



JavaServer™ Faces Specification

Version 1.0, Expert Draft 20021223

December 23, 2002

Craig R. McClanahan, editor

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

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

Copyright 2002 Sun Microsystems, Inc.

4150 Network Circle, Santa Clara, California 95054, U.S.A
All rights reserved.

NOTICE

The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Sun Microsystems, Inc. ("Sun") and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this license and the Export Control and General Terms as set forth in Sun's website Legal Terms. By viewing, downloading or otherwise copying the Specification, you agree that you have read, understood, and will comply with all of the terms and conditions set forth herein.

Subject to the terms and conditions of this license, Sun hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense) under Sun's intellectual property rights to review the Specification internally for the purposes of evaluation only. Other than this limited license, you acquire no right, title or interest in or to the Specification or any other Sun intellectual property. The Specification contains the proprietary and confidential information of Sun and may only be used in accordance with the license terms set forth herein. This license will expire ninety (90) days from the date of Release listed above and will terminate immediately without notice from Sun if you fail to comply with any provision of this license. Upon termination, you must cease use of or destroy the Specification.

TRADEMARKS

No right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors is granted hereunder. Sun, Sun Microsystems, the Sun logo, Java, the Java Coffee Cup logo, J2EE, and JavaServer are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY SUN. SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Sun and its licensors from any claims based on your use of the Specification for any purposes other than those of internal evaluation, and from any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

(LFI#117738/Form ID#011801)

JavaServer™ Faces Specification i

JSF.1.Overview	12
JSF.1.1. Solving Practical Problems of the Web	12
JSF.1.2. Specification Audience	13
JSF.1.2.1. Page Authors	13
JSF.1.2.2. Component Writers	14
JSF.1.2.3. Application Developers	15
JSF.1.2.4. Tool Providers	15
JSF.1.2.5. JSF Implementors	16
JSF.1.3. Introduction to JSF APIs	16
JSF.1.4. Using the JSF APIs	16
JSF.2.Request Processing Lifecycle	18
JSF.2.1. Reconstitute Request Tree	19
JSF.2.2. Apply Request Values	20
JSF.2.3. Handle Request Events	20
JSF.2.4. Process Validations	20
JSF.2.5. Update Model Values	21
JSF.2.6. Invoke Application	21
JSF.2.7. Render Response	22
JSF.3.User Interface Component Model	24
JSF.3.1. UIComponent	24
JSF.3.1.1. Component Type	25
JSF.3.1.2. Component Identifiers	25
JSF.3.1.3. Client Identifiers	25
JSF.3.1.4. Component Tree Manipulation	26
JSF.3.1.5. Component Tree Navigation	27
JSF.3.1.6. Model Object References	27
JSF.3.1.7. Local Values	27
JSF.3.1.8. Generic Attributes	27
JSF.3.1.9. Request Event Processing	28
JSF.3.1.10. Update Model Values Processing	29
JSF.3.1.11. Validation Processing	29
JSF.3.1.12. Decoding and Encoding	31
JSF.3.1.13. Facet Management	33
JSF.3.1.14. Lifecycle Phase Handler Methods	34
JSF.3.2. NamingContainer and UINamingContainer	35
JSF.3.3. Tree	36
JSF.3.4. TreeFactory	36

JSF.3.5.	Event Classes	37
JSF.3.5.1.	FacesEvent, RequestEvent, and ApplicationEvent	37
JSF.3.5.2.	Standard FacesEvent Implementations	38
JSF.3.5.2.1.	CommandEvent	38
JSF.3.5.2.2.	FormEvent	38
JSF.3.6.	Validator Classes	39
JSF.3.6.1.	Validator	39
JSF.3.6.2.	Standard Validator Implementations	39
JSF.3.6.2.1.	DoubleRangeValidator	40
JSF.3.6.2.2.	LengthValidator	41
JSF.3.6.2.3.	LongRangeValidator	42
JSF.3.6.2.4.	RequiredValidator	42
JSF.3.6.2.5.	StringRangeValidator	43
JSF.4.	Standard User Interface Components	44
JSF.4.1.	UICommand	45
JSF.4.2.	UIForm	45
JSF.4.3.	UIGraphic	46
JSF.4.4.	UIInput	46
JSF.4.5.	UIOutput	47
JSF.4.6.	UIPanel	47
JSF.4.7.	UIParameter	47
JSF.4.8.	UISelectBoolean	48
JSF.4.9.	SelectItem	48
JSF.4.10.	UISelectItem	48
JSF.4.11.	UISelectItems	49
JSF.4.12.	UISelectMany	49
JSF.4.13.	UISelectOne	50
JSF.4.14.	UITextEntry	50
JSF.5.	Per-Request State Information	52
JSF.5.1.	FacesContext	52
JSF.5.1.1.	Servlet API Components	52
JSF.5.1.2.	Locale	53
JSF.5.1.3.	Request and Response Trees	53
JSF.5.1.4.	Application Events Queue	53
JSF.5.1.5.	Message Queues	54
JSF.5.1.6.	Lifecycle Management Objects	54
JSF.5.1.7.	Model Reference Expression Evaluation	54
JSF.5.1.8.	Request Events Queue	55

JSF.5.1.9. ResponseStream and ResponseWriter	55
JSF.5.2. Message	56
JSF.5.3. MessageImpl	56
JSF.5.4. ResponseStream	56
JSF.5.5. ResponseWriter	57
JSF.5.6. MessageResources	58
JSF.5.7. MessageResourcesFactory	59
JSF.5.8. FacesContextFactory	60
JSF.6.Lifecycle Management	62
JSF.6.1. Lifecycle	62
JSF.6.2. Phase	63
JSF.6.3. ApplicationHandler	64
JSF.6.4. ViewHandler	64
JSF.6.5. LifecycleFactory	65
JSF.7.Rendering Model	66
JSF.7.1. RenderKit	66
JSF.7.2. Renderer	67
JSF.7.3. RenderKitFactory	69
JSF.7.4. Converter	70
JSF.7.5. ConverterFactory	71
JSF.7.6. Standard HTML RenderKit Implementation	71
JSF.7.6.1. Renderer Types for UICommand Components ...	74
JSF.7.6.2. Renderer Types for UIForm Components	75
JSF.7.6.3. Renderer Types for UIGraphic Components	76
JSF.7.6.4. Renderer Types for UIInput Components	77
JSF.7.6.5. Renderer Types for UIOutput Components	79
JSF.7.6.6. Renderer Types for UIPanel Components	81
JSF.7.6.7. Renderer Types for UISelectBoolean Components	82
JSF.7.6.8. Renderer Types for UISelectMany Components ..	83
JSF.7.6.9. Renderer Types for UISelectOne Components ...	84
JSF.8.Integration With JSP	86
JSF.8.1. UIComponent Custom Actions	86
JSF.8.2. Using UIComponent Custom Actions in JSP Pages ...	88
JSF.8.2.1. Declaring the Custom Actions Tag Library	88
JSF.8.2.2. Defining Components in a Page	88
JSF.8.2.3. Creating Components and Overriding Attributes ..	89
JSF.8.2.4. Representing Component Hierarchies	89
JSF.8.2.5. Registering Request Event Handlers and Validators	90

JSF.8.2.6.	Interoperability with Other Custom Action Libraries	91
JSF.8.2.7.	Composing Pages from Multiple Sources	92
JSF.8.3.	The <f:use_faces/> Tag	92
JSF.8.4.	UIComponent Custom Action Implementation Requirements	93
JSF.8.5.	JSF Core Tag Library	93
JSF.8.5.1.	<f:attribute>	95
JSF.8.5.2.	<f:event_handler>	96
JSF.8.5.3.	<f:parameter>	97
JSF.8.5.4.	<f:use_faces>	99
JSF.8.5.5.	<f:validate_doublerange>	100
JSF.8.5.6.	<f:validate_length>	101
JSF.8.5.7.	<f:validate_longrange>	102
JSF.8.5.8.	<f:validate_required>	103
JSF.8.5.9.	<f:validate_stringrange>	104
JSF.8.5.10.	<f:validator>	105
JSF.8.6.	Standard HTML RenderKit Tag Library	106
JSF.9.	Using JSF In Web Applications	108
JSF.9.1.	Web Application Deployment Descriptor	108
JSF.9.1.1.	Servlet Definition	109
JSF.9.1.2.	Servlet Mapping	109
JSF.9.1.3.	Application Configuration Parameters	109
JSF.9.2.	Included Classes and Resources	110
JSF.9.2.1.	Application-Specific Classes and Resources	110
JSF.9.2.2.	Servlet and JSP API Classes (javax.servlet.*)	110
JSF.9.2.3.	JavaServer Faces API Classes (javax.faces.*)	110
JSF.9.2.4.	JavaServer Faces Implementation Classes	111
JSF.9.2.4.1.	FactoryFinder	111
JSF.9.2.4.2.	FacesServlet	112
JSF.9.2.4.3.	FacesTag	113
JSF.9.2.4.4.	FacesBodyTag	113
JSF.9.3.	Included Configuration Files	113

Preface

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

Other JavaTM Platform Specifications

JSF is based on the following Java API specifications:

- JavaServer PagesTM Specification, version 1.2 (JSPTM)
<<http://java.sun.com/products/jsp/>>
- JavaTM Servlet Specification, version 2.3 (Servlet)
<<http://java.sun.com/products/servlet/>>
- JavaTM 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 PagesTM Standard Tag Library, version 1.0 (JSTL)
<<http://java.sun.com/products/jsp/jstl/>>
- Portlet Specification, under development in JSR-168
<<http://www.jcp.org/jsr/detail/168.jsp>>

Related Documents and Specifications

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

- Hypertext Markup Language (HTML), version 4.01
<<http://www.w3.org/TR/html4/>>

- Extensible HyperText Markup Language (XHTML), version 1.0
<<http://www.w3.org/TR/xhtml1>>
- Extensible Markup Language (XML), version 1.0 (Second Edition)
<<http://www.w3.org/TR/REC-xml>>
- XForms 1.0 (currently in Working Draft state)
<<http://www.w3.orgTR/xforms/>>

Providing Feedback

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

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

Acknowledgements

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

JSF.1

Overview

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

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

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

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

JSF.1.1 Solving Practical Problems of the Web

JSF’s core architecture is designed to be independent of specific protocols and markup; however it is aimed directly at solving many of the common problems encountered when writing applications for HTML clients which communicate via HTTP to a Java application server that supports servlets and JavaServer Pages (JSP)

based applications. These applications are typically form-based, comprised of one or more HTML pages which the user interacts with to complete a task or set of tasks. JSF tackles the following challenges associated with these applications:

- managing UI component state across requests
- supporting encapsulation of the differences in markup across different browsers and clients
- supporting form processing (multi-page, more than one per page, etc)
- providing a strongly typed event model which allows the application to write server-side handlers (independent of HTTP) for client generated events
- validating request data and providing appropriate error reporting
- enabling type conversion when migrating component values (Strings) to/from application data objects (often not Strings)
- handling error and exceptions, and reporting errors in human-readable form back to the application user

JSF.1.2 Specification Audience

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

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 represented as JSP custom actions, which will support configuring the attributes of those components

as custom action attributes in the JSP page. In addition, the pages produced by a page author will be the used by the JSF framework to create component tree hierarchies that represent the components on those pages.

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

JSF.1.2.2 Component Writers

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

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

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

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

- A page author creating a custom “widget” for use on a particular page
- An application developer providing components that correspond to specific data objects in the application’s business domain
- A specialized team within a larger development group responsible for creating standardized components for reuse across applications
- Third party library and framework providers creating component libraries that are portable across JSF implementations
- Tool providers whose tools can leverage the specific capabilities of those libraries in development of JSF-based applications

- JSF implementors who provide implementation-specific component libraries as part of their JSF product suite

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

JSF.1.2.3 Application Developers

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

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

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

- Expose the underlying data required by the user interface layer as objects that are accessible from the web tier (such as via request or session attributes in the Servlet API), via *model reference expressions*, as described in Section FIXME
- Provide application-level event handlers for the *Command Events* and *Form Events* that are enqueued by JSF components during the request processing lifecycle, as described in Section FIXME

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

JSF.1.2.4 Tool Providers

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

Examples of possible tools include:

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

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

JSF.1.2.5 JSF Implementors

Finally, *JSF implementors* will provide runtime environments that implement all of the requirements described in this specification. Typically, a JSF implementor will be the provider of a Java2 Enterprise Edition (J2EE) application server, although it is also possible to provide a JSF implementation that is portable across J2EE servers.

Advanced features of the JSF APIs (such as the Lifecycle and Phase APIs described in Chapter FIXME/6) allow JSF implementors, as well as application developers, to customize and extend the basic functionality of JSF in a portable way. These features provide a rich environment for server vendors to compete on features and quality of service aspects of their implementations, while maximizing the portability of JSF-based applications across different JSF implementations.

JSF.1.3 Introduction to JSF APIs

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

FIXME - add summary of each package here.

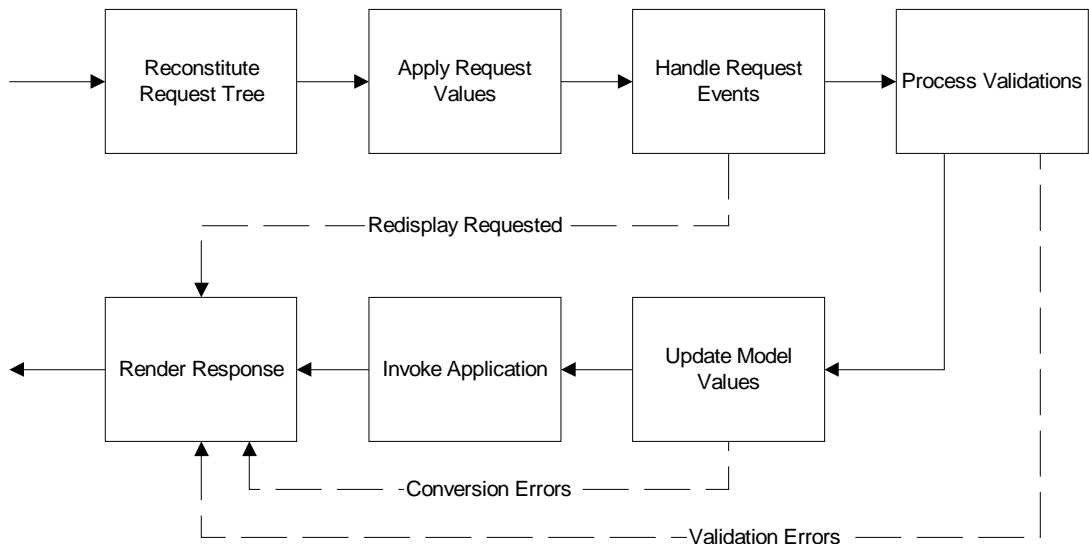
JSF.1.4 Using the JSF APIs

FIXME - some simple examples of using JSF.

JSF.2

Request Processing Lifecycle

Each servlet request 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). In order to be utilized, such state information must meet the following requirements:
 - It must have a tree identifier that corresponds to the request URI used to submit this request. In other words, if the JSF implementation has saved state information for a tree identifier `/foo` in the user's session, but receives a request with URI `/faces/bar`, the saved state information will be thrown away.
- If such state information is available, use it to construct a component `Tree`, including any required wire-up of event 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()` method¹.
- If no such state information is available, or if any saved state information was for a different page than the one processing this request, acquire a component `Tree` for the requested tree identifier, by calling the `getTree()` method of the `TreeFactory` instance for this web application, and store it in the `FacesContext` by calling `setRequestTree()`. The tree identifier for this tree is extracted from the *extra path info* portion of the request URI. This means that, when using the standard servlet mapping of `/faces/*` to `javax.faces.webapp.FacesServlet`, a request URI `/faces/menu.jsp` will select tree identifier `/menu.jsp`.
- Call the `setLocale()` method of the `FacesContext` instance for the current request, passing a `Locale` instance derived from the saved state information (if any); otherwise, acquire the default `Locale` to be used by calling the `getLocale()` method of the `ServletRequest` for this request.

At the end of this phase, the `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).

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

JSF.2.2 Apply Request Values

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

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

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.

If the called `processDecodes()` method returned `false`, control must be transferred to the *Render Response* phase of the request processing lifecycle. Otherwise, control will proceed to the *Handle Request Events* phase.

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 of the current component (such as clicking a node in a tree control to expand or contract it), or for some other component in the request component tree (such as clicking a checkbox causing a change in the set of options to be shown in a combobox list), and client device limitations dictate that such events must be processed in the server. 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, the JSF implementation must call the `processEvents()` method of the root component in the request component tree. This will normally cause the `processEvents()` method of each component in the tree to be called recursively, as described in the Javadocs for the `UIComponent.processEvents()` method.

If the called `processEvents()` method returned `false`, control must be transferred to the *Render Response* phase of the request processing lifecycle. Otherwise, control will proceed to the *Process Validations* phase.

JSF.2.4 Process Validations

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

During the *Process Validations* phase of the request processing lifecycle, the JSF implementation must call the `processValidators()` method of the root component in the request component tree. This will normally cause the `processValidators()` method of each component in the tree to be called recursively, as described in the Javadocs for the `UIComponent.processValidators()` method.

If the called `processValidators()` method returned `false`, control must be transferred to the *Render Response* phase of the request processing lifecycle. 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 now appropriate to update the application's model data in preparation for performing any application events that have been enqueued.

During the *Update Model Values* phase, the JSF implementation must call the `processUpdates()` method of the root component in the request component tree. This will normally cause the `processUpdates()` method of each component in the tree to be called recursively, as described in the Javadocs for the `UIComponent.updateModel()` method.

If the called `processUpdates()` method returned `false`, control must be transferred to the *Render Response* phase of the request processing lifecycle. Otherwise, control will proceed to the *Invoke Application* phase.

JSF.2.6 Invoke Application

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

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

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

- 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`.
- Enqueue one or more additional application events to be handled during this phase.
- 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.

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 `Renderers`, 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 order¹.
- During the rendering process, additional components may be added to the response component tree based on information available to the `ViewHandler` implementation². 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.
- 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, calls to its `encodeXXX()` methods must be performed in the manner described in Section [FIXME/3.1.11](#).

Upon completion of rendering, the completed state of the response 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?]

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](#).

[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 anonymous or they can be given a *component identifier* by the framework user. Components in the tree can be located based on *component identifiers*, which must be unique within the scope of the nearest ancestor to the component that is a *naming container*.

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.

JSF3.1.1 Component Type

- `public abstract String getComponentType();`

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

JSF3.1.2 Component Identifiers

- `public String getComponentId();`
- `public void setComponentId(String componentId);`

Every component can be named by a *component identifier*, which must be unique among the components that share a common *naming container* parent in a component tree. For maximum portability, component identifiers must conform to the following rules:

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

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

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

JSF3.1.3 Client Identifiers

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

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

- to serve as anchors for client side accessibility labels

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

- `public String getClientId(FacesContext context);`

The client identifier is obtained by executing the following algorithm:

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

JSF.3.1.4 Component Tree Manipulation

- `public UIComponent getParent();`

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

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

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

- `public void clearChildren();`

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

- `public boolean containsChild(UIComponent component);`

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

- `public UIComponent getChild(int index);`

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

- `public int getChildCount();`

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

- `public Iterator getChildren();`

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

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

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

JSF.3.1.5 Component Tree Navigation

- `public UIComponent findComponent(String expression);`

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

JSF.3.1.6 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.3.1.7 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.3.1.8 Generic Attributes

- `public Object getAttribute(String name);`
- `public Iterator getAttributeNames();`

```
■ public void setAttribute(String name, Object value);
```

The render-independent characteristics of components are generally represented as JavaBean properties with getter and setter methods. 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 `Renderers`.
- Configuration parameters to be utilized by `Converters`, `RequestEventHandlers`, and `Validators` associated with the component.
- General purpose association of application-specific objects with components.

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

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.3.1.9 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 `RequestEventHandlers` can be associated with each component in the tree.

```
■ public boolean processEvent(FacesContext context,  
    RequestEvent 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 `false` 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 `RequestEvents`, and Section [FIXME/2.4](#) for further details on how and when the event processing methods are called by the JSF implementation.

JSE3.1.10 Update Model Values Processing

```
■ public boolean 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.
- If the current value of the `valid` property of this component is `false`, 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 or updating error occurs, enqueue an appropriate `Message` by calling the `addMessage()` method on the `FacesContext` instance for the current request, and set the `valid` property of this component to `false`, and return `false`.
- If updating was successful, set the `valid` property of this component to `true`, and return `true`.

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.3.1.11 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 is executed, zero or more `Validators` can be associated with each component in the tree.

```
■ public boolean validate(FacesContext context);
```

This method performs any correctness checks on a component's local value that have been defined within the component class itself. If all such correctness checks are satisfied, this method must set the `valid` property of this component to `true`, and return `true` to the caller. Otherwise, this method must (a) enqueue one or more `Messages` describing the failures, by calling an appropriate `addMessage()` method on the `FacesContext` instance for the current request; (b) set the `valid` property of this component to `false`; and (c) return `false` to the caller.

```
■ public boolean processValidators(FacesContext context);
```

This method is called on each component in the request component tree, during the *Process Validations* phase of the request processing lifecycle. It must perform the following processing:

- Skip validation processing if the current `valid` property of this component is `false`.
- Call the `validate()` method on this component to perform any self-validation that has been defined.
- Call the `validate()` method on each registered `Validator` instance for this component.
- If the call to the `validate()` method of this component (or any call to the `validate()` method of a registered `Validator`) returned `false`, set the `valid` property of this component to `false` and return `false` to the caller.
- Otherwise, return `true` to the caller, indicating that all validations of this component were successful.

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

- `public boolean isValid();`

The `isValid()` method provides an inexpensive way for external logic to determine whether the local value of the component is appropriate, based on the current request processing lifecycle being performed. Specifically, the `valid` property of a component is adjusted at the following points:

- During the *Apply Request Values* phase, the `valid` property will be set to `true` or `false` by the `decode()` method, based on the success or failure of decoding a new local value for this component from the current request.
- During the *Process Validations* phase, the `valid` property will be set to `false` by the `processValidators()` method, if any `validate()` method that processed this component returned `false`.
- During the *Update Model Values* phase, the `valid` property will be set to `false` by the `updateModel()` method if a problem performing the update has occurred.

JSF3.1.12 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 `encodeXXXX()` 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/2.2` and `FIXME/2.7` for more information on how and when the decoding and encoding methods of the component, or the associated `Renderer`, are called.

```
■ public boolean decode(FacesContext context) throws  
    IOException;
```

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

- If this component has a `rendererType` property value set, acquire an instance of the corresponding `Renderer` from the `RenderKit` associated with the request tree, and delegate decoding to it by calling the `decode()` method, and returning the value that the `Renderer` returns.
- If this component does not have a `rendererType` property value set, perform any required decoding directly, as follows:
 - 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, set the `valid` property to `true`, and return `true` to the caller.
 - If conversion is unsuccessful, save enough state information for the corresponding `encodeXxx()` methods to reproduce the incorrect value¹, and set the `valid` property to `false`. In addition, one or more validation error messages should be registered, by calling the `addMessage()` method on the `FacesContext` instance for this request, describing the nature of the conversion failure. Finally, return `false` to the caller to indicate that a conversion error occurred.
 - Optionally, enqueue one or more `RequestEvent` events, on either the current component or any other component in the request component tree, by calling the `addRequestEvent()` method on the `FacesContext` instance for this request. These events will be processed by the destination components, during the *Handle Request Events* phase of the request processing lifecycle.
 - Optionally, enqueue one or more `CommandEvent` or `FormEvent` events, by calling the `addApplicationEvent()` method on the `FacesContext` instance for this request. These events will be processed by the application, during the *Invoke Application* phase of the request processing lifecycle.

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();
```

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.

- `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 managing the encoding of 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 nested inside elements rendered by the parent component¹. 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, in the following order:

- The `encodeBegin()` method of this component will be called. For components that generate nested markup elements, this method will generally render the beginning tag (such as `<form>`). Components must consult the `isRendered()` method to determine if they should actually output any content.
- If the `rendersChildren` property of this component is `true`, the `encodeChildren()` method of this component will be called. This method has the responsibility of performing whatever rendering is required for all descendant components.
- If the `rendersChildren` property of this component is `false`, all child components will be recursively rendered following the same algorithm as is described for this component.
- The `encodeEnd()` method of this component will be called. For components that generate nested markup elements, this method will generally render the ending tag (such as `</form>`). Components must consult the `isRendered()` method to determine if they should actually output any content.
-

For components that do not support a notion of nested markup elements, all of the rendering should, by convention, be placed in the `encodeEnd()` method.

The default implementations in `UIComponentBase` delegate to the appropriate `Renderer` (if `rendererType` is not null); otherwise they do nothing.

- `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.

```
■ public void setRendered(boolean rendered);  
■ public boolean isRendered();
```

The “rendered” property of a component determines whether or not the component should be rendered to the client device. Setting this property to `false` will cause the component not to be rendered.

JSF.3.1.13 Facet Management

JavaServer Faces supports a traditional model of composing complex components out of simple components via parent-child relationships that organize the entire set of components into a tree, as described in Section [FIXME/3.1.3](#). However, an additional useful facility would be to define particular subordinate components that have a specific *role* with respect to the owning component, which is typically independent of the parent-child relationship. An example might be a “data grid” control, where the children represent the columns to be rendered in the grid. It is useful to be able to identify a component that represents the column header and/or footer, separate from the usual child collection that represents the column data.

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

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

See the API Javadocs in Section [FIXME](#) for more details about the functionality of these methods.

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

```
public UIHeader getHeader() {
```

```

        return ((UIHeader) getFacet("header"));
    }

    public void setHeader(UIHeader header) {
        addFacet("header", header);
    }

```

JSF.3.1.14 Lifecycle Phase Handler Methods

The following methods are called, on the root node of the request component tree, by the *Apply Request Values*, *Handle Request Events*, *Process Validations*, and *Update Model Values* phases of the request processing lifecycle, as described in Chapter *FIXME/2*. The default implementation of these methods (as described in the corresponding Javadocs) performs a recursive walk over the facets and children of the root node, resulting in a call to the component's own handling method for that phase.

- `public boolean processDecodes(FacesContext context) throws IOException;`
- `public boolean processEvents(FacesContext context);`
- `public boolean processValidators(FacesContext context);`
- `public boolean processUpdates(FacesContext context);`

Advanced `UIComponent` implementations may override these methods to specialize how facets and children of a particular component are handled. Components that wish to specialize only their own processing for these lifecycle phases, without affecting how the component tree is recursively processed, should override the `decode()`, `processEvent()`, `validate()`, or `updateModel()` method instead.

JSF.3.2 NamingContainer and UINamingContainer

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

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

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

```
■ public String generateClientId();
```

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

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

Consider the following usage:

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

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

JSF.3.3 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 implementation 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 response¹. This is performed by selecting an appropriate tree identifier, and calling the `getTree()` method of the `TreeFactory` instance for this web application.

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

JSF.3.4 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:

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

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

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

JSF.3.5 Event Classes

JSF.3.5.1 FacesEvent, RequestEvent, and ApplicationEvent

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

- A `RequestEvent` can be enqueued 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.
- An `ApplicationEvent` can be enqueued 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 `RequestEvent` or `ApplicationEvent`, which are subclasses of the base class `FacesEvent`. Standard subclasses of `ApplicationEvent` are provided for the types of events used by an application. Component writers will generally define custom `RequestEvent` 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 and application developers. `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.3.5.2 Standard FacesEvent Implementations

JSF.3.5.2.1 CommandEvent

`CommandEvent` is a concrete subclass of `ApplicationEvent` 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.3.5.2.2 FormEvent

`FormEvent` is a concrete subclass of `ApplicationEvent` 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.3.6 Validator Classes

JSF.3.6.1 Validator

A `Validator` is an interface implemented by classes that can perform validation (i.e., correctness checks) on a `UIComponent`. Zero or more `Validators` 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 boolean 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, and this method must return `false` to the caller. If no rules violation is found, this method returns `true` to the caller.

JSF.3.6.2 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 `Validators` are being configured programmatically.
- Standard `Validators` must attempt to convert attribute values for configuration parameters that they recognize into the internal data types required by the `Validator` implementation. For example, the standard `LengthValidator` (see Section [FIXME/3.5.2.3](#)) accepts attributes to specify the minimum and maximum length of a character string. This validator must accept values of any Java primitive wrapper type (such as `java.lang.Integer` or `java.lang.Double`) or `java.lang.String` (which will be typical when the attribute's value is set from a JSP page) into an internal `int` value.
- Errors encountered during the processing of configuration parameters (such as conversion failure, or a missing required parameter) should cause the addition of an appropriate `Message` instance via `FacesContext.addMessage()`, rather than throwing an exception. The content of the message should describe the configuration problem that was encountered.
- To support internationalization, `Message` instances should be created by passing a *message identifier* (along with optional substitution parameters) to an appropriate `MessageResources` instance. The message identifiers for such standard messages are also defined by manifest `String` constants in the implementation classes.
- Unless otherwise specified, components with a `null` local value cause the validation checking by this `Validator` to be skipped. If a component should be required to have a non-`null` value, a separate instance of `RequiredValidator` should be registered in order to enforce this rule.

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

JSF.3.6.2.1 DoubleRangeValidator

DoubleRangeValidator checks the local value of a component, which must be of any numeric 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 Messages 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.3.6.2.2 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.3.6.2.3 LongRangeValidator

LongRangeValidator checks the local value of a component, which must be of any numeric type convertible to long (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 Messages 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.3.6.2.4 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.3.6.2.5 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 client identifier of this component, with a value of “submit”, enqueue a `FormEvent` to the application, passing the form name of the form being submitted and the command name of this `UICommand`.
- `encodeBegin()` - Render the beginning of an HTML submit button, using the client identifier of this component as the “name” attribute, and “submit” as the value to be returned to the server if this button is selected by the user.
- `encodeEnd()` - Render the ending of an HTML submit button.

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

JSF.4.3 UIGraphic

The `UIGraphic` class 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 `` element with a `src` element based on the local value of this component. URL rewriting will have been applied, to maintain session identity in the absence of cookies.

JSF.4.4 UIInput

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

The default rendering functionality for `UIInput` components is:

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

JSF.4.5 UIOutput

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

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

JSF.4.6 UIPanel

The `UIPanel` class represents a user interface component that is primarily a container for its children. The default implementation sets its `rendersChildren` property to `true`, but has no rendering behavior of its own. Subclasses of `UIPanel` will typically implement layout management for child components, either directly (via its `encodeXxx()` methods) or indirectly (via implementation in an appropriate `Renderer`). The latter scenario -- delegating rendering to a `Renderer` -- means that `UIPanel` can be used as a component implementation class for most layout management purposes, without the need to subclass.

JSF.4.7 UIParameter

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

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

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

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

This component has no default rendering functionality.

JSF.4.8 UISelectBoolean

The `UISelectBoolean` class 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.4.9 SelectItem

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

```
■ SelectItem(Object value, String label, String description);
```

- `public String getDescription();`
- `public String getLabel();`
- `public Object getValue();`

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

JSF.4.10 UISelectItem

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

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

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

JSF.4.11 UISelectItems

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

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

- Single instance of `SelectItem` -- This instance is added to the set of available options.
- Array of `SelectItem` -- Each instance in this array is added to the set of available options, in ascending order.

- Collection of `SelectItem`¹ -- Each element in this Collection will be added to the set of available options, in the order that an iterator over the Collection returns them.

JSF.4.12 UISelectMany

The `UISelectMany` class 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 `selectedValues` 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.4.13 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);`

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

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.4.14 `UITextEntry`

(Replaced by `UIInput`).

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 encapsulate this contextual information.

JSF.5.1 FacesContext

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

JSF.5.1.1 Servlet API Components

```
■ 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 `getSession()` method will only return a non-null session instance if the current request is actually an `HttpServletRequest`, and a session is already in existence. To create and acquire a reference to a new session, call the standard `HttpServletRequest.getSession()` or `HttpServletRequest.getSession(true)` method.

JSF.5.1.2 **Locale**

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

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

JSF.5.1.3 **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 `Messages`, either those associated with the specified `UIComponent`, or those associated with no `UIComponent` if the parameter is `null`. The `getMessages()` method returns an `Iterator` over all queued `Messages`, whether or not they are associated with a particular `UIComponent`.

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

JSF.5.1.6 Lifecycle Management Objects

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

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

JSF.5.1.7 Model Reference Expression Evaluation

```
■ public Class getModelType(String modelReference);
■ public Object getModelValue(String modelReference);
```

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

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

JSF.5.1.8 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. `UIComponents` or `Renderers` that wish to create output in a binary format should call `getResponseStream()` to acquire a stream capable of binary output. Correspondingly, `UIComponents` or `Renderers` 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 localized¹ Message instance for the specified message identifier, optionally modified by substitution parameters² 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. Typical 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 implementors-- applications will not generally call it directly. The factory instance can be acquired, by JSF implementations or by application code, by executing:

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

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

```
■ public FacesContext getFacesContext(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](#).

- `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](#). This method must execute all `Phase` instances registered for this `Lifecycle` instance, in

the order described in Chapter FIXME/2. Execution of each phase must be accomplished by a call to the `executePhase()` method, which returns a *state change indicator* value describing the action that should be taken next.

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

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 implementation 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 corresponding 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 redispatch 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 application'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 `Renderers` 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 `Renderers` would return `true` from a call to a `supportsComponentType()` method). However, `RenderKit` instances are encouraged to explicitly enumerate the component classes that are associated with its `Renderers`, 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 registered `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 `Renderers` 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 `Renderers` to register them with the standard `RenderKit` provided by the JSF implementation, at application startup time.

## JSF.7.2 **Renderer**

A `Renderer` instance implements the decoding and encoding functionality of components, during the *Apply Request Values* and *Render Response* phases of the request processing lifecycle, when the component has a non-null value for the `rendererType` property.

```
■ public boolean decode(FacesContext context, UIComponent
 component) throws IOException;
```

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

- Extract from the incoming request (typically from parameters, headers, and/or cookies) the data representing the new value for this component.
- Attempt to convert this data into an object of the appropriate type for this component.
- If conversion is successful, save the converted object as the local value of this component, and return `true` to the caller.
- If conversion is unsuccessful, save enough state information for the corresponding `encodeXxx()` methods to reproduce the incorrect value, enqueue one or more error messages by calling the `addMessage()` method of the `FacesContext` instance for this request, and return `false` to the caller.
- Optionally, enqueue one or more `FacesEvent` events, on either the current component or any other component in the request component tree, by calling the `addRequestEvent()` method on the `FacesContext` instance for this request. These events will be processed by the destination components, during the *Handle Request Events* phase of the request processing lifecycle.
- Optionally, enqueue one or more `CommandEvent` or `FormEvent` events, by calling the `addApplicationEvent()` method on the `FacesContext` instance for the current request. These events will be processed by the application, during the *Invoke Application* phase of the request processing lifecycle.
- `public void encodeBegin(FacesContext context, UIComponent component) throws IOException;`
- `public void encodeChildren(FacesContext context, UIComponent component) throws IOException;`
- `public void encodeEnd(FacesContext context, UIComponent component) throws IOException;`

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

For components utilizing the *delegated implementation* programming model, these methods will be called during the *Render Response* phase of the request processing lifecycle. These methods have the same responsibilities as the corresponding

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

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

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

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

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

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

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

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

## JSF.7.3 RenderKitFactory

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

```
RenderKitFactory factory = (RenderKitFactory)
```

```
 FactoryFinder.getFactory(FactoryFinder.RENDER_KIT_FACTORY);
```

The `RenderKitFactory` implementation class supports the following methods:

- `public RenderKit getRenderKit(String renderKitId);`
- `public RenderKit getRenderKit(String renderKitId, FacesContext context);`

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

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

`RenderKitFactory.DEFAULT_RENDER_KIT`. Additional render kit identifiers, and corresponding instances, can also be made available.

```
■ public Iterator getRenderKitIds();
```

This method returns an `Iterator` over the set of render kit identifiers supported by this factory. This set must include the value specified by

`RenderKitFactory.DEFAULT_RENDER_KIT`.

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

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

## JSF.7.4 Converter

A `javax.faces.convert.Converter` is a specialized class that can perform String-to-Object conversions (during the *Apply Request Values* phase of the request processing lifecycle) and Object-to-String conversions (during the *Render Response* phase of the request processing lifecycle). This allows `Renderer` instances that know how to deal with the representation of model data objects as a single String (a very common use case) to perform conversions to and from arbitrary (often application-specific) model data types, without requiring the creation of specialized `Renderers` for each data type.

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

Typically, a `Converter` will be a stateless object that requires no extra configuration information to perform its responsibilities. However, in some cases it is useful to provide configuration parameters to the converter (such as a `java.text.DateFormat` pattern for a `Converter` that supports `java.util.Date` model objects). Such configuration information may be provided via named attributes of the `UIComponent` instance that is passed to the `Converter`'s methods. The attribute names used by a `Converter` to configure its

behavior should be documented in the usage instructions for that `Converter`, and should be selected to not conflict with other render-independent and render-dependent attributes of the underlying component.

`Converter` instances should be programmed so that the conversions they perform are symmetric. In other words, if a model data object is converted to a `String` (via a call to the `getAsString()` method), it should be possible to call `getAsObject()` and pass this converted `String` as the value parameter, and retrieve a model data object that is semantically equivalent to the original one.

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

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

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

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

## JSF.7.5 ConverterFactory

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

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

The `ConverterFactory` implementation class supports the following methods:

- `public Converter getConverter(String converterId)`;

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

- `public Iterator getConverterIds()`;

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

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

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

## JSF.7.6 Standard HTML RenderKit Implementation

To ensure application portability, all JSF implementations are required to include support for a `RenderKit`, and the associated `Renderers`, that meet the requirements defined in this section, to generate textual markup that is compatible with HTML/4.01. JSF implementors, and other parties, may also provide additional `RenderKit` libraries, or additional `Renderers` that are added to the standard `RenderKit` at application startup time, but applications must ensure that the standard `Renderers` are made available for the web application to utilize them.

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

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

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

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

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

**TABLE 7-1** Commonly Used `Renderer` Attributes

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

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

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

## JSF.7.6.1      Renderer Types for UICommand Components

TABLE 7-1      Renderer Types for UICommand Components

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

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

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



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

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

## JSF.7.6.2      **Renderer Types for UIForm Components**

**TABLE 7-1**      **Renderer Types for UIForm Components**

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

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

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

`UIForm` component with `formName="logonForm"`

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

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

### JSF.7.6.3      Renderer Types for UIGraphic Components

TABLE 7-1    Renderer Types for UIGraphic Components

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

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

## JSF.7.6.4      **Renderer Types for UIInput Components**

**TABLE 7-1      Renderer Types for UIInput Components**

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

**TABLE 7-1**    **Renderer Types for UIInput Components**

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

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

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

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

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

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

## JSE.7.6.5      **Renderer Types for UIOutput Components**

**TABLE 7-1**    **Renderer Types for UIOutput Components**

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

**TABLE 7-1**    **Renderer Types for UIOutput Components**

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

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

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

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

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

## JSF.7.6.6      **Renderer Types for UIPanel Components**

**TABLE 7-1      Renderer Types for UIPanel Components**

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

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

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

### JSF.7.6.7      **Renderer Types for UISelectBoolean Components**

**TABLE 7-1      Renderer Types for UISelectBoolean Components**

| Renderer Type | Data Type | Render Dependent Attributes | Decode Behavior                 | Encode Behavior                                                                                                 |
|---------------|-----------|-----------------------------|---------------------------------|-----------------------------------------------------------------------------------------------------------------|
| Checkbox      | boolean   |                             | Convert as necessary and store. | Get current value, convert to boolean, render as an HTML <code>&lt;input&gt;</code> element of type “checkbox”. |

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



## JSF.7.6.8      **Renderer Types for UISelectMany Components**

**TABLE 7-1**    **Renderer Types for UISelectMany Components**

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

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

The set of available choices is constructed by scanning the child components for one or more instances of `UISelectItem` (see Section [FIXME/4.10](#)) or `UISelectItems` (see Section [FIXME/4.11](#)) components. Existence of these child components causes the addition of a single `SelectItem` instance (for a `UISelectItem`) or a set of `SelectItem` instances (for a `UISelectItems`) to the set of available choices. For

each available choice, the `itemLabel` property defines the visual appearance of this choice, while the `itemValue` property defines the value that will be added to the array of current selections for this component.

## JSF.7.6.9 Renderer Types for UISelectOne Components

TABLE 7-1 Renderer Types for UISelectOne Components

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

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

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

# JSF.8

## Integration With JSP

JSF supports (but does not require) using JavaServer Pages (JSP) as the page description language for JSF pages. This JSP support is provided by providing custom actions so that a JSF user interface can be easily defined in a JSP page by adding tags corresponding to JSF UI components. A page author should be able to use JSF components in conjunction with the other custom actions (including the JSP Standard Tag Library), as well as standard HTML content and layout embedded in the page

### JSF.8.1 UIComponent Custom Actions

A JSP custom action for a JSF `UIComponent` is constructed by combining properties and attributes of a Java UI component class with the rendering attributes supported by a specific `Renderer` from a concrete `RenderKit`. For example, assume the existence of a concrete `RenderKit`, `HTMLRenderKit`, which supports three `Renderer` types for the `UITextEntry` component:

TABLE 8-1 EXAMPLE RENDERER TYPES

| RendererType | Render-Dependent Attributes |
|--------------|-----------------------------|
| "Text"       | "columns"                   |
| "Secret"     | "columns", "secretChar"     |
| "Multiline"  | "columns", "rows"           |

The tag library descriptor (TLD) file for the corresponding tag library, then, would define three custom tags -- one per `Renderer`. Below is an example of the tag definition for the "input\_text" tag<sup>1</sup>:

1. This example illustrates a non-normative convention for naming tags based on a combination of the component name and the renderer type. This convention is useful, but not required; custom actions may be given any desired tag name; however the convention is rigorously followed in the Standard HTML `RenderKit` Tag Library.

```

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

```

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

## JSF.8.2 Using UIComponent Custom Actions in JSP Pages

The following subsections define how a page author utilizes the custom actions provided by the `RenderKit` implementor in the JSP pages that create the user interface of a JSF-based web application.

### JSF.8.2.1 Declaring the Custom Actions Tag Library

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

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

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

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

### JSF.8.2.2 Defining Components in a Page

A JSF `UIComponent` custom action can be placed at any desired position in a JSP page that contains the `taglib` directive for the corresponding tag library. For example,

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

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

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

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 conjunction with the actions that represent UI components.

FIXME - document current `<output_body>` requirement on included pages

### JSF.8.2.3 Creating Components and Overriding Attributes

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

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

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

### JSF.8.2.4 Representing Component Hierarchies

Nested structures of `UIComponent` custom actions will generally mirror the hierarchical relationships of the corresponding `UIComponent` instances in the 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:

```
<h:form id="logonForm">
<table border="0">
<tr>
<td><h:output_label id="usernameLabel"
 for="/logonForm/username">Username:
</h:output_label></td>
<td><h:input_text id="username"
```

```

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

```

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 may, but need not, 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:

```

<h:input_text id="username" modelReference="logonBean.username">
 <f:validate_required/>
 <f:validate_length minimum="6"/>
</h:input_text>

```



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

## JSF.8.2.6 Interoperability with Other Custom Action Libraries

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

- All JSF tags nested inside of, and positioned as siblings to the custom tags, must have a manually defined *component identifier*.
- Conditional custom actions (such as the `<c:if>` and `<c:choose>` actions in the JSP Standard Tag Library) may dynamically determine whether nested body content inside these tags is rendered or not. JSF component custom actions nested inside such tags will not be invoked if the outer tag chooses to skip its body. Therefore, no components will be created (in the 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 `UIComponent`s (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 `URLConnection`, 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 <f:use\_faces/> Tag

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

### JSF.8.4 UIComponent Custom Action Implementation Requirements

The custom action implementation classes for `UIComponent` custom actions must conform to all of the requirements defined in the JavaServer Pages Specification. In addition, they must meet the following JSF-specific requirements [[FIXME](#) - make more precise]

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

### JSF.8.5 JSF Core Tag Library

All JSF implementations must provide a tag library containing core actions (described below) that are independent of a particular `RenderKit`. The corresponding tag library descriptor must meet the following requirements:

- Must declare a tag library version (`<tlib-version>`) value of “**1.0**”.
- Must declare a JSP version dependency (`<tlib-version>`) value of “**1.2**”.
- Must declare a URI (`<uri>`) value of “**http://java.sun.com/jsf/core**”.
- Must be included in the `META-INF` directory of a JAR file containing the corresponding implementation classes, suitable for inclusion with a web application, such that the tag library descriptor will be located automatically by the algorithm described in Section 7.3 of the JavaServer Pages Specification (version 1.2).

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

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

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

- **Name** -- The name of this tag, as used in a JSP page.

- **Short Description** -- A summary of the behavior implemented by this tag.
- **Syntax** -- One or more examples of using this tag, with the required and optional sets of attributes that may be used together.
- **Body Content** -- The type of nested content for this tag, using one of the standard values `empty`, `JSP`, or `tagdependent` as described in the JSP specification. This section also describes restrictions on the types of content (template text, JSF standard tags, JSF component tags, and/or other custom tags) that can be nested in the body of this tag.
- **Attributes** -- A table containing one row for each defined attribute for this tag. The following columns provide descriptive information about each attribute:
  - *Name* -- Name of this attribute, as it must be used in the page.
  - *Dyn* -- Boolean value indicating whether this attribute accepts dynamic values (documented in the `<rtexprvalue>` element).
  - *Type* -- Fully qualified Java class or primitive type of this attribute.
  - *Description* -- The functional meaning of this attribute's value.
- **Constraints** -- Additional constraints enforced by this action, such as combinations of attributes that may be used together.
- **Description** -- Details about the functionality provided by this action.

### JSF.8.5.1 <f:attribute>

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

#### **Syntax:**

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

#### **Body Content:**

empty.

#### **Attributes:**

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

#### **Constraints:**

- Must be nested inside a `UIComponent` tag.

#### **Description:**

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

## JSF.8.5.2 <f:event\_handler>

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

### **Syntax:**

```
<f:event_handler type="classname"/>
```

### **Body Content:**

empty.

### **Attributes:**

Name	Dyn	Type	Description
type	false	String	Fully qualified class name of a class that extends <code>javax.faces.event.RequestEventHandler</code> .

### **Constraints:**

- Must be nested inside a `UIComponent` tag.

### **Description:**

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

### JSF.8.5.3 <f:parameter>

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

#### **Syntax:**

*Syntax 1: Unnamed direct value*

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

*Syntax 2: Unnamed indirect value*

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

*Syntax 3: Named direct value*

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

*Syntax 4: Named indirect value*

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

#### **Body Content:**

empty.

#### **Attributes:**

Name	Dyn	Type	Description
id	false	String	Component identifier of a <code>UIParameter</code> component
modelReference	false	String	Model reference expression to acquire the value of this parameter
name	false	String	Name of the parameter to be set
value	false	String	Value of the parameter to be set

#### **Constraints:**

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

#### **Description:**

Implements the standard `FacesTag` functionality to create a new `UIParameter` component, if one does not already exist in the response component tree. Such components are used to dynamically configure parameter values for component tags

in which it this tag is nested, such as substitution values for MessageFormat replacement patterns ({0}) in the `output_message` tag of the Standard HTML Tag Library.

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

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



## JSF.8.5.4     <f:use\_faces>

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

### **Syntax:**

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

### **Body Content:**

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

### **Attributes:**

None.

### **Constraints:**

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

### **Description:**

Provides the JSF implementation a convient place to perform state saving during the *Render Response* phase of the request processing lifecycle, if the implementation elects to save state as part of the response. The actual processing performed by this action is determined by the JSF implementation.

## JSF.8.5.5 <f:validate\_doublerange>

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

### **Syntax:**

*Syntax 1: Maximum only specified*

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

*Syntax 2: Minimum only specified*

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

*Syntax 3: Both maximum and minimum are specified*

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

### **Body Content:**

empty.

### **Attributes:**

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

### **Constraints:**

- Must be nested inside a `UIComponent` tag whose value is (or is convertible to) a double.
- Must specify either the `maximum` attribute, the `minimum` attribute, or both.
- If both limits are specified, the maximum limit must be greater than the minimum limit.

### **Description:**

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

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

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

## JSF.8.5.6 <f:validate\_length>

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

### **Syntax:**

*Syntax 1: Maximum only specified*

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

*Syntax 2: Minimum only specified*

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

*Syntax 3: Both maximum and minimum are specified*

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

### **Body Content:**

empty.

### **Attributes:**

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

### **Constraints:**

- Must be nested inside a `UIComponent` tag whose value is a `String`.
- Must specify either the `maximum` attribute, the `minimum` attribute, or both.
- If both limits are specified, the maximum limit must be greater than the minimum limit.

### **Description:**

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

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

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

## JSF.8.5.7 <f:validate\_longrange>

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

### **Syntax:**

*Syntax 1: Maximum only specified*

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

*Syntax 2: Minimum only specified*

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

*Syntax 3: Both maximum and minimum are specified*

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

### **Body Content:**

empty.

### **Attributes:**

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

### **Constraints:**

- Must be nested inside a `UIComponent` tag whose value is (or is convertible to) a long.
- Must specify either the `maximum` attribute, the `minimum` attribute, or both.
- If both limits are specified, the maximum limit must be greater than the minimum limit.

### **Description:**

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

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

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

## JSF.8.5.8    <f:validate\_required>

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

### **Syntax:**

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

### **Body Content:**

empty.

### **Attributes:**

None.

### **Constraints:**

- Must be nested inside a `UIComponent` tag.

### **Description:**

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

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

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

## JSF.8.5.9 <f:validate\_stringrange>

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

### **Syntax:**

*Syntax 1: Maximum only specified*

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

*Syntax 2: Minimum only specified*

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

*Syntax 3: Both maximum and minimum are specified*

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

### **Body Content:**

empty.

### **Attributes:**

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

### **Constraints:**

- Must be nested inside a `UIComponent` tag whose value is (or is convertible to) a `String`.
- Must specify either the `maximum` attribute, the `minimum` attribute, or both.
- If both limits are specified, the maximum limit must be greater than the minimum limit.

### **Description:**

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

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

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

### JSF.8.5.10 <f:validator>

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

#### **Syntax:**

```
<f:validator type="classname"/>
```

#### **Body Content:**

empty.

#### **Attributes:**

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

#### **Constraints:**

- Must be nested inside a `UIComponent` tag.

#### **Description:**

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

## JSF.8.6 Standard HTML RenderKit Tag Library

All JSF implementations must provide a tag library containing actions (described below) that correspond to each valid combination of a supported component class (see Chapter FIXME/4) and a Renderer from the Standard HTML RenderKit (see Section FIXME/7.6) that supports that component type. The tag library descriptor for this tag library must meet the following requirements:

- Must declare a tag library version (<tlib-version>) value of “1.0”.
- Must declare a JSP version dependency (<tlib-version>) value of “1.2”.
- Must declare a URI (<uri>) value of “<http://java.sun.com/jsf/html>”.
- Must be included in the META-INF directory of a JAR file containing the corresponding implementation classes, suitable for inclusion with a web application, such that the tag library descriptor will be located automatically by the algorithm described in Section 7.3 of the JavaServer Pages Specification (version 1.2).

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

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

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

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





# JSF.9

---

## Using JSF In Web Applications

This specification provides JSF implementors significant freedom to differentiate themselves through innovative implementation techniques, as well as value-added features. However, to ensure that web applications based on JSF can be executed unchanged across different JSF implementations, the following additional requirements, defining how a JSF-based web application is assembled and configured, must be supported by all JSF implementations.

### JSF.9.1 Web Application Deployment Descriptor

JSF-based applications are *web applications* that conform to the requirements of the Servlet Specification (version 2.3 or later), and also use the facilities defined in this specification. Conformant web applications are packaged in a *web application archive* (WAR), with a well-defined internal directory structure. A key element of a WAR is the *web application deployment descriptor*, an XML document that describes the configuration of the resources in this web application. This document is included in the WAR file itself, at resource path “/WEB-INF/web.xml”.

Portable JSF-based web applications must include the following configuration elements, in the appropriate portions of the web application deployment descriptor. Element values that are rendered in *italics* represent values that the application developer is free to choose. Element values rendered in **bold** represent values that must be utilized exactly as shown.

Executing the request processing lifecycle via other mechanisms is also allowed (for example, an MVC-based application framework can incorporate calling the correct Phase implementations in the correct order); however, all JSF implementations must support the functionality described in this chapter to ensure application portability.

### JSF.9.1.1 Servlet Definition

JSF implementations must provide request processing lifecycle services through a standard servlet, defined by this specification. This servlet must be defined, in the deployment descriptor, as follows:

```
<servlet>
 <servlet-name> faces-servlet-name </servlet-name>
 <servlet-class>
 javax.faces.webapp.FacesServlet
 </servlet-class>
</servlet>
```

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

### JSF.9.1.2 Servlet Mapping

All requests to a web application are mapped to a particular servlet based on matching a URL pattern (as defined in the Servlet Specification) against the portion of the request URL after the context path that selected this web application. The following mapping for the standard servlet providing portable JSF lifecycle processing 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<sup>1</sup>. 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.9.2.4.1 FactoryFinder

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

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

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

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

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

JSF implementations must also include implementations of the several factory classes. In order to be dynamically instantiated according to the algorithm defined above, the factory implementation class must include a public, no-arguments constructor. Factory class implementations must be provided for the following factory names:

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

#### JSF.9.2.4.2FacesServlet

`FacesServlet` is an implementation of `javax.servlet.Servlet` that accepts incoming requests and passes them to the appropriate `Lifecycle` implementation for processing. This servlet must be declared in the web application deployment descriptor, as described in Section FIXME/9.1.1, and mapped to a standard URL pattern as described in Section FIXME/9.1.2.

- `public void init(ServletConfig config);`

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

- `public void destroy();`

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

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

For each incoming request, the following processing is performed:

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

#### JSF.9.2.4.3FacesTag

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

- `protected UIComponent findComponent();`

Using the current value of the `id` property of this tag handler, locate and return the corresponding `UIComponent` instance from the 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.