and use that value.
- If this `UIComponent` custom action is creating a *facet* (that is, we are nested inside an `<f:facet>` custom action), determine if there is a facet of the component associated with our parent `UIComponent` custom action, with the specified facet name, and proceed as follows:
  - If such a facet already exists, take no additional action.
  - If no such facet already exists, create a new `UIComponent` (by calling the `createComponent()` method on the `Application` instance for this web application, passing the value returned by `getComponentType()`, set the component identifier to the specified value, call `setProperties()` passing

the new component instance, and add the new component as a facet of the component associated with our parent `UIComponent` custom action, under the specified facet name.

- If this `UIComponent` custom action is not creating a facet (that is, we are not nested inside an `<f:facet>` custom action), determine if there is a child component of the component associated with our parent `UIComponent` custom action, with the specified component identifier, and proceed as follows:
  - If such a child already exists, take no additional action.
  - If no such child already exists, create a new `UIComponent` (by calling the `createComponent()` method on the `Application` instance for this web application, passing the value returned by `getComponentType()`, set the component identifier to the specified value, call `setProperties()` passing the new component instance, and add the new component as a child of the component associated with our parent `UIComponent` custom action.

9.2.4 Deleting Components on Redisplay

In addition to the support for dynamically creating new components, as described above, UIComponent custom actions will also *delete* child components (and facets) that are already present in the component tree, but are not rendered on this display of the page. For example, consider a UIComponent custom action that is nested inside a JSTL `<c:if>` custom action whose condition is true when the page is initially rendered. As described in this section, a new UIComponent will have been created and added as a child of the `UIComponent` corresponding to our parent `UIComponent` custom action. If the page is re-rendered, but this time the `<c:if>` condition is `false`, the previous child component will be removed.

9.2.5 Representing Component Hierarchies

Nested structures of UIComponent custom actions will generally mirror the hierarchical relationships of the corresponding UIComponent instances in the view that is associated with each JSP page. For example, assume that a UIForm component (whose component id is logonForm) contains a UIPanel component used to manage the layout. You might specify the contents of the form like this:

```
<h:form id=”logonForm”>
  <h:panelGrid columns=”2”>
    <h:outputLabel for=”username”>
      <h:outputText value=”Username:”/>
    </h:outputLabel>
    <h:inputText id=”username”
       value=”#{logonBean.username}”/>
    <h:outputLabel for=”password”>
      <h:outputText value=”Password:”/>
    </h:outputLabel>
    <h:inputSecret id=”password”
       value=”#{logonBean.password}”/>
    <h:commandButton id=”submitButton” type=”SUBMIT”
       action=”#{logonBean.logon}”/>
    <h:commandButton id=”resetButton” type=”RESET”/>
  </h:panelGrid>
</h:form>
```

9.2.6 Registering Converters, Event Listeners, and Validators

Each JSF implementation is required to provide the core tag library (see Section 9.4 “JSF Core Tag Library”), which includes custom actions that (when executed) create instances of a specified Converter, ValueChangeListener, ActionListener or Validator implementation class, and register the created instance with the UIComponent associated with the most immediately surrounding UIComponent custom action.

Using these facilities, the page author can manage all aspects of creating and configuring values associated with the view, without having to resort to Java code. For example:

```
<h:inputText id=”username” value=”#{logonBean.username}”>
   <f:validateLength minimum=”6”/>
</h:inputText>
```

associates a validation check (that the value entered by the user must contain at least six characters) with the username `UIInput` component being described.

Following are usage examples for the `valueChangeListener` and `actionListener` custom actions.

```
<h:inputText id=”maxUsers”>
   <f:convertNumber integerOnly=”true”/>
   <f:valueChangeListener
      type="custom.MyValueChangeListener"/>
</h:inputText>
<h:commandButton label="Login">
   <f:actionListener type="custom.MyActionListener"/>
</h:commandButton>
```

This example causes a `Converter` and a `ValueChangeListener` of the user specified type to be instantiated and added as to the enclosing `UIInput` component, and an `ActionListener` is instantiated and added to the enclosing `UICommand` component. If the user specified type does not implement the proper listener interface a `JSPException` must be thrown.

9.2.7 Using Facets

A *Facet* is a subordinate UIComponent that has a special relationship to its parent `UIComponent`, as described in Section 3.1.9 “Facet Management”. Facets can be defined in a JSP page using the `<f:facet>` custom action. Each facet action must have one and only one child UIComponent custom action[4]. For example:

```
<h:dataTable ...>
   <f:facet name=”header”>
      <h:outputText value=”Customer List”/>
   </f:facet>
   <h:column>
      <f:facet name=”header”>
         <h:outputText value=”Account Id”/>
      </f:facet>
      <h:outputText id=”accountId” value=
         ”#{customer.accountId}”/>
   </h:column>
   ...
</h:dataTable>
```

9.2.8 Interoperability with JSP Template Text and Other Tag Libraries

It is permissible to use other tag libraries, such as the JSP Standard Tag Library (JSTL) in the same JSP page with `UIComponent` custom actions that correspond to JSF components, subject to certain restrictions. When JSF component actions are nested inside custom actions from other libraries, or combined with template text, the following behaviors must be supported:

- JSF component custom actions nested inside a custom action that conditionally renders its body (such as JSTL’s `<c:if>` or `<c:choose>`) must contain a manually assigned `id` attribute.
- JSF component custom actions may **not** be nested inside a custom action that iterates over its body (such as JSTL’s <c:forEach>). Instead, you should use a Renderer that performs its own iteration (such as the Table renderer used by `<h:dataTable>`).

4. If you need multiple components in a facet, nest them inside a <h:panelGroup> custom action that is the value of the facet.

- Components that are added to the component tree programmatically (as opposed to by being represented by `UIComponent` custom actions) will not be rendered, unless they are children of a `UIComponent`, or its corresponding `Renderer`, returns `true` from the `getRendersChildren()` method, and takes responsibility for performing the corresponding rendering.
- Nesting JSP template text and non-UIComponent custom actions (or UIComponent custom actions that buffer their output) inside a UIComponent custom action for which the rendersChildren property (of the renderer or the component) is `true` is not allowed. For most scenarios where this would be desirable, the <f:verbatim> custom action from the JSF Core Tag Library (see Section 9.4.17 "<f:verbatim>") may be used
- Interoperation with the JSTL Internationalization-Capable Formatting library (typically used with the "`fmt`" prefix) is restricted as follows:
  - The `<fmt:parseDate>` and `<fmt:parseNumber>` custom actions should not be used. The corresponding JSF facility is to use an `<h:inputText>` component custom action with an appropriate `DateTimeConverter` or `NumberConverter`.
  - The `<fmt:requestEncoding>` custom action should not be used. By the time it is executed, the request parameters will have already been parsed, so any change in the setting here will have no impact. JSF handles character set issues automatically in most cases. To use a fixed character set in exceptional circumstances, use the a "`<%@ page contentType="[content-type];[charset]" %>`" directive.
  - The `<fmt:setLocale/>` custom action should not be used. Even though it might work in some circumstances, it would result in JSF and JSTL assuming different locales. If the two locales use different character sets, the results will be undefined. Applications should use JSF facilities for setting the `locale` property on the `UIViewRoot` component to change locales for a particular user.

9.2.9 Composing Pages from Multiple Sources

JSP pages can be composed from multiple sources using several mechanisms:

- The `<%@include%>` directive performs a compile-time inclusion of a specified source file into the page being compiled[5]. From the perspective of JSF, such inclusions are transparent—the page is compiled as if the inclusions had been performed before compilation was initiated.
- Several mechanisms (including the `<jsp:include>` standard action, the JSTL `<c:import>` custom action when referencing a resource in the same webapp, and a call to `RequestDispatcher.include()` for a resource in the same webapp)

5. In a JSP 2.0 or later environment, the same effect can be accomplished by using <include-prelude> and <include-coda> elements in the <jsp-config> element in the web application deployment descriptor.

perform a runtime dynamic inclusion of the results of including the response content of the requested page resource in place of the include action. Any JSF components created by execution of JSF component custom actions in the included resource will be grafted onto the component tree, just as if the source text of the included page had appeared in the calling page at the position of the include action.

- For mechanisms that aggregate content by other means (such as use of an `HttpURLConnection`, a `RequestDispatcher.include()` on a resource from a different web application, or accessing an external resource with the JSTL `<c:import>` custom action on a resource from a different web application, only the response content of the aggregation request is available. Therefore, any use of JSF components in the generation of such a response are not combined with the component tree for the current page.

9.3 UIComponent Custom Action Implementation Requirements

The custom action implementation classes for `UIComponent` custom actions must conform to all of the requirements defined in the JavaServer Pages Specification. In addition, they must meet the following JSF-specific requirements:

- Extend the `UIComponentTag` or `UIComponentBodyTag` base class, so that JSF implementations can recognize `UIComponent` custom actions versus others.
- Provide a public `getComponentType()` method that returns a String-valued component type registered with the `Application` instance for this web application. The value returned by this method will be passed to `Application.createComponent()` when a new `UIComponent` instance associated with this custom action is to be created.
- Provide a public `getRendererType()` method that returns a String-valued renderer type registered with the `RenderKit` instance for the currently selected `RenderKit`, or `null` if there should be no associated `Renderer`. The value returned by this method will be used to set the `rendererType` property of any UIComponent created by this custom action.
- Provide setter methods taking a String-valued parameter for all set-able (from a custom action) properties of the corresponding `UIComponent` class, and all additional set-able (from a custom action) attributes supported by the corresponding `Renderer`.
- Provide a protected `setProperties()` method of type `void` that takes a `UIComponent` instance as parameter. The implementation of this method must perform the following tasks:

- Call `super.setProperties()`, passing the same `UIComponent` instance received as a parameter.
- For each non-null custom action attribute that corresponds to a property based attribute to be set on the underlying component, call either `setValueBinding()` or `getAttributes().put()`, depending on whether or not a value binding expression was specified as the custom action attribute value (performing any required type conversion). For example, assume that title is the name of a render-dependent attribute for this component:

```
protected void setProperties(UIComponent component) {
  super.setProperties(component);
  if (title != null) {
    if (isValueReference(title)) {
      ValueBinding vb =
        getFacesContext().getApplication().
        createValueBinding(title);
      component.setValueBinding(“title”, vb);
    } else {
      component.getAttributes().put(“title”, title);
    }
  }
  ...
}
```

- For each non-null custom action attribute that corresponds to a method based attribute to be set on the underlying component, the value of the attribute must be a method reference expression. Call `setMethodBinding()`, or throw a

FacesException if the value of the attribute is not a method reference exception For example, assume that valueChangeListener is the name of an attribute for this component:

```
protected void setProperties(UIComponent component) {
  super.setProperties(component);
  if (valueChangeListener != null) {
    if (isValueReference(valueChangeListener)) {
      Class args[] = { ValueChangeEvent.class };
      MethodBinding vb =
      FacesContext.getCurrentInstance().getApplication().createV
alueBinding(valueChangeListener, args);
      input.setValueChangeListener(vb);
    } else {
      Object params [] = {valueChangeListener};
      throw new
        javax.faces.FacesException(Util.getExceptionMessage(Util
.INVALID_EXPRESSION_ID, params));
    }
  }
  ...
}
```

- Non-null custom action attributes that correspond to a writable property to be set on the underlying component are handled in a similar fashion. For example, assume a custom action for the UIData component is being created that needs to deal with the rows property (which is of type int):

```
protected void setProperties(UIComponent component) {
  super.setProperties(component);
  if (rows != null) {
    if (isValueReference(rows)) {
      ValueBinding vb =
        FacesContext.getCurrentInstance().getApplication().
        createValueBinding(rows);
      component.setValueBinding("rows", vb);
    } else {
      ((UIData) component).setRows(Integer.parseInt(rows));
    }
  }
  ...
}
```

- Optionally, provide a public `release()` method of type `void`, taking no parameters, to be called when the JSP page handler releases this custom action instance. If implemented, the method must perform the following tasks:
  - Call `super.release()` to invoke the superclass's release functionality.
  - Clear the instance variables representing the values for set-able custom action attributes (for example, by setting String values to null).
- Optionally provide overridden implementations for the following methods to fine tune the behavior of your `UIComponent` custom action implementation class: `encodeBegin()`, `encodeChildren()`, `encodeEnd()`, `getDoEndValue()`, and `getDoStartValue()`.

It is technically possible to override other public and protected methods of the `UIComponentTag` or `UIComponentBodyTag` base class; however, it is likely that overriding these methods will interfere with the functionality that other portions of the JSF implementation are assuming to be present, so overriding these methods is strongly discouraged.

The definition of each `UIComponent` custom action in the corresponding tag library descriptor (TLD) must conform to the following requirements:

- The `<body-content>` element for the custom action itself must specify `JSP`.
- The `<rtexprvalue>` element for each custom action attribute that is destined to be passed through to the underlying UIComponent (as a property or a component attribute) must be set to `false`.

9.4 JSF Core Tag Library

All JSF implementations must provide a tag library containing core actions (described below) that are independent of a particular `RenderKit`. The corresponding tag library descriptor must meet the following requirements:

- Must declare a tag library version (`<tlib-version>`) value of `1.0`.
- Must declare a JSP version dependency (`<jsp-version>`) value of `1.2`.
- Must declare a URI (`<uri>`) value of `http://java.sun.com/jsf/core`.
- Must be included in the `META-INF` directory of a JAR file containing the corresponding implementation classes, suitable for inclusion with a web application, such that the tag library descriptor will be located automatically by the algorithm described in Section 7.3 of the *JavaServer Pages Specification* (version 1.2).

Each custom action included in the JSF Core Tag Library is documented in a subsection below, with the following outline for each action:

- **Name**—The name of this custom action, as used in a JSP page.

- **Short Description**—A summary of the behavior implemented by this custom action.
- **Syntax**—One or more examples of using this custom action, with the required and optional sets of attributes that may be used together.
- **Body Content**—The type of nested content for this custom action, using one of the standard values `empty`, `JSP`, or `tagdependent` as described in the JSP specification. This section also describes restrictions on the types of content (template text, JSF core custom actions, JSF `UIComponent` custom actions, and/or other custom actions) that can be nested in the body of this custom action.
- **Attributes**—A table containing one row for each defined attribute for this custom action. The following columns provide descriptive information about each attribute:
  - *Name*—Name of this attribute, as it must be used in the page. If the name of the attribute is in *italics*, it is required.
  - *Expr*—The type of dynamic expression (if any) that can be used in this attribute value. Legal values are stephane.bastian@otrix.com (this may be a literal or a value binding expression), MB (this may be a method binding expression), or NONE (this attribute accepts literal values only).
  - *Type*—Fully qualified Java class or primitive type of this attribute.
  - *Description*—The functional meaning of this attribute's value.
- **Constraints**—Additional constraints enforced by this action, such as combinations of attributes that may be used together.
- **Description**—Details about the functionality provided by this custom action.

9.4.1 <f:actionListener>

Register an `ActionListener` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

```
<f:actionListener type="fully-qualified-classname"/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
type	VB	String	Fully qualified Java class name of an `ActionListener` to be created and registered

Constraints

- Must be nested inside a `UIComponent` custom action.
- The corresponding `UIComponent` implementation class must implement `ActionSource`, and therefore define a public `addActionListener()` method that accepts an `ActionListener` parameter.
- The specified listener class must implement `javax.faces.event.ActionListener`.

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentTag.getParentUIComponentTag()`. If the `getCreated()` method of this instance returns `true`, instantiate an instance of the specified class, and register it by calling `addActionListener()`.

As an alternative, you may also register a method in a backing bean class to receive `ActionEvent` notifications, by using the `actionListener` attribute on the corresponding `UIComponent` custom action.

9.4.2 <f:attribute>

Add an attribute on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

```
<f:attribute name="attribute-name" value="attribute-value"/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
name	VB	String	Name of the component attribute to be set
value	VB	Object	Value of the component attribute to be set

Constraints

- Must be nested inside a `UIComponent` custom action.

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentTag.getParentUIComponentTag()`. If the associated component does not already have a component attribute with a name specified by this custom action's `name` attribute, create a component attribute with the name and value specified by this custom action's attributes.

The implementation class for this action must be, or extend, `javax.faces.webapp.AttributeTag`.

9.4.3 <f:convertDateTime>

Register a `DateTimeConverter` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

```
<f:convertDateTime
  [dateStyle=”{default|short|medium|long|full}”]
  [locale=”{locale” | string}]
  [pattern=”pattern”]
  [timeStyle=”{default|short|medium|long|full}”]
  [timeZone=”{timeZone| string}”]
  [type=”{date|time|both}”]/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
date-Style	VB	String	Predefined formatting style which determines how the date component of a date string is to be formatted and parsed. Applied only if type is "date" or "both".
locale	VB	Locale or String	Locale whose predefined styles for dates and times are used during formatting or parsing. If not specified, the Locale returned by FacesContext.getViewRoot().getLocale() will be used. Value must be either a VB expression that evaluates to a java.util.Locale instance, or a String that is valid to pass as the first argument to the constructor java.util.Locale(String language, String country). The empty string is passed as the second argument.
pattern	VB	String	Custom formatting pattern which determines how the date/time string should be formatted and parsed.
time-Style	VB	String	Predefined formatting style which determines how the time component of a date string is to be formatted and parsed. Applied only if type is "time" or "both".
time-Zone	VB	timezone or String	Time zone in which to interpret any time information in the date string. Value must be either a VB expression that evaluates to a java.util.TimeVone instance, or a String that is a timezone ID as described in the javadocs for java.util.TimeZone.getTimeZone().
type	VB	String	Specifies whether the string value will contain a date, time, or both.

Constraints

- Must be nested inside a `UIComponent` custom action whose component class implements `ValueHolder`, and whose value is a `java.util.Date` (or appropriate subclass).

- If `pattern` is specified, the pattern syntax must use the pattern syntax specified by `java.text.SimpleDateFormat`.
- If `pattern` is not specified, formatted strings will contain a date value, a time value, or both depending on the specified `type`. When date or time values are included, they will be formatted according to the specified `dateStyle` and `timeStyle`, respectively.
- if `type` is not specified:
  - if `dateStyle` is set and `timeStyle` is not, `type` defaults to `date`
  - if `timeStyle` is set and `dateStyle` is not, `type` defaults to `time`
  - if both `dateStyle` and `timeStyle` are set, `type` defaults to `both`

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentTag.getParentUIComponentTag()`. If the `getCreated()` method of this instance returns `true`, create, call `createConverter()` and register the returned Converter instance on the associated UIComponent.

The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.webapp.ConverterTag`.
- The `createConverter()` method must call the `createConverter()` method of the `Application` instance for this application, passing converter id "javax.faces.DateTime". It must then cast the returned instance to `javax.faces.convert.DateTimeConverter` and configure its properties based on the specified attributes for this custom action, and return the configured instance.
- If the type attribute is not specified, it defaults as follows:
  - If dateStyle is specified but timeStyle is not specified, default to date.
  - If dateStyle is not specified but timeStyle is specified, default to time.
  - If both dateStyle and timeStyle are specified, default to both.
  - It is an error if

9.4.4 <f:convertNumber>

Register a `NumberConverter` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

```
<f:convertNumber
  [currencyCode="currencyCode"]
  [currencySymbol="currencySymbol"]
  [groupingUsed="{true|false}"]
  [integerOnly="{true|false}"]
  [locale="locale"]
  [maxFractionDigits="maxFractionDigits"]
  [maxIntegerDigits="maxIntegerDigits"]
  [minFractionDigits="minFractionDigits"]
  [minIntegerDigits="minIntegerDigits"]
  [pattern="pattern"]
  [type="{number|currency|percent}"]/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
currencyCode	VB	String	ISO 4217 currency code, applied only when formatting currencies.
currencySymbol	VB	String	Currency symbol, applied only when formatting currencies.
groupingUsed	VB	boolean	Specifies whether formatted output will contain grouping separators.
integerOnly	VB	boolean	Specifies whether only the integer part of the value will be parsed.
locale	VB	java.util.Locale	Locale whose predefined styles for numbers are used during formatting or parsing. If not specified, the Locale returned by FacesContext.getViewRoot().getLocale() will be used.
maxFractionDigits	VB	int	Maximum number of digits that will be formatted in the fractional portion of the output.
maxIntegerDigits	VB	int	Maximum number of digits that will be formatted in the in.teger portion of the output
minFractionDigits	VB	int	Minimum number of digits that will be formatted in the fractional portion of the output.
minIntegerDigits	VB	int	Minimum number of digits that will be formatted in the integer portion of the output.
pattern	VB	String	Custom formatting pattern which determines how the number string should be formatted and parsed.
type	VB	String	Specifies whether the value will be parsed and formatted as a number, currency, or percentage.

Constraints

- Must be nested inside a `UIComponent` custom action whose component class implements `ValueHolder`, and whose value is a numeric wrapper class or primitive.
- If `pattern` is specified, the pattern syntax must use the pattern syntax specified by `java.text.DecimalFormat`.

- If `pattern` is not specified, formatting and parsing will be based on the specified `type`.

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentTag.getParentUIComponentTag()`. If the `getCreated()` method of this instance returns `true`, create, call `createConverter()` and register the returned Converter instance on the associated UIComponent.

The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.webapp.ConverterTag`.
- The `createConverter()` method must call the `createConverter()` method of the `Application` instance for this application, passing converter id "javax.faces.Number". It must then cast the returned instance to `javax.faces.convert.NumberConverter` and configure its properties based on the specified attributes for this custom action, and return the configured instance.

9.4.5 <f:converter>

Register a named `Converter` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

```
<f:converter converterId="converterId"/>
```

Body Content

empty

Attributes

Name	Expr	Type	Description
converterId	VB	String	Converter identifier of the converter to be created.

Constraints

- Must be nested inside a `UIComponent` custom action whose component class implements `ValueHolder`.

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentTag.getParentUIComponentTag()`. If the `getCreated()` method of this instance returns `true`, create, call `createConverter()` and register the returned Converter instance on the associated UIComponent.

The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.webapp.ConverterTag`.
- The `createConverter()` method must call the `createConverter()` method of the `Application` instance for this application, passing converter id specified by their converterId attribute.

The implementation class for this action must be, or extend, `javax.faces.webapp.ConverterTag`.

9.4.6 <f:facet>

Register a named facet (see Section 3.1.9 "Facet Management") on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

```
<f:facet name="facet-name"/>
```

Body Content

JSP. However, only a single UIComponent custom action (and any related nested JSF custom actions) is allowed; no template text or other custom actions may be present.

Attributes

Name	Expr	Type	Description
name	NONE	String	Name of the facet to be created

Constraints

- Must be nested inside a `UIComponent` custom action.
- Exactly one `UIComponent` custom action must be nested inside this custom action (although the nested component custom action could itself have nested children).

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentTag.getParentUIComponentTag()`. If the associated component does not already have a facet with a name specified by this custom action's `name` attribute, create a facet with this name from the `UIComponent` custom action that is nested within this custom action.

The implementation class must be, or extend, `javax.faces.webapp.FacetTag`.

9.4.7 <f:loadBundle>

Load a resource bundle localized for the locale of the current view, and expose it (as a Map) in the request attributes for the current request.

Syntax

```
<f:loadBundle basename="resource-bundle-name" var="attributeKey"/>
```

Body Content

empty

Attributes

Name	Expr	Type	Description
basename	VB	`String`	Base name of the resource bundle to be loaded.
var	NONE	String	Name of a request scope attribute under which the resource bundle will be exposed as a Map.

Constraints

- Must be nested inside an `<f:view>` custom action.

Description

Load the resource bundle specified by the `basename` attribute, localized for the Locale of the `UIViewRoot` component of the current view, and expose its key-values pairs as a `Map` under the attribute key specified by the `var` attribute. In this way, value binding expressions may be used to conveniently retrieve localized values.

If the `get()` method for the `Map` instance exposed by this custom action is passed a key value that is not present (that is, there is no underlying resource value for that key), the literal string "???foo???" (where "foo" is replaced by the key the String representation of the key that was requested) must be returned, rather than the standard `Map` contract return value of `null`.

9.4.8 <f:param>

Add a child `UIParameter` component to the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

Syntax 1: Unnamed value

```
<f:param [id="componentId"] value="parameter-value"
  [binding="componentReference"]/>
```

Syntax 2: Named value

```
<f:param [id="componentId"]
  [binding="componentReference"]
  name="parameter-name" value="parameter-value"/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
binding	VB	ValueBinding	Value binding expression to a backing bean property bound to the component instance for the UIComponent created by this custom action
`id`	`NONE`	`String`	Component identifier of a `UIParameter` component
`name`	`VB`	`String`	Name of the parameter to be set
`value`	`VB`	`String`	Value of the parameter to be set

Constraints

- Must be nested inside a `UIComponent` custom action.

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentTag.getParentUIComponentTag()`. If the `getCreated()` method of this instance returns `true`, create a new `UIParameter` component, and attach it as a child of the associated `UIComponent`.

The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.UIComponentTag`.
- The `getComponentType()` method must return “`Parameter`”.
- The `getRendererType()` method must return `null`.

9.4.9 <f:selectItem>

Add a child `UISelectItem` component to the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

Syntax 1: Directly Specified Value

```
<f:selectItem [id="componentId"]
  [binding="componentReference"]
  [itemDisabled="{true|false}"]
  itemValue="itemValue"
  itemLabel="itemLabel"
  [itemDescription="itemDescription"]/>
```

Syntax 2: Indirectly Specified Value

```
<f:selectItem [id="componentId"]
  [binding="componentReference"]
  value="selectItemValue"/>
```

Body Content

empty

Attributes

Name	Expr	Type	Description
binding	VB	ValueBinding	Value binding expression to a backing bean property bound to the component instance for the UIComponent created by this custom action.
`id`	`NONE`	`String`	Component identifier of a `UISelectItem` component.
`itemDescription`	`VB`	`String`	Description of this option (for use in development tools).
itemDisabled	VB	boolean	Flag indicating whether the option created by this component is disabled.
itemLabel	VB	String	Label to be displayed to the user for this option.
itemValue	VB	Object	Value to be returned to the server if this option is selected by the user.
value	VB	javax.faces.model.SelectItem	Value binding pointing at a SelectItem instance containing the information for this option.

Constraints

- Must be nested inside a `UIComponent` custom action that creates a `UISelectMany` or `UISelectOne` component instance.

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentTag.getParentUIComponentTag()`. If the `getCreated()` method of this instance returns `true`, create a new `UISelectItem` component, and attach it as a child of the associated `UIComponent`.

The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.UIComponentTag`.
- The `getComponentType()` method must return “`SelectItem`”.
- The `getRendererType()` method must return `null`.

9.4.10 <f:selectItems>

Add a child `UISelectItems` component to the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

<f:selectItems [id=”*componentId*”]
  [binding=”*componentReference*”]
  value=”*selectItemsValue*”/>

Body Content

empty

Attributes

Name	Expr	Type	Description
binding	VB	ValueBinding	Value binding expression to a backing bean property bound to the component instance for the UIComponent created by this custom action.
`id`	NONE	`String`	Component identifier of a `UISelectItem` component.
`value`	VB	javax.faces.model.SelectItem, see description for specific details	Value binding expression pointing at one of the following instances: 1. an individual javax.faces.model.SelectItem 2. a java language array of javax.faces.model.SelectItem 3. a java.util.Collection of javax.faces.model.SeleccItem 4. A java.util.Map where the keys are converted to Strings and used as labels, and the corresponding values are converted to Strings and used as values for newly created javax.faces.model.SelectItem instances. The instances are created in the order of the iterator over the keys provided by the Map.

Constraints

- Must be nested inside a `UIComponent` custom action that creates a `UISelectMany` or `UISelectOne` component instance.

Description

Locate the closest parent UIComponent custom action instance by calling UIComponentTag.getParentUIComponentTag(). If the getCreated() method of this instance returns true, create a new UISelectItems component, and attach it as a child of the associated UIComponent.

The implementation class for this action must meet the following requirements:

- Must extend javax.faces.UIComponentTag.
- The getComponentType() method must return "javax.faces.SelectItems".
- The getRendererType() method must return null.

9.4.11 <f:subview>

Container action for all JSF core and component custom actions used on a nested page included via `<jsp:include>` or any custom action that dynamically includes another page from the same web application, such as JSTL's `<c:import>`.

Syntax

```
<f:subview id="componentId"
               [binding="componentReference"]
               [rendered="{true|false}"]>
   Nested template text and custom actions
</f:subview>
```

Body Content

JSP. May contain any combination of template text, other JSF custom actions, and custom actions from other custom tag libraries.

Attributes

Name	Expr	Type	Description
binding	VB	ValueBinding	Value binding expression to a backing bean property bound to the component instance for the UIComponent created by this custom action.
id	NONE	String	Component identifier of a `UINamingContainer` component
rendered	VB	Boolean	Whether or not this subview should be rendered.

Constraints

- Must be nested inside a `<f:view>` custom action (although this custom action might be in a page that is including the page containing the `<f:subview>` custom action.
- Must not contain an `<f:view>` custom action.
- Must have an `id` attribute whose value is unique within the scope of the parent naming container.
- May be placed in a parent page (with `<jsp:include>` or `<c:import>` nested inside), or within the nested page.

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentTag.getParentUIComponentTag()`. If the `getCreated()` method of this instance returns `true`, create a new `UINamingContainer` component, and attach it as a child of the associated `UIComponent`. Such a component provides a scope within which child component identifiers must still be unique, but allows child components to have the same simple identifier as child components nested in some other naming container. This is useful in several scenarios:

```
“main.jsp”
<f:view>
  <c:import url=”foo.jsp”/>
  <c:import url=”bar.jsp”/>
</f:view>

“foo.jsp”
<f:subview id=”aaa”>
  ... components and other content ...
</f:subview>

“bar.jsp”
<f:subview id=”bbb”>
  ... components and other content ...
</f:subview>
```

In this scenario, <f:subview> custom actions in imported pages establish a naming scope for components within those pages. Identifiers for <f:subview> custom actions nested in a single <f:view> custom action must be unique, but it is difficult for the page author (and impossible for the JSP page compiler) to enforce this restriction.

```
“main.jsp”
<f:view>
  <f:subview id=”aaa”>
    <c:import url=”foo.jsp”/>
  </f:subview>
  <f:subview id=”bbb”>
    <c:import url=”bar.jsp”/>
  </f:subview>
</f:view>

“foo.jsp”
... components and other content ...

“bar.jsp”
... components and other content ...
```

In this scenario, the <f:subview> custom actions are in the including page, rather than the included page. As in the previous scenario, the “id” values of the two subviews must be unique; but it is much easier to verify using this style.

It is also possible to use this approach to include the same page more than once, but maintain unique identifiers:

```
“main.jsp”
<f:view>
  <f:subview id=”aaa”>
    <c:import url=”foo.jsp”/>
  </f:subview>
  <f:subview id=”bbb”>
    <c:import url=”foo.jsp”/>
  </f:subview>
</f:view>

“foo.jsp”
... components and other content ...
```

In all of the above examples, note that foo.jsp and bar.jsp may not contain <f:view>.

The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.UIComponentTag`.
- The `getComponentType()` method must return "`NamingContainer`".
- The `getRendererType()` method must return `null`.

9.4.12 <f:validateDoubleRange>

Register a `DoubleRangeValidator` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

Syntax 1: Maximum only specified

```
<f:validateDoubleRange maximum="543.21"/>
```

Syntax 2: Minimum only specified

```
<f:validateDoubleRange minimum="123.45"/>
```

Syntax 3: Both maximum and minimum are specified

```
<f:validateDoubleRange maximum="543.21" minimum="123.45"/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
`maximum`	`VB`	`double`	Maximum value allowed for this component
`minimum`	`VB`	`double`	Minimum value allowed for this component

Constraints

- Must be nested inside a `EditableValueHolder` custom action whose value is (or is convertible to) a double.
- Must specify either the `maximum` attribute, the `minimum` attribute, or both.
- If both limits are specified, the maximum limit must be greater than the minimum limit.

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentTag.getParentUIComponentTag()`. If the `getCreated()` method of this instance returns `true`, create, call `createValidator()` and register the returned `Validator` instance on the associated `UIComponent`.

The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.webapp.ValidatorTag`.
- The `createValidator()` method must call the `createValidator()` method of the `Application` instance for this application, passing validator id “javax.faces.DoubleRange”. It must then cast the returned instance to `javax.faces.validator.DoubleRangeValidator` and configure its properties based on the specified attributes for this custom action, and return the configured instance.

9.4.13 <f:validateLength>

Register a `LengthValidator` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

Syntax 1: Maximum only specified

```
<f:validateLength maximum="16"/>
```

Syntax 2: Minimum only specified

```
<f:validateLength minimum="3"/>
```

Syntax 3: Both maximum and minimum are specified

```
<f:validateLength maximum="16" minimum="3"/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
`maximum`	`VB`	`int`	Maximum length allowed for this component
`minimum`	`VB`	`int`	Minimum length allowed for this component

Constraints

- Must be nested inside a `EditableValueHolder` custom action whose value is a `String`.
- Must specify either the `maximum` attribute, the `minimum` attribute, or both.
- If both limits are specified, the maximum limit must be greater than the minimum limit.

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentTag.getParentUIComponentTag()`. If the `getCreated()` method of this instance returns `true`, create, call `createValidator()` and register the returned `Validator` instance on the associated `UIComponent`.

The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.webapp.ValidatorTag`.
- The `createValidator()` method must call the `createValidator()` method of the `Application` instance for this application, passing validator id "javax.faces.Length". It must then cast the returned instance to `javax.faces.validator.LengthValidator` and configure its properties based on the specified attributes for this custom action, and return the configured instance.

9.4.14 <f:validateLongRange>

Register a `LongRangeValidator` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

Syntax 1: Maximum only specified

```
<f:validateLongRange maximum="543"/>
```

Syntax 2: Minimum only specified

```
<f:validateLongRange minimum="123"/>
```

Syntax 3: Both maximum and minimum are specified

```
<f:validateLongRange maximum="543" minimum="123"/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
`maximum`	`VB`	`long`	Maximum value allowed for this component
`minimum`	`VB`	`long`	Minimum value allowed for this component

Constraints

- Must be nested inside a `EditableValueHolder` custom action whose value is (or is convertible to) a long.
- Must specify either the `maximum` attribute, the `minimum` attribute, or both.
- If both limits are specified, the maximum limit must be greater than the minimum limit.

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentTag.getParentUIComponentTag()`. If the `getCreated()` method of this instance returns `true`, create, call `createValidator()` and register the returned `Validator` instance on the associated `UIComponent`.

The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.webapp.ValidatorTag`.
- The `createValidator()` method must call the `createValidator()` method of the `Application` instance for this application, passing validator id "javax.faces.LongRange". It must then cast the returned instance to `javax.faces.validator.LongRangeValidator` and configure its properties based on the specified attributes for this custom action, and return the configured instance.

9.4.15 <f:validator>

Register a named `Validator` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

```
<f:validator validatorId="validatorId"/>
```

Body Content

empty

Attributes

Name	Expr	Type	Description
validatorId	VB	String	Validator identifier of the validator to be created.

Constraints

- Must be nested inside a `UIComponent` custom action whose component class implements `EditableValueHolder`.

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentTag.getParentUIComponentTag()`. If the `getCreated()` method of this instance returns `true`, create, call `createValidator()` and register the returned `Validator` instance on the associated `UIComponent`.

The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.webapp.ValidatorTag`.
- The `createValidator()` method must call the `createValidator()` method of the `Application` instance for this application, passing validator id specified by the `validatorId` attribute, and return the configured instance.

9.4.16 <f:valueChangeListener>

Register a `ValueChangeListener` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

```
<f:valueChangeListener type="fully-qualified-classname"/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
type	VB	String	Fully qualified Java class name of a `ValueChangeListener` to be created and registered

Constraints

- Must be nested inside a `UIComponent` custom action.
- The corresponding `UIComponent` implementation class must implement `EditableValueHolder`, and therefore define a public `addValueChangeListener()` method that accepts an `ValueChangeListener` parameter.
- The specified listener class must implement `javax.faces.event.ValueChangeListener`.

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentTag.getParentUIComponentTag()`. If the `getCreated()` method of this instance returns `true`, instantiate an instance of the specified class, and register it by calling `addValueChangeListener()`.

As an alternative, you may also register a method in a backing bean class to receive `ValueChangeEvent` notifications, by using the `valueChangeListener` attribute on the corresponding `UIComponent` custom action.

9.4.17 <f:verbatim>

Register a child `UIOutput` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action which renders nested body content.

Syntax

```
<f:verbatim [escape="{true|false}"]/>
```

Body Content

JSP. However, no UIComponent custom actions, or custom actions from the JSF Core Tag Library, may be nested inside this custom action.

Attributes

Name	Expr	Type	Description
`escape`	`VB`	`boolean`	If `true`, generated markup is escaped in a manner appropriate for the markup language being rendered. Default value is false.

Constraints

- Must be implemented as a `UIComponentBodyTag`.

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentTag.getParentUIComponentTag()`. If the `getCreated()` method of this instance returns `true`, creates a new `UIOutput` component, and add it as a child of the `UIComponent` associated with the located instance. The `rendererType` property of this `UIOutput` component must be set to "javax.faces.Text", and the `transient` property must be set to `true`. Also, the value (or value binding, if it is an expression) of the `escape` attribute must be passed on to the renderer as the value the `escape` attribute on the `UIOutput` component.

9.4.18 <f:view>

Container for all JSF core and component custom actions used on a page.

Syntax

```
<f:view [locale="locale">
   Nested template text and custom actions
</f:view>
```

Body Content

JSP. May contain any combination of template text, other JSF custom actions, and custom actions from other custom tag libraries.

Attributes

Name	Expr	Type	Description
`locale`	VB	`String` or `Locale`	Name of a Locale to use for localizing this page (such as en_uk), or value binding expression that returns a `Locale` instance

Constraints

- Any JSP-created response using actions from the JSF Core Tag Library, as well as actions extending `javax.faces.webapp.UIComponentTag` from other tag libraries, must be nested inside an occurrence of the `<f:view>` action.
- JSP page fragments included via the standard `<%@ include %>` directive need not have their JSF actions embedded in a `<f:view>` action, because the included template text and custom actions will be processed as part of the outer page as it is compiled, and the `<f:view>` action on the outer page will meet the nesting requirement.
- JSP pages included via `<jsp:include>` or any custom action that dynamically includes another page from the same web application, such as JSTL's `<c:import>`, must use an `<f:subview>` (either inside the included page itself, or surrounding the <jsp:include> or custom action that is including the page).
- If the `locale` attribute is present, its value overrides the `Locale` stored in `UIViewRoot`, normally set by the `ViewHandler, and the doStartTag() method must store it by calling UIViewRoot.setLocale()`.
- The `doStartTag()` method must call `javax.servlet.jsp.jstl.core.Config.set()`, passing the `ServletRequest` instance for this request, the constant `javax.servlet.jsp.jstl.core.Config.FMT_LOCALE`, and the `Locale` returned by calling `UIViewRoot.getLocale()`.

Description

Provides the JSF implementation a convenient place to perform state saving during the *render response* phase of the request processing lifecycle, if the implementation elects to save state as part of the response.

The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.UIComponentBodyTag`.
- The `getComponentType()` method must return “ViewRoot”.
- The `getRendererType()` method must return `null`.

Please refer to the javadocs for `javax.faces.application.StateManager` for details on what the tag handler for this tag must do to implement state saving.

9.5 Standard HTML RenderKit Tag Library

All JSF implementations must provide a tag library containing actions that correspond to each valid combination of a supported component class (see Chapter 4 "Standard User Interface Components") and a `Renderer` from the Standard HTML RenderKit (see Section 8.5 "Standard HTML RenderKit Implementation") that supports that component type. The tag library descriptor for this tag library must meet the following requirements:

- Must declare a tag library version (`<tlib-version>`) value of `1.0`.
- Must declare a JSP version dependency (`<jsp-version>`) value of `1.2`.
- Must declare a URI (`<uri>`) value of `http://java.sun.com/jsf/html`.
- Must be included in the `META-INF` directory of a JAR file containing the corresponding implementation classes, suitable for inclusion with a web application, such that the tag library descriptor will be located automatically by the algorithm described in Section 7.3 of the *JavaServer Pages Specification* (version 1.2).

The custom actions defined in this tag library must specify the following return values for the `getComponentType()` and `getRendererType()` methods, respectively:.

TABLE 9-2 Standard HTML RenderKit Tag Library

getComponentType()	getRendererType()	custom action name
javax.faces.Column	(null)*	column
javax.faces.HtmlCommandButton	javax.faces.Button	commandButton
javax.faces.HtmlCommandLink	javax.faces.Link	commandLink
javax.faces.HtmlDataTable	javax.faces.Table	dataTable
javax.faces.HtmlForm	javax.faces.Form	form
javax.faces.HtmlGraphicImage	javax.faces.Image	graphicImage
javax.faces.HtmlInputHidden	javax.faces.Hidden	inputHidden
javax.faces.HtmlInputSecret	javax.faces.Secret	inputSecret
javax.faces.HtmlInputText	javax.faces.Text	inputText

TABLE 9-2 Standard HTML RenderKit Tag Library

getComponentType()	getRendererType()	custom action name
javax.faces.HtmlInputTextarea	javax.faces.Textarea	inputTextarea
javax.faces.HtmlMessage	javax.faces.Message	message
javax.faces.HtmlMessages	javax.faces.Messages	messages
javax.faces.HtmlOutputFormat	javax.faces.Format	outputFormat
javax.faces.HtmlOutputLabel	javax.faces.Label	outputLabel
javax.faces.HtmlOutputLink	javax.faces.Link	outputLink
javax.faces.HtmlOutputText	javax.faces.Text	outputText
javax.faces.HtmlPanelGrid	javax.faces.Grid	panelGrid
javax.faces.HtmlPanelGroup	javax.faces.Group	panelGroup
javax.faces.HtmlSelectBooleanCheckbox	javax.faces.Checkbox	selectBooleanCheckbox
javax.faces.HtmlSelectManyCheckbox	javax.faces.Checkbox	selectManyCheckbox
javax.faces.HtmlSelectManyListbox	javax.faces.Listbox	selectManyListbox
javax.faces.HtmlSelectManyMenu	javax.faces.Menu	selectManyMenu
javax.faces.HtmlSelectOneListbox	javax.faces.Listbox	selectOneListbox
javax.faces.HtmlSelectOneMenu	javax.faces.Menu	selectOneMenu
javax.faces.HtmlSelectOneRadio	javax.faces.Radio	selectOneRadio

* This component has no associated Renderer, so the getRendererType() method must return null instead of a renderer type.

The tag library descriptor for this tag library (and the corresponding tag handler implementation classes) must meet the following requirements:

- The tag library descriptor must provide attribute declarations, and a the tag handler implementation class must provide a public setter method taking a `String` parameter, for the render-independent properties of the corresponding components, and render-dependent properties of the corresponding renderers.
- The tag library descriptor entry for each attribute must specify an `<rtexprvalue>` of `false`.
- For a non-`null` `action` attribute on custom actions related to `UICommand` components (`commandButton`, `commandLink`), the `setProperties()` method of the tag handler implementation class must:
  - If the specified `String` value is not a value binding expression, create a `MethodBinding` instance that will return this value when its `invoke()` method is called, and store it as the value of the `action` attribute on the underlying component.
  - Otherwise, create a `MethodBinding` instance for the specified expression, and store that instance as the value of the action attribute on the underlying component.
- For other non-`null` attributes that correspond to `MethodBinding` attributes on the underlying components (`actionListener`, `validator`, `valueChangeListener`), the `setProperties()` method of the tag handler implementation class must:
  - Throw an exception if the specified `String` value is not a value binding expression.
  - Create a `MethodBinding` instance for the specified expression, and store that instance as the value of the corresponding component property.
- For any non-`null` `id`, `scope`, or `var` attribute, the `setProperties()` method of the tag handler implementation class must simply set the value of the corresponding component attribute.
- For all other non-`null` attributes, the `setProperties()` of the tag handler implementation class method must:
  - If the specified `String` value is not a value binding expression, set the corresponding attribute on the underlying component (after performing any necessary type conversion).
  - If the specified `String` value is a value binding expression, create a `ValueBinding` instance for that expression, and call the `setValueBinding()` method on the underlying component, passing the attribute name and the `ValueBinding` instance as parameters.

CHAPTER 10

Using JSF in Web Applications

This specification provides JSF implementors significant freedom to differentiate themselves through innovative implementation techniques, as well as value-added features. However, to ensure that web applications based on JSF can be executed unchanged across different JSF implementations, the following additional requirements, defining how a JSF-based web application is assembled and configured, must be supported by all JSF implementations.

10.1 Web Application Deployment Descriptor

JSF-based applications are *web applications* that conform to the requirements of the *Java Servlet Specification* (version 2.3 or later), and also use the facilities defined in this specification. Conforming web applications are packaged in a *web application archive* (WAR), with a well-defined internal directory structure. A key element of a WAR is the *web application deployment descriptor*, an XML document that describes the configuration of the resources in this web application. This document is included in the WAR file itself, at resource path `/WEB-INF/web.xml`.

Portable JSF-based web applications must include the following configuration elements, in the appropriate portions of the web application deployment descriptor. Element values that are rendered in *italics* represent values that the application developer is free to choose. Element values rendered in **bold** represent values that must be utilized exactly as shown.

Executing the request processing lifecycle via other mechanisms is also allowed (for example, an MVC-based application framework can incorporate calling the correct phase implementations in the correct order); however, all JSF implementations must support the functionality described in this chapter to ensure application portability.

10.1.1 Servlet Definition

JSF implementations must provide request processing lifecycle services through a standard servlet, defined by this specification. This servlet must be defined, in the deployment descriptor of an application that wishes to employ this portable mechanism, as follows:

```
<servlet>
   <servlet-name> faces-servlet-name </servlet-name>
   <servlet-class>
      javax.faces.webapp.FacesServlet
   </servlet-class>
</servlet>
```

The servlet name, denoted as `faces-servlet-name` above, may be any desired value; however, the same value must be used in the servlet mapping (see Section 10.1.2 “Servlet Mapping”).

In addition to `FacesServlet`, JSF implementations may support other ways to invoke the JavaServer Faces request processing lifecycle, but applications that rely on these mechanisms will not be portable.

10.1.2 Servlet Mapping

All requests to a web application are mapped to a particular servlet based on matching a URL pattern (as defined in the *Java Servlet Specification*) against the portion of the request URL after the context path that selected this web application. JSF implementations must support web application that define a `<servlet-mapping>` that maps any valid `url-pattern` to the `FacesServlet`. Prefix or extension mapping may be used. When using prefix mapping, the following mapping is recommended, but not required:

```
<servlet-mapping>
   <servlet-name> faces-servlet-name </servlet-name>
   <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

When using extension mapping the following mapping is recommended, but not required:

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

In addition to `FacesServlet`, JSF implementations may support other ways to invoke the JavaServer Faces request processing lifecycle, but applications that rely on these mechanisms will not be portable.

10.1.3 Application Configuration Parameters

Servlet containers support application configuration parameters that may be customized by including `<context-param>` elements in the web application deployment descriptor. All JSF implementations are required to support the following application configuration parameter names:

- `javax.faces.CONFIG_FILES` -- Comma-delimited list of context-relative resource paths under which the JSF implementation will look for application configuration resources (see Section 10.3.3 "Application Configuration Resource Format"), before loading a configuration resource named "`/WEB-INF/faces-config.xml`" (if such a resource exists).
- `javax.faces.DEFAULT_SUFFIX` -- The default suffix for extension-mapped resources that contain JSF components. If not specified, the default value ".jsp" must be used.
- `javax.faces.LIFECYCLE_ID` -- Lifecycle identifier of the `Lifecycle` instance to be used when processing JSF requests for this web application. If not specified, the JSF default instance, identified by `LifecycleFactory.DEFAULT_LIFECYCLE`, must be used.
- `javax.faces.STATE_SAVING_METHOD` -- The location where state information is saved. Valid values are "server" (typically saved in `HttpSession`) and "client (typically saved as a hidden field in the subsequent form submit). If not specified, the default value "server" must be used.

JSF implementations may choose to support additional configuration parameters, as well as additional mechanisms to customize the JSF implementation; however, applications that rely on these facilities will not be portable to other JSF implementations.

10.2 Included Classes and Resources

A JSF-based application will rely on a combination of APIs, and corresponding implementation classes and resources, in addition to its own classes and resources. The web application archive structure identifies two standard locations for classes and resources that will be automatically made available when a web application is deployed:

- `/WEB-INF/classes` -- A directory containing unpacked class and resource files.
- `/WEB-INF/lib` -- A directory containing JAR files that themselves contain class files and resources.

In addition, servlet and portlet containers typically provide mechanisms to share classes and resources across one or more web applications, without requiring them to be included inside the web application itself.

The following sections describe how various subsets of the required classes and resources should be packaged, and how they should be made available.

10.2.1 Application-Specific Classes and Resources

Application-specific classes and resources should be included in `/WEB-INF/classes` or `/WEB-INF/lib`, so that they are automatically made available upon application deployment.

10.2.2 Servlet and JSP API Classes (javax.servlet.*)

These classes will typically be made available to all web applications using the shared class facilities of the servlet container. Therefore, these classes should not be included inside the web application archive.

10.2.3 JSP Standard Tag Library (JSTL) API Classes (javax.servlet.jsp.jstl.*)

These classes describe the APIs for the JSP Standard Tag Library. They are generally packaged in a JAR file named `jstl.jar` (although this name is not required). The JSTL API classes should be installed using the shared class facility of your servlet container; however, they may also be included inside the web application archive (in the `/WEB-INF/lib` directory).

At some future time, JSP Standard Tag Library might become part of the Java2 Enterprise Edition (J2EE) platform, at which time the container will be required to provide these classes through a shared class facility.

10.2.4 JSP Standard Tag Library (JSTL) Implementation Classes

These classes and resources comprise the implementation of the JSTL APIs that is provided by a JSTL implementor. Typically, such classes will be provided in the form of one or more JAR files, which can be either installed with the container's shared class facility, or included inside the web application archive (in the `/WEB-INF/lib` directory).

10.2.5 JavaServer Faces API Classes (javax.faces.*)

These classes describe the fundamental APIs provided by all JSF implementations. They are generally packaged in a JAR file named `jsf-api.jar` (although this name is not required). The JSF API classes should be installed using the shared classes facility of your servlet container; however, they may also be included inside the web application archive (in the `/WEB-INF/lib` directory).

At some future time, JavaServer Faces might become part of the Java2 Enterprise Edition (J2EE) platform, at which time the container will be required to provide these classes through a shared class facility.

10.2.6 JavaServer Faces Implementation Classes

These classes and resources comprise the implementation of the JSF APIs that is provided by a *JSF implementor*. Typically, such classes will be provided in the form of one or more JAR files, which can be either installed with the container's shared class facility, or included in the `/WEB-INF/lib` directory of a web application archive.

10.2.6.1 FactoryFinder

`javax.faces.FactoryFinder` implements the standard discovery algorithm for all factory objects specified in the JavaServer Faces APIs. For a given factory class name, a corresponding implementation class is searched for based on the following algorithm. Items are listed in order of decreasing search precedence:

1. If a default JavaServer Faces configuration file (/WEB-INF/faces-config.xml) is bundled into the `web application, and it` contains a factory entry of the given factory class name, that factory class is used.

2. If the JavaServer Faces configuration resource(s) named by the `javax.faces.CONFIG_FILES` `ServletContext` init parameter (if any) contain any factory entries of the given factory class name, those factories are used, with the last one taking precedence.

3. If there are any META-INF/faces-config.xml resources bundled any JAR files in the `web ServletContext's resource paths`, the factory entries of the given factory class name in those files are used, with the last one taking precedence.

4. If a `META-INF/services/{factory-class-name}` resource is visible to the web application class loader for the calling application (typically as a result of being present in the manifest of a JAR file), its first line is read and assumed to be the name of the factory implementation class to use.

5. If none of the above steps yield a match, the JavaServer Faces implementation specific class is used.

If any of the factories found on any of the steps above happen to have a one-argument constructor, with argument the type being the abstract factory class, that constructor is invoked, and the previous match is passed to the constructor. For example, say the container vendor provided an implementation of `FacesContextFactory`, and identified it in `META-INF/services/javax.faces.context.FacesContextFactory` in a jar on the webapp ClassLoader. Also say this implementation provided by the container vendor had a one argument constructor that took a `FacesContextFactory` instance. The `FactoryFinder` system would call that one-argument constructor, passing the implementation of `FacesContextFactory` provided by the JavaServer Faces implementation.

If a Factory implementation does not provide a proper one-argument constructor, it must provide a zero-arguments constructor in order to be successfully instantiated.

Once the name of the factory implementation class is located, the web application class loader for the calling application is requested to load this class, and a corresponding instance of the class will be created. A side effect of this rule is that each web application will receive its own instance of each factory class, whether the JavaServer Faces implementation is included within the web application or is made visible through the container's facilities for shared libraries.

```
public static Object getFactory(String factoryName);
```

Create (if necessary) and return a per-web-application instance of the appropriate implementation class for the specified JavaServer Faces factory class, based on the discovery algorithm described above.

JSF implementations must also include implementations of the several factory classes. In order to be dynamically instantiated according to the algorithm defined above, the factory implementation class must include a public, no-arguments constructor. Factory class implementations must be provided for the following factory names:

- `javax.faces.application.ApplicationFactory` (`FactoryFinder.APPLICATION_FACTORY`)—Factory for `Application` instances.
- `javax.faces.context.FacesContextFactory` (`FactoryFinder.FACES_CONTEXT_FACTORY`)—Factory for `FacesContext` instances.
- `javax.faces.lifecycle.LifecycleFactory` (`FactoryFinder.LIFECYCLE_FACTORY`)—Factory for `Lifecycle` instances.
- `javax.faces.render.RenderKitFactory` (`FactoryFinder.RENDER_KIT_FACTORY`)—Factory for `RenderKit` instances.

10.2.6.2 FacesServlet

`FacesServlet` is an implementation of `javax.servlet.Servlet` that accepts incoming requests and passes them to the appropriate `Lifecycle` implementation for processing. This servlet must be declared in the web application deployment descriptor, as described in Section 10.1.1 “Servlet Definition”, and mapped to a standard URL pattern as described in Section 10.1.2 “Servlet Mapping”.

```
public void init(ServletConfig config) throws ServletException;
```

Acquire and store references to the `FacesContextFactory` and `Lifecycle` instances to be used in this web application.

```
public void destroy();
```

Release the `FacesContextFactory` and `Lifecycle` references that were acquired during execution of the `init()` method.

```
public void service(ServletRequest request, ServletResponse
response) throws IOException, ServletException;
```

For each incoming request, the following processing is performed:

- Using the `FacesContextFactory` instance stored during the `init()` method, call the `getFacesContext()` method to acquire a `FacesContext` instance with which to process the current request.

- Call the execute() method of the saved Lifecycle instance, passing the FacesContext instance for this request as a parameter. If the execute() method throws a FacesException, re-throw it as a ServletException with the FacesException as the root cause.
- Call the render() method of the saved Lifecycle instance, passing the FacesContext instance for this request as a parameter. If the render() method throws a FacesException, re-throw it as a ServletException with the FacesException as the root cause.
- Call the release() method on the FacesContext instance, allowing it to be returned to a pool if the JSF implementation uses one.

The FacesServlet implementation class must also declare two static public final String constants whose value is a context initialization parameter that affects the behavior of the servlet:

- CONFIG_FILES_ATTR -- the context initialization attribute that may optionally contain a comma-delimited list of context relative resources (in addition to /WEB-INF/faces-config.xml which is always processed if it is present) to be processed. The value of this constant must be "javax.faces.CONFIG_FILES".
- LIFECYCLE_ID_ATTR -- the lifecycle identifier of the Lifecycle instance to be used for processing requests to this application, if an instance other than the default is required. The value of this constant must be "javax.faces.LIFECYCLE_ID".

10.2.6.3 UIComponentTag

UIComponentTag is an implementation of javax.servlet.jsp.tagext.Tag, and must be the base class for any JSP custom action that corresponds to a JSF UIComponent. See Chapter 9 "Integration with JSP, and the Javadocs for UIComponentTag, for more information about using this class as the base class for your own UIComponent custom action classes.

10.2.6.4 UIComponentBodyTag

UIComponentBodyTag is a subclass of UIComponentTag, so it inherits all of the functionality described in the preceding section. In addition, this class implements the standard functionality provided by javax.servlet.jsp.BodyTagSupport, so it is useful as the base class for JSF custom action implementations that must process their body content. See Chapter 9 "Integration with JSP, and the Javadocs for UIComponentBodyTag, for more information about using this class as the base class for your own UIComponent custom action classes

10.2.6.5 AttributeTag

JSP custom action that adds a named attribute (if necessary) to the `UIComponent` associated with the closest parent `UIComponent` custom action. See Section 9.4.2 “<f:attribute>”.

10.2.6.6 ConverterTag

JSP custom action (and convenience base class) that creates and registers a `Converter` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action. See Section 9.4.3 “<f:convertDateTime>”, Section 9.4.4 “<f:convertNumber>”, and Section 9.4.5 “<f:converter>”.

10.2.6.7 FacetTag

JSP custom action that adds a named facet (see Section 3.1.9 “Facet Management”) to the UIComponent associated with the closest parent UIComponent custom action. See Section 9.4.6 “<f:facet>”.

10.2.6.8 ValidatorTag

JSP custom action (and convenience base class) that creates and registers a `Validator` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action. See Section 9.4.12 “<f:validateDoubleRange>”, Section 9.4.13 “<f:validateLength>”, Section 9.4.14 “<f:validateLongRange>”, and Section 9.4.15 “<f:validator>”.

10.3 Application Configuration Resources

This section describes the JSF support for portable application configuration resources used to configure application components.

10.3.1 Overview

JSF defines a portable configuration resource format (as an XML document) for standard configuration information. One or more such application resources will be loaded automatically, at application startup time, by the JSF implementation. The information parsed from such resources will augment the information provided by the JSF implementation, as described below.

In addition to their use during the execution of a JSF-based web application, configuration resources provide information that is useful to development tools created by Tool Providers. The mechanism by which configuration resources are made available to such tools is outside the scope of this specification.

10.3.2 Application Startup Behavior

At application startup time, before any requests are processed, the JSF implementation must process zero or more application configuration resources, located according to the following algorithm:

- Search for all resources named "`META-INF/faces-config.xml`" in the `ServletContext` resource paths for this web application, and load each as a JSF configuration resource (in reverse order of the order in which they are returned by `getResources()`).
- Check for the existence of a context initialization parameter named `javax.faces.CONFIG_FILES`. If it exists, treat it as a comma-delimited list of context relative resource paths (starting with a "`/`"), and load each of the specfied resources.
- Check for the existence of a web application configuration resource named "`/WEB-INF/faces-config.xml`", and load it if the resource exists.

This algorithm provides considerable flexibility for developers that are assembling the components of a JSF-based web application. For example, an application might include one or more custom `UIComponent` implementations, along with associated `Renderer`s, so it can declare them in an application resource named "`/WEB-INF/faces-config.xml`" with no need to programmatically register them with `Application` instance. In addition, the application might choose to include a component library (packaged as a JAR file) that includes a "`META-INF/faces-config.xml`" resource. The existence of this resource causes components, renderers, and other JSF implementation classes that are stored in this library JAR file to be automatically registered, with no action required by the application.

XML parsing errors detected during the loading of an application resource file are fatal to application startup, and must cause the application to not be made available by the container. Whether or not a validating parse is performed is up to the JSF implementation; it is recommended that the JSF implementation provide a configuration parameter to control whether or not validation occurs.

10.3.3 Application Configuration Resource Format

Application configuration resources must conform to the XML document description shown below. In addition, they must include the one of the following DOCTYPE declarations:

```
<!DOCTYPE faces-config PUBLIC
  “-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN”
  “http://java.sun.com/dtd/web-facesconfig_1_0.dtd”>
```

```
<!DOCTYPE faces-config PUBLIC
  “-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN”
  “http://java.sun.com/dtd/web-facesconfig_1_1.dtd”>
```

The only difference between the 1.0 and 1.1 DTDs is the presence of facet support in 1.1. 1.1 is backwards compatible with 1.0. The actual Document Type Description that corresponds to the 1.1 DOCTYPE declaration is as follows. Please see the binary distribution for the 1.0 DTD:

```
<!--

 Copyright 2004 Sun Microsystems, Inc. All rights reserved.

 SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.

-->



<!--

    DTD for the JavaServer Faces Application Configuration File
(Version 1.1)
```

```
     To support validation of your configuration file(s),
include the following

     DOCTYPE element at the beginning (after the “xml”
declaration):


     <!DOCTYPE faces-config PUBLIC

      “-//Sun Microsystems, Inc.//DTD JavaServer Faces Config
1.1//EN”

      “http://java.sun.com/dtd/web-facesconfig_1_1.dtd”>


     $Id: web-facesconfig_1_1.dtd,v 1.2 2004/04/09 18:11:35
eburns Exp $


-->




<!-- ==================== Defined Types ======================
============ -->




<!--

     An “Action” is a String that represents a method binding

     expression that points at a method with no arguments that
returns a

     String.  It must be bracketed with “#{}”, for example,

     “#{cardemo.buyCar}”.

-->

<!ENTITY % Action “CDATA”>
```

```
<!--

     A "ClassName" is the fully qualified name of a Java class
that is

     instantiated to provide the functionality of the enclosing
element.

-->

<!ENTITY % ClassName "CDATA">




<!--

     An "Identifier" is a string of characters that conforms to
the variable

     naming conventions of the Java programming language (JLS
Section ?.?.?).

-->

<!ENTITY % Identifier "CDATA">




<!--

     A "JavaType" is either the fully qualified name of a Java
class that is

     instantiated to provide the functionality of the enclosing
element, or

     the name of a Java primitive type (such as int or char).
The class name

     or primitive type may optionally be followed by "[]" to
indicate that

     the underlying data must be an array, rather than a scalar
variable.

-->

<!ENTITY % JavaType "CDATA">
```

```
<!--
    A “Language” is a lower case two-letter code for a language
as defined
     by ISL-639.
-->
<!ENTITY % Language “CDATA”>



<!--
     A “ResourcePath” is the relative or absolute path to a
resource file
     (such as a logo image).
-->
<!ENTITY % ResourcePath “CDATA”>



<!--
     A “Scope” is the well-known name of a scope in which
managed beans may
     optionally be defined to be created in.
-->
<!ENTITY % Scope “(request|session|application)”>



<!--
     A “ScopeOrNone” element defines the legal values for the
```

```
     <managed-bean-scope> element's body content, which includes

     all of the scopes respresented by the "Scope" type, plus
the

     "none" value indicating that a created bean should not be

     stored into any scope.

-->

<!ENTITY % ScopeOrNone "(request|session|application|none)">




<!--

     A "ViewIdPattern" is a pattern for matching view
identifiers in

     order to determine whether a particular navigation rule
should be

     fired.  It must contain one of the following values:


     - The exact match for a view identifier that is recognized

       by the the ViewHandler implementation being used (such as

        "/index.jsp" if you are using the default ViewHandler).


     - A proper prefix of a view identifier, plus a trailing "*"

        character.  This pattern indicates that all view
identifiers that

       match the portion of the pattern up to the asterisk will
match the

       surrounding rule.  When more than one match exists, the
match with

        the longest pattern is selected.


     - An "*" character, which means that this pattern applies
to all
```

```
       view identifiers.

-->

<!ENTITY % ViewIdPattern “CDATA”>




<!-- ==================== Top Level Elements ==================
============ -->




<!--

    The “faces-config” element is the root of the configuration
information

     hierarchy, and contains nested elements for all of the
other configuration

     settings.

-->

<!ELEMENT faces-config
((application|factory|component|converter|managed-
bean|navigation-rule|referenced-bean|render-
kit|lifecycle|validator)*)>

<!ATTLIST faces-config

        xmlns CDATA #FIXED
“http://java.sun.com/JSF/Configuration”>




<!-- ==================== Definition Elements =================
============ -->




<!--
```

```
     The “application” element provides a mechanism to define
the various

     per-application-singleton implementation artifacts for a
particular web

    application that is utilizing JavaServer Faces.  For nested
elements

    that are not specified, the JSF implementation must provide
a suitable

     default.

-->

<!ELEMENT application      ((action-listener|default-render-kit-
id|message-bundle|navigation-handler|view-handler|state-
manager|property-resolver|variable-resolver|locale-config)*)>




<!--

     The “factory” element provides a mechanism to define the
various

     Factories that comprise parts of the implementation of
JavaServer

    Faces.  For nested elements that are not specified, the JSF

     implementation must provide a suitable default.

-->

<!ELEMENT factory      ((application-factory|faces-context-
factory|lifecycle-factory|render-kit-factory)*)>




<!--

     The “attribute” element represents a named, typed, value
associated with

     the parent UIComponent via the generic attributes
mechanism.
```

```
     Attribute names must be unique within the scope of the
parent (or related)

     component.

-->

<!ELEMENT attribute       (description*, display-name*, icon*,
attribute-name, attribute-class, default-value?, suggested-
value?, attribute-extension*)>


<!--

     Extension element for attribute.  May contain
implementation

     specific content.

-->

<!ELEMENT attribute-extension ANY>




<!--

     The “component” element represents a concrete UIComponent
implementation

     class that should be registered under the specified type
identifier,

     along with its associated properties and attributes.
Component types must

     be unique within the entire web application.


     Nested “attribute” elements identify generic attributes
that are recognized

     by the implementation logic of this component.  Nested
“property” elements

     identify JavaBeans properties of the component class that
may be exposed
```

```
    for manipulation via tools.

-->

<!ELEMENT component       (description*, display-name*, icon*,
component-type, component-class, facet*, attribute*, property*,
component-extension*)>



<!--

    Extension element for component.  May contain
implementation

    specific content.

-->

<!ELEMENT component-extension ANY>



<!--

   Define the name and other design-time information for a
facet that is

   associated with a renderer or a component.

-->



<!ELEMENT facet        (description*, display-name*, icon*,
facet-name, facet-extension*)>



<!--

    Extension element for facet.  May contain implementation

    specific content.

-->

<!ELEMENT facet-extension ANY>



<!--
```

```
     The “facet-name” element represents the facet name under
which a

     UIComponent will be added to its parent.  It must be of
type

     “Identifier”.

-->

<!ELEMENT facet-name (#PCDATA)>



<!--

     The “converter” element represents a concrete Converter
implementation

     class that should be registered under the specified
converter identifier.

    Converter identifiers must be unique within the entire web
application.



     Nested “attribute” elements identify generic attributes
that may be

     configured on the corresponding UIComponent in order to
affect the

     operation of the Converter.  Nested “property” elements
identify JavaBeans

     properties of the Converter implementation class that may
be configured

     to affect the operation of the Converter.

-->

<!ELEMENT converter       (description*, display-name*, icon*,
(converter-id | converter-for-class), converter-class,
attribute*, property*)>




<!--
```

```
     The “icon” element contains “small-icon” and “large-icon”
elements that

    specify the resoruce paths for small and large GIF or JPG
icon images

     used to represent the parent element in a GUI tool.

-->

<!ELEMENT icon              (small-icon?, large-icon?)>

<!ATTLIST icon              xml:lang         %Language;
#IMPLIED>


<!--

     The “lifecycle” element provides a mechanism to specify

     modifications to the behaviour of the default Lifecycle

     implementation for this web application.

-->

<!ELEMENT lifecycle        (phase-listener*)>


<!--

      The “locale-config” element allows the app developer to
declare the

      supported locales for this application.

-->

<!ELEMENT locale-config (default-locale?, supported-locale*)>


<!--

     The “managed-bean” element represents a JavaBean, of a
particular class,

     that will be dynamically instantiated at runtime (by the
default

    VariableResolver implementation) if it is referenced as the
first element
```

```
     of a value binding expression, and no corresponding bean
can be

     identified in any scope.  In addition to the creation of
the managed bean,

     and the optional storing of it into the specified scope,
the nested

     managed-property elements can be used to initialize the
contents of

     settable JavaBeans properties of the created instance.

-->

<!ELEMENT managed-bean (description*, display-name*, icon*,
managed-bean-name, managed-bean-class, managed-bean-scope,
(managed-property* | map-entries | list-entries))>




<!--

     The “managed-property” element represents an individual
property of a

    managed bean that will be configured to the specified value
(or value set)

    if the corresponding managed bean is automatically created.

-->

<!ELEMENT managed-property (description*, display-name*, icon*,
property-name, property-class?, (map-entries|null-
value|value|list-entries))>




<!--

     The “map-entry” element reprsents a single key-entry pair
that

     will be added to the computed value of a managed property
of type

     java.util.Map.
```

```
-->

<!ELEMENT map-entry (key, (null-value|value))>




<!--

     The “map-entries’ element represents a set of key-entry
pairs that

     will be added to the computed value of a managed property
of type

     java.util.Map.  In addition, the Java class types of the
key and entry

     values may be optionally declared.

-->

<!ELEMENT map-entries (key-class?, value-class?, map-entry*)>




<!--

    The base name of a resource bundle representing the message
resources

     for this application.  See the JavaDocs for the
“java.util.ResourceBundle”

    class for more information on the syntax of resource bundle
names.

-->


<!ELEMENT message-bundle (#PCDATA)>




<!--

     The “navigation-case” element describes a particular
combination of
```

```
     conditions that must match for this case to be executed,
and the

     view id of the component tree that should be selected next.

-->

<!ELEMENT navigation-case (description*, display-name*, icon*,
from-action?, from-outcome?, to-view-id, redirect?)>




<!--

     The "navigation-rule" element represents an individual
decision rule

     that will be utilized by the default NavigationHandler

     implementation to make decisions on what view should be
displayed

     next, based on the view id being processed.

-->

<!ELEMENT navigation-rule (description*, display-name*, icon*,
from-view-id?, navigation-case*)>




<!--

     The "property" element represents a JavaBean property of
the Java class

     represented by our parent element.



     Property names must be unique within the scope of the Java
class

     that is represented by the parent element, and must
correspond to

     property names that will be recognized when performing
introspection
```

```
     against that class via java.beans.Introspector.

-->

<!ELEMENT property         (description*, display-name*, icon*,
property-name, property-class, default-value?, suggested-value?,
property-extension*)>


<!--

     Extension element for property.  May contain implementation

     specific content.

-->

<!ELEMENT property-extension ANY>


<!--

    The “referenced-bean” element represents at design time the
promise

     that a Java object of the specified type will exist at
runtime in some

    scope, under the specified key.  This can be used by design
time tools

    to construct user interface dialogs based on the properties
of the

     specified class.  The presence or absence of a referenced
bean

     element has no impact on the JavaServer Faces runtime
environment

     inside a web application.

-->

<!ELEMENT referenced-bean (description*, display-name*, icon*,
referenced-bean-name, referenced-bean-class)>
```

```
<!--

     The “render-kit” element represents a concrete RenderKit
implementation

     that should be registered under the specified render-kit-
id.  If no

     render-kit-id is specified, the identifier of the default
RenderKit

     (RenderKitFactory.DEFAULT_RENDER_KIT) is assumed.

-->

<!ELEMENT render-kit       (description*, display-name*, icon*,
render-kit-id?, render-kit-class?, renderer*)>



<!--

     The “renderer” element represents a concrete Renderer
implementation

     class that should be registered under the specified
component family

    and renderer type identifiers, in the RenderKit associated
with the

     parent “render-kit” element.  Combinations of component
family and renderer

     type must be unique within the RenderKit associated with
the parent

     “render-kit” element.


     Nested “attribute” elements identify generic component
attributes that

     are recognized by this renderer.

-->

<!ELEMENT renderer        (description*, display-name*, icon*,
component-family, renderer-type, renderer-class, facet*,
attribute*, renderer-extension*)>
```

```
<!--
     Extension element for renderer.  May contain implementation
     specific content.
-->
<!ELEMENT renderer-extension ANY>


<!--
     The "validator" element represents a concrete Validator
implementation
     class that should be registered under the specified
validator identifier.
    Validator identifiers must be unique within the entire web
application.


     Nested "attribute" elements identify generic attributes
that may be
     configured on the corresponding UIComponent in order to
affect the
     operation of the Validator.  Nested "property" elements
identify JavaBeans
     properties of the Validator implementation class that may
be configured
     to affect the operation of the Validator.
-->
<!ELEMENT validator        (description*, display-name*, icon*,
validator-id, validator-class, attribute*, property*)>



<!--
```

```
     The “list-entries” element represents a set of
initialization

    elements for a managed property that is a java.util.List or
an

     array.  In the former case, the “value-class” element can
optionally

    be used to declare the Java type to which each value should
be

     converted before adding it to the Collection.

-->

<!ELEMENT list-entries    (value-class?, (null-value|value)*)>



<!-- ==================== Subordinate Elements ================
============ -->



<!--

     The “action-listener” element contains the fully qualified
class name

     of the concrete ActionListener implementation class that
will be called

     during the Invoke Application phase of the request
processing lifecycle.

     It must be of type “ClassName”.

-->

<!ELEMENT action-listener (#PCDATA)>


<!--

     The “application-factory” element contains the fully
qualified class
```

```
     name of the concrete ApplicationFactory implementation
class that

     will be called when
FactoryFinder.getFactory(APPLICATION_FACTORY) is

     called. It must be of type “ClassName”.

-->

<!ELEMENT application-factory (#PCDATA)>



<!--

     The “attribute-class” element represents the Java type of
the value

     associated with this attribute name.  It must be of type
“ClassName”.

-->

<!ELEMENT attribute-class (#PCDATA)>




<!--

     The “attribute-name” element represents the name under
which the

     corresponding value will be stored, in the generic
attributes of the

     UIComponent we are related to.

-->

<!ELEMENT attribute-name  (#PCDATA)>




<!--

     The “component-class” element represents the fully
qualified class name
```

```
    of a concrete UIComponent implementation class.  It must be
of
     type "ClassName".
-->
<!ELEMENT component-class (#PCDATA)>



<!--
     The "component-family" element represents the component
family for
     which the Renderer represented by the parent "renderer"
element will be
     used.
-->
<!ELEMENT component-family (#PCDATA)>


<!--
     The "component-type" element represents the name under
which the
     corresponding UIComponent class should be registered.
-->
<!ELEMENT component-type  (#PCDATA)>



<!--
     The "converter-class" element represents the fully
qualified class name
     of a concrete Converter implementation class.  It must be
of
     type "ClassName".
```

```
-->
<!ELEMENT converter-class (#PCDATA)>



<!--
     The "converter-for-class" element represents the fully
qualified class name
     for which a Converter class will be registered.  It must
be of
     type "ClassName".
-->
<!ELEMENT converter-for-class (#PCDATA)>



<!--
     The "converter-id" element represents the identifier under
which the
     corresponding Converter class should be registered.
-->
<!ELEMENT converter-id    (#PCDATA)>


<!--
     The "default-render-kit-id" element allows the application
to define
     a renderkit to be used other than the standard one.
-->
<!ELEMENT default-render-kit-id   (#PCDATA)>


<!--
```

```
      The “default-locale” element declares the default locale
for this

      application instance.  It must be specified as

     :language:[_:country:[_:variant:]] without the colons, for
example

     “ja_JP_SJIS”.  The separators between the segments may be
‘-’ or

      ‘_’.

-->

<!ELEMENT default-locale (#PCDATA)>


<!--

    The “default-value” contains the value for the property or
attribute

    in which this element resides.  This value differs from the

     “suggested-value” in that the property or attribute must
take the

     value, whereas in “suggested-value” taking the value is
optional.

-->

<!ELEMENT default-value (#PCDATA)>


<!--

    The “description” element contains a textual description of
the element

     it is nested in, optionally flagged with a language code
using the

     “xml:lang” attribute.

-->

<!ELEMENT description      ANY>
```

```
<!ATTLIST description     xml:lang        %Language;
#IMPLIED>




<!--

     The “display-name” element is a short descriptive name
describing the

     entity associated with the element it is nested in,
intended to be

    displayed by tools, and optionally flagged with a language
code using

     the “xml:lang” attribute.

-->

<!ELEMENT display-name    (#PCDATA)>

<!ATTLIST display-name    xml:lang        %Language;
#IMPLIED>


<!--

     The “faces-context-factory” element contains the fully
qualified

     class name of the concrete FacesContextFactory
implementation class

     that will be called when

     FactoryFinder.getFactory(FACES_CONTEXT_FACTORY) is called.
It must

     be of type “ClassName”.

-->

<!ELEMENT faces-context-factory (#PCDATA)>


<!--
```

```
     The “from-action” element contains an action reference
expression

    that must have been executed (by the default ActionListener
for handling

     application level events) in order to select this
navigation rule.  If

     not specified, this rule will be relevant no matter which
action reference

     was executed (or if no action reference was executed).


     This value must be of type “Action”.

-->

<!ELEMENT from-action (#PCDATA)>




<!--

     The “from-outcome” element contains a logical outcome
string returned

     by the execution of an application action method selected
via an

     “actionRef” property (or a literal value specified by an
“action”

     property) of a UICommand component.  If specified, this
rule will be

     relevant only if the outcome value matches this element’s
value.  If

     not specified, this rule will be relevant no matter what
the outcome

     value was.

-->

<!ELEMENT from-outcome    (#PCDATA)>
```

```
<!--
     The "from-view-id" element contains the view identifier of
the view
     for which the containing navigation rule is relevant.  If
no
     "from-view" element is specified, this rule applies to
navigation
    decisions on all views.  If this element is not specified,
a value
     of "*" is assumed, meaning that this navigation rule
applies to all
     views.


     This value must be of type "ViewIdPattern".
-->
<!ELEMENT from-view-id    (#PCDATA)>




<!--
    The "key" element is the String representation of a map key
that
    will be stored in a managed property of type java.util.Map.
-->
<!ELEMENT key                (#PCDATA)>




<!--
    The "key-class" element defines the Java type to which each
"key"
```

```
    element in a set of “map-entry” elements will be converted
to.  It

     must be of type “ClassName”.  If omitted,
“java.lang.String”

     is assumed.

-->

<!ELEMENT key-class        (#PCDATA)>




<!--

     The “large-icon” element contains the resource path to a
large (32x32)

    icon image.  The image may be in either GIF or JPG format.

-->

<!ELEMENT large-icon       (#PCDATA)>



<!--

     The “lifecycle-factory” element contains the fully
qualified class name

    of the concrete LifecycleFactory implementation class that
will be called

    when FactoryFinder.getFactory(LIFECYCLE_FACTORY) is called.
It must be

     of type “ClassName”.

-->

<!ELEMENT lifecycle-factory (#PCDATA)>



<!--

     The “managed-bean-class” element represents the fully
qualified class
```

```
     name of the Java class that will be used to instantiate a
new instance

    if creation of the specified managed bean is requested.  It
must be of

     type “ClassName”.



     The specified class must conform to standard JavaBeans
conventions.

     In particular, it must have a public zero-arguments
constructor, and

     zero or more public property setters.

-->

<!ELEMENT managed-bean-class  (#PCDATA)>




<!--

     The “managed-bean-name” element represents the attribute
name under

     which a managed bean will be searched for, as well as
stored (unless

     the “managed-bean-scope” value is “none”).  It must be of
type

     “Identifier”.

-->

<!ELEMENT managed-bean-name (#PCDATA)>




<!--

     The “managed-bean-scope” element represents the scope into
which a newly
```

```
     created instance of the specified managed bean will be
stored (unless

     the value is “none”).  It must be of type “ScopeOrNone”.

-->

<!ELEMENT managed-bean-scope (#PCDATA)>


<!--

     The “navigation-handler” element contains the fully
qualified class name

    of the concrete NavigationHandler implementation class that
will be called

     during the Invoke Application phase of the request
processing lifecycle,

     if the default ActionListener (provided by the JSF
implementation) is used.

     It must be of type “ClassName”.

-->

<!ELEMENT navigation-handler (#PCDATA)>


<!--

The “phase-listener” element contains the fully qualified class

name of the concrete PhaseListener implementation class that
will be

registered on the Lifecycle. It must be of type “ClassName”.

-->

<!ELEMENT phase-listener (#PCDATA)>


<!--

     The “redirect” element indicates that navigation to the
specified
```

```
     “to-view-id” should be accomplished by performing an HTTP
redirect

     rather than the usual ViewHandler mechanisms.

-->

<!ELEMENT redirect EMPTY>


<!--

     The “suggested-value” contains the value for the property
or

     attribute in which this element resides.  This value is
advisory

     only and is intended for tools to use when populating
pallettes.

-->

<!ELEMENT suggested-value (#PCDATA)>



<!--


     The “view-handler” element contains the fully qualified
class name

    of the concrete ViewHandler implementation class that will
be called

     during the Restore View and Render Response phases of the
request

     processing lifecycle.  The faces implementation must
provide a

     default implementation of this class

-->

<!ELEMENT view-handler (#PCDATA)>
```

```
<!--

     The “state-manager” element contains the fully qualified
class name

    of the concrete StateManager implementation class that will
be called

     during the Restore View and Render Response phases of the
request

     processing lifecycle.  The faces implementation must
provide a

     default implementation of this class

-->

<!ELEMENT state-manager (#PCDATA)>


<!--

     The “null-value” element indicates that the managed
property in which we

     are nested will be explicitly set to null if our managed
bean is

    automatically created.  This is different from omitting the
managed

     property element entirely, which will cause no property
setter to be

     called for this property.


     The “null-value” element can only be used when the
associated

     “property-class” identifies a Java class, not a Java
primitive.

-->

<!ELEMENT null-value EMPTY>
```

```
<!--
     The “property-class” element represents the Java type of
the value
     associated with this property name.  It must be of type
“JavaType”.
    If not specified, it can be inferred from existing classes;
however,
    this element should be specified if the configuration file
is going
     to be the source for generating the corresponding classes.
-->
<!ELEMENT property-class  (#PCDATA)>



<!--
     The “property-name” element represents the JavaBeans
property name
     under which the corresponding value may be stored.
-->
<!ELEMENT property-name   (#PCDATA)>



<!--
     The “property-resolver” element contains the fully
qualified class name
    of the concrete PropertyResolver implementation class that
will be used
     during the processing of value binding expressions.
     It must be of type “ClassName”.
-->
```

```
<!ELEMENT property-resolver (#PCDATA)>



<!--

     The “referenced-bean-class” element represents the fully
qualified class

     name of the Java class (either abstract or concrete) or
Java interface

    implemented by the corresponding referenced bean.  It must
be of type

     “ClassName”.

-->

<!ELEMENT referenced-bean-class  (#PCDATA)>



<!--

    The “referenced-bean-name” element represents the attribute
name under

     which the corresponding referenced bean may be assumed to
be stored,

    in one of the scopes defined by the “Scope” type.  It must
be of type

     “Identifier”.

-->

<!ELEMENT referenced-bean-name (#PCDATA)>



<!--

     The “render-kit-id” element represents an identifier for
the

     RenderKit represented by the parent “render-kit” element.
```

```
-->

<!ELEMENT render-kit-id    (#PCDATA)>




<!--

     The "render-kit-class" element represents the fully
qualified class name

     of a concrete RenderKit implementation class.  It must be
of

     type "ClassName".

-->

<!ELEMENT render-kit-class (#PCDATA)>




<!--

    The "renderer-class" element represents the fully qualified
class name

     of a concrete Renderer implementation class.  It must be of

     type "ClassName".

-->

<!ELEMENT renderer-class   (#PCDATA)>


<!--

     The "render-kit-factory" element contains the fully
qualified class name

     of the concrete RenderKitFactory implementation class that
will be called

     when FactoryFinder.getFactory(RENDER_KIT_FACTORY) is
called. It must be

     of type "ClassName".
```

```
-->

<!ELEMENT render-kit-factory (#PCDATA)>


<!--

     The “renderer-type” element represents a renderer type
identifier for the

     Renderer represented by the parent “renderer” element.

-->

<!ELEMENT renderer-type   (#PCDATA)>


<!--

     The “small-icon” element contains the resource path to a
large (16x16)

     icon image.  The image may be in either GIF or JPG format.

-->

<!ELEMENT small-icon      (#PCDATA)>


<!--

      The “supported-locale” element allows authors to declare
which

      locales are supported in this application instance.


     It must be specified as :language:[_:country:[_:variant:]]
without

      the colons, for example “ja_JP_SJIS”.  The separators
between the

      segments may be ‘-’ or ‘_’.

-->

<!ELEMENT supported-locale (#PCDATA)>
```

```
<!--
     The "to-view" element contains the view identifier of the
next view
     that should be displayed if this navigation rule is
matched.  It
     must be of type "ViewId".
-->
<!ELEMENT to-view-id       (#PCDATA)>



<!--
     The "validator-class" element represents the fully
qualified class name
     of a concrete Validator implementation class.  It must be
of
     type "ClassName".
-->
<!ELEMENT validator-class (#PCDATA)>



<!--
     The "validator-id" element represents the identifier under
which the
     corresponding Validator class should be registered.
-->
<!ELEMENT validator-id    (#PCDATA)>



<!--
```

```
     The “value” element is the String representation of a
literal

    value to which a scalar managed property will be set, or a
value

     binding expression (“#{...}”) that will be used to
calculate the

    required value.  It will be converted as specified for the
actual

     property type.

-->

<!ELEMENT value (#PCDATA)>




<!--

     The “value-class” element defines the Java type to which
each

     “value” element’s value will be converted to, prior to
adding it to

     the “list-entries” list for a managed property that is a

     java.util.List, or a “map-entries” map for a managed
property that

     is a java.util.Map.  It must be of type “ClassName”.  If
omitted,

     “java.lang.String” is assumed.

-->

<!ELEMENT value-class (#PCDATA)>


<!--

     The “variable-resolver” element contains the fully
qualified class name

    of the concrete VariableResolver implementation class that
will be used
```

```
    during the processing of value binding expressions.

    It must be of type “ClassName”.

-->

<!ELEMENT variable-resolver (#PCDATA)>



<!-- ============================ Identifier Attributes =======
============= -->



<!ATTLIST action-listener                      id ID #IMPLIED>

<!ATTLIST application                           id ID #IMPLIED>

<!ATTLIST application-factory                 id ID #IMPLIED>

<!ATTLIST attribute                             id ID #IMPLIED>

<!ATTLIST attribute-class                      id ID #IMPLIED>

<!ATTLIST attribute-extension                 id ID #IMPLIED>

<!ATTLIST attribute-name                        id ID #IMPLIED>

<!ATTLIST component                             id ID #IMPLIED>

<!ATTLIST component-class                      id ID #IMPLIED>

<!ATTLIST component-extension                 id ID #IMPLIED>

<!ATTLIST component-family                     id ID #IMPLIED>

<!ATTLIST component-type                        id ID #IMPLIED>

<!ATTLIST converter                             id ID #IMPLIED>

<!ATTLIST converter-class                      id ID #IMPLIED>

<!ATTLIST converter-for-class                 id ID #IMPLIED>

<!ATTLIST converter-id                           id ID #IMPLIED>

<!ATTLIST default-locale                        id ID #IMPLIED>

<!ATTLIST default-render-kit-id              id ID #IMPLIED>
```

```
<!ATTLIST default-value                    id ID #IMPLIED>
<!ATTLIST description                      id ID #IMPLIED>
<!ATTLIST display-name                     id ID #IMPLIED>
<!ATTLIST faces-config                     id ID #IMPLIED>
<!ATTLIST faces-context-factory            id ID #IMPLIED>
<!ATTLIST facet                            id ID #IMPLIED>
<!ATTLIST facet-extension                  id ID #IMPLIED>
<!ATTLIST facet-name                       id ID #IMPLIED>
<!ATTLIST factory                          id ID #IMPLIED>
<!ATTLIST from-action                      id ID #IMPLIED>
<!ATTLIST from-outcome                     id ID #IMPLIED>
<!ATTLIST from-view-id                     id ID #IMPLIED>
<!ATTLIST icon                             id ID #IMPLIED>
<!ATTLIST key                              id ID #IMPLIED>
<!ATTLIST key-class                        id ID #IMPLIED>
<!ATTLIST large-icon                       id ID #IMPLIED>
<!ATTLIST lifecycle                        id ID #IMPLIED>
<!ATTLIST lifecycle-factory                id ID #IMPLIED>
<!ATTLIST list-entries                     id ID #IMPLIED>
<!ATTLIST locale-config                    id ID #IMPLIED>
<!ATTLIST managed-bean                     id ID #IMPLIED>
<!ATTLIST managed-bean-class               id ID #IMPLIED>
<!ATTLIST managed-bean-name                id ID #IMPLIED>
<!ATTLIST managed-bean-scope               id ID #IMPLIED>
<!ATTLIST managed-property                 id ID #IMPLIED>
<!ATTLIST map-entries                      id ID #IMPLIED>
<!ATTLIST map-entry                        id ID #IMPLIED>
<!ATTLIST message-bundle                   id ID #IMPLIED>
```

```
<!ATTLIST navigation-case                        id ID #IMPLIED>
<!ATTLIST navigation-handler                     id ID #IMPLIED>
<!ATTLIST navigation-rule                        id ID #IMPLIED>
<!ATTLIST null-value                             id ID #IMPLIED>
<!ATTLIST phase-listener                         id ID #IMPLIED>
<!ATTLIST property                               id ID #IMPLIED>
<!ATTLIST property-class                         id ID #IMPLIED>
<!ATTLIST property-extension                     id ID #IMPLIED>
<!ATTLIST property-name                          id ID #IMPLIED>
<!ATTLIST property-resolver                      id ID #IMPLIED>
<!ATTLIST redirect                               id ID #IMPLIED>
<!ATTLIST referenced-bean                        id ID #IMPLIED>
<!ATTLIST referenced-bean-class                  id ID #IMPLIED>
<!ATTLIST referenced-bean-name                   id ID #IMPLIED>
<!ATTLIST render-kit                             id ID #IMPLIED>
<!ATTLIST render-kit-class                       id ID #IMPLIED>
<!ATTLIST render-kit-factory                     id ID #IMPLIED>
<!ATTLIST render-kit-id                          id ID #IMPLIED>
<!ATTLIST renderer                               id ID #IMPLIED>
<!ATTLIST renderer-class                         id ID #IMPLIED>
<!ATTLIST renderer-extension                     id ID #IMPLIED>
<!ATTLIST renderer-type                          id ID #IMPLIED>
<!ATTLIST small-icon                             id ID #IMPLIED>
<!ATTLIST state-manager                          id ID #IMPLIED>
<!ATTLIST suggested-value                        id ID #IMPLIED>
<!ATTLIST supported-locale                       id ID #IMPLIED>
<!ATTLIST to-view-id                             id ID #IMPLIED>
<!ATTLIST validator                              id ID #IMPLIED>
```

```
<!ATTLIST validator-class                          id ID #IMPLIED>
<!ATTLIST validator-id                             id ID #IMPLIED>
<!ATTLIST value                                    id ID #IMPLIED>
<!ATTLIST value-class                              id ID #IMPLIED>
<!ATTLIST variable-resolver                        id ID #IMPLIED>
<!ATTLIST view-handler                             id ID #IMPLIED>
```

10.3.4 Configuration Impact on JSF Runtime

The following XML elements[1] in application configuration resources cause registration of JSF objects into the corresponding factories or properties. It is an error if the value of any of these elements cannot be correctly parsed, loaded, set, or otherwise used by the implementation.

- **/faces-config/component** -- Create or replace a component type / component class pair with the `Application` instance for this web application.
- **/faces-config/converter** -- Create or replace a converter id / converter class or target class / converter class pair with the `Application` instance for this web application.
- **/faces-config/render-kit** -- Create and register a new `RenderKit` instance with the `RenderKitFactory`, if one does not already exist for the specified `render-kit-id`.
- **/faces-config/render-kit/renderer** -- Create or replace a component family + renderer id / renderer class pair with the `RenderKit` associated with the render-kit element we are nested in.
- **/faces-config/validator** -- Create or replace a validator id / validator class pair with the `Application` instance for this web application.

For components, converters, and validators, it is legal to replace the implementation class that is provided (by the JSF implementation) by default. This is accomplished by specifying the standard value for the `<component-type>`, `<converter-id>`, or `<validator-id>` that you wish to replace, and specifying your implementation class. To avoid class cast exceptions, the replacement implementation class must be a subclass of the standard class being replaced. For example, if you declare a custom

1. Identified by XPath selection expressions.

`Converter` implementation class for the standard converter identifier `javax.faces.Integer`, then your replacement class must be a subclass of `javax.faces.convert.IntegerConverter`.

For replacement `Renderer`s, your implementation class must extend `javax.faces.render.Renderer`. However, to avoid unexpected behavior, your implementation should recognize all of the render-dependent attributes supported by the Renderer class you are replacing, and provide equivalent decode and encode behavior.

The following XML elements cause the replacement of the default implementation class for the corresponding functionality, provided by the JSF implementation. See Section 10.3.5 "Delegating Implementation Support" for more information about the classes referenced by these elements:

- **/faces-config/application/action-listener** -- Replace the default `ActionListener` used to process `ActionEvent events with an` instance with the class specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is an `ActionListener`.
- **/faces-config/application/navigation-handler** -- Replace the default `NavigationHandler` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `NavigationHandler`.
- **/faces-config/application/property-resolver** -- Replace the default `PropertyResolver` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `PropertyResolver`.
- **/faces-config/application/state-manager** -- Replace the default `StateManager` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `StateManager`.
- **/faces-config/application/variable-resolver** -- Replace the default `VariableResolver` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `VariableResolver`.
- **/faces-config/application/view-manager** -- Replace the default `ViewManager` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `ViewManager`.

The following XML elements cause the replacement of the default implementation class for the corresponding functionality, provided by the JSF implementation. Each of the referenced classes must have a public zero-arguments constructor:

- **/faces-config/factory/application-factory** -- Replace the default `ApplicationFactory` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is an `ApplicationFactory`.

- **/faces-config/factory/faces-context-factory** -- Replace the default `FacesContextFactory` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `FacesContextFactory`.
- **/faces-config/factory/lifecycle-factory** -- Replace the default `LifecycleFactory` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `LifecycleFactory`.
- **/faces-config/factory/render-kit-factory** -- Replace the default `RenderKitFactory` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `RenderKitFactory`.

The following XML elements cause the addition of event listeners to standard JSF implementation objects, as follows. Each of the referenced classes must have a public zero-arguments constructor.

- **/faces-config/lifecycle/phase-listener** -- Instantiate a new instance of the specified class, which must implement `PhaseListener`, and register it with the `Lifecycle` instance for the current web application.

In addition, the following XML elements influence the runtime behavior of the JSF implementation, even though they do not cause registration of objects that are visible to a JSF-based application.

- **/faces-config/managed-bean** -- Make the characteristics of a managed bean with the specified `managed-bean-name` available to the default `VariableResolver` implementation.
- **/faces-config/navigation-rule** -- Make the characteristics of a navigation rule available to the default `NavigationHandler` implementation.

10.3.5 Delegating Implementation Support

When providing a replacement for the default `PropertyResolver`, `VariableResolver`, `ActionListener`, `NavigationHandler`, `ViewHandler`, or `StateManager`, the decorator design pattern is leveraged, so that if you provide a constructor that takes a single argument of the appropriate type, the custom implementation receives a reference to the implementation that was previously fulfilling the role. In this way, the custom implementation is able to override just a subset of the functionality (or provide only some additional functionality) and delegate the rest to the existing implementation.

For example, say you wanted to provide a custom `ViewHandler` that was the same as the default one, but provided a different implementation of the `calculateLocale()` method. Consider this code excerpt from a custom `ViewHandler`:

```
public class MyViewHandler extends ViewHandler {

  public MyViewHandler() { }

  public MyViewHandler(ViewHandler handler) {
    super();
    oldViewHandler = handler;
  }

  private ViewHandler oldViewHandler = null;

  // Delegate the renderView() method to the old handler
  public void renderView(FacesContext context, UIViewRoot view)
      throws IOException, FacesException {
    oldViewHandler.renderView(context, view);
  }

  // Delegate other methods in the same manner

  // Overridden version of calculateLocale()
  public Locale calculateLocale(FacesContext context) {
    Locale locale = ... // Custom calculation
    return locale;
  }

}
```

The second constructor will get called as the application is initially configured by the JSF implementation, and the previously registered `ViewHandler` will get passed to it.

10.3.6 Example Application Configuration Resource

The following example application resource file defines a custom `UIComponent` of type `Date`, plus a number of `Renderer`s that know how to decode and encode such a component:

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">

<faces-config>

<!-- Define our custom component -->
<component>
  <description>
    A custom component for rendering user-selectable dates in
    various formats.
  </description>
  <display-name>My Custom Date</display-name>
  <component-type>Date</component-type>
  <component-class>
    com.example.components.DateComponent
  </component-class>
</component>

<!-- Define two renderers that know how to deal with dates -->
<render-kit>
  <!-- No render-kit-id, so add them to default RenderKit -->
  <renderer>
    <display-name>Calendar Widget</display-name>
    <component-family>MyComponent</component-family>
    <renderer-type>MyCalendar</renderer-type>
    <renderer-class>
      com.example.renderers.MyCalendarRenderer
    </renderer-class>
  </renderer>
  <renderer>
    <display-name>Month/Day/Year</display-name>
    <renderer-type>MonthDayYear</renderer-type>
    <renderer-class>
      com.example.renderers.MonthDayYearRenderer
    </renderer-class>
  </renderer>
</render-kit>

</faces-config>
```

Additional examples of configuration elements that might be found in application configuration resources are in Section 5.3.1.4 “Managed Bean Configuration Example” and Section 7.4.3 “Example NavigationHandler Configuration”.

CHAPTER 11

Lifecycle Management

In Chapter 2 “Request Processing Lifecycle,” the required functionality of each phase of the request processing lifecycle was described. This chapter describes the standard APIs used by JSF implementations to manage and execute the lifecycle. Each of these classes and interfaces is part of the `javax.faces.lifecycle` package.

Page authors, component writers, and application developers, in general, will not need to be aware of the lifecycle management APIs—they are primarily of interest to tool providers and JSF implementors.

11.1 Lifecycle

Upon receipt of each JSF-destined request to this web application, the JSF implementation must acquire a reference to the `Lifecycle` instance for this web application, and call its `execute()` and `render()` methods to perform the request processing lifecycle. The `Lifecycle` instance invokes appropriate processing logic to implement the required functionality for each phase of the request processing lifecycle, as described in Section 2.2 “Standard Request Processing Lifecycle Phases”.

```
public void execute(FacesContext context) throws FacesException;

public void render(FacesContext context) throws FacesException;
```

The `execute()` method performs phases up to, but not including, the *Render Response* phase. The `render()` method performs the Render Response phase. This division of responsibility makes it easy to support JavaServer Faces processing in a portlet-based environment.

As each phase is processed, registered `PhaseListener` instances are also notified. The general processing for each phase is as follows:

- From the set of registered `PhaseListener` instances, select the relevant ones for the current phase, where "relevant" means that calling `getPhaseId()` on the `PhaseListener` instance returns the phase identifier of the current phase, or the special value `PhaseId.ANY_PHASE`.
- Call the `beforePhase()` method of each relevant listener, in the order that the listeners were registered.
- If no called listener called the `FacesContext.renderResponse()` or `FacesContext.responseComplete()` method, execute the functionality required for the current phase.
- Call the `afterPhase()` method of each relevant listener, in the reverse of the order that the listeners were registered.
- If the `FacesContext.responseComplete()` method has been called during the processing of the current request, or we have just completed the *Render Response* phase, perform no further phases of the request processing lifecycle.
- If the `FacesContext.renderResponse()` method has been called during the processing of the current request, and we have not yet executed the *Render Response* phase of the request processing lifecycle, ensure that the next executed phase will be *Render Response*

```
public void addPhaseListener(PhaseListener listener);

public void removePhaseListener(PhaseListener listener);
```

These methods register or deregister a `PhaseListener` that wishes to be notified before and after the processing of each standard phase of the request processing lifecycle. The webapp author can declare a `PhaseListener` to be added using the `phase-listener` element of the application configuration resources file. Please see *Section 11.3 "PhaseListener"*.

11.2 PhaseEvent

This class represents the beginning or ending of processing for a particular phase of the request processing lifecycle, for the request encapsulated by the `FacesContext` instance passed to our constructor.

```
public PhaseEvent(FacesContext context, PhaseId phaseId);
```

Construct a new PhaseEvent representing the execution of the specified phase of the request processing lifecycle, on the request encapsulated by the specified FacesRequest instance.

```
public FacesContext getFacesContext();

public PhaseId getPhaseId();
```

Return the properties of this event instance. The specified FacesContext instance will also be returned if getSource() (inherited from the base EventObject class) is called.

11.3 PhaseListener

This interface must be implemented by objects that wish to be notified before and after the processing for a particular phase of the request processing lifecycle, on a particular request. Implementations of PhaseListener must be programmed in a thread-safe manner.

```
public PhaseId getPhaseId();
```

The PhaseListener instance indicates for which phase of the request processing lifecycle this listener wishes to be notified. If PhaseId.ANY_PHASE is returned, this listener will be notified for all standard phases of the request processing lifecycle.

```
public void beforePhase(PhaseEvent event);

public void afterPhase(PhaseEvent event);
```

The beforePhase() method is called before the standard processing for a particular phase is performed, while the afterPhase() method is called after the standard processing has been completed. The JSF implementation must guarantee that, if beforePhase() has been called on a particular instance, then afterPhase() will also be called.

PhaseListener implementations may affect the remainder of the request processing lifecycle in several ways, including:

- Calling `renderResponse()` on the `FacesContext` instance for the current request, which will cause control to transfer to the *Render Response* phase of the request processing lifecycle, once processing of the current phase is complete.
- Calling responseComplete() on the FacesContext instance for the current request, which causes processing of the request processing lifecycle to terminate once the current phase is complete.

11.4 LifecycleFactory

A single instance of `javax.faces.lifecycle.LifecycleFactory` must be made available to each JSF-based web application running in a servlet or portlet container. The factory instance can be acquired by JSF implementations or by application code, by executing:

```
LifecycleFactory factory = (LifecycleFactory)
   FactoryFinder.getFactory(FactoryFinder.LIFECYCLE_FACTORY);
```

The `LifecycleFactory` implementation class supports the following methods:

```
public void addLifecycle(String lifecycleId, Lifecycle lifecycle);
```

Register a new `Lifecycle` instance under the specified lifecycle identifier, and make it available via calls to the `getLifecycle` method for the remainder of the current web application's lifetime.

```
public Lifecycle getLifecycle(String lifecycleId);
```

The `LifecycleFactory` implementation class provides this method to create (if necessary) and return a `Lifecycle` instance. All requests for the same lifecycle identifier from within the same web application will return the same `Lifecycle` instance, which must be programmed in a thread-safe manner.

Every JSF implementation must provide a `Lifecycle` instance for a default lifecycle identifier that is designated by the `String` constant `LifecycleFactory.DEFAULT_LIFECYCLE`. For advanced uses, a JSF implementation may support additional lifecycle instances, named with unique lifecycle identifiers.

```
public Iterator getLifecycleIds();
```

This method returns an iterator over the set of lifecycle identifiers supported by this factory. This set must include the value specified by `LifecycleFactory.DEFAULT_LIFECYCLE`.