JavaServer™ Faces Specification

Version 1.0, Expert Draft 20020730

July 30, 2002

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: `jsr127-comments@sun.com`

JavaServer(TM) Faces Specification ("Specification")
Version: 1.0
Status: Pre-FCS
Release: July 9, 2002

JSF.1

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).

JSF.1.1 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/>

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>

JSF.1.2 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/>

JSF.1.3 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.

JSF.1.4 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
- dealing with differences in markup across different browsers and versions
- 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
- 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 render-independent and render-dependent attributes of those components as custom action attributes in the JSP page. In addition, the pages produced by a page author will be the source of *metadata* about the components used in those pages, which will be 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 representation of the persistent information, such as Entity Enterprise JavaBeans (Entity EJBs)
- 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

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 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.

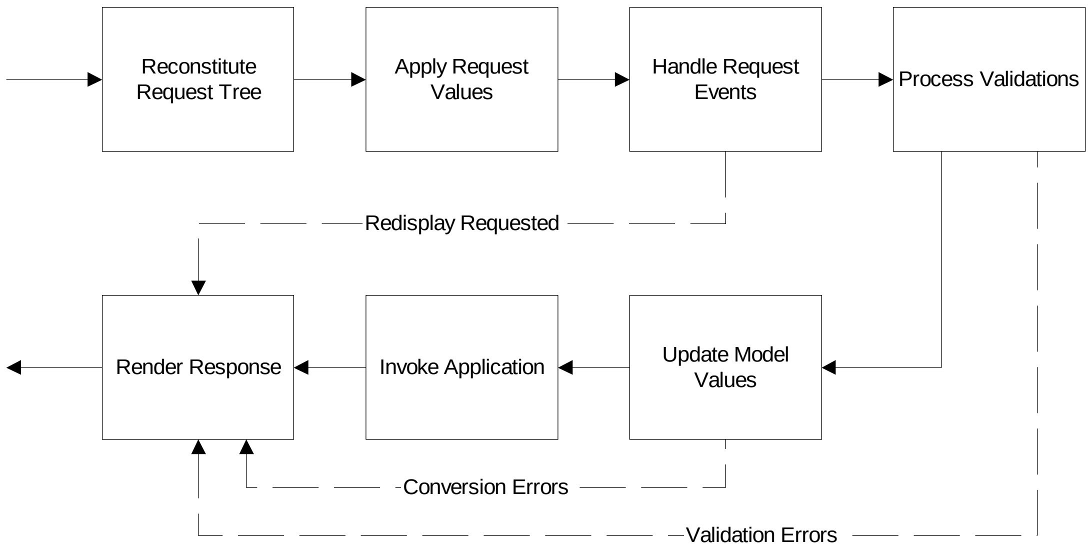
JSF.1.4 Using the JSF APIs

FIXME - some simple examples of using JSF.

JSF.2

Request Processing Lifecycle

Each servlet request processed by a JSF implementation goes through a well-defined *request processing lifecycle* made up of seven *phases*, to create the corresponding servlet response. The phases are illustrated in the following diagram:



The normal processing flow is indicated by solid arrows, while alternative transfers to the *Render Response* phase are indicated dotted line arrows.

The detailed processing to be performed by the JSF implementation in each phase is described in the Sections below.

JSF.2.1 Reconstitute Request Tree

The JSF implementation must perform the following tasks during the *Reconstitute Request 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 request component tree (if any was saved by a previous JSF_generated response that generated the page from which this request was submitted).
- If such state information is available, use it to construct a component `Tree`, including any required wire-up of event handlers 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 `setRequestTree()` method[1].
- If no such state information is available, create a component `Tree` with no components, configured to use the default `RenderKit` implementation, and store it in the `FacesContext` by calling `setRequestTree()`. The tree identifier for this tree is derived from the request URI for the current request by [FIXME - specify portable algorithm].
- 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 `requestTree` property of the `FacesContext` instance for the current request will reflect the saved configuration of the component tree generated by the previous response (if any).

JSF.2.2 Apply Request Values

During the *Apply Request Values* phase of the request processing lifecycle, the JSF implementation must walk down the components in the request tree, and ask each of them to decode the relevant information from the request (parameters, headers, cookies, etc.), and to update its current value. The following algorithm is applied to each component in turn:

- Consult the `rendererType` property of this component, to determine whether or not it has been set.

1. The FacesContext implementation will also set its responseTree property to the same Tree instance. See Section FIXME for more information.

- If no `rendererType` has been set, the component is assumed to be responsible for its own decoding. The JSF implementation will call the component's `decode()` method to perform this task.
- If a `rendererType` has been set, it is assumed that encoding of this component was delegated to a `Renderer`, so that delegation of decoding is also required. The Renderer instance to use is acquired by acquiring the `RenderKit` associated with the request tree, and calling its `getRenderer()` method. The `decode()` method of this `Renderer` instance will be called, passing the component for which decoding should occur.

The `decode()` method, on either the component instance or the associated `Renderer` instance, has the following responsibilities and options:

- Extract from the request the parameter(s) and/or other values that will be used to construct the new value associated with with this component.
- Attempt to construct a new value for this component, including any type conversions necessary.
- If construction of the new value is successful, store it as the local value of this component, by calling its `setValue()` method. In addition, call `setValue(true)` on the component to set the current validity of the local value.
- If construction of the new value is not successful (such as a failed String->numeric conversion), store enough information inside the component so that the incorrect data can be redisplayed when this component is encoded during the *Render Response* phase. In addition, a `Message` documenting the conversion failure should be queued, by calling an appropriate `addMessage()` method on the `FacesContext` instance for the current request. In addition, call `setValue(false)` on the component to set the current validity of the local value.
- Optionally, enqueue one or more `FacesEvent` events, on either the current component or any other component in the request component tree. These events will be processed during the *Handle Request Events* phase.
- Optionally, enqueue one or more `CommandEvent` or `FormEvent` events, to be processed during the *Invoke Application* phase, by calling the `addApplicationEvent()` method of the `FacesContext` instance for this request.

At the end of this phase, all components in the request 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). Optionally, events and validation error messages might also have been queued.

JSF.2.3 Handle Request Events

Request events are typically used when some aspect of the current request indicates the need for a change in visual representation, either for the current component (such as clicking a node in a tree control to expand or contract it), or for some other component in the component tree (such as clicking a checkbox causing a change in the set of options to be shown in a combobox list). Separating event processing into a separate phase allows event processing methods to assume that **all** components have been assigned their new values.

During the *Handle Request Events* phase of the request processing lifecycle, the JSF implementation must call `getRequestEventsCount()`on the associated `FacesContext`. If the number of queued events is greater than zero, walk down the components in the request tree, and ask each of them to handle any events that have been queued for that component by calling the `processEvents()` method on each component[1].

Request Event handlers have the following responsibilities and options:

- Update the state of the current component in the request component tree, based on the event that has occurred.
- Update the state of other components in the request component tree that are affected by the event that has occurred.
- Change the response component tree to select an outbound page different than the inbound page being processed, by creating a new `Tree` instance with a tree identifier that is meaningful to the `ViewHandler` being utilized, and saving it by calling the `setResponseTree()` method on the FacesContext instance for the current request.
- Add and remove components in the response component tree, and/or customize their attributes.
- Change the `RenderKit` that will be utilized during the *Render Response* phase for the current response.
- Indicate that control should be transferred to the *Render Response* phase when request event processing has been completed, by returning `true` from the `processEvent()` method.

Once all events for all components have been processed, the JSF implementation proceeds to the *Process Validations* phase, or directly to the *Render Response* phase, if any of the event handlers that were called returned `true`.

JSF.2.4 Process Validations

As part of the original creation of the component tree for this request, zero or more `Validator` instances may have been registered for each component. During the *Process Validations* phase of the request processing lifecycle, the JSF implementation must walk down the components in the request tree, and call the processValidators() method on each component. This method (described in Section FIXME/3.1.10) will cause the `validate()` method of the component itself, along with any registered `Validator`s associated with this component, to be called.

1. As a performenace optimization, the call to processEvents() may be skipped if a call to getRequestEventsCount(UIComponent) for the specified component returns zero.

The validate() method of the component, and of each registered `Validator,` has the following responsibilities and options:

- Report the component attributes that this Validator instance uses for configuration (if any), via the `getAttributeNames()` method[1], as well as an `AttributeDescriptor` for each supported attribute via the `getAttributeDescriptor()` method.
- Examine the local value of the specified component for correctness, based on the rules being checked by this `Validator`. If any violations are identified, enqueue a corresponding `Message` by calling an appropriate `addMessage()` method on the `FacesContext` instance for this request. In addition, `setValid(false)` must be called on the corresponding component instance.
- Change the response component tree to select an outbound page different than the inbound page being processed, by creating a new `Tree` instance with a tree identifier that is meaningful to the `ViewHandler` being utilized, and saving it by calling the `setResponseTree()` method on the FacesContext instance for the current request.
- Add and remove components in the response component tree, and/or customize their attributes.
- Change the `RenderKit` that will be utilized during the *Render Response* phase for the current response.

Once all registered validations have been performed, the state of the queued message on the `FacesContext` is examined. If one or more messages have been queued, the JSF implementation will transfer control to the *Render Response* phase. Otherwise, control will proceed to the *Update Model Values* phase.

JSF.2.5 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, that the local value of every component in the request component tree has been updated, and that it is appropriate to update the application's model data in preparation for performing any application events that have been enqueued. This occurs during the *Update Model Values* phase, where the JSF implementation must walk down the components in the request tree, and call the `updateModel()` method on each component.

It is possible that conversion errors will occur during the executions of the `updateModel()` method. If such errors occur, enqueue appropriate validation error `Message`s to the `FacesContext` instance for this request.

After model value updates have been completed for all components, control should be forwarded to the *Render Response* phase of the request processing lifecycle if any conversion errors were recorded. Otherwise, the JSF implementation must walk

1. For example, a generic Validator used to perform length checks on an incoming text value might use attributes to configure the minimum and maximum allowed number of characters.

down the request component tree again, and clear the local values for all components that have a non-`null` modelReference property value, by calling the `setValue()` method with a `null` parameter.

JSF.2.6 Invoke Application

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

- Consult the `applicationHandler` property of the `FacesContext` instance that is processing the current request.
- If no application event handler has been registered, there is no mechanism for processing application events, so any such queued events will be ignored.
- If an application event handler has been registered, call the getApplicationEventsCount() method on the FacesContext instance for the current request.
- If the number of queued application events is greater than zero, 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:

- Change the response component tree to select an outbound page different than the inbound page being processed, by creating a new `Tree` instance with a tree identifier that is meaningful to the `ViewHandler` being utilized, and saving it by calling the `setResponseTree()` method on the FacesContext instance for the current request.
- Add and remove components in the response 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`.
- Cause control to be transferred immediately to the *Render Response* phase of the request processing lifecycle, bypassing any remaining application events that have been queued, by returning `true` from the `processEvent()` method.

[FIXME - do we need a “response events” queuing and processing phase here?]

JSF.2.7 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 response 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 response 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 response 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 response component tree first. If the component already exists (because the response component tree is still the same as the request component tree, or 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.

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.

- 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 response tree is selected for rendering, it must be processed in the following manner:

- When a component is being rendered after a conversion failure detected during the *Apply Request Values* phase, the encoding methods of the component (if not delegating) or the `Renderer` (if delegating) should reproduce the original (incorrect) input.
- Consult the `rendererType` property of the component, to determine whether this component wishes to delegate encoding to an appropriate `Renderer` instance. If this property is null, the component will be responsible for rendering itself. If this property is non-null, use this value to ask the RenderKit associated with the response tree to return an appropriate `Renderer` instance for this renderer type.
- Call the `encodeBegin()` method of the component (if not delegating) or the `Renderer` (if delegating).
- Consult the rendersChildren property of the component, to determine whether this component is responsible for rendering its child components as well as itself.
- If the `rendersChildren` property is `true`, call the encodeChildren() method of the component (if not delegating) or the Renderer (if delegating). In addition, the JSF implementation must ensure that the child components are not themselves selected for rendering.
- Call the `encodeEnd()` method of the component (if not delegating) or the `Renderer` (if delegating).

Upon completion of rendering, the completed state of the resopnse 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 Request 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 - say something about whether response.sendRedirect() and/or response.sendError() is allowed?]

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

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*.

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 located based on *component identifiers*, which must be unique within the children of a parent node, and a *compound identifier* that is unique within an entire component tree.

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

JSF.3.1 UIComponent

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.

JSF.3.1.1 Component Type

- `public abstract String getComponentType();`

All concrete UIComonent 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);`
- `public String getCompoundId();`

Every component is named by a *component identifier*, which must be unique among the components that share a common parent in a component tree. Component identifiers are used by JSF implementations, as they render components, to name request parameters for a subsequent request from the JSF-generated page. 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 identifiers be as short as possible.

Individual components within a component tree are uniquely identified by a *compound identifier*, which is calculated by treating the nodes of the component tree as a hierarchical namespace of all the components in the tree, and then composing the path to the desired node according to the following rules (FIXME - improve precision and conciseness):

- Start with a slash (‘/’) character, which represents the root node.
- Append the component identifier of the eldest ancestor (under the root node) of the component whose compound identifier is being composed.
- For each additional ancestry level, append a slash (‘/’) character and the component identifier of the next lower ancestor.
- End with the component identifier of the component being named.

For example, in a component tree consisting of the following nodes (identified by their component identifiers - FIXME replace ASCII art):

foo --> bar and baz which has child bop

the compound identifier of the “bop” component is “`/foo/baz/bop`”.

JSF.3.1.3 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.3.1.4 Component Tree Navigation

- `public UIComponent findComponent(String expression);`

Treat the specified `navigation expression` as an absolute or relative path to a desired destination comonent elsewhere in the current component tree. The syntax of valid expressions is analogous to the use of absolute and relative pathnames in filesystems of operating systems such as Unix (FIXME-more rigorous definition). For example, the following expressions are interpreted as described:

- **`/`** - Return the root node of the component tree this component is a part of.
- **`foo`** - Return the child of the current component whose component identifier is “foo”.
- **`../foo`** - Return the child of this component’s parent whose component identifier is “foo” (i.e. a sibling of the current node).
- **`/foo/bar`** - Return the component whose compound identifier is “/foo/bar”.

JSF.3.1.5 Model Object References

- `public String getModelReference();`
- `public void setModelReference(String modelReference);`

FIXME - describe how model references work, syntax, forward reference to how currentValue() does its thing.

JSF.3.1.6 Local Values

- `public Object getValue();`
- `public void setValue(Object value);`
- `public Object currentValue(FacesContext context);`

During the execution of the request processing lifecycle for a JSF-based page, a `UIComponent` representing an input value (such as a text field) will be asked to store a copy of the value entered by the user, and hold it for redisplay, even if the information entered by the user is syntactically or semantically invalid. This is temporarily stored input is called the *local value* of the component, and is stored in the `value` property. See section FIXME for a description of how and when the local value is utilized.

For components that are associated with an object in the model data of an application (that is, components with a non-null model reference expression in the `modelReference` property), the `currentValue()` method is used to retrieve the local value if there is one, or to retrieve the underlying model object if there is no local value. If there is no model reference expression, `currentValue()` returns the local value if any; otherwise it returns `null`.

JSF.3.1.7 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. 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 `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.

To facilitate the use of component classes in tools, all JavaBean properties available via getter and setter methods must also be available as attributes under the same name. 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.

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

JSF.3.1.8 Request Event Processing

- `public void addRequestEventHandler(RequestEventHandler handler);`
- `public void clearRequestEventHandlers();`
- `public Iterator getRequestEventHandlers();`
- `public void removeRequestEventHandler(RequestEventHandler handler);`

As the component tree for an incoming request is being constructed, or at any later time until the *Handle Request Events* phase of the request processing lifecycle, zero or more `RequestEventHandler`s can be associated with each component in the tree.

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

During the *Apply Request Values* phase of the request processing lifecycle, components can queue events to be processed later during the *Handle Request Events* phase, by calling `addRequestEvent()` on the associated `FacesContext`. Request events can be used, for example, to deal with scenarios where a particular input should change the visual appearance of the same component (such as expanding or contracting a node in a tree control), or a different component (such as clicking a checkbox changing the list of options available in a `UISelectOne` component).

Later, during the *Handle Request Events* phase, the JSF implementation will call the `processEvents()` method of each component to which one or more events have been queued. The default implementation of `processEvents()` will in turn call the `processEvent()` method of the component, and any registered request event handlers for this component, once per event. If any of the `processEvent()` method calls on this component or registered request event handler returns true, `processEvents()` will itself return `true` to indicate that control should be transferred to the *Render Response* phase, once event processing has been completed for all components.

See Section FIXME/3.4 for more information about `FacesEvent`s, and Section FIXME/2.4 for further details on how and when the event processing methods are called by the JSF implementation.

JSF.3.1.9 Update Model Values Processing

- `public void updateModel(FacesContext context);`

During the *Update Model Values* phase of the request processing lifecycle, the JSF implementation will walk down the request component tree and call this method on each component. The following processing must be performed:

- If the `modelReference` property of this component is null, take no further action.
- Update the model data corresponding to the model reference, setting it to the local value of this component (performing type conversions as needed)
- If a conversion error occurs, enqueue an appropriate `Message` by calling an `addMessage()` method on the FacesContext instance for the current request

The default `UIComponent` implementation performs the update step by calling the `setModelValue()` method on the FacesContext instance for the current request, passing the model reference expression and the result of calling `getValue()` on this component as parameters. Concrete subclasses of `UIComponent` may override this method when more sophisticated logic is required to perform the update.

JSF.3.1.10 Validation Processing

- `public void addValidator(Validator validator);`
- `public void clearValidators();`

- `public Iterator getValidators();`
- `public void removeValidator(Validator validator);`

As the component tree for an incoming request is being constructed, or at any later time until the *Process Validations* phase of the request processing lifecycle, zero or more `Validator`s can be associated with each component in the tree. During the *Process Validations* phase, the JSF implementation will call `getValidators()` to retrieve the set of Validators associated with each component, and will call the `validate()` method of that Validator to check the correctness of the local value for this component.

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

During the *Process Validations* phase of the request processing lifecycle, the JSF implementation will call the `processValidators()` method of each component of the tree. This method must call the `validate()` method of the component itself, followed by the `validate()` method of each `Validator` that has been registered for this component.

See Section FIXME/3.5 for more information about `Validator`s, and Section FIXME/2.5 for more information on how and when the validation methods are called by the JSF implementation.

- `public boolean isValid();`
- `public void setValid(boolean valid);`

The isValid() method returns an indication of whether the construction of a new value, performed by the *Apply Request Values* phase of the request processing lifecycle, was successful or not. Components that do their own decoding will cause isValid() to return the correct results based on the success or failure of the conversion attempted during the Apply Request Values phase of the request processing lifecycle (see Section FIXME/2.2). If decoding is delegated to a `Renderer`, the `Renderer` must call `setValid()` to specify the success or failure of the conversion that it performed.

During the *Process Validations* phase of the request processing lifecycle, validation errors will also cause the value of the `valid` property to be set to `false`.

JSF.3.1.11 Decoding and Encoding

- `public String getRendererType();`
- `public void setRendererType(String rendererType);`

The most visible behaviors of a `UIComponent` are the paired operations of *decoding* (converting incoming request parameters into a local value of the type represented by this component) and *encoding* (converting the current value of this component into the corresponding markup, such as HTML elements, that represents it in the response that is being created. JSF supports two programming models for defining how these operations are performed:

- *Direct implementation* - the corresponding functional logic is directly implemented in the component implementation class, in the `decode()` and `encodeXxxxx()` family of methods.
- *Delegated implementation* - the corresponding functional logic is delegated to the decode() and encodeXxx() methods of a `Renderer` that is associated with this component at runtime.

The choice of programming model is based on the presence or absence of a non-null value for the `rendererType` property. If this property is set to `null`, JSF will assume that the component class directly implements these behaviors, and will call the corresponding methods on the component instance itself. If this property is not `null`, JSF will identify a `Renderer` instance of the corresponding `rendererType`, by calling the `getRenderer()` method of the `RenderKit` being used to process this request (for decoding) or response (for encoding).

See Sections FIXME/2.3 and FIXME/2.8 for more information on how and when the decoding and encoding methods of the component, or the associated `Renderer`, are called.

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

This method is called during the *Apply Request Values* phase of the request processing lifecycle. This method has the following responsibilities:

- Extract from the incoming request (tyipcally 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.
- If conversion is unsuccessful, save enough state information for the corresponding `encodeXxx()` methods to reproduce the incorrect value[1].

If conversion is unsuccessful, one more more validation errors may also be added to the message list associated with the `FacesContext` for the current request. See section FIXME/2.5 for more details on validation error message processing.

1. This feature is required in order to meet the user expectation that, when errors occur, the user will be presented with the values he or she originally entered (even if they are syntactically or semantically incorrect) so that they can be repaired and resubmitted.

The default implementation of `decode()` in `UIComponentBase` must delegate to the corresponding `Renderer`, if `rendererType` is not `null`; otherwise it does nothing except set the `valid` property to `true`.

- `public boolean getRendersChildren();`
- `public void encodeBegin(FacesContext context) throws IOException;`
- `public void encodeChildren(FacesContext context) throws IOException;`
- `public void encodeEnd(FacesContext context) throws IOException;`

As noted earlier in Section FIXME/3.1.3, 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 the 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.

The encodeXxx() methods will be called during the *Render Response* phase of the request processing lifecycle, as follows. The default implementations in `UIComponentBase` delegate to the appropriate `Renderer` (if `rendererType` is not `null`); otherwise they do nothing.

For components whose `rendersChildren` property is false, the following calls will occur:

- The `encodeBegin()` method of this component will be called.
- All child components will be rendered as defined by their own characteristics.
- The `encodeEnd()` method of this component will be called

For components whose `rendersChildren` property is true, the following calls will occur:

- The `encodeBegin()` method of this component will be called.
- The `encodeChildren()` method of this component will be called.
- The `encodeEnd()` method of this component will be called.

For components that generate nested markup elements, the `encodeBegin()` method will generally render the beginning tag (i.e. <form>), and the `encodeEnd()` method will generally render the corresponding ending tag (i.e. </form>). For components that do not support a notion of nested markup elements, all of the rendering should, by convention, be placed in the `encodeBegin()` method.

- `public boolean getRendersSelf();`

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.

This method returns a hint to development tools that this component class has implementations of the `decode()` and `encodeXxx()` methods, making it suitable for the *direct implementation* programming model. For example, a tool that generates a JSP custom tag library for every combination of a `UIComponent` class and a `Renderer` that supports it would also want to create a custom tag for supported components that return `true` for this property.

JSF.3.2 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 Request 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.
- Prior to the *Render Response* 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.

1. Unless the application does this, the request tree and response tree will be the same Tree instance.

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

JSF.3.3 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 signatures:

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

This method instantiates a `Tree` instance, preconfigured with a default `renderKitId` provided by the JSF implementation, as well as components with initial properties and attributes for use on the corresponding page.

JSF.3.4 Event Classes

JSF.3.4.1 FacesEvent

While components are being processed during the *Apply Request Values* phase of the request processing lifecycle, events can be queued for later processing to two different destinations:

- To the same component, or to different components in the request component tree, for processing during the Handle Request Events phase, by calling `addRequestEvent()` on the `FacesContext` instance for the current request.
- To the `ApplicationHandler` configured for the `Lifecycle` instance that is performing the request processing lifecycle for this request, for processing during the Invoke Application phase, by calling `addApplicationEvent()` on the `FacesContext` instance for the current request.

Events queued for either destination are encapsulated in an object of type `FacesEvent` (or a concrete subclass of `FacesEvent`). Standard subclasses of `FacesEvent` are provided for the types of events used by an application. Component writers will generally define custom `FacesEvent` subclasses for events that are recognized by related components from the same component library.

`FacesEvent` is the base class for all event objects defined by JavaServer Faces, as well as for all custom event classes defined by component writers. `FacesEvent` itself is a subclass of `java.util.EventObject`, because it conforms to the design patterns of the JavaBeans Specification. This class, and the concrete subclasses provided by JavaServer Faces, are in the `javax.faces.event` package.

This class, and its concrete subclasses, should be defined as immutable objects with property getters, but no property setters.

- `FacesEvent(UIComponent source)`

`FacesEvent` provides a constructor that accepts a `UIComponent` that, by convention, was the source of this event. Subclasses will often provide constructors that accept additional parameters to specify values for additional property getters defined by the subclass. Such subclass constructors must ensure that the superclass constructor is called, so that all superclass properties are initialized correctly.

- `public UIComponent getComponent();`

Return the `UIComponent` that was the source of this `FacesEvent`. It is a type-safe alias for the `getSource()` method defined by `java.util.EventObject`.

JSF.3.4.2 Standard FacesEvent Implementations

JSF.3.4.2.1 CommandEvent

`CommandEvent` is a concrete subclass of `FacesEvent` that indicates an application command that should be processed by the application, during the *Invoke Application* phase of the request processing lifecycle. Such events are queued during the *Apply Request Values* phase, by calling the `addApplicationEvent()` method on the `FacesContext` instance for the request that is being processed.

- `CommandEvent(UIComponent source, String commandName);`

`CommandEvent` provides a constructor that accepts the source component, as well as the command name of the command to be executed by the application during the *Invoke Application* phase of the request processing lifecycle.

- `public String getCommandName();`

Return the command name of the command to be executed.

JSF.3.4.2.2 FormEvent

`FormEvent` is a concrete subclass of `FacesEvent` that indicates a form submission that should be processed by the application, during the *Invoke Application* phase of the request processing lifecycle. Such events are queued during the *Apply Request Values* phase, by calling the `addApplicationEvent()` method on the `FacesContext` instance for the request that is being processed.

- `FormEvent(UIComponent source, String formName, String commandName);`

`FormEvent` provides a constructor that accepts the source component, as well as the form name of the form that has been submitted to the application for processing during the *Invoke Application* phase of the request processing lifecycle, and the name of the command that caused this form to be submitted.

- `public String getFormName();`

Return the form name of the form that has been submitted.

- `public String getCommandName();`

Return the command name of the submit button that caused this form to be submitted.

JSF.3.5 Validator Classes

JSF.3.5.1 Validator

A `Validator` is an interface implemented by classes that can perform validation (i.e., correctness checks) on a UIComponent. Zero or more `Validator`s can be registered with each component in the request component tree (during the *Reconstitute Request Tree* phase), and the `validate()` method of each registered Validator will be called during the *Process Validations* phase of the request processing lifecycle.

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.

Individual `Validator` instances must examine the component passed to the `validate()` method for conformance to any rules being enforced by that instance. Rules can be formulated based solely on the value of this `UIComponent`, or in relationship to the values of other components in the request component tree (conveniently accessible via the `findComponent()` method of the passed `UIComponent`). If a rule violation is encountered, one or more `Message` instances can be added to the `FacesContext` for the current request, by calling the `addMessage()` method. See Section FIXME/5.2 for more information about the `Message` APIs.

`Validator` is the base interface for all validator objects defined by JavaServer Faces, as well as for all custom validators defined by component writers, application developers, and tool providers. This interface, and the concrete implementations provided by JavaServer Faces, are in the `javax.faces.validator` package.

- `public void validate(FacesContext context, UIComponent component);`

This method will be called, during the *Process Validations* phase, for each `Validator` that is registered on each `UIComponent` instance in the request component tree. The method should examine the component's local value, and ensure that the value is in conformance with the rules enforced by this `Validator`. If any rules violations are found, one or more `Message` instances should be added to the `FacesContext` instance for the current request, which will cause the messages to be returned to the user for correction. In addition, the setValid(false) method on the offending component should be called on validation failures.

- `public Iterator getAttributeNames();`
- `public AttributeDescriptor getAttributeDescriptor(String name);`

For maximum generality, `Validator` classes may define configuration properties that are stored as attributes of the `UIComponent` instance they are associated with. For example, a `Validator` that checks the length of a `UITextEntry` component's input might declare the minimum and/or maximum length constraints as attributes, so that the same `Validator` class can be used on different components, with different limits.

Components that accept configuration properties in this manner must implement the `getAttributeNames()` method to return a list of attribute names that this Validator recognizes, and the `getAttributeDescriptor()` method to return an `AttributeDescriptor` instance for each recognized attribute name.

Attribute names that begin with "javax.faces." are reserved for use by the standard `Validator` classes provided by JSF.

JSF.3.5.2 Standard Validator Implementations

A variety of general purpose `Validator` subclasses are provided by JSF, in the `javax.faces.validator` package. These subclasses share the method signatures of the underlying `Validator` base class, along with the following common characteristics:

- Standard `Validators` that accept configuration parameters as component attributes define manifest String constants for the attribute names that they recognize, as well as supporting the required API-level support for `getAttributeNames()` and `getAttributeDescriptor()`. These constants are useful when `Validator`s are being configured programmatically.
- To support internationalization, `Message` instances are 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.

JSF.3.5.2.1 DoubleRangeValidator

`DoubleRangeValidator` checks the local `value` of a component, which must be of a floating point type (or whose String value is convertible to double), against maximum and/or minimum values specified by configuration attributes (either or both limits may be specified). The following configuration attribute names are recognized:

- *`javax.faces.validator.DoubleRangeValidator.MAXIMUM`* (`DoubleRangeValidator.MAXIMUM_ATTRIBUTE_NAME`) - the name of a configuration attribute that, if present, contains the maximum allowed value.
- *`javax.faces.validator.DoubleRangeValidator.MINIMUM`* (`DoubleRangeValidator.MINIMUM_ATTRIBUTE_NAME`) - the name of a configuration attribute that, if present, contains the minimum allowed value.

The following message identifiers are used on `Message`s added by this `Validator`:

- *`javax.faces.validator.DoubleRangeValidator.LIMIT`* (`DoubleRangeValidator.LIMIT_MESSAGE_ID`) - Either the maximum or minimum configuration attribute is not of the correct type (a floating point type or String convertible to double).
- *`javax.faces.validator.DoubleRangeValidator.MAXIMUM`* (`DoubleRangeValidator.MAXIMUM_MESSAGE_ID`) - The current value of this component exceeds the specified maximum. The offending value may be used in the error message text by using the `{0}` placeholder.
- *`javax.faces.validator.DoubleRangeValidator.MINIMUM`* (`DoubleRangeValidator.MINIMUM_MESSAGE_ID`) - The current value of this component is smaller than the specified minimum. The offending value may be used in the error message text by using the `{0}` placeholder.
- *`javax.faces.validator.DoubleRangeValidator.TYPE`* (`DoubleRangeValidator.TYPE_MESSAGE_ID`) - The current `value` of this component is not of the correct type (`a floating point type or String convertible to double`).

JSF.3.5.2.2 LongRangeValidator

`LongRangeValidator` checks the local `value` of a component, which must be of an integer type (or whose String value is convertible to a long), against maximum and/or minimum values specified by configuration attributes (either or both limits may be specified). The following configuration attribute names are recognized:

- *`javax.faces.validator.LongRangeValidator.MAXIMUM`* (`LongRangeValidator.MAXIMUM_ATTRIBUTE_NAME`) - the name of a configuration attribute that, if present, contains the maximum allowed value.
- *`javax.faces.validator.LongRangeValidator.MINIMUM`* (`LongRangeValidator.MINIMUM_ATTRIBUTE_NAME`) - the name of a configuration attribute that, if present, contains the minimum allowed value.

The following message identifiers are used on `Message`s added by this `Validator`:

- *`javax.faces.validator.LongRangeValidator.LIMIT`* (`LongRangeValidator.LIMIT_MESSAGE_ID`) - Either the maximum or minimum configuration attribute is not of the correct type (an integer type or String convertible to long).
- *`javax.faces.validator.LongRangeValidator.MAXIMUM`* (`LongRangeValidator.MAXIMUM_MESSAGE_ID`) - The current value of this component exceeds the specified maximum. The offending value may be used in the error message text by using the `{0}` placeholder.
- *`javax.faces.validator.LongRangeValidator.MINIMUM`* (`LongRangeValidator.MINIMUM_MESSAGE_ID`) - The current value of this component is smaller than the specified minimum. The offending value may be used in the error message text by using the `{0}` placeholder.
- *`javax.faces.validator.LongRangeValidator.TYPE`* (`LongRangeValidator.TYPE_MESSAGE_ID`) - The current `value` of this component is not of the correct type (an integer type or String convertible to long).

JSF.3.5.2.3 LengthValidator

`LengthValidator` checks the length (i.e. number of characters) of the local `value` of a component, which must be of type `java.lang.String (or convertible to a String)`, against maximum and/or minimum values specified by configuration attributes (either or both limits may be specified). The following configuration attribute names are recognized:

- *`javax.faces.validator.LengthValidator.MAXIMUM`* (`LengthValidator.MAXIMUM_ATTRIBUTE_NAME`) - the name of a configuration attribute that, if present, contains the maximum allowed value.
- *`javax.faces.validator.LengthValidator.MINIMUM`* (`LengthValidator.MINIMUM_ATTRIBUTE_NAME`) - the name of a configuration attribute that, if present, contains the minimum allowed value.

The following message identifiers are used on `Messages` added by this `Validator`:

- *`javax.faces.validator.LengthValidator.LIMIT`* (`LengthValidator.LIMIT_MESSAGE_ID`) - Either the maximum or minimum configuration attribute is not of the correct type (an integer type or String convertible to long).
- *`javax.faces.validator.LengthValidator.MAXIMUM`* (`LengthValidator.MAXIMUM_MESSAGE_ID`) - The length of the current value of this component exceeds the specified maximum. The offending value may be used in the error message text by using the `{0}` placeholder.
- *`javax.faces.validator.LengthValidator.MINIMUM`* (`LengthValidator.MINIMUM_MESSAGE_ID`) - The length of the current value of this component is smaller than the specified minimum. The offending value may be used in the error message text by using the `{0}` placeholder.

JSF.3.5.2.4 RequiredValidator

`RequiredValidator` checks for a non-null local value of a component. 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. No configuration attribute names are recognized by this `Validator`.

The following message identifiers are used on `Messages` added by this `Validator`:

- *`javax.faces.validator.RequiredValidator.FAILED`* (`RequiredValidator.FAILED_MESSAGE_ID`) - The local `value` of the component is `null`.

JSF.3.5.2.5 StringRangeValidator

`StringRangeValidator` checks the local `value` of a component, which must be of type `java.lang.String`, against maximum and/or minimum values specified by configuration attributes (either or both limits may be specified). The following configuration attribute names are recognized:

- *`javax.faces.validator.StringRangeValidator.MAXIMUM`* (`StringRangeValidator.MAXIMUM_ATTRIBUTE_NAME`) - the name of a configuration attribute that, if present, contains the maximum allowed value (as an instance of `java.lang.String`).
- *`javax.faces.validator.StringRangeValidator.MINIMUM`* (`StringRangeValidator.MINIMUM_ATTRIBUTE_NAME`) - the name of a configuration attribute that, if present, contains the minimum allowed value (as an instance of `java.lang.String`).

The following message identifiers are used on `Messages` added by this `Validator`:

- *javax.faces.validator.StringRangeValidator.LIMIT* (StringRangeValidator.LIMIT_MESSAGE_ID) - Either the maximum or minimum configuration attribute is not of the correct type (java.lang.String).
- *javax.faces.validator.StringRangeValidator.MAXIMUM* (StringRangeValidator.MAXIMUM_MESSAGE_ID) - The current value of this component exceeds the specified maximum. The offending value may be used in the error message text by using the {0} placeholder.
- *javax.faces.validator.StringRangeValidator.MINIMUM* (StringRangeValidator.MINIMUM_MESSAGE_ID) - The current value of this component is smaller than the specified minimum. The offending value may be used in the error message text by using the {0} placeholder.
- *javax.faces.validator.StringRangeValidator.TYPE* (StringRangeValidator.TYPE_MESSAGE_ID) - The current value of this component is not of the correct type (java.lang.String).

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 are subclasses of `javax.faces.component.UIComponent`.

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 configured (via the use of a `RenderKit` for the corresponding `Tree`, and the `rendererType` property on each component) to use the *delegated implementation* programming model, as described in Section FIXME/2.1.10.

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 current value of this component, enqueue a `FormEvent` to the application, passing the form name of the form being submitted and the command name of this `UICommand`.
- `encodeEnd()` - Render an HTML submit button, using the current value of this component as the “name” attribute.

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/form/yyyyy", where "xxxxx" is the context path of the current web application, "yyyyy" is the form name of this form. 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 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 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 `Renderers` 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.

JSF.4.5 UIPanel

The `UIPanel` class 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 acomponent implementation class for layout management, in many cases, without the need to subclass.

JSF.4.6 UISelectBoolean

The `UISelectBoolean` class 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.

JSF.4.7 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.4.8 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.4.9 UISelectItems

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.

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.

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

JSF.4.10 UISelectMany

The `UISelectMany` class 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.

JSF.4.11 UISelectOne

The `UISelectOne` class 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.

JSF.4.12 UITextEntry

The `UITextEntry` class represents a user interface component that represents a text entry field in an input form. It is typically rendered as either a single-line text field or a multi-line text box.

- `public String getText();`
- `public void setText(String text);`

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

The `text` property identifies the current field content of the text to be displayed by this component.

The default rendering functionality for `UITextEntry` 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 <input type=”text”> element..

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 pool and recycle `FacesContext` instances rather than creating new ones for each request.

JSF.5.1.1 Servlet API Components

- `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 at the beginning of the request. (FIXME - how to deal with new sessions created by the application during a request.)

- `public HttpSession getHttpSession(boolean create);`

If the `create` parameter is `false`, return the `HttpSession` instance for the current request that was used to initialize this FacesContext instance for the current request (if any). If the `create` parameter is `true`, create a new `HttpSession` associated with this request if there is not one in existence, and return it.

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 Request and Response Trees

- `public Tree getRequestTree();`
- `public void setRequestTree(Tree tree);`
- `public Tree getResponseTree();`
- `public void setResponseTree(Tree tree);`

During the *Reconstitute Request 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 `setRequestTree()` to establish it. The `setRequestTree()` method cannot be called again until `release()` has been called for this `FacesContext` instance.

Unless and until `setResponseTree()` is called later, the `getResponseTree()` method must return the same `Tree` instance as the one returned by `getRequestTree()`. This facilitates handling the situation where the current page needs to be redisplayed in response to request event handling, or the existence of validation error messages that must be returned to the user for correction.

JSF.5.1.4 Application Events Queue

- `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 request 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 `Message`s, 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 Request Events Queue

- `public Iterator getRequestEvents(UIComponent component);`
- `public int getRequestEventsCount();`
- `public int getRequestEventsCount(UIComponent component);`
- `public void addRequestEvent(UIComponent component, FacesEvent event);`

During the inbound phases of the request processing lifecycle up to and including the *Apply Request Values* phase, user interface components can queue events (via a call to the `addRequestEvent()` method) to be handled by the the same component, or other components, during the *Handle Request Events* phase of the request processing lifecycle. The total number of queued request events can be retrieved by calling `getRequestEventsCount()`, the total number of queued request events for a particular component can be retrieved by calling `getRequestEventsCount(UIComponent)`, and the queued events for a particular component can be retrieved by calling `getRequestEvents(UIComponent)`.

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.

- `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:

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.

- **`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.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.

In addition, the Lifecycle class defines symbolic constants for each defined phase of the request processing lifecycle. These constants are guaranteed to start at zero and be defined in ascending numeric sequence, so that the values will represent the execution order of the defined phases. The defined values are `RECONSTITUTE_REQUEST_TREE_PHASE` (0), `APPLY_REQUEST_VALUES_PHASE` (10), `HANDLE_REQUEST_EVENTS_PHASE` (20), `PROCESS_VALIDATIONS_PHASE` (30), `UPDATE_MODEL_VALUES_PHASE` (40), `INVOKE_APPLICATION_PHASE` (50), and `RENDER_RESPONSE_PHASE` (60).

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

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

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. It must accomplish the effect of the following algorithm (although the implementation details need not match these precise steps):

1. Select the phase identifier of the next phase to be executed. Upon initial entry, this will be the phase identifier for the *Reconstitute Request Tree* phase. Upon subsequent repetitions, the selected value will depend upon the *state change indicator* value that is returned by the `execute()` method of the previously executed `Phase` instance.

2. Acquire a list of all of the `Phase` instances associated with this phase identifier. This list will include: (a) all `Phase` instances registered with the `registerBefore()` method on the `LifecycleFactory` instance for this web application, **most** recently registered instance first; (b) the `Phase` instance that implements the standard functionality for this phase, provided by the JSF implementation; and (c) all `Phase` instances registered with the `registerAfter()` method on the LifecycleFactory instance for this web application, **least** recently registered instance first. [FIXME - this needs to be updated to reflect the fact that Lifecycle instances can compose their own set of phases as needed]

3. For each Phase instance in the list acquired in the previous step, perform the following:

   a. Call the `executePhase()` method of this `Lifecycle` instance, which will in turn call the `execute()` method of the Phase instance, and remember the returned state change indicator value.

   b. If the returned state change indicator value was Phase.GOTO_EXIT, exit immediately from the `execute()` method of this `Lifecycle` instance.

   c. If the returned state change indicator value was Phase.GOTO_RENDER, return to Step 1 of this algorithm, causing the *Render Response* phase identifier to become the next selected phase identifier.

   d. If the returned state change indicator value was Phase.GOTO_NEXT, repeat this step for the next `Phase` instance in the list, if there is one.

   e. Otherwise, proceed to the next Phase.

4. Return to Step 1, incrementing the selected phase identifier value.

- `public ApplicationHandler getApplicationHandler();`

- `public void setApplicationHandler(ApplicationHandler handler);`

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 response 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 response component tree into a context-relative resource path, and performs a `RequestDispatcher.forward()` to the corresponding resource.

JSF.6.2 Phase

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

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 request 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 `setResponseTree()` 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 response 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 request or response 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 void 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. This method 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 call setValid(true) on this component instance.
- If conversion is unsuccessful, save enough state information for the corresponding `encodeXxx()` methods to reproduce the incorrect value[1], and call setValid(false) on this component instance.

If conversion is unsuccessful, one or more validation errors may also be added to the message list associated with the `FacesContext` for the current request. See section FIXME for more details on validation error message processing.

- `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[2]. 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, the `encodeXxx()` methods of the corresponding Renderer instance will be called during the *Render Response* phase of the request processing lifecycle, as follows. For components whose `rendersChildren` property is `false`, the following calls will occur:

- The `encodeBegin()` method of the `Renderer` instance utilized for this component will be called.
- All child components will be rendered as defined by their own characteristics.

1. This feature is required in order to meet the user expectation that, when errors occur, the user will be presented with the values he or she originally entered (even if they are syntactically or semantically incorrect) so that they can be repaired and resubmitted.
2. 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.

- The `encodeEnd()` method of the `Renderer` instance utilized for this component will be called.

For components whose `rendersChildren` property is `true`, the following calls will occur:

- The `encodeBegin()` method of the `Renderer` instance utilized for this component will be called.
- The `encodeChildren()` method of the `Renderer` instance utilized for this component will be called.
- The `encodeEnd()` method of the `Renderer` instance utilized for this component will be called.

For components that generate nested markup elements, the `encodeBegin()` method will generally render the beginning tag (i.e. <form>), and the `encodeEnd()` method will generally render the corresponding ending tag (i.e. </form>). For components that do not support a notion of nested markup elements, all of the rendering should, by convention, be placed in the `encodeBegin()` method.

- `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 (FIXME - code box paragraph style?)

```
RenderKitFactory factory = (RenderKitFactory)
      FactoryFinder.getFactory(FactoryFinder.RENDER_KIT_FACTORY);
```

The `RenderKitFactory` implementation class supports the following methods:

- `public RenderKit getRenderKit(String renderKitId) throws FacesException;`

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 request or response component tree by calling the `setRenderKit()` method of the corresponding `Tree` instance.

JSF.7.4 Standard 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. JSF implementors, and other parties, may also provide additional `RenderKit` libraries, but applications must ensure that the corresponding implementation classes are made available in the web application that utilizes them.

FIXME - specify requirements for the basic HTML `RenderKit`.

JSF.8

Integration With JSP

JSF supports 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 Attributes
“Input”	“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 “textentry_input” tag[1]:

```
<tag>
```

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.

```
        <name>textentry_input</name>
        <tagclass>acme.html.tags.InputTag</tagclass>
        <bodycontent>empty</bodycontent>
        <attribute>
            <name>id</name>
            <required>true</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 "Input", while the `id` and `modelReference` attributes are derived from the UITextEntry component class itself. `RenderKit` implementors must 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://www.mycompany.com/HTMLRenderKit"
prefix="faces" %>
```

declares the unique resource identifier of the tag library being used (provided by the supplier of that tag library), as well as the tag prefix to be used within the current page. It is recommended, but not required, that “`faces`” be used as the tag prefix.

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,

```
<faces:textentry_input id=”/logonForm/username”
       modelReference=”logonBean.username”/>
```

represents a text entry field, to be displayed with the “Input” renderer type, for the user name field of a logon form component that is nested inside a root component. The *id* attribute specifies the `UIComponent` instance, from within the response component tree, that this tag corresponds to, and may contain one of two types of values:

- The *compound identifier* of the component, as defined in Section FIXME/3.1.2.
- A *navigation expression* to be passed to the findComponent() method of the UIComponent instance for the JSF custom action that we are nested inside, as described in Section FIXME/3.1.4. In most cases, a navigation expression consisting of the *component identifier* of the desired component will return the desired results.

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

During the *Render Response* phase of the request processing lifecycle, the appropriate encoding methods of 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.

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 response 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 based (FIXME - does the `Renderer` act as a factory for this?) and added as a child component of the component represented by the component tag in which this tag 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 `TextEntry_Input` tag supports a `columns` attribute, which allows the page author to configure the number of characters in the rendered input field, like this:

```
<faces:textentry_input id="/logonForm/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 response 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:

```
<faces:UseFaces>
<faces:form_standard id="/logonForm"[1]>
<table border="0">
<tr>
      <td><faces:output_label id="/logonForm/usernameLabel"/></td>
      <td><faces:textentry_input id="/logonForm/username"
            modelReference="logonBean.username"/></td>
</tr>
<tr>
      <td><faces:output_label id="passwordLabel"/></td>
      <td><faces:textentry_secret id="password"
```

1. If the UIForm was itself the root node of the response component tree, the appropriate identifier would be "/" instead of "/logonForm", and the compound identifier of the username field would be "/username".

```
              modelReference="logonBean.password"/></td>
</tr>
<tr>
      <td><faces:command_submit id="/logonForm/submit"/></td>
      <td> &nbsp; </td>
</tr>
</table>
</faces:form_standard>
</Faces:UseFaces>
```

When absolute identifiers are specified for the `id` attribute, (i.e. "/logonForm/username" in the example above), component nesting on the page need not match the corresponding component tree, because the JSF implementation can identify the requested component directly. When a navigation expression is specified (i.e. "password" in the example above), however, the nesting must correspond, because the absolute identifier "/logonForm/password" will be computed for password component, based on the rules described in Section FIXME/3.1.4.

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

JSF.8.2.5 Registering Request Event Handlers and Validators

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 response component tree, without having to resort to Java code. For example:

```
<faces:textentry_input id="/logonForm/username"
      modelReference="logonBean.username">
      <faces:validator
       className="javax.faces.validator.RequiredValidator"/>
      <faces:validator
       className="javax.faces.validator.LengthValidator"/>
```

```
        <faces:attribute value="6"
         name="javax.faces.validator.LengthValidator.MINIMUM"/>
<faces:textentry_input>
```

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:

- 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 response 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 Request 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 response 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 Request 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 response 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 behave cause the JSF 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 response component tree for the current page.

JSF.8.3 The UseFaces 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 response component tree during the execution of its `doEndTag()` method. This tag must be defined in the tag library descriptor of the Custom Actions Tag Library described above, and the implementation class must conform to the following requirements:

- [FIXME - list requirements, including recognizing that it was used inside a page included via <jsp:include>, where the included component tree is grafted on to the one for the outer page]

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 Standard JSF Tag Library

FIXME - describe a standard tag library descriptor (with a well-known URI) that contains the “eventhandler”, “validator”, and “attribute” tags.

JSF.8.6 Standard UIComponent Custom Action Library

As described in Section FIXME, all JSF implementations are required to provide a standard `RenderKit` implementation (along with a library of associated `Renderers`) that applications can count on and still remain portable. In addition, all JSF implementations must also provide a library of UIComponent custom actions that correspond to this `RenderKit`. This library must satisfy the following requirements:

FIXME - specify requirements for the standard custom tag library that corresponds to the basic HTML `RenderKit`.

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.FacesContextFactory`* (`FactoryFinder.FACES_CONTEXT_FACTORY`) - Factory for `FacesContext` instances.
- *`javax.faces.context.LifecycleFactory`* (`FactoryFinder.LIFECYCLE_FACTORY`) - Factory for `Lifecycle` 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 response 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.