JavaServer™ Faces Specification

Version 1.0, Expert Draft 20030121

January 21, 2003

Craig R. McClanahan, editor

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

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

JavaServer™ Faces Specification i

JavaServer(TM) Faces Specification ("Specification")
Version: 1.0
Status: Pre-FCS, Early Access Draft
Release: September 6, 2002

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).

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>

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

- JavaServer Pages™ Standard Tag Library, version 1.0 (JSTL)
  <http://java.sun.com/products/jsp/jstl/>
- 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>
- XForms 1.0 (currently in Working Draft state) <http://www.w3.orgTR/xforms/>

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 is the result of collaborative work involving many individuals on the JSR-127 expert group ... (FIXME - individual acknowledgements as needed)

JSF.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 which 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.

JSF.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 aimed directly at solving many of the common problems encountered when writing applications for HTML clients which communicate via HTTP to a Java application server that supports servlets and JavaServer Pages (JSP) based applications. These applications are typically form-based, comprised of one or more HTML pages which the user interacts with 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, etc)
- providing a strongly typed event model which 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 component values (Strings) to/from application data objects (often not Strings)
- handling error and exceptions, and reporting errors in human-readable form back to the application user

JSF.1.2 Specification Audience

The JSF 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 audiencies, 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.

JSF.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 language(s) (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).

From the perspective of JSF, 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 reprented 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 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.

JSF.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 representaiton of this component on an incoming request, as well as conversion into the internal form that is appropriate for this component.

As discussed in Chapter FIXME, 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 (FIXME - add this to a references list). 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.

JSF.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 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 *model reference expressions*, as described in Section FIXME
- Provide application-level event handlers for the *Command Events* and *Form Events* that are enqueued by JSF components during the request processing lifecycle, as described in Section FIXME

Application modules interact with JSF through standard Java 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.

JSF.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.

JSF.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 Java2 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 (such as the Lifecycle and Phase APIs described in Chapter FIXME/6) 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.

JSF.1.3 Introduction to JSF APIs

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

FIXME - add summary of each package here.

JSF.1.4 Using the JSF APIs

FIXME - some simple examples of using JSF.

JSF.2

Request Processing Lifecycle

Each servlet request that involves a JSF component tree 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 servlet response that was created by the execution of the *Render Response* phase of the request processing lifecycle.
- *Non-Faces Response* -- A servlet 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 servlet request that was sent from a previously generated *Faces Response*. Examples would be a hyperlink or form submit from a rendered user interface comonent, where the request URI was crafted (by the component or renderer that created it) to trigger the Faces Request Generates Faces Response request processing lifecycle.
- *Non-Faces Request* -- A servlet request that was sent to an application component, rather than via the JSF implementation.

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.

JSF.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.

JSF.2.1.1 Non-Faces Request Generates Faces Response

An application that is processing a Non-Faces Request may desire to utilize 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 FIXME/2.4.1, below.
- Create A New Component Tree, as described in Section FIXME/2.4.2, below.
- Call `renderResponse()` on the `FacesContext` instance that was acquired. This signals the JSF implementation that processing should immediately proceed to the *Render Response* phase of the request processing lifecycle.
- Call the `execute()` method on the `Lifecycle` instance that was acquired. This signals the JSF implementation to begin processing at the next appropriate phase of the request processing lifecycle. Because of thecall to `renderResponse()` in the previous step, control will proceed immediately to rendering.

JSF.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 FIXME/9.1.2. 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 handler will be invoked at appropriate times as the standard request processing lifecycle, described in the following diagram, is invoked.

[ED. NOTE -- Revised diagram goes here]

The behavior of the individual phases of the request processing lifecycle are described in individual subsections of Section FIXME/2.2, below. Note that, at the conclusion of several phases of the request processing lifecycle, common event processing logic (as described in Section FIXME/2.3, below) is performed to broadcast any `FacesEvents` generated by components in the component tree to interested event listeners.

JSF.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 servlet response that is sent back to the client. In some circumstances, however, this behavior might not be desireable. 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 binary (such as an image) or text (such as XML) response instead of a response intended for human consumption (such as HTML sent back to a browser).

In any of these scenarios, the application will have used the standard mechanisms of the servlet API to create the response headers and content. What remains is the necessity 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 the application handler.

JSF.2.2 Standard Request Processing Lifecycle Phases

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

JSF.2.2.1 Reconstitute Component Tree

The JSF implementation must perform the following tasks during the *Reconstitute Component Tree* phase of the request processing lifecycle:

- Acquire, from the incoming request and/or saved information in the user's session, the state information required to construct the component tree (if any was saved by a previous JSF-generated response that generated the page from which this request was submitted). In order to be utilized, such state information must meet the following requirements:

    - It must have a tree identifier that corresponds to the request URI used to submit this request. In other words, if the JSF implementation has saved state information for a tree identifier **/foo** in the user's session, but receives a request with URI **/faces/bar**, the saved state information will be thrown away.
- If such state information is available, use it to construct a component `Tree`, including any required wire-up of event listeners and validators, and setting the appropriate `RenderKit` instance. Save the created `Tree` instance in the `FacesContext` for the current request, by passing it as a parameter to the `setTree()` method.
- If no such state information is available, or if any saved state information was for a different page than the one processing this request, acquire a component `Tree` for the requested tree identifier, by calling the `getTree()` method of the `TreeFactory` instance for this web application, and store it in the `FacesContext` by calling `setTree()`. The tree identifier for this tree is extracted from the *extra path info* portion of the request URI. This means that, when using the standard servlet mapping of **/faces/*** to `javax.faces.webapp.FacesServlet`, a request URI `/faces/menu.jsp` will select tree identifier `/menu.jsp`.
- Call the `setLocale()` method of the `FacesContext` instance for the current request, passing a `Locale` instance derived from the saved state information (if any); otherwise, acquire the default `Locale` to be used by calling the `getLocale()` method of the `ServletRequest` for this request.

At the end of this phase, the `tree` property of the `FacesContext` instance for the current request will reflect the saved configuration of the component tree generated by the previous Faces Response (if any).

JSF.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 it's current value 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 root component in 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.

During the decoding of component values, events may have been queued by the components and/or renderers whose `decode()` method was invoked. Perform the *Common Event Processing*, described in Section FIXME/2.3, below, to ensure that all such events are broadcast to all interested event listeners.

At the end of this phase, all components in the component tree will have been updated with new values included in this request (or enough data to reproduce incorrect input will have been stored, if there were conversion errors). Conversions 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 have been set to `false`.

If any of the `decode()` methods that was 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 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 *Process Validations* phase.

JSF.2.2.3 Process Validations

As part of the creation of the component tree 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 root component in the component tree. This will normally cause the `processValidators()` method of each component in the tree to be called recursively, as described in the Javadocs for the `UIComponent.processValidators()` method.

During the processing of validations, events may have been queued by the components and/or `Validator`s whose `validate()` method was invoked. Perform the *Common Event Processing*, described in Section FIXME/2.3, below, to ensure that all such events are broadcast to all interested event listeners.

At the end of this phase, all configured validations will have been completed. 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 was 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 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. If there is at least one message queued via a call to the `addMessage()` method on the `FacesContext` instance for the current request (indicating a conversion failure or a validation failure), control must be transferred to the *Render Response* phase of the request processing lifecycle. Otherwise, control must proceed to the *Update Model Values* phase.

JSF.2.2.4 Update Model Values

If this phase of the request processing lifecycle is reached, it can be 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 root component in the component tree. This will normally cause the `processUpdates()` method of each component in the tree to be called recursively, as described in the Javadocs 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. Perform the *Common Event Processing*, described in Section FIXME/2.3, below, to ensure that all such events are broadcast to all interested event 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 was 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. [FIXME -- error detection for conversion errors during model updates???]. Otherwise, control must proceed to the *Invoke Application* phase.

JSF.2.2.5 Invoke Application

[ED. NOTE -- The behavior of this phase will be revised in a future version of this specification. The current mechanism is a placeholder until a more sophisticated interface design is completed.]

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

- If no application events have been queued for this request, take no further action.
- Call the `getApplicationHandler()` method of the `FacesContext` instance for the current request, to get a reference to the registered application event handler (see section FIXME/6.3 for more information). If no such handler has been registered, throw a `FacesException`.
- Call the `getApplicationEvents()` method on the `FacesContext` instance for the current request, and dispatch each queued event to the `processEvent()` method on the application event handler.
- If the `processEvent()` method of the application event handler returns `true`, return Phase.GOTO_RENDER in order to proceed immediately to the *Render Response* phase. Otherwise, continue event handling until all queued application events have been processed.

Application event handlers can perform whatever application-level functions are appropriate to deal with the event(s) that have been dispatched. However, application event handlers may perform the following operations that are directly relevant to JSF request lifecycle processing:

- Create a new component tree for the response, instead of reusing the one associated with the current request. See Section FIXME/2.4.2 for details.
- Add and remove components in the component tree, and/or customize their attributes.
- Change the `RenderKit` that will be utilized during the *Render Response* phase for the current response.
- Enqueue error messages by calling an appropriate `addMessage()` method on the `FacesContext`.
- Enqueue one or more additional application events to be handled during this phase.

If the application event handler called `responseComplete()` on the `FacesContext` instance for the current request, lifecycle processing of the current request must be immediately terminated. Otherwise, control must proceed to the *Invoke Application* phase.

JSF.2.2.6 Render Response

JSF supports a range of approaches that JSF implementations may utilize in creating the response text that corresponds to the contents of the response component tree, 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 derived from the tree identifier of the component tree [FIXME - specify `NavigationHandler` mapping mechanism that supports abstraction and i18n support for this purpose].
- If all of the response content is being derived from the encoding methods of the component or associated `Renderer`s, the component tree should be walked in the same depth-first manner as was used in earlier phases to process the request 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 check for the existence of the corresponding component in the component tree first. If the component already exists (perhaps because a previous phase has precreated one or more components), the component attributes that already exist must override any attribute settings that the `ViewHandler` attempts to make.
- 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 will (or will have) rendered the content of this child component when the parent or ancestor was selected.

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 FIXME/3.1.11.

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 FIXME/8.

Upon completion of rendering, the completed state of the component tree must be saved in either the response being created, in the user's session, or some combination of the above, in an implementation-defined manner. This state information must be made accessible on a subsequent request, so that the *Reconstitute Component Tree* can access it. For example, the saved state information could be encoded in an <input type="hidden"> field inside an HTML <form> to be submitted by the user.

[FIXME - say something about setting response characteristics such as cookies and headers -- most particularly the content type header?]

[FIXME - suggestion that encoders do URL rewriting on hyperlinks to maintain session state?]

JSF.2.3 Common Event Processing

[ED. NOTE -- The actual implementation of the current event processing model is a little awkward, and shares responsibilities between the JSF implementation and the component's broadcast() method clumsily. This needs to be reviewed and refined.]

For a complete description of the event processing model for JavaServer Faces components, see Section FIXME/3.?.

During several phases of the request processing lifecycle, as described in Section FIXME/2.2 above, the possibility exists for events to be queued, which must now be broadcast to interested event listeners. The JSF implementation must take to perform this broadcast.

- Call the `getFacesEvents()` method on the `FacesContext` instance associated with the current request, to an acquire an `Iterator` over the events (if any) that have been queued.
- For each queued event, call `getComponent()` to acquire a reference to the source component for that event, and call the `broadcast()` method on that component, passing the event instance and an identifier for the current phase of the request processing lifecycle.

In turn, the `broadcast()` method of each `UIComponent` class must broadcast the event to all event listeners who have registered an interest, on this source component, for events of the specified type, and then return a boolean flag indicating whether this event has been completely handled and that the JSF implementation must remove it from the event queue (by calling the `remove()` method on the `Iterator` being processed. See the JavaDocs 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. The `Iterator` implementation returned by `getFacesEvents()` must support dynamic modifications to the event list that is being iterated over, as described in the JavaDocs for the `getFacesEvents()` method.

JSF.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 FIXME/2.1, above, for each request processing lifecycle scenario in which the activity is relevant.

JSF.2.4.1 Acquire Faces Object References

This phase is only required when the servlet 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 follows:

JSF.2.4.1.1 Acquire and Configure Lifecycle Reference

As described in Section FIXME/6.1, 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 not be portable to any other JSF implementation.

If this is the first time you have acquired a reference to the `Lifecycle` instance for a particular lifecycle identifier, you may configure the `ApplicationHandler` that your application will provide to process application events, and/or the `ViewHandler` that the JSF implementation should use during theRender Response phase of the request processing lifecycle.

If you do not define an ApplicationHandler instance, no application events will be processed during the *Invoke Application* phase of the request processing lifecycle. As described in Section FIXME/6.1, a default ViewHandler implementation is used if you do not specify a different one.

JSF.2.4.1.2 Acquire and Configure FacesContext Reference

[ED. NOTE -- the details of the following configuration will change in a subsequent version of this specification.]

As described in Section FIXME/5.1, 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 wanting 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);
```

JSF.2.4.2 Create A New Component Tree

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

[FIXME -- consider abstracting the nitty gritty details into easier-to-user utility methods.]

JSF.2.4.2.1 Create A New Component Tree

As described in Section FIXME/3.2, component trees are represented by a data structure rooted in an instance of `javax.faces.tree.Tree`, and identified by a tree identifier whose meaning depends on the ViewHandler implementation to be used during the *Render Response* phase of the request processing lifecycle[1]. In addition, it provides a TreeFactory that may be utilized to construct new component trees, as follows:

```
String treeId = ... identifier of the desired Tree ...;
TreeFactory tFactory = (TreeFactory)
FactoryFinder.getFactory(FactoryFinder.TREE_FACTORY);
Tree tree = tFactory.getTree(facesContext, treeId);
```

The `Tree` instance returned by the `getTree()` method will minimally contain a single `UIComponent` provided by the JSF implementation, which will encapsulate any implementation-specific component management that is required. Optionally, a

1. 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

JSF implementation's `TreeFactory` may support the automatic population of the returned `Tree` with additional components, perhaps based on some external metadata description.

JSF.2.4.2.2 Configure the Desired RenderKit

The `Tree` instance provided by the `TreeFactory`, as described in the previous subsection, will automatically be configured to utilize the default javax.faces.render.`RenderKit` implementation provided by the JSF implementation, as described in Section FIXME/7.1. This `RenderKit` must support the standard components and `Renderer`s described later in this specification, so its using it will 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 from the standard `RenderKitFactory`, and then assigned to the `Tree` 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);
tree.setRenderKit(renderKit);
```

As described in Chapter FIXME/7, 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 (perhaps based on particular characteristics of the servlet request, such as which user agent is being used or the preferred `Locale` of the user), as long as they support Renderers with the same renderer types.

JSF.2.4.2.3 Switch To The New Tree

Now that the application has a reference to a `Tree` instance, it can tell the JSF implementation to utilize the new `Tree` as follows:

```
facesContext.setTree(tree);
```

Note that, once this call is performed, the component tree that was previously created by a Faces Request (if any) is no longer accessible. Be sure that you have retrieved any information you want from the previous `Tree` instance before switching.

JSF.2.4.2.4 Configure The Tree's Components

At any time, the application can add new components to the component tree, 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 `UIComponent` in the component tree as follows:

```
FooComponent component = ... create a FooComponent instance ...;

facesContext.getTree().getRoot().addChild(component);
```

JSF.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 application, via *model reference expressions*.

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

- Converters -- Pluggable support class to convert the local 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 *component tree*. The components in the tree have parent-child relationships with other components, starting at the *root element* of the tree. 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.

JSF.3.1 UIComponent and UIComponentBase

The base class for all user interface components is an abstract class, `javax.faces.component.UIComponent`. This class defines the state information and behavioral contracts for all components through a Java API, which means that components are independent of a rendering technology such as JavaServer Pages (JSP). A standard set of components (described in Chapter FIXME/4) 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` subclasses for use within a particular application. To assist such developers, a concrete 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.

JSF.3.1.1 Component Type

- `public abstract String getComponentType();`

All concrete `UIComponent` subclasses must override this abstract method and return a non-`null` value. The component type is used to categorize components by a means other than its Java class name or class parentage. When component rendering is delegated to an external `RenderKit`, the component type may be used to select one of the available `Renderer`s that know how to visualize components of a particular type.

JSF.3.1.2 Component Identifiers

- `public String getComponentId();`
- `public void setComponentId(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. For maximum portability, component identifiers must conform to the following rules:

- Composed of letters ('a'..'z', 'A'..'Z'), digits ('0'..'9'), dashes ('-'), and underscores ('_') from the USASCII character set.
- Must start with a letter.

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

If a component has been given a name, it must be unique in the namespace of the closest ancestor to that component that is a naming container. The root component in the tree must be a naming container.

JSF.3.1.3 Client Identifiers

Client identifiers are used by JSF implementations, as they decode and encodecomponents, 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

The client identifier is derived from the component identifier, if present, or it is generated by the closest ancestor to the component that is a naming container.

- `public String getClientId(FacesContext context);`

For component classes derived from `UIComponentBase`, the client identifier is obtained by executing the following algorithm:

Look up this component's "clientId" attribute. If non-null, return it. If null, see if we have a Renderer and if so, delegate to it. If we don't have a Renderer, get the component id for this UIComponent. If null, generate one using the closest naming container that is an ancestor of this UIComponent, then set the generated id as the componentId of this UIComponent. Prepend to the component id the component ids of each naming container up to, but not including, the root, separated by the UIComponent.SEPARATOR_CHAR. In all cases, save the result as the value of the "clientId" attribute.

JSF.3.1.4 Component Tree Manipulation

- `public UIComponent getParent();`

Components that have been added as children of another component can identify the parent by calling `getParent()`. For the root node component of a component tree, or any component that is not part of a component tree, this method will return `null`.

- `public void addChild(UIComponent component);`
- `public void addChild(int index, UIComponent component);`

Adds a new child component to the set of children associated with the current component, either at the end of the list, or at the specified (zero-relative) position.

- `public void clearChildren();`

Remove all children from the child list of the current component.

- `public boolean containsChild(UIComponent component);`

Return `true` if the specified component instance is a child of the current component.

- `public UIComponent getChild(int index);`

Return the component at the specified (zero-relative) position in the child list for this component.

- `public int getChildCount();`

Return the number of child components that are associated with this component.

- `public Iterator getChildren();`

Return an `Iterator` over the child components associated with this component.

- `public void removeChild(int index);`
- `public void removeChild(UIComponent component);`

Remove the specified component from the child list for this component.

JSF.3.1.5 Component Tree Navigation

- `public UIComponent findComponent(String expression);`

Find the named component by locating the closest ancestor to this component that is a naming container and calling its `findComponentInNamespace()` method.

JSF.3.1.6 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 FIXME/3.1.4. However, an additional useful facility would be 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 FIXME/3.1.13.

- `public void addFacet(String name, UIComponent facet);`
- `public UIComponent getFacet(String name);`
- `public Iterator getFacetNames();`
- `public void removeFacet(String name);`

See the API Javadocs in Section FIXME for more details about the functionality of these methods.

For easy use of components that use facets, it is recommended that component authors 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 that must be of type `UIHeader` should have methods with signatures and functionality as follows:

```
public UIHeader getHeader() {
   return ((UIHeader) getFacet(“header”));
}
public void setHeader(UIHeader header) {
   addFacet(“header”, header);
}
```

JSF.3.1.7 NamingContainer and UINamingContainer

The `NamingContainer` interface defines a simple, flexible way for components to have names on the client and server side of a Faces application. The API provides a default implementation of `NamingContainer` in the class `UINamingContainer`. Any implementation of `NamingContainer` must also implement `UIComponent`.

Any JSF implementation must make the root of the component tree implement `NamingContainer`. This is most easily accomplished by having the root be an instance of `UINamingContainer`.

- `public void addComponentToNamespace(UIComponent namedComponent);`

- `public void removeComponentFromNamespace(UIComponent namedComponent);`
- `public UIComponent findComponentInNamespace(String name);`
- `public String generateClientId();`

The add and remove methods above simply add to or remove from the namespace of this naming container, using the *component identifier* of the argument component as the name.

`findComponentInNamespace()` looks in the namespace of this naming container for a component under the argument name. If the argument name contains `UIComponent.SEPARATOR_CHAR` characters, each segment between the `UIComponent.SEPARATOR_CHAR` characters is treated as a component identifier in the previous naming container's namespace.

Consider the following usage: `namingComponent.findComponent("containerOne.containerTwo.leafTwo")`, where `namingComponent` is a `UIComponent` that implements `NamingContainer`. This call will end up calling NamingContainer.`findComponentInNamespace`(). The first two segments must be the component identifiers of NamingContainer instances. `containerOne` is in `namingComponent's` namespace, `containerTwo` is in `containerOne's` namespace, and `leafTwo` is in containerTwo's namespace.

`generateClientId()` returns a string of the form "`id`*N* " where *N* is a serial number generated by the NamingContainer. This method is called when the *client identifier* of a component is requested, but the component has no `component identifier`.

JSF.3.1.8 Generic Attributes

- `public Object getAttribute(String name);`
- `public Iterator getAttributeNames();`
- `public void setAttribute(String name, Object value);`

The render-independent characteristics of components are generally represented as JavaBean properties with getter and setter methods (see the following section for details). 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.
- Configuration parameters to be utilized by `Converter`s, event listeners, and `Validator`s associated with the component.
- General purpose association of application-specific objects with components.

The attributes for a component may be of any Java object type, and are keyed by attribute name (a String). Null attribute values are not allowed; calling `setAttribute()` with a null value parameter is equivalent to removing this attribute. The names of all attributes that currently have a (non-null) value are available via the `getAttributeNames()` method.

Attribute names that begin with "`javax`." or "`javax.faces`" are reserved for use by the JSF implementation.

JSF.3.1.9 Render-Independent Properties

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

Name	Access	Type	Description
componentId	RW	String	The component identifier, as described in Section FIXME/ 3.1.2
componentTy pe	RO	String	The component type, as described in Section FIXME/3.1.1.
modelReferen ce	RW	String	Model reference expression used to retrieve data values from model tier beans (during *Render Response* phase), or to transfer component values to model tier beans (during *Apply Request Values* phase). The expression syntax corresponds to a variable reference in the expression language defined by JSTL 1.0, but without the "${" and "}" delimiters.
parent	RO	UIComponent	The parent component for which this component is a child.
rendered	RW	boolean	A flag that, if set to "true", indicates that this component should be rendered during *Render Response* phase of the request processing lifecycle.
rendererType	RW	String	Identifier of the `Renderer` instance (from the set of `Renderer`s 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.

Name	Access	Type	Description
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` returns `false` for this property.
rendersSelf	RO	boolean	A flag that, if set to `true`, indicates that this component has concrete implementations of the decode() and encodeXxx() methods, and is therefore suitable for use even if rendererType is not set. The default implementation in `UIComponentBase` returns `false` for this property.
valid	RW	boolean	A flag that, if set to `true`, indicates that the local value of this component was successfully converted, passes all validations, and was successfully stored during the `Update Model Values` phase of the request processing lifecycle. JSF classes set this property to `false` at various points in the request processing lifecycle, as described in Section FIXME/2.2, to indicate that a problem was encountered.
value	RW	Object	The local value of a JSF component. During the *Apply Request Values* phase of the request processing lifecycle, the decode() method of the component (or renderer) will convert incoming request parameters into an object representing the new value of the component, and store it in this property. See below for information on the `currentValue()` method.

The method names for the render-independent property getters and setters must conform to the design patterns in the JavaBeans specification. For boolean properties, the "is" form of the method name must be utilized (so the `valid` property is retrieved by calling `isValid()`).

To facilitate the use of component classes in tools, all of these JavaBean properties available via getter and setter methods must also be available as attributes under the same name as the property. In other words, a model reference expression stored by a call to the `setModelReference()` will be accessible as an attribute by calling `getAttribute("modelReference")`, and vice versa. This implies that property getter and setter method implementations, in component classes, will generally call the corresponding attribute methods internally.

In addition to the `value` property, which stores a component's local value, `UIComponent` also supports a `currentValue()` method that returns the local value if there is one, or uses the value of the `modelReference` property to retrieve a value from a web tier bean.

JSF.3.1.10 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, PhaseId phaseId) throws AbortProcessingException;`

The `broadcast()` method is called during the common event processing (see Section FIXME/2.3) at the end of several request processing lifecycle pahses. For more information about the event and listener model, see Section FIXME/3.3, below.

- `public void decode(FacesContext context) throws IOException;`

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 the request parameters of 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). However, overriding this method allows a component writer to create a custom component, complete with its own decoding, encoding, and validation behavior, in a single class if desired.

- `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. 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). However, overriding these methods allows a component writer to create a custom component, complete with its own decoding, encoding, and validation behavior, in a single class if desired.

- `public void validate(FacesContext context);`

This method is called during the *Process Validations* phase of the request processing lifecycle, and allows the component author to embed any desired correctness checks directly in the component class. The default implementation in `UIComponentBase` performs no correctness checks. For more information about the validation model that JSF provides, see Section FIXME/3.4, below.

- `public void updateModel(FaceContext context);`

This method is called during the *Update Model Values* phase of the request processing lifecycle, and is responsible for transferring the local value of this component back to a model tier bean via the model reference expression (with appropriate conversions as necessary). The default implementation in `UIComponentBase` calls the `setModelValue()` method on the `FacesContext` instance for the current request, and then clears the local value, as described in the JavaDocs for this method.

JSF.3.1.11 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.

- `public void processDecodes(FacesContext context) throws IOException;`

This method is called (on the root component of a component tree) during the *Apply Request Values* phase of the request processing lifecycle. The default implementation in `UIComponentBase` ensures that the `processDecodes()` method of all facets of this component, then all children of this component, followed by the `decode()` method of this component, are called.

- `public void processValidators(FacesContext context);`

This method is called (on the root component of a component tree) during the *Process Validations* phase of the request processing lifecycle. The default implementation in `UIComponentBase` ensures that the `processValidators()` method of all facets and children of this component are called appropriately, followed by calls to the `validate()` method of all validators registered to this component, and then the `validate()` method of this component itself.

- `public void processUpdates(FacesContext context);`

This method is called (on the root component of a component tree) during the *Update Model Values* phase of the request processing lifecycle. The default implementation in `UIComponentBase` ensures that the `processUpdates()` method of all facets of this component, then all children of this component, followed by the `updateModel()` method of this component, are called.

JSF.3.2 Conversion Model

[ED. NOTE -- The description of Converter and ConverterFactory, currently in Sections 7.4 and 7.5, will be migrated here in a future version of this specification.]

JSF.3.3 Event and Listener Model

JSF.3.3.1 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.

JSF.3.3.2 Event Classes

All events that are broadcast by JSF user interface components must extend the `javax.faces.event.FacesEvent` 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 communicate 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.

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).
- `ValueChangedEvent` -- Emitted by a `UIInput` component (or appropriate subclass) when a new local value has been created, and has passed all validations.

JSF.3.3.3 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 addition to the event handling methods, `FacesListener` defines a `getPhaseId()` method, which must return a phase identifier (of type `PhaseId`) describing the request processing lifecycle phase during which this listener is interested in receiving events. A special value (`PhaseId.ANY_PHASE`) is also available to indicate that the listener is interested in receiving events during any phase.

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, received on its `processAction()` method.
- `ValueChangedListener` -- a listener that is interested in receiving ValueChangedEvent events, received on its processValueChanged() method.

JSF.3.3.4 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 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 (so listener implementation classes must be `Serializable`).

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`.

JSF.3.3.5 Event Queueing

During the processing being performed by any phase of the request processing lifecycle, events may be created and queued by calling the `addFacesEvent()` method on the FacesContext instance for the current request. At the end of the phase, 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.

JSF.3.3.6 Event Broadcasting

As described in Section FIXME/2.3, at the end of each request processing lifecycle phase that may cause events to be queued, the JSF implementation will iterate over the queued events and call the `broadcast()` method to actually notify the registered listeners. The `broadcast()` method's responsibilities include:

- Based on the implementation class of the `FacesEvent` that is passed as a parameter, identify the list of listeners who have registere an interest in that event type (that is, if a `FooEvent` is passed, the `broadcast()` method must identify listeners registered on this component by calls to `addFooListener()`).
- For each interested listener, call the appropriate event processing method on that listener instance, passing the event that has occurred (cast to the appropriate concrete event type class).
- Determine whether there are any further listeners interested in this event (registered with an interest in a future phase, or registered for event processing in any phase). If there are any further interested listeners, return `true`; otherwise, return `false`.

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 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).

JSF.3.3.7 Application Events

[ED. NOTE -- This information in this section will change in the next public version of this specification. The following describes the current placeholder implementation for communicating form events to an application handler.]

All of the event processing described earlier in this section has related to *user interface* events, whose primary purpose is to modify (in some way) the state of the component tree and possibly causing the component tree to be rerendered. An additional mechanism is necessary in order to trigger higher level application business logic components, typically based on values from more than one component in the tree. In the typical HTML-based web browser scenario, for example, a form submit is an example of this use case. Such a case can be handed by extending the event processing model to include the idea of *application events*.

JSF provides an abstract base class for application events, `javax.faces.event.ApplicationEvent`, and a concrete subclass for form submit events, `FormEvent`. Form events may be queued to the application by calling the `addApplicationEvent()` method on the FacesContext instance for the current request. A standard `UICommand` component representing a submit button queues such an event, in addition to the `ActionEvent` described above, when appropriate.

Queued application events will be processed during the *Invoke Application* phase of the request processing lifecycle, as described in Section FIXME/2.2.5. Each queued event will be transmitted to the `ApplicationHandler` instance for this application, as described in Section FIXME/6.3.

JSF.3.4 Validation Model

JSF.3.4.1 Overview

JSF supports a mechanism for registering zero or more *validators* on each component in the component tree. A validator’s purpose is to perform correctness 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 correctness checking in a `validate()` method that is part of the component class.

JSF.3.4.2 Validator Classes

A validator must implement the `javax.faces.validator.Validator` interface, which contains a `validate()` method signature. A convenient base class (`ValidatorBase`) is also provided.. It is recommended that application-provided validators subclass `ValidatorBase`, rather than implement `Validator`, to provide protection against future evolution of the `Validator` interface.

General purpose validators may require configuration values in order to define the precise correctness 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 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 common validators, as described in Section FIXME/3.4.5 below

JSF.3.4.3 Validation Registration

The `UIComponent` interface includes an addValidator() method to register an additional validator for this component, and a `removeValidator()` method to remove an existing registration.

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 (so validator implementation classes must be `Serializable`).

JSF.3.4.4 Validation Processing

During the *Process Validations* phase of the request processing lifecycle (as described in Section FIXME/2.2.3), the JSF implementation will ensure that the `validate()` method of each registered `Validator`, as well as the `validate()` method of the component itself, is called for each component in the component tree. The responsibilities of each `validate()` method include:

- Perform the correctness check for which this validator was registered.
- If violation(s) of the correctness rules are found, set the `valid` property of this component to `false`, and add zero or more messages describing the problem by calling `addMessage()` on the `FacesContext` instance for the current request.

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.
- 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).

JSF.3.4.5 Standard Validator Implementations

JavaServer Faces defines a standard suite of `Validator` implementations that perform a variety of commonly required correctness 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 properties on the `Validator` implementation class.
- To support internationalization, `Message` instances should be created by passing a *message identifier* (along with optional substitution parameters) to an appropriate `MessageResources` instance. The message identifiers for such standard messages are also defined by manifest String constants in the implementation classes.

- 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 separate instance of `RequiredValidator` should be registered in order to enforce this rule.

Localizable message template strings may contain placeholder elements, as defined in the API JavaDocs for the `java.text.MessageFormat` class. Several of the `getMessage()` method signatures on the `MessageResources` class allow replacement value(s) for these placeholders to be included. The definitions of the standard message identifiers, below, will note cases where replacement value(s) will be included in the message addition.

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

- `DoubleRangeValidator` -- Checks the local value of a component, which must be of any numeric type (or whose `String` value is convertible to `double`), against specified maximum and/or minimum values.
- `LengthValidator` -- Checks the length (i.e. number of characters) of the local value of a component, which must be of type `String` (or convertible to a `String`), against maximum and/or minimum values.
- `LongRangeValidator` -- Checks the local value of a component, which must be of any numeric time convertible to `long` (or whose `String` value is convertible to `long`), against maximum and/or minimum values.
- `RequiredValidator` --Checks the local value of a component for being non-null. If the local value is a zero-length String, it will pass the check enforced by this `Validator`; use a separate `LengthValidator` to enforce a requirement for a minimum String length greater than zero. [FIXME - the definition of this validator's behavior is being reviewed.]
- `StringRangeValidator` -- Checks the local value of a component, which must be of type `String` (or convertible to a `String`), against specified maximum and/or minimum values.

JSF.4

Standard User Interface Components

In addition to the abstract base class `UIComponent`, 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 `UIComponent` base class). It is anticipated that the number of standard component classes will grow in future versions of the JavaServer Faces specification. All of the concrete classes are part of the `javax.faces.component` package, and (unless otherwise noted) are subclasses of `javax.faces.component.UIComponentBase`.

Each of these classes defines the render-independent characteristics of the corresponding component as JavaBeans properties. The class descriptions also specify minimal implementations of the `decode()` (where appropriate) and `encodeXxx()` methods, which implement very basic decoding and encoding that is compatible with HTML/4.01. It is assumed, however, that rendering will normally be delegated to `Renderer` instances acquired from the `RenderKit` associated with our component tree.

The standard `UIComponent` subclasses also define a symbolic String constant named TYPE, which specifies the value that will be returned by `getComponentType()` for this class, which will be equal to the fully qualified class name of the standard component implementation class.

Component types should be globally unique across JSF implementations. The suggested convention is to use a "package name" syntax similar to that used for Java packages, although the types do not need to correspond to class names. Component type values starting with "javax.faces.*" are reserved for use by standard component types defined in this Specification.

JSF.4.1 UICommand

The `UICommand` class represents a user interface component 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.

- `public String getCommandName();`
- `public void setCommandName(String commandName);`

The `commandName` property getter and setter methods are typesafe aliases for getValue() and setValue(). This allows the command name to be retrieved (via a call to currentValue()) from the locally configured value, or indirectly via the model reference expression.

The `commandName` property identifies the command or action that should be executed when this command is activated. Command names need not be unique across a single component tree; it is common to provide users more than one way to request the same command. They must be comprised of characters that are legal in a URL.

The default rendering functionality for `UICommand` components is:

- `decode()` - If there is a request parameter on the incoming request that matches the client identifier of this component, with a value of "submit", enqueue a `FormEvent` to the application, passing the form name of the form being submitted and the command name of this `UICommand`.
- `encodeBegin()` - Render the beginning of an HTML submit button, using the client identifier of this component as the "name" attribute, and "submit" as the value to be returned to the server if this button is selected by the user.
- `encodeEnd()` - Render the ending of an HTML submit button.

`UICommand` is a source of `ActionEvent` events, which are emitted when the corresponding user interface control is activated. It includes methods to register and deregister `ActionListener` instances interested in those events. See Section FIXME/3.3 for more details on the event and listener model provided by JSF.

JSF.4.2 UIForm

The `UIForm` class represents a user interface component that corresponds to an input form to be presented to the user, and whose child components represent (among other things) the input fields to be included when the form is submitted.

- `public String getFormName();`

- `public void setFormName(String formName);`

The `formName` property getter and setter methods are typesafe aliases for getValue() and setValue(). This allows the form name to be retrieved (via a call to currentValue()) from the locally configured value, or indirectly via the model reference expression.

The `formName` property identifies the form that is being submitted, and will generally be used by the application (during the *Invoke Application* phase of the request processing lifecycle) to dispatch to the corresponding application logic for processing this form.. They must be comprised of characters that are legal in a URL.

The default rendering functionality for `UIForm` components is:

- `encodeBegin()` - Render an HTML <form> element, with an `action` attribute calculated as "/xxxxx/faces/yyyyy", where "xxxxx" is the context path of the current web application, "yyyyy" is the response tree identifier of the page being rendered. URL rewriting will have been applied, to maintain session identity in the absence of cookies.
- `encodeEnd()` - Render the </form> element required to close the <form> element created during `encodeBegin()`.

JSF.4.3 UIGraphic

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

- `public String getURL();`
- `public void setURL(String url);`

The `url` property getter and setter methods are typesafe aliases for getValue() and setValue(). This allows the URL of the image to be retrieved (via a call to currentValue()) from the locally configured value, or indirectly via the model reference expression.

The `url` property contains a URL for the image to be displayed by this component. If the value begins with a slash ('/') character, it is assumed to be relative to the context path of the current web application. Otherwise, the URL is used without modification.

The default rendering functionality for `UIGraphic` components is:

- `encodeEnd()` - Render an HTML <img> element with a `src` element based on the local value of this component. URL rewriting will have been applied, to maintain session identity in the absence of cookies.

JSF.4.4 UIInput

The `UIInput` class represents a user interface component that both displays the current value of the component to the user (as `UIOutput` components do), also cause the inclusion of updated values on the subsequent request, which will be decoded during the *Apply Request Values* phase of the request processing lifecycle. There are no restrictions on the data type of the local value, or the object referenced by the model reference expression (if any); however, individual `Renderer`s will generally impose restrictions on the types of data they know how to display.

The default rendering functionality for `UIInput` components is:

- `decode()` - Set the local value to the value of the corresponding request parameter, or null if the corresponding request parameter is not present.
- `encode()` - Retrieve the current value of this component, convert it to a String (if necessary), and render an HTML <input type=”text”> element.

`UIInput` is a source of `ValueChangedEvent` events, which are emitted when the a new value has been established for an input component, and all validations have been passed successfully. (Subclasses of `UIInput` also inherit this behavior.) It includes methods to register and deregister `ValueChangedListener` instances interested in those events. See Section FIXME/3.3 for more details on the event and listener model provided by JSF.

JSF.4.5 UIOutput

The `UIOutput` class represents a user interface component that displays the unmodified current value of this component to the user. The user cannot manipulate this component; it is for display purposes only. There are no restrictions on the data type of the local value, or the object referenced by the model reference expression (if any); however, individual `Renderer`s will generally impose restrictions on the types of data they know how to display.

The default rendering functionality for `UIOutput` components is:

- `encodeEnd()` - Retrieve the current value of this component, convert it to a String (if necessary), and render directly to the response.

Because `UIOutput` components are write only, they must override the `updateModel()` method inherited from UIComponent and perform no model update processing.

JSF.4.6 UIPanel

The `UIPanel` class (extends `UIOutput`) represents a user interface component that is primarily a container for its children. The default implementation sets its `rendersChildren` property to `true`, but has no rendering behavior of its own.

Subclasses of `UIPanel` will typically implement layout management for child components, either directly (via its `encodeXxx()` methods) or indirectly (via implementation in an appropriate `Renderer`). The latter scenario -- delegating rendering to a `Renderer` -- means that `UIPanel` can be used as a component implementation class for most layout management purposes, without the need to subclass.

JSF.4.7 UIParameter

The `UIParameter` class (extends `UIOuptut`) represents a user interface component that has no visible rendering behavior. Instead, it provides a convenient mechanism to provide configuration parameter values to a parent component (such as substitution parameters for an internationalized message being processed with java.text.MessageFormat, or adding request parameters to a generated hyperlink).

Parent components should retrieve the value of a parameter component by calling `currentValue()`. In this way, the parameter value can be set directly on the component (via `setValue()`) or retrieved indirectly via the model reference expression.

- `public String getName();`
- `public void setName(String name);`

In some scenarios, it is necessary to provide a parameter name, in addition to the parameter value that is accessible via the `currentValue()` method. Renderers that support or require parameter names on their nested `UIParameter` child components should document their use of this property.

This component has no default rendering functionality.

JSF.4.8 UISelectBoolean

The `UISelectBoolean` class (extends `UIInput`) represents a user interface component which can have a single, boolean value of `true` or `false`. It is most commonly rendered as a checkbox.

- `public boolean isSelected();`
- `public void setSelected(boolean selected);`

The `selected` property getter and setter methods are typesafe aliases for getValue() and setValue(). This allows the selected state of the component to be retrieved (via a call to currentValue()) from the locally configured value, or indirectly via the model reference expression.

The default rendering functionality for `UISelectBoolean` components is:

- `decode()` - Set the local value to `true` or `false`, based on the corresponding request parameter included with this request (if any).
- `encodeBegin()` - Render an HTML <input type=”checkbox”> element.

As a subclass of `UIInput`, `UISelectBoolean` inherits the capability to broadcast ValueChangedEvent events, as described in Section FIXME/3.3.

JSF.4.9 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.

- `SelectItem(Object value, String label, String description);`
- `public String getDescription();`
- `public String getLabel();`
- `public Object getValue();`

Each `SelectItem` has three immutable properties: `value` is the value that will be returned (as a request parameter) if the user selects this item, `label` is the visible content that enables the user to determine the meaning of this item, and `description` is a description of this item that may be used within development tools, but is not rendered as part of the response.

JSF.4.10 UISelectItem

The `UISelectItem` class represents a single `SelectItem` that will be included in the list of available options in a `UISelectMany` or `UISelectOne` component that is the direct parent of this component. This component has no decoding or encoding behavior of its own -- its purpose is simply to configure the behavior of the parent component.

- `public String getItemDescription();`
- `public void setItemDescription(String itemDescription);`
- `public String getItemLabel();`
- `public void setItemLabel(String itemLabel);`
- `public String getItemValue();`
- `public void setItemValue(String itemValue);`

These attributes are used to configure an instance of `SelectItem` that will be added to the set of available options for our parent UISelectMany or UISelectOne tag.

JSF.4.11 UISelectItems

The `UISelectItem` class represents a set of `SelectItem` instances that will be included in the list of available options in a `UISelectMany` or `UISelectOne` component that is the direct parent of this component. This component has no decoding or encoding behavior of its own -- its purpose is simply to configure the behavior of the parent component.

When assembling the list of available options, our parent UISelectMany or UISelectOne tag must use the currentValue() method to acquire the current value of this component. This value must be of one of the following types, which causes the specified behavior:

- Single instance of SelectItem -- This instance is added to the set of available options.
- Array of SelectItem -- Each instance in this array is added to the set of available options, in ascending order.
- Collection of SelectItem[1] -- Each element in this Collection will be added to the set of available options, in the order that an iterator over the Collection returns them.

JSF.4.12 UISelectMany

The `UISelectMany` class (extends `UIInput`) represents a user interface component that represents the user's choice of a zero or more items from among a discrete set of available items. It is typically rendered as a combo box or a set of checkboxes.

- `public Object[] getSelectedValues();`
- `public void setSelectedValues(Object selectedValues[]);`

The `selectedValue`s property getter and setter methods are typesafe aliases for getValue() and setValue(). This allows the values of the currently selected items (if any) to be retrieved (via a call to currentValue()) from the locally configured value, or indirectly via the model reference expression.

The `selectedValues` property contains the *values* of the currently selected items (if any). If there is no current value, or none of the specified values match the value of any item on the items list, the component may be rendered with no item currently selected.

The list of available items for this component is specified by nesting zero or more child components of type UISelectItem (see section FIXME) or UISelectItems (see Section FIXME) inside this component.

The default rendering functionality for `UISelectMany` components is:

- `decode()` - Set the local value to a String array containing the values of the corresponding request parameter, or `null` if the corresponding request parameter is not present.
- `encodeEnd()` - Retrieve the current value of this component, convert it to a String (if necessary), and render an HTML <select> element. As the available items are rendered, any item whose value matches one of the Strings in the current value of this component will cause this item to be preselected.

As a subclass of `UIInput`, `UISelectMany` inherits the capability to broadcast ValueChangedEvent events, as described in Section FIXME/3.3.

1. This includes any implementation of java.util.List and java.util.Set.

JSF.4.13 UISelectOne

The `UISelectOne` class (extends `UIInput`) represents a user interface component that represents the user's choice of a single item from among a discrete set of available items. It is typically rendered as a combo box or a set of radio buttons.

- `public Object getSelectedValue();`
- `public void setSelectedValue(Object selectedValue);`

The `selectedValue` property getter and setter methods are typesafe aliases for getValue() and setValue(). This allows the value of the currently selected item (if any) to be retrieved (via a call to currentValue()) from the locally configured value, or indirectly via the model reference expression.

The `selectedValue` property contains the *value* of the currently selected item (if any). If there is no current value, or the specified value does not match the value of any item on the items list, the component may be rendered with no item currently selected.

The list of available items for this component is specified by nesting zero or more child components of type UISelectItem (see section FIXME) or UISelectItems (see Section FIXME) inside this component.

The default rendering functionality for `UISelectOne` components is:

- `decode()` - Set the local value to the value of the corresponding request parameter, or `null` if the corresponding request parameter is not present.
- `encodeEnd()` - Retrieve the current value of this component, convert it to a String (if necessary), and render an HTML <select> element. As the available items are rendered, any item whose value matches the current value of this component will cause this item to be preselected.

As a subclass of `UIInput`, `UISelectOne` inherits the capability to broadcast ValueChangedEvent events, as described in Section FIXME/3.3.

JSF.5

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 enapsulate this contextual information.

JSF.5.1 FacesContext

JSF defines the `javax.faces.context.FacesContext` abstract base class for representing all of the contextual information associated with a 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 FIXME, below. 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.

JSF.5.1.1 Servlet API Components

[ED. NOTE -- Direct access to servlet API objects will likely be abstracted away in a future version of this specification.]

- `public HttpSession getHttpSession();`
- `public ServletContext getServletContext();`
- `public ServletRequest getServletRequest();`

- `public ServletResponse getServletResponse();`

The `FacesContext` instance provides immediate access to all of the components defined by the servlet container within which a JSF-based web application is deployed. The `getHttpSession()` method will only return a non-null session instance if the current request is actually an `HttpServletRequest`, and a session is already in existence. To create and acquire a reference to a new session, call the standard `HttpServletRequest.getSession()` or `HttpServletRequest.getSession(true)` method.

JSF.5.1.2 Locale

- `public Locale getLocale();`
- `public void setLocale(Locale locale);`

When the `FacesContext` for this request is created, the JSF implementation will call `setLocale()` to record the locale of the current user (either stored in the user's session, if any, or the one returned by calling `getLocale()`on the current servlet request). This Locale may be used to support localized decoding and encoding operations. At any time during the request processing lifecycle, `setLocale()` be called to modify the localization behavior for the remainder of the current request.

JSF.5.1.3 Component Tree

- `public Tree getTree();`
- `public void setTree(Tree tree);`

During the *Reconstitute Component Tree* phase of the request processing lifecycle, the JSF implementation will identify the component tree (if any) to be used during the inbound processing phases of the lifecycle, and call `setTree()` to establish it.

The `setTree()` method can also be called by the application (during the *Invoke Application* phase of the request processing lifecycle), to change the component tree that will be rendered (during the *Render Response* phase) to a new one. If the application does not do this, the component tree originally created for this request (as modified by the processing during previous phases of the request processing lifecycle) is the one that will get rendered.

JSF.5.1.4 Application Events Queue

[ED. NOTE -- The mechanism for communicating events to the application will change in a future version of this specification. The following describes the current placeholder implementation.]

- `public Iterator getApplicationEvents();`
- `public int getApplicationEventsCount();`

- `public void addApplicationEvent(FacesEvent event);`

During the inbound phases of the request processing lifecycle (but normally during the *Apply Request Values* phase), user interface components or event handlers can queue events (via a call to the `addApplicationEvent()` method) to be handled by the web application. During the *Invoke Application* phase, the JSF implementation will call `getApplicationEvents()` and dispatch each event to the application, via the `ApplicationHandler` instance associated with the `Lifecycle` instance that is processing this request.

JSF.5.1.5 Message Queues

- `public void addMessage(UIComponent component, Message 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 `component` is `null`), or related to a specific component.

- `public int getMaximumSeverity();`
- `public Iterator getMessages(UIComponent component);`
- `public Iterator getMessages();`

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 `UIComponent`. The `getMessages(UIComponent)` methods return an `Iterator` over queued `Messages`, either those associated with the specified `UIComponent`, or those associated with no `UIComponent` if the parameter is `null`. The `getMessages()` method returns an `Iterator` over all queued Messages, whether or not they are associated with a particular `UIComponent`.

For more information about the `Message` class, see Sections FIXME.

JSF.5.1.6 Lifecycle Management Objects

- `public ApplicationHandler getApplicationHandler();`
- `public ViewHandler getViewHandler();`

Return the application handler and view handler instances, respectively, that will be utilized during the *Invoke Application* and *Render Response* phases of the request processing lifecycle, respectively.

JSF.5.1.7 Model Reference Expression Evaluation

- `public Class getModelType(String modelReference);`

- `public Object getModelValue(String modelReference);`
- `public void setModelValue(String modelReference, Object value);`

FIXME - description of how model reference expression evaluation actually works.

JSF.5.1.8 FacesEvents Queue

- `public Iterator getFacesEvents();`
- `public void addFacesEvent(FacesEvent event);`

During the inbound phases of the request processing lifecycle up to and including the *Update Model Values* phase, user interface components can queue events (via a call to the `addFacesEvent()` method) to be handled by event listeners registered on the source component, at the end of each request processing phase. The JSF implementation will call `getFacesEvent()` to process the events that have been queued in that phase, as described in section FIXME/2.x.

JSF.5.1.9 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.

[FIXME - clarify when setResponseStream/setResponseWriter are called, and by whom]

JSF.5.2 Message

Each message queued within a `FacesContext` is an instance of the `javax.faces.context.Message` abstract class. The following method signatures are supported to retrieve the properties of the completed message:

- `public String getDetail();`
- `public int getSeverity();`
- `public String getSummary();`

The message properties are defined as follows:

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

JSF.5.3 MessageImpl

`MessageImpl` is a concrete implementation of Message that can serve as a convenient base class for messages provided by JSF implementations, component libraries, or by applications. It offers the following additional signatures above those defined by `Message`.

- `public MessageImpl();`
- `public MessageImpl(int severity, String summary, String detail);`

These constructors support the creation of uninitialized and initialized `Message` instances, respectively.

- `public void setDetail(String detail);`
- `public void setSeverity(int severity);`
- `public void setSummary(String summary);`

These property setters support modification of the properties of the `MessageImpl` instance.

JSF.5.4 ResponseStream

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

JSF.5.5 ResponseWriter

ResponseWriter is an abstract class representing a character output stream for the current response. 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.

[FIXME - The existence of the following methods, here rather than on a separate wrapper class, needs to be reviewed. It causes complexity in the concrete implementations JspResponseWriter and ServletResponseWriter.]

- `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) 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()`, or `endDocument()`.

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

- `public void writeText(Object text) throws IOException;`
- `public void writeText(char text) throws IOException;`
- `public void writeText(char text[]) 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. Any currently open element created by a call to `startElement()` is closed first.

JSF.5.6 MessageResources

The abstract class `javax.faces.context.MessageResources` represents a mechanism by which localized messages associated with a unique (per `MessageResources` instance) message identifier. JSF implementations make multiple `MessageResources` instances available via a `MessageResourcesFactory` (see the next section for details).

- `public Message getMessage(FacesContext context, String messageId);`
- `public Message getMessage(FacesContext context, String messageId, Object params[]);`

Return a localized[1] `Message` instance for the specified message identifier, optionally modified by substitution parameters[2] in the second method signature. If the specified message identifier is not recognized by this `MessageResources` instance, these methods return `null` instead of a `Message` instance.

1. Localization is performed relative to the locale property of the specified FacesContext.

2. Tyipcal MessageResources implementations support message template strings that can be passed to instances of java.text.MessageFormat, with placeholders like "{0}" for inserting substitution parameters.

- `public Message getMessage(FacesContext context, String messageId, Object param0);`
- `public Message getMessage(FacesContext context, String messageId, Object param0, Object param1);`
- `public Message getMessage(FacesContext context, String messageId, Object param0, Object param1, Object param2);`
- `public Message getMessage(FacesContext context, String messageId, Object param0, Object param1, Object param2, Object param3);`

Convenience for creating messages based on one, two, three, or four substitution parameters.

JSF.5.7 MessageResourcesFactory

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

```
MessageResourcesFactory factory = (MessageResourcesFactory)
      FactoryFinder.getFactory
      (FactoryFinder.MESSAGE_RESOURCES_FACTORY);
```

The MessageResourcesFactory implementation class supports the following methods:

- `public MessageResources getMessageResources(String messageResourcesId);`

This method creates (if necessary) and returns a `MessageResources` instance. All requests for the same message resources identifier, from within the same web application, will return the same `MessageResources` instance, which must be programmed in a thread-safe manner.

Every JSF implementation must provide two MessageResources instances, associated with the message resources identifiers named by the following String constants:

- **`MessageResourcesFactory.FACES_API_MESSAGES`** - Identifier for a `MessageResources` instance defining the message identifiers used in `javax.faces.*` classes defined in this specification (such as the predefined `Validator` implementations).
- **`MessageResourcesFactory.FACES_IMPL_MESSAGES`** - Identifier for a `MessageResources` instance defining the message identifiers used in the classes comprising a JSF implementation.

Additional `MessageResources` instances can be registered at any time (see below).

- `public Iterator getMessageResourcesIds();`

Return an `Iterator` over the message resource identifiers for all `MessageResources` implementations available via this MessageResourcesFactory. This Iterator must include the standard message resources identifiers described earlier in this section.

- `public void addMessageResources(String messageResourcesId, MessageResources messageResources);`

Register an additional `MessageResources` instance under the specified message resources identifier. This method may be called at any time by JSF implementations, JSF-based applications, and third party component libraries utilized by an application. Once registered, the `MessageResources` instance is available for the remainder of the lifetime of this web application.

JSF.5.8 FacesContextFactory

A single instance of `javax.faces.context.FacesContextFactory` must be made available to each JSF-based web application running in a servlet container. This class is primarily of use by JSF imlementors-- 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(ServletContext context, ServletRequest request, ServletResponse response, Lifecycle lifecycle);`

Create (if necessary) and return a FacesContext instance that has been configured based on the specified parameters.

JSF.5.9 Tree

As mentioned in section FIXME/3.1.3, above, JavaServer Faces supports the ability to combine multiple `UIComponent` instances into a tree structure, with a single component as the root node. Beyond this, JSF also supports the ability to create component trees dynamically (using the `TreeFactory` API described in the following section), with preconfigured components and other attributes. These trees are represented by instances of the abstract class `javax.faces.tree.Tree` with the following method signatures:

- `public String getRenderKitId();`

- `public void setRenderKitId(String renderKitId);`

The `RenderKit` instance associated with a `Tree` is used as a factory for `Renderer` instances for components that are configured with a non-null `rendererType` property for delegated implemtation of decode and encode operations. Component trees will often be constructed with a default `RenderKit` provided by the JSF implementation. In addition, JSF-based applications can choose the `RenderKit` that will be used to create a particular response, so that the same components can be rendered in different ways for different clients.

- `public UIComponent getRoot();`

This method returns the root node of the component tree associated with this `Tree` instance.

- `public String getTreeId();`

This method returns the *tree identifier* of the component tree associated with this `Tree` instance. This identifier is utilized in two different phases of the request processing lifecycle:

- During the *Reconstitute Component Tree* phase, an appropriate tree identifier is extracted from the incoming request as part of the state information being created from the incoming request data. This tree will later be updated by subsequent phases of request processing.
- During the *Invoke Application Handler* phase, JSF-based applications may create a tree that corresponds to the output page to be created by this response[1]. This is performed by selecting an appropriate tree identifier, and calling the `getTree()` method of the `TreeFactory` instance for this web application.

Tree identifiers must be composed of characters that are legal in URLs, optionally including slash (‘/’) characters.

JSF.5.10 TreeFactory

[FIXME - The functionality described below is not required if we are dispensing with the idea of external metadata for a component tree. The remaining need is a way to portably instantiate `Tree` instances with a given tree identifier and `RenderKit`.]

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

```
TreeFactory factory = (TreeFactory)
   FactoryFinder.getFactory(FactoryFinder.TREE_FACTORY);
```

The `TreeFactory` implementation class provides the following method:

1. Unless the application does this, the response will be based on the same tree as the one manipulated by previous request processing lifecycle phases.

- `public Tree getTree(FacesContext context, String treeId) throws FacesException;`

This method instantiates a `Tree` instance, preconfigured with a default `renderKitId` provided by the JSF implementation.

JSF.6

Lifecycle Management

In Chapter FIXME/2, 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.

JSF.6.1 Lifecycle

Upon receipt of each JSF-destined request to this web application, the JSF implementation must acquire a `Lifecycle` instance with which to manage the request processing lifecycle, as described in Section FIXME/6.5. The `Lifecycle` instance acts as a state machine, and invokes appropriate `Phase` instances to implement the required functionality for each phase, as described in Chapter FIXME/2.

- `public void execute(FacesContext context) throws FacesException;`
- `public int executePhase(FacesContext context, Phase phase) throws FacesException;`

[ED. NOTE -- The executePhase() method will likely be removed in a future version of this specification.]

The JSF implementation must call the `execute()` method to perform the entire request processing lifecycle [FIXME - ongoing discussion about whether the JSF implementation can use the adapter pattern and bypass this], once it has acquired an appropriate `FacesContext` for this request as described in Chapter FIXME/2. This method must execute all `Phase` instances registered for this `Lifecycle` instance, in the order described in Chapter FIXME/2. Execution of each phase must be accomplished by a call to the `executePhase()` method, which returns a *state change indicator* value describing the action that should be taken next.

- `public ApplicationHandler getApplicationHandler();`
- `public void setApplicationHandler(ApplicationHandler handler);`

[ED. NOTE -- The mechanisms for passing events to application components will be revised in a future version of this specification.]

In order to process application events during the *Invoke Application* phase of the request processing lifecycle, the application must register an instance of a class that implements the `ApplicationHandler` interface, as described in Section FIXME/6.4, below.

Unless overridden (by a call to setApplicationHandler() before the first request is processed), the JSF implementation must provide a default ApplicationHandler implementation that throws an `IllegalStateException` indicating that no application handler has been configured.

- `public ViewHandler getViewHandler();`
- `public void setViewHandler(ViewHandler handler);`

A `ViewHandler` (see Section FIXME, below) is a pluggable mechanism for performing the required processing to transmit the component tree to the `ServletResponse` that is associated with the `FacesContext` for a request, during the *Render Response* phase of the request processing lifecycle.

Unless overridden (by a call to `setViewHandler()` before the first request is processed), the JSF implemenation must provide a default `ViewHandler` implementation that converts the `treeId` property of the component tree into a context-relative resource path, and performs a `RequestDispatcher.forward()` to the corresponding resource.

JSF.6.2 Phase

[ED. NOTE -- This class is likely to be removed from a future version of this specification, because there is no need for a JSF-based application to utilize it.]

The standard processing performed by each phase of the request processing lifecycle, as described in Chapter FIXME/2, must be implemented in a subclass of `Phase` that is provided by the JSF implementation.

- `public int execute(FacesContext context) throws FacesException;`

Perform the functionality required by the specification of the current phase of the request processing lifecycle, and return a *state change indicator* that indicates whether and where lifecycle processing should continue. The following symbolic constants define the valid return values, and their corresonding meanings:

- `Phase.GOTO_EXIT` - Indicates that the entire request processing lifecycle has been completed, and no further `Phase` instances should be invoked. This value is normally returned only from the `Phase` instance that implements the *Render Response* phase; however, it must be returned from **any** `Phase` instance that completes the creation of the servlet response for the current request (for example, by performing an HTTP redirect).
- `Phase.GOTO_NEXT` - Indicates that processing should proceed with the next registered `Phase` instance in the normal sequence. This is the typical value, and should be returned in all cases other than when returning `GOTO_EXIT` or `GOTO_RENDER` is appropriate.
- `Phase.GOTO_RENDER` - Indicates that processing should immediately proceed to the *Render Response* phase, skipping any intervening Phase instances. For example, this return value will be used when the *Process Validations* phase has detected one or more validation errors, and wishes to bypass the *Update Model Values* and *Invoke Application* phases in order to redisplay the current page.

JSF.6.3 ApplicationHandler

[ED. NOTE -- The behavior of this API will be revised in a future version of this specification. The current mechanism is a placeholder until a more sophisticated interface design is completed.]

A JSF-based application must register a class that implements the `ApplicationHandler` interface in order to process application events during the *Invoke Application* phase of the request processing lifecycle. This is typically done at application startup time, by utilizing the `LifecycleFactory` described in Section FIXME to create the `Lifecycle` instance that will be utilized, and then calling its `setApplicationHandler()` method.

- `public boolean processEvent(FacesContext context, FacesEvent event);`

During the *Invoke Application* phase of the request processing lifecycle, the JSF implementation will call the `processEvent()` method on the registered `ApplicationHandler` instance for each `FacesEvent` that has been queued in the `FacesContext`, until either an event handler returns `true` (indicating a desire to proceed immediately to the *Render Response* phase of the request processing lifecycle), or until all queued application events have been processed.

FIXME - do we want to use reflection to dispatch to a particular method name based on the command or form name?

During application event processing, the application may perform functional actions required by the applcation's logic. In particular, it may decide that it wishes to display a response page other than the one that corresponds to the component tree, and perform the following steps to switch to a different page for the response:

- Call the `getTree()` method of the `TreeFactory` instance for this web application, passing the tree identifier corresponding to the new output page.
- Optionally, modify the default attributes and properties of the components in the returned `Tree`, in order to customize the response page that is about to be rendered
- Call the `setTree()` method of the `FacesContext` instance for the request currently being processed.

JSF.6.4 ViewHandler

A JSF implementation must register an implementation of the ViewHandler interface for use during the *Render Response* phase of the request processing lifecycle. Prior to the first request being processed by a `Lifecycle` instance, an alternative ViewHandler implementation may be registered by a call to the `setViewHandler()` method.

- `public void renderView(FacesContext context) throws IOException, ServletException;`

Perform whatever actions are required to render the component tree to the `ServletResponse` associated with the specified `FacesContext`.

JSF.6.5 LifecycleFactory

A single instance of `javax.faces.lifecycle.LifecycleFactory` must be made available to each JSF-based web application running in a servlet container. This class is primarily of use to tools providers -- applications will not generally call it directly. 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 `getLifecycle()` 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`.

JSF.7

Rendering Model

As discussed in earlier chapters, JSF 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.

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 RenderKits 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`.

JSF.7.1 RenderKit

A `RenderKit` instance is optionally associated with a component tree, and supports 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.DEFAULT_RENDER_KIT` as described below) that is utilized if no other `RenderKit` is selected.

- `public Iterator getComponentClasses();`

Return an `Iterator` over the set of `UIComponent` subclasses known to be supported by the `Renderer`s registered with this `RenderKit`. The set of classes returned by this method are not guaranteed to be the complete set of component classes supported (i.e. component classes for which one of the included `Renderer`s would return `true` from a call to a `suppportsComponentType()` method). However, `RenderKit` instances are encouraged to explicitly enumerate the component classes that are associated with its `Renderer`s, to assist development tools in providing more information in their user interfaces.

- `public Renderer getRenderer(String rendererType);`

Create (if necessary) and return a `Renderer` instance corresponding to the specified `rendererType`, which will typically be the value of the rendererType property of a `UIComponent` about to be decoded or encoded.

- `public Iterator getRendererTypes(UIComponent component);`
- `public Iterator getRendererTypes(String componentType);`

Return an `Iterator` over the set of renderer types for reigistered `Renderer` instances (if any) that are managed by this `RenderKit` instance and know how to support components of the specified component class or type.

- `public Iterator getRendererTypes();`

Return an `Iterator` over all of the renderer types for the `Renderer`s registered with this `RenderKit` instance.

- `public void addComponentClass(Class componentClass);`
- `public void addRenderer(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 FIXME/7.3), or integrate additional supported component classes and/or Renderer instances into an existing RenderKit instance. For example, it will be common to for an application that requires custom component classes and `Renderer`s to register them with the standard `RenderKit` provided by the JSF implementation, at application startup time.

JSF.7.2 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 boolean decode(FacesContext context, UIComponent component) throws IOException;`

For components utilizing the *delegated implementation* programming model, this method will be called during the *Apply Request Values* phase of the request processing lifecycle. It has the following responsibilities:

- Extract from the incoming request (typically from parameters, headers, and/or cookies) the data representing the new value for this component.
- Attempt to convert this data into an object of the appropriate type for this component.
- If conversion is successful, save the converted object as the local value of this component, and return `true` to the caller.
- If conversion is unsuccessful, save enough state information for the corresponding `encodeXxx()` methods to reproduce the incorrect value, enqueue one or more error messages by calling the `addMessage()` method of the `FacesContext` instance for this request, and return `false` to the caller.
- Optionally, enqueue one or more `FacesEvent` events, on either the current component or any other component in the component tree, by calling the `addFacesEvent()` method on the `FacesContext` instance for this request. These events will be processed at the end of each relevant phase of the request processing lifecycle, as described in Section FIXME/2.3.
- Optionally, enqueue one or more `CommandEvent` or `FormEvent` events, by calling the `addApplicationEvent()` method on the `FacesContext` instance for the current request. These events will be processed by the application, during the *Invoke Application* phase of the request processing lifecycle.
- `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;`

As noted earlier in Section FIXME, components can be organized into component trees with children. Some component implementations (such as a complex table control) will wish to take responsibility for encoding all of their child components, as well as themselves. Other components (such as one that represents an HTML form) will want to allow child components to render themselves[1]. The preference of a particular component class to render its children or not is indicated by the `rendersChildren` property of that component.

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

1. This approach is also useful in rendering scenarios such as JSP pages, where embedded HTML markup in between the tags representing JSF components is used to manage the layout of the page.

encodeBegin(), encodeChildren(), and encodeEnd() methods of UIComponent (described in Section FIXME/3.1.11) when the component implements the *direct implementation* programming model.

- public boolean supportsComponentType(UIComponent component);
- public boolean supportsComponentType(String componentType);

These methods return true if the specified component class or type is supported by this Renderer instance. For supported components, the getAttributeNames() and getAttributeDescriptor() methods can be utilized to acquire metadata information useful in configuring the attributes of these components, when it is known that this particular Renderer instance will be utilized.

- public Iterator getAttributeNames(UIComponent component);
- public Iterator getAttributeNames(String componentType);

Return an Iterator over the attribute names supported by this Renderer for the specified component class or type. These methods are useful to tool providers in building user interfaces to configure the properties and attributes of a particular component, when it is known that this particular Renderer will be utilized.

- public AttributeDescriptor getAttributeDescriptor(UIComponent component, String name);
- public AttributeDescriptor getAttributeDescriptor(String componentType, String name);

Return an AttributeDescriptor for the specified attribute name, as supported for the specified component class or type. These methods are useful to tool providers in building user interfaces to configure the properties and attributes of a particular component, when it is known that this particular Renderer will be utilized.

JSF.7.3 RenderKitFactory

A single instance of javax.faces.render.RenderKitFactory must be made available to each JSF-based web application running in a servlet 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(String renderKitId);
- public RenderKit getRenderKit(String renderKitId, FacesContext context);

This method creates (if necessary) and returns a `RenderKit` instance. All requests for the same render kit identifier will return the same `RenderKit` instance, which must be programmed in a thread-safe manner.

Every JSF implementation must provide a `RenderKit` instance for a default render kit identifier that is designated by the String constant `RenderKitFactory.DEFAULT_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.DEFAULT_RENDER_KIT`.

- `public void addRenderKit(String renderKitId, RenderKit renderKit);`

At any time, additional `RenderKit` instances, and their corresponding identifiers, can be registered by the JSF implementation, by included component libraries, or by applications themselves. Once a `RenderKit` instance has been registered, it may be associated with a component tree by calling the `setRenderKit()` method of the corresponding `Tree` instance.

JSF.7.4 Converter

[ED. NOTE -- Will be moved into Chapter 3 in a future version of this specification].

A `javax.faces.convert.Converter` is a specialized class that can perform String-to-Object conversions (during the *Apply Request Values* phase of the request processing lifecycle) and Object-to-String conversions (during the *Render Response* phase of the request processing lifecycle). This allows `Renderer` instances that know how to deal with the representation of model data objects as a single String (a very common use case) to perform conversions to and from arbitrary (often application-specific) model data types, wiithout requiring the creation of specialized `Renderer`s for each data type.

Converter instances are identified by a web-application-wide unique *converter identifier*, which is registered (along with a corresponding `Converter` instance) in the `ConverterFactory` instance for this web application (see Section FIXME/7.5 for more information). A single `Converter` instance is associated with each converter identifier, so `Converter`s must be programmed in a thread-safe manner.

Typically, a `Converter` will be a stateless 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 may be provided via named attributes of the `UIComponent` instance that is passed to the `Converter`'s methods. The attribute names used by a `Converter` to configure its behavior should be documented in the usage instructions for that `Converter`, and should be selected to not conflict with other render-independent and render-dependent attributes of the underlying component.

`Converter` instances 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 this converted String as the value parameter, and retrieve a model data object that is semantically equivalent to the original one.

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

This method will be called during the *Apply Request Values* phase of the request processing lifecycle, by `Renderer` instances that have been configured to utilize `Converter`s. It should convert the specified String value into an appropriate instance of an Object of the underlying Java class for which this `Converter` exists. If conversion is not successful, a `ConverterException` should be thrown, with a message string that has been localized based on the `Locale` associated with the specified `FacesContext` instance.

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

This method will be called during the *Render Response* phase of the request processing lifecycle, by `Renderer` instances that have been configured to utilize `Converter`s. It should convert the model data object into a String value that is suitable for rendering by one of the `encodeXxx()` methods of the calling `Renderer`.

JSF.7.5 ConverterFactory

[ED. NOTE -- Will be moved into Chapter 3 in a future version of this specification].

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

```
ConverterFactory factory = (ConverterFactory)
      FactoryFinder.getFactory(FactoryFinder.CONVERTER_FACTORY);
```

The ConverterFactory implementation class supports the following methods:

- `public Converter getConverter(String converterId);`

This method creates (if necessary) and returns a `Converter` instance. All requests for the same converter identifier will return the same `Converter` instance, which must be programmed in a thread-safe manner.

- `public Iterator getConverterIds();`

This method returns an `Iterator` over the set of converter identifiers supported by this factory

- `public void addConverter(String converterId, Converter converter);`

At any time, additional `Converter` instances, and their corresponding identifiers, can be registered by the JSF implementation, by included component libraries, or by applications themselves.

JSF.7.6 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 `Renderer`s that are added to the standard `RenderKit` at application startup time, but applications must ensure that the standard `Renderer`s are made available for the web application to utilize them.

The JavaBean properties of standard component classes can be considered to be *render-independent* attributes of the components -- indeed, their values are available through the standard `getAttribute()` method or via property getter method calls. These values, as the term implies, are independent of the rendering technology that will be used to create the user interface. For example, a `UICommand` component has a `commandName` render-independent attribute that identifies the application command that should be performed when this command is selected. A `UICommand` can be displayed in many different ways, using many different rendering technologies -- but the *meaning* of a command name does not change.

Page authors can configure the precise details of the rendering that should be performed for JSF components by setting *render-dependent* attributes on those components, and by selecting a specific `Renderer` (from the RenderKit associated with this response) to be utilized by specifying the `rendererType` attribute[1]. As the name implies, these attributes are meaningful only to the `Renderer` actually selected (via a choice of `RenderKit`, and configuration of a `rendererType` attribute on each component) to perform the encoding and decoding processing for a particular interaction. Attributes that are not recognized by a particular `Renderer` are simply ignored.

The set of render-dependent attribute names and values will typically have a close correspondence to concepts in the markup language or technology used to actually create the response content, and to read the subsequent request. Indeed, this is the case for the Standard HTML RenderKit implementation -- to maximize familiarity for users already familiar with HTML, most of the render-dependent attribute names recognized by the library have a one-to-one correspondence with attribute names on the corresponding HTML elements that will be created.

1. For JSP page authors that use the Standard HTML RenderKit Tag Library described in Section FIXME/8.6, the rendererType setting is made implicitly by virtue of the action name selected for each component.

The required `Renderer`s are organized into subsections based on the standard `UIComponent` class supported by those `Renderer`s. Many of these Renderers share common attributes (or sets of attributes) -- to minimize the repetitiveness of the Renderer descriptions, the common rules are documented here.

TABLE 7-1 Commonly Used Renderer Attributes

Attribute Name	Description
columnClasses	A comma-delimited list of CSS style classes that will be used in sequence for the column elements of a table-oriented layout (using the HTML `class` attribute). If the rendered table has more columns than the number of class names in this list, the list will be restarted from the beginning as many times as necessary; however, the first column of a new row will always be assigned the first style class in this list.
converter	Either a String that contains the *converter id* of a `Converter` instance to be looked up in the `ConverterFactory` for this web application, or a `Converter` instance to be used directly, as described in Section FIXME/7.4.
dateStyle	Date style pattern name as defined by the `java.text.DateFormat` class. Valid values: `short`, `medium`, `long`, or `full`.
formatPattern	String version of a formatting pattern suitable for use with a `java.text.DateFormat` or `java.text.NumberFormat` instance.
numberStyle	Number style supported by the `java.text.NumberFormat` class. Valid values are: `currency`, `integer`, `number`, or `percent`.
rowClasses	A comma-delimited list of CSS style classes that will be used in sequence for the row elements of a table-oriented layout (using the HTML `class` attribute). If the rendered table has more columns than the number of class names in this list, the list will be restarted from the beginning as many times as necessary.
timeStyle	Time style pattern name as defined by the `java.text.DateFormat` class. Valid values) `short`, `medium`, `long`, or `full`.
timezone	A time zone identifier as defined by the `java.util.TimeZone` class.
xxxClass	A CSS style class to be used when rendering this component, where “xxx” corresponds to the component class name without the “UI” prefix, and converted to lower case[1].

1. In general, component attributes passed directly to the HTML output will have the same name as the corresponding HTML attribute, so one would expect an attribute named “class” to be used for this purpose. However, due to technical restrictions in JSP pages, this attribute name is difficult to use (requires a BeanInfo class). Therefore, an easy-to-remember convention for the CSS style class name has been substituted.

FIXME -- figure out a way to consisely document the attributes that correspond 1:1 with HTML/4.01 attributes and are simply passed through.

JSF.7.6.1 Renderer Types for UICommand Components

TABLE 7-1 Renderer Types for UICommand Components

Renderer Type	Data Type	Render Dependent Attributes	Decode Behavior	Encode Behavior
Button	String	label,type	Submit `FormEvent` to application as per default `UICommand` behavior.	Render an HTML `<input>` element with the specified `type` (submit, reset). The button text is specified literally by the `label` attribute.
Button	String	key, bundle, type	Submit `FormEvent` to application as per default `UICommand` behavior.	Render an HTML `<input>` element with the specified type (submit, reset). The button text is localized based on the `key` and `bundle` attributes.
Hyperlink	String	(none)	Submit `CommandEvent` to application.	Render an HTML `<a>` element that generates a hyperlink back to the tree identifier of the tree, so that a command event can be enqueued.
Hyperlink	String	href	(none)	Render an HTML `<a>` element that links to the specified URL. Nested `UIParameter` components (with the `name` attribute defined) can be used to configure request parameters that will be added to the specified URL. URL rewriting will be performed to maintain session state in the absence of cookies.

A `UICommand` component (see Section FIXME/4.1) is intended to represent a user interface component that a user can "activate" in some manner, which will trigger some application defined action. For HTML, two common method of rendering a `UICommand` are supported:

- As an HTML `<input>` element, used for submit and reset buttons nested inside a `<form>` element. Triggering such a button will cause a `FormEvent` to be submitted to the application, identifying which form and which submit button were submitted.
- As an HTML `<a>` (hyperlink) element, used to forward control to either submit a `CommandEvent` to the application (if no `href` attribute is specified), or forward control to an arbitrary URL (if `href` is specified).

The following requirements apply to the “Hyperlink” `Renderer` described in this section:

- Nested `UIGraphic` and `UIOutput` components can be used to provide the image background and/or text of the rendered button or hyperlink.

JSF.7.6.2 Renderer Types for UIForm Components

TABLE 7-1 Renderer Types for UIForm Components

Renderer Type	Data Type	Render Dependent Attributes	Decode Behavior	Encode Behavior
Form	(none)		Per specified default behavior for the UIForm decode() method.	Per the specified default behavior for the UIForm encodeXxx() methods.

The `UIForm` component (see Section FIXME/4.2) contains a set of UIInput components representing request parameters that will be included when the form is submitted, as well as `UICommand` components that represent submit and reset buttons. It will be rendered as an HTML `<form>` element, and all nested child components will be themselves rendered in between the beginning and ending elements of the form.

It will be quite common to manage the generated layout of a form by nesting a `UIPanel` component inside to generate the required layout information. For example, consider a component tree as follows, with increasing levels of indentation representing child components of the immediately preceding component:

`UIForm` component with formName=”logonForm”

- `UIPanel` component with rendererType=”Grid” and columns=”2”.
  - `UIOutput` component with rendererType=”Label” for the username prompt.
  - `UIInput` component with rendererType=”Text” for the username field
  - `UIOutput` component with rendererType=”Label” for the password prompt.
  - `UIInput` component with rendererType=”Secret” for the password field.

The “Grid” renderer will generate an HTML `<table>` element with two columns per row, and lay its nested components out inside `<td>` elements automatically, without the page author needing to be aware of how this is done. In addition, the appearance of the generated table can be managed using CSS stylesheets, by using the appropriate render-dependent attributes defined by the “Grid” renderer (see Section FIXME/7.6.6).

JSF.7.6.3 Renderer Types for UIGraphic Components

TABLE 7-1 Renderer Types for UIGraphic Components

Renderer Type	Data Type	Render Dependent Attributes	Decode Behavior	Encode Behavior
Image	String			Render an HTML `<img>` element with a `src` attribute set from the current value of the underlying component. URL rewriting is performed to maintain session state in the absence of cookies.
Image		key, bundle		Render an HTML `<img>` element with a `src` attribute localized from the resource bundle. URL rewriting will be performed to maintain session statee in the absence of cookies.

The `UIGraphic` component (see Section FIXME/4.3) represents an output-only display of a graphical image. For HTML, the "Image" `Renderer` creates an `<img>` element whose `src` attribute is either the current value of this component (no `key` attribute present), or is a localized value looked up in an appropriate resource bundle (`key` attribute present).

JSF.7.6.4 Renderer Types for UIInput Components

TABLE 7-1 Renderer Types for UIInput Components

Renderer Type	Data Type	Render Dependent Attributes	Decode Behavior	Encode Behavior
Date	Date, long	dateStyle, timezone	Get `DateFormat` instance as for encoding, parse text input, convert as necessary and store.	Get current value, convert to `Date`, create localized `DateFormat` date instance for the specified `dateStyle`, set time zone (if specified), render formatted result as an HTML `<input>` element of type "text".
DateTime	Date, long	formatPattern, timezone	Get `DateFormat` instance as for encoding, parse text input, convert as necessary and store.	Get current value, convert to `Date`, create localized `SimpleDateFormat` based on `formatPattern`, set time zone (if specified), render formatted result as an HTML `<input>` element of type "text".
DateTime	Date, long	dateStyle, timeStyle, timezone	Get `DateFormat` instance as for encoding, parse text input, convert as necessary and store.	Get current value, convert to `Date`, create localized `DateFormat` date/time instance set time zone (if specified), render formatted result as an HTML `<input>` element of type "text".
Hidden	Any	converter	Convert as necessary and store.	Get current value, convert to String, render as an HTML `<input>` element of type "hidden".
Number	numeric	formatPattern	Get `NumberFormat` instance as for encoding, parse text input, convert as necessary and store.	Get current value, convert to numeric (as appropriate), create localized `NumberFormat` instance based on `numberStyle`, render formatted result as an HTML `<input>` element of type "text".[1]
Number	numeric	numberStyle	Get `NumberFormat` instance as for encoding, parse text input, convert as necessary, and store.	Get current value, convert to numeric (as appropriate), create localized `NumberFormat` instance based on numberStyle, render as an HTML `<input>` element of type "text".

TABLE 7-1 Renderer Types for UIInput Components

Renderer Type	Data Type	Render Dependent Attributes	Decode Behavior	Encode Behavior
Secret	Any	converter, redisplay	Convert as necessary and store.	Get current value, convert to String, render as an HTML `<input>` element of type “password”. The `redisplay` attribute is a boolean flag (default=false) indicating whether or not the current value should be displayed[2].
Text	Any	converter	Convert as necessary and store.	Get current value, convert to String, render as an HTML `<input>` element of type “text”.
Textarea	Any	converter	Convert as necessary and store.	Get current value, convert to String, render as an HTML `<textarea>` element.
Time	Date, long	timeStyle, timezone	Get `DateFormat` instance as for encoding, parse text input, convert as necessary and store.	Get current value, convert to Date, create localized DateFormat time instance for the specified timeStyle, set time zone (if specified), render formatted result as an HTML `<input>` element of type “text”.

1. FIXME - When numberStyle is set to CURRENCY, we need to specify currencyCode and currencySymbol like the JSTL <fmt:formatNumber> tag does. It may be worth having a separate Renderer Type for this use case.
2. Although such a field displays asterisks on most browsers, the plaintext value is still easily visible when a user uses the “View Source” menu option, and this can be a security risk in some environments.

The `UIInput` component (see Section FIXME/4.4) represents a user interface component that both displays information (when a response is being rendered) and updates its current value on a subsequent request (typically a form submit). A variety of `Renderer`s are available to deal with common cases involving the display of Strings, numbers, and date/time values in a variety of localized or custom-pattern scenarios.

In HTML, the most common mechanism for rendering `UIInput` components is an <input> element of type “text”. For maximum flexibility, the “Text” renderer will accept current values of any type, and perform an appropriate conversion to String (when rendering the response) or from String (when processing the input). The details of conversion can be managed by selecting one of the following approaches:

- If no `converter` attribute is specified, any non-String value will be converted to String by calling its `toString()` method during rendering. On subsequent request processing, no conversion is performed -- the local value will be stored as a String.

- If a `converter` attribute is specified, it must be either a String (used to look up a `Converter` instance in the `ConverterFactory` for our web application), or a `Converter` instance that will be used directly to manage conversions.

JSF.7.6.5 Renderer Types for UIOutput Components

TABLE 7-1 Renderer Types for UIOutput Components

Renderer Type	Data Type	Render Dependent Attributes	Decode Behavior	Encode Behavior
Date	Date, long	dateStyle, timezone		Get current value, convert to `Date`, create localized `DateFormat` date instance for the specified `dateStyle`, set time zone (if specified), render formatted result.
DateTime	Date, long	formatPattern, timezone		Get current value, convert to `Date`, create localized `SimpleDateFormat` based on `formatPattern`, set time zone (if specified), render formatted result.
DateTime	Date, long	dateStyle, timeStyle, timezone		Get current value, convert to `Date`, create localized `DateFormat` date/time instance set time zone (if specified), render formatted result.
Errors	(none)	clientId		Render the set of error messages (if any) that are associated with the specified client identifier (if `clientId` is specified), or the global errors not associated with any component (if `clientId` is a zero-length string), or all errors (if `clientId` is not specified).
Label	(none)	for		Render the nested components surrounded by an HTML `<label>` element with the specified `for` attribute (which must match the rendered id attribute for some input field).
Message	Any	converter		Get current value, convert to String, treat as a MessageFormat pattern (nested UIParameter components may be used to provide values for parameter substitution), render formatted result.[1]
Message	Any	converter, key, bundle		Get localized resource from resource bundle, treat as a MessageFormat pattern (nested UIParameter components may be used to provide values for parameter substitution), render formatted result.

TABLE 7-1 Renderer Types for UIOutput Components

Renderer Type	Data Type	Render Dependent Attributes	Decode Behavior	Encode Behavior
Number	numeric	formatPattern		Get current value, convert to numeric (as appropriate), create localized `NumberFormat` instance based on `numberStyle`, render formatted result.[2]
Number	numeric	numberStyle		Get current value, convert to numeric (as appropriate), create localized `NumberFormat` instance based on numberStyle, render formatted result.
Text	Any	converter		Get current value, convert to String, render literally.
Time	Date, long	timeStyle, timezone		Get current value, convert to Date, create localized DateFormat time instance for the specified timeStyle, set time zone (if specified), render formatted result.

1. FIXME - Need to specify an optimization to skip actually processing the MessageFormat when not necessary, such as when there are no {0} type parameters in the message text, or when no substitution values are available.
2. FIXME - When numberStyle is set to CURRENCY, we need to specify currencyCode and currencySymbol like the JSTL <fmt:formatNumber> tag does. It may be worth having a separate Renderer Type for this use case.

The `UIOutput` component (see Section FIXME/4.5) represents a user interface component that only displays information when a response is being rendered. No processing is performed on a subsequent request. A variety of `Renderer`s are available to deal with common cases involving the display of Strings, numbers, and date/time values in a variety of localized or custom-pattern scenarios.

In HTML, it is very common to directly render the current value of the component as a character String. For maximum flexibility, the “Text” renderer will accept current values of any type, and perform an appropriate conversion to String (when rendering the response). The details of conversion can be managed by selecting one of the following approaches:

- If no `converter` attribute is specified, any non-String value will be converted to String by calling its `toString()` method during rendering.
- If a `converter` attribute is specified, it must be either a String (used to look up a `Converter` instance in the `ConverterFactory` for our web application), or a `Converter` instance that will be used directly to manage conversions.

JSF.7.6.6 Renderer Types for UIPanel Components

TABLE 7-1 Renderer Types for UIPanel Components

Renderer Type	Data Type	Render Dependent Attributes	Decode Behavior	Encode Behavior
Data	array, Collection, Map	var		Nested child components represent the template for a row to be generated for each element of a collection represented by the current value of this component. Each element of the collection will be exposed as a request attribute named by var, so that model reference expressions in the template components can access the values for the current element. There should be one child component per column to be created.
Grid	(none)	columnClasses, columns, footerClass, headerClass, rowClasses		Render an HTML `<table>` element, from our child components, based on the following rules: (a) if `headerClass` is specified, the first child will be rendered as a header row across all the columns, formatted according to this CSS style class; (b) Each intervening child is rendered in a separate table data element, with a new row started each columns children; (c) if `footerClass` is specified, the last child will be rendered as a footer row across all the columns, formatting according to this CSS style class.
Group	(none)			Render the child components as specified by their own renderer type settings. This renderer exists to allow the creation of arbitrary groups where a parent component expects to see a single child.
List	(none)	columnClasses, footerClass, headerClass, rowClasses		Render an HTML `<table>` element, from our child components, based on the following rules: (a) if `headerClass` is specified, the first child should be a `UIPanel` with a renderer type of `Group`, representing the column headers for the table to be rendered; (b) next children should be zero or more `UIPanel` components with a renderer type of `Data` that represent the data collection(s) to be iterated over, and whose children represent a template for each row to be rendered; (c) if footerClas is specified, the last child should be a `UIPanel` with a renderer type of `Group`, representing the column footeres for the table to be rendered.

The `UIPanel` component (see Section FIXME/4.5) represents a container for child components whose actual rendering will be managed by the `UIPanel` component (or its corresponding `Renderer`) instead of by the components themselves. It is commonly used to compose complex objects out of simple ones. The standard `Renderer`s for HTML manage a variety of complex layout scenarios by embedding appropriate HTML elements (such as `<table>`, `<tr>`, and `<td>`) so that the individual components do not need to worry about these responsibilities.

In many cases (such as the nested components inside a `UIPanel` using the “Grid” or “List” `Renderer`s, the requirements above require that each cell of the ultimately rendered table be generated by a single child of the outer `UIPanel`. In many cases, it would be useful to compose the contents of such a cell with an arbitrary set of grandchild components. This can be easily accomplished by using a `UIPanel` component with a renderer type of “Group” to represent a single component (from the perspective of the parent `UIPanel`) that has an arbitrarily complex internal structure.

JSF.7.6.7 Renderer Types for UISelectBoolean Components

TABLE 7-1 Renderer Types for UISelectBoolean Components

Renderer Type	Data Type	Render Dependent Attributes	Decode Behavior	Encode Behavior
Checkbox	boolean		Convert as necessary and store.	Get current value, convert to boolean, render as an HTML `<input>` element of type “checkbox”.

The `UISelectBoolean` component (see Section FIXME/4.8) is a specialized subclass of `UIInput` whose value is restricted to being a boolean `true` or `false`. In HTML, such a component is rendered as an HTML `<input>` element of type “checkbox”, and the decode behavior performed on a subsequent form submit will always ensure that the local value of this component reflects the checked or unchecked state of the checkbox in the user interface.

JSF.7.6.8 Renderer Types for UISelectMany Components

TABLE 7-1 Renderer Types for UISelectMany Components

Renderer Type	Data Type	Render Dependent Attributes	Decode Behavior	Encode Behavior
Checkbox	String	layout	Convert as necessary and store.	Get current value, convert to `String`, get available items from nested `UISelectItem` and `UISelectItems` components, render as a series of HTML `<input>` elements of type "checkbox", laid out according to the layout attribute (PAGE_DIRECTION or LINE_DIRECTION).
Listbox	String[]		Convert as necessary and store.	Get current value(s), convert to `String[]`, get available items from nested `UISelectItem` and `UISelectItems` components, render as an HTML `<select>` element that displays all possible values (so does not need a scrollbar).
Menu	String[]	size	Convert as necessary and store.	Get current value(s), convert to `String[]`, get available items from nested `UISelectItem` and `UISelectItems` components, render as an HTML `<select>` element that displays the number of elements specified by the `size` attribute (default=1).

The `UISelectMany` component (see Section FIXME/4.12) represents a user interface component that offers a fixed set of choices to the user, from which zero or more selections can be made. The data value associated with this component should be an array of objects reflecting the currently selected items from the set of available choices. The different Renderers described above offer different choices for how the set of available choices should be displayed.

The set of available choices is constructed by scanning the child components for one or more instances of `UISelectItem` (see Section FIXME/4.10) or `UISelectItems` (see Section FIXME/4.11) components. Existence of these child components causes the addition of a single `SelectItem` instance (for a `UISelectItem`) or a set of `SelectItem` instances (for a `UISelectItems`) to the set of available choices. For each available choice, the `itemLabel` property defines the visual appearance of this choice, while the `itemValue` property defines the value that will be added to the array of current selections for this component.

JSF.7.6.9 Renderer Types for UISelectOne Components

TABLE 7-1 Renderer Types for UISelectOne Components

Renderer Type	Data Type	Render Dependent Attributes	Decode Behavior	Encode Behavior
Listbox	String		Convert as necessary and store.	Get current value, convert to `String`, get available items from nested `UISelectItem` and `UISelectItems` components, render as an HTML `<select>` element that displays all possible values (so does not need a scrollbar).
Menu	String	size	Convert as necessary and store.	Get current value, convert to `String`, get available items from nested `UISelectItem` and `UISelectItems` components, render as an HTML `<select>` element that displays the number of elements specified by the `size` attribute (default=1).
Radio	String	layout	Convert as necessary and store.	Get current value, convert to `String`, get available items from nested `UISelectItem` and `UISelectItems` components, render as a series of radio buttons laid out according to the `layout` attribute (PAGE_DIRECTION, LINE_DIRECTION).

The `UISelectOne` component (see Section FIXME/4.13) represents a user interface component that offers a fixed set of choices to the user, from which zero or one selections can be made. The data value associated with this component should be an object reflecting the currently selected value from the set of available choices. The different `Renderers` described above offer different choices for how the set of available choices should be displayed.

The set of available choices is constructed by scanning the child components for one or more instances of `UISelectItem` (see Section FIXME/4.10) or `UISelectItems` (see Section FIXME/4.11) components. Existence of these child components causes the addition of a single `SelectItem` instance (for a `UISelectItem`) or a set of `SelectItem` instances (for a `UISelectItems`) to the set of available choices. For each available choice, the `itemLabel` property defines the visual appearance of this choice, while the `itemValue` property defines the value that will be returned for this component, if the corresponding item is selected.

JSF.8

Integration With JSP

JSF supports (but does not require) 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 tags corresponding to JSF UI components. A page author should be able to use JSF components in conjunction with the other custom actions (including the JSP Standard Tag Library), as well as standard HTML content and layout embedded in the page

JSF.8.1 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 `UITextEntry` component:

TABLE 8-1 EXAMPLE RENDERER TYPES

RendererType	Render-Dependent Attributes
“Text”	“columns”
“Secret”	“columns”, “secretChar”
“Multiline”	“columns”, “rows”

The tag library descriptor (TLD) file for the corresponding tag library, then, would define three custom tags -- one per `Renderer`. Below is an example of the tag definition for the “input_text” tag[1]:

1. This example illustrates a non-normative convention for naming tags 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 tag name; however the convention is rigorously followed in the Standard HTML RenderKit Tag Library.

```
<tag>
   <name>input_text</name>
   <tagclass>acme.html.tags.InputTag</tagclass>
   <bodycontent>JSP</bodycontent>
   <attribute>
      <name>id</name>
      <required>false</required>
   </attribute>
   <attribute>
      <name>modelReference</name>
      <required>false</required>
   </attribute>
   <attribute>
      <name>columns</name>
      <required>false</required>
   </attribute>
   ...
</tag>
```

Note that the `columns` attribute is derived from the `Renderer` of type “Text”, while the `id` and `modelReference` attributes are derived from the UIInput component class itself. `RenderKit` implementors will provide a JSP tag library which includes component tags corresponding to each of the component classes (or types) supported by each of the `RenderKit`’s `Renderers`. See FIXME for details on the `RenderKit` and `Renderer` APIs.

JSF.8.2 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.

JSF.8.2.1 Declaring the Custom Actions Tag Library

The page author must use the standard JSP `taglib` directive to declare the URI of the tag library to be utilized, as well as the tag prefix used (within this page) to identify tags 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” %>
```

declares the unique resource identifiers of the tag libraries being used (provided by the supplier of that tag libraries) for the two standard tag libraries that every JSF implementation must provide, as well as the tag prefixes to be used within the current page for referencing actions from these libraries[1].

Tag libraries for `UIComponent` custom actions can interoperate with additional tag libraries provided by others, including the JSP Standard Tag Library (JSTL), subject to the following restrictions: FIXME - document restrictions.

JSF.8.2.2 Defining 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. For example,

```
<h:input_text id=”username” modelReference=”logonBean.username”/>
```

represents a text entry field, to be displayed with the “Text” renderer type, for the user name field of a logon form component that is nested inside a root component. The *id* attribute specifies the *component id* of a `UIComponent` instance, from within the component tree, to which this tag corresponds. If no *id* is specified, one will be automatically generated by the JSF implementation.

Custom actions that correspond to JSF `UIComponent` instances must subclass either `javax.faces.webapp.FacesTag` (see Section FIXME / 9.2.4.3) or `javax.faces.webapp.FacesBodyTag` (see Section FIXME / 9.2.4.4), depending on whether the tag needs to support `javax.servlet.jsp.tagext.BodyTag` functionality or not.

1. 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, are are used consistently in tag library examples throughout this specification.

During the *Render Response* phase of the request processing lifecycle, the appropriate encoding methods of a `Renderer` of the type associated with this tag will be utilized to generate the representation of this component in the response page.

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 conjuction with the actions that represent UI components.

FIXME - document current <output_body> requirement on included pages

JSF.8.2.3 Creating Components and Overriding Attributes

As component tags are encountered during the processing of a JSP page, the tag mplementation must check the component tree for the existence of a corresponding `UIComponent`. Based on the results of this check:

- If no such `UIComponent` exists, a new component will be created and added as a child component of the component represented by the UIComponent custom action in which this action is nested (or the root component if this is the outermost component tag). All attributes specified on the component tag are used to initialize the attributes of the corresponding `UIComponent` instance.
- If such a `UIComponent` already exists, no new component is created. Instead, any attributes specified on the component tag are used to customize the attributes of the corresponding `UIComponent`, **unless** a value for that attribute has already been set (such as by the application handler in the *Invoke Application* phase).

Our example `<input_text>` tag supports a `columns` attribute, which allows the page author to configure the number of characters in the rendered input field, like this:

```
<h:input_text id=”username”
       modelReference=”logonBean.username” columns=”32”/>
```

JSF.8.2.4 Representing Component Hierarchies

Nested structures of UIComponent custom actions will generally mirror the hierarchical relationships of the corresponding `UIComponent` instances in the component tree that is associated with each JSP page. For example, assume that a UIForm component (whose component id is *logonForm*) is nested inside a root `UIPanel` component. You might specify the contents of the form like this:

```
<h:form id=”logonForm”>
<table border=”0”>
<tr>
       <td><h:output_label id=”usernameLabel”
```

```
            for=”/logonForm/username”>Username:
            </h:output_label></td>
      <td><h:input_text id=”username”
            modelReference=”logonBean.username”/></td>
</tr>
<tr>
      <td><h:output_label id=”passwordLabel”
            for=”/logonForm/password”>Username:
            </h:output_label></td>
      <td><h:input_secret id=”password”
            modelReference=”logonBean.password”/></td>
</tr>
<tr>
      <td><h:command_button id=”submitButton”
            type=”SUBMIT” commandName=”submit”/></td>
      <td> &nbsp; </td>
</tr>
</table>
</h:form>
```

Component tags with the `rendersChildren` property set to `true` will cause the child components in the component tree to be rendered along with the parent component. Therefore, the nested child components may, but need not, appear in the JSP page as tags.

JSF.8.2.5 Registering Request Event Handlers and Validators

[FIXME -- Add information on how to register a facet as well.]

Each JSF implementation is required to provide a custom tag library (see Section FIXME), which includes tags that (when executed) create instances of a specified `RequestEventHandler` or `Validator` implementation class, and register the created instance with the `UIComponent` associated with our most immediately surrounding UIComponent custom action. In addition, it is possible to register arbitrary attribute values (which are commonly used to configure validators) on the component itself.

Using these facilities, the page author can manage all aspects of creating and configuring values associated with the component tree, without having to resort to Java code. For example:

```
<h:input_text id=”username” modelReference=”logonBean.username”>
```

```
        <f:validate_required/>
        <f:validate_length minimum="6"/>
</h:input_text>
```

associates two validators (a check for required input, and a minimum length check) with the username component being described.

JSF.8.2.6 Interoperability with Other Custom Action Libraries

It is permissible to use other custom action libraries, such as the JSP Standard Tag Library (JSTL) in the same JSP page with UIComponent custom actions that correspond to JSF components. When JSF component actions are nested inside tags from other libraries, the following behaviors must be supported:

- All JSF tags nested inside of, and positioned as siblings to the custom tags, must have a manually defined *component identifier*.
- Conditional custom actions (such as the `<c:if>` and `<c:choose>` actions in the JSP Standard Tag Library) may dynamically determine whether nested body content inside these tags is rendered or not. JSF component custom actions nested inside such tags will not be invoked if the outer tag chooses to skip its body. Therefore, no components will be created (in the component tree) to correspond to these custom actions. However, any such components created during a request processing lifecycle phase prior to *Render Response* must be included in the saved state information that will be communicated to the *Reconstitute Component Tree* phase of the subsequent request.
- Iterative custom actions (such as the `<c:forEach>` and `<c:forTokens>` actions in the JSP Standard Tag Library) may dynamically choose to process their nested body content zero or more times. JSF component custom actions nested within the body of such a tag must operate as follows:
  - If the body of the iterative custom action is never executed, nested JSF component custom actions will not be invoked. Therefore, no components will be created (in the component tree) to correspond to these custom actions. However, any such components created during a request processing lifecycle phase prior to *Render Response* must be included in the saved state information that will be communicated to the *Reconstitute Component Tree* phase of the subsequent request.
  - If the body of the iterative custom action is executed exactly once, nested JSF component custom actions must be treated exactly as if they were nested inside a conditional custom action (see the previous paragraph) that chose to render its nested body content.

- If the body of the iterative custom action is executed more than once, [FIXME - need a mechanism to have a UIComponent child be an **array** of UIComopnents (one per iteration) as well as supporting model reference expressions that can use the loop index from the `LoopTagStatus` instance exported by a JSTL iteration tag.

In addition to the general interoperability requirements described above, the following additional requirements must be satisfied when interoperating with tags from the JSP Standard Tag Library (JSTL):

- [FIXME - interaction with any `LocalizationContext` of tags from the "I18n-capable formatting tag library" described in Chapter 8 of the JSTL specification]
- [FIXME - other specific interoperability requirements to be determined]

[FIXME - we will probably need to specify that JSF depends on the availability of JSTL APIs so we can deal with things like `javax.servlet.jsp.jstl.core.LoopTagStatus` and `javax.servlet.jsp.jstl.fmt.LocalizationContext`).

JSF.8.2.7 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. From the perspective of JSF, such inclusions are transparent -- the page is compiled as if the inclusions had been performed before compilation was initiated.
- The `<jsp:include>` standard action performs 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 tags in the included page 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. [FIXME - do we need to restrict this to flush="false"?]
- Other mechanisms that perform a `RequestDispatcher.include()` on a JSP page from the same web application (such as use of the `include()` method on the `PageContext` object associated with a page, or accessing an internal resource with the `<c:import>` tag of the JSP Standard Tag Library) must cause the component tree to be manipulated in the same manner as that described for the `<jsp:include>` standard 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 acquired via `ServletContext.getServletContext()`, or accessing an external resource with the `<c:import>` tag of the JSP Standard Tag Library), 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.

JSF.8.3 The <f:use_faces/> Tag

All of the JSF component tags used on a given page must be nested inside a UseFaces tag, whose implementation will be responsible for saving the state of the component tree during the execution of its `doEndTag()` method. This tag must be defined in the tag library descriptor of the JSF Core Tag Library described in Section FIXME/8.5.

JSF.8.4 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 [FIXME - make more precise]

- Extend the `FacesTag` or `FacesBodyTag` base class, so that JSF implementations can recognize UIComponent custom actions versus others.
- Usage and interpretation of the `id` attribute
- Rules about overrides being applied only if the corresponding attribute was given a value in the page; otherwise, defaults from the component tree apply
- Other behavioral requirements specific to JSF?

JSF.8.5 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 (<tlib-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).

JSP pages using the JSF standard tag library must declare it in a `taglib` like this (using the suggested prefix value):

```
<%@ taglib uri=”http://java.sun.com/jsf/core” prefix=”f” %>
```

Each action included in the Standard JSF Tag Library is documented in a subsection below, with the following outline for each action:

- **Name** -- The name of this tag, as used in a JSP page.
- **Short Description** -- A summary of the behavior implemented by this tag.
- **Syntax** -- One or more examples of using this tag, with the required and optional sets of attributes that may be used together.
- **Body Content** -- The type of nested content for this tag, 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 standard tags, JSF component tags, and/or other custom tags) that can be nested in the body of this tag.
- **Attributes** -- A table containing one row for each defined attribute for this tag. The following columns provide descriptive information about each attribute:
  - *Name* -- Name of this attribute, as it must be used in the page.
  - *Dyn* -- Boolean value indicating whether this attribute accepts dynamic values (documented in the <rtexprvalue> element).
  - *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 action.

JSF.8.5.1 <f:attribute>

Add a String-valued attribute for a `UIComponent` tag in which we are nested.

Syntax:

```
<f:attribute name="attribute-name" value="attribute-value"/>
```

Body Content:

empty.

Attributes:

Name	Dyn	Type	Description
name	false	String	Name of the component attribute to be set
value	false	String	Value of the component attribute to be set

Constraints:

- Must be nested inside a UIComponent tag.

Description:

Locates the `UIComponent` for the closest parent tag whose implementation class extends `javax.faces.webapp.FacesTag`. If this component does not already have a component attribute with a name specified by this tag's `name` attribute, create a String valued component attribute with the name and value specified by this tag's attributes.

JSF.8.5.2 <f:parameter>

Specify an optionally named `UIParameter` component for a parent component.

Syntax:

Syntax 1: Unnamed direct value

```
<f:parameter id="componentId" value="parameter-value"/>
```

Syntax 2: Unnamed indirect value

```
<f:parameter id="componentId"
        modelReference="model-reference-expression"/>
```

Syntax 3: Named direct value

```
<f:parameter id="componentId" name="parameter-name"
        value="parameter-value"/>
```

Syntax 4: Named indirect value

```
<f:parameter id="componentId" name="parameter-name"
        modelReference="model-reference-expression"/>
```

Body Content:

empty.

Attributes:

Name	Dyn	Type	Description
id	false	String	Component identifier of a UIParameter component
modelReferen ce	false	String	Model reference expression to acquire the value of this parameter
name	false	String	Name of the parameter to be set
value	false	String	Value of the parameter to be set

Constraints:

- Must be nested inside a `UIComponent` tag.
- Must specify exactly one of the `modelReference` or `value` attributes.

Description:

Implements the standard FacesTag functionality to create a new `UIParameter` component, if one does not already exist in the component tree. Such components are used to dynamically configure parameter values for component tags in which it this tag is nested, such as substitution values for MessageFormat replacement patterns ({0}) in the `output_message` tag of the Standard HTML Tag Library.

The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.FacesTag`.
- The `createComponent()` method must create and return a new `UIParameter` instance.
- The `getRendererType()` method must return `null`.

JSF.8.5.3 <f:use_faces>

Container tag for all JSF core and component tags used on a page.

Syntax:

```
<f:use_faces>
       Nested template text and tags
</f:use_faces>
```

Body Content:

JSP. May contain any combination of template text, other JSF tags, and tags from other custom tag libraries.

Attributes:

None.

Constraints:

- Any JSP-created response using actions from the JSF Core Tag Library, as well as actions extending `javax.faces.webapp.FacesTag` from other tag libraries, must be nest their use of such tags inside an occurrence of the `<f:use_faces>` action.
- JSP page fragments included via the standard <%@ include %> directive need not have their JSF actions embedded in a `<f:use_faces>` action, because the included template text and tags will be processed as part of the outer page as it is compiled, and the `<f:use_faces>` action on the outer page will meet the nesting requirement.
- JSP pages included via `RequestDispatcher.include()` or `<jsp:include>` may use an `<f:use_faces>`, but they need not do so if the calling page already contains such an action.

Description:

Provides the JSF implementation a convient 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 actual processing performed by this action is determined by the JSF implementation.

JSF.8.5.4 <f:validate_doublerange>

Register a `javax.faces.validator.DoubleRangeValidator` instance on our surrounding component

Syntax:

Syntax 1: Maximum only specified

`<f:validate_doublerange maximum="543.21"/>`

Syntax 2: Minimum only specified

`<f:validate_doublerange minimum="123.45"/>`

Syntax 3: Both maximum and minimum are specified

`<f:validate_doublerange maximum="543.21" minimum="123.45"/>`

Body Content:

empty.

Attributes:

Name	Dyn	Type	Description
maximum	false	double	Maximum value allowed for this component
minmum	false	double	Minimum value allowed for this component

Constraints:

- Must be nested inside a `UIComponent` tag 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:

Creates, configures, and registers an instance of `javax.faces.validator.DoubleRangeValidator` on the `UIComponent` created by the `UIComponent` action within which we are nested.

The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.ValidatorTag`.
- The `createValidator()` method must create, configure, and return a new `DoubleRangeValidator` instance.

JSF.8.5.5 <f:validate_length>

Register a `javax.faces.validator.LengthValidator` instance on our surrounding component

Syntax:

Syntax 1: Maximum only specified

```
<f:validate_length maximum="16"/>
```

Syntax 2: Minimum only specified

```
<f:validate_length minimum="3"/>
```

Syntax 3: Both maximum and minimum are specified

```
<f:validate_length maximum="16" minimum="3"/>
```

Body Content:

empty.

Attributes:

Name	Dyn	Type	Description
maximum	false	int	Maximum length allowed for this component
minmum	false	int	Minimum length allowed for this component

Constraints:

- Must be nested inside a `UIComponent` tag 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:

Creates, configures, and registers an instance of `javax.faces.validator.LengthValidator` on the `UIComponent` created by the `UIComponent` action within which we are nested.

The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.ValidatorTag`.
- The `createValidator()` method must create, configure, and return a new `LengthValidator` instance.

JSF.8.5.6 <f:validate_longrange>

Register a `javax.faces.validator.LongRangeValidator` instance on our surrounding component

Syntax:

Syntax 1: Maximum only specified

`<f:validate_longrange maximum="543"/>`

Syntax 2: Minimum only specified

`<f:validate_longrange minimum="123"/>`

Syntax 3: Both maximum and minimum are specified

`<f:validate_longrange maximum="543" minimum="123"/>`

Body Content:

empty.

Attributes:

Name	Dyn	Type	Description
maximum	false	long	Maximum value allowed for this component
minmum	false	long	Minimum value allowed for this component

Constraints:

- Must be nested inside a `UIComponent` tag 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:

Creates, configures, and registers an instance of `javax.faces.validator.LongRangeValidator` on the `UIComponent` created by the `UIComponent` action within which we are nested.

The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.ValidatorTag`.
- The `createValidator()` method must create, configure, and return a new `LongRangeValidator` instance.

JSF.8.5.7 <f:validate_required>

Register a `javax.faces.validator.RequiredValidator` instance on our surrounding component

Syntax:

```
<f:validate_required/>
```

Body Content:

empty.

Attributes:

None.

Constraints:

- Must be nested inside a `UIComponent` tag.

Description:

Creates, configures, and registers an instance of `javax.faces.validator.RequiredValidator` on the `UIComponent` created by the `UIComponent` action within which we are nested.

The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.ValidatorTag`.
- The `createValidator()` method must create, configure, and return a new `RequiredValidator` instance.

JSF.8.5.8 <f:validate_stringrange>

Register a `javax.faces.validator.StringRangeValidator` instance on our surrounding component

Syntax:

Syntax 1: Maximum only specified

`<f:validate_stringrange maximum="XYZ"/>`

Syntax 2: Minimum only specified

`<f:validate_stringrange minimum="ABC"/>`

Syntax 3: Both maximum and minimum are specified

`<f:validate_stringrange maximum="XYZ" minimum="ABC"/>`

Body Content:

empty.

Attributes:

Name	Dyn	Type	Description
maximum	false	String	Maximum value allowed for this component
minmum	false	String	Minimum value allowed for this component

Constraints:

- Must be nested inside a `UIComponent` tag whose value is (or is convertible to) 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:

Creates, configures, and registers an instance of `javax.faces.validator.StringValidator` on the `UIComponent` created by the `UIComponent` action within which we are nested.

The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.ValidatorTag`.
- The `createValidator()` method must create, configure, and return a new `StringRangeValidator` instance.

JSF.8.5.9 <f:validator>

Create and register a Validator for a `UIComponent` tag in which we are nested.

Syntax:

`<f:validator type=”`*`classname`*`”/>`

Body Content:

empty.

Attributes:

Name	Dyn	Type	Description
type	false	String	Fully qualified class name of a class that extends javax.faces.validator.Validator.

Constraints:

- Must be nested inside a UIComponent tag.

Description:

Locates the `UIComponent` for the closest parent tag whose implementation class extends `javax.faces.webapp.FacesTag`. If this component was created by this execution of the surrounding tag, creates a new instance of a `Validator` subclass whose fully qualified class name is specified by the `type` attribute, and registers this instance with the component by calling `addValidator()`. The specified class must have a public zero-arguments constructor.

JSF.8.6 Standard HTML RenderKit Tag Library

All JSF implementations must provide a tag library containing actions (described below) that correspond to each valid combination of a supported component class (see Chapter FIXME/4) and a Renderer from the Standard HTML RenderKit (see Section FIXME/7.6) that supports that comopnent 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 (<tlib-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).

JSP pages using the JSF standard tag library must declare it in a `taglib` like this (using the suggested prefix value):

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
```

Each action included in the Standard JSF Tag Library is documented in a subsection below, with the following outline for each action:

FIXME -- specify the mapping of component+rendererType to tags for all the combinations listed in Section FIXME/7.6.

JSF.9

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.

JSF.9.1 Web Application Deployment Descriptor

JSF-based applications are *web applications* that conform to the requirements of the Servlet Specification (version 2.3 or later), and also use the facilities defined in this specification. Conformant 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.

JSF.9.1.1 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, 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 the next section).

JSF.9.1.2 Servlet Mapping

All requests to a web application are mapped to a particular servlet based on matching a URL pattern (as defined in the Servlet Specification) against the portion of the request URL after the context path that selected this web application. The following mapping for the standard servlet providing portable JSF lifecycle procesing must be supported:

```
<servlet-mapping>
      <servlet-name> faces-servlet-name </servlet-name>
      <url-pattern>
            /faces/*
      </url-pattern>
</servlet-mapping>
```

The servlet name, denoted by *faces-servlet-name* above, may be any desired value; however, the same value must be used in the Servlet Definition (see the previous section).

JSF.9.1.3 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.lifecycle.LIFECYCLE_ID`** - Lifecycle identifier of the `Lifecycle` instance to be used when processing JSF requests in this web application. If not specified, the JSF default instance, identified by `LifecycleFactory.DEFAULT_LIFECYCLE`, will be used.

FIXME - list of other standard `javax.faces.xxxxx` parameters to be supported

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.

JSF.9.2 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 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 archive itself.

The following sections describe how various subsets of the required classes and resources should be packaged, and how they should be made available.

JSF.9.2.1 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.

JSF.9.2.2 Servlet and JSP API Classes (javax.servlet.*)

These classes will typically be made available to all web applications using the shared classes facilities of the servlet container[1]. Therefore, these classes should not be included inside the web application archive.

1. This is already a requirement for all J2EE containers.

JSF.9.2.3 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 (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.

JSF.9.2.4 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.

JSF.9.2.4.1FactoryFinder

The `javax.faces.FactoryFinder` class implements a standard discovery mechanism for locating JSF-implementation-specific implementations of several factory classes defined by JSF. The discovery method is static, so that it may be conveniently called from JSF implementation or application code, without requiring a reference to some global container.

- `public static Object getFactory(String factoryName);`

Create (if necessary) and return an instance of the implementation class for the specified factory name. This method must ensure that there is exactly one instance for each specified factory name, per web application supported in a servlet container. For a given factory name, the following process is used to identify the name of the factory implementation class to be instantiated:

- If there is a system property whose name matches the requested factory name, its value is assumed to be the name of the factory implementation class to use.
- If there is a `faces.properties` resource file visible to the web application class loader of the calling application, and if this resource file defines a property whose name matches the requested factory name, its value is assumed to be the name of the factory implementation class to use.
- If a `META-INF/services/{factory-name}` resource file is visible to the web application class loader of the calling application (typically as a result of it being included in a JAR file containing the corresponding implementation class), the first line of this resource file is assumed to contain the name of the factory implementation class to use.
- If none of the above conditions are satisfied, a `FacesException` is thrown.

Once the name of the factory implementation class to use is identified, `FactoryFinder` must load this class from the web application class loader, and then instantiate an instance for the current web application. Any subsequent request for the same factory name, from the same web application, must cause the previously created instance to be returned.

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.context.ConverterFactory* (`FactoryFinder.CONVERTER_FACTORY`) - Factory for `Converter` instances.
- *javax.faces.context.FacesContextFactory* (`FactoryFinder.FACES_CONTEXT_FACTORY`) - Factory for `FacesContext` instances.
- *javax.faces.context.LifecycleFactory* (`FactoryFinder.LIFECYCLE_FACTORY`) - Factory for `Lifecycle` instances.
- *javax.faces.context.MessageResourcesFactory* (`FactoryFinder.MESSAGE_RESOURCES_FACTORY`) - Factory for `MessageResources` instances.
- *javax.faces.context.RenderKitFactory* (`FactoryFinder.RENDER_KIT_FACTORY`) - Factory for `RenderKit` instances.
- *javax.faces.context.TreeFactory* (`FactoryFinder.TREE_FACTORY`) - Factory for `Tree` instances.

JSF.9.2.4.2FacesServlet

`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 FIXME/9.1.1, and mapped to a standard URL pattern as described in Section FIXME/9.1.2.

- `public void init(ServletConfig config);`

Acquire and store references to the `FacesContextFactory` and `LifecycleFactory` instances to be used in this web application.

- `public void destroy();`

Release the FacesContextFactory and LifecycleFactory references that were acquired during execution of the `init()` method.

- `public void service(ServletRequest request, ServletResponse response);`

For each incoming request, the following processing is performed:

- Using the `FacesContextFactory` instance stored during the `init()` method, call the `createFacesContext()` method to acquire a `FacesContext` instance with which to process the current request.
- Store the `FacesContext` instance as a request attribute under key "`javax.faces.context.FacesContext`" (`FacesContext.FACES_CONTEXT_ATTR`).
- Using the `LifecycleFactory` instance stored during the `init()` method, call the `createLifecycle()` method to acquire a `Lifecycle` instance with which to perform the request processing lifecycle for the current request. [FIXME - how to portably specify the lifecycle identifier to be used]
- Call the `execute()` method of the `Lifecycle` instance, passing the `FacesContext` instance for this request as a parameter. If the `execute()` method throws a `FacesException`, rethrow it as a `ServletException` with the `FacesException` as the root cause.
- Remove the request attribute containing the `FacesContext` instance.
- Call the `release()` method on the FacesContext instance, allowing it to be returned to a pool if the JSF implementation uses one.

JSF.9.2.4.3FacesTag

`FacesTag` is a subclass of javax.servlet.jsp.tagext.TagSupport, and must be the base class for any JSP custom tag that corresponds to a JSF `UIComponent`. It supports all of the standard functionality of `TagSupport`, plus the following functionality that is specific to JSF:

- `protected UIComponent findComponent();`

Using the current value of the `id` property of this tag handler, locate and return the corresponding `UIComponent` instance from the component tree, in accordance with the rules outlined in Section FIXME/8.2.2, above.

If your tag needs to implement `javax.servlet.jsp.tagext.BodyTag` support, it should subclass `FacesBodyTag` (described in the following section) instead.

JSF.9.2.4.4FacesBodyTag

`FacesBodyTag` is a subclass of `FacesTag`, so it inherits all of the JSF-specific 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.

JSF.9.3 Included Configuration Files

JSF defines portable configuration file formats (as XML documents) for standard configuration information such as message catalogs ... FIXME - document the formats and default locations here, with references back to the configuration parameters that support customization of these locations.