JavaServer™ Faces Specification

Version 2.0

Ed Burns, Roger Kitain, editors

See <https://javaserverfaces-spec-public.dev.java.net/>
to comment on and discuss this specification.

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

March 2009

Submit comments about this document to `jsr-314-comments@jcp.org`

SUN IS WILLING TO LICENSE THIS SPECIFICATION TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS LICENSE AGREEMENT ("AGREEMENT"). PLEASE READ THE TERMS AND CONDITIONS OF THIS AGREEMENT CAREFULLY. BY DOWNLOADING THIS SPECIFICATION, YOU ACCEPT THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU ARE NOT WILLING TO BE BOUND BY THEM, SELECT THE "DECLINE" BUTTON AT THE BOTTOM OF THIS PAGE AND THE DOWNLOADING PROCESS WILL NOT CONTINUE.

Specification: JSR-000314 JavaServer(tm) Faces Specification ("Specification")

Version: 2.0

Status: Proposed Final Draft

Release: 27March 2009

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

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 Agreement.

Subject to the terms and conditions of this license, including your compliance with Paragraphs 1 and 2 below, Sun hereby grants you a fully-paid, non-exclusive, non-transferable, limited license (without the right to sublicense) under Sun's intellectual property rights to:

1.Review the Specification for the purposes of evaluation. This includes: (i) developing implementations of the Specification for your internal, non-commercial use; (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Technology.

2.Distribute implementations of the Specification to third parties for their testing and evaluation use, provided that any such implementation:

(i) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented;

(ii)is clearly and prominently marked with the word "UNTESTED" or "EARLY ACCESS" or "INCOMPATIBLE" or "UNSTABLE" or "BETA" in any list of available builds and in proximity to every link initiating its download, where the list or link is under Licensee's control; and

(iii)includes the following notice:

"This is an implementation of an early-draft specification developed under the Java Community Process (JCP) and is made available for testing and evaluation purposes only. The code is not compatible with any specification of the JCP."

The grant set forth above concerning your distribution of implementations of the specification is contingent upon your agreement to terminate development and distribution of your "early draft" implementation as soon as feasible following final completion of the specification. If you fail to do so, the foregoing grant shall be considered null and void.

No provision of this Agreement shall be understood to restrict your ability to make and distribute to third parties applications written to the Specification.

Other than this limited license, you acquire no right, title or interest in or to the Specification or any other Sun intellectual property, and the Specification may only be used in accordance with the license terms set forth herein. This license will expire on the earlier of: (a) two (2) years from the date of Release listed above; (b) the date on which the final version of the Specification is publicly released; or (c) the date on which the Java Specification Request (JSR) to which the Specification corresponds is withdrawn. In addition, this license 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.

"Licensor Name Space" means the public class or interface declarations whose names begin with "java", "javax", "com.sun" or their equivalents in any subsequent naming convention adopted by Sun through the Java Community Process, or any recognized successors or replacements thereof

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, 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 hold Sun (and its licensors) harmless from any claims based on your use of the Specification for any purposes other than the limited right of evaluation as described above, 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.

GENERAL TERMS

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

Contents

Preface

This is the JavaServer Faces 2.0 (JSF 2.0) specification, developed by the JSR-314 expert group under the Java Community Process (see <http://www.jcp.org> for more information about the JCP).

Changes between 1.2 Final and Early Draft Review 2

This section gives a change-by-change accounting of the modifications to the spec since the draft listed in the title of this section. Readers interested in a user level overview should consult Section "Compatibility with and Migration from JavaServer Faces 1.2".

Section 2.1 "Request Processing Lifecycle Scenarios"

Modified to define and explain resource requests and responses.

Section 2.2 "Standard Request Processing Lifecycle Phases"

Specify how and when the `currentPhaseId` property of the current `FacesContext` must be updated.

Section 2.2.1 "Restore View"

Modified to indicate that the `PostAddToViewEvent` event must be sent after the view was created. Also specify that if the VDL is Facelets, the tree must be fully constructed before exiting Restore View.

Change how the "binding" attribute is handled. In the case of a programmatically created view, manually traverse the tree and send each node the `AfterRestoreViewEvent`. In the case of a normally restored tree, the "binding" attribute is handled by `UIViewRoot.processRestoreState()`, which is already called from `StateManager.restoreView()`.

Modify the non-faces-request case to include view parameter processing.

Section 2.2.2 "Apply Request Values"

Specified additional behavior to recognize partial requests and to perform partial processing.

Section 2.2.2.1 “Apply Request Values Partial Processing”

Specified behavior for partial processing.

Section 2.2.3 “Process Validations”

Specified additional behavior to recognize partial requests and to perform partial processing.

Section 2.2.3.1 “Partial Validations Partial Processing”

Specified behavior for partial processing.

Section 2.2.4 “Update Model Values”

Specified additional behavior to recognize partial requests and to perform partial processing.

Section 2.2.4.1 “Update Model Values Partial Processing”

Specified behavior for partial processing.

Section 2.2.6 “Render Response”

Generalized to remove JSP specific language.

Added the requirement for (partial requests) to prevent writing to the response at the start of this phase (to prevent content from being written outside f:view)

Section 2.5.2.4 “Localized Application Messages”

Added message key for Bean Validation.

Section 2.5.4 “Resource Handling”

Add non-normative section traversing this feature.

Section 2.5.5 “View Parameters”

Add non-normative section traversing this feature.

Section 2.5.6 “Bookmarkability”

Add non-normative section traversing this feature.

Section 2.5.7 “JSR 303 Bean Validation”

Add non-normative section traversing this feature.

Section 2.5.8 “Ajax”

Add non-normative section traversing this feature.

Section 2.5.9 “Component Behaviors”

Add non-normative section traversing this feature.

New Section 2.6 “Resource Handling”

This section is the starting point for the specification of the Resource Handler facility, which is also specified in the JavaDocs and the Standard RenderKit Docs.

New Section 2.6.2 “Rendering Resources”

This section briefly talks about how resources (such as images, stylesheets and scripts) use the resource handling mechanism.

New Section 2.6.2.1 “Relocatable Resources”

This section outlines the mechanism that script and stylesheet resources use to render themselves in a different location (with respect to tag or component placement in the view).

New Section 2.6.2.2 “Resource Rendering Using Annotations”

This section describes the use of an annotation to mark that a component requires a resource.

Section 3.1.8 “Component Tree Navigation”

Added descriptions for UIComponent.getCurrentComponent and UIComponent.getCurrentCompositeComponent.

Added descriptions for visitTree().

Section 3.1.10 “Managing Component Behavior”

Described additional method implementations of the BehaviorHolder interface.

Section 3.1.11 “Generic Attributes”

Described additional responsibilities for Map get() method if the component instance is a composite component.

Section 3.1.11.1 “Special Attributes”

Describe UIComponent contants that are used in attribute Map(s).

Section 3.1.13 “Component Specialization Methods”

Mentioned the default behavior of UIComponentBasse encodeChildren if no associated renderer. Mentioned encodeBegin() must publish PreRenderComponentEvent.

Section 3.1.14 “Lifecycle Management Methods”

Added pointers to `pushComponentToEL`() `popComponentFromEL`() in support of “component” implicit object.

Section 3.1.15 “Utility Methods”

Added UIComponent utility method getResourceBundleMap().

Section 3.2.6.1 “Properties”

Mentioned ResourceDependency/ResourceDependencies lookup for ValueHolder setConverter method.

Section 3.2.7.2 “Methods”

Mentioned ResourceDependency/ResourceDependencies lookup for EditableValueHolder addValidator method..

Section 3.2.8 “SystemEventListenerHolder”

Added section describing this new behavioral interface.

Section 3.3.2 “Converter”

Added verbage about Resource annotations attached to Converters.

Section 3.4.1 “Overview”

Updated UML diagram of event package

Moved existing event content to be in new subsection: Section 3.4.2 “Application Events”, and created a new subsection Section 3.4.3 “System Events”

Section 3.4.2.6 “Event Broadcasting”

Clarification made: throwing an AbortProcessingException tells an implementation that no further broadcast of the current event occurs. Does not affect future events.

Section 3.4.3.1 “Event Classes”

Added descriptions for PostConstructApplicationEvent and PreDestroyApplicationEvent.

Section 3.4.3.4 “Declarative Listener Registration”

New section for declarative events.

Section 3.4.3.5 “Listener Registration By Annotation”

Added verbiage about ListenerFor and ListenersFor annotations.

Section 3.5.2 “Validator Classes”

Added verbage about Resource annotations attached to Validators.

Section 3.5.2 “Validator Classes”

Add “javax.faces.RegularExpressionValidator” standard validator

Section 3.5.2 “Validator Classes”

Added validaor requirements with respect to dealing with null or empty values.

Section 3.5.3 “Validation Registration”

Added default validator registration requirements.

Section 3.5.5 “Standard Validator Implementations”

Added requirements for BeanValidator and RequiredValidator.

Section 3.5.6 “Bean Validation Integration”

Bean Validation integration.

Section 3.7 “Component Behavior Model”

Section describes adding behavior to the component model.

Section 4.1.19.2 “Properties”

Specify the `viewMap` property on UIViewRoot.

Section 4.1.19.3 “Methods”

Specify new methods on UIViewRoot for handling resources for the view.

Section 4.1.19.4 “Events”

Added UIViewRoot getPhaseListeners().

Section 4.1.19.5 “Partial Processing”

Specify additional behavior for UIViewRoot methods to faciliate partial processing.

Section 4.2.1.2 “Methods”

Specify iterator() method for DataModel.

Section 3.6 “Composite User Interface Components”

New section specifying composite components.

Section 5.2.1 “MethodExpression Syntax and Semantics”

Modify content relating to managed-bean-scope to include “view” scope.

Section 5.3 “The Managed Bean Facility”

Mention `META-INF/managed-beans.xml`.

Section 5.4.1 “Managed Bean Lifecycle Annotations”

Modify `@PostConstruct` to state that an exception thrown during the @PostConstruct must cause a log message to be logged.

Modify content to clarify when `@PreDestroy` must be called in the case of view scoped managed beans.

Section 5.6.1.1 “Faces Implicit Object ELResolver For JSP” and Section 5.6.2.1 “Implicit Object ELResolver for Facelets and Programmatic Access”

Specify how the new implicit object ”`resource`“ must be handled by the Implicit Object ELResolver.

Specify how `viewScope`, `component`, and `compositeComponent` are resolved.

Section 5.6.1.2 “ManagedBean ELResolver”

Modify setValue() to allow for atomic lazy creation. This eliminates the need to do a get() before doing a set().

Section 5.6.2.1 “Implicit Object ELResolver for Facelets and Programmatic Access”

Add a new implicit object: “resource”. This allows easily encoding resources into markup using EL expressions

Corrected behavior of getType with respect to "requestScope", "sessionScope", or “applicationScope” - should return null, not Object.class.

Section 5.6.2.5 “Resource ELResolver”

This section specifies the behavior of the Resource EL Resolver

Section 5.6.2.2 “Composite Component Attributes ELResolver”

New ELResolver that ensures that #{compositeComponent.attrs} resolves to a special Map.

Section 5.6.2.9 “ScopedAttribute ELResolver”

Specify that setPropertyResolved(true) is called in all cases.

Section 6.1.2 “Attributes”

Add new section after 6.1.1 documenting the new `Map` returned from `FacesContext.getAttributes()`.

Section 6.1.8 “ResponseStream and ResponseWriter”

Add FacesContext enableResponseWriting method.

Section 6.1.10 “Partial Processing Methods”

Specify the FacesContext contants and methods that facilitate partial request processing.

Section 6.1.11 “Partial View Context”

Specify this class is used to facilitate partial view processing and partial view rendering.

Section 6.1.12 “Access To The Current FacesContext Instance”

Specify how this method must behave during application startup time.Corrected access keyword for FacesContext.setCurrentInstance() to be `protected` instead of `public`.

Section 6.1.13 “CurrentPhaseId”

New property to access the current phase.

Section 6.2 “ExceptionHandler”

New property to access the `ExceptionHandler` for this request.

Section 6.7 “ExceptionHandlerFactory”

New factory for `ExceptionHandler` instances.

Section 6.8 “ExternalContextFactory”

New factory for `ExternalContext.`

Section 7.1.8 “ProjectStage Property”

This section documents the new ProjectStage property. This is similar in use to the RAILS_ENV environment variable from the Ruby on Rails framework.

Section 7.1.13 “System Event Methods”

New section describing system events.

Section 7.4.2 “Default NavigationHandler Algorithm”

Specify how to handle implicit navigation.

Specify how to handle conditional navigation

Require that `context.getFlash().setRedirect(true)` is called if the navigation is a redirect.

Specify that an informative message must be rendered in the page if there is no outcome match and `ProjectStage` is not `Production`.

Special handling for view parameters and redirect.

Section 7.5.1 “Overview”

In `createView()`, if the VDL is Facelets, make sure the view is fully populated before returning.

Section 7.5.2 “Default ViewHandler Implementation”

Move the viewId derivation algorithm to be inside of the new `ViewHandler.deriveViewId()` method and specify it to deal with the new DEFAULT_SUFFIX definition.

Modify getActionURL() to remove the use of DEFAULT_SUFFIX and instead take a simpler implementation.

Refactored VDL specific logic into new `ViewDeclarationLanguage` class.

Section 7.6 “ViewDeclarationLanguage”

New section which coveres how Facelets and JSP are handled via the ViewDeclarationLanguage class.

Section 8.1 “RenderKit”

New methods on `RenderKit`: `getComponentFamilies()` and `getRendererTypes()`.

Section 8.2 “Renderer”

Mentioned the ListenerFor annotation.

Section 8.3 “ClientBehaviorRenderer”

Renderer for component Behavior.

Section 9.4.3 “<f:convertDateTime>”

Extends ConverterELTag, not ConverterTag.

Section 9.4.4 “<f:convertNumber>”

Extends ConverterELTag, not ConverterTag

Section 9.4.14 “<f:validateDoubleRange>”

Extends ValidatorELTag, not ValidatorTag

Section 9.4.16 “<f:validateRegex>”

New standard validator

Section 9.4.17 “<f:validateLongRange>”

Extends ValidatorELTag, not ValidatorTag

Section 9.4.21 “<f:view>”

Extends UIComponentELTag, not UIComponentBodyTag

Section “Facelets and its use in Web Applications”

New chapter insterted after Chapter 9, titled, “Integration with Facelets”. This implies increasing the remaining chapter numbers by one.

Section 10.4.1.1 “<f:ajax>”

Declarative Ajax tag.

Section “Override default Ajax action. “button1” is associated with the Ajax “execute=’cancel’” action:”

Bean Validation tag.

Section 10.4.1.5 “<f:validateRequired>”

Bean Validation tag.

Section 11.1.3 “Application Configuration Parameters”

New `javax.faces.PROJECT_STAGE` ServletContext init param.

New `javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL` ServletContext init param.

New `javax.faces.DISABLE_FACES_VDL_VIEWHANDLER` ServletContext init param.

Modify `javax.faces.DEFAULT_SUFFIX` init param, add `javax.faces.FACELETS_DEFAULT_SUFFIX`, `javax.faces.FACELETS_VIEW_MAPPINGS` init params.

New `javax.faces.VALIDATE_EMPTY_FIELDS` ServletContext init param.

New `javax.faces.PARTIAL_STATE_SAVING` ServletContext init param

New `javax.faces.FULL_STATE_SAVING_VIEW_IDS` servlet context init param.

Explicitly ignore “`/WEB-INF/faces-config.xml`” in `javax.faces.CONFIG_FILES`, if present.

Section 11.4.2 “Application Startup Behavior”

Change rules to support ordering of configuration resources.

Section 11.4.5 “Configuration Impact on JSF Runtime”

Specify requirements for handling `resource-handler` elements within the application configuration resources.

Specify requirements for handling `faces-lifecycle-listener` elements within the application configuration resources.

Declare `exception-handler-factory`.

Declare `discovery-handler-factory`.

Declare `page-declaration-language-factory`.

Section 11.4.6 “Delegating Implementation Support”

List decoratable artifacts.

Section 11.4.7 “Ordering of Artifacts”

Define the rules for ordering of configuratino resources.

Section 11.5 “Annotations that correspond to and may take the place of entries in the Application Configuration Resources”

New section detailing new annotations.

Section 12.2 “PhaseEvent”

Statement should read: encapsulated by FacesContext...

Chapter 13 “Ajax Integration

New chapter describing how Ajax will integrate with JavaServer Faces.

Section 13.1 “JavaScript Resource”

This section describes the standard Ajax JavaScript resource that will be used in JavaServer Faces.

Section 13.1.1 “JavaScript Resource Loading”

This section describes how the Ajax resource will leverage the resource loading feature.

Section 13.1.1.1 “The Annotation Approach”

This section mentions the use of the resource annotation to specify that a component or renderer requires the Ajax ressource.

Section 13.1.1.2 “The Resource API Approach”

Component authors can also specify that a custom component or renderer requires the Ajax resource by using the resource APIs.

Section 13.1.1.3 “The Page Declaration Language Approach”

Page authors can make the Ajax resource available through the standard resource tags.

Section 13.2 “JavaScript Namespacing”

This section discusses the JavaScript namespacing requirements for the Ajax resource to avoid collisions with other JavaScript libraries.

Section 13.3 “Ajax Interaction”

This section describes the JavaScript functions that will be available to allow clients to perform Ajax interactions with JavaServer Faces.

Section 13.3.1 “Sending an Ajax Request”

This section describes the process of sending an Ajax request to the server.

Section 13.3.2 “Ajax Request Queueing”

Higher level requirements about queueing Ajax requests before they are sent.

Section 13.3.3 “Request Callback Function”

Describes the functionality when a response comes back from the server.

Section 13.3.4 “Receiving The Ajax Response”

Describes the requirements of javax.faces.Ajax.ajaxResponse - the function that gets called from the Ajax request callback function.

Section 13.3.5 “Monitoring Events On The Client”

Describes the JavaScript functions used to register event and error callback functions that will be notified when events and errors occur.

Section 13.3.5.1 “Monitoring Events For An Ajax Request”

Details about specifying the “onevent” attribute.

Section 13.3.5.2 “Monitoring Events For All Ajax Requests”

Specifics about the jsf.ajax.addOnEvent function.

Section 13.3.5.3 “Sending Events”

Details about sending client side events.

Section 13.3.6 “Handling Errors On the Client”

Specifics about the JavaScript functions to use for handling errors on the client.

Section 13.3.6.1 “Handling Errors For An Ajax Request”

Details about specifying “onerror” attribute.

Section 13.3.6.2 “Handling Errors For All Ajax Requests”

Details about jsf.ajax.addOnError function.

Section 13.3.6.3 “Signaling Errors”

Specifics about signaling client side errors.

Section 13.3.7 “Handling Errors On The Server”

Specifics about exception handling on the server for Ajax requests.

Section 13.4 “Partial View Traversal”

This section provides a summary of how Faces can process one or more components in a view - know as partial processing.

Section 13.4.1 “Partial Traversal Strategy”

This section provides a summary of how frameworks can plug in strategies for performing partial view processing and partial view rendering.

Section 13.4.2 “Partial View Processing”

This section describes how one or more components can be processed in the “execute” portion of the request processing lifecycle.

Section 13.4.3 “Partial View Rendering”

This section describes how one or more components can be processed in the “render” portion of the request processing lifecycle.

Section 13.4.4 “Sending The Response to The Client”

Describes the server side responsibilities for preparing and sending the response markup back to the client.

Section 13.4.4.1 “Writing The Partial Response”

Describes the PartialResponseWriter requirements.

Chapter 14 “JavaScript API

New Chapter - JavaScript API for JSF 2.0

Section 14.1 “Collecting and Encoding View State”

Describes the JavaScript function that can be used to return encoded state for a given form.

Section 14.1.1 “Use Case”

Simple example of using the jsf.getViewState function.

Section 14.2 “Initiating an Ajax Request”

Describes the JavaScript function used to send Ajax requests.

Section 14.2.1 “Usage”

jsf.ajax.request function syntax and arguments.

Section 14.2.3 “Default Values”

Default values for the “execute” and “render” arguments.

Section 14.2.4 “Request Sending Specifics”

Implementation requirements for the jsf.ajax.reqeust function.

Section 14.2.5 “Use Case”

Simple example of the request function.

Section 14.5 “Determining An Application’s Project Stage”

This section summarizes the implementation requirements for the jsf.ajax.response function.

Section 14.4 “Registering Callback Functions”

This section describes the functions that can be used to register callback functions that will be notified when events and errors occur.

Section 14.4.1 “Request/Response Event Handling”

Describes the specifics of using the JavaScript API to register event handling callback functions.

Section 14.4.1.1 “Use Case”

Simple example of jsf.ajax.addOnEvent function.

Section 14.4.2 “Error Handling”

Describes the specifics of using the JavaScript API to register error handling callback functions.

Section 14.4.2.1 “Use Case”

Simple example of jsf.ajax.addOnError function.

Section 14.5 “Determining An Application’s Project Stage”

Describes the function used to determine an application’s project stage.

Section 14.5.1 “Use Case”

Simple example of jsf.getProjectStage function.

Section 14.6 “Script Chaining”

Describes the jsf.util.chain function that can be used to chain function calls.

Section 1.1 “XML Schema Definition for Application Configuration Resource file”

Add the `resource-handler` element.

Add the `faces-lifecycle-listener` element and its children.

Section 1.2 “XML Schema Definition for Ajax Response”

New section - the layout for the Ajax response.

Standard HTML RenderKit specification

component-family: javax.faces.Graphic renderer-type: javax.faces.Image

Spec for what to do if “name”, “library” or “target” attributes are present

component-family: javax.faces.Output renderer-type: javax.faces.Body

This is a new Renderer in the standard-html-renderkit

component-family: javax.faces.Output renderer-type: javax.faces.Head

This is a new Renderer in the standard-html-renderkit

component-family: javax.faces.Output renderer-type: javax.faces.resource.Script

This is a new Renderer in the standard-html-renderkit

component-family: javax.faces.Output renderer-type: javax.faces.resource.Stylesheet

This is a new Renderer in the standard-html-renderkit

General Changes

The numbers in the text below refer to issue numbers in the issue tracker found at <https://javaserverfaces-spec-public.dev.java.net/servlets/ProjectIssues>.

- 100 - New methods on `RenderKit`: `getComponentFamilies()` and `getRendererTypes()`.
- 170 - Allow commandButton to have f:param children.
- 175 - Non-normatively clarify that the value of the "src" attribute will have the context-root prepended to it if the value starts with "/".
- 199 - Spec updates to clarify commandLink.
- 201 - Clean up convertNumber locale attribute to match with convertDateTime locale attribute.

- 226 - Require that SelectOneListBox and all similar renderers set "" if no request parameter can be found for the component instance.
- 228 - add selectedClass and unselectedClass to selectManyCheckbox
- 232 - Make javax.faces.model.DataModel implement Iterable.
- 259 - For selectOneRadio and selectManyCheckbox, normatively require the "style" and "border" elements to end up on the respective attributes on the rendered "table".
- 310 - Add context-param for setting the default timezone of DateTimeConverter instances
- 311 - Make the documentation in the spec prose document and the javadocs for the “first” property of UIData be consistent. In both cases, the value returned is relative to “zero”, not “one”.
- 317 - Make it so you if you try to do setValue on a managedBean that is not yet instantiated, it gets automatically instantiated first.
- 331 - add getPhaseListeners() to UIViewRoot.
- 361 - Section 3.1.5, fix missed ValueBinding to ValueExpression change.

Compatibility with and Migration from JavaServer Faces 1.2

This section provides a user-level survey of topic relating to migrating a JSF application between the version in the title of the section and the version of the spec in this document. This is *not* an exhaustive HOWTO.

Related Technologies

Other Java™ Platform Specifications

JSF is based on the following Java API specifications:

- JavaServer Pages™ Specification, version 2.1 (JSP™) <http://java.sun.com/products/jsp/>
- Java™ Servlet Specification, version 2.5 (Servlet) <http://java.sun.com/products/servlet/>
- Java™2 Platform, Standard Edition, version 5.0 <http://java.sun.com/j2se/>
- Java™2 Platform, Enterprise Edition, version 5.0 <http://java.sun.com/j2ee/>
- JavaBeans™ Specification, version 1.0.1 <http://java.sun.com/products/javabeans/docs/spec.html>
- JavaServer Pages™ Standard Tag Library, version 1.2 (JSTL) <http://java.sun.com/products/jsp/jstl/>

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

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

- Portlet Specification, 1.0 JSR-168 <http://www.jcp.org/jsr/detail/168.jsp>
- Portlet Specification, 2.0 JSR-286 <http://www.jcp.org/jsr/detail/286.jsp>
- JSF Portlet Bridge Specification, JSR-301 <http://www.jcp.org/jsr/detail/301.jsp>

Related Documents and Specifications

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

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

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

The JSP tag library for the HTML_BASIC standard RenderKit is specified in the TLDDocs and incorporated by reference in this Specification.

Terminology

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

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

Providing Feedback

We welcome any and all feedback about this specification. Please email your comments to <jsr-314-comments@jcp.org>.

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

Acknowledgements

The JavaServer Faces Specification (version 2.0) is the result of the diligent efforts of the JSR-314 Expert Group, working under the auspices of the Java Community Process.

1

Overview

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

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

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

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

1.1 Solving Practical Problems of the Web

JSF's core architecture is designed to be independent of specific protocols and markup. However it is also aimed directly at solving many of the common problems encountered when writing applications for HTML clients that communicate via HTTP to a Java application server that supports servlets and JavaServer Pages (JSP) based applications. These applications are typically form-based, and are comprised of one or more HTML pages with which the user interacts to complete a task or set of tasks. JSF tackles the following challenges associated with these applications:

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

1.2 Specification Audience

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

1.2.1 Page Authors

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

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

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

1.2.2 Component Writers

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

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

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

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

- A page author creating a custom "widget" for use on a particular page

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

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

1.2.3 Application Developers

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

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

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

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

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

1.2.4 Tool Providers

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

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

- Application generators that convert high level descriptions of an entire application into the set of pages, UI components, and application modules needed to provide the required application functionality

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

1.2.5 JSF Implementors

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

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

1.3 Introduction to JSF APIs

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

1.3.1 package `javax.faces`

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

Please see *Section 11.2.6.1 "FactoryFinder"*.

1.3.2 package `javax.faces.application`

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

Please see *Section 7.1 "Application"*.

1.3.3 package `javax.faces.component`

This package contains fundamental APIs for user interface components.

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

1.3.4 package javax.faces.component.html

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

1.3.5 package javax.faces.context

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

Please see *Section 6.1 "FacesContext"*.

1.3.6 package javax.faces.convert

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

Please see *Section 3.3 "Conversion Model"*.

1.3.7 package javax.faces.el

As of version 1.2 of this specification, all classes and interfaces in this package have been deprecated in favor of the Unified Expression Language (EL) from JSP 2.1.

Please see *Chapter 5 "Expression Language and Managed Bean Facility.*

1.3.8 package javax.faces.lifecycle

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

Please see *Chapter 2 "Request Processing Lifecycle.*

1.3.9 package javax.faces.event

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

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

1.3.10 package javax.faces.render

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

Please see *Chapter 8 "Rendering Model.*

1.3.11 package `javax.faces.validator`

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

Please see *Section 3.5 "Validation Model"*

1.3.12 package `javax.faces.webapp`

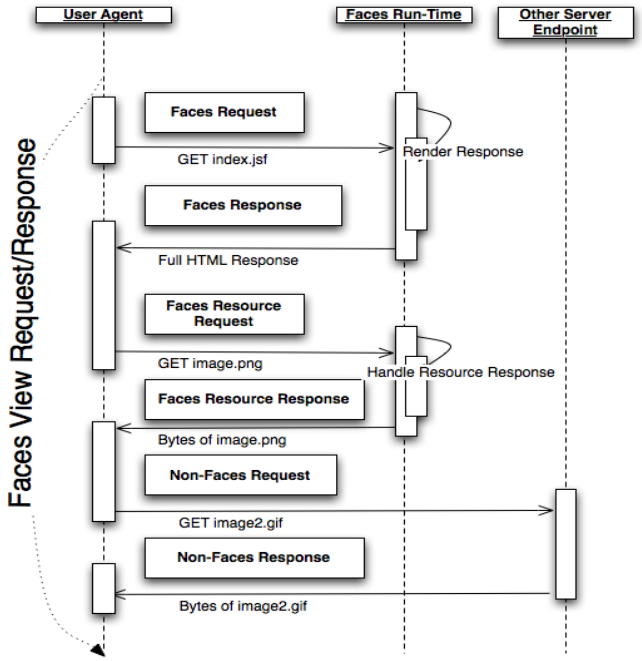
Classes required for integration of JavaServer Faces into web applications, including a standard servlet, base classes for JSP custom component tags, and concrete tag implementations for core tags.

Please see *Chapter 11 "Using JSF in Web Applications*.

2

Request Processing Lifecycle

Web user interfaces generally follow a pattern where the user-agent sends one or more requests to the server with the end goal of displaying a user-interface. In the case of Web browsers, an initial HTTP GET or POST request is made to the server, which responds with a document which the browser interprets and automatically makes subsequent requests on the user's behalf. The responses to each of these subsequent requests are usually images, JavaScript files, CSS Style Sheets, and other artifacts that fit "into" the original document. If the JSF lifecycle is involved in rendering the initial response, the entire process of initial request, the response to that request, and any subsequent requests made automatically by the user-agent, and their responses, is called a *Faces View Request/Response* for discussion. The following graphic illustrates a Faces View Request/Response.



Each Faces View Request/Response goes through a well-defined *request processing lifecycle* made up of *phases*. There are three different scenarios that must be considered, each with its own combination of phases and activities:

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

Where the terms being used are defined as follows:

- *Faces Response*—A response that was created by the execution of the *Render Response* phase of the request processing lifecycle.
- *Non-Faces Response*—A response that was not created by the execution of the *render response* phase of the request processing lifecycle. Examples would be a servlet-generated or JSP-rendered response that does not incorporate JSF components, a response that sets an HTTP status code other than the usual 200 (such as a redirect), or a response

whose HTTP body consists entirely of the bytes of an in page resource, such as a JavaScript file, a CSS file, an image, or an applet. This last scenario is considered a special case of a Non-Faces Response and will be referred to as a *Faces Resource Response* for the remainder of this specification.

- *Faces Request*—A request that was sent from a previously generated *Faces response*. Examples would be a hyperlink or form submit from a rendered user interface component, where the request URI was crafted (by the component or renderer that created it) to identify the view to use for processing the request. Another example is a request for a resource that the user-agent was instructed to fetch an artifact such as an image, a JavaScript file, a CSS stylesheet, or an applet. This last scenario is considered a special case of a Faces Request and will be referred to as a *Faces Resource Request* for the remainder of this specification.
- *Non-Faces Request*—A request that was sent to an application component (e.g. a servlet or JSP page), rather than directed to a Faces view.

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

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

2.1 Request Processing Lifecycle Scenarios

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

2.1.1 Non-Faces Request Generates Faces Response

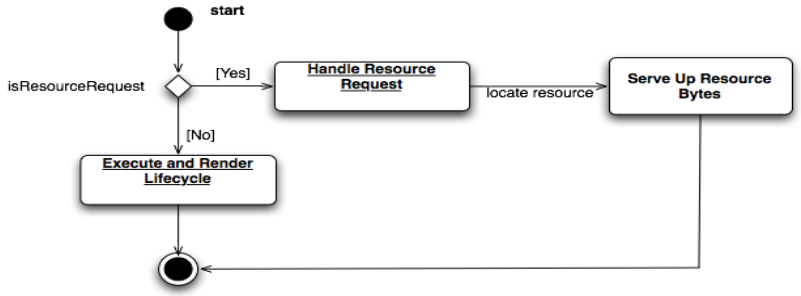
An application that is processing a non-Faces request may use JSF to render a Faces response to that request. In order to accomplish this, the application must perform the common activities that are described in the following sections:

- Acquire Faces object references, as described in Section 2.4.1 “Acquire Faces Object References”, below.
- Create a new view, as described in Section 2.4.2 “Create And Configure A New View”, below.
- Store the view into the `FacesContext` by calling the `setViewRoot()` method on the `FacesContext.`

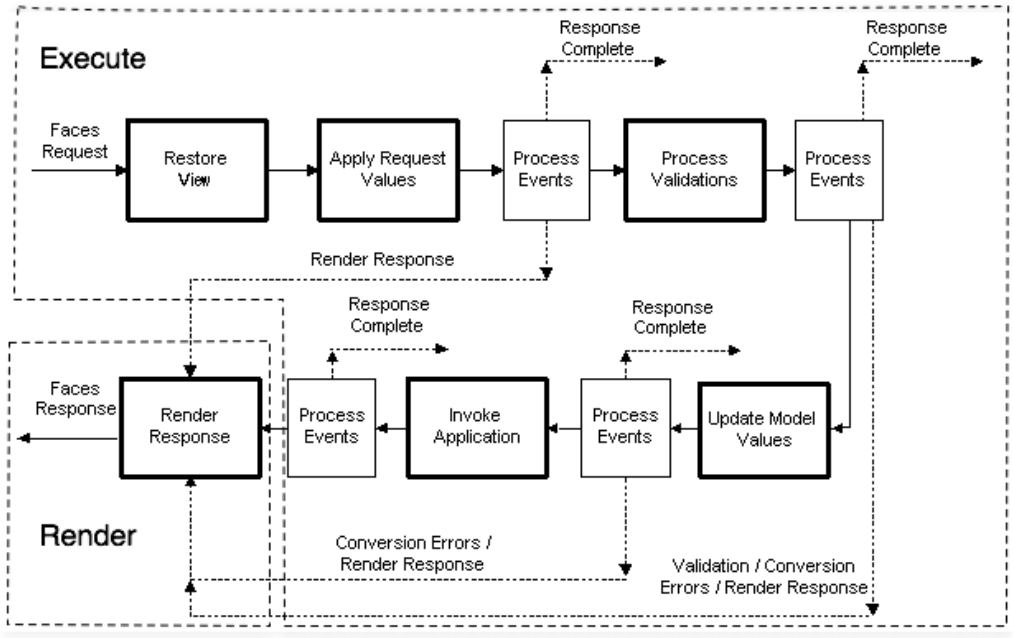
2.1.2 Faces Request Generates Faces Response

The most common lifecycle will be the case where a previous Faces response includes user interface controls that will submit a subsequent request to this web application, utilizing a request URI that is mapped to the JSF implementation’s controller, as described in Section 11.1.2 “Servlet Mapping”. Because such a request will be initially handled by the JSF

implementation, the application need not take any special steps—its event listeners, validators, and application actions will be invoked at appropriate times as the standard request processing lifecycle, described in the following diagrams, is invoked.



The "Handle Resource Request" box, and its subsequent boxes, are explained in Section 2.6 "Resource Handling". The following diagram explains the "Execute and Render Lifecycle" box.



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

2.1.3 Faces Request Generates Non-Faces Response

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

- A Faces Request needs to be redirected to a different web application resource (via a call to `HttpServletResponse.sendRedirect`).
- A Faces Request causes the generation of a response using some other technology (such as a servlet, or a JSP page not containing JSF components).
- A Faces Request causes the generation of a response simply by serving up the bytes of a resource, such as an image, a JavaScript file, a CSS file, or an applet

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

2.2 Standard Request Processing Lifecycle Phases

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

[P1-start-currentPhaseId]The default request lifecycle processing implementation must ensure that the `currentPhaseId` property of the `FacesContext` instance for this request is set with the proper `PhaseId` constant for the current phase as early as possible at the beginning of each phase.[P1-end]

2.2.1 Restore View

[P1-start-restoreView]The JSF implementation must perform the following tasks during the *Restore View* phase of the request processing lifecycle:

- Call `initView()` on the `ViewHandler`. This will set the character encoding properly for this request.
- Examine the `FacesContext` instance for the current request. If it already contains a `UIViewRoot`:
  - Set the `locale` on this `UIViewRoot` to the value returned by the `getRequestLocale()` method on the `ExternalContext` for this request.
  - Call `doTreeTraversal()` on the `UIViewRoot`, passing a `ContextCallback` implementation that calls the `processEvent()` method of the component. The argument `event` must be an instance of `PostRestoreStateEvent` whose `component` property is the current component in the traversal.
  - Take no further action during this phase, and return. The presence of a UIViewRoot already installed in the FacesContext before the Restore View Phase implementation indicates that the phase should assume the view has already been restored by other means.
- Derive the `viewId` by calling `deriveViewId()` on the `ViewHandler`. If calling this method throws `UnsupportedOperationException`, the implementation must ensure that the steps specified in section Section 7.5.2 “Default ViewHandler Implementation” regarding `deriveViewId()` are carried out to derive the `viewId`.
- Determine if this request is a postback or initial request by executing the following algorithm. Find the render-kit-id for the current request by calling `calculateRenderKitId()` on the `Application`’s `ViewHandler`. Get that `RenderKit`’s `ResponseStateManager` and call its `isPostback()` method, passing the current `FacesContext`.
- If the request is a postback, call `ViewHandler.restoreView()`, passing the `FacesContext` instance for the current request and the view identifier, and returning a `UIViewRoot` for the restored view. If the return from `ViewHandler.restoreView()` is null, throw a `ViewExpiredException` with an appropriate error message. `javax.faces.application.ViewExpiredException` is a `FacesException` that must be thrown to signal to the application that the expected view was not returned for the view identifier. An application may choose to perform some action based on this exception.

  Store the restored `UIViewRoot` in the `FacesContext`.
- If the request is not a postback, try to obtain the `ViewDeclarationLanguage` from the `ViewHandler`, for the current `viewId`. If no such instance can be obtained, call `facesContext.renderResponse()`. Otherwise, call `getViewMetadata()` on the `ViewDeclarationLanguage` instance. If the result is non-`null`, call

`createViewMetadata()` on the `ViewMetadata` instance. Call `ViewMetadata.getViewParameters()`. If the result is a non-empty `Collection`, **do not** call `facesContext.renderResponse()`, otherwise do call `facesContext.renderResponse()`. If it turns out that the previous call to `createViewMetadata()` did not create a `UIViewRoot` instance, call `createView()` on the `ViewHandler`.

Call `renderResponse()` on the `FacesContext`.

- Publish an `PostAddToViewEvent` with the created `UIViewRoot` as the event source.[P1-end]

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

2.2.2 Apply Request Values

The purpose of the *Apply Request Values* phase of the request processing lifecycle is to give each component the opportunity to update its current state from the information included in the current request (parameters, headers, cookies, and so on). When the information from the current request has been examined to update the component's current state, the component is said to have a "local value".

[P1-start-applyRequestDecode]During the *Apply Request Values* phase, the JSF implementation must call the `processDecodes()` method of the `UIViewRoot` of the component tree.[P1-end] 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. [P1-start-partialDecode] The `processDecodes()` method must determine if the current request is a "partial request" by calling `FacesContext.isPartialRequest()`. If `FacesContext.isPartialRequest()` returns `true`, perform the sequence of steps as outlined in Section 2.2.2.1 "Apply Request Values Partial Processing".[P1-end] Details of the decoding process follow. [P1-start-applyRequestConversion]For `UIInput` components, data conversion must occur as described in the `UIInput` Javadocs.[P1-end]

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

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

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

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

If any of the `decode()` methods that were invoked, or an event listener that processed a queued event, called `responseComplete()` on the `FacesContext` instance for the current request, clear the remaining events from the event queue and terminate lifecycle processing of the current request. [P1-start-applyRequestComplete]If any of the `decode()` methods that were invoked, or an event listener that processed a queued event, called `renderResponse()` on the `FacesContext` instance for the current request, clear the remaining events from the event queue and transfer control to the *Render Response* phase of the request processing lifecycle. Otherwise, control must proceed to the *Process Validations* phase.[P1-end]

2.2.2.1 Apply Request Values Partial Processing

[P1-start-apply-partial-processing]Call FacesContext.getPartialViewContext(). Call PartialViewContext.processPartial() passing the FacesContext, PhaseID.APPLY_REQUEST_VALUES as arguments. [P1-end]

2.2.3 Process Validations

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

[P1-start-validation]During the *Process Validations* phase of the request processing lifecycle, the JSF implementation must call the `processValidators()` method of the `UIViewRoot` of the tree.[P1-end] This will normally cause the `processValidators()` method of each component in the tree to be called recursively, as described in the API reference for the `UIComponent.processValidators()` method. [P1-start-partialValidate] The `processValidators()` method must determine if the current request is a "partial request" by calling `FacesContext.isPartialRequest()`. If `FacesContext.isPartialRequest()` returns `true`, perform the sequence of steps as outlined in Section 2.2.3.1 "Partial Validations Partial Processing".[P1-end] Note that `EditableValueHolder` components whose `immediate` property is set to `true` will have had their conversion and validation processing performed during *Apply Request Values* phase.

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

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

If any of the `validate()` methods that were invoked, or an event listener that processed a queued event, called `responseComplete()` on the `FacesContext` instance for the current request, clear the remaining events from the event queue and terminate lifecycle processing of the current request. [P1-start-validationValidate]If any of the `validate()` methods that were invoked, or an event listener that processed a queued event, called `renderResponse()` on the `FacesContext` instance for the current request, clear the remaining events from the event queue and transfer control to the *Render Response* phase of the request processing lifecycle. Otherwise, control must proceed to the *Update Model Values* phase.[P1-end]

2.2.3.1 Partial Validations Partial Processing

[P1-start-val-partial-processing]Call FacesContext.getPartialViewContext(). Call PartialViewContext.processPartial() passing the FacesContext, PhaseID.PROCESS_VALIDATIONS as arguments. [P1-end]

2.2.4 Update Model Values

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

[P1-start-updateModel]During the *Update Model Values* phase, the JSF implementation must call the `processUpdates()` method of the `UIViewRoot` component of the tree.[P1-end] This will normally cause the `processUpdates()` method of each component in the tree to be called recursively, as described in the API reference for the `UIComponent.processUpdates()` method. [P1-start-partialUpdate] The `processUpdates()` method must determine if the current request is a "partial request" by calling `FacesContext.isPartialRequest()`. If `FacesContext.isPartialRequest()` returns `true`, perform the sequence of steps as outlined in Section 2.2.4.1 "Update Model Values Partial Processing". [P1-end]The actual model update for a particular component is done in the `updateModel()` method for that component.

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

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

If any of the `updateModel()` methods that were invoked, or an event listener that processed a queued event, called `responseComplete()` on the `FacesContext` instance for the current request, clear the remaining events from the event queue and terminate lifecycle processing of the current request. [P1-start-updateModelComplete]If any of the `updateModel()` methods that was invoked, or an event listener that processed a queued event, called `renderResponse()` on the `FacesContext` instance for the current request, clear the remaining events from the event queue and transfer control to the *Render Response* phase of the request processing lifecycle. Otherwise, control must proceed to the *Invoke Application* phase.[P1-end]

2.2.4.1 Update Model Values Partial Processing

[P1-start-upd-partial-processing]Call FacesContext.getPartialViewContext(). Call PartialViewContext.processPartial() passing the FacesContext, PhaseID.UPDATE_MODEL_VALUES as arguments. [P1-end]

2.2.5 Invoke Application

If this phase of the request processing lifecycle is reached, it is assumed that all model updates have been completed, and any remaining event broadcast to the application needs to be performed. [P1-start-invokeApplication]The implementation must ensure that the `processApplication()` method of the `UIViewRoot` instance is called.[P1-end] The default behavior of this method will be to broadcast any queued events that specify a phase identifier of `PhaseId.INVOKE_APPLICATION`. If `responseComplete()`was called on the `FacesContext` instance for the current request, clear the remaining events from the event queue and terminate lifecycle processing of the current request. If `renderResponse()` was called on the `FacesContext` instance for the current request, clear the remaining events from the event queue.

Advanced applications (or application frameworks) may replace the default `ActionListener` instance by calling the `setActionListener()` method on the `Application` instance for this application. [P1-start-invokeApplicationListener]However, the JSF implementation must provide a default `ActionListener` instance that behaves as described in Section 7.1.1 "ActionListener Property".[P1-end]

2.2.6 Render Response

This phase accomplishes two things:

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

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

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

Because of the number of possible options, the mechanism for implementing the *Render Response* phase cannot be specified precisely. [P1-start-renderResponse]However, all JSF implementations of this phase must conform to the following requirements:

- If it is possible to obtain a `ViewDeclarationLanguage` instance for the current `viewId`, from the `ViewHandler`, its `buildView()` method must be called.
- Publish the `javax.faces.event.PreRenderViewEvent`.
- JSF implementations must provide a default `ViewHandler` implementation that is capable of handling views written in JSP as well as views written in the Faces Page Description Language (VDL). In the case of JSP, the ViewHandler must perform a `RequestDispatcher.forward()` call to a web application resource whose context-relative path is equal to the view identifier of the component tree.
- If all of the response content is being derived from the encoding methods of the component or associated `Renderer`s, the component tree should be walked in the same depth-first manner as was used in earlier phases to process the component tree, but subject to the additional constraints listed here. Generally this is handled by a call to `ViewHandler.renderView()`.
- If the response content is being interleaved from additional sources and the encoding methods, the components may be selected for rendering in any desired order[1].
- During the rendering process, additional components may be added to the component tree based on information available to the `ViewHandler` implementation[2]. However, before adding a new component, the `ViewHandler` implementation must first check for the existence of the corresponding component in the component tree. If the component already exists (perhaps because a previous phase has pre-created one or more components), the existing component's properties and attributes must be utilized.
- Under no circumstances should a component be selected for rendering when its parent component, or any of its ancestors in the component tree, has its `rendersChildren` property set to true. In such cases, the parent or ancestor component must render the content of this child component when the parent or ancestor was selected.
- If the `isRendered()` method of a component returns `false`, the renderer for that component must not generate any markup, and none of its facets or children (if any) should be rendered.
- It must be possible for the application to programmatically modify the component tree at any time during the request processing lifecycle (except during the rendering of the view) and have the system behave as expected. For example, the following must be permitted. Modification of the view during rendering may lead to undefined results. It must be possible to allow components added by the templating system (such as JSP) to be removed from the tree before rendering. It must be possible to programmatically add components to the tree and have them render in the proper place in the hierarchy. It must be possible to re-order components in the tree before rendering. These manipulations do require that any components added to the tree have ids that are unique within the scope of the closest parent `NamingContainer` component. The value of the `rendersChildren` property is handled as expected, and may be either `true` or `false`.
- For partial requests, where partial view rendering is required, there must be no content written outside of the view (outside `f:view`). Response writing must be disabled. Response writing must be enabled again at the start of `encodeBegin`.

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

When each particular component in the component tree is selected for rendering, calls to its `encodeXxx()` methods must be performed in the manner described in Section 3.1.13 “Component Specialization Methods”. For components that implement `ValueHolder` (such as `UIInput` and `UIOutput`), data conversion must occur as described in the `UIOutput` Javadocs.

Upon completion of rendering, the completed state of the view must have been saved using the methods of the class `StateManager.` This state information must be made accessible on a subsequent request, so that the *Restore View* can access it.[P1-end] For more on `StateManager`, see Section 7.7.3 “State Saving Methods.”

2.3 Common Event Processing

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

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

[P1-start-eventBroadcast]For each queued event, the `broadcast()` method of the source `UIComponent` must be called to broadcast the event to all event listeners who have registered an interest, on this source component for events of the specified type, after which the event is removed from the event queue.[P1-end] See the API reference for the `UIComponent.broadcast()` method for the detailed functional requirements.

It is also possible for event listeners to cause additional events to be enqueued for processing during the current phase of the request processing lifecycle. [P1-start-eventOrder]Such events must be broadcast in the order they were enqueued, after all originally queued events have been broadcast, before the lifecycle management method returns.[P1-end]

2.4 Common Application Activities

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

2.4.1 Acquire Faces Object References

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

2.4.1.1 Acquire and Configure Lifecycle Reference

[P1-start-lifeReference]As described in Section 12.1 “Lifecycle”, the JSF implementation must provide an instance of `javax.faces.lifecycle.Lifecycle` that may be utilized to manage the remainder of the request processing lifecycle.[P1-end] An application may acquire a reference to this instance in a portable manner, as follows:

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

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

2.4.1.2 Acquire and Configure FacesContext Reference

[P1-start-contextReference]As described in Section 6.1 “FacesContext”, the JSF implementation must provide an instance of `javax.faces.context.FacesContext` to contain all of the per-request state information for a Faces Request or a Faces Response. An application that is processing a Non-Faces Request, but wants to create a Faces Response, must acquire a reference to a `FacesContext` instance as follows

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

where the `context`, `request`, and `response` objects represent the corresponding instances for the application environment.[P1-end] For example, in a servlet-based application, these would be the `ServletContext`, `HttpServletRequest`, and `HttpServletResponse` instances for the current request.

2.4.2 Create And Configure A New View

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

2.4.2.1 Create A New View

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

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

[P1-start-createViewRoot]The `UIViewRoot` instance returned by the `createView()` method must minimally contain a single `UIViewRoot` provided by the JSF implementation, which must encapsulate any implementation-specific component management that is required.[P1-end] Optionally, a JSF implementation's `ViewHandler` may support the automatic population of the returned `UIViewRoot` with additional components, perhaps based on some external metadata description.

[P1-start-createView]The caller of `ViewHandler.createView()` must cause the `FacesContext` to be populated with the new `UIViewRoot.` Applications must make sure that it is safe to discard any state saved in the view rooted at the `UIViewRoot` currently stored in the `FacesContext`.[P1-end] If Facelets is the page definition language, `FacesContext.setViewRoot()`must be called before returning from `ViewHandler.createView().` Refer to Section 7.5.2 "Default ViewHandler Implementation" for more `ViewHandler` details.

2.4.2.2 Configure the Desired RenderKit

[P1-start-defaultRenderkit]The `UIViewRoot` instance provided by the `ViewHandler`, as described in the previous subsection, must automatically be configured to utilize the default `javax.faces.render.RenderKit` implementation provided by the JSF implementation, as described in Section 8.1 "RenderKit". This `RenderKit` must support the standard components and `Renderer`s described later in this specification, to maximize the portability of your application.[P1-end]

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

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

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

[P1-start-calcRenderkitId]The default `ViewHandler` must call `calculateRenderKitId()` on itself and set the result into the `UIViewRoot`'s `renderKitId` property.[P1-end] This allows applications that use alternative `RenderKit`s to dynamically switch on a per-view basis.

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

2.4.2.3 Configure The View's Components

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

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

2.4.2.4 Store the new View in the FacesContext

[P1-start-setViewRoot]Once the view has been created and configured, the `FacesContext` instance for this request must be made aware of it by calling `setViewRoot()`.[P1-end]

2.5 Concepts that impact several lifecycle phases

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

2.5.1 Value Handling

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

2.5.1.1 Apply Request Values Phase

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

2.5.1.2 Process Validators Phase

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

2.5.1.3 Executing Validation

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

2.5.1.4 Update Model Values Phase

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

2.5.2 Localization and Internationalization (L10N/I18N)

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

2.5.2.1 Determining the active `Locale`

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

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

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

The `UIViewRoot`'s `Locale` is determined and set by the `ViewHandler` during the execution of the `ViewHandler`'s `createView()` method. [P1-start-locale]This method must cause the active `Locale` to be determined by looking at the user's preferences combined with the application's stated supported locales.[P1-end] Please see the javadocs for details.

The application can call `UIViewRoot.setLocale()` directly, but it is also possible for the page author to override the `UIViewRoot`'s locale by using the `locale` attribute on the `<f:view>` tag. [P1-start-localeValue]The value of this attribute must be specified as `language[{-|_}country[{-|_}variant]]` without the colons, for example "`ja_JP_SJIS`". The separators between the segments must be '`-`' or '_'.[P1-end]

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

2.5.2.2 Determining the Character Encoding

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

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

[P1-start-setLocale]At the beginning of the render-response phase, the ViewHandler must ensure that the response Locale is set to be that of the UIViewRoot, for example by calling `ServletResponse.setLocale()` when running in the servlet environment.[P1-end] Setting the response Locale may affect the response character encoding, see the Servlet and Portlet specifications for details.

[P1-start-encoding]At the end of the render-response phase, the ViewHandler must store the response character encoding used by the underlying response object (e.g., the servlet or portlet response) in the session (if and only if a session already exists) under a well known, implementation-dependent key.

On a subsequent postback, before any of the ExternalContext methods for accessing request parameters are invoked, the ViewHandler must examine the Content-Type header to read the charset attribute and use its value to set it as the request encoding for the underlying request object. If the Content-Type header doesn't contain a charset attribute, the encoding previously stored in the session (if and only if a session already exists), must be used to set the encoding for the underlying request object. If no character encoding is found, the request encoding must be left unmodified.[P1-end]

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

2.5.2.3 Localized Text

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

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

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

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

2.5.2.4 Localized Application Messages

This section describes how JSF handles localized error and informational messages that occur as a result of conversion, validation, or other application actions during the request processing lifecycle. The JSF class `javax.faces.application.FacesMessage` is provided to encapsulate summary, detail, and severity information for a message. [P1-start-bundle]A JSF implementation must provide a `javax.faces.Messages ResourceBundle` containing all of the necessary keys for the standard messages. The required keys (and a non-normative indication of the intended message text) are as follows:

- javax.faces.component.UIInput.CONVERSION -- {0}: Conversion error occurred
- javax.faces.component.UIInput.REQUIRED -- {0}: Validation Error: Value is required
- javax.faces.component.UIInput.UPDATE -- {0}: An error occurred when processing your submitted information
- javax.faces.component.UISelectOne.INVALID -- {0}: Validation Error: Value is not valid
- javax.faces.component.UISelectMany.INVALID -- {0}: Validation Error: Value is not valid
- javax.faces.converter.BigDecimalConverter.DECIMAL={2}: "{0}" must be a signed decimal number.
- javax.faces.converter.BigDecimalConverter.DECIMAL_detail={2}: "{0}" must be a signed decimal number consisting of zero or more digits, that may be followed by a decimal point and fraction. Example: {1}
- javax.faces.converter.BigIntegerConverter.BIGINTEGER={2}: "{0}" must be a number consisting of one or more digits.
- javax.faces.converter.BigIntegerConverter.BIGINTEGER_detail={2}: "{0}" must be a number consisting of one or more digits. Example: {1}
- javax.faces.converter.BooleanConverter.BOOLEAN={1}: "{0}" must be 'true' or 'false'.
- javax.faces.converter.BooleanConverter.BOOLEAN_detail={1}: "{0}" must be 'true' or 'false'. Any value other than 'true' will evaluate to 'false'.
- javax.faces.converter.ByteConverter.BYTE={2}: "{0}" must be a number between 0 and 255.
- javax.faces.converter.ByteConverter.BYTE_detail={2}: "{0}" must be a number between 0 and 255. Example: {1}
- javax.faces.converter.CharacterConverter.CHARACTER={1}: "{0}" must be a valid character.
- javax.faces.converter.CharacterConverter.CHARACTER_detail={1}: "{0}" must be a valid ASCII character.
- javax.faces.converter.DateTimeConverter.DATE={2}: "{0}" could not be understood as a date.
- javax.faces.converter.DateTimeConverter.DATE_detail={2}: "{0}" could not be understood as a date. Example: {1}
- javax.faces.converter.DateTimeConverter.TIME={2}: "{0}" could not be understood as a time.
- javax.faces.converter.DateTimeConverter.TIME_detail={2}: "{0}" could not be understood as a time. Example: {1}
- javax.faces.converter.DateTimeConverter.DATETIME={2}: "{0}" could not be understood as a date and time.
- javax.faces.converter.DateTimeConverter.DATETIME_detail={2}: "{0}" could not be understood as a date and time. Example: {1}
- javax.faces.converter.DateTimeConverter.PATTERN_TYPE={1}: A 'pattern' or 'type' attribute must be specified to convert the value "{0}".
- javax.faces.converter.DoubleConverter.DOUBLE={2}: "{0}" must be a number consisting of one or more digits.
- javax.faces.converter.DoubleConverter.DOUBLE_detail={2}: "{0}" must be a number between 4.9E-324 and 1.7976931348623157E308 Example: {1}
- javax.faces.converter.EnumConverter.ENUM={2}: "{0}" must be convertible to an enum.
- javax.faces.converter.EnumConverter.ENUM_detail={2}: "{0}" must be convertible to an enum from the enum that contains the constant "{1}".
- javax.faces.converter.EnumConverter.ENUM_NO_CLASS={1}: "{0}" must be convertible to an enum from the enum, but no enum class provided.
- javax.faces.converter.EnumConverter.ENUM_NO_CLASS_detail={1}: "{0}" must be convertible to an enum from the enum, but no enum class provided.
- javax.faces.converter.FloatConverter.FLOAT={2}: "{0}" must be a number consisting of one or more digits.

- javax.faces.converter.FloatConverter.FLOAT_detail={2}: "{0}" must be a number between 1.4E-45 and 3.4028235E38 Example: {1}
- javax.faces.converter.IntegerConverter.INTEGER={2}: "{0}" must be a number consisting of one or more digits.
- javax.faces.converter.IntegerConverter.INTEGER_detail={2}: "{0}" must be a number between -2147483648 and 2147483647 Example: {1}
- javax.faces.converter.LongConverter.LONG={2}: "{0}" must be a number consisting of one or more digits.
- javax.faces.converter.LongConverter.LONG_detail={2}: "{0}" must be a number between -9223372036854775808 to 9223372036854775807 Example: {1}
- javax.faces.converter.NumberConverter.CURRENCY={2}: "{0}" could not be understood as a currency value.
- javax.faces.converter.NumberConverter.CURRENCY_detail={2}: "{0}" could not be understood as a currency value. Example: {1}
- javax.faces.converter.NumberConverter.PERCENT={2}: "{0}" could not be understood as a percentage.
- javax.faces.converter.NumberConverter.PERCENT_detail={2}: "{0}" could not be understood as a percentage. Example: {1}
- javax.faces.converter.NumberConverter.NUMBER={2}: "{0}" is not a number.
- javax.faces.converter.NumberConverter.NUMBER_detail={2}: "{0}" is not a number. Example: {1}
- javax.faces.converter.NumberConverter.PATTERN={2}: "{0}" is not a number pattern.
- javax.faces.converter.NumberConverter.PATTERN_detail={2}: "{0}" is not a number pattern. Example: {1}
- javax.faces.converter.ShortConverter.SHORT={2}: "{0}" must be a number consisting of one or more digits.
- javax.faces.converter.ShortConverter.SHORT_detail={2}: "{0}" must be a number between -32768 and 32767 Example: {1}
- javax.faces.converter.STRING={1}: Could not convert "{0}" to a string.
- javax.faces.validator.BeanValidator.MESSAGE -- {0}
- javax.faces.validator.DoubleRangeValidator.MAXIMUM -- {1}: Validation Error: Value is greater than allowable maximum of ''{0}''
- javax.faces.validator.DoubleRangeValidator.MINIMUM -- {1}: Validation Error: Value is less than allowable minimum of ''{0}''
- javax.faces.validator.DoubleRangeValidator.NOT_IN_RANGE -- {2}: Validation Error: Specified attribute is not between the expected values of {0} and {1}.
- javax.faces.validator.DoubleRangeValidator.TYPE -- {0}: Validation Error: Value is not of the correct type
- javax.faces.validator.LengthValidator.MAXIMUM -- {1}: Validation Error: Value is greater than allowable maximum of ''{0}''
- javax.faces.validator.LengthValidator.MINIMUM -- {1}: Validation Error: Value is less than allowable minimum of ''{0}''
- javax.faces.validator.LongRangeValidator.MAXIMUM -- {1}: Validation Error: Value is greater than allowable maximum of ''{0}''
- javax.faces.validator.LongRangeValidator.MINIMUM -- {1}: Validation Error Value is less than allowable minimum of ''{0}''
- javax.faces.validator.LongRangeValidator.NOT_IN_RANGE={2}: Validation Error: Specified attribute is not between the expected values of {0} and {1}.
- javax.faces.validator.LongRangeValidator.TYPE -- {0}: Validation Error: Value is not of the correct type

The following message keys are deprecated:

- javax.faces.validator.NOT_IN_RANGE -- Specified attribute is not between the expected values of {0} and {1}[P1-end]

A JSF application may provide its own messages, or overrides to the standard messages by supplying a `<message-bundle>` element to in the application configuration resources. Since the `ResourceBundle` provided in the Java platform has no notion of summary or detail, JSF adopts the policy that `ResourceBundle` key for the message looks up the message summary. The detail is stored under the same key as the summary, with `_detail` appended. [P1-start-bundleKey]These `ResourceBundle` keys must be used to look up the necessary values to create a localized `FacesMessage` instance. Note that the value of the summary and detail keys in the `ResourceBundle` may contain parameter substitution tokens, which must be substituted with the appropriate values using `java.text.MessageFormat`.[P1-end] Replace the last parameter substitution token shown in the messages above with the input component's `label` attribute. For example, `{1}` for `"DoubleRangeValidator.MAXIMUM", {2}` for `"ShortConverter.SHORT".` The `label` attribute is a generic attribute. Please see *Section 3.1.11 "Generic Attributes"* and *Section 8.6 "Standard HTML RenderKit Implementation"* for more information on these attributes. If the input component's `label` attribute is not specified, use the component's client identifier.

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

[P1-start-facesMessage]The following algorithm must be used to create a `FacesMessage` instance given a message key.

- Call `getMessageBundle()` on the `Application` instance for this web application, to determine if the application has defined a resource bundle name. If so, load that ResourceBundle and look for the message there.
- If not there, look in the `javax.faces.Messages` resource bundle.
- In either case, if a message is found, use the above conventions to create a `FacesMessage` instance.[P1-end]

2.5.3 State Management

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

2.5.3.1 State Management Considerations for the Custom Component Author

Since the component developer cannot know what the state saving method will be at runtime, they must be aware of state management. As shown in *Section FIGURE 4-1 "The* javax.faces.component package", all JSF components implement the `StateHolder` interface. As a consequence the standard components provide implementations of `StateHolder` to suit their needs. [P1-start-componentStateHolder]A custom component that extends `UIComponent` directly, and does not extend any of the standard components must implement `StateHolder` manually.[P1-end]Please see *Section 3.2.4 "StateHolder"* for details.

A custom component that **does** extend from one of the standard components and maintains its own state, in addition to the state maintained by the superclass must take special care to implement `StateHolder` correctly. [P1-start-saveState]Notably, calls to `saveState()` must not alter the state in any way.[P1-end] The subclass is responsible for saving and restoring the state of the superclass. Consider this example. My custom component represents a "slider" ui widget. As such, it needs to keep track of the maximum value, minimum value, and current values as part of its state.

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

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

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

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

2.5.3.2 State Management Considerations for the JSF Implementor

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

Key Players in State Management

- `ViewHandler` the entry point to the state management system. Uses a helper class, `StateManager`, to do the actual work. In the JSP case, delegates to the tag handler for the `<f:view>` tag for some functionality.
- `StateManager` abstraction for the hard work of state saving. Uses a helper class, `ResponseStateManager`, for the rendering technology specific decisions.
- `ResponseStateManager` abstraction for rendering technology specific state management decisions.
- `UIComponent` directs process of saving and restoring individual component state.

2.5.4 Resource Handling

This section only applies to pages written using Facelets for JSF 2 and later. Section 2.6 “Resource Handling” is the starting point for the normative specification for Resource Handling. This section gives a non-normative overview of the feature. The following steps walk through the points in the lifecycle where this feature is encountered. Consider a Faces web application that contains resources that have been packaged into the application as specified in Section 2.6.1 “Packaging Resources”. Assume each page in the application includes references to resources, specifically scripts and stylesheets. The first diagram in this chapter is helpful in understanding this example.

Consider an initial request to the application.

- The `ViewHandler` calls `ViewDeclarationLanguage.buildView()`. This ultimately causes the `processEvent()` method for the `javax.faces.resource.Script` and `javax.faces.resource.Stylesheet` renderers (which implement `ComponentSystemEventListener)` to be called after each component that declares them as their renderer is added to the view. This method is specified to take actions that cause the resource to be rendered at the correct part in the page based on user-specified or application invariant rules. Here’s how it works.

  Every `UIComponent` instance in a view is created with a call to some variant of `Application.createComponent()`. The specification for this method now includes some annotation processing requirements. If the component or its renderer has an `@ListenerFor` or `@ListenersFor` annotation, and the `Script` and `Stylesheet` renderers must, the component or its renderer are added as a component scoped listener for the appropriate event. In the case of `Script` and `Stylesheet` renderers, they must listen for the `PostAddToViewEvent`.

  When the `processEvent()` method is called on a `Script` or `Stylesheet` renderer, the renderer takes the specified action to move the component to the proper point in the tree based on what kind of resource it is, and on what hints the page author has declared on the component in the view.

- The `ViewHandler` calls `ViewDeclarationLanguage.renderView()`. The view is traversed as normal and because the components with `Script` and `Stylesheet` renderers have already been reparented to the proper place in the view, the normal renderering causes the resource to be encoded as described in Section 2.6.2 “Rendering Resources”.

The browser then parses the completely rendered page and proceeds to issue subsequent requests for the resources included in the page.

Now consider a request from the browser for one of those resources included in the page.

- The request comes back to the Faces server. The `FacesServlet` is specified to call `ResourceHandler.isResourceRequest()` as shown in the diagram in Section 2.1.2 “Faces Request Generates Faces Response”. In this case, the method returns `true`. The `FacesServlet` is specified to call `ResourceHandler.handleResourceRequest()` to serve up the bytes of the resource.

2.5.5 View Parameters

This section only applies to pages written using Facelets for JSF 2 and later. The normative specification for this feature is spread out across several places, including the Page Declaration Language Documentation for the `<f:metadata>` element, the javadocs for the `UIViewParameter`, `ViewHandler`, and `ViewDeclarationLanguage` classes, and the spec language requirements for the default `NavigationHandler` and the Request Processing Lifecycle. This leads to a very diffuse field of specification requirements. To aid in understanding the feature, this section provides a non-normative overview of the feature. The following steps walk through the points in the lifecycle where this feature is encountered. Consider a web application that uses this feature exclusively on every page. Therefore every page has the following features in common.

- Every page has an `<f:metadata>` tag, with at least one `<f:viewParameter>` element within it.
- Every page has at least one `<h:link>` or `<h:button>` with the appropriate parameters nested within it.
- No other kind of navigation components are used in the application.

Consider an initial request to the application.

- As specified in section Section 2.2.1 “Restore View”, the restore view phase of the request processing lifecycle detects that this is an initial request and tries to obtain the `ViewDeclarationLanguage` instance from the `ViewHandler` for this `viewId`. Because every page in the app is written in Facelets for JSF 2.0, there is a `ViewDeclarationLanguage` instance. Restore view phase calls `ViewDeclarationLanguage.getViewMetadata()`. Because every view in this particular app does have `<f:metadata>` on every page, this method returns a `ViewMetadata` instance. Restore view phase calls `MetaData.createMetadataView()`. This method creates a `UIViewRoot` containing only children declared in the `<f:metadata>` element. Restore view phase calls `ViewMetadata.getViewParameters()`. Because every `<f:metadata>` in the app has at least one `<f:viewParameter>` element within it, this method returns a non empty `Collection<UIViewParameter>`. Restore view phase uses this fact to decide that the lifecycle **must not** skip straight to render response, as is the normal action taken on initial requests.
- The remaining phases of the request processing lifecycle execute: apply request values, process validations, update model values, invoke application, and finally render response. Because the view only contains `UIViewParameter` children, only these children are traversed during the lifecycle, but because this is an initial request, with no query parameters, none of these compnents take any action during the lifecycle.
- Because the pages exclusively use `<h:link>` and `<h:button>` for their navigation, the renderers for these components are called during the rendering of the page. As specified in the renderkit docs for the renderers for those components, markup is rendered that causes the browser to issue a GET request with query parameters.

Consider when the user clicks on a link in the application. The browser issues a GET request with query parameters

- Restore view phase takes the same action as in the previously explained request. Because this is a GET request, no state is restored from the previous request.
- Because this is a request with query parameters, the `UIViewParameter` children **do** take action when they are traversed during the normal lifecycle, reading values during the apply request values phase, doing conversion and processing validators attached to the `<f:viewParam>` elements, if any, and updating models during the update model values phase. Because there are only `<h:link>` and `<h:button>` navigation elements in the page, no action action will happen during the invoke application phase. The response is re-rendered as normal. In such an application, the only navigation to a new page happens by virtue of the browser issuing a GET request to a different viewId.

2.5.6 Bookmarkability

Prior to JSF 2, every client server interaction was an HTTP POST. While this works fine in many situations, it does not work well when it comes to bookmarking pages in a web application. Version 2 of the specification introduces bookmarking capability with the use of two new Standard HTML RenderKit additions.

Provided is a new component (UIOutcomeTarget) that provides properties that are used to produce a hyperlink at render time. The component can appear in the form of a button or a link. This feature introduces a concept known as “preemptive navigation”, which means the target URL is determined at Render Response time - before the user has activated the component. This feature allows the user to leverage the navigation model while also providing the ability to generate bookmarkable non-faces requests.

2.5.7 JSR 303 Bean Validation

Version 2 of the specification introduces support for JSR 303 Bean Validation. [p1-beanValidationRequired]A JSF implentation must support JSR 303 Bean Validation if the environment in which the JSF runtime is included requires JSR 303 Bean Validation. Currently the only such environment is when JSF is included in a Java EE 6 runtime.[p1-end]

A detailed description of the usage of Bean Validation with JSF is beyond the scope of this section, but this section will provide a brief overview of the feature, touching on the points of interest to a spec implementor. Consider a simple web application that has one page, written in Facelets for JSF 2, that has a several text fields inside of a form. This

application is running in a JSF runtime in an environment that does require JSR 303 Bean Validation, and therefore this feature is available. Assume that every text field is bound to a managed bean property that has at leas one Bean Validation constraint annotation attached to it.

During the render response phase that always precedes a postback, due to the specification requirements in Section 3.5.3 “Validation Registration”, every `UIInput` in this application has an instance of `Validator` with id `javax.faces.Bean` attached to it.

During the process validations phase, due to the specification for the `validate()` method of this `Validator`, Bean Validation is invoked automatically, for the user specified validation constraints, whenever such components are normally validated. The `javax.faces.Bean` standard validator also ensures that every `ConstraintViolation` that resulted in attempting to validate the model data is wrapped in a `FacesMessage` and added to the `FacesContext` as normal with every other kind of validator.

2.5.8 Ajax

JSF and Ajax have been working well together for a number of years. this has led to the sprouting of many JSF Ajax frameworks. Although many of these frameworks may appear different, they all contribute to a dynamic request response experience. The variations in the way these frameworks provide that experience causes component compatibility problems when using components from different libraries together in the same web application.

JSF 2 introduces Ajax into the specification, and it builds upon important concepts from a variety of existing JSF Ajax frameworks. The specification introduces a JavaScript library for performing basic Ajax operations. The library helps define a standard way of sending an Ajax request, and processing an Ajax response, since these are problem areas for component compatability. The specification provides two ways of adding Ajax to JSF web applications. Page authors may use the JavaScript library directly in their pages by attaching the Ajax request call to a JSF component via a JavaScript event (such as onclick). They may also take a more declarative aproach and use a core Facelets tag (<f:ajax/>) that they can nest within JSF components to “Ajaxify” them. It is also possible to “Ajaxify” regions of a page by “wrapping” the tag around component groups.

The server side aspects of JSF Ajax frameworks work with the standard JSF lifecycle. In addition to providing a standard page authoring experience, the specification also standardizes the server side processing of Ajax requests. Selected components in a JSF view can be priocessed (known as partial processing) and selected components can be rendered to the client (known as partial rendering).

2.5.9 Component Behaviors

The JSF 2 specification introduces a new type of attached object known as component behaviors. Component behaviors play a similar role to converters and validators in that they are attached to a component instance in order to enhance the component with additional functionality not defined by the component itself. While converters and validators are currently limited to the server-side request processing lifecycle, component behaviors have impact that extends to the client, within the scope of a particular instance component in a view. In particular, the `ClientBehavior` interface defines a contract for behaviors that can enhance a component's rendered content with behavior-defined "scripts". These scripts are executed on the client in response to end user interaction, but can also trigger postbacks back into the JSF request processing lifecycle.

The usage pattern for client behaviors is as follows:

- The page author attaches a client behavior to a component, typically by specifying a behavior tag as a child of a component tag.
- When attaching a client behavior to a component, the page author identifies the name of a client "event" to attach to. The set of valid events are defined by the component.
- At render time, the component (or renderer) retrieves the client behavior and asks it for its script.
- The component (or renderer) renders this script at the appropriate location in its generated content (eg. typically in a DOM event handler).

- When the end user interacts with the component's content in the browser, the behavior-defined script is executed in response to the page author-specified event.
- The script provides some client-side interaction, for example, hiding or showing content or validating input on the client, and possibly posts back to the server.

The first client behavior to provided by the JSF specification is the `AjaxBehavior`. This behavior is exposed to a page author as a Facelets `<f:ajax>` tag, which can be embedded within any of the standard HTML components as follows:

```
<h:commandButton>
    <f:ajax event="mouseover"/>
</h:commandButton>
```

When activated in response to end user activity, the `<f:ajax>` client behavior generates an Ajax request back into the JSF request processing lifecycle.

The component behavior framework is extensible and allows developers to define custom behaviors and also allows component authors to enhance custom components to work with behaviors.

2.5.10 System Events

System Events are normatively specified in Section 3.4.3 “System Events”. This section provides an overview of this feature as it relates to the lifecycle.

System events expand on the idea of lifecycle `PhaseEvent`s. With `PhaseEvent`s, it is possible to have application scoped `PhaseListeners` that are given the opportunity to act on the system before and after each phase in the lifecycle. System events provide a much more fine grained insight into the system, allowing application **or** component scoped listeners to be notified of a variety of kinds of events. The set of events supported in the core specification is given in Section 3.4.3.1 “Event Classes”. To accomodate extensibility, users may define their own kinds of events.

The system event feature is a simple publish/subscribe event model. There is no event queue, events are published immediately, and always with a call to `Application.publishEvent()`. There are several ways to declare interest in a particular kind of event.

- Call `Application.subscribeToEvent()` to add an application scoped listener.
- Call `UIComponent.subscribeToEvent()` to add a component scoped listener.
- Use the `<f:event>` tag to declare a component scoped listener.
- Use the `@ListenerFor` or `@ListenersFor` annotation. The scope of the listener is determined by the code that processes the annotation.
- Use the `<system-event-listener>` element in an application configuration resource to add an application scoped listener.

This feature is conceptually related to the lifecycle because there are calls to `Application.publishEvent()` sprinkled throughout the code that gets executed when the lifecycle runs.

2.6 Resource Handling

As shown in the diagram in Section 2.1.2 “Faces Request Generates Faces Response”, [P1-start isResourceRequest rules] the JSF run-time must determine if the current Faces Request is a *Faces Resource Reques*t or a *View Request*. This must be accomplished by calling `Application.getResourceHandler().isResourceRequest()`. [P1-end] Most of the normative specification for resource handling is contained in the Javadocs for `ResourceHandler` and its related classes. This section contains the specification for resource handling that fits best in prose, rather than in Javadocs.

2.6.1 Packaging Resources

ResourceHandler defines a path based packaging convention for resources. The default implementation of ResourceHandler must support packaging resources in the web application root or in the classpath, according to the following specification.Other implementations of ResourceHandler are free to package resources however they like.

2.6.1.1 Packaging Resources into the Web Application Root

[P1-start web app packaging] The default implementation must support packaging resources in the web application root under the path

`resources/<resourceIdentifier>`

relative to the web app root. Resources packaged into the web app root must be accessed using the `getResource*()` methods on `ExternalContext.`[P1-end]

2.6.1.2 Packaging Resources into the Classpath

[P1-start classpath packaging]For the default implementation, resources packaged in the classpath must reside under the JAR entry name:

`META-INF/resources/<resourceIdentifier>`

Resources packaged into the classpath must be accessed using the `getResource*()` methods of the `ClassLoader` obtained by calling the `getContextClassLoader()` method of the curreth `Thread`.[P1-end]

2.6.1.3 Resource Identifiers

`<resourceIdentifier>` consists of several segments, specified as follows.

[P1-start requirements for something to be considered a valid resourceIdentifier]

`[localePrefix/][libraryName/][libraryVersion/]resourceName[/resourceVersion]`

The run-time must enforce the following rules to consider a `<resourceIdentifier>`s valid. A `<resourceIdentifier>` that does not follow these rules must not be considered valid and must be ignored silently.

- Every character in a resource identifier must be a valid character suitable for use in a string passed to the constructor of `java.io.File` that takes a single `String` argument.
- Segments in square brackets [] are optional.
- The segments must appear in the order shown above.
- If *libraryVersion* is present, it must be preceded by *libraryName.*

- If *libraryVersion* is present, any leaf files under *libraryName* must be ignored.
- If *resourceVersion* is present, it must be preceded by *resourceName.*
- There must be a '/' between adjacent segments in a `<resourceIdentifier>`
- If *libraryVersion* or *resourceVersion* are present, both must be a '_' separated list of integers, neither starting nor ending with '_'
- If *resourceVersion* is present, it must be a version number in the same format as *libraryVersion*. An optional "file extension" may be used with the *resourceVersion.* If "file extension" is used, a "." character, followed by a "file extension" must be appended to the version number. See the following table for an example.

[P1-end]

The following examples illustrate the nine valid combinations of the above resource identifier segments.

localePrefix [optional]	libraryName [optional]	library Version [optional]	resourceName [required]	resource Version [optional]	Description	actual resourceIdentifier
			duke.gif		A non-localized, non-versioned image resource called 'duke.gif', not in a library	duke.gif
	corporate		duke.gif		A non-localized, non-versioned image resource called 'duke.gif' in a library called 'corporate'	corporate/duke.gif
	corporate	2_3	duke.gif		A non-localized, non-versioned image resource called 'duke.gif', in version 2_3 of the 'corporate' library	corporate/2_3/duke.gif
	basic	2_3	script.js	1_3_4.js	A non-localized, version 1.3.4 script resource called "script.js", in versioned 2_3 library called "basic".	basic/2_3/script.js/1_3_4.js
de			header.css		A non-versioned style resource called 'header.css' localized for locale "de"	de/header.css
de_AT			footer.css	1_4_2.css	Version 1_4_2 of style resource 'footer.css', localized for locale 'de_AT"	de_AT/footer.css/1_4_2.css
zh	extraFancy		menu-bar.css	2_4.css	Version 2_4 of style resource called, "menu-bar.css" in non-versioned library, 'extraFancy", localized for locale "zh"	zh/extraFancy/menu-bar.css/2_4.css
ja	mild	0_1	ajaxTransaction.js		Non-versioned script resource called, 'ajaxTransaction.js", in version 0_1 of library called "mild", localized for locale "ja"	ja/mild/0_1/ajaxTransaction.js
de_ch	grassy	1_0	bg.png	1_0.png	Version 1_0 of image resource called 'bg.png", in version 1_0 of library called 'grassy" localized for locale "de_ch"	de_ch/grassy/1_0/bg.png/1_0.png

2.6.1.4 Libraries of Localized and Versioned Resources

An important feature of the resource handler is the ability for resources to be localized, versioned, and collected into libraries. The localization and versioning scheme is completely hidden behind the API of `ResourceHandler` and `Resource` and is not exposed in any way to the JSF run-time.

[P1-start resource versioning] The default implementation of `ResourceHandler.createResource()`, for all variants of that method, must implement the following to discover which actual resource will be encapsulated within the returned `Resource` instance. An implementation may perform caching of the resource metadata to improve performance if the `ProjectStage` is `ProjectStage.Production`.

Using the *resourceName* and *libraryName* arguments to `createResource()`, and the resource packaging scheme specified in Section 2.6.1.1 “Packaging Resources into the Web Application Root”, Section 2.6.1.2 “Packaging Resources into the Classpath”, and Section 2.6.1.3 “Resource Identifiers”, discover the file or entry that contains the bytes of the resource. If there are multiple versions of the same library, and *libraryVersion* is not specified, the library with th highest version is chosen. If there are multiple versions of the same resource, and *resourceVersion* is not specified, the resource with the highest version is chosen. The algorithm is specified in pseudocode.

```
function createResource(resourceName, libraryName) {
  var prefix = web app root resource prefix;
  var resourceLoader = web app resource loader;
  // these are shorthand for the prefix and resource loading
  // facility specified in Section 2.6.1.1. They are
  // not actual API per se.
  var resource = null;
  var resourceId = deriveResourceId(prefix, resourceLoader,
        resourceName, libraryName);
  if (null == resourceId) {
    prefix = classpath resource prefix;
    resourceLoader = classpath resource loader;
    // these are shorthand for the prefix and resource
    // loading facility specified in Section 2.6.1.2. They are
    // not actual API per se.
    resourceId = deriveResourceId(prefix, resourceLoader,
        resourceName, libraryName);
  }
  if (null != resourceId) {
    resource = create the resource using the resourceId;
  }
  return resource;
}

function deriveResourceId(prefix, resourceLoader,
                          resourceName, libraryName) {
  var localePrefix = getLocalePrefix();
  var resourceVersion = null;
```

```
var libraryVersion = null;
var resourceId;
if (null != localePrefix) {
  prefix = localePrefix + ‘/’ + prefix;
}
if (null != libraryName) {
  var libraryPaths = resourceLoader.getResourcePaths(
    prefix + ‘/’ + libraryName);
  if (null != libraryPaths && !libraryPaths.isEmpty()) {
    libraryVersion = // execute the comment
    // Look in the libraryPaths for versioned libraries.
    // If one or more versioned libraries are found, take
    // the one with the “highest” version number as the value
    // of libraryVersion. If no versioned libraries
    // are found, let libraryVersion remain null.
  }
  if (null != libraryVersion) {
    libraryName = libraryName + ‘/’ + libraryVersion;
  }
  var resourcePaths = resourceLoader.getResourcePaths(
    prefix + ‘/’ + libraryName + ‘/’ + resourceName);
  if (null != resourcePaths && !resourcePaths.isEmpty()) {
    resourceVersion = // execute the comment
    // Look in the resourcePaths for versioned resources.
    // If one or more versioned resources are found, take
    // the one with the “highest” version number as the value
    // of resourceVersion. If no versioned libraries
    // are found, let resourceVersion remain null.
  }
  if (null != resourceVersion) {
    resourceId = prefix + ’/’ + libraryName + ’/’ +
      resourceName + ‘/’ + resourceVersion;
  }
  else {
    resourceId = prefix + ’/’ + libraryName + ’/’ + resourceName;
  }
} // end of if (null != libraryName)
else {
  // libraryName == null
  var resourcePaths = resourceLoader.getResourcePaths(
    prefix + ‘/’ + resourceName);
  if (null != resourcePaths && !resourcePaths.isEmpty()) {
```

```
      resourceVersion = // execute the comment
      // Look in the resourcePaths for versioned resources.
      // If one or more versioned resources are found, take
      // the one with the “highest” version number as the value
      // of resourceVersion. If no versioned libraries
      // are found, let resourceVersion remain null.
    }
    if (null != resourceVersion) {
      resourceId = prefix + ’/’ + resourceName + ’/’ +
        resourceVersion;
    }
    else {
      resourceId = prefix + ’/’ + resourceName;
    }
  } // end of else, when libraryName == null
  return resourceId;
}

function getLocalePrefix() {
  var localePrefix;
  var appBundleName = facesContext.application.messageBundle;
  if (null != appBundleName) {
    var locale =
      facesContext.application.viewHandler.calculateLocale();
    ResourceBundle appBundle = ResourceBundle.getBundle(
      appBundleName, locale);
    localePrefix = appBundle.getString(ResourceHandler.
      LOCALE_PREFIX);
  }
  // Any MissingResourceException instances that are encountered
  // in the above code must be swallowed by this method, and null
  // returned;
  return localePrefix;
}
```

[P1-end]

2.6.2 Rendering Resources

Resources such as images, stylesheets and scripts use the resource handling mechanism as outlined in Section 2.6.1 “Packaging Resources”. So, for example:

```
<h:graphicImage name=”Planets.gif” library=”images”/>
<h:graphicImage value=”#{resource[‘images:Planets.gif’]}”/>
```

These entries render exactly the same markup. In addition to using the `name` and `library` attributes, stylesheet and script resources can be "relocated" to other parts of the view. For example, we could specify that a script resource be rendered within an HTML "head", "body" or "form" element in the page.

2.6.2.1 Relocatable Resources

Relocatable resources are resources that can be told where to render themselves, and this rendered location may be different than the resource tag placement in the view. For example, a portion of the view may be described in the page description language as follows:

```
<f:view contentType="text/html"/>
   <h:head>
      <meta http-equiv="Content-Type" content="text/html;
       charset=iso-8859-1" />
      <title>Example View</title>
   </h:head>
   <h:body>
      <h:form>
         <h:outputScript name=″ajax.js″ library=″javax.faces″
            target=″head″/>
      </h:form>
   </h:body>
</f:view>
```

The `<h:outputScript>` tag refers to the renderer, `ScriptRenderer,` that listens for `PostAddToViewEvent` event types:

```
@ListenerFor(facesEventClass=PostAddToViewEvent.class,
sourceClass=UIOutput.class)
public class ScriptRenderer extends Renderer implements
ComponentSystemEventListener {...
```

Refer to Section 3.4 "Event and Listener Model". When the component for this resource is added to the view, the `ScriptRenderer processEvent` method adds the component to a facet (named by the `target` attribute) under the view root. using the `UIViewRoot` component resource methods as described in Section 4.1.19.3 "Methods".

The <h:head> and <h:body> tags refer to the renderers HeadRenderer and BodyRenderer respectively, described in Section 8.6 "Standard HTML RenderKit Implementation". During the rendering phase, the encode methods for these renderers render all component resources under the facet child (named by `target`) under the `UIViewRoot` using the `UIViewRoot` component resource methods as described in Section 4.1.19.3 "Methods".

Existing component libraries (with existing head and body components), that want to use this resource loading feature must follow the rendering requirements described in Section 8.6 "Standard HTML RenderKit Implementation".

2.6.2.2 Resource Rendering Using Annotations

Components and renderers may be declared as requiring a resource using the `@ResourceDependency` annotation. The implementation must scan for the presence of this annotation on the component that was added to the `List` of child components. Check for the presence of the annotation on the renderer for this component (if there is a renderer for the component). The annotation check must be done immediately after the component is added to the `List.` Refer to Section 3.1.7 "Component Tree Manipulation"for detailed information.

3

User Interface Component Model

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

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

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

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

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

3.1 UIComponent and UIComponentBase

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

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

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

3.1.1 Component Identifiers

```
public String getId();

public void setId(String componentId);
```

[N/T-start may-component-identifier] Every component may be named by a *component identifier* that must conform to the following rules:

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

[P1-end] To minimize the size of responses generated by JavaServer Faces, it is recommended that component identifiers be as short as possible.

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

3.1.2 Component Type

While not a property of `UIComponent`, the `component-type` is an important piece of data related to each `UIComponent` subclass that allows the `Application` instance to create new instances of `UIComponent` subclasses with that type. Please see *Section 7.1.11 "Object Factories"* for more on `component-type`.

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

3.1.3 Component Family

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

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

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

3.1.4 ValueExpression properties

Properties and attributes of standard concrete component classes may be *value expression enabled*. This means that, rather than specifying a literal value as the parameter to a property or attribute setter, the caller instead associates a `ValueExpression` (see Section 5.8.3 "ValueBinding") whose `getValue()` method must be called (by the property getter) to return the actual property value to be returned if no value has been set via the corresponding property setter. If a property or attribute value has been set, that value must be returned by the property getter (shadowing any associated value binding expression for this property).

Value binding expressions are managed with the following method calls:

```
public ValueExpression getValueExpression(String name);

public void setValueExpression(String name, ValueExpression
expression);
```

where `name` is the name of the attribute or property for which to establish the value expression. [P1-start setValueExpression rules] The implementation of `setValueExpression` must detemine if the expression is a literal by calling `ValueExpression.isLiteralText()` on the `expression` argument. If the `expression` argument is literal text, then `ValueExpression.getValue()` must be called on the `expression` argument. The result must be used as the `value` argument, along with the `name` argument to this component's `getAttributes().put(name, value)` method call. [P1-end] [P1-start which properties are value expression enabled] For the standard component classes defined by this specification, all attributes, and all properties other than `id` and `parent`, are value expression enabled. [P1-end]

In previous versions of this specification, this concept was called "value binding". Methods and classes referring to this concept are deprecated, but remain implemented to preserve backwards compatibility.

```
public ValueBinding getValueBinding(String name);

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

Please consult the javadoc for these methods to learn how they are implemented in terms of the new "value expression" concept.

3.1.5 Component Bindings

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

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

- [P3-start how a component binding is used from a JSP page] When a component instance is first created (typically by virtue of being referenced by a `UIComponentELTag` in a JSP page), the JSF implementation will retrieve the `ValueExpression` for the name `binding`, and call `getValue()` on it. If this call returns a non-null `UIComponent` value (because the JavaBean programmatically instantiated and configured a component already), that instance will be added to the component tree that is being created. If the call returns `null`, a new component instance will be created, added to the component tree, and `setValue()` will be called on the `ValueExpression` (which will cause the property on the JavaBean to be set to the newly created component instance). [P3-end]
- [P1-start how a component binding is used when restoring the tree]When a component tree is recreated during the *Restore View* phase of the request processing lifecycle, for each component that has a `ValueExpression` associated with the name "binding", `setValue()` will be called on it, passing the recreated component instance. [P1-end]

Component bindings are often used in conjunction with JavaBeans that are dynamically instantiated via the Managed Bean Creation facility (see *Section 5.8.1 "VariableResolver and the Default VariableResolver"*). It is strongly recommend that application developers place managed beans that are pointed at by component binding expressions in "request" scope. This is because placing it in session or application scope would require thread-safety, since `UIComponent` instances depends on running inside of a single thread. There are also potentially negative impacts on memory management when placing a component binding in "session" scope.

3.1.6 Client Identifiers

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

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

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

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

3.1.7 Component Tree Manipulation

```
public UIComponent getParent();

public void setParent(UIComponent parent);
```

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

```
public List<UIComponent> getChildren();
```

Return a mutable `List` that contains all of the child `UIComponent`s for this component instance. [P1-start requirements of UIComponent.getChildren()] The returned `List` implementation must support all of the required and optional methods of the `List` interface, as well as update the parent property of children that are added and removed, as described in the Javadocs for this method. [P1-end] Note that the `add()` methods have a special requirement to cause the `PostAddToViewEvent` method to be fired, as well as the processing of the `ResourceDependency` annotation. See the javadocs for `getChildren()` for details.

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

A convenience method to return the number of child components for this component. [P2-start UIComponent.getChildCount requirements.] If there are no children, this method must return 0. The method must not cause the creation of a child component list, so it is preferred over calling `getChildren().size()` when there are no children. [P2-end]

3.1.8 Component Tree Navigation

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

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

```
public Iterator<UIComponent> getFacetsAndChildren();
```

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

```
public boolean invokeOnComponent(FacesContext context, String
clientId, ContextCallback callback) throws FacesException;
```

Starting at `this` component in the view, search for the UIComponent whose `getClientId()` method returns a String that exactly matches the argument `clientId` using the algorithm specified in the Javadocs for this method. If such a UIComponent is found, call the `invokeContextCallback()` method on the argument `callback` passing the current `FacesContext` and the found UIComponent. Upon normal return from the callback, return `true` to the caller. If the callback throws an exception, it must be wrapped inside of a `FacesException` and re-thrown. If no such UIComponent is found, return `false` to the caller.

Special consideration should be given to the implementation of `invokeOnComponent()` for UIComponent classes that handle iteration, such as `UIData`. Iterating components manipulate their own internal state to handle iteration, and doing so alters the clientIds of components nested within the iterating component. Implementations of `invokeOnComponent()` must guarantee that any state present in the component or children is restored before returning. Please see the Javadocs for `UIData.invokeOnComponent()` for details.

The `ContextCallback` interface is specified as follows..

```
public interface ContextCallback {
    public void invokeContextCallback(FacesContext context,
                UIComponent target);

}
```

Please consult the Javadocs for more details on this interface.

```
public static UIComponent getCurrentComponent(FacesContext
context);
```

Returns the `UIComponent` instance that is currently being processed.

```
public static UIComponent
getCurrentCompositeComponent(FacesContext context);
```

Returns the closest ancestor component relative to `getCurrentComponent` that is a composite component, or `null` if no such component is exists.

```
public boolean visitTree(VisitContext context,
                         VisitCallback callback);
```

Uses the visit API introduced in version 2 of the specification to perform a flexible and customizable visit of the tree from this instance and its children. Please see the package description for the package `javax.faces.component.visit` for the normative specification.

3.1.9 Facet Management

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

To meet this requirement, JavaServer Faces components offer support for *facets*, which represent a named collection of subordinate (but non-child) components that are related to the current component by virtue of a unique *facet name* that represents the role that particular component plays. Although facets are not part of the parent-child tree, they participate in request processing lifecycle methods, as described in Section 3.1.14 "Lifecycle Management Methods".

```
public Map<String, UIComponent> getFacets();
```

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

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

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

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

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

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

3.1.10 Managing Component Behavior

`UIComponentBase` provides default implementations for the methods from the `javax.faces.component.behavior.BehaviorHolder` interface. `UIComponentBase` does not implement the `javax.faces.component.behavior.BehaviorHolder` interface, but it provides the default implementations to simplify subclass implemenations. Refer to Section 3.7 "Component Behavior Model" for more information.

```
public void addBehavior(String eventName, Behavior behavior)
```

This method attaches a `Behavior` to the component for the specified `eventName. The eventName` must be one of the values in the `Collection` returned from `getEventNames(). F`or example, it may be desired to have some behavior defined when a "click" event occurs. The behavior could be some client side behavior in the form of a script executing, or a server side listener executing.

```
public Collection<String> getEventNames()
```

Returns the logical event names that can be associated with behavior for the component.

```
public Map<String, List<Behavior>> getBehaviors()
```

Returns a `Map` defining the association of events and behaviors. They keys in the `Map` are event names.

```
public String getDefaultEventName()
```

Returns the default event name (if any) for the component.

3.1.11 Generic Attributes

```
public Map<String, Object> getAttributes();
```

The render-independent characteristics of components are generally represented as JavaBean component properties with getter and setter methods (see Section 3.1.12 "Render-Independent Properties"). In addition, components may also be associated with generic attributes that are defined outside the component implementation class. Typical uses of generic attributes include:

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

The attributes for a component may be of any Java programming language object type, and are keyed by attribute name (a String). However, see Section 7.7.2 "State Saving Alternatives and Implications" for implications of your application's choice of state saving method on the classes used to implement attribute values.

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

[P1-start attribute property transparency rules] The `Map` returned by `getAttributes()` must also support attribute-property transparency, which operates as follows:

- When the `get()` method is called, if the specified attribute name matches the name of a readable JavaBeans property on the component implementation class, the value returned will be acquired by calling the appropriate property getter method, and wrapping Java primitive values (such as int) in their corresponding wrapper classes (such as `java.lang.Integer`) if necessary. If the specified attribute name does not match the name of a readable JavaBeans property on the component implementation class, consult the internal data-structure to in which generic attributes are stored. If no entry exists in the internal data-structure, see if there is a `ValueExpression` for this attribute name by calling `getValueExpression()`, passing the attribute name as the key. If a `ValueExpression` exists, call `getValue()` on it, returning the result. If an `ELException` is thrown wrap it in a `FacesException` and re-throw it. If the current component instance contains an attribute with the key "javax.faces.application.Resource.ComponentResource" it is a composite component and the `get()` method must take additional responsibilities:
  - If the value returned from `get()` is a `ValueExpression`, call `ValueExpression.getValue(ELContext)` and return the result. Otherwise, return the actual value.
- When the `put()` method is called, if the specified attribute name matches the name of a writable JavaBeans property on the component implementation class, the appropriate property setter method will be called. If the specified attribute name does not match the name of a writable JavaBeans property, simply put the value in the data-structure for generic attributes.
- When the `remove()` method is called, if the specified attribute name matches the name of a JavaBeans property on the component, an `IllegalArgumentException` must be thrown.
- When the `containsKey()` method is called, if the specified attribute name matches the name of a JavaBeans property, return `false`. Otherwise, return `true` if and only if the specified attribute name exists in the internal data-structure for the generic attributes.

The `Map` returned by `getAttributes()` must also conform to the entire contract for the `Map` interface. [P1-end]

3.1.11.1 Special Attributes

UIComponent Constants

```
public static final String CURRENT_COMPONENT =
"javax.faces.component.CURRENT_COMPONENT";
```

This is used as a key in the `FacesContext` attributes Map to indicate the component that is currently being processed.

```
public static final String CURRENT_COMPOSITE_COMPONENT =
"javax.faces.component.CURRENT_COMPOSITE_COMPONENT";
```

This is used as a key in the `FacesContext` attributes Map to indicate the composite component that is currently being processed.

```
public static final String BEANINFO_KEY =
"javax.faces.component.BEANINFO_KEY";
```

This is a key in the component attributes Map whose value is a `java.beans.BeanInfo` describing the composite component.

```
public static final String FACETS_KEY =
"javax.faces.component.FACETS_KEY";
```

This is a key in the composite component BeanDescriptor whose value is a Map<PropertyDescriptor> that contains meta-information for the declared facets for the composite component.

```
public static final String COMPOSITE_COMPONENT_TYPE_KEY =
"javax.faces.component.COMPOSITE_COMPONENT_TYPE";
```

This is a key in the composite component BeanDescriptor whose value is a `ValueExpression` that evaluates to the `component-type` of the composite component root.

```
public static final String COMPOSITE_FACET_NAME =
"javax.faces.component.COMPOSITE_FACET_NAME";
```

This is a key in the Map<PropertyDescriptor> that is returned by using the key `FACETS_KEY.` The value of this constant is also used as the key in the `Map` returned from `getFacets().` In this case, the value of this key is the `facet` (the `UIPanel)` that is the parent of all the components in the `composite implementation` section of the composite component VDL file.

Refer to the `javax.faces.component.UIComponent` Javadocs for more detailed information.

3.1.12 Render-Independent Properties

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

Name	Access	Type	Description
`id`	RW	String	The component identifier, as described in Section 3.1.1 “Component Identifiers”.
`parent`	RW	`UIComponent`	The parent component for which this component is a child or a facet.
`rendered`	RW	`boolean`	A flag that, if set to `true`, indicates that this component should be processed during all phases of the request processing lifecycle. The default value is “true”.
`rendererType`	RW	`String`	Identifier of the `Renderer` instance (from the set of `Renderer` instances supported by the `RenderKit` associated with the component tree we are processing. If this property is set, several operations during the request processing lifecycle (such as `decode` and the `encodeXxx` family of methods) will be delegated to a `Renderer` instance of this type. If this property is not set, the component must implement these methods directly.
`rendersChildren`	RO	`boolean`	A flag that, if set to `true`, indicates that this component manages the rendering of all of its children components (so the JSF implementation should not attempt to render them). The default implementation in `UIComponentBase` delegates this setting to the associated `Renderer`, if any, and returns `false` otherwise.
`transient`	RW	boolean	A flag that, if set to `true`, indicates that this component must not be included in the state of the component tree. The default implementation in `UIComponentBase` returns `false` for this property.

The method names for the render-independent property getters and setters must conform to the design patterns in the JavaBeans specification. See Section 7.7.2 “State Saving Alternatives and Implications” for implications of your application’s choice of state saving method on the classes used to implement property values.

3.1.13 Component Specialization Methods

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

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

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

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

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

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

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

public void encodeChildren(FacesContext context) throws
IOException;

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

These methods are called during the *Render Response* phase of the request processing lifecycle. `encodeAll()` will cause this component and all its children and facets that return `true` from `isRendered()` to be rendered, regardless of the value of the `getRendersChildren()` return value. `encodeBegin()`, `encodeChildren()`, and `encodeEnd()`have the responsibility of creating the response data for the beginning of this component, this component’s children (only called if the `rendersChildren` property of this component is `true`), and the ending of this component, respectively. Typically, this will involve generating markup for the output technology being supported, such as creating an HTML `<input>` element for a `UIInput` component. For clients that support it, the encode methods might also generate client-side scripting code (such as JavaScript), and/or stylesheets (such as CSS). The default implementations in `UIComponentBase` `encodeBegin()` and `encodeEnd()` delegate to a corresponding `Renderer`, if the `rendererType` property is `true`, and do nothing otherwise. [P1-start-comp-special]The default implementation in UIComponentBase `encodeChildren()` must iterate over its children and call `encodeAll()` for each child component. `encodeBegin()` must publish a `PreRenderComponentEvent`.[P1-end]

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

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

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

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

3.1.14 Lifecycle Management Methods

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

In order to support the "component" implicit object (See Section 5.6.2.1 "Implicit Object ELResolver for Facelets and Programmatic Access"), the following methods have been added to `UIComponent`

```
protected void pushComponentToEL(FacesContext context);
protected void popComponentFromEL(FacesContext context)
```

`pushComponentToEL()` and `popComponentFromEL()` must be called inside each of the lifecycle management methods in this section as specified in the javadoc for that method.

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

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

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

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

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

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

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

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

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

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

3.1.15 Utility Methods

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

Return the FacesContext instance for the current request.

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

Return the `Renderer` that is associated this `UIComponent`, if any, based on the values of the `family` and `rendererType` properties currently stored as instance data on the `UIComponent`.

```
protected void addFacesListener(FacesListener listener);

protected void removeFacesListener(FacesListener listener);
```

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

```
public Map<String, String> getResourceBundleMap();
```

Return a `Map of the ResourceBundle for this component. Please consult the Javadocs for more information.`

3.2 Component Behavioral Interfaces

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

3.2.1 ActionSource

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

3.2.1.1 Properties

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

Name	Access	Type	Description
`action`	RW	`MethodBinding`	**DEPRECATED** A `MethodBinding` (see Section 5.8.4 "MethodBinding") that must (if non-`null`) point at an action method (see Section 7.3 "Application Actions"). The specified method will be called during the *Apply Request Values* or *Invoke Application* phase of the request processing lifecycle, as described in Section 2.2.5 "Invoke Application". This method is replaced by the `actionExpression` property on `ActionSource2.` See the javadocs for the backwards compatibility implementation strategy.
actionListener	RW	MethodBinding	**DEPRECATED** A `MethodBinding` (see Section 5.8.4 "MethodBinding") that (if non-`null`) must point at a method accepting an `ActionEvent`, with a return type of `void`. Any `ActionEvent` that is sent by this `ActionSource` will be passed to this method along with the `processAction()` method of any registered `ActionListener`s, in either Apply Request Values or Invoke Application phase, depending upon the state of the `immediate` property. See the javadocs for the backwards compatibility implementation strategy.
immediate	RW	boolean	A flag indicating that the default `ActionListener` should execute immediately (that is, during the *Apply Request Values* phase of the request processing lifecycle, instead of waiting for *Invoke Application* phase). The default value of this property must be `false`.

3.2.1.2 Methods

`ActionSource` adds no new processing methods.

3.2.1.3 Events

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

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

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

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

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

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

```
public void addActionListener(ActionListener listener);

public void removeActionListener(ActionListener listener);
```

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

3.2.2 ActionSource2

The `ActionSource2` interface extends `ActionSource` and provides a JavaBeans property analogous to the `action` property on `ActionSource`. This allows the `ActionSource` concept to leverage the new Unified EL API.

3.2.2.1 Properties

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

Name	Access	Type	Description
`actionExpression`	RW	`javax.el.MethodExpression`	A `MethodExpression` (see Section 5.8.4 “MethodBinding”) that must (if non-`null`) point at an action method (see Section 7.3 “Application Actions”). The specified method will be called during the *Apply Request Values* or *Invoke Application* phase of the request processing lifecycle, as described in Section 2.2.5 “Invoke Application”.

3.2.2.2 Methods

`ActionSource2` adds no new processing methods.

3.2.2.3 Events

`ActionSource2` adds no new events.

3.2.3 NamingContainer

`NamingContainer` is a marker interface. Components that implement `NamingContainer` have the property that, for all of their children that have non-`null` component identifiers, all of those identifiers are unique. This property is enforced by the `renderView()` method on `ViewHandler`. In JSP based applications, it is also enforced by the `UIComponentELTag`. Since this is just a marker interface, there are no properties, methods, or events. Among the standard components, `UIForm` and `UIData` implement `NamingContainer`. See *Section 4.1.4 "UIForm"* and *Section 4.1.3.3 "UIData"* for details of how the *NamingContainer* concept is used in these two cases.

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

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

3.2.4 StateHolder

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

3.2.4.1 Properties

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

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

3.2.4.2 Methods

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

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

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

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

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

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

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

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

3.2.4.3 Events

`StateHolder` does not originate any standard events.

3.2.5 PartialStateHolder

3.2.6 ValueHolder

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

3.2.6.1 Properties

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

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

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

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

The `Converter` property must be inspected for the presence of `ResourceDependency` and `ResourceDependencies` annotations as described in the Javadocs for the `setConverter` method.

3.2.6.2 Methods

ValueHolder adds no methods.

3.2.6.3 Events

`ValueHolder` does not originate any standard events.

3.2.7 EditableValueHolder

The `EditableValueHolder` interface (extends `ValueHolder`, see *Section 3.2.6 "ValueHolder"*) describes additional features supported by editable components, including `ValueChangeEvents` and `Validators`.

3.2.7.1 Properties

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

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

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

3.2.7.2 Methods

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

```
public void addValidator(Validator validator);

public void removeValidator(Validator validator);
```

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

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

The `addValidator`'s `Validator` argument must be inspected for the presense of the `ResourceDependency` and `ResourceDependencies` annotations as described in the Javadocs for the `addValidator` method.

3.2.7.3 Events

`EditableValueHolder` is a source of `ValueChangeEvent`, `PreValidateEvent` and `PostValidate` events. These are emitted during calls to `validate()`, which happens during the *Process Validations* phase of the request processing lifecycle. The `PreValidateEvent` is published immediately before the component gets validated. `PostValidate` is published after validation has occurred, regardless if the validation was successful or not. If the validation for the component did pass successfully, and the previous value of this component differs from the current value, the `ValueChangeEvent` is published. The following methods allow listeners to register and deregister for `ValueChangeEvent`s. See Section 3.4 "Event and Listener Model" for more details on the event and listener model provided by JSF.

```
public void addValueChangeListener(ValueChangeListener listener);

public void removeValueChangeListener(ValueChangeListener
listener);
```

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

3.2.8 SystemEventListenerHolder

Classes that implement this interface agree to maintain a list of `SystemEventListener` instances for each kind of `SystemEvent` they can generate. This interface enables arbitrary Objects to act as the source for `SystemEvent` instances.

3.2.8.1 Properties

This interface contains no JavaBeans properties

3.2.8.2 Methods

The following method gives the JSF runtime access to the list of listeners stored by this instance.:

```
public List<FacesLifecycleListener>
getListenersForEventClass(Class<? extends SystemEvent>
facesEventClass);
```

During the processing for `Application.publishEvent()`, if the `source` argument to that method implements `SystemEventListenerHolder`, the `getListenersForEventClass()` method is invoked on it, and each listener in the list is given an opportunity to process the event, as specified in the javadocs for `Application.publishEvent()`.

3.2.8.3 Events

While the class that implements `SystemEventListenerHolder` is indeed a source of events, it is a call to `Application.publishEvent()` that causes the event to actually be emitted. In the interest of maximum flexibility, this interface does not define how listeners are added, removed, or stored. See Section 3.4 “Event and Listener Model” for more details on the event and listener model provided by JSF.

3.2.9 ClientBehaviorHolder

[P1-start-addBehavior] Components must implement the `ClientBehaviorHolder` interface to add the ability for attaching `ClientBehavior` instances (see Section 3.7 “Component Behavior Model”). Components that extend UIComponentBase only need to implement the getEventNames() method and specify "implements ClientBehaviorHolder". UIComponentBase provides base implementations for all other methods. [P1-end] The concrete HTML component classes that come with JSF implement the `ClientBehaviorHolder` interface.

```
public void addClientBehavior(String eventName, ClientBehavior
behavior);
```

Attach a `ClientBehavior` to a component implementing this `ClientBehaviorHolder` interface for the specified event. A default implementation of this method is provided in `UIComponentBase` to make it easier for subclass implementations to add behaviors.

```
public Collection<String> getEventNames();
```

Return a `Collection` of logical event names that are supported by the component implementing this `ClientBehaviorHolder` interface. [P1-start-getEventNames]The `Collection` must be non null and unmodifiable.[P1-end]

```
public Map<String, List<ClientBehavior>> getClientBehaviors();
```

Return a `Map` containing the event-client behavior association. Each event in the Map may contain one or more ClientBehavior instances that were added via the `addClientBehavior()` method.

[P1-start-getBehaviors]Each key value in this `Map` must be one of the event names in the `Collection` returned from `getEventNames()`.[P1-end]

```
public String getDefaultEventName();
```

Return the default event name for this component behavior if the component defines a default event.

3.3 Conversion Model

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

3.3.1 Overview

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

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

To transform data formats between these views, JavaServer Faces provides an ability to plug-in an optional `Converter` for each `ValueHolder`, which has the responsibility of converting the internal data representation between the two views. The application developer attaches a particular `Converter` to a particular `ValueHolder` by calling `setConverter`, passing an instance of the particular converter. A `Converter` implementation may be acquired from the `Application` instance (see Section 7.1.11 “Object Factories”) for your application.

3.3.2 Converter

JSF provides the `javax.faces.convert.Converter` interface to define the behavioral characteristics of a `Converter`. Instances of implementations of this interface are either identified by a *converter identifier*, or by a class for which the `Converter` class asserts that it can perform successful conversions, which can be registered with, and later retrieved from, an `Application`, as described in Section 7.1.11 “Object Factories”.

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

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

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

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

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

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

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

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

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

[P1-start-converter-resource]If the class implementing `Converter` has a `ResourceDependency` annotation or a `ResourceDependencies` annotation, the action described in the Javadocs for the `Converter` interface must be followed when `ValueHolder.setConverter` is called.[P1-end]

3.3.3 Standard Converter Implementations

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

[P1-start standard converters] The following converter id values must be registered to create instances of the specified Converter implementation classes:

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

- `javax.faces.Short` -- An instance of `javax.faces.convert.ShortConverter` (or a subclass of this class).

[P1-end] See the Javadocs for these classes for a detailed description of the conversion operations they perform, and the configuration properties that they support.

[P1-start by-Class converters] A JSF implementation must register converters for all of the following classes using the by-type registration mechanism:

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

[P1-end] See the Javadocs for these classes for a detailed description of the conversion operations they perform, and the configuration properties that they support.

[P1-start allowing string converters] A compliant implementation must allow the registration of a converter for class `java.lang.String` and `java.lang.String.TYPE` that will be used to convert values for these types. [P1-end]

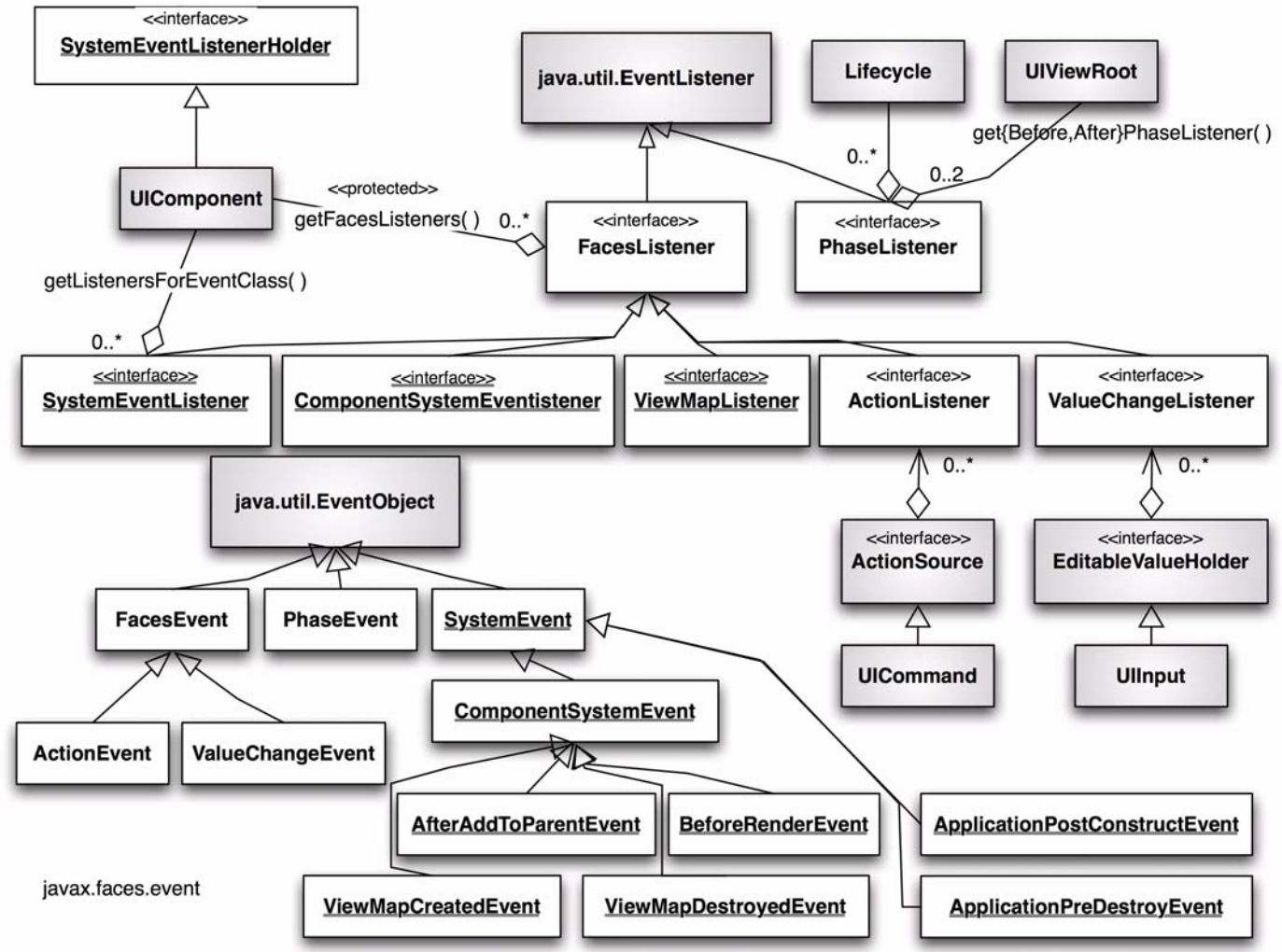
3.4 Event and Listener Model

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

3.4.1 Overview

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

A `UIComponent` subclass may choose to emit *events* that signify significant state changes, and broadcast them to *listeners* that have registered an interest in receiving events of the type indicated by the event's implementation class. At the end of several phases of the request processing lifecycle, the JSF implementation will broadcast all of the events that have been queued to interested listeners. As of JSF version 2, the specification also defines *system events*. System events are events that are not specific to any particular application, but rather stem from specific points in time of running a JSF application. The following UML class diagram illustrates the key players in the event model. Boxes shaded in gray indicate classes or interfaces defined outside of the `javax.faces.event` package.


<<interface>>
SystemEventListenerHolder
java.util.EventListener
Lifecycle
UIViewRoot
get{Before,After}PhaseListener( )
0..*
0..2
UIComponent
<<protected>>
getFacesListeners( )
0..*
<<interface>>
FacesListener
<<interface>>
PhaseListener
getListenersForEventClass( )
0..*
<<interface>>
SystemEventListener
<<interface>>
ComponentSystemEventistener
<<interface>>
ViewMapListener
<<interface>>
ActionListener
<<interface>>
ValueChangeListener
0..*
0..*
java.util.EventObject
<<interface>>
ActionSource
<<interface>>
EditableValueHolder
FacesEvent
PhaseEvent
SystemEvent
UICommand
UIInput
ComponentSystemEvent
ActionEvent
ValueChangeEvent
AfterAddToParentEvent
BeforeRenderEvent
ApplicationPostConstructEvent
javax.faces.event
ViewMapCreatedEvent
ViewMapDestroyedEvent
ApplicationPreDestroyEvent

3.4.2 Application Events

Application events are events that are specific to a particular application. Application events are the standard events that have been in JSF from the beginning.

3.4.2.1 Event Classes

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

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

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

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

`FacesEvent` has a `phaseId` property (of type `PhaseId`, see Section 3.4.2.3 “Phase Identifiers”) used to identify the request processing lifecycle phase after which the event will be delivered to interested listeners.

```
public PhaseId getPhaseId();

public void setPhaseId(PhaseId phaseId);
```

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

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

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

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

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

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

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

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

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

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

3.4.2.2 Listener Classes

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

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

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

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

3.4.2.3 Phase Identifiers

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

3.4.2.4 Listener Registration

A concrete `UIComponent` subclass that emits events of a particular type must include public methods to register and deregister a listener implementation. [P1-start listener methods must conform to javabeans naming] In order to be recognized by development tools, these listener methods must follow the naming patterns defined in the *JavaBeans Specification*. [P1-end] For example, for a component that emits `FooEvent` events, to be received by listeners that implement the `FooListener` interface, the method signatures (on the component class) must be:

```
public void addFooListener(FooListener listener);

public FooListener[] getFooListeners();

public void removeFooListener(FooListener listener);
```

The application (or other components) may register listener instances at any time, by calling the appropriate add method. The set of listeners associated with a component is part of the state information that JSF saves and restores. Therefore, listener implementation classes must have a public zero-argument constructor, and may implement `StateHolder` (see Section 3.2.4 “StateHolder”) if they have internal state information that needs to be saved and restored.

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

3.4.2.5 Event Queueing

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

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

3.4.2.6 Event Broadcasting

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

During event broadcasting, a listener processing an event may:

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

3.4.3 System Events

System Events are introduced in version 2 of the specification and represent specific points in time for a JSF application. `PhaseEvent`s also represent specific points in time in a JSF application, but the granularity they offer is not as precise as System Events. For more on `PhaseEvent`s, please see Section 12.2 “PhaseEvent”.

3.4.3.1 Event Classes

All system events extend from the base class `SystemEvent`. `SystemEvent` has a similar API to `FacesEvent`, but the `source` of the event is of type `Object` (instead of `UIComponent`), `SystemEvent` has no `PhaseId` property and `SystemEvent` has no `queue()` method because `SystemEvent`s are never queued. `SystemEvent` shares `isAppropriateListener()` and `processListener()` with `FacesEvent`. For the specification of these methods see `3.4.2.1`.

System events that originate from or are associated with specific component instances should extend from `ComponentSystemEvent`, which extends `SystemEvent` and adds a `getComponent()` method, as specifiifed in `3.4.2.1`.

The specification defines the following `SystemEvent` subclasses, all in package `javax.faces.event`.

- `ExceptionQueuedEvent` indicates a non-expected `Exception` has been thrown. Please see Section 6.2 “ExceptionHandler” for the normative specification.
- `PostConstructApplicationEvent` must be published immediately after application startup. Please see Section 11.4.2 “Application Startup Behavior” for the normative specification.
- `PreDestroyApplicationEvent` must be published as immediately before application shutdown. Please see Section 11.4.3 “Application Shutdown Behavior” for the normative specification

The specification defines the following `ComponentSystemEvent` classes, all in package `javax.faces.event`.

- `InitialStateEvent` must be published with a direct call to `UIComponent.processEvent()`, during the `apply()` method of the class `javax.faces.webapp.vdl.ComponentHandler`. Please see the javadocs for the normative specification.
- `PostAddToViewEvent` indicates that the `source` component has just been added to the view. Please see Section 3.1.7 “Component Tree Manipulation” for a reference to the normative specification.
- `PostConstructViewMapEvent` indicates that the `Map` that is the view scope has just been created. Please see, the UIViewRoot Section 4.1.19.4 “Events” for a reference to the normative specification.
- `PostRestoreStateEvent` indicates that an individual component instance has just had its state restored. Please see the `UIViewRoot` Section 4.1.19.4 “Events” for a reference to the normative specification.
- `PostValidateEvent` indicates that an individual component instance has just been validated. Please see the `EditableValueHolder` Section 3.2.7.3 “Events” for the normative specification.
- `PreDestroyViewMapEvent` indicates that the `Map` that is the view scope is about to be destroyed. Please see, the UIViewRoot Section 4.1.19.2 “Properties” for the normative specification.
- `PreRenderComponentEvent` indicates that the `source` component is about to be rendered. Please see Section 3.1.7 “Component Tree Manipulation” for a reference to the normative specification.
- `PreRenderViewEvent` indicates that the `UIViewRoot` source component is about to be rendered. Please see Section 2.2.6 “Render Response” for the normative specification.
- `PreValidateEvent` indicates that an individual component instance is about to be validated. Please see the `EditableValueHolder` Section 3.2.7.3 “Events” for the normative specification.

3.4.3.2 Listener Classes

Unlike application events, the creation of new event types for system events does not require the creation of new listener interfaces. All `SystemEvent` types can be listened for by listeners that implement `javax.faces.event.SystemEventListener`. Please see the javadocs for that class for the complete specification.

As a developer convenience, the listener interface `ComponentSystemEventListener` has been defined for those cases when a `SystemEventListener` is being attached to a specific `UIComponent` instance. `ComponentSystemEventListener` lacks the `isListenerForSource()` method because it is implcictly defined by virture of the listener being added to a specific component instance.

3.4.3.3 Programmatic Listener Registration

System events may be listened for at the Application level, using `Application.subscribeToEvent()` or at the component level, by calling `subscribeToEvent()` on a specific component instance. The specification for `Application.subscribeToEvent()` may be found in `Section 7.1.13 “System Event Methods”`.

The following methods are defined on `UIComponent` to support per-component system events.

```
public void subscribeToEvent(Class<? extends SystemEvent>
eventClass, ComponentSystemEventListener componentListener);
public void unsubscribeFromEvent(Class<? extends SystemEvent>
eventClass, ComponentSystemEventListener componentListener);
```

See the javadoc for `UIComponent` for the normative specification of these methods.

In addition to the above methods, the `@ListenerFor` and `@ListenersFor` annotations allow components, renderers, validators and converters to declare that they want to register for system events. Please see the javadocs for those annotations for the complete specification.

3.4.3.4 Declarative Listener Registration

Page authors can subscribe to events using the <f:event/> tag. This tag will allow the application developer to specify the method to be called when the specifed event fires for the component of which the tag is a child. The tag usage is as follows:

```
<h:inputText value="#{myBean.text}">
  <f:event type="beforeRender"
           listener="#{myBean.beforeTextRender}" />
</h:inputText>
```

The *type* attribute specifies the type of event, and can be any of the specification-defined events or one of any user-defined events, but must be a `ComponentSystemEvent`, using either the short-hand name for the event or the fully-qualified class name (e.g., `com.foo.app.event.CustomEvent`). If the event can not be found, a `FacesException` listing the offending event type will be thrown. Please see the tlddocs for the `<f:event />` tag for the normative specification of the declarative event feature.

The method signature for the `MethodExpression` pointed to by the *listener* attribute must match the signature of `javax.faces.event.ComponentSystemEventListener.processEvent()`.

3.4.3.5 Listener Registration By Annotation

The `ListenerFor` and `ListenersFor` annotations can be applied to components and rendererers. Classes tagged with the `ListenerFor` annotation are installed as listeners. The `ListenersFor` annotation is a container annotation tp specify multiple `ListenerFor` annotations for a single class. Please refer to the Javadocs for the `ListenerFor` and `ListenersFor classes for more details.`

3.4.3.6 Listener Registration By Application Configuration Resources

A `<system-event-listener>` element, within the `<application>` element of an application configuration resource, declares an application scoped listener and causes a call to `Application.subscribeToEvent()`.

3.4.3.7 Event Broadcasting

System events are broadcast immediately by calls to `Application.publishEvent()` Please see Section 7.1.13 “System Event Methods” for the normative specification of `publishEvent()`.

3.5 Validation Model

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

3.5.1 Overview

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

3.5.2 Validator Classes

A validator must implement the `javax.faces.validator.Validator` interface, which contains a `validate()` method signature.

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

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

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

3.5.3 Validation Registration

The `EditableValueHolder` interface (implemented by `UIInput`) includes an `addValidator` method to register an additional validator for this component, and a `removeValidator` method to remove an existing registration. In JSF 1.1 there was the ability to set a `MethodBinding` that points to a method that adheres to the `validate` signature in the `Validator` interface, which will be called after the Validator instances added by calling addValidator() have been invoked. In JSF 1.2, this has been replaced by providing a new wrapper class that implements `Validator`, and accepts a `MethodExpression` instance that points to the same method that the `MethodBinding` pointed to in JSF 1.1. Please see the javadocs for `EditableValueHolder.setValidator()`.

The application (or other components) may register validator instances at any time, by calling the `addValidator` method. The set of validators associated with a component is part of the state information that JSF saves and restores. Validators that wish to have configuration properties saved and restored must also implement `StateHolder` (see Section 3.2.4 "StateHolder").

In addition to validators which are registered explicitly on the component, either through the Java API or in the view markup, zero or more "default validators" can be declared in the application configuration resources, which will be registered on all `UIInput` instances in the component tree unless explicitly disabled. [P1-start-validator-reg]The default validators are appended after any locally defined validators once the `EditableValueHolder` is populated and added to the component tree. See the javadocs for `UIInput.encodeEnd()` for the normative specification. A default validator must not be added to a `UIInput` if a validator having the same id is already present.

The typical way of registering a default validator id is by declaring it in a configuration resource, as follows:

```
<faces-config>
  <application>
    <default-validators>
    <validator-id>javax.faces.Bean</validator-id>
    </default-validators>
  <application/>
</faces-config>
```

A default validator may also be registered using the `isDefault` attribute on the `@FacesValidator` annotation on a `Validator` class, as specified in Section 11.5.1 “Requirements for scanning of classes for annotations”.

The during application startup, the runtime must cause any default validators declared either in the application configuration resources, or via a `@FacesValidator` annotation with `isDefault` set to `true` to be added with a call to `Application.addDefaultValidatorId()`. This method is declared in Section 7.1.11.1 “Default Validator Ids”.

Any configuration resource that declares a list of default validators overrides any list provided in a previously processed configuration resource. If an empty `<default-validators/>` element is found in a configuration resource, the list of default validators must be cleared.

In environments that include Bean Validation, the following additional actions must be taken at startup time. If the `javax.faces.validator.DISABLE_BEAN_VALIDATOR` `<context-param>` exists and its value is `true`, the following step must be skipped:

- The runtime must guarantee that the validator id `javax.faces.Bean` is included in the result from a call to `Application.getDefaultValidatorInfo()` (see Section 7.1.11.1 “Default Validator Ids”), regardless of any configuration found in the application configuration resources or via the `@FacesValidator` annotation.[P1-end]

3.5.4 Validation Processing

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

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

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

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

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

3.5.5 Standard Validator Implementations

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

- Standard `Validators` accept configuration information as either parameters to the constructor that creates a new instance of that `Validator`, or as JavaBeans component properties on the `Validator` implementation class.
- To support internationalization, `FacesMessage` instances should be created. The message identifiers for such standard messages are also defined by manifest String constants in the implementation classes. It is the user's responsibility to ensure the content of a `FacesMessage` instance is properly localized, and appropriate parameter substitution is performed, perhaps using `java.text.MessageFormat`.
- See the javadocs for `UIInput.validateValue()` for further normative specification regarding validation.
- Concrete Validator implementations must define a public static final String constant VALIDATOR_ID, whose value is the standard identifier under which the JSF implementation must register this instance (see below).

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

[P1-start standard validators] The following standard `Validator` implementations (in the `javax.faces.validator` package) are provided:

- `DoubleRangeValidator`—Checks the local value of a component, which must be of any numeric type, against specified maximum and/or minimum values. Standard identifier is "javax.faces.DoubleRange".
- `LengthValidator`—Checks the length (i.e. number of characters) of the local value of a component, which must be of type `String`, against maximum and/or minimum values. Standard identifier is "javax.faces.Length".
- `LongRangeValidator`—Checks the local value of a component, which must be of any numeric type convertible to `long`, against maximum and/or minimum values. Standard identifier is "javax.faces.LongRange".
- `RegexValidator`—Accepts a "pattern" attribute that is interpreted as a regular expression from the `java.util.regex` package. The local value of the component is checked fora match against this regular expression. Standard identifier is "javax.faces.RegularExpression"
- `BeanValidator`- The implementation must ensure that this validator is only available when running in an environment in which JSR-303 Beans Validation is available. Please see the javadocs for `BeanValidator.validate()` for the specification.Standard identifier is "javax.faces.Bean"
- RequiredValidator - Analogous to setting the required attribute to true on the EditableValueHolder. Enforces that the local value is not empty. Reuses the logic and error messages defined on UIInput. The standard identifier for this validator is "javax.faces.Required"

`MethodExpressionValidator`—Wraps a `MethodExpression` and interprets it as pointing to a method that performs validation. Any exception thrown when the expression is invoked is wrapped in a `ValidatorException` in similar fashion as the above validators. [P1-end]

3.5.6 Bean Validation Integration

If the implementation is running in a container environment that requires Bean Validation, it must expose the bean validation as described in this specification.

As stated in the specification goals of JSR 303, validation often gets spread out across the application, from user interface components to persistent objects. Bean Validation strives to avoid this duplication by defining a set of metadata that can be used to express validation constraints that are sharable by any layer of the application. Since its inception, JSF has supported a "field level validation" approach. Rather than requiring the developer to define validators for each input component (i.e., `EditableValueHolder`), the BeanValidator can be automatically applied to all fields on a page so that the work of enforcing the constraints can be delegated to the Bean Validation provider.

3.5.6.1 Bean Validator Activation

[P1-BeanValidationIntegration]If Bean Validation is present in the runtime environment, the system must ensure that the `javax.faces.Bean` standard validator is added with a call to `Application.addDefaultValidatorId()`.[P1-end] This has the effect that `UIInput.encodeEnd()` will cause Bean Validation to be called for every field in the application.

If Bean Validation is present, and the `javax.faces.VALIDATE_EMPTY_FIELDS` `<context-param>` is not explicitly set to `false`, JSF will validate `null` and empty fields so that the `@NotNull` and `@NotEmpty` constraints from Bean Validation can be leveraged. The next section describes how the reference to the Bean Validation ValidatorFactory is obtained by that validator.

3.5.6.2 Obtaining a ValidatorFactory

The Bean Validation ValidatorFactory is the main entry point into Bean Validation and is responsible for creating Validator instances. [P1-start-validatoryfactory]A ValidatorFactory is retrieved using the following algorithm:

- If the servlet context contains a ValidatorFactory instance under the attribute named javax.faces.validator.beanValidator.ValidatorFactory, this instance is used by JSF to acquire Validator instances (specifically in the BeanValidator). This key should be defined in the constant named VALIDATOR_FACTORY_KEY on BeanValidator.
- If the servlet context does not contain such an entry, JSF looks for a Bean Validation provider in the classpath. If present, the standard Bean Validation bootstrap strategy is used. If not present, Bean Validation integration is disabled. If the BeanValidator is used an no ValidatorFactory can be retrieved, a FacesException is raised. The standard Bean Validation bootstrap procedure is shown here:

```
ValidatorFactory validatorFactory =
    Validation.buildDefaultValidatorFactory();
```

Once instantiated, the result can be stored in the servlet context attribute mentioned as a means of caching the result. If JSF is running in an EE6 environment, Bean Validation will be available, as defined by the EE6 specification, and thus activated in JSF. The EE container will be responsible for making the ValidatorFactory available as an attribute in the ServletContext as mentioned above.[P1-end]

3.5.6.3 Localization of Bean Validation Messages

To ensure proper localization of the messages, JSF should provide a custom BeanValidation MessageInterpolator resolving the Locale according to JSF defaults and delegating to the default MessageInterpolator as defined in ValidationFactory.getMessageInterpolator(). A possible implementation is shown here:

```
public class JsfMessageInterpolator implements
   MessageInterpolator {

   private final MessageInterpolator delegate;

   public JsfMessageInterpolator(MessageInterpolator delegate) {
        this.delegate = delegate;
    }

   public String interpolate(String message, ConstraintDescriptor
     constraintDescriptor,Object value) {
        Locale locale =
        FacesContext.getCurrentInstance().getViewRoot().
          getLocale();
          return this.delegate.interpolate(
            message, constraintDescriptor, value, locale );
    }

   public String interpolate(String message, ConstraintDescriptor
     constraintDescriptor, Object value, Locale locale) {
        return this.delegate.interpolate(message,
         constraintDescriptor, value, locale);
   }
}
```

Once a ValidatorFactory is obtained, as described in Section 3.5.6.2 “Obtaining a ValidatorFactory”, JSF receives a Validator instance by providing the custom message interpolator to the validator state.

```
//could be cached
MessageInterpolator jsfMessageInterpolator = new
    JsfMessageInterpolator(
      validatorFactory.getMessageInterpolator() );

//...

Validator validator = validatorFactory
          .usingContext()
          .messageInterpolator(jsfMessageInterpolator)
          .getValidator();
```

The local value is then passed to the Validator.validateValue() method to check for constraint violations. Since Bean Validation defines a strategy for localized message reporting, the BeanValidator does not need to concern itself with producing the validation message. Instead, the BeanValidator should accept the interpolated message returned from Bean Validation API, which is accessed via the method getInterpolatedMessage() on the ContraintFailure class, and use it as

the replacement value for the first numbered placeholder for the key javax.faces.validator.BeanValidator.MESSAGE (i.e., {0}). To encourage use of the Bean Validation message facility, the default message format string for the BeanValidator message key must be a single placeholder, as shown here:

```
javax.faces.validator.BeanValidator.MESSAGE={0}
```

Putting the Bean Validation message resolution in full control of producing the displayed message is the recommended approach. However, to allow the developer to align the messages generated by the BeanValidator with existing JSF 1.2 validators, the developer may choose to override this message key in an application resource bundle and reference the component label, which replaces the second numbered placeholder (i.e., {1}).

```
javax.faces.validator.BeanValidator.MESSAGE={1}: {0}
```

This approach is useful if you are already using localized labels for your input components and are displaying the messages above the form, rather than adjacent to the input.

3.6 Composite User Interface Components

3.6.1 Non-normative Background

To aid implementors in providing a spec compliant runtime for composite components, this section provides a non-normative background to motivate the discussion of the composite component feature. The composite component feature enables developers to write real, reusable, JSF UI components without any Java code or configuration XML.

3.6.1.1 What does it mean to be a JSF User Interface component?

JSF is a component based framework, and JSF UI components are the main point of JSF. But what is a JSF UI component, really? Conceptually, a JSF UI Component is a software artifact that represents a reusable, self contained piece of a user interface. A very narrow definition for "JSF UI Component" is imposed at runtime. This definition can be summarized as

> A JSF UI Component is represented at runtime by an instance of a Java class that includes `javax.faces.component.UIComponent` as an ancestor in its inheritance hierarchy.

It is easy to write a class that adheres to this definition, but in practice, component authors need to do more than just this in order to get the most from JSF and to conform to user's expectations of what a JSF UI Component is. For example, users expect a JSF UI Component can do some or all of the following:

- be exposed to the page-author via a markup tag with sensible attributes
- emit events (such a `ValueChangeEvent` or `ActionEvent`)
- allow attaching listeners
- allow attaching a `Converter` and/or `Validator`(s)
- render itself to the user-agent, with full support for styles, localization and accessibility
- support delegated rendering to allow for client device independence
- read values sent from the user-agent and correctly adapt them to the faces lifecycle
- correctly handle saving and restoring its state across multiple requests from the user-agent

Another important dimension to consider regarding UI components is the context in which the developer interacts with the component. There are generally two such contexts.

- In the context of a markup page, such as a JSP or Facelet page. In this context the developer interacts with the UI component using a markup element, setting attributes on that element, and nesting child elements within that component markup element.
- In the context of code, such as a listener, a managed-bean, or other programming language context. In this context, the developer is writing JavaCode that is either passed the UI component as an argument, or obtains a reference to the UI component in some other way.

3.6.1.2 How does one make a custom JSF User Interface component (JSF 1.2 and earlier)?

To satisfy a user's expectations for a JSF UI component, the component author must adhere to one of the following best practices.

- extend the custom component class from an existing subclass of `UIComponent` that most closely represents the meaning and behavior of the piece of the UI you are encapsulating in the component.
- extend the custom component class directly from `UIComponentBase` and implement the appropriate "behavioral interface"(s) that most closely represents the meaning and behavior of the piece of the UI you are encapsulating in the component. See Section 3.2 "Component Behavioral Interfaces" for more.

Note that the first best practice includes the second one "for free" since the stock `UIComponent` subclasses already implement the appropriate behavioral interfaces.

When following either best practice, the JSF UI component developer must follow several steps to make the component available for use in markup pages or in code, including but not necessarily limited to

- Make entries in a `faces-config.xml` file, linking the component class to its `component-type`, which enables the `Application.createComponent()` method to create instances of the component.
- Make entries in a `faces-config.xml` file to declare a `Renderer` that provides client-device independence.
- Provide a JSP or Facelet tag handler that allows the page author to build UIs that include the component, and to customize each instance of the component with listeners, properties and model associations. This includes making the association between the `Renderer` and the `UIComponent`.
- Provide a `Renderer` that provides client device independency for the component
- Make entries in a `faces-config.xml` file that links the `Renderer` and its Java class.

These steps are complex, yet the components one creates by following them can be very flexible and powerful. By making some simplifying assumptions, it is possible to allow the creation of components that are just as powerful but require far less complexity to develop. This is the whole point of composite components: to enable developers to write real, reusable, JSF UI components without any Java code or configuration XML.

3.6.1.3 How does one make a composite component?

The composite component feature builds on two features introduced in JSF 2.0: resources (Section 2.6 "Resource Handling") and Facelets (Chapter 10 "Facelets and its use in Web Applications"). Briefly, a composite component is any Facelet markup file that resides inside of a resource library. For example, if a Facelet markup file named `loginPanel.xhtml` resides inside of a resource library called `ezcomp`, then page authors can use this component by declaring the xml namespace `xmlns:ez="http://java.sun.com/jsf/composite/ezcomp"` and including the tag `<ez:loginPanel />` in their pages. Naturally, it is possible for a composite component author to declare an alternate XML namespace for their composite components, but doing so is optional.

Any valid Facelet markup is valid for use inside of a composite component, including the templating features specified in Section 10.4.3 "Facelet Templating Tag Library". In addition, the tag library specified in Section 10.4.4 "Composite Component Tag Library" must be used to declare the metadata for the composite component. Future versions of the JSF specification may relax this requirement, but for now at least the `<composite:interface>` and `<composite:implementation>` sections are required when creating a composite component.

3.6.1.4 A simple composite component example

Create the page that uses the composite component, `index.xhtml`.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ez="http://java.sun.com/jsf/composite/ezcomp">
<h:head>
<title>A simple example of EZComp</title>
</h:head>

<h:body>

<h:form>

    <ez:loginPanel id="loginPanel">

      <f:actionListener for="loginEvent"
                        binding="#{bean.loginEventListener}" />

    </ez:loginPanel>

</h:form>

</h:body>

</html>
```

The only thing special about this page is the `ez` namespace declaration and the inclusion of the `<ez:loginPanel />` tag on the page. The occurrence of the string "`http://java.sun.com/jsf/composite/`" in a Facelet XML namespace declaration means that whatever follows that last "`/`" is taken to be the name of a resource library. For any usage of this namespace in the page, such as `<ez:loginPanel />`, a Facelet markup file with the corresponding name is loaded and taken to be the composite component, in this case the file `loginPanel.xhtml`. The implementation requirements for this and other Facelet features related to composite components are specified in Section 10.3.3 "Requirements specific to composite components".

Create the composite component markup page. In this case, `loginPanel.xhtml` resides in the `./resources/ezcomp` directory relative to the `index.xhtml` file.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:composite="http://java.sun.com/jsf/composite">
<head>

<title>Not present in rendered output</title>

</head>

<body>

<composite:interface>

  <composite:actionSource name="loginEvent" />

</composite:interface>

<composite:implementation>

   <p>Username: <h:inputText id="usernameInput" /></p>

   <p>Password: <h:inputSecret id="passwordInput" /></p>

   <p><h:commandButton id="loginEvent" value="login"/>

</composite:implementation>

</body>

</html>
```

The `<composite:interface>` section declares the public interface that users of this component need to understand. In this case, the component declares that it contains an implementation of `ActionSource2` (see Section 3.2.2 “ActionSource2”), and therefore anything one can do with an `ActionSource2` in a Facelet markup page you one do with the composite component. (See Section 3.2 “Component Behavioral Interfaces” for more on `ActionSource2` and other behavioral interfaces). The `<composite:implementation>` section defines the implementation of this composite component.

3.6.1.5 Walk through of the run-time for the simple composite component example

This section gives a non-normative traversal of the composite component feature using the previous expample as a guide. Please refer to the javadocs for the normative specification for each method mentioned below. Any text in *italics* is a term defined in Section 3.6.1.6 “Composite Component Terms”.

1. The user-agent requests the `index.html` from Section 3.6.1.4 “A simple composite component example”. This page contains the ‘`xmlns:ez="http://java.sun.com/jsf/composite/ezcomp"`‘ declaration and an occurrence of the `<ez:loginPanel>` tag. Because this page contains a usage of a composite component, it is called a *using page* for discussion.

The runtime notices the use of an xml namespace beginning with "`http://java.sun.com/jsf/composite/`". Takes the substring of the namespace after the last "/", exclusive, and looks for a resource library with the name "`ezcomp`" by calling `ResourceHandler.libraryExists()`.

2. The runtime encounters the `<ez:loginPanel>` component in the *using page*. This causes `Application.createComponent(FacesContext, Resource)` to be called. This method instantiates the *top level component* but does not populate it with children. Pay careful attention to the javadocs for this method. Depending on the circumstances, the *top level component* instance can come from a developer supplied Java Class, a Script, or an implementation specific java class. This method calls `ViewDeclarationLanguage.getComponentMetadata(FacesContext, Resource)`, which obtains the *composite component BeanInfo* (and therefore also the *composite component BeanDescriptor*) that exposes the *composite component metadata*. The *composite component metadata* also includes any *attached object targets* exposed by the *composite component author*. One thing that `Application.createComponent(FacesContext, Resource)` does to the component before returning it is set the component's renderer type to be `javax.faces.Composite`. This is important during rendering.

   Again, `Application.createComponent(FacesContext, Resource)` does not populate the *top level component* with children. Subsequent processing done as the runtime traverses the rest of the page takes care of that. One very important aspect of that subsequent processing is ensuring that all of the `UIComponent` children in the *defining page* are placed in a facet underneath the *top level component*. The name of that facet is given by the `UIComponent.COMPOSITE_FACET_NAME` constant.

3. After the children of the *composite component tag* in the *using page* have been processed by the VDL implementation, the VDL implementation must call `VDLUtils.retargetAttachedObjects()`. This method examines the *composite component metadata* and retargets any attached objects from the *using page* to their approriate *inner component* targets.

4. Because the renderer type of the composite component was set to `javax.faces.Composite`, the *composite component renderer* is invoked to render the composite component.

3.6.1.6 Composite Component Terms

The following terms are commonly used to describe the composite component feature.

Attached Object Any artifact that can be attached to a `UIComponent` (composite or otherwise). Usually, this means a `Converter`, `Validator`, `ActionListener`, or `ValueChangeListener`.

Attached Object Target Part of the *composite component metadata* that allows the *composite component author* to expose the semantics of an inner component to the *using page author* without exposing the rendering or implementation details of the inner component.

Composite Component A tree of `UIComponent` instances, rooted at a *top level component*, that can be thought of and used as a single component in a view. The component hierarchy of this subtree is described in the *composite component defining page*.

Composite Component Author The individual or role creating the *composite component*. This usually involves authoring the *composite component defining page*.

Composite Component `BeanDescriptor` A constituent element of the *composite component metadata*. This version of the spec uses the JavaBeans API to expose the component metadata for the composite component. Future versions of the spec may use a different API to expose the component metadata.

Composite Component `BeanInfo` The main element of the *composite component metadata*.

Composite Component Declaration The section of markup within the *composite component defining page* that includes the `<composite:interface>` section and its children.

Composite Component Definition The section of markup within the *composite component defining page* that includes the `<composite:implementation>` section and its children.

Composite Component Library A resource library that contains a *defining page* for each *composite component* that the *composite component author* wishes to expose to the *using page author*.

Composite Component Metadata Any data about the *composite component*. The normative specification for what must be in the *composite component metadata* is in the javadocs for `ViewDeclarationLanguage.getComponentMetadata()`.

Composite Component RendererA new renderer in the `HTML_BASIC` render kit that knows how to render a *composite component*.

Composite Component Tag The tag in the *using page* that references a *composite component* declared and defined in a *defining page*.

Defining page The markup page, usually Facelets markup, that contains the *composite component declaration* and *composite component definition*.

Inner Component Any `UIComponent` inside of the *defining page* or a page that is referenced from the *defining page*.

Top level component The `UIComponent` instance in the tree that is the parent of all `UIComponent` instances within the *defining page* and any pages used by that *defining page*.

Using Page The VDL page in which a *composite component tag* is used.

Using Page Author The individual or role that creates pages that use the *composite component*.

3.6.2 Normative Requirements

This section contains the normative requirements for the composite component runtime, or pointers to other parts of the specification that articulate those requirements in the appropriate context.

TABLE 3-1 References to Composite Component Requirements in Context

Section	Feature
Section 5.6.2.1 "Implicit Object ELResolver for Facelets and Programmatic Access"	Ability for the *composite component author* to refer to the *top level component* from an EL expression, such as `#{compositeComponent.children[3]}`.
Section 5.6.2.2 "Composite Component Attributes ELResolver"	Ability for the *composite component author* to refer to attributes declared on the *composite component tag* using EL expressions such as `#{compositeComponent.attrs.usernameLabel}`
Section 7.1.11 "Object Factories"	Methods called by the VDL page to create a new instance of a *top level component* for eventual inclusion in the view
Section 10.3.3 "Requirements specific to composite components"	Requirements of the Facelet implementation relating to Facelets.
Section 10.4.4 "Composite Component Tag Library"	Tag handlers for the `composite` tag library

3.6.2.1 Composite Component Metadata

In the current version of the specification, only composite `UIComponent`s must have component metadata. It is possible that future versions of the specification will broaden this requirement so that all `UIComponent`s must have metadata.

This section describes the implementation of the *composite component metadata* that is returned from the method `ViewDeclarationLanguage.getComponentMetadata()`. This method is formally declared in Section 7.6.2.3 "ViewDeclarationLanguage.getComponentMetadata()", but for reference its signature is repeated here.

```
public BeanInfo getComponentMetadata(FacesContext context,
Resource componentResource)
```

The specification requires that this method is called from `Application.createComponent(FacesContext context, Resource componentResource)`. See the javadocs for that method for actions that must be taken based on the composite component metadata returned from `getComponentMetadata()`.

The default implementation of this method must support authoring the component metadata using tags placed inside of a `<composite:interface />` element found on a *defining page*. This element is specified in the Facelets taglibrary docs.

Composite component metadata currently consists of the following information:

- The *composite component* `BeanInfo`, returned from this method.
- The `Resource` from which the composite component was created.
- The *composite component* `BeanDescriptor`.

  This `BeanDescriptor` must be returned when `getBeanDescriptor()` is called on the composite component `BeanInfo`.

  The composite component `BeanDescriptor` exposes the following information.

  - The "displayName", "name", "shortDescription", "expert", "hidden", and "preferred" attributes of the `<composite:interface/ >` element are exposed using the corresponding methods on the composite component `BeanDescriptor`. Any additional attributes on `<composite:interface/ >` are exposed as attributes accessible from the `getValue()` and `attributeNames()` methods on `BeanDescriptor` (inherited from `FeatureDescriptor`). The return type from `getValue()` must be a `javax.el.ValueExpression` for such attributes.
  - The list of exposed `AttachedObjectTarget`s to which the *page author* can attach things such as listeners, converters, or validators.

    The VDL implementation must populate the composite component metadata with a `List<AttachedObjectTarget>` that includes all of the inner components exposed by the composite component author for use by the page author.

    This List must be exposed in the value set of the composite component `BeanDescriptor` under the key `AttachedObjectTarget.ATTACHED_OBJECT_TARGETS_KEY`.

    For example, if the defining page has

```
<composite:interface>
  <composite:editableValueHolder name=″username″ />
  <composite:actionSource name=″loginEvent″ />
  <composite:actionSource name=″allEvents″
                          targets=″loginEvent,cancelEvent″ />
<composite:interface>
```

    The list of attached object targets would consist of instances of implementations of the following interfaces from the package `javax.faces.webapp.vdl`.

    i. EditableValueHolderAttachedObjectTarget

    ii. ActionSource2AttachedObjectTarget

    iii. ActionSource2AttachedObjectTarget

- A `ValueExpression` that evaluates to the component type of the composite component. By default this is "`javax.faces.NamingContainer`" but the composite component page author can change this, or provide a script-based `UIComponent` implementation that is required to implement `NamingContainer`.

  This `ValueExpression` must be exposed in the value set of the composite component `BeanDescriptor` under the key `UIComponent.COMPOSITE_COMPONENT_TYPE_KEY`.

- A `Map<String, PropertyDescriptor>` representing the facets declared by the composite component author for use by the page author.

  This `Map` must be exposed in the value set of the composite component BeanDescriptor under the key `UIComponent.FACETS_KEY`.

- Any attributes declared by the composite component author using `<composite:attribute/ >` elements must be exposed in the array of `PropertyDescriptor`s returned from `getPropertyDescriptors()` on the composite component `BeanInfo`.

  For each such attribute, for any `String` or `boolean` valued `JavaBeans` properties on the interface `PropertyDescriptor` (and its superinterfaces) that are also given as attributes on a `<composite:attribute/ >` element, those properties must be exposed as properties on the `PropertyDescriptor` for that markup element. Any additional attributes on `<composite:attribute/ >` are exposed as attributes accessible from the `getValue()` and `attributeNames()` methods on `PropertyDescriptor`. The return type from getValue() must be a `ValueExpression`.

3.7 Component Behavior Model

This section describes the facilities for adding Behavior attached objects to JavaServer Faces components.

3.7.1 Overview

JSF supports a mechanism for enhancing components with additional behaviors that are not explicitly defined by the component author.

At the root of the behavior model is he Behavior interface. This interface serves as a supertype for additional behavior contracts. The ClientBehavior interface extends the Behavior interface by providing a contract for defining reusable scripts that can be attached to any component that implements the ClientBehaviorHolder interface. The ClientBehaviorHolder interface defines the set of attach points, or "events", to which a ClientBehavior may be attached. For example, an "AlertBehavior" implementation might display a JavaScript alert when attached to a component and activated by the end user.

While client behaviors typically add client-side capabilities, they are not limited to client. Client behaviors can also participate in the JSF request processing lifecycle. JSF's AjaxBehavior is a good example of such a cross-tier behavior. The AjaxBehavior both triggers an Ajax request from the client and also delivers AjaxBehaviorEvents to listeners on the server.

The standard HTML components provided by JSF are all client behavior-ready. That is, all of the standard HTML components implement the ClientBehaviorHolder interface and allow client behaviors to be attached to well defined events. .

3.7.2 Behavior Interface

The Behavior interface is the root of the component behavior model. It defines a single method to enable generic behavior event delivery.

```
public void broadcast(BehaviorEvent event)
        throws AbortProcessingException
```

This method is called by UIComponent implementations to re-broadcast behavior events that were queued by by calling UIComponent.queueEvent.

3.7.3 BehaviorBase

The BehaviorBase abstract class implements the broadcast method from the Behavior interface. BehaviorBase also implements the PartialStateHolder interface (see Section 3.2.5 “PartialStateHolder”). It also provides behavior event listener registration methods.

```
public void broadcast(BehaviorEvent event)
        throws AbortProcessingException
```

This method delivers the BehaviorEvent to listeners that were registered via addBehaviorListener.

The following methods are provided for add and removing BehaviorListeners..

```
protected void addBehaviorListener(BehaviorListener listener)
```

```
protected void removeBehaviorListener(BehaviorListener listener);
```

3.7.4 The Client Behavior Contract

The `ClientBehavior` interface extends the `Behavior` interface and lays the foundation on which behavior authors can define custom script producing behaviors. The logic for producing these scripts is defined in the `getScript`() method.

```
public String getScript(BehaviorContext behaviorContext)
```

This method returns a String that is an executable script that can be attached to a client side event handler. The `BehaviorContext` argument contains information that may be useful for `getScript` implementations.

In addition to client side functionality, client behaviors can also post back to the server and participate in the request processing lifecycle. ..

```
public void decode(FacesContext context,UIComponent component)
```

This method can perform request decoding and queue server side events..].

```
public Set<ClientBehaviorHint> getHints()
```

This method provides information about the client behavior implementation that may be useful to components and renderers that interact with the client behavior.

Refer to the javadocs for these methods for more details.

3.7.5 ClientBehavorHolder

Components that support client behaviors must implement the ClientBehaviorHolder interface. Refer to Section 3.2.9 “ClientBehaviorHolder” for more details.

3.7.6 ClientBehaviorRenderer

Client behaviors may implement script generation and decoding in a client behavior class or delegate to a ClientBehaviorRenderer. Refer to Section 8.3 “ClientBehaviorRenderer” for more specifics.

3.7.7 ClientBehaviorContext

The specification provides a `ClientBehaviorContext` that contains information that may be used at script rendering time. Specifically it includes:

- FacesContext
- UIComponent that the current behavior is attached to
- The name of the event that the behavior is associated with
- The identifier of the source - this may correspond to the identifier of the source of the behavior
- A collection of parameters that submitting behaviors should include when posting back to the server

The `ClientBehaviorContext` is created with the use of this static method:

```
public static ClientBehaviorContext
createClientBehaviorContext(FacesContext context,UIComponent
component,String eventName,tring
sourceId,Collection<ClientBehaviorContext.Parameter> parameters)
```

This method must throw a NullPointerException if `context, component` or `eventName` is null.

3.7.8 ClientBehaviorHint

The `ClientBehaviorHint` enum is used to convey information about the client behavior implementation. Currently, only one hint is provided.

```
SUBMITTING
```

This hint indicates that a client behavior implementation posts back to the server.

3.7.9 ClientBehaviorBase

`ClientBehaviorBase` is an extension of `BehaviorBase` that implements the `ClientBehavior` interface. It It is a convenience class that contains default implementations for the methods in `ClientBehavior` plus additional methods::

```
public String getScript(BehaviorContext behaviorContext)
```

The default implementation calls getRenderer to retrieve the `ClientBehaviorRenderer`. If a `ClientBehaviorRenderer` is found, it is used to obtain the script. If no `ClientBehaviorRenderer` is found, this method returns null.

```
public void decode(FacesContext context,UIComponent component)
```

The default implementation calls getRenderer to retrieve the `ClientBehaviorRenderer`. If a `ClientBehaviorRenderer` is found, it is used to perform decoding. If no `ClientBehaviorRenderer` is found, no decoding is performed.

```
public Set<ClientBehaviorHint> getHints()
```

The default implementation returns an empty set

```
public String getRendererType();
```

This method identifies the `ClientBehaviorRenderer` type. By default, no `ClientBehaviorRenderer` type is provided. Subclasses should either override this method to return a valid type or override the getScript and decode methods if a `ClientBehaviorRenderer` is not available..

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

This method returns the `ClientBehaviorRenderer` instance that is associated with this `ClientBehavior.` It uses the renderer type returned from get`RendererType()` to look up the renderer on the RenderKit using `RenderKit.getClientBehaviorRenderer.`

3.7.10 Behavior Event / Listener Model

The behavior event / listener model is an extension of the JSF event / listener model as described in Section 3.4 “Event and Listener Model”. `BehaviorHolder` components are responsible for broadcasting `BehaviorEvents` to `behaviors.`

3.7.10.1 Event Classes

Behaviors can broadcast events in the same way that UIComponents can broadcast events. At the root of the behavior event hierarchy is `BehaviorEvent` that extends `javax.faces.event.FacesEvent.` All events that are broadcast by JSF behaviors must extend the `javax.faces.event.BehaviorEvent` abstract base class. The parameter list for the constructor(s) of this event class must include a `UIComponent`, which identifies the component from which the event will be broadcast to interested listeners, and a `Behavior` which identifies the behavior associated

with the component. The source component can be retrieved from the event object itself by calling `getComponent` and the behavior can be retrieved by calling `getBehavior`. Additional constructor parameters and/or properties on the event class can be used to relay additional information about the event.

In conformance to the naming patterns defined in the *JavaBeans Specification*, event classes typically have a class name that ends with `Event`. The following method is available to determine the `Behavior` for the event (in addition to the other methods inherited from `javax.faces.event.FacesEvent:`

```
public Behavior getBehavior()
```

3.7.10.2 Listener Classes

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

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

3.7.10.3 Listener Registration

`BehaviorListener` registration follows the same conventions as outlined in Section 3.7.10.3 “Listener Registration”.

3.7.11 Ajax Behavior

3.7.11.1 AjaxBehavior

The specification defines a single concrete `ClientBehavior` implementation: `javax.faces.component.behavior.AjaxBehavior`. This class extends `javax.faces.component.behavior.ClientBehaviorBase`. The presence of this behavior on a component causes the rendering of JavaScript that will produce an Ajax request to the server using the JavaScript API outlined in Section “JavaScript API”. This behavior may also broadcast `javax.faces.event.AjaxBehaviorEvents` to registered `javax.faces.event.AjaxBehaviorListener` implementations. Refer to the javadocs for more details about `AjaxBehavior.` [P1-start-ajaxbehavior]This behavior must define the behavior id “javax.faces.behavior.Ajax”. The renderer type must also be “javax.faces.behavior.Ajax”.[P1-end]

3.7.11.2 Ajax Behavior Event / Listener Model

Corresponding to the standard behavior event classes described in the previous section the specification supports an event listener model for broadcasting and handling `AjaxBehavior` events.

javax.faces.event.AjaxBehaviorEvent

This event type extends from `javax.faces.event.BehaviorEvent` and it is broadcast from an `AjaxBehavior.` This class follows the standard JSF event / listener model, incorporating the usual methods as outlined in Section 3.4 "Event and Listener Model". This class is responsible for invoking the method implementation of `javax.faces.event.AjaxBehaviorListener.processAjaxBehavior.` Refer to the javadocs for more complete details about this class.

javax.faces.event.AjaxBehaviorListener

This listener type extends from `javax.faces.event.BehaviorListener` and it is invoked in response to `AjaxBehaviorEvents.`

```
public void processAjaxBehavior(AjaxBehaviorEvent event)
```

`AjaxBehaviorListener` implementations implement this method to provide server side functionality in response to `AjaxBehaviorEvents.` See the javadocs for more details about this class.

3.7.12 Adding Behavior To Components

Using the `ClientBehaviorHolder` interface (`Section 3.2.9 "ClientBehaviorHolder"`) `ClientBehavior` instances can be added to components. For `ClientBehavior` implementations that extend `UIComponentBase`, the minimal requirement is to override `getEventNames()` to return a non-empty collection of the event names exposed by the `ClientBehaviorHolder`. A optional default event name may be specified as well. For example:

Here's an example code snippet from one of the Html components:

```
public class HtmlCommandButton extends
javax.faces.component.UICommand implements ClientBehaviorHolder {
...
private static final Collection<String> EVENT_NAMES =
Collections.unmodifiableCollection(Arrays.asList("blur","change"
,"click","action",...));

    public Collection<String> getEventNames() {
        return EVENT_NAMES; }

    public String getDefaultEventName() {
        return "action";    }
...
```

Users of the component will be able to attach `ClientBehavior` instances to any of the event names specified by the `getEventNames()` implementation by calling `ClientBehaviorHolder.addBehavior(eventName, clientBehavior)`.

3.7.13 Behavior Registration

JSF provides methods for registering `Behavior` implementations and these methods are similar to the methods used to register converters and validators. Refer to Section 7.1.11 "Object Factories" for the specifics about these methods.

3.7.13.1 XML Registration

JSF provides the usual faces-config.xml registration of custom component behavior implementations.

```
<behavior>
  <behavior-id>custom.behavior.Greet</behavior-id>
  <behavior-class>greet.GreetBehavior</behavior-class>
</behavior>
```

3.7.13.2 Registration By Annotation

JSF provides the `@FacesBehavior` annotation for registering custom behavior implementations.

```
@FacesBehavior(value="custom.behavior.Greet")
public class GreetBehavior extends BehaviorBase implements
Serializable {
...
}
```

4

Standard User Interface Components

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

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

4.1 Standard User Interface Components

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

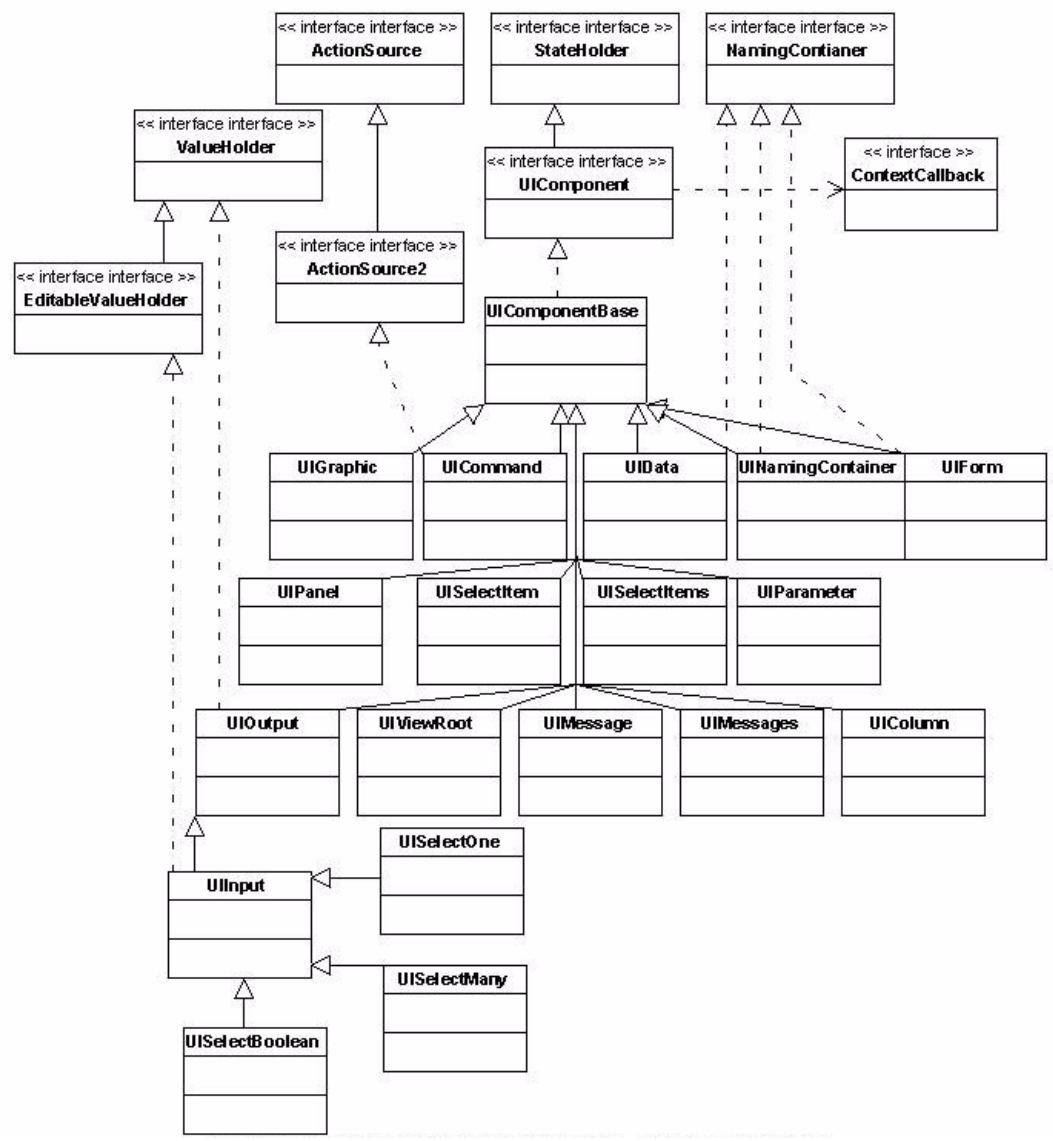
[P1-start-componentConstant]The implementation for each standard `UIComponent` class must specify two public static final String constant values:

- `COMPONENT_TYPE` -- The standard component type identifier under which the corresponding component class is registered with the `Application` object for this application. This value may be used as a parameter to the `createComponent()` method.
- `COMPONENT_FAMILY` -- The standard component family identifier used to select an appropriate Renderer for this component.[P1-end]

For all render-independent properties in the following sections (except for `id`, `scope`, and `var`) the value may either be a literal, or it may come from a value expression. Please see Section 5.1 “Value Expressions” for more information.

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

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



4.1.1 UIColumn

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

4.1.1.1 Component Type

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

4.1.1.2 Properties

`UIColumn` adds the following render-independent properties:

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

[P1-start-uicolumn]`UIColumn` specializes the behavior of render-independent properties inherited from the parent class as follows:

- The default value of the `family` property must be set to "javax.faces.Column".
- The default value of the `rendererType` property must be set to `null`.[P1-end]

4.1.1.3 Methods

`UIColumn` adds no new processing methods.

4.1.1.4 Events

`UIColumn` adds no new event handling methods.

4.1.2 UICommand

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

4.1.2.1 Component Type

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

4.1.2.2 Properties

`UICommand` adds the following render-independent properties.

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

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

`[P1-start-uicommand]UICommand` components specialize the behavior of render-independent properties inherited from the parent class as follows:

- The default value of the `family` property must be set to "javax.faces.Command".
- The default value of the `rendererType` property must be set to "javax.faces.Button".[P1-end]

4.1.2.3 Methods

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

4.1.2.4 Events

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

4.1.3 UIData

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

4.1.3.1 Component Type

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

4.1.3.2 Properties

`UIData` adds the following render-independent properties.

Name	Access	Type	Description
`dataModel`	protected RW	`DataModel`	The internal value representation of the `UIData` instance. Subclasses might write to this property if they want to restore the internal model during the *Restore View Phase* or if they want to explicitly refresh the model for the *Render Response* phase.
`first`	RW	`int`	Zero-relative row number of the first row in the underlying data model to be displayed, or zero to start at the beginning of the data model.
`footer`	RW	`UIComponent`	Convenience methods to get and set the “footer” facet for this component.
`header`	RW	UIComponent	Convenience methods to get and set the “header” facet for this component.
rowCount	RO	int	The number of rows in the underlying `DataModel`, which can be -1 if the number of rows is unknown.
rowAvailable	RO	boolean	Return `true` if there is row data available for the currently specified `rowIndex`; else return `false`.
rowData	RO	Object	The data object representing the data for the currently selected `rowIndex` value.
rowIndex	RW	int	Zero-relative index of the row currently being accessed in the underlying `DataModel`, or -1 for no current row. See below for further information.
rows	RW	int	The number of rows (starting with the one identified by the `first` property) to be displayed, or zero to display the entire set of available rows.
value	RW	Object	The `DataModel` instance representing the data to which this component is bound, or a collection of data for which a `DataModel` instance is synthesized. See below for more information.
var	RW	String	The request-scope attribute (if any) under which the data object for the current row will be exposed when iterating.

See Section 3.2.3 “NamingContainer” for information about properties introduced by the implemented classes.

`[P1-start-uidata]UIData` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to “javax.faces.Data”.
- The default value of the `rendererType` property must be set to “`javax.faces.Table`”.[P1-end]

The current value identified by the `value` property is normally of type `DataModel`. [P1-start-uidataModel]However, a `DataModel` wrapper instance must automatically be provided by the JSF implementation if the current value is of one of the following types:

- `java.util.List`
- Array of `java.util.Object`
- `java.sql.ResultSet` (which therefore also supports `javax.sql.RowSet`)
- `javax.servlet.jsp.jstl.sql.Result`
- Any other Java object is wrapped by a `DataModel` instance with a single row.[P1-end]

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

4.1.3.3 Methods

`UIData` adds no new processing methods. However, the getDataModel() method is now protected, so implementations have access to the underlying data model. See Section 3.2.3 “NamingContainer” for information about methods introduced by the implemented classes.

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

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

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

[P1-start-uidataDecode]`UIData` specializes the behavior of the `processDecodes()`, `processValidators()`, and `processUpdates()` methods inherited from its parent as follows:

- For each of these methods, the `UIData` implementation must iterate over each row in the underlying data model, starting with the row identified by the `first` property, for the number of rows indicated by the `rows` property, by calling the `setRowIndex()` method.
- When iteration is complete, set the `rowIndex` property of this component, and of the underlying `DataModel`, to zero, and remove any request attribute exposed via the `var` property.[P1-end]

`UIData` specializes the behavior of `invokeOnComponent()` inherited from `UIComponentBase` to examine the argument `clientId` and extract the `rowIndex`, if any, and position the data properly before proceeding to locate the component and invoke the callback. Upon normal or exception return from the callback the data must be repositioned to match how it was before invoking the callback. Please see the javadocs for `UIData.invokeOnComponent()` for more details.

4.1.3.4 Events

`UIData` adds no new event handling methods. SeeSection 3.2.3 “NamingContainer” for information about event handling introduced by the implemented classes.

4.1.4 UIForm

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

[P1-start-uiformEncodeEnd]The `encodeEnd()` method of the renderer for `UIForm` must call `ViewHandler.writeState()` *before* writing out the markup for the closing tag of the form.[P1-end]This allows the state for multiple forms to be saved.

4.1.4.1 Component Type

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

4.1.4.2 Properties

`UIForm` adds the following render-independent properties.

Name	Access	Type	Description
`prependId`	RW	`boolean`	If true, this `UIForm` instance does allow its id to be pre-pendend to its descendent's id during the generation of clientIds for the descendents. The default value of this property is `true`.

`[P1-start-uiform]UIForm` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "`javax.faces.Form`".
- The default value of the `rendererType` property must be set to "`javax.faces.Form`".[P1-end]

4.1.4.3 Methods.

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

[P1-start-uiform-setSubmitted]The `setSubmitted()` method of each `UIForm` instance in the view must be called during the *Apply Request Values* phase of the request processing lifecycle, during the processing performed by the `UIComponent.decode()` method. If this `UIForm` instance represents the form actually being submitted on this request, the parameter must be set to `true`; otherwise, it must be set to `false`.[P1-end] The standard implementation of `UIForm` delegates the responsibility for calling this method to the `Renderer` associated with this instance.

[P1-start-uiform-submitted]The value of a `UIForm`'s `submitted` property must not be saved as part of its state.[P1-

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

end]

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

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

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

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

[P1-start-uiformSaveState]The `saveState()` method of UIForm must call `setSubmitted(false)` before calling `super.saveState()` as an extra precaution to ensure the submitted state is not persisted across requests.[P1-end]

```
protected String getContainerClientId(FacesContext context);
```

[P1-start-uiformPrependId]Override the parent method to ensure that children of this `UIForm` instance in the view have the form's `clientId` prepended to their `clientId`s if and only if the form's `prependId` property is `true`.[P1-end]

4.1.4.4 Events

`UIForm` adds no new event handling methods.

4.1.5 UIGraphic

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

4.1.5.1 Component Type

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

4.1.5.2 Properties

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

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

[P1-start-uigraphic]`UIGraphic` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "javax.faces.Graphic".
- The default value of the `rendererType` property must be set to "`javax.faces.Image`".[P1-end]

4.1.5.3 Methods

`UIGraphic` adds no new processing methods.

4.1.5.4 Events

`UIGraphic` does not originate any standard events.

4.1.6 UIInput

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

4.1.6.1 Component Type

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

4.1.6.2 Properties

`UIInput` adds the following renderer independent properties.:

Name	Access	Type	Description
`requiredMessage`	RW	`String`	ValueExpression enabled property. If non-null, this property is used as the `summary` and `detail` strings of the `FacesMessage` that is queued on the `FacesContext` instead of the default message for the required validaiton failure. Note that the message is fully internationalizable via either the `f:loadBundle` tag or via `ResourceBundle` access from the EL.
`converterMessage`	RW	`String`	ValueExpression enabled property. If non-null, this property is used as the `summary` and `detail` strings of the `FacesMessage` that is queued on the `FacesContext` instead of the default message for conversion failure. Note that the message is fully internationalizable via either the `f:loadBundle` tag or via `ResourceBundle` access from the EL.
`validatorMessage`	RW	`String`	ValueExpression enabled property. If non-null, this property is used as the `summary` and `detail` strings of the `FacesMessage` that is queued on the `FacesContext` instead of the default message for validation failure. Note that the message is fully internationalizable via either the `f:loadBundle` tag or via `ResourceBundle` access from the EL.

See Section 3.2.7 "EditableValueHolder" for information about properties introduced by the implemented interfaces.

[P1-start-uiinput]UIInput specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "`javax.faces.Input`".
- The default value of the `rendererType` property must be set to "`javax.faces.Text`".
- The `Converter` specified by the `converter` property (if any) must also be used to perform String->Object conversions during decoding.[P1-end]
- If the `value` property has an associated `ValueExpression`, the `setValue()` method of that `ValueExpression` will be called during the *Update Model Values* phase of the request processing lifecycle to push the local value of the component back to the corresponding model bean property.

4.1.6.3 Methods

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

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

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

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

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

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

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

```
public void resetValue();
```

Perform the algorithm described in the javadoc to reset this `UIInput` to the state where it has no local value. This method does not touch the value expresson associated with the "`value`" property.

4.1.6.4 Events

All events are described in *Section 3.2.7 "EditableValueHolder"*.

4.1.7 UIMessage

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

4.1.7.1 Component Type

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

4.1.7.2 Properties

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

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

[P1-start-uimessage]`UIMessage` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "`javax.faces.Message`".
- The default value of the `rendererType` property must be set to "`javax.faces.Message`".[P1-end]

4.1.7.3 Methods.

`UIMessage` adds no new processing methods.

4.1.7.4 Events

`UIMessage` adds no new event handling methods.

4.1.8 UIMessages

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

4.1.8.1 Component Type

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

4.1.8.2 Properties

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

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

`[P1-stat-uimessages]UIMessages` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "`javax.faces.Messages`".
- The default value of the `rendererType` property must be set to "`javax.faces.Messages`".[P1-end]

4.1.8.3 Methods.

`UIMessages` adds no new processing methods.

4.1.8.4 Events

`UIMessages` adds no new event handling methods.

4.1.9 UIOutcomeTarget

UIOutcomeTarget (`UIOutput`) is a component that has a value and an outcome, either which may optionally be retrieved from a model tier bean via a value expression (see Section 5.1 “Value Expressions”), and is displayed to the user as a hyperlink, appearing in the form of a link or a button. The user cannot modify the value of the hyperlink, as it's for display purposes only. The target URL of the hyperlink is derived by passing the outcome to the `ConfigurationNavigationHandler` to retrieve the matching `NavigationCase` and then using the `ViewHandler` to translate the `NavigationCase` into an action URL. When the client activates the hyperlink, typically by clicking it, the target URL is retrieved using a non-faces request and the response is rendered.

This component introduces a scenario known as "preemptive navigation". The navigation case is resolved during the Render Response phase, before the client activates the link (and may never activate the link). The predetermined navigation is pursued after the client activates the link. In contrast, the UICommand components resolve and execute the navigation at once, after the Invoke Application phase.

The `UIOutcomeTarget` component allows the developer to leverage the navigation model while at the same time being able to generate bookmarkable, non-faces requests to be included in the response.

4.1.9.1 Component Type

The standard component type for UIOutcomeTarget is "javax.faces.OutcomeTarget".

4.1.9.2 Properties

The following render-independent properties are added by thec component:

Name	Access	Type	
Outcome	RW	String	The logical outcome that is used to resolve a NavigationCase which in turn is used to build the target URL of this component. Default value is the current view ID.
includePageParams	RW	boolean	Flag indicating whether the page parameters should be appended to the query string of the target URL. Default value is "false".

[P1-start-uioutcometarget]UIOutcomeTarget specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the family property must be set to "javax.faces.UIOutcomeTarget"
- The default value of the rendererType property must be set to "javax.faces.Link" [P1-end]

4.1.9.3 Methods

The UIOutcomeTarget adds no event handling methods.

4.1.9.4 Events

The UIOutcomeTarget adds no event handling methods.

4.1.10 UIOutput

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

4.1.10.1 Component Type

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

4.1.10.2 Properties

`UIOutput` adds no new render-independent properties. See Section 3.2.6 “ValueHolder” for information about properties introduced by the implemented classes.

`[P1-start-uioutput]UIOutput` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to “javax.faces.Output”.
- The default value of the `rendererType` property must be set to “javax.faces.Text”.[P1-end]

4.1.10.3 Methods

`UIOutput` adds no new processing methods. See Section 3.2.6 “ValueHolder” for information about methods introduced by the implemented interfaces.

4.1.10.4 Events

UIOutput does not originate any standard events. See Section 3.2.6 “ValueHolder” for information about events introduced by the implemented interfaces.

4.1.11 UIPanel

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

4.1.11.1 Component Type

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

4.1.11.2 Properties

`UIPanel` adds no new render-independent properties.

`[P1-start-uipanel]UIPanel` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "`javax.faces.Panel`".
- The default value of the `rendererType` property must be set to `null`.[P1-end]

4.1.11.3 Methods

`UIPanel` adds no new processing methods.

4.1.11.4 Events

`UIPanel` does not originate any standard events

4.1.12 UIParameter

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

4.1.12.1 Component Type

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

4.1.12.2 Properties

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

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

[P1-start-uiparameter]`UIParameter` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "javax.faces.Parameter".
- The default value of the `rendererType` property must be set to `null`.[P1-end]

4.1.12.3 Methods

`UIParameter` adds no new processing methods.

4.1.12.4 Events

`UIParameter` does not originate any standard events

4.1.13 UISelectBoolean

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

4.1.13.1 Component Type

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

4.1.13.2 Properties

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

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

[P1-start-uiselectboolean]`UISelectBoolean` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "`javax.faces.SelectBoolean`".
- The default value of the `rendererType` property must be set to "`javax.faces.Checkbox`".[P1-end]

4.1.13.3 Methods

`UISelectBoolean` adds no new processing methods.

4.1.13.4 Events

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

4.1.14 UISelectItem

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

4.1.14.1 Component Type

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

4.1.14.2 Properties

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

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

[P1-start-uiselectitem]`UISelectItem` specializes the behavior of render-independent properties inherited

- The default value of the `family` property must be set to "javax.faces.SelectItem".
- The default value of the `rendererType` property must be set to `null`.
- If the `value` property is non-`null`, it must contain a `SelectItem` instance used to configure the selection item specified by this component.
- If the `value` property is a value expression, it must point at a `SelectItem` instance used to configure the selection item specified by this component.
- If the `value` property is `null`, and there is no corresponding value expression, the `itemDescription`, `itemDisabled`, `itemLabel` and `itemValue` properties must be used to construct a new `SelectItem` representing the selection item specified by this component.[P1-end]

4.1.14.3 Methods

`UISelectItem` adds no new processing methods.

4.1.14.4 Events

`UISelectItem` does not originate any standard events.

4.1.15 UISelectItems

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

4.1.15.1 Component Type

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

4.1.15.2 Properties

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

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

[P1-start-uiselectitems]`UISelectItems` specializes the behavior of render-independent properties inherited

- The default value of the `family` property must be set to "`javax.faces.SelectItems`".
- The default value of the `rendererType` property must be set to `null`.
- If the `value` property (or the value returned by a value expression associated with the `value` property) is non-null, it must contain a `SelectItem` bean, an array of `SelectItem` beans, a `Collection` of `SelectItem` beans, or a `Map`, where each map entry is used to construct a `SelectItem` bean with the key as the `label` property of the bean, and the value as the `value` property of the bean (which must be of the same basic type as the value of the parent component's value).[P1-end]

4.1.15.3 Methods

`UISelectItems` adds no new processing methods.

4.1.15.4 Events

`UISelectItems` does not originate any standard events.

4.1.16 UISelectMany

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

4.1.16.1 Component Type

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

4.1.16.2 Properties

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

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

[P1-start-uiselectmany]`UISelectMany` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "`javax.faces.SelectMany`".
- The default value of the `rendererType` property must be set to "`javax.faces.Listbox`".[P1-end]
- See the class Javadocs for `UISelectMany` for additional requirements related to implicit conversions for the `value` property.

4.1.16.3 Methods

[P1-start-uselectmany-validate]`UISelectMany` must provide a specialized `validate()` method which ensures that any decoded values are valid options (from the nested `UISelectItem` and `UISelectItems` children).[P1-end]

4.1.16.4 Events

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

4.1.17 UISelectOne

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

4.1.17.1 Component Type

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

4.1.17.2 Properties

`UISelectOne` adds no new render-independent properties.

`[P1-start-uiselectone]UISelectOne` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "`javax.faces.SelectOne`".
- The default value of the `rendererType` property must be set to "`javax.faces.Menu`".[P1-end]

4.1.17.3 Methods

`[P1-start-uiselectone-validate]UISelectOne` must provide a specialized `validate()` method which ensures that any decoded value is a valid option (from the nested `UISelectItem` and `UISelectItems` children).[P1-end]

4.1.17.4 Events

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

4.1.18 UIViewParameter

`UIViewParameter` (extends `UIInput`) is a component that allows the query parameters included in the request by `UIOutcomTarget` renderers to participate in the lifecycle. Please see the javadocs for the normative speficication of this component.Events.

4.1.19 UIViewRoot

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

4.1.19.1 Component Type

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

4.1.19.2 Properties

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

Name	Access	Type	Description
locale	RW	java.util.Locale	The Locale to be used in localizing the response for this view.
renderKitId	RW	String	The id of the `RenderKit` used to render this page.
`viewId`	RW	`String`	The view identifier for this view.
`beforePhaseListener`	RW	MethodExpression	`MethodExpression` that will be invoked before all lifecycle phases except for *Restore View*.
`afterPhaseListener`	RW	MethodExpression	MethodExpression that will be invoked after all lifecycle phases except for *Restore View*.
viewMap	RW	java.util.Map	The `Map` that acts as the interface to the data store that is the "view scope".

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

[P1-start-viewmap]The viewMap property is lazily created the first time it is accessed, and it is destroyed when a different `UIViewRoot` instance is installed from a call to `FacesContext.setViewRoot()`. After the Map is created a `PostConstructViewMapEvent` must be published using `UIViewRoot` as the event source. Immediately before the Map is destroyed, a `PreDestroyViewMapEvent` must be published using `UIViewRoot` as the event source. [P1-end]

[P1-start-uiviewroot]`UIViewRoot` specializes the behavior of render-independent properties inherited from the parent component as follows:

- The default value of the `family` property must be set to "`javax.faces.ViewRoot`".
- The default value of the `rendererType` property must be set to `null`.[P1-end]

4.1.19.3 Methods

The following methods are used for adding `UIComponent` resources to a target area in the view, and they are also used for retrieving `UIComponent` resources from a target area in the view.

```
public void addComponentResource(FacesContext context,
UIComponent componentResource);
```

Add `componentResource,` that is assumed to represent a resource instance, to the current view. A resource instance is rendered by a resource `Renderer` (such as `ScriptRenderer, StylesheetRenderer`) as described in the Standard HTML RenderKit. This method will cause the resource to be rendered in the "head" element of the view.

```
public void addComponentResource(FacesContext context,
UIComponent componentResource, String target);
```

Add `componentResource,` that is assumed to represent a resource instance, to the current view at the specified `target` location. [P1-start-addComponentResource] The resource must be added using the algorithm outlined in this method's Javadocs.[P1-end]

```
public List<UIComponent> getComponentResources(String target);
```

Return a `List` of `UIComponent` instances residing under the facet identified by `target.` Each `UIComponent` instance in the `List` represents a resource. [P1-start-getCompRes] The `List` must be formulated in accordance with this method's Javadocs. [P1-end]

`UIViewRoot` specializes the behavior of the `UIComponent.queueEvent()` method to maintain a list of queued events that can be transmitted later. It also specializes the behavior of the `processDecodes()`, `processValidators()`, `processUpdates()`, and `processApplication()` methods to broadcast queued events to registered listeners. `UIViewRoot` clears any remaining events from the event queue in these methods if `responseComplete()` or `renderResponse()` has been set on the `FacesContext.` Please see Section 2.2.2 "Apply Request Values", Section 2.2.3 "Process Validations", Section 2.2.4 "Update Model Values" and Section 2.2.5 "Invoke Application" for more details.

4.1.19.4 Events

`UIViewRoot` is a source of `PhaseEvent` events, which are emitted when the instance moves through all phases of the request processing lifecycle except *Restore View*. This phase cannot emit events from `UIViewRoot` because the `UIViewRoot` instance isn't created when this phase starts. See Section 12.2 "PhaseEvent" and Section 12.3 "PhaseListener"for more details on the event and listener class.

```
public void addPhaseListener(PhaseListener listener);

public void removePhaseListener(VPhaseListener listener);

public List<PhaseListener> getPhaseListeners();
```

[P1-start-events]`UIViewRoot` must listen for the top level `PostAddToViewEvent` event sent by the *Restore View* phase. Refer to *Section 2.2.1 "Restore View"* for more details about the publishing of this event. Upon receiving this event, `UIViewRoot` must cause any "after" *Restore View* phase listeners to be called.[P1-end]

`UIViewRoot` is also the source for several kinds of system events. The system must publish a `PostAddToViewEvent`, with the `UIViewRoot` as the source, during the *Restore View* phase, immediately after the new `UIViewRoot` is set into the `FacesContext` for the request. The system must publish a `PreRenderView` event,

with `UIViewRoot` as the source, during the *Render Response* phase, immediately before `ViewHandler.renderView()` is called. The `UIViewRoot` instance itself must override `processRestoreState()` and directly call `processEvent()`, passing a `PostRestoreStateEvent` instance as specified in the Javadocs for `UIViewRoot.processRestoreState()`.

4.1.19.5 Partial Processing

`UIViewRoot` adds special behavior to `processDecodes, processValidators, processUpdates, getRendersChildren and encodeChildren` to facilitate partial processing - namely the ability to have one or more components processed through the `execute` and/or `render` phases of the request processing lifecycle. Refer to Section 13.4 “Partial View Traversal”, Section 13.4.2 “Partial View Processing”, Section 13.4.3 “Partial View Rendering” for an overview of partial processing. [P1-start-viewroot-partial]`UIViewRoot` must perform partial processing as outlined in the Javadocs for the “processXXX” and “encodeXXX” methods if the current request is a partial request.[P1-end]

4.2 Standard UIComponent Model Beans

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

4.2.1 DataModel

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

4.2.1.1 Properties

An instance of `DataModel` supports the following properties:

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

4.2.1.2 Methods

`DataModel` must provide an `iterator()` to iterate over the row data for this model.

4.2.1.3 Events

No events are generated for this component.

4.2.1.4 Concrete Implementations

[P1-start-datamodel]The JSF implementation must provide concrete implementations of DataModel (in the javax.faces.model package) for the following data wrapping scenarios:

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

Each concrete `DataModel` implementation must extend the `DataModel` abstract base class, and must provide a constructor that accepts a single parameter of the object type being wrapped by that implementation (in addition to a zero-args constructor).[P1-end] See the JavaDocs for specific implementation requirements on `DataModel` defined methods, for each of the concrete implementation classes.

4.2.2 SelectItem

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

4.2.2.1 Properties

An instance of `SelectItem` supports the following properties:

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

4.2.2.2 Methods

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

4.2.2.3 Events

An instance of SelectItem supports no events.

4.2.3 SelectItemGroup

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

4.2.3.1 Properties

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

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

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

4.2.3.2 Methods

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

4.2.3.3 Events

An instance of `SelectItemGroup` supports no events.

5

Expression Language and Managed Bean Facility

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

JavaServer Faces relies on the Unified Expression Language (Unified EL, or just EL) provided by version 2.1 of the JavaServer Pages specification (JSR-245). The EL is described in a separate specification document delivered as part of the JSP 2.1 spec. Please consult that document for complete details about the EL.

Versions 1.0 and 1.1 of JavaServer Faces included a built in expression language and required an implementation of it. The API for this old JSF EL is still preserved as deprecated classes and methods, and implementations must still support that API. Please consult the Section 2.1.0.1 “Guide to Deprecated Methods Relating to the Unified EL and their Corresponding Replacements” for details. This chapter will focus exclusively on how Faces leverages and integrates with the Unified EL. It does not describe how the Unified EL operates.

5.1 Value Expressions

5.1.1 Overview

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

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

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

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

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

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

Value expressions can also be used to set a value from the user into the item obtained by evaluating the expression. For example:

```
<h:inputText value="#{employee.number}"/>
```

When the page is rendered, the expression is evaluated as an r-value and the result is displayed as the default value in the text field. When the page is submitted, the expression is evaluated as an l-value, and the value entered by the user (subject to conversion and validation as usual) is pushed into the expression.

5.1.2 Value Expression Syntax and Semantics

Please see Section 1.2 of the *Expression Language Specification, Version 2.1* for the complete specification of ValueExpression syntax and semantics.

5.2 MethodExpressions

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

- During *Apply Request Values* or *Invoke Application* phase (depending upon the state of the `immediate` property), components that implement the `ActionSource2` behavioral interface (see Section 3.2.2 “ActionSource2”) utilize `MethodExpressions` as follows:
  - If the `actionExpression` property is specified, it must be a `MethodExpression` expression that identifies an Application Action method (see Section 7.3 “Application Actions”) that takes no parameters and returns a String.
  - It’s possible to have a method expression act as an `ActionListener` by using the classs `MethodExpressionActionListener` to wrap a method expression and calling the `addActionListener()` method on the `ActionSource`. The method expression wrapped inside the `MethodExpressionActionListener must` identify a public method that accepts an `ActionEvent` (see Section 3.4.2.1 “Event Classes”) instance, and has a return type of `void`. The called method has exactly the same responsibilities as the `processAction()` method of an `ActionListener` instance (see Section 3.4.2.2 “Listener Classes”) that was built in to a separate Java class.
- During the *Apply Request Values* or *Process Validations* phase (depending upon the state of the `immediate` property), components that implement `EditableValueHolder` (such as `UIInput` and its subclasses) components (see Section 3.2.7 “EditableValueHolder”) utilize method expressions as follows:
  - The user can use the `MethodExpressionValidator` class to wrap a method expression that identifies a public method that accepts a `FacesContext` instance and a `UIComponent` instance, and an `Object` containing the value to be validated, and has a return type of `void`. This `MethodExpressionValidator` instance can then be added as a normal `Validator` using the `EditableValueHolder.addValidator()` method. The called method has exactly the same responsibilities as the `validate()` method of a `Validator` instance (see Section 3.5.2 “Validator Classes”) that was built in to a separate Java class.
  - The user can use the `MethodExpressionValueChangeListener` class to wrap a method expression that identifies a public method that accepts a `ValueChangeEvent` (see Section 3.4.2.1 “Event Classes”) instance, and has a return type of `void`. This `MethodExpressionValueChangeListener` instance can then be added

as a normal `ValueChangeListener` using EditableValueHolder.addValueChangeListener(). The called method has exactly the same responsibilities as the `processValueChange()` method of a `ValueChangeListener` instance (see Section 3.4.2.2 “Listener Classes”) that was built in to a separate Java class.

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

TABLE 5-1 component properties whose type is **DEPRECATED** MethodBinding

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

Note that for any of the parameters for the above methods may also be a subclass of what is listed above. For the above properties that are marked as **DEPRECATED,** wrapper classes have been added that wrap a MethodExpression and implement the appropriate listener interface, allowing the wrapped expression to be added as a strongly typed listener, using the normal `add*()` pattern Here is the list of such wrapper classes:

TABLE 5-2 MethodExpression wrappers to take the place of **DEPRECATED** MethodBinding properties

component listener property	Wrapper class	method signature
`actionListener`	`javax.faces.event.MethodExpressionActionListener`	`public void <methodName>(javax.faces.event.ActionEvent);`
`validator`	`javax.faces.validator.MethodExpressionValidator`	`public void <methodName>(javax.faces.context.FacesContext, javax.faces.component.UIComponent, java.lang.Object);`
`valueChangeListener`	`javax.faces.event.MethodExpressionValueChangeListener`	`public void <methodName>(javax.faces.event.ValueChangeEvent);`

The `MethodBinding` typed `action` property of `ActionSource` is deprecated and has been replaced by the `MethodExpression` typed `actionExpression` property of `ActionSource2.`

5.2.1 MethodExpression Syntax and Semantics

The exact syntax and semantics of MethodExpression are now the domain of the Unified EL. Please see Section 1.2.1.2 of the *Expression Language Specification, Version 2.1.*

5.3 The Managed Bean Facility

Perhaps the biggest value-add of bringing EL concepts to Faces happens when the EL is combined with the managed bean facility. This feature allows the user to configure an entire complex tree of POJO beans, including how they should be scoped and populated with initial values, and expose them to EL expressions. Please see *Section 5.3.1 "Managed Bean Configuration Example"*.

The Managed Bean Creation facility is configured by the existence of `<managed-bean>` elements in one or more application configuration resources (see Section 11.4 "Application Configuration Resources"). Note that a special provision has been made for application configuration resource files residing within `META-INF/managed-beans.xml` entries on the application classpath. Please see Section 11.4.4 "Application Configuration Resource Format" for the normative spec requirement. Such elements describe the characteristics of a bean to be created, and properties to be initialized, with the following nested elements:

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

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

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

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

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

- <key-class> -- Optional element specifying the fully qualified class name for keys in the map to be created. If not specified, java.lang.String is used.
- <value-class> -- Optional element specifying the fully qualified class name for values in the map to be created. If not specified, java.lang.String is used.
- <map-entry> -- Zero or more elements that define the actual key-value pairs for a single entry in the map. Nested inside is a <key> element to define the key, and then exactly one of <null-value>, <value> to define the value. These elements have the same meaning as when nested in a <managed-property> element, except that they refer to an individual map entry's value instead of the entire property value.

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

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

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

- Properties are assigned in the order that their <managed-property> elements are listed in the application configuration resource.
- If a managed bean has writeable properties that are not mentioned in <managed-property> elements, the values of those properties are not assigned any values.
- The bean instantiation and population with properties must be done lazily, when an EL expression causes the bean to be referenced. For example, this is the case when a ValueExpression or MethodExpression has its getValue() or setValue() method called.
- Due to the above mentioned laziness constraint, any error conditions that occur below are only required to be manifested at runtime. However, it is conceivable that tools may want to detect these errors earlier; this is perfectly acceptable. The presense of any of the errors described below, until the end of this section, must not prevent the application from deploying and being made available to service requests.
- [P1-start managed bean config error conditions] It is an error to specify a managed bean class that does not exist, or that cannot be instantiated with a public, zero-args constructor.
- It is an error to specify a <property-name> for a property that does not exist, or does not have a public setter method, on the specified managed bean class.
- It is an error to specify a <value> element that cannot be converted to the type required by a managed property, or that, when evaluated, results in a value that cannot be converted to the type required by a managed property. [P1-end]
- If the type of the property referenced by the <managed-property> element is a Java enum, the contents of the <value> element must be a String that yields a valid return from java.lang.Enum.valueOf(PROPERTY_CLASS, VALUE) where PROPERTY_CLASS is the java.lang.Class for the property and VALUE is the contents of the <value> element in the application configuration resource. If any exception is thrown from Enum.valueOf() it is an error.
- [P1-start managed bean scope errors] It is an error for a managed bean created through this facility to have a property that points at an object stored in a scope with a (potentially) shorter life span. Specifically, this means, for an object created with the specified <managed-bean-scope>, then <value> evaluations can only point at created objects with the specified managed bean scope:
  - none -- none
  - application -- none, application
  - session -- none, application, session
  - view -- none, application, session, view
  - request -- none, application, session, view, request [P1-end]
- If a bean points to a property whose value is a mixed expression containing literal strings and expressions, the net scope of the mixed expression is considered to be the scope of the narrowest sub-expression, excluding expressions in the none scope.

- [P1-start implicit objects in request scope] Data accessed via an implicit object is also defined to be in a scope. The following implicit objects are considered to be in request scope:
  - `cookie`
  - `facesContext`
  - `header`
  - `headerValues`
  - `param`
  - `paramValues`
  - `request`
  - `requestScope`
  - `view` [P1-end]
- [P1-start implicit objects in session scope] The only implicit objects in session scope are `session` and `sessionScope` [P1-end]
- [P1-start implicit objects in application scope] The following implicit objects are considered to be in application scope:
  - `application`
  - `applicationScope`
  - `initParam` [P1-end]
- [P1-start cyclic references error] It is an error to configure cyclic references between managed beans. [P1-end]
- [P1-start managed bean names correctness] Managed bean names must conform to the syntax of a Java language identifier. [P1-end]

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

For `<map-entries>`:

1. Call the property getter, if it exists.
2. If the getter returns `null` or doesn't exist, create a `java.util.HashMap`, otherwise use the returned `java.util.Map`.
3. Add all entries defined by nested `<map-entry>` elements in the order they are listed, converting key values defined by nested `<key>` elements to the type defined by `<key-class>` and entry values defined by nested `<value>` elements to the type defined by `<value-class>`. If a value is given as a value expression, evaluate the reference and store the result, converting to `<value-class>` if necessary. If `<key-class>` and/or `<value-class>` are not defined, use `java.lang.String`. Add `null` for each `<null-value>` element.
4. If a new `java.util.Map` was created in step 2), set the property by calling the setter method, or log an error if there is no setter method.

For `<list-entries>`:

1. Call the property getter, if it exists.
2. If the getter returns `null` or doesn't exist, create a `java.util.ArrayList`, otherwise use the returned `Object` (an array or a `java.util.List`).
3. If a `List` was returned or created in step 2), add all elements defined by nested `<value>` elements in the order they are listed, converting values defined by nested `<value>` elements to the type defined by `<value-class>`. If a value is given as a value expression, evaluate the reference and store the result, converting to `<value-class>` if necessary. If a `<value-class>` is not defined, use the value as-is (i.e., as a `java.lang.String`). Add null for each `<null-value>` element.

4. If an array was returned in step 2), create a `java.util.ArrayList` and copy all elements from the returned array to the new `List`, wrapping elements of a primitive type. Add all elements defined by nested `<value>` elements as described in step 3).

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

6. If a new `java.util.List` was created in step 2) and the property is a java array, convert the `List` into an array of the property type, and set it by calling the setter method, or log an error if there is no setter method.

7. If a new `java.util.List` was created in step 4), convert the `List` to an array of the proper type for the property and set the property by calling the setter method, or log an error if there is no setter method.

5.3.1 Managed Bean Configuration Example

The following <managed-bean> elements might appear in one or more application configuration resources (see Section 11.4 “Application Configuration Resources”) to configure the behavior of the Managed Bean Creation facility.

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

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

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

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

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

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

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

5.4 Leveraging Java EE 5 Annotations in Managed Beans

JSF Implementations that are running as a part of Java EE 5 must allow managed bean implementations to use the annotations specified in section 14.5 of the Servlet 2.5 Specification to allow the container to inject references to container managed resources into a managed bean instance before it is made accessible to the JSF application. Only beans declared to be in `request`, `session`, or `application` scope are eligble for resource injection.

Please consult the Java 2 Platform Entprise Edition Specification 5.0 for complete details of this feature. Here is a summary of the valid injection annotations one may use in a managed bean. [P1-start valid annotations in a managed bean]

`@Resource`

`@Resources`

`@EJB`

`@EJBs`

`@WebServiceRef`

`@WebServiceRefs`

`@PersistenceContext`

`@PersistenceContexts`

`@PersistenceUnit`

`@PersistenceUnits` [P1-end]

Following is an example of valid usages of this feature in a managed bean

```
public class User extends Object {
  private @EJB ShoppingCart cart;
  private @Resource Inventory inventory;
  private DataSource customerData;

  @Resource(name=”customerData”)
  private void setCustomerData(DataSource data) {
    customerData = data;
  }

  public String getOrderSummary() {
    // Do something with the injected resources
    // And generate a textual summary of the order
  }

}
```

This example illustrates that the above annotations can be attached to instance variables or to JavaBeans setters. The JSF implementation running in a Java EE 5 container must guarantee that the injections are performed before the bean is handed back to the user. Generally, this is done by performing the injection immediately after the lazy instantiation of the managed bean.

5.4.1 Managed Bean Lifecycle Annotations

JSF implementations running in a Java EE 5 compliant container must support attaching the `@PostConstruct` and `@PreDestroy` annotations to aid in awareness of the managed-bean lifecycle.

Methods on managed beans declared to be in `request`, `view`, `session`, or `application` scope, annotated with `@PostConstruct`, must be called by the JSF implementation after resource injection is performed (if any) but before the bean is placed into scope.

[P1-start rules governing invocation of @PostConstruct annotated methods]If the method throws an unchecked exception, the JSF implementation must not put the managed-bean into service, a message must be logged, and further methods on that managed bean instance must not be called. [P1-end]

Methods on managed beans declared to be in `request`, `session`, or `application` scope, annotated with `@PreDestroy`, must be called by the JSF implementation before the bean is removed from its scope or before the scope itself is destroyed, whichever comes first. In the case of a managed bean placed in `view` scope, methods annotated with `@PreDestroy` must only be called when the view scope is destroyed. See the javadoc for `FacesContext.setViewRoot()`. This annotation must be supported in all cases where the above `@PostConstruct` annotation is supported.

[P1-start rules governing invocation of @PreDestroy annotated methods] If the method throws an unchecked exception, the JSF implementation may log it, but the exception must not otherwise alter the execution.

Refer to the Java EE specification section 2.5 and the Common Annotations for the JavaTM PlatformTM specification section 2.5 for more details.[P1-end]

5.5 How Faces Leverages the Unified EL

This section is non-normative and covers the major players in the Unified EL and how they relate to JavaServer Faces. The number one goal in this version of the JavaServer Faces specification is to export the concepts behind the JSF EL into the Unified EL, which is part of the the JavaServer Pages version 2.1 specification, and then rely on those facilities to get the work done. Readers interested in how to implement the Unified EL itself must consult the Unified EL Spec document.

5.5.1 ELContext

The ELContext is a handy little "holder" object that gets passed all around the Unified EL API. It has two purposes.

- To allow technologies that use the Unified EL, such as JavaServer Faces, the JSF Page Declaration Language (JSF VDL), and JSP, to store any context information specific to that technology so it can be leveraged during expression evaluation. For example the expression "`${view.viewId}`" is specific to Faces. It means, "find the `UIViewRoot` instance for the current view, and return its `viewId`". The Unified EL doesn't know about the "view" implicit object or what a UIViewRoot is, but JavaServer Faces does. The Unified EL has plugin points that will get called to resolve "view", but to do so, JavaServer Faces needs access to the `FacesContext` from within the callstack of EL evaluation. Therefore, the `ELContext` comes to the rescue, having been populated with the `FacesContext` earlier in the request processing lifecycle.
- To allow the pluggable resolver to tell the Unified EL that it did, in fact, resolve a property and that further resolvers must not be consulted. This is done by setting the "`propertyResolved`" property to `true`.

The complete specification for ELResolver may be found in Chapter 2 of the *Expression Language Specification, Version 2.1*.

5.5.1.1 Lifetime, Ownership and Cardinality

An ELContext instance is created the first time `getELContext`() is called on the `FacesContext` for this request. Please see *Section 6.1.3 "ELContext"* for details. Its lifetime ends the same time the `FacesContext's` lifetime ends. The `FacesContext` maintains the owning reference to the `ELContext`. There is at most one `ELContext` per `FacesContext`.

5.5.1.2 Properties

Name	Access	Type	Description
`ELResolver`	RO	`javax.el.ELResolver`	Return the ELResolver instance described in *Section 5.6.1 "Faces ELResolver for JSP Pages"*
`propertyResolved`	RW	boolean	Set by an ELResolver implementation if it successfully resolved a property. See *Section 5.5.2 "ELResolver"* for how this property is used.

5.5.1.3 Methods

Here is a subset of the methods that are relevant to Faces.

```
public Object getContext(Class key);
void putContext(Class key, Object contextInstance);
...
```

As mentioned in *Section 6.1.3 "ELContext"*, the `putContext()` method is called, passing the current `FacesContext` instance the first time the system asks the `FacesContext` for its `ELContext`. The `getContext()` method will be called by any `ELResolver` instances that need to access the `FacesContext` to perform their resolution.

5.5.1.4 Events

The creation of an ELContext instance precipitates the emission of an `ELContextEvent` from the `FacesContext` that created it. Please see *Section 6.1.3 "ELContext"* for details.

5.5.2 ELResolver

Faces 1.1 used the `VariableResolver` and `PropertyResolver` classes as the workhorses of expression evaluation. The Unified API has the `ELResolver` instead. The ELResolver concept is the heart of the Unified EL. When an expression is evaluated, the ELResolver is responsible for resolving each segment in the expression. For example, in rendering the component behind the tag "`<h:outputText value="#{user.address.street}" />`" the ELResolver is called three times. Once to resolve "user", again to resolve the "address" property of user, and finally, to resolve the "street" property of "address". The complete specification for ELResolver may be found in Chapter 2 of the *Expression Language Specification, Version 2.1*.

[N/T-start two ELResolver impls] As described in more detail in *Section 5.6.1 "Faces ELResolver for JSP Pages",* Faces must provide two implementations of `ELResolver`. [P1-end]Which of these two implementations is actually used to resolve an expression depends on where the expresison is evaluated. If the expression is evaluated in a markup page, the ELResolver for markup pages is used. If the expression is evaluated in java VM hosted code from Faces, another ELResolver is used that is tailored for use inside of Faces java VM hosted code. During the course of evaluation of an expression, a variety of sources must be considered to help resolve each segment of the expression. These sources are linked in a chain-like fashion. Each link in the chain has the opportunity to resolve the current segment. If it does so, it must set the "`propertyResolved`" property on the `ELContext`, to `true`. If not, it must not modify the value of the "`propertyResolved`" property. If the "`propertyResolved`" property is not set to `true` the return value from the `ELResolver` method is ignored by the system.

5.5.2.1 Lifetime, Ownership, and Cardinality

ELResolver instances have application lifetime and scope. The JSP container maintains one top level ELResolver (into which a Faces specific ELResolver is added) accessible from `JspContext.getELContext().getELResolver()`. This ELResolver instance is also used from the JSF VDL, even though JSF VDL pages do not themselves use JSP. Faces maintains one `ELResolver` (separate from the one handed to the JSP container) accessible from `FacesContext.getELContext().getELResolver()` and `Application.getELResolver()`.

5.5.2.2 Properties

ELResolver has no proper JavaBeans properties

5.5.2.3 Methods

Here is a subset of the methods that are relevant to Faces.

```
public Object getValue(ELContext context, Object base, Object
property);
void setValue(ELContext context, Object base, Object property,
Object value);
...
```

`getValue()` looks at the argument `base` and tries to return the value of the property named by the argument `property`. For example, if base is a JavaBean, `property` would be the name of the JavaBeans property, and the resolver would end up calling the *getter* for that property.

`setValue()` looks at the argument `base` and tries to set the argument `value` into the property named by the argument `property`. For example, if base is a JavaBean, `property` would be the name of the JavaBeans property, and the resolver would end up calling the *setter* for that property.

There are other methods, such as `isReadOnly()` that are beyond the scope of this document, but described completely in the Unified EL Specification.

5.5.2.4 Events

`ELResolver` precipitates no events.

5.5.3 ExpressionFactory

Faces 1.1 used the Application class as a factory for `ValueBinding` and `MethodBinding` instances. The Unified EL has the `ExpressionFactory` class instead. It is a factory for `ValueExpression` and `MethodExpression` instances.

5.5.3.1 Lifetime, Ownership, and Cardinality

`ExpressionFactory` instances are application scoped. The `Application` object maintains the `ExpressionFactory` instance used by Faces (See *Section 7.1.9 "Acquiring ExpressionFactory Instance")*. The `JspApplicationContext` object maintains the `ExpressionFactory` used by the JSP container (and therefore by the JSF VDL). It is permissible for both of these access methods to yield the same java object instance.

5.5.3.2 Properties

`ExpressionFactory` has no properties.

5.5.3.3 Methods

```
public MethodExpression createMethodExpression(ELContext context,
String expression, FunctionMapper fnMapper, Class[] paramTypes);
public ValueExpression createValueExpression(ELContext context,
String expression, Class expectedType, FunctionMapper fnMapper);
```

These methods take the human readable expression string, such as `"#{user.address.street}"` and return an object oriented representation of the expression. Which method one calls depends on what kind of expression you need. The Faces `Application` class has convenience methods specific to Faces needs for these concepts, please see Section 7.1.10 “Programmatically Evaluating Expressions”.

5.5.3.4 Events

`ExpressionFactory` precipitates no events.

5.6 ELResolver Instances Provided by Faces

This section provides details on what an implementation of the JavaServer Faces specification must do to support the Unified EL for usage in a Faces application.

Section 5.5.2 “ELResolver” mentions that a Faces implementation must provide two implementations of ELResolver. One ELResolver, let’s call it the *`Faces`* *ELResolver For Markup Pages*, is plugged in to the top level resolver chain returned from `JspContext.getELContext().getELResolver()`. This top level resolver chain is used by the page description language container (JSP or JSF Page Declaration Language), and possibly by tag handlers, to resolve expressions. The other `ELResolver`, let’s call it the *ELResolver for Facelets and Programmatic Access*, is used by Facelets markup pages, and is returned from `FacesContext.getELContext().getELResolver()` and `Application.getELResolver()`, and is used to resolve expressions that appear programmatically. See the javadocs for `javax.el.ELResolver` for the specification and method semantics for each method in `ELResolver`. The remainder of this section lists the implementation requirements for these two resolvers.

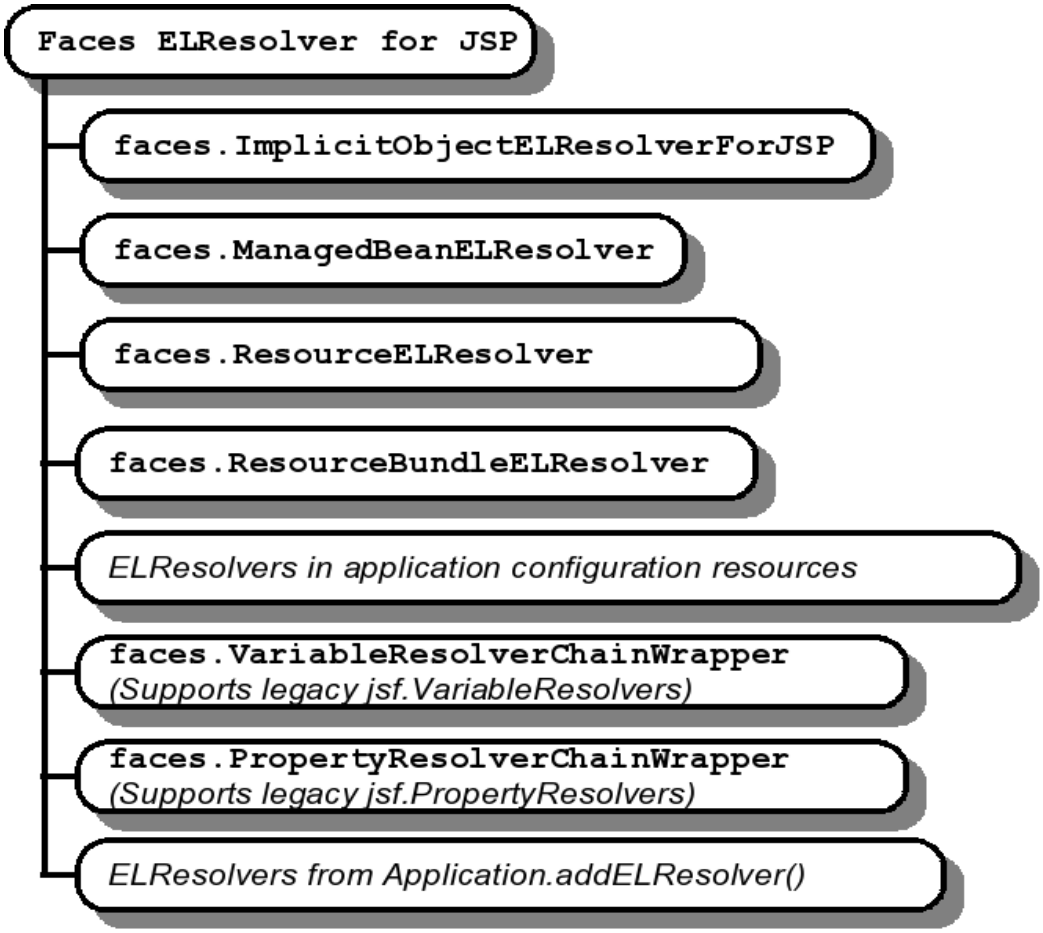
5.6.1 Faces ELResolver for JSP Pages

As mentioned in *Section 5.5.2 “ELResolver”*, during the course of evaluation of an expression, a variety of sources must be considered to help resolve each segment of the expression. These sources are linked in a chain-like fashion. Each link in the chain has the opportunity to resolve the current segment. The Unified EL provides a container class to support this multi-source variable resolution: `javax.el.CompositeELResolver`. The implementation for the *Faces ELResolver for JSP Pages* is described as a set of `ELResolvers` inside of a `CompositeELResolver` instance, but any implementation strategy is permissible as long as the semantics are preserved.

This diagram shows the set of `ELResolver` instances that must be added to the *Faces ELResolver for JSP Pages*. This instance must be handed to the JSP container via a call to `JspFactory.getDefaultFactory().getJspApplicationContext().addELResolver()` at application startup time. Even though we are making a JSP API call to install this `ELResolver`, we do not require using JSP to develop JSF applications. It also shows the order in which they must be added. [P2-start there are 18 methods in the

below tables, each can corresponding to a method on a particular ELResolver. With clever testing, it is possible to write assertions for these. Testing the legacy VariableResolver and PropertyResolvers is not included in this 18 methods number. These classes may be tested simply by noting that the methods do indeed get called on a user-provided VariableResolver or PropertyResolver.] [P1-end]

TABLE 5-3 Faces ELResolver for JSP Pages



The semantics of each ELResolver are given below, either in tables that describe what must be done to implement each particular method on `ELResolver`, or in prose when such a table is inappropriate.

5.6.1.1 Faces Implicit Object ELResolver For JSP

This resolver relies on the presence of another, JSP specific, implicit object ELResolver in the chain by only resolving the “facesContext” and “view” implicit objects.

TABLE 5-4 Faces ImplicitObjectELResolver for JSP

ELResolver method	implementation requirements
getValue	If base is non-null, return null. If base is null and property is null, throw PropertyNotFoundException. If base is null and property is a String equal to "facesContext", call setPropertyResolved(true) on the argument ELContext and return the FacesContext for this request. If base is null and property is a String equal to "view", call setPropertyResolved(true) on the argument ELContext and return the UIViewRoot for this request by calling facesContext.getUIViewRoot(). This ELResolver must also support the implicit object "resuorce" as specified in Section 5.6.2.1 "Implicit Object ELResolver for Facelets and Programmatic Access"
getType	If base is non-null, return null. If base is null and property is null, throw PropertyNotFoundException. If base is null and property is a String equal to "facesContext" or "view", call setPropertyResolved(true) and return null; Otherwise, just return null;This ELResolver must also support the implicit object "resuorce" as specified in Section 5.6.2.1 "Implicit Object ELResolver for Facelets and Programmatic Access"
setValue	If base is null and property is null, throw PropertyNotFoundException. If base is null and property is a String equal to "facesContext" or "view", throw javax.el.PropertyNotWriteable, since "view" and "facesContext" are read-only.This ELResolver must also support the implicit object "resuorce" as specified in Section 5.6.2.1 "Implicit Object ELResolver for Facelets and Programmatic Access"

ELResolver method	implementation requirements
isReadOnly	If base is non-null, return false. If base is null and property is null, throw PropertyNotFoundException. If base is null and property is a String equal to “facesContext” or “view”, call setPropertyResolved(true) on the argument ELContext and return true. Otherwise return false;This ELResolver must also support the implicit object “resuorce” as specified in Section 5.6.2.1 “Implicit Object ELResolver for Facelets and Programmatic Access”
getFeatureDescriptors	If base is non-null, return null. If base is null, return an Iterator containing three java.beans.FeatureDescriptor instances, one for the “view” property, one for the “facesContext” property and one for the “resource” property. It is required that all of the FeatureDescriptor instances in the Iterator set Boolean.TRUE as the value of the ELResolver.RESOLVABLE_AT_DESIGN_TIME attribute. The name and displayName of the FeatureDescriptor must be “view”, “facesContext”, “ or “resource” as appropriate. FacesContext.class, UIViewRoot.class, or ResourceHandler.class must be stored as the value of the ELResolver.TYPE attribute, as approriate. The shortDescription must be a suitable description depending on the implementation. The expert and hidden properties must be false. The preferred property must be true.
getCommonPropertyType	If base is non-null, return null. If base is null and return String.class.

5.6.1.2 ManagedBean ELResolver

This is the means by which the managed bean creation facility described in *Section 5.3 “The Managed Bean Facility”* is called into play during EL resolution.

TABLE 5-5 ManagedBeanELResolver

ELResorver method	implementation requirements
getValue	If base is non-null, return null. If base is null and property is null, throw PropertyNotFoundException. If property matches the name of an entry in the request, session, or application scopes, in that order, return null. If base is null, and property matches one of the managed-bean-name declarations in the application configuration resources, instantiate the bean, populate it with properties as described in *Section 5.3 "The Managed Bean Facility"*, store it in the scope specified by the managed-bean-scope declaration for this this managed-bean, call setPropertyResolved(true) on the argument ELContext, and return the freshly instantiated managed-bean. Otherwise, return null.
getType	If base is null and property is null, throw PropertyNotFoundException. Otherwise return null;
setValue	If base is null and property is null, throw PropertyNotFoundException. Otherwise, if base is null, and property matches one of the managed-bean-name declarations in the application configuration resources, and a managed bean with that managed-bean-name does not yet exist in the specified scope, instantiate the bean, populate it with properties as described in *Section 5.3 "The Managed Bean Facility"*, store it in the scope specified by the managed-bean-scope declaration for this this managed-bean and return. If the managed bean does exist, take no action and return. In either case (the bean exists or does not exist), the actual setting will happen by virtue of the BeanELResolver. Otherwise take no action and return.

ELResorver method	implementation requirements
isReadOnly	If base is non-null, return false. If base is null and property is null, throw PropertyNotFoundException. If base is null return false. We never set the propertyResloved property in this method because the set responsibility is taken care of by the ScopedAttributeELResolver.
getFeatureDescriptors	If base is non-null, return null. If base is null, return an Iterator containing java.beans.FeatureDescriptor instances for each managed-bean in the application-configuration resources. It is required that all of the FeatureDescriptor instances in the Iterator set Boolean.TRUE as the value of the ELResolver.RESOLVABLE_AT_DESIGN_TIME attribute. The name and displayName of the FeatureDescriptor must be the managed-bean-name. The actual java Class instance for the managed-bean-class must be stored as the value of the ELResolver.TYPE attribute. The shortDescription of the FeatureDescriptor must be the description of the managaged-bean element, if present, null otherwise. The expert and hidden properties must be false. The preferred property must be true.
getCommonPropertyType	If base is non-null, return null. If base is null, return Object.class.

5.6.1.3 Resource ELResolver

Please see Section 5.6.2.5 “Resource ELResolver” for the specification of this ELResolver.

5.6.1.4 ResourceBundle ELResolver for JSP Pages

This is the means by which resource bundles defined in the application configuration resources are called into play during EL resolution.

TABLE 5-6 ResourceBundleELResolver

ELResorver method	implementation requirements
getValue	If base is non-null, return null. If base is null and property is null, throw PropertyNotFoundException. If base is null and property is a String equal to the value of the <var> element of one of the <resource-bundle>'s in the application configuration resources, use the Locale of the current UIViewRoot and the base-name of the resource-bundle to load the ResourceBundle. Call setPropertyResolved(true). Return the ResourceBundle. Otherwise, return null.
getType	If base is non-null, return null. If base is null and property is null, throw PropertyNotFoundException. If base is null and property is a String equal to the value of the <var> element of one of the <resource-bundle>'s in the application configuration resources, call setPropertyResolved(true) and return ResourceBundle.class.
setValue	If base is null and property is null, throw PropertyNotFoundException. If base is null and property is a String equal to the value of the <var> element of one of the <resource-bundle>'s in the application configuration resources throw javax.el.PropertyNotWriteable, since ResourceBundles are read-only.

ELResorver method	implementation requirements
isReadOnly	If base is non-null, return null. If base is false and property is null, throw PropertyNotFoundException. If base is null and property is a String equal to the value of the <var> element of one of the <resource-bundle>'s in the application configuration resources, call setPropertyResolved(true) on the argument ELContext and return true. Otherwise return false;
getFeatureDescriptors	If base is non-null, return null. If base is null, return an Iterator containing java.beans.FeatureDescriptor instances, one for each <resource-bundle> in the <application> element. It is required that all of these FeatureDescriptor instances set Boolean.TRUE as the value of the ELResolver.RESOLVABLE_AT_DESIGN_TIME attribute. The name of the FeatureDescriptor must be the var element of the <resource-bundle>. The displayName of the FeatureDescriptor must be the display-name of the <resource-bundle>. ResourceBundle.class must be stored as the value of the ELResolver.TYPE attribute. The shortDescription must be a suitable description depending on the implementation. The expert and hidden properties must be false. The preferred property must be true.
getCommonPropertyType	If base is non-null, return null. If base is null, return string.Class.

5.6.1.5 ELResolvers in the application configuration resources

The `<el-resolver>` element in the application configuration resources will contain the fully qualified classname to a class with a public no-arg constructor that implements `javax.el.ELResolver`. These are added to the *Faces ELResolver for JSP Pages* and the *Faces ELResolver for Facelets and Programmatic Access* in the order in which they occur in the application configuration resources.

5.6.1.6 VariableResolver Chain Wrapper

This is the means by which `VariableResolver` instances that have been specified in `<variable-resolver>` elements inside the application configuration resources are allowed to affect the EL resolution process. If there are one or more `<variable-resolver>` elements in the application configuration resources, an instance of ELResolver with the following semantics must be created and added to the *Faces ELResolver for JSP Pages* as indicated in the *Section TABLE 5-3 "Faces ELResolver for JSP Pages"*.

By virtue of the decorator pattern described in *Section 11.4.6 "Delegating Implementation Support"*, the default `VariableResolver` will be at the end of the `VariableResolver` chain (See *Section 5.8.1 "VariableResolver and the Default VariableResolver"*), if each custom `VariableResolver` chose to honor the full decorator pattern. If the custom `VariableResolver` chose not to honor the decorator pattern, the user is stating that they want to take over complete control of the variable resolution system. Note that the head of the `VariableResolver` chain is no longer accessible by calling `Application.getVariableResolver()` (Please see *Chapter 7 "VariableResolver Property* for what it returns). The head of the `VariableResolver` chain is kept in an implementation specific manner.

The semantics of the ELResolver that functions as the VariableResolver chain wrapper are described in the following table.

TABLE 5-7 ELResolver that is the VariableResolver Chain Wrapper

ELResorver method	implementation requirements
getValue	If base is non-null, return null. If base is null and property is null, throw PropertyNotFoundException. Otherwise, call setPropertyResolved(true) on the argument ELContext. Get the ELContext from the FacesContext. Get the head of the VariableResolver chain and call resolveVariable(facesContext, property) and return the result. Catch any exceptions that may be thrown by resolveVariable(), call setPropertyResolved(false) on the argument ELContext, and rethrow the exception wrapped in an javax.el.ELException.
getType	If base is null and property is null, throw PropertyNotFoundException. return null;
setValue	If base is null and property is null throw PropertyNotFoundException.
isReadOnly	If base is null and property is null throw PropertyNotFoundException. return false;
getFeatureDescriptors	return null;
getCommonPropertyType	If base is null, we return String.class. If base is non-null, return null;

5.6.1.7 PropertyResolver Chain Wrapper

This is the means by which `propertyResolver` instances that have been specified in `<property-resolver>` elements inside the application configuration resources are allowed to affect the EL resolution process. If there are one or more `<property-resolver>` elements in the application configuration resources, an instance of `ELResolver` with the following semantics must be created and added to the *Faces ELResolver for JSP Pages* as indicated in the *Section TABLE 5-3 "Faces ELResolver for JSP Pages"*.

By virtue of the decorator pattern described in *Section 11.4.6 "Delegating Implementation Support"*, the default `propertyResolver` will be at the end of the `propertyResolver` chain (See, *Section 5.8.2 "PropertyResolver and the Default PropertyResolver"*), if each custom `propertyResolver` chose to honor the full decorator pattern. If the custom `propertyResolver` chose not to honor the decorator pattern, then the user is stating that they want to take over complete control of the `propertyResolution` system. Note that the head of the `propertyResolver` chain is no longer accessible by calling `Application.getPropertyResolver()` (Please see *Chapter 7 "PropertyResolver Property* for what it returns). The head of the property resolver chain is kept in an implementation specific manner.

The semantics of the ELResolver that functions as the property resolver chain wrapper are described in the following table.

TABLE 5-8 ELResolver that is the PropertyResolver Chain Wrapper

ELResorver method	implementation requirements

`getValue`, `getType`, `isReadOnly`, `setValue`	`If base or property are null, return null (or false if the method returns boolean).`
	`Call setPropertyResolved(true) on the argument ELContext.`
	`Get the ELContext from the FacesContext.`
	`Get the head of the propertyResolver chain.`
	`If base is a List or java language array, coerce the property to an int and call the corresponding method on the head of the property resolver chain that takes an int for property, returning the result (except in the case of setValue()).`
	`Otherwise, call the corresponding method on the head of the property resolver chain that takes an Object for property, returning the result (except in the case of setValue()).`
	`If an Exception is thrown by calling the above methods on the PropertyResolver chain, catch it, call setPropertyResolved(false) on the argument ELContext, and rethrow the Exception wrapped (snuggly) in a javax.el.ELException.`

`getFeatureDescriptors`	`return null;`
`getCommonPropertyType`	`If base is null, return null. If base is non-null, return Object.class.`

5.6.1.8 ELResolvers from Application.addELResolver()

Any such resolvers are considered at this point in the *Faces ELResolver for JSP Pages* in the order in which they were added.

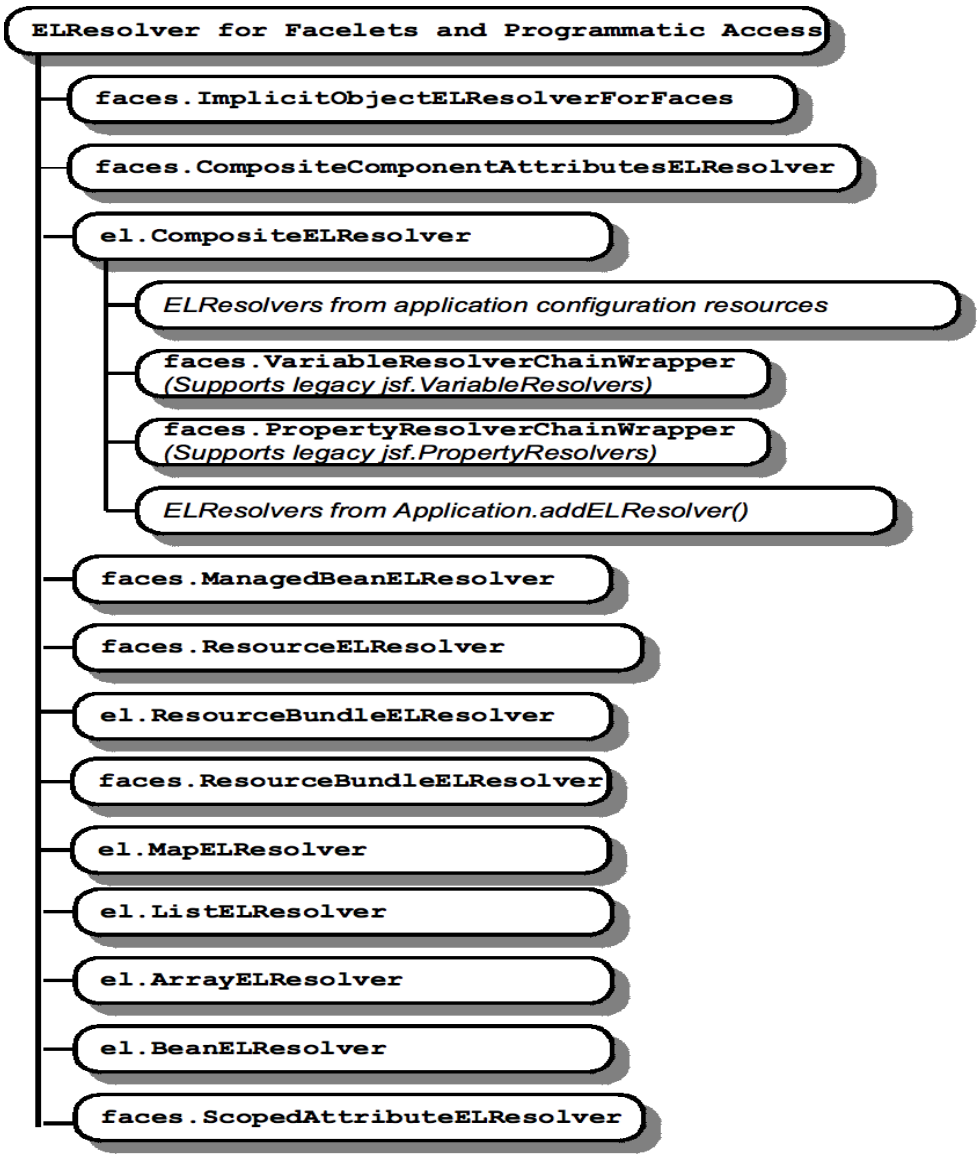
5.6.2 ELResolver for Facelets and Programmatic Access

This section documents the requirements for the second `ELResolver` mentioned in *Section 5.6 "ELResolver Instances Provided by Faces"*, the one that is used for Facelets and for programmatic expression evaluation from Faces java code.

The implementation for the *ELResolver for Programmatic Access* is described as a set of `ELResolvers` inside of a `CompositeELResolver` instance, but any implementation strategy is permissible as long as the semantics are preserved. .

This diagram shows the set of `ELResolver` instances that must be added to the *ELResolver for Programmatic Access*. This instance must be returned from `Application.getELResolver()` and `FacesContext.getELContext().getELResolver()`. It also shows the order in which they must be added. [P1-start there are 12 methods in the below tables that can be tested for assertion. The remainder of the section is covered by the tests in 5.6.1][P1-end]

TABLE 5-9 `ELResolver` for Facelets and Programmatic Access



The semantics of each `ELResolver` are given below, either in tables that describe what must be done to implement each particular method on `ELResolver`, in prose when such a table is inappropriate, or as a reference to another section where the semantics are exactly the same.

5.6.2.1 Implicit Object ELResolver for Facelets and Programmatic Access

This resolver differs from the one in the Section 5.6.1.1 “Faces Implicit Object ELResolver For JSP” in that it must resolve all of the implicit objects, not just `facesContext` and `view`.

TABLE 5-10 ImplicitObjectELResolver for Programmatic Access

ELResolver method	implementation requirements
getValue	If base is non-null, return null. If base is null and property is null, throw PropertyNotFoundException. If base is null and property is a String equal to *implicitObject*, call setPropertyResolved(true) on the argument ELContext and return *result*, where *implicitObject* and *result* are as follows: *implicitObject -> result* application -> externalContext.getContext() applicationScope -> externalContext.getApplicationMap() component -> the component most recently pushed to UIComponent.pushComponentToEL() compositeComponent -> the component returned from UIComponent.getCurrentCompositeComponent(). cookie -> externalContext.getRequestCookieMap() facesContext -> the FacesContext for this request component -> the top of the stack of UIComponent instances, as pushed via calls to UIComponent.pushComponentToEL(). See Section 3.1.14 “Lifecycle Management Methods” header -> externalContext.getRequestHeaderMap() headerValues -> externalContext.getRequestHeaderValuesMap() initParam -> externalContext.getInitParameterMap() param -> externalContext.getRequestParameterMap() paramValues -> externalContext.getRequestParameterValuesMap() request -> externalContext.getRequest() requestScope -> externalContext.getRequestMap() resource -> facesContext.getApplication().getResourceHandler() session -> externalContext.getSession() sessionScope -> externalContext.getSessionMap() view -> facesContext.getViewRoot() viewScope -> facesContext.getViewRoot().getViewMap() resource -> facesContext.getApplication().getResourceHandler() If base is null, and property doesn’t match one of the above *implicitObjects,* return null.

ELResolver method	implementation requirements
getType	If base is non-null, return null. If base is null and property is null, throw PropertyNotFoundException. If base is null and property is a String equal to “application”, “component”, “compositeComponent”, “cookie”, “facesContext”, “header”, “headerValues”, “initParam”, “param”, “paramValues”, “request”, “resource”, “session”, or “view”, call setPropertyResolved(true) on the argument ELContext and return null to indicate that no types are accepted to setValue() for these attributes. If base is null and property is a String equal to “requestScope”, “sessionScope”, or “applicationScope”, call setPropertyResolved(true) on the argument ELContext and return null. Otherwise, null;
setValue	If base is non-null, return null. If base is null and property is null, throw PropertyNotFoundException. If base is null and property is a String equal to “applicationScope”, “requestScope”, “sessionScope”, “application”, “component”, “compositeComponent”, “cookie”, “facesContext”, “header”, “headerValues”, “initParam”, “param”, “paramValues”, “request”, “resource”, “session”, or “view”, throw javax.el.PropertyNotWriteableException, since these implicit objects are read-only. Otherwise return null.
isReadOnly	If base is non-null, return (or false if the method returns boolean). If base is null and property is null, throw PropertyNotFoundException. If base is null and property is a String equal to “applicationScope”, “component”, “compositeComponent”, “requestScope”, “sessionScope”, “application”, “cookie”, “facesContext”, “header”, “headerValues”, “initParam”, “param”, “paramValues”, “request”, “resource”, “session”, or “view”, call setPropertyResolved(true) on the argument ELContext and return true. Otherwise return null.

ELResolver method	implementation requirements
getFeatureDescriptors	If base is non-null, return null. If base is null, return an Iterator containing 17 java.beans.FeatureDescriptor instances, one for eath of the following properties: application, component, compositeComponent, cookie, facesContext, header, headerValues, initParam, param, paramValues, request, resource, session, view, applicationScope, sessionScope, and requestScope. It is required that all of these FeatureDescriptor instances set Boolean.TRUE as the value of the ELResolver.RESOLVABLE_AT_DESIGN_TIME attribute. For the name and short of FeatureDescriptor, return the implicit object name. The appropriate Class must be stored as the value of the ELResolver.TYPE attribute as follows: implicitObject -> ELResolver.TYPE value application -> Object.class applicationScope -> Map.class component -> UIComponent.class compositeComponent -> UIComponent.class cookie -> Map.class facesContext -> FacesContext.class header -> Map.class headerValues -> Map.class initParam -> Map.class param -> Map.class paramValues -> Map.class request -> Object.class resource -> Object.class requestScope -> Map.class session -> Object.class sessionScope -> Map.class view -> UIViewRoot.class The shortDescription must be a suitable description depending on the implementation. The expert and hidden properties must be false. The preferred property must be true.
getCommonPropertyType	If base is non-null, return null. If base is null and return String.class

5.6.2.2 Composite Component Attributes ELResolver

This ELResolver makes it so expressions that refer to the attributes of a composite component get correctly evaluated. For example, the expression `#{compositeComponent.attrs.usernameLabel}` says, “find the current composite component, call its `getAttributes()` method, within the returned `Map` look up the value under the key “usernameLable”. If the value is a `ValueExpression`, call `getValue()` on it and the result is returned as the evaluation of the expression. Otherwise, if the value is *not* a `ValueExpression` the value itself is returned as the evaluation of the expression.”

TABLE 5-11 Composite Component Attributes ELResolver

ELResolver method	implementation requirements
getValue	If base is non-null, is an instance of UIComponent, is a composite component, and property is non-null and is equal to the string “attrs”, return a Map implementation with the following characteristics. Wrap the attributes map of the composite component and delegate all calls to the composite component attributes map with the following exceptions: get(): if the result of calling get() on the composite component attributes map is a ValueExpression, call getValue() on it and return the result. put(): call get() on the attributes map, using the argument key to put() as the argument to get(). If the result is a ValueExpression, call setValue() on the ValueExpression, passing the value argument to put() as the second argument to setValue(). Otherwise, take no action.
getType	return null.
setValue	Take no action.
isReadOnly	Take no action and return true.
getFeatureDescriptors	Take no action.
getCommonPropertyType	Return String.class

5.6.2.3 The CompositeELResolver

As indicated in *Section TABLE 5-9 “ELResolver for Facelets and Programmatic Access”*, following the ImplicitObjectELResolver, the semantics obtained by adding a `CompositeELResolver` must be inserted here. This `ELResolver` contains the following `ELResolvers`, described in the referenced sections.

1. *Section 5.6.1.5 “ELResolvers in the application configuration resources”*
2. *Section 5.6.1.6 “VariableResolver Chain Wrapper”*
3. *Section 5.6.1.7 “PropertyResolver Chain Wrapper”*
4. *Section 5.6.1.8 “ELResolvers from Application.addELResolver()”*

5.6.2.4 ManagedBean ELResolver

This resolver has the same semantics as the one in *Section 5.6.1.2 “ManagedBean ELResolver”*.

5.6.2.5 Resource ELResolver

This resolver is a means by which `Resource` instances are encoded into a faces request such that a subsequent faces resource request from the browser can be satisfied using the `ResourceHandler` as described in *Section 2.6 “Resource Handling”*.

TABLE 5-12 ResourceELResolver

ELResorver method	implementation requirements
`getValue`	`If base and property are not null, and base is an instance of ResourceHandler (as will be the case with an expression such as #{resource['ajax.js']}, perform the following. (Note: This is possible due to the ImplicitObjectELResolver returning the ResourceHandler, see Section 5.6.2.1 "Implicit Object ELResolver for Facelets and Programmatic Access")` ■ If `property` does not contain a colon character ':', treat `property` as the *resourceName* and pass `property` to `ResourceHandler.createResource(` *`resourceName`*`)`. ■ If `property` contains a single colon character ':', treat the content before the ':' as the *libraryName* and the content after the ':' as the *resourceName* and pass both to `ResourceHandler.createResource(` *`resourceName, libraryName)`* ■ If `property` contains more than one colon character ':', throw a localized `ELException`, including `property`. `If one of the above steps results in the creation of a non-null Resource instance, call ELContext.setPropertyResolved(true) and return the result of calling the getRequestPath() method on the Resource instance.`
`getType`	`Return null. This resolver only performs lookups.`
`setValue`	`Take no action.`
`isReadOnly`	`Return false in all cases.`
`getFeatureDescriptors`	`Return null.`
`getCommonPropertyType`	`If base is non-null, return null.` `If base is null, return Object.class.`

5.6.2.6 el.ResourceBundleELResolver

This entry in the chain must have the semantics the same as the class `javax.el.ResourceBundleELResolver`. The default implementation just includes an instance of this resolver in the chain.

5.6.2.7 ResourceBundle ELResolver for Programmatic Access

This resolver has the same semantics as the one in *Section 5.6.1.4 "ResourceBundle ELResolver for JSP Pages"*.

5.6.2.8 Map, List, Array, and Bean ELResolvers

These ELResolver instances are provided by the Unified EL API and must be added in the following order: `javax.el.MapELResolver, javax.el.ListELResolver, javax.el.ArrayELResolver, javax.el.BeanELResolver`. These actual ELResolver instances must be added. It is not compliant to simply add other resolvers that preserve these semantics.

5.6.2.9 ScopedAttribute ELResolver

This ELResolver is responsible for doing the scoped lookup that makes it possible for expressions to pick up anything stored in the request, session, or application scopes by name.

TABLE 5-13 Scoped Attribute ELResolver

ELResorver method	implementation requirements
getValue	If base is non-null, return null. If base is null and property is null, throw PropertyNotFoundException. Use the argument property as the key in a call to externalContext.getRequestMap().get(). If this returns non-null, call setPropertyResolved(true) on the argument ELContext and return the value. Use the argument property as the key in a call to externalContext.getSessionMap().get(). If this returns non-null, call setPropertyResolved(true) on the argument ELContext and return the value. Use the argument property as the key in a call to externalContext.getApplicationMap().get(). If this returns non-null, call setPropertyResolved(true) on the argument ELContext and return the value. Otherwise call setPropertyResloved(true) and return null;
getType	If base is non-null, return null. If base is null and property is null, throw PropertyNotFoundException. Otherwise, setPropertyResolved(true) and return Object.class to indicate that any type is permissable to pass to a call to setValue().
setValue	If base is non-null, return null. If base is null and property is null, throw PropertyNotFoundException. Consult the Maps for the request, session, and application, in order, looking for an entry under the key property. If found, replace that entry with argument value. If not found, call externalContext.getRequestMap().put(property, value). Call setPropertyResolved(true) and return;

ELResorver method	implementation requirements
isReadOnly	If base is false, setPropertyResolved(true) return false; Otherwise, return false;
getFeatureDescriptors	If base is non-null, return null. If base is null, return an Iterator of java.beans.FeatureDescriptor instances for all attributes in all scopes. The FeatureDescriptor name and shortName is the name of the scoped attribute. The actual runtime type of the attribute must be stored as the value of the ELResolver.TYPE attribute. Boolean.TRUE must be set as the value of the ELResolver.RESOLVABLE_AT_DESIGN_TIME attribute. The shortDescription must be a suitable description depending on the implementation. The expert and hidden properties must be false. The preferred property must be true.
getCommonPropertyType	If base is non-null, return null. If base is null return String.class.

5.7 Current Expression Evaluation APIs

5.7.1 ELResolver

This class is the Unified EL's answer to Faces's `VariableResolver` and `PropertyResolver`. It turns out that variable resolution can be seen as a special case of property resolution with the base object being `null`. Please see *Section 5.5.2 "ELResolver"* for more details.

5.7.2 ValueExpression

This class is the Unified EL's answer to Faces's `ValueBinding`. It is the main object oriented abstraction for al EL expression that results in a value either being retrieved or set. Please see Chapter 2 of the *Expression Language Specification, Version 2.1.*

5.7.3 MethodExpression

This class is the Unified EL's answer to Faces's `MethodBinding`. It is the main object oriented abstraction for al EL expression that results in a method being invoked. Please see Chapter 2 of the *Expression Language Specification, Version 2.1.*

5.7.4 Expression Evaluation Exceptions

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

- `javax.el.ELException` (which extends `java.lang.Exception`)—used to report a problem evaluating a value exception dynamically.
- `MethodNotFoundException` (which extends `javax.el.ELException`)—used to report that a requested public method does not exist in the context of evaluation of a method expression.
- `javax.el.PropertyNotFoundException` (which extends `javax.el.ELException`)—used to report that a requested property does not exist in the context of evaluation of a value expression.
- `javax.el.PropertyNotWriteableException` (which extends `javax.el.ELException`)—used to indicate that the requested property could not be written to when evaluating the expression.

5.8 Deprecated Expression Evaluation APIs

Applications written for version 1.0 and 1.1 of the Faces specification must continue to run in this version of the specification. This means deprecated APIs. This section describes the migration story for these APIs that implementations must follow to allow 1.0 and 1.1 based applications to run.

5.8.1 VariableResolver and the Default VariableResolver

User-provided VariableResolver instances will still continue to work by virtue of *Section 5.6.1.6 "VariableResolver Chain Wrapper"*. The decorator pattern described in *Section 11.4.6 "Delegating Implementation Support"* must be supported. Users wishing to affect EL resolution are advised to author a custom ELResolver instead. These will get picked up as specified in *Section 5.6.1.5 "ELResolvers in the application configuration resources"*.

The JSF implementation must provide a default `VariableResolver` implementation that gets the `ELContext` from the argument `FacesContext` and calls `setPropertyResolved(false)` on it

The `VariableResolver` chain is no longer accessible from `Application.getVariableResolver()`. The chain must be kept in an implementation dependent manner, but accessible to the ELResolver described in *Section 5.6.1.6 "VariableResolver Chain Wrapper"*.

5.8.2 PropertyResolver and the Default PropertyResolver

User-provided propertyResolver instances will still continue to work by virtue of *Section 5.6.1.6 "VariableResolver Chain Wrapper"*. The decorator pattern described in *Section 11.4.6 "Delegating Implementation Support"* must be supported. Users wishing to affect EL resolution are advised to author a custom ELResolver instead. These will get picked up as specified in *Section 5.6.1.5 "ELResolvers in the application configuration resources"*.

The JSF implementation must provide a default `propertyResolver` implementation that gets the `ELContext` from the argument `FacesContext` and calls `setPropertyResolved(false)` on it.

The `PropertyResolver` chain is no longer accessible from `Application.getpropertyResolver()`. The chain must be kept in an implementation dependent manner, but accessible to to the ELResolver described in *Section 5.6.1.7 "PropertyResolver Chain Wrapper"*.

5.8.3 ValueBinding

The `ValueBinding` class encapsulates the actual evaluation of a value binding. Instances of `ValueBinding` for specific references are acquired from the `Application` instance by calling the `createValueBinding` method (see Section 7.9.3 “Acquiring ValueBinding Instances”).

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

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

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

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

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

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

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

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

5.8.4 MethodBinding

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

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

Evaluate the method binding (see Section 5.2.1 “MethodExpression Syntax and Semantics”) and call the identified method, passing the specified parameters. Return any value returned by the invoked method, or return `null` if the invoked method is of type `void`.

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

Evaluate the method binding (see Section 5.2.1 “MethodExpression Syntax and Semantics”) and return the `Class` representing the return type of the identified method. If this method is of type `void`, return `null` instead.

5.8.5 Expression Evaluation Exceptions

Four exception classes are defined to report errors related to the evaluation of value exceptions [Note that these exceptions are deprecated]:

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

6

Per-Request State Information

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

6.1 FacesContext

JSF defines the `javax.faces.context.FacesContext` abstract base class for representing all of the contextual information associated with processing an incoming request, and creating the corresponding response. A `FacesContext` instance is created by the JSF implementation, prior to beginning the request processing lifecycle, by a call to the `getFacesContext` method of `FacesContextFactory`, as described in Section 6.6 "FacesContextFactory". When the request processing lifecycle has been completed, the JSF implementation will call the `release` method, which gives JSF implementations the opportunity to release any acquired resources, as well as to pool and recycle `FacesContext` instances rather than creating new ones for each request.

6.1.1 Application

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

[P1-start-application]The JSF implementation must ensure that the `Application` instance for the current web application is available via this method, as a convenient alternative to lookup via an `ApplicationFactory`.[P1-end]

6.1.2 Attributes

```
public Map<Object,Object> getAttributes();
```

[P1-start-attributes]Return a mutable `Map` representing the attributes associated wth this `FacesContext` instance. This `Map` is useful to store attributes that you want to go out of scope when the Faces lifecycle for the current request ends, which is not always the same as the request ending, especially in the case of `Servlet` filters that are invoked after the Faces lifecycle for this request completes. Accessing this `Map` does not cause any events to fire, as is the case with the other maps: for request, session, and application scope.[P1-end]

6.1.3 ELContext

```
public ELContext getELContext();
```

Return the `ELContext` instance for this `FacesContext` instance. This `ELContext` instance has the same lifetime and scope as the `FacesContext` instance with which it is associated, and may be created lazily the first time this method is called for a given `FacesContext` instance. [P1-start-elcontext]Upon creation of the ELContext instance, the implementation must take the following action:

- Call the `ELContext.putContext(java.lang.Class, java.lang.Object)` method on the instance, passing in `FacesContext.class` and the `this` reference for the `FacesContext` instance itself.
- If the `Collection` returned by `javax.faces.Application.getELContextListeners()` is non-empty, create an instance of `ELContextEvent` and pass it to each `ELContextListener` instance in the `Collection` by calling the `ELContextListener.contextCreated(javax.el.ELContextEvent)` method.[P1-end]

6.1.4 ExternalContext

It is sometimes necessary to interact with APIs provided by the containing environment in which the JavaServer Faces application is running. In most cases this is the servlet API, but it is also possible for a JavaServer Faces application to run inside of a portlet. JavaServer Faces provides the `ExternalContext` abstract class for this purpose. [P1-start-externalContext]This class must be implemented along with the `FacesContext` class, and must be accessible via the `getExternalContext` method in `FacesContext`.[P1-end]

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

[P1-start externalContext during Init] The default implementation must return a valid value when this method is called during startup time. See the javadocs for this method for the complete specification. [P1-end]

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

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

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

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

See the JavaDocs for the normative specification.

6.1.5 ViewRoot

```
public UIViewRoot getViewRoot();

public void setViewRoot(UIViewRoot root);
```

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

6.1.6 Message Queue

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

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

```
public Interator<String> getClientIdsWithMessages();

public Severity getMaximumSeverity();

public Iterator<FacesMessage> getMessages(String clientId);

public Iterator<FacesMessage> getMessages();
```

[P1-start-messageQueue]The `getClientIdsWithMessages()` method must return an `Iterator` over the client identifiers for which at least one `Message` has been queued. This method must be implemented so the clientIds are returned in the order of calls to `addMessage()`.[P1-end] The `getMaximumSeverity()` method returns the highest severity level on any `Message` that has been queued, regardless of whether or not the message is associated with a specific client identifier or not. The `getMessages(String)` method returns an `Iterator` over queued `Message`s, either those associated with the specified client identifier, or those associated with no client identifier if the parameter is `null`. The `getMessages()` method returns an `Iterator` over all queued `Messages`, whether or not they are associated with a particular client identifier. Both of the `getMessage()` variants must be implemented such that the messages are returned in the order in which they were added via calls to `addMessage()`.

For more information about the `Message` class, see Section 6.3 “FacesMessage”.

6.1.7 RenderKit

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

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

6.1.8 ResponseStream and ResponseWriter

```
public ResponseStream getResponseStream();

public void setResponseStream(ResponseStream responseStream);

public ResponseWriter getResponseWriter();

public void setResponseWriter(ResponseWriter responseWriter);

public void enableResponseWriting(boolean enable);
```

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

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

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

The `enableResponseWriting` method is useful to enable or disable the writing of content to the current `ResponseWriter` instance in this `FacesContext`. [P1-start-enableWriting]If the `enable` argument is `false`, content should not be written to the response if an attempt is made to use the current `ResponseWriter`.

6.1.9 Flow Control Methods

```
public void renderResponse();

public void responseComplete();

public boolean getRenderResponse();

public boolean getResponseComplete();
```

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

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

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

In some circumstances, it is possible that both `renderResponse()` and `responseComplete()` might have been called for the request. [P1-start-flowControl]In this case, the JSF implementation must respect the `responseComplete()` call (if it was made) before checking to see if `renderResponse()` was called.[P1-end]

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

6.1.10 Partial Processing Methods

```
public PartialViewContext getPartialViewContext();
```

[P1-start-getpartialViewContext]The `getPartialViewContext()`method must return an instance of `PartialViewContext` either by creating a new instance, or returning an existing instance from the `FacesContext`.[P1-end-getpartialViewcontext]

6.1.11 Partial View Context

The PartialViewContext contains the constants, properties and methods to facilitate partial view processing and partial view rendering. Refer to Section 13.4.2 “Partial View Processing” and Section 13.4.3 “Partial View Rendering”. Refer to the JavaDocs for the javax.faces.context.PartialViewContext class for method requirements.

6.1.12 Access To The Current FacesContext Instance

```
public static FacesContext getCurrentInstance();

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

Under most circumstances, JSF components, and application objects that access them, are passed a reference to the `FacesContext` instance for the current request. However, in some cases, no such reference is available. The `getCurrentInstance()` method may be called by any Java class in the current web application to retrieve an instance of the `FacesContext` for this request. [P1-start-currentInstance]The JSF implementation must ensure that this value is set correctly before `FacesContextFactory` returns a `FacesContext` instance, and that the value is maintained in a thread-safe manner.[P1-end]

[P1-start facesContextDuringInit] The default implementation must allow this method to be called during application startup time, before any requests have been serviced. If called during application startup time, the instance returned must have the special properties as specified on the javadocs for `FacesContext.getCurrentInstance()`The . [P1-end]

6.1.13 CurrentPhaseId

The default lifecycle implementation is responsible for setting the `currentPhaseId` property on the `FacesContext` instance for this request, as specified in Section 2.2 “Standard Request Processing Lifecycle Phases”. The following table describes this property.

Name	Access	Type	Description
`currentPhaseId`	RW	`PhaseId`	The `PhaseId` constant for the current phase of the request processing lifecycle

6.1.14 ExceptionHandler

The `FacesContextFactory` ensures that each newly created `FacesContext` instance is initialized with a fresh instance of `ExceptionHandler`, created from `ExceptionHandlerFactory`.The following table describes this property.

Name	Access	Type	Description
`exceptionHandler`	RW	`ExceptionHandler`	Set by `FacesContextFactory.getFacesContext()`, this class is the default exception handler for any unexpected Exceptions that happen during the Faces lifecycle. See the Javadocs for `ExceptionHandler` for details.

Please see Section 12.3 “PhaseListener” for the circumstances under which `ExceptionHandler` is used.

6.1.15 Flash

The `Flash` provides a way to pass temporary objects between the user views generated by the faces lifecycle. Anything one places in the flash will be exposed to the next view encountered by the same user session and then cleared out..

Name	Access	Type	Description
`flash`	R	`Flash`	See the javadocs for the complete specification.

6.2 ExceptionHandler

`ExceptionHandler` is the central point for handling *unexpected* `Exceptions` that are thrown during the Faces lifecycle. The `ExceptionHandler` must *not* be notified of any `Exceptions` that occur during application startup or shutdown.

Several places in the Faces specification require an `Exception` to be thrown as a result of normal lifecycle processing. [P1-start_expected_exceptions]The following expected `Exception` cases **must not** be handled by the ExceptionHandler.

- All cases where a `ValidatorException` is specified to be thrown or caught
- All cases where a `ConverterException` is specified to be thrown or caught
- The case when a MissingResourceException is thrown during the processing of the `<f:loadBundle />` tag.
- If an exception is thrown when the runtime is processing the `@PostConstruct` or `@PreDestroy` annotation on a managed bean.

All other `Exception` cases must not be swallowed, and must be allowed to flow up to the `Lifecycle.execute`() method where the individual lifecycle phases are implemented. [P1-end_expected_exceptions] At that point, all `Exceptions` are passed to the `ExceptionHandler` as described in Section 12.3 “PhaseListener”.

Any code that is not a part of the core Faces implementation may leverage the `ExceptionHandler` in one of two ways.

6.2.1 Default ExceptionHandler implementation

The default ExceptionHandler must implement the following behavior for each of its methods

```
public ExceptionEvent getHandledExceptionEvent();
```

Return the first “handled” `ExceptionEvent`, that is, the one that was actually re-thrown.

```
public Iterable<ExceptionEvent> getHandledExceptionEvents();
```

The default implementation must return an `Iterable` over all `ExceptionEvents` that have been handled by the `handle()` method.

```
public Throwable getRootCause(Throwable t);
```

Unwrap the argument `t` until the unwrapping encounters an `Object` whose `getClass()` is not equal to `FacesException.class` or `javax.el.ELException.class`. If there is no root cause, `null` is returned.

```
public Iterable<ExceptionEvent> getUnhandledExceptionEvents();
```

Return an `Iterable` over all `ExceptionEvents` that have not yet been handled by the `handle()` method.

```
public void handle() throws FacesException;
```

Inspect all unhandled `ExceptionEvent` instances in the order in which they were queued by calls to `Application.publishEvent(ExceptionEvent.class, eventContext)`.

For each `ExceptionEvent` in the list, call its `getContext()` method and call `getException()` on the returned result. Upon encountering the first such `Exception` that is **not** an instance of `javax.faces.event.AbortProcessingException`, the corresponding `ExceptionEvent` must be set so that a subsequent call to `getHandledExceptionEvent()` or `getHandledExceptionEvents()` returns that `ExceptionEvent` instance. The implementation must also ensure that subsequent calls to `getUnhandledExceptionEvents()` do **not** include that `ExceptionEvent` instance. Let *toRethrow* be either the result of calling `getRootCause()` on the `Exception`, or the `Exception` itself, whichever is non-`null`. Re-wrap *toThrow* in a `ServletException` or (`PortletException`, if in a portlet environment) and throw it, allowing it to be handled by any `<error-page>` declared in the web application deployment descriptor or by the default error page as described elsewhere in this section.

There are two exceptions to the above processing rules. In both cases, the `Exception` must be logged and **not** re-thrown.

- If an unchecked `Exception` occurs as a result of calling a method annotated with `PreDestroy` on a managed bean.
- If the `Exception` originates inside the `ELContextListener.removeElContextListener()` method

The `FacesException` must be thrown if and only if a problem occurs while performing the algorithm to handle the `Exception`, not as a means of conveying a handled Exception itself.

```
public boolean isListenerForSource(Object source);
```

The default implementation must return `true` if and only if the source argument is an instance of `ExceptionEventContext`.

```
public void processEvent(SystemEvent exceptionEvent) throws
AbortProcessingException;
```

The default implementation must store the argument `exceptionEvent` in a strongly ordered queue for later processing by the `handle()` method.

6.2.2 Backwards Compatible ExceptionHandler

[P1-startPreJsf2ExceptionHandler]The runtime must provide an `ExceptionHandlerFactory` implementation with the fully qualified java classname of `javax.faces.webapp.PreJsf2ExceptionHandlerFactory` that creates `ExceptionHandler` instances that behave exactly like the default `ExceptionHandler` except that the `handle()` method behaves as follows.

> Versions of JSF prior to 2.0 stated in Section 12.3 “PhaseListener” “Any exceptions thrown during the `beforePhase()` listeners must be caught, logged, and swallowed...Any exceptions thrown during the `afterPhase()` liseteners must be caught, logged, and swallowed.” The `PreJsf2ExceptionHandler` restores this behavior for backwards compatibilty.

The implementation must allow users to install this `ExceptionHandlerFactory` into the application by nesting `<exception-handler-factory>javax.faces.webapp.PreJsf2ExceptionHandlerFactory</exception-handler-factory>` inside the <factory> element in the application configuration resource.[P1-endPreJsf2ExceptionHandler]

6.2.3 Default Error Page

If no <error-page> elements are declared in the web application deployment descriptor, the runtime must provide a default error page that contains the following information.

- The stack trace of the Exception
- The UIComponent tree at the time the ExceptionEvent was handled.
- All scoped variables in request, view, session and application scope.
- If the error happens during the execution of the page declaration language page (VDL)
  - The physical file being traversed at the time the Exception was thrown, such as /user.xhtml
  - The line number within that physical file at the time the Exception was thrown
  - Any available error message(s) from the VDL page, such as: “The prefix "foz" for element "foz:bear" is not bound.”
- The viewId at the time the ExceptionEvent was handled

If Application.getProjectStage() returns ProjectStage.Development, the runtime must guarantee that the above debug information is available to be included in any Facelet based error page using the <ui:include /> with a src attribute equal to the string “javax.faces.error.xhtml”.

6.3 FacesMessage

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

```
public FacesMessage();

public FacesMessage(String summary, String detail);

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

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

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

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

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

The message properties are defined as follows:

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

- summary—Localized summary text for this FacesMessage. This is normally a relatively short message that concisely describes the nature of the problem being reported by this FacesMessage.

6.4 ResponseStream

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

6.5 ResponseWriter

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

```
public void close() throws IOException;

public void flush() throws IOException;

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

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

public void write(int c) throws IOException;

public void write(String s) throws IOException;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.6 FacesContextFactory

[P1-start-facesContextFactory]A single instance of `javax.faces.context.FacesContextFactory` must be made available to each JSF-based web application running in a servlet or portlet container.[P1-end] 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);
```

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

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

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

6.7 ExceptionHandlerFactory

[P1-start-exceptionHandlerFactory]A single instance of `javax.faces.context.ExceptionHandlerFactory` must be made available to each JSF-based web application running in a servlet or portlet container.[P1-end] The factory instance can be acquired, by JSF implementations or by application code, by executing:

```
ExceptionHandlerFactory factory =
(ExceptionHandlerFactory)
FactoryFinder.getFactory(Factory-
Finder.EXCEPTION_HANDLER_FACTORY);
```

The `ExceptionHandlerFactory` implementation class provides the following method signature to create an `ExceptionHandler` instance:

```
public ExceptionHandler getExceptionHandler(FacesContext
currentContext);
```

Create and return a `ExceptionHandler` instance that has been configured based on the specified parameters.

6.8 ExternalContextFactory

[P1-start-externalContextFactory]A single instance of `javax.faces.context.ExternalContextFactory` must be made available to each JSF-based web application running in a servlet or portlet container.[P1-end] 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:

```
ExternalContextFactory factory =
(ExternalContextFactory)
FactoryFinder.getFactory(Factory-
Finder.EXTERNAL_CONTEXT_FACTORY);
```

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

```
public ExternalContext getExternalContext(Object context, Object
request, Object response);
```

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

7

Application Integration

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

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

7.1 Application

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

7.1.1 ActionListener Property

```
public ActionListener getActionListener();

public void setActionListener(ActionListener listener);
```

Return or replace an `ActionListener` instance that will be utilized to process `ActionEvent` events during the *Apply Request Values* or *Invoke Application* phase of the request processing lifecycle. [P1-start defualt ActionListener requirements] The JSF implementation must provide a default implementation `ActionListener` that performs the following functions:

- The `processAction()` method must call `FacesContext.renderResponse()` in order to bypass any intervening lifecycle phases, once the method returns.
- The `processAction()` method must next determine the logical outcome of this event, as follows:

- If the originating component has a non-null `action` property, retrieve the `MethodBinding` and call `invoke()` to perform the application-specified processing in this action method. If the method returns non-`null`, call `toString()` on the result and use the value returned as the logical outcome. See Section 3.2.1.1 “Properties” for a decription of the `action` property.
- Otherwise, the logical outcome is `null`.

- The `processAction()` method must finally retrieve the `NavigationHandler` instance for this application, and pass the logical outcome value (determined above) as a parameter to the `handleNavigation()` method of the `NavigationHandler` instance. [P1-end]

See the Javadocs for `getActionListener()` for important backwards compatability information.

7.1.2 DefaultRenderKitId Property

```
public String getDefaultRenderKitId();

public void setDefaultRenderKitId(String defaultRenderKitId);
```

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

[P1-start defualtRenderKit called after startup] Unless the application has provided a custom `ViewHandler` that supports the use of multiple RenderKit instances in the same application, this method may only be called at application startup, before any Faces requests have been processed. [P1-end] This is a limitation of the current Specification, and may be lifted in a future release.

7.1.3 NavigationHandler Property

```
public NavigationHandler getNavigationHandler();

public void setNavigationHandler(NavigationHandler handler);
```

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

7.1.4 StateManager Property

```
public StateManager getStateManager();

public void setStateManager(StateManager manager);
```

Return or replace the `StateManager` instance that will be utilized during the *Restore View* and *Render Response* phases of the request processing lifecycle to manage state persistence for the components belonging to the current view. A default implementation must be provided, which operates as described in Section 7.7 “StateManager”.

7.1.5 ELResolver Property

```
public ELResolver getELResolver();

public void addELResolver(ELResolver resolver);
```

[N/T-start elresolver test] Return the `ELResolver` instance to be used for all EL resolution. This is actually an instance of `javax.el.CompositeELResolver` that must contain the `ELResolver` instances as specified in *Section 5.6.2 "ELResolver for Facelets and Programmatic Access"*. [N/T-end]

[N/T-start addELResolver ordering] `addELResolver` must cause the argument `resolver` to be added at the end of the list in the `javax.el.CompositeELResolver` returned from `getELResolver`(). See the diagram in *Section 5.6.2 "ELResolver for Facelets and Programmatic Access"* [N/T-end]

7.1.6 ELContextListener Property

```
public addELContextListener(ELContextListener listener);

public void removeELContextListener(ELContextListener listener);

public ELContextListener[] getELContextListeners();
```

`addELContextListener`() registers an `ELContextListener` for the current Faces application. This listener will be notified on creation of `ELContext` instances, and it will be called once per request.

`removeELContextListener`() removes the argument `listener` from the list of `ELContextListener`s. If `listener` is `null`, no exception is thrown and no action is performed. If `listener` is not in the list, no exception is thrown and no action is performed.

`getELContextListeners()` returns an array representing the list of listeners added by calls to `addELContextListener`().

7.1.7 ViewHandler Property

```
public ViewHandler getViewHandler();

public void setViewHandler(ViewHandler handler);
```

See Section 7.5 "ViewHandler" for the description of the ViewHandler. The JSF implementation must provide a default `ViewHandler` implementation. This implementation may be replaced by calling `setViewHandler()` before the first time the *Render Response* phase has executed. [P1-start setViewHandler() called after startup] If a call is made to `setViewHandler()` after the first time the *Render Response* phase has executed, the call must be ignored by the implementation. [P1-end]

7.1.8 ProjectStage Property

```
public ProjectStage getProjectStage();
```

[P1-start getProjectStage]This method must return the enum constant from the class `javax.faces.application.ProjectStage` as specified in the corresponding application init parameter, JNDI entry, or default Value. See Section 11.1.3 “Application Configuration Parameters”.[P1-end]

7.1.9 Acquiring ExpressionFactory Instance

```
public ExpressionFactory getExpressionFactory();
```

Return the `ExpressionFactory` instance for this application. This instance is used by the `evaluateExpressionGet` (*`Section 7.1.10 “Programmatically Evaluating Expressions”`*) `convenience method.`

[P1-start getExpressionFactory requirements] The default implementation simply returns the `ExpressionFactory` from the JSP container by calling `JspFactory.getDefaultFactory().getJspApplicationContext(servletContext).getExpressionFactory()`. [P1-end]

7.1.10 Programmatically Evaluating Expressions

```
public Object evaluateExpressionGet(FacesContext context, String
expression, Class expectedType)
```

Get a value by evaluating an expression.

Call `getExpressionFactory().createValueExpression()` passing the argument `expression` and `expectedType`. Call `FacesContext.getELContext()` and pass it to `ValueExpression.getValue()`, returning the result.

It is also possible and sometimes desireable to obtain the actual `ValueExpression` or `MethodExpression` instance directly. This can be accomplished by using the `createValueExpression()` or `createMethodExpression()` methods on the `ExpressionFactory` returned from `getExpressionFactory()`.

7.1.11 Object Factories

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

```
public UIComponent createComponent(String componentType);
public UIComponent createComponent(String componentType,
                                   String rendererType);

public Converter createConverter(Class targetClass);

public Converter createConverter(String converterId);

public Validator createValidator(String validatorId);

public Behavior createBehavior(String behaviorId);
```

Each of these methods creates a new instance of an object of the requested type[1], based on the requested identifier. The names of the implementation class used for each identifier is normally provided by the JSF implementation automatically (for standard classes described in this Specification), or in one or more application configuration resources (see Section 11.4 “Application Configuration Resources”) included with a JSF web application, or embedded in a JAR file containing the corresponding implementation classes.

All variants `createConverter()` must take some action to inspect the converter for `@ResourceDependency` and `@ListenerFor` annotations.

```
public UIComponent createComponent(ValueExpression
componentExpression, FacesContext context, String componentType);
```

[P1-start createComponent(ValueExpression) requirements] This method has the following behavior:

- Call the `getValue()` method on the specified `ValueExpression`, in the context of the specified `FacesContext`. If this results in a non-null `UIComponent` instance, return it as the value of this method.
- If the getValue() call did not return a component instance, create a new component instance of the specified component type, pass the new component to the s`etValue()` method of the specified `ValueExpression`, and return it.[P1-end]

```
public UIComponent createComponent(FacesContext context, Resource
componentResource);
```

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

All variants `createComponent()` must take some action to inspect the component for `@ResourceDependency` and `@ListenerFor` annotations. Please see the JavaDocs and Section 3.6.2.1 “Composite Component Metadata” for the normative specification relating to this method.

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

public void addConverter(Class targetClass, String
converterClass);

public void addConverter(String converterId, String
converterClass);

public void addValidator(String validatorId, String
validatorClass);

public void addBehavior(String behaviorId, String behaviorClass);
```

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

```
public Iterator<String> getComponentTypes();

public Iterator<String> getConverterIds();

public Iterator<Class> getConverterTypes();

public Iterator<String> getValidatorIds();

public Iterator<String> getBehaviorIds();
```

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

7.1.11.1 Default Validator Ids

From the list of mappings of `validatorId` to fully qualified class name, added to the application via calls to `addValidator()`, the application maintains a subset of that list under the heading of default validator ids. The following methods provide access to the default validator ids registered on an application:

```
public void addDefaultValidatorId(String validatorId);
public Map<String,String> getDefaultValidatorInfo();
```

The required callsites for these methods are specified in Section 3.5.3 “Validation Registration”.

7.1.12 Internationalization Support

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

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

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

```
public String getMessageBundle();

public void setMessageBundle(String messageBundle);
```

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

7.1.13 System Event Methods

System events are described in Section 3.4.3 “System Events”. This section describes the methods defined on `Application` that support system events

7.1.13.1 Subscribing to system events

```
public abstract void subscribeToEvent(Class<? extends SystemEvent>
systemEventClass, SystemEventListener listener)
public abstract void subscribeToEvent(Class<? extends SystemEvent>
systemEventClass, Class sourceClass, SystemEventListener
listener);
public abstract void publishEvent(Class<? extends SystemEvent>
systemEventClass, SystemEventListenerHolder source);
public void publishEvent(Class<? extends SystemEvent>
systemEventClass, Class<?> sourceBaseType, Object source)
```

The first variant of `subscribeToEvent()` subscribes argument `listener` to have its `isListenerForSource()` method, and (depending on the result from `isListenerForSource()`) its `processEvent()` method called any time any call is made to `Application.publishEvent(Class<? extends SystemEvent> systemEventClass, SystemEventListenerHolder source)` where the first argument in the call to `publishEvent()` is equal to the first argument to `subscribeToEvent()`. [P1-start eventClassAndInheritance] *NOTE*: The implementation must not support subclasses for the `systemEventClass` and/or `sourceClass` arguments to `subscribeToEvent()` or `publishEvent()`.[P1-end] For example, consider two event types, `SuperEvent` and `SubEvent extends SuperEvent`. If a listener subscribes to `SuperEvent.class` events, but later someone publishes a `SubEvent.class` event (which extends `SuperEvent`), the listener for `SuperEvent.class` must not be called.

The second variant of `subscribeToEvent()` is equivalent to the first, with the additional constraint the the `sourceClass` argument to `publishEvent()` must be equal to the `Class` object obtained by calling `getClass()` on the `source` argument to `publishEvent()`.

See the javadocs for both variants of `subscribeForEvent()` for the complete specification of these methods.

`publishEvent()` is called by the system at several points in time during the runtime of a JSF application. The specification for when `publishEvent()` is called is given in the javadoc for the event classes that are listed in Section 3.4.2.1 “Event Classes”. See the javadoc for `publishEvent()` for the complete specification.

7.1.13.2 Unsubscribing from system events

```
public abstract void unsubscribeFromEvent(Class<? extends
SystemEvent> systemEventClass, SystemEventListener listener);
public abstract void unsubscribeFromEvent(Class<? extends
SystemEvent> systemEventClass, Class sourceClass,
SystemEventListener listener);
```

See the javadocs for both variants of `unsubscribeFromEvent()` for the complete specification.

7.2 ApplicationFactory

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

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

The `ApplicationFactory` implementation class supports the following methods:

```
public Application getApplication();

public void setApplication(Application application);
```

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

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

7.3 Application Actions

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

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

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

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

7.4 NavigationHandler

7.4.1 Overview

A single `NavigationHandler` instance is responsible for consuming the logical outcome returned by an application action that was invoked, along with additional state information that is available from the `FacesContext` instance for the current request, and (optionally) selecting a new view to be rendered. If the outcome returned by the applicationaction is `null`, and none of the navigation cases that map to the current view identifier have a non-null condition expression, the same view must be re-displayed. This is a change from the old behavior. As of JSF 2.0, the NavigationHandler is consulted even on a null outcome, but under this circumstance it only checks navigation cases that do not specify an outcome (no <from-outcome>) and have a condition expression (specified with <if>). This is the only case where the same view (and component tree) is re-used..

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

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

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

Prior to JSF 2, the NavigationHandler's sole task was to execute the navigation for a given scenario. JSF 2 introduces the `ConfigurableNavigationHandler` interface, which extends the contract of the `NavigationHandler` to include two additional methods that accomodate runtime inspection of the NavigationCases that represent the rule-based navigation metamodel. The method `getNavigationCase` consults the `NavigationHandler` to determine which `NavigationCase` the `handleNavigation` method would resolve for a given "from action" expression and logical

outcome combination. The method `getNavigation`Cases returns a java.util.Map of all the `NavigationCase` instances known to this `NavigationHandler`. Each key in the map is a from view ID and the cooresponding value is a java.util.Set of NavigationCases for that from view ID.

```
public void getNavigationCase(FacesContext context, String
fromAction, String outcome);
public Map<String, Set<NavigationCase>> getNavigationCases();
```

[P1-start-configurablenavhandler]A JSF 2 compliant-implemention must ensure that its `NavigationHandler` implements the `ConfigurableNavigationHandler` interface. The `handleNavigation` and `getNavigation`Case methods should use the same logic to resolve a `NavigationCase`, which is outlined in the next section.[P1-end]

7.4.2 Default NavigationHandler Algorithm

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

The behavior of the default `NavigationHandler` implementation is configured, at web application startup time, from the contents of zero or more *application configuration resources* (see Section 11.4 "Application Configuration Resources"). The configuration information is represented as zero or more `<navigation-rule>` elements, each keyed to a matching pattern for the *view identifier* of the current view expressed in a `<from-view-id>` element. This matching pattern must be either an exact match for a view identifier (such as "/index.jsp" if you are using the default `ViewHandler`), or the prefix of a component view id, followed by an asterisk ("*") character. A matching pattern of "*", or the lack of a `<from-view-id>` element inside a `<navigation-rule>` rule, indicates that this rule matches any possible component view identifier.

Nested within each `<navigation-rule>` element are zero or more `<navigation-case>` elements that contain additional matching criteria based on the action reference expression value used to select an application action to be invoked (if any), and the logical outcome returned by calling the `invoke()` method of that application action[2]. As of JSF 2, navigation cases support a condition element, <if>, whose content must be a single, contiguous value expression expected to resolve to a boolean value (if the content does not match this requirement, the condition is ignored)[3]. When the <if> element is present, the value expression it contains must evalute to true when the navigation case is being consulted in order for the navigation case to match[4]. Finally, the <navigation-case> element contains a <to-view-id> element, whose content is either the view identifier or a value expression that resolves to the view identifier. If the navigation case is a match, this view identifier is to be selected and stored in the FacesContext for the current request following the invocation of the NavigationHandler. See below for an example of the configuration information for the default `NavigationHandler` might be configured.

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

[P1-start navigation handler requirements] The default `NavigationHandler` implementation must behave as if it were performing the following algorithm (although optimized implementation techniques may be utilized):

2. It is an error to specify more than one <navigation-case>, nested within one or more <navigation-rule> elements with the same <from-view-id> matching pattern, that have exactly the same combination of <from-xxx>, unless each is discriminated by a unique <if> element.

3. The presence of the <if> element in the absense of the <from-outcome> element is characterized as an alternate, contextual means of obtai

ning a logical outcome and thus the navigation case is checked even when the application action returns a a null (or void) outcome value.

4. Note that multiple conditions can be checked using the built-in operators and grouping provided by the Expression Language (e.g., and, or, not).

- If no navigation case is matched by a call to the handleNavigation() method, this is an indication that the current view should be redisplayed. As of JSF 2.0, a null outcome does not unconditionally cause all navigation rules to be skipped.
- Find a `<navigation-rule>` element for which the view identifier (of the view in the `FacesContext` instance for the current request) matches the `<from-view-id>` matching pattern of the `<navigation-rule>`. Rule instances are considered in the following order:
  - An exact match of the view identifier against a `<from-view-id>` pattern that does not end with an asterisk ("*") character.
  - For `<from-view-id>` patterns that end with an asterisk, an exact match on characters preceding the asterisk against the prefix of the view id. If the patterns for multiple navigation rules match, pick the longest matching prefix first.
  - If there is a `<navigation-rule>` with a `<from-view-id>` pattern of only an asterisk[5], it matches any view identifier.
- From the `<navigation-case>` elements nested within the matching `<navigation-rule>` element, locate a matching navigation case by matching the `<from-action>` and `<from-outcome>` values against the `fromAction` and outcome parameter values passed to the `handleNavigation()` method. To match an outcome value of null, the `<from-outcome>` must be absent and the `<if>` element present. Regardless of outcome value, if the `<if>` element is present, evaluate the content of this element as a value expression and only select the navigation case if the expression resolves to true. Navigation cases are checked in the following order:
  - Cases specifying both a `<from-action>` value and a `<from-outcome>` value are matched against the `action` expression and `outcome` parameters passed to the `handleNavigation()` method (both parameters must be not null, and both must be equal to the corresponding condition values, in order to match).
  - Cases that specify only a `<from-outcome>` value are matched against the `outcome` parameter passed to the `handleNavigation()` method (which must be not null, and equal to the corresponding condition value, to match).
  - Cases that specify only a `<from-action>` value are matched against the `action` expression parameter passed to the `handleNavigation()` method (which must be non-null, and equal to the corresponding condition value, to match; if the <if> element is absent, only match a non-null outcome; otherwise, match any outcome).
  - Any remaining case is assumed to match so long as the outcome parameter is non-null or the <if> element is present.
  - For cases that match up to this point and contain an <if> element, the condition value expression must be evaluated and the resolved value true for the case to match.
- If a matching `<navigation-case>` element was located, and the <redirect/> element was *not* specified in this <navigation-case> (or the application is running in a Portlet environment, where redirects are not possible), use the `<to-view-id>` element of the matching case to request a new `UIViewRoot` instance from the `ViewHandler` instance for this application, and pass it to the `setViewRoot()` method of the `FacesContext` instance for the current request. Then, exit the algorithm. If the content of <to-view-id> is a value expression, first evaluate it to obtain the value of the view id.
- If a matching `<navigation-case>` element was located, and the `<redirect/>` element *was* specified in this `<navigation-case>`, call `getRedirectURL()` on the `ViewHandler`, passing the current `FacesContext`, the `<to-view-id>`, any name=value parameter pairs specified within `<view-param>` elements within the `<redirect>` element, and the value of the `include-view-params` attribute of the `<redirect />` element if present, `false`, if not. The return from this method is the value to be sent to the client to which the redirect will occurr. Call `getFlash().setRedirect(true)` on the current `FacesContext`. Cause the current response to perform an HTTP redirect to this path, and call `responseComplete()` on the `FacesContext` instance for the current request. If the content of <to-view-id> is a value expression, first evaluate it to obtain the value of the view id.
- If no matching `<navigation-case>` element was located, return to Step 1 and find the next matching `<navigation-rule>` element (if any). If there are no more matching rule elements, execute the following algorithm to search for an implicit match based on the current `outcome`.
  - Let `outcome` be *viewIdToTest*.

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

- Examine the *viewIdToTest* for the presence of a "?" character, indicating the presence of a URI query string. If one is found, remove the query string from *viewIdToTest*, including the leading "?" and let it be *queryString*, look for the string "`faces-redirect=true`" within the query string. If found, let *isRedirect* be `true`, otherwise let *isRedirect* be `false`. Look for the string "`includeViewParams=true`". If found, let *includeViewParams* be `true`, otherwise let *includeViewParams* be `false`. When performing preemptive navigation, redirect is implied, even if the navigation case doesn't indicate it, and the query string must be preserved. Refer to Section 4.1.9 "UIOutcomeTarget" for more information on preemptive navigation.
- If *viewIdToTest* does not have a "file extension", take the file extension from the current `viewId` and append it properly to *viewIdToTest*.
- If *viewIdToTest* does not begin with "/", take the current `viewId` and look for the last "`/`". If not found, prepend a "`/`" and continue. Otherwise remove all characters in `viewId` after, but not including, "`/`", then append *viewIdToTest* and let the result be *viewIdToTest*.
- Obtain the current ViewHandler and call its `deriveViewId()` method, passing the current `FacesContext` and *viewIdToTest*. If `UnsupportedOperationException` is thrown, which will be the case if the `ViewHandler` is a Pre JSF 2.0 `ViewHandler`, the implementation must ensure the algorithm described for `ViewHandler.deriveViewId()` specified in `Section 7.5.2 "Default ViewHandler Implementation"` is performed. Let the result be *implicitViewId*.
- If the *implicitViewId* is non-`null`, take the following action. If *isRedirect* is `true`, append the *queryString* to *implicitViewId*. Let *virtualNavigationCase* be a conceptual `<navigation-case>` element whose `fromViewId` is the current `viewId`, `fromAction` is passed through from the arguments to `handleNavigation()`, `fromOutcome` is passed through from the arguments to `handleNavigation(), toViewId` is *implicitViewId*, and `redirect` is the value of *isRedirect,* and `include-view-params` is *includeViewParams*. Treat *virtualNavigationCase* as a matching navigation case and return to the first step above that starts with "If a matching `<navigation-case>` element was located...".

- If none of the above steps found a matching `<navigation-case>`, if `ProjectStage` is not `Production` render a message in the page that explains that there was no match for this outcome.

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

Query string parameters may be contributed by three different sources: the outcome (implicit navigation), a nested `<f:param>` on the component tag (e.g., `<h:link>`, `<h:button>`, `<h:commandLink>`, `<h:commandButton>`), and view parameters. When a redirect URL is built, whether it be by the `NavigationHandler` on a redirect case or a `UIOutcomeTarget` renderer, the query string parameter sources should be consulted in the following order:

- the outcome (implicit navigation)
- view parameter
- nested `<f:param>`

If a query string parameter is found in two or more sources, the latter source must replace all instances of the query string parameter from the previous source(s).

[P1-end]

7.4.3 Example NavigationHandler Configuration

The following <navigation-rule> elements might appear in one or more application configuration resources (see Section 11.4 “Application Configuration Resources”) to configure the behavior of the default NavigationHandler implementation:

```
<navigation-rule>

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

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

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

</navigation-rule>
```

```
<navigation-rule>

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

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

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

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

</navigation-rule>
```

```
<navigation-rule>

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

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

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

</navigation-rule>
```

```
<navigation-rule>

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

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

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

</navigation-rule>
```

```
public void savePizza();

<navigation-rule>
  <description>
     Pizza topping selection navigation handling
  </description>
  <from-view-id>/selectToppings.xhtml</from-view-id>
  <navigation-case>
    <description>
     Case where pizza is saved but there is additional cost
    </description>
    <display-name>Pizza saved w/ extras</display-name>
    <from-action>#{pizzaBuilder.savePizza}</from-action>
    <if>#{pizzaBuilder.additionalCost}</if>
    <to-view-id>/approveExtras.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <description>
     Case where pizza is saved and additional pizzas are needed
    </description>
    <display-name>
     Pizza saved, additional pizzas needed
    </display-name>
    <from-action>#{pizzaBuilder.savePizza}</from-action>
    <if>#{not order.complete}</if>
    <to-view-id>/createPizza.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <description>
      Handle case where pizza is saved and order is complete
    </description>
    <display-name>Pizza complete</display-name>
    <from-action>#{pizzaBuilder.savePizza}</from-action>
    <if>#{order.complete}</if>
    <to-view-id>/cart.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

```
public String placeOrder();

<navigation-rule>
  <description>
    Cart navigation handling
  </description>
  <from-view-id>/cart.xhtml</from-view-id>
  <navigation-case>
    <description>
      Handle case where account has one click delivery enabled
    </description>
   <display-name>Place order w/ one-click delivery</display-name>
    <from-action>#{pizzaBuilder.placeOrder}</from-action>
    <if>#{account.oneClickDelivery}</if>
    <to-view-id>/confirmation.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <description>
      Handle case where delivery information is required
    </description>
    <display-name>
     Place order w/o one-click delivery
    </display-name>
    <from-action>#{pizzaBuilder.placeOrder}</from-action>
    <if>#{not account.oneClickDelivery}</if>
    <to-view-id>/delivery.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

7.5 ViewHandler

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

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

7.5.1 Overview

ViewHandler defines the public APIs described in the following paragraphs

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

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

```
public void initView(FacesContext) throws FacesException;
public String calculateCharacterEncoding(FacesContext context);
```

The `initView()` method must be called as the first method in the implementation of the *Restore View Phase* of the request processing lifecycle, immediately after checking for the existence of the `FacesContext` parameter. See the javadocs for this method for the specification..

```
public String deriveViewId(FacesContext context, String input);
```

The `deriveViewId()` method is an encapsulation of the viewId derivation algorithm in previous versions of the specification. This method looks at the argument `input`, and the current request and derives the `viewId` upon which the lifecycle will be run.

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

Create and return a new `UIViewRoot` instance, initialized with information from the specified `FacesContext` and view identifier parameters.

If the view being requested is a Facelet view, the `createView()` method must ensure that the `UIViewRoot` is fully populated with all the children defined in the VDL page before `createView()` returns.

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

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

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

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

```
public void renderView(FacesContext context, UIViewRoot
viewToRender) throws IOException, FacesException;
```

This method must be called during the *Render Response* phase of the request processing lifecycle. It must provide a valid `ResponseWriter` or `ResponseStream` instance, storing it in the `FacesContext` instance for the current request (see Section 6.1.8 “ResponseStream and ResponseWriter”), and then perform whatever actions are required to cause the view currently stored in the `viewRoot` of the `FacesContext` instance for the current request to be rendered to the corresponding writer or stream. It must also interact with the associated `StateManager` (see Section 7.7 “StateManager”), by calling the `getSerializedView()` and `saveView()` methods, to ensure that state information for current view is saved between requests.

```
public UIViewRoot restoreView(FacesContext context, String viewId)
throws IOException;
```

This method must be called from the *`Restore View`* phase of the request processing lifecycle. It must perform whatever actions are required to restore the view associated with the specified `FacesContext` and `viewId`.

It is the caller's responsibility to ensure that the returned `UIViewRoot` instance is stored in the `FacesContext` as the new `viewRoot` property. In addition, if `restoreView()` returns `null` (because there is no saved state for this view identifier), the caller must call `createView()`, and call `renderResponse()` on the `FacesContext` instance for this request.

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

Take any appropriate action to either immediately write out the current view's state information (by calling `StateManager.writeState()`), or noting where state information may later be written. This method must be called once per call to the `encodeEnd()` method of any renderer for a `UIForm` component, in order to provide the `ViewHandler` an opportunity to cause saved state to be included with each submitted form.

```
public ViewDeclarationLanguage getViewDeclarationLanguage();
```

See the javadocs for this method for the specification.

7.5.2 Default ViewHandler Implementation

The terms *view identifier* and `viewId` are used interchangeably below and mean the context relative path to the web application resource that produces the view, such as a JSP page or a Facelets page. In the JSP case, this is a context relative path to the jsp page representing the view, such as `/foo.jsp`. In the Facelets case, this is a context relative path to the XHTML page representing the view, such as `/foo.xhtml`.

JSF implementations must provide a default `ViewHandler` implementation, along with a default `ViewDeclarationLanguageFactory` implementation that vends `ViewDeclarationLanguage` implementations designed to support the rendering of JSP pages containing JSF components and Facelets pages containing JSF components. The default `ViewHandler` is specified in this section and the default `ViewDeclarationLanguage` implementations are specified in the following section.

[P1-start ViewHandler.deriveViewId() requirements] The `deriveViewId()` method must fulfill the following responsibilities:

- If prefix mapping (such as "/faces/*") is used for `FacesServlet`, if argument `input` is not-`null`, the `viewId` is set as the argument `input`. Otherwise, the `viewId` is set from the extra path information of the request URI.
- If suffix mapping (such as "*.faces") is used for `FacesServlet`, the `viewId` is set using following algorithm.

  If argument `input` is non-`null`, let *requestViewId* be the value of argument `input`. Otherwise, obtain the servlet path information of the request URI and let this value be *requestViewId*.

  Consult the javadocs for `ViewHandler.FACELETS_VIEW_MAPPINGS_PARAM_NAME` and perform the steps necessary to obtain a value for that param (or its alias as in the javadocs). Let this be *faceletsViewMappings*.

  Obtain the value of the context initialization parameter named by the symbolic constant `ViewHandler.DEFAULT_SUFFIX_PARAM_NAME` (if no such context initialization parameter is present, use the value of the symbolic constant `ViewHandler.DEFAULT_SUFFIX`). Let this be *jspDefaultSuffixes*. For each entry in the list from *jspDefaultSuffixes*, replace the suffix of *requestViewId* with the current entry from *jspDefaultSuffixes*. For discussion, call this *candidateViewId*. For each entry in *faceletsViewMappings*, If the current entry is a prefix mapping entry, skip it and continue to the next entry. If *candidateViewId* is exactly equal to the current entry, consider the algorithm complete with the result being *candidateViewId*. If the current entry is a wild-card extension mapping, apply it non-destructively to *candidateViewId* and look for a physical resource with that name. If present, consider the algorithm complete with the result being the name of the physical resource. Otherwise look for a physical resource with the name *candidateViewId*. If such a resource exists, consider the algorithm complete with the result being *candidateViewId*. If there are no entries in *faceletsViewMappings*, look for a physical resource with the name *candidateViewId*. If such a resource exists, *candidateViewId* is the correct `viewId`.

Otherwise, obtain the value of the context initialization parameter named by the symbolic constant `ViewHandler.FACELETS_SUFFIX_PARAM_NAME`. (if no such context initialization parameter is present, use the value of the symbolic constant `ViewHandler.DEFAULT_FACELETS_SUFFIX`). Let this be *faceletsDefaultSuffix*. Replace the suffix of *requestViewId* with *faceletsDefaultSuffix*. For discussion, call this *candidateViewId*. If a physical resource exists with that name, *candidateViewId* is the correct `viewId`.

Otherwise, if a physical resource exists with the name *requestViewId* let that value be `viewId`.[P1-end]

[P1-start ViewHandler.calculateCharacterEncoding() requirements] The `calculateCharacterEncoding()` method must fulfill the following responsibilities:

- Examine the `Content-Type` request header. If it has a `charset` parameter extract it and return it.
- If not, test for the existence of a session by calling `getSession(false)` on the `ExternalContext` for this `FacesContext`. If the session is non-`null`, look in the `Map` returned by the `getSessionMap()` method of the `ExternalContext` for a value under the key given by the value of the symbolic constant `javax.faces.application.ViewHandler.CHARACTER_ENCODING_KEY`. If a value is found, convert it to a String and return it. [P1-end]

[P1-start calculateLocale() requirements] The `calculateLocale()` method must fulfill the following responsibilities:

- Attempt to match one of the locales returned by the `getLocales()` method of the `ExternalContext` instance for this request, against the supported locales for this application as defined in the application configuration resources. Matching is performed by the algorithm described in Section JSTL.8.3.2 of the JSTL Specification. If a match is found, return the corresponding `Locale` object.
- Otherwise, if the application has specified a default locale in the application configuration resources, return the corresponding `Locale` object.
- Otherwise, return the value returned by calling `Locale.getDefault()`.[P1-end]

[P1-start calculateRenderKitId() requirements] The `calculateRenderKitId()` method must fulfill the following responsibilities:

- Return the value of the request parameter named by the symbolic constant `ResponseStateManager.RENDER_KIT_ID_PARAM` if it is not `null`.
- Otherwise, return the value returned by `Application.getDefaultRenderKitId()` if it is not `null`.
- Otherwise, return the value specified by the symbolic constant `RenderKitFactory.HTML_BASIC_RENDER_KIT.`

[P1-start createView() requirements] The `createView()` method must obtain a reference to the `ViewDeclarationLanguage` for this `viewId` and call its `ViewDeclarationLanguage.createView()` method, returning the result and not swallowing any exceptions thrown by that method.[P1-end]

[P1-start getActionURL() requirements] The `getActionURL()` method must fulfill the following responsibilities:

- If the specified `viewId` does not start with a "/", throw `IllegalArgumentException`.
- If prefix mapping (such as "/faces/*") is used for `FacesServlet`, prepend the context path of the current application, and the specified prefix, to the specified viewId and return the completed value. For example "`/cardemo/faces/chooseLocale.jsp`".
- If suffix mapping (such as "*.faces") is used for `FacesServlet`, the following algorithm must be followed to derive the result.

  If the argument `viewId` has no extension, the result is `contextPath + viewId + mapping`, where `contextPath` is the context path of the current application, `viewId` is the argument `viewId` and `mapping` is the value of the mapping (such as "*.faces").

  If the argument `viewId` has an extension, and this extension is not `mapping`, the result is `contextPath + viewId.substring(0, period) + mapping`.

  If the argument `viewId` has an extension, and this extension is `mapping`, the result is `contextPath + viewId`.

  For example "`/cardemo/chooseLocale.faces`"[P1-end]

[P1-start getResourceURL() requirements] The `getResourceURL()` method must fulfill the following responsibilities:

- If the specified path starts with a "/", prefix it with the context path for the current web application, and return the result.
- Otherwise, return the specified `path` value unchanged.[P1-end]

[P1-start initView() requirements] The `initView()` method must fulfill the following responsibilities:

- See the javadocs for this method for the specification.[P1-end]

[P1-start renderView() requirements] The `renderView()` method must obtain a reference to the `ViewDeclarationLanguage` for the `viewId` of the argument `viewToRender` and call its `ViewDeclarationLanguage.createView()` method, returning the result and not swallowing any exceptions thrown by that method.[P1-end]

[P1-start restoreView() requirements]The `restoreView()` method must obtain a reference to the `ViewDeclarationLanguage` for the `viewId` of the argument `viewToRender` and call its `ViewDeclarationLanguage.createView()` method, returning the result and not swallowing any exceptions thrown by that method.[P1-end]

The `writeState()` method must fulfill the following responsibilities:

- Obtain the saved state stored in a thread-safe manner during the invocation of `renderView()` and pass it to the `writeState()` method of the `StateManager` for this application. [N/T-end]

In applications whose views are not written in JSP or Facelets, these responsibilities must be performed by a custom `ViewHandler` and/or `ViewDeclarationLanguage` implementation.

7.6 ViewDeclarationLanguage

To support the introduction of Facelets into the core specification, whilst preserving backwards compatibility with existing JSP applications, the concept of the *View Declaration Language* was formally introduced in version 2 of the specification. A View Declaration Language (VDL) is a syntax used to declare user interfaces comprised of instances of JSF `UIComponent`s. Under this definition, both JSP and Facelets are examples of an implementation of a VDL. Any of the responsibilities of the `ViewHandler` that specifically deal with the VDL sub-system are now the domain of the VDL implementation. These responsibilities are defined on the `ViewDeclarationLanguage` class.

7.6.1 ViewDeclarationLanguageFactory

`ViewDeclarationLanguageFactory` is a factory object that creates (if needed) and returns a new `ViewDeclarationLanguage` instance based on the VDL found in a specific view.

The factory mechanism specified in Section 11.2.6.1 "FactoryFinder" and the decoration mechanism specified in Section 11.4.6 "Delegating Implementation Support" are used to allow decoration or replacement of the `ViewDeclarationLanguageFactory`.

```
public ViewDeclarationLanguage getViewDeclarationLanguage(String
viewId)
```

Return the `ViewDeclarationLanguage` instance suitable for handling the VDL contained in the page referenced by the argument viewId. [P1-start_required_ViewDeclarationLanguageImpls]The default implementation must return a valid `ViewDeclarationLanguage` instance for views written in either JSP or Facelets. [P1-end_required_ViewDeclarationLanguageImpls]Whether the instance returned is the same for a JSP or a Facelet view is an implementation detail.

7.6.2 Default ViewDeclarationLanguage Implementation

For each of the methods on `ViewDeclarationLanguage`, the required behavior is broken into three segments:

- Behavior required of all compliant implementations
- Behavior required of the implementation that handles Facelet pages
- Behavior required of the implementation that handles JSP pages

Any implementation strategy is valid as long as these requirements are met.

7.6.2.1 ViewDeclarationLanguage.createView()

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

[P1-start createView() requirements] The `createView()` method must fulfill the following responsibilities.

All implementations must:

- If there is an existing `UIViewRoot` available on the `FacesContext`, this method must copy its `locale` and `renderKitId` to this new view root. If not, this method must call `calculateLocale()` and `calculateRenderKitId()`, and store the results as the values of the `locale` and `renderKitId`, proeprties, respectively, of the newly created `UIViewRoot`.
- If no `viewId` could be identified, or the `viewId` is exactly equal to the servlet mapping, send the response error code `SC_NOT_FOUND` with a suitable message to the client.
- Create a new `UIViewRoot` object instance using `Application.createComponent(UIViewRoot.COMPONENT_TYPE)`.
- The new `UIViewRoot` instance must be passed to `FacesContext.setViewRoot()`. This enables the broadest possible range of implementations for how tree creation is actually implemented.

The JSP and Facelet implementation is not required to take any additional action.

All implementations must:

- Return the newly created `UIViewRoot`.

[P1-end]

7.6.2.2 ViewDeclarationLanguage.buildView()

```
public void buildView(FacesContext context, UIComponent root)
```

[P1-start buildView() requirements] The `buildView()` method must fulfill the following responsibilities.

All implementations must:

- The implementation must guarantee that the page is executed in such a way that the `UIComponent` tree described in the VDL page is completely built and populated, rooted at the new `UIViewRoot` instance created previously.

[P1-end]

7.6.2.3 ViewDeclarationLanguage.getComponentMetadata()

```
public BeanInfo getComponentMetadata(FacesContext context,
Resource componentResource)
```

[P1-start getComponentMetaData() requirements] The `getComponentMetadata()` method must fulfill the following responsibilities:

All implementations must:

- Return a reference to the component metadata for the composite component represented by the argument `componentResource`, or `null` if the metadata cannot be found. The implementation may share and pool what it ends up returning from this method to improve performance.

The Facelets implementation must

- Support argument `componentResource` being a Facelet markup file that is to be interpreted as a composite component as specified in Section 3.6.2.1 “Composite Component Metadata”.

The JSP implementation is not required to support argument `componentResource` being a JSP markup file. In this case, `null` must be returned from this method.[P1-end]

7.6.2.4 ViewDeclarationLanguage.getViewMetadata() and getViewParameters()

```
public ViewMetadata getViewMetadata(FacesContext context, String
viewId)
```

[P1-start getViewtMetaData() requirements] The `getViewMetadata()` method must fulfill the following responsibilities:

All implementations must:

- Return a reference to the view metadata for the view represented by the argument `viewId`, or `null` if the metadata cannot be found. The implementation may share and pool what it ends up returning from this method to improve performance.

The Facelets implementation must support argument `viewId` being a Facelet markup file from which the view metadata should be extracted.

The JSP implementation is not required to support argument `viewId` being a JSP markup file. In this case, `null` must be returned from this method.[P1-end]

ViewMetadata Contract

```
public UIViewRoot createMetadataView()
```

The content of the metadata is provided by the page author as a special `<f:facet/>` of the `UIViewRoot`. The name of this facet is given by the value of the symbolic constant `UIViewRoot.METADATA_FACET_NAME`. The `UIViewRoot` return from this method must have that facet, and its children as its only children. This facet may contain `<f:viewParameter>` children. Each such element is the metadata will cause a `UIViewParameter` to be added to the view. Because `UIViewParameter` extends `UIInput` it is valid to attach any of the kinds of attached objects to an `<f:viewParameter>` that are valid for any element that represents any other kind of `UIInput` in the view.

]

```
public Collection<UIViewParameter> getViewParameters(UIViewRoot)
```

Convenience method that uses the view metadata specification above to obtain the `List<UIViewParameter>` for the argument viewId.

7.6.2.5 ViewDeclarationLanguage.getScriptComponentResource()

```
public Resource getScriptComponentResource(FacesContext context,
Resource componentResource)
```

[P1-start getScriptComponentResource() requirements] The `getScriptComponentResource()` method must fulfill the following responsibilities:

The Facelets implementation must:

- Take implementation specific action to discover a `Resource` given the argument `componentResource`. The returned `Resource` if non-`null`, must point to a script file that can be turned into something that extends `UIComponent` and implements `NamingContainer`.

The JSP implementation is not required to support this method. In this case, `null` must be returned from this method.[P1-end]

7.6.2.6 ViewDeclarationLanguage.renderView()

```
public void renderView(FacesContext context, String viewId)
```

[P1-start renderView() requirements] The `renderView()` method must fulfill the following responsibilities:

All implementations must:

- Return immediately if calling `isRendered()` on the argument `UIViewRoot` returns `true`.

The JSP implementation must:

- If the current request is a `ServletRequest`, call the `set()` method of the `javax.servlet.jsp.jstl.core.Config` class, passing the current `ServletRequest`, the symbolic constant `Config.FMT_LOCALE`, and the `locale` property of the specfied `UIViewRoot`. This configures JSTL with the application's preferred locale for rendering this response.
- Update the JSTL locale attribute in request scope so that JSTL picks up the new locale from the `UIViewRoot`. This attribute must be updated before the JSTL `setBundle` tag is called because that is when the new `LocalizationContext` object is created based on the locale.
- Create a wrapper around the current response from the `ExternalContext` and set it as the new response in the `ExternalContext`. Otherwise, omit this step. This wrapper must buffer all content written to the response so that it is ready for output at a later point. This is necessary to allow any content appearing after the `<f:view>` tag to appear in the proper position in the page.
- Execute the JSP page to build the view by treating the `viewId` as a context-relative path (starting with a slash character), by passing it to the `dispatch()` method of the `ExternalContext` associated with this request. Otherwise, continue to the next step. This causes control to pass to the JSP container, and then to `UIComponentClassicTagBase`. Please consult the javadocs for that class for the specification of how to handle building the view by executing the JSP page.

- Store the wrapped response in a thread-safe manner for use below. Otherwise, omit this step. The default implementation uses the request scope for this purpose.
- Restore the original response into the `ExternalContext`.
- If the `FacesContext` has a non-`null` `ResponseWriter` create a new writer using its `cloneWithWriter()` method, passing the response's `Writer` as the argument. Otherwise, use the current `RenderKit` to create a new `ResponseWriter`.
- Set the new `ResponseWriter` into the `FacesContext`, saving the old one aside.

All implementations must:

- Call `saveView()` on the `StateManager` for this application, saving the result in a thread-safe manner for use in the `writeState()` method of `ViewHandler`.
- Call `startDocument()` on the `ResponseWriter`.

The Facelets implementation must:

- Call `encodeAll()` on the `UIViewRoot`.

The JSP implementation must:

- Output any content in the wrapped response from above to the response, removing the wrapped response from the thread-safe storage.

All implementations must:

- Call `endDocument()` on the `ResponseWriter`.

The JSP implementation must:

- If the old `ResponseWriter` was not `null`, place the old `ResponseWriter` back into the `FacesContext`.

The Facelets implementation must

- Close the writer used to write the response.[P1-end]

7.6.2.7 ViewDeclarationLanguage.restoreView()

```
public UIViewRoot restoreView(FacesContext context, String viewId)
```

[P1-start restoreView() requirements]The `restoreView()` method must fulfill the following responsibilities:

All implementations must:

- If no `viewId` could be identified, return `null`.
- Call the `restoreView()` method of the associated `StateManager`, passing the `FacesContext` instance for the current request and the calculated `viewId`, and return the returned `UIViewRoot`, which may be `null`.[P1-end]

7.7 StateManager

`StateManager` directs the process of saving and restoring the view between requests. The `StateManager` instance for an application is retrieved from the `Application` instance, and therefore cannot know any details of the markup language created by the `RenderKit` being used to render a view. Therefore, the `StateManager` utilizes a helper object (see Section 8.4 "ResponseStateManager"), that is provided by the `RenderKit` implementation, and is therefore aware of the markup language details. The JSF implementation must provide a default `StateManager` implementation that supports the behavior described below.

7.7.1 Overview

Conceptually, the state of a view can be divided into two pieces:

- *Tree Structure*. This includes component parent-child relationships, including facets.
- *Component State*. This includes:
  - Component attributes and properties, and
  - `Validator`s, `Converter`s, `FacesListener`s, and other objects attached to a component. The manner in which these *attached objects* are saved is up to the component implementation. For attached objects that may have state, the `StateHolder` interface (see Section 3.2.4 "StateHolder") is provided to allow these objects to preserve their own attributes and properties. If an attached object does not implement `StateHolder`, but does implement `Serializable`, it is saved using standard serialization. Attached objects that do not implement either `StateHolder` or `Serializable` must have a public, zero-arg constructor, and will be restored only to their initial, default object state[6].

  It is beneficial to think of this separation between tree structure and tree state to allow the possibility that implementations can use a different mechanism for persisting the structure than is used to persist the state. For example, in a system where the tree structure is stored statically, as an XML file, for example, the system could keep a DOM representation of the trees representing the webapp UI in memory, to be used by all requests to the application.

7.7.2 State Saving Alternatives and Implications

JSF implementations support two primary mechanisms for saving state, based on the value of the `javax.faces.STATE_SAVING_METHOD` initialization parameter (see Section 11.1.3 "Application Configuration Parameters"). The possible values for this parameter give a general indication of the approach to be used, while allowing JSF implementations to innovate on the technical details:

- *client* -- Cause the saved state to be included in the rendered markup that is sent to the client (such as in a hidden input field for HTML). The state information must be included in the subsequent request, making it possible for JSF to restore the view without having saved information on the server side. It is advisable that this information be encrypted and tamper evident, since it is being sent down to the client, where it may persist for some time.
- *server* -- Cause the saved state to be stored on the server in between requests. Implementations that wish to enable their saved state to fail over to a different container instance must keep this in mind when implementing their server side state saving strategy. The default implementation Serializes the view in both the *client* and *server* modes. In the *server* mode, this serialized view is stored in the session and a unique key to retrieve the view is sent down to the client. By storing the serialized view in the session, failover may happen using the usual mechanisms provided by the container.

The values of all component attributes and properties (as well as the saved state of attached objects) must implement `java.io.Serializable`.

7.7.3 State Saving Methods.

```
public Object saveView(FacesContext context);
```

[P1-start saveView() requirements] This method causes the tree structure and component state of the view contained in the argument `FacesContext` to be collected, stored, and returned in a `java.lang.Object` instance that must implement `java.io.Serializable`. If `null` is returned from this method, there is no state to save.[P1-end]

6. The implementation classes for attached object must include a public zero-arguments constructor.

The returned object must represent the entire state of the view, such that a request processing lifecycle can be run against it on postback. Special care must be taken to guarantee that objects attached to component instances, such as listeners, converters, and validators, are also saved. The `StateHolder` interface is provided for this reason.

This method must also enforce the rule that component ids within a `NamingContainer` must be unique

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

Save the state represented in the specified `Object` instance, in an implementation dependent manner.

7.7.4 State Restoring Methods

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

Restore the tree structure and the component state of the view for this `viewId` to be restored, in an implementation dependent manner. If there is no saved state information available for this `viewId`, this method returns `null`.

The default implementation of this method calls through to `restoreTreeStructure()` and, if necessary `restoreComponentState()`.

7.7.5 Convenience Methods

```
public boolean isSavingStateInClient(FacesContext context);
```

[P1-start isSavingStateInClient() requirements] Return `true` if and only if the value of the `ServletContext` init parameter named by the value of the constant `StateManager.STATE_SAVING_METHOD_PARAM_NAME` is equal to the value of the constant `STATE_SAVING_METHOD_CLIENT`. Return `false` otherwise. [P1-end]

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

Return the current view state as a `String.` [P1-start-getViewState] This method must call `ResposeStateManger.getViewState.`[P1-end] Refer to Section 8.4 “ResponseStateManager” for more details.

7.8 ResourceHandler

The normative specification for this class is in the javadoc for `javax.faces.application.ResourceHandler`. See also Section 2.6 “Resource Handling”.

```
public ResourceHandler getResourceHandler();

public void setResourceHandler(ResourceHandler impl);
```

7.9 Deprecated APIs

7.9.1 PropertyResolver Property

```
public PropertyResolver getPropertyResolver();
[deprecated]

public void setPropertyResolver(PropertyResolver resolver);
[deprecated]
```

[N/T-start getPropertyResolver() requirements] `getPropertyResolver()` must return a `PropertyResolver` instance that wraps the `ELResolver` instance that Faces provides to the unified EL. [N/T-end] The `PropertyResolver` instance will be utilized to evaluate each . or [] operator when processing value expressions. This method has been **deprecated** for `getELResolver()` (see Section 7.1.5 "ELResolver Property").

`setPropertyResolver`() replaces the `PropertyResolver` instance that will be utilized to evaluate each `.` or `[]` operator when processing a value binding expression. A default implementation must be provided, which operates as described in *Section 5.8.2 "PropertyResolver and the Default PropertyResolver"*. This method has been **deprecated**. See the Javadocs for `setPropertyResolver`().

7.9.2 VariableResolver Property

```
public VariableResolver getVariableResolver();
[deprecated]

public void setVariableResolver(VariableResolver resolver);
[deprecated]
```

[N/T-start getVariableResolver() requirements] `getVariableResolver()` must return the `VariableResolver` that wraps the `ELResolver` instance that Faces provides to the unified EL. The `VariableResolver` instance will be utilized to convert the first name in a value expression into a corresponding object. The implementation must pass `null` as the base argument for any methods invoked on the underlying `ELResolver`. This method has been **deprecated** for `getELResolver`(). [N/T-end]

`setVariableResolver` replaces the `VariableResolver` instance that will be utilized to resolve method and value bindings. A default implementation must be provided, which operates as described in *Section 5.8.1 "VariableResolver and the Default VariableResolver"*. The method has been **deprecated**. See the Javadocs for `setVariableResolver`().

7.9.3 Acquiring ValueBinding Instances

```
public ValueBinding createValueBinding(String ref);
[deprecated]
```

Create and return a `ValueBinding` that can be used to evaluate the specified value binding expression. Call through to `createValueExpression`, passing the argument `ref`, `Object.class` for the expectedType, and `null` for the `fnMapper`. To avoid nondeterministic behavior, it is recommended that applications (or frameworks) wishing to plug in their own resolver implementations do so before `createValueBinding()` is called for the first time. This method has been **deprecated** for `createValueExpression()` (Section 7.1.10 “Programmatically Evaluating Expressions”

7.9.4 Acquiring MethodBinding Instances

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

Create and return a `MethodBinding` that can be used to evaluate the specified method binding expression, and invoke the specified method. The implementation must call through to `createMethodExpression,` passing the given arguments, and wrap the result in a `MethodBinding` implementation, returning it. The method that is invoked must have parameter signatures that are compatible with the classes in the `params` parameter[7] (which may be `null` or a zero-length array if the method to be called takes no parameters). The actual parameters to be passed when the method is executed are specified on the `invoke()` call of the returned `MethodBinding` instance.

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

7.9.5 Object Factories

```
public UIComponent createComponent(ValueBinding componentBinding,
FacesContext context, String componentType);
[deprecated]
```

Special version of the factory for UIComponent instances that is used when evaluating component binding expression properties. The implementation of this method must wrap the argument `componentBinding` in an implementation of `ValueExpression` and call through to `createComponent(javax.el.ValueExpression, javax.faces.FacesContext, java.lang.String).` This method has been deprecated for `createComponent`() using `ValueExpression` (see Section 7.1.11 “Object Factories”)

7.9.6 StateManager

This method causes the tree structure and component state of the view contained in the argument `FacesContext` to be collected, stored, and returned in a `StateManager.SerializedView` instance. If `null` is returned from this method, there is no state to save.

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

This method must also enforce the rule that component ids within a `NamingContainer` must be unique

```
public void writeState(FacesContext context,
StateManager.SerializedView state) throws IOException;
[deprecated]
```

Save the state represented in the specified `SerializedView` instance, in an implementation dependent manner.

```
protected Object getTreeStructureToSave(FacesContext context);
[deprecated]
```

This method must create a `Serializable` object that represents the tree structure of the component tree for this view. Tree structure is comprised of parent-child relationships, including facets. The `id` of each component and facet must also be saved to allow the naming containers in the tree to be correctly restored when this view is restored.

```
protected Object getComponentStateToSave(FacesContext context);
[deprecated]
```

This method must create a `Serializable` object representing the component state (attributes, properties, and attached objects) of the component tree for this view. Attached objects that wish to save and restore their own state must implement `StateHolder`.

7.9.7 ResponseStateManager

This method causes the tree structure and component state of the view contained in the argument `FacesContext` to be collected, stored, and returned in a `StateManager.SerializedView` instance. If `null` is returned from this method, there is no state to save.

This method must also enforce the rule that component ids within a `NamingContainer` must be unique

```
public void writeState(FacesContext context,
StateManager.SerializedView state) throws IOException;
[deprecated]
```

Save the state represented in the specified `SerializedView` instance, in an implementation dependent manner.

```
protected Object getTreeStructureToRestore(FacesContext context,
String viewId);
[deprecated]
```

The implementation must inspect the current request and return the tree structure Object passed to it on a previous invocation of `writeState()`..

```
protected Object getComponentStateToRestore(FacesContext context,
String viewId);
[deprecated]
```

The implementation must inspect the current request and return the component state Object passed to it on a previous invocation of `writeState()`.

8

Rendering Model

JavaServer Faces supports two programming models for decoding component values from incoming requests, and encoding component values into outgoing responses - the *direct implementation* and *delegated implementation* models. When the *direct implementation* model is utilized, components must decode and encode themselves. When the *delegated implementation* programming model is utilized, these operations are delegated to a `Renderer` instance associated (via the `rendererType` property) with the component. This allows applications to deal with components in a manner that is predominantly independent of how the component will appear to the user, while allowing a simple operation (selection of a particular `RenderKit`) to customize the decoding and encoding for a particular client device or localized application user.

Component writers, application developers, tool providers, and JSF implementations will often provide one or more `RenderKit` implementations (along with a corresponding library of `Renderer` instances). In many cases, these classes will be provided along with the `UIComponent` classes for the components supported by the `RenderKit`. Page authors will generally deal with `RenderKit`s indirectly, because they are only responsible for selecting a render kit identifier to be associated with a particular page, and a `rendererType` property for each `UIComponent` that is used to select the corresponding `Renderer`.

8.1 RenderKit

A `RenderKit` instance is optionally associated with a view, and supports components using the *delegated implementation* programming model for the decoding and encoding of component values. It also supports `Behavior` instances for the rendering of client side behavior and decoding for queuing `BehaviorEvents.` Refer to Section 3.7 “Component Behavior Model” for more details about this feature. [P1-start-renderkit]Each JSF implementation must provide a default `RenderKit` instance (named by the render kit identifier associated with the String constant `RenderKitFactory.HTML_BASIC_RENDER_KIT` as described below) that is utilized if no other `RenderKit` is selected.[P1-end]

```
public Renderer getRenderer(String family, String rendererType);
```

Return the `Renderer` instance corresponding to the specified component `family` and `rendererType (if any),` which will typically be the value of the `rendererType` property of a `UIComponent` about to be decoded or encoded

```
public ClientBehaviorRenderer getClientBehaviorRenderer(String
type);
```

Return the `ClientBehaviorRenderer` instance corresponding to the specified behavior type.

```
public void addRenderer(String family, String rendererType,
Renderer renderer);
```

```
public void addClientBehaviorRenderer(String type,
ClientBehaviorRenderer renderer);
```

```
public Iterator<String> getClientBehaviorRendererTypes();
```

Applications that wish to go beyond the capabilities of the standard `RenderKit` that is provided by every JSF implementation may either choose to create their own `RenderKit` instances and register them with the `RenderKitFactory` instance (see Section 8.5 “RenderKitFactory”), or integrate additional (or replacement) supported `Renderer` instances into an existing `RenderKit` instance. For example, it will be common for an application that requires custom component classes and `Renderer`s to register them with the standard `RenderKit` provided by the JSF implementation, at application startup time See Section 11.4.8 “Example Application Configuration Resource”for an example of a `faces-config.xml` configuration resource that defines two additional Renderer instances to be registered in the default `RenderKit`.

```
public ResponseWriter createResponseWriter(Writer writer, String
contentTypeList, String characterEncoding);
```

Use the provided `Writer` to create a new `ResponseWriter` instance for the specified character encoding.

The `contentTypeList` parameter is an "Accept header style" list of content types for this response, or `null` if the `RenderKit` should choose the best fit. [P1-start-contentTypeList]The `RenderKit` must support a value for the `contentTypeList` argument that comes straight from the `Accept` HTTP header, and therefore requires parsing according to the specification of the `Accept` header.[P1-end] Please see Section 14.1 of RFC 2616 (the HTTP 1.1 RFC) for the specification of the `Accept` header.

Implementors are advised to consult the `getCharacterEncoding()` method of class `javax.faces.servlet.ServletResponse` to get the required value for the characterEncoding parameter for this method. Since the `Writer` for this response will already have been obtained (due to it ultimately being passed to this method), we know that the character encoding cannot change during the rendering of the response. Please see Section 6.5 “ResponseWriter”

```
public ResponseStream createResponseStream(OuputStream out);
```

Use the provided `OutputStream` to create a new `ResponseStream` instance.

```
public ResponseStateManager getResponseStateManager();
```

Return an instance of `ResponseStateManager` to handle rendering technology specific state management decisions..

```
public Iterator<String> getComponentFamilies();
public Iterator<String> getRendererTypes(String componentFamily);
```

The first method returns an `Iterator` over the `component-family` entries supported by this `RenderKit`. The second one can be used to get an `Iterator` over the `renderer-type` entries for each of the `component-family` entries returned from the first method.

8.2 Renderer

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

```
public void decode(FacesContext context, UIComponent component);
```

For components utilizing the *delegated implementation* programming model, this method will be called during the *apply request values* phase of the request processing lifecycle, for the purpose of converting the incoming request information for this component back into a new local value. See the API reference for the `Renderer.decode()` method for details on its responsibilities.

```
public void encodeBegin(FacesContext context, UIComponent
component) throws IOException;

public void encodeChildren(FacesContext context, UIComponent
component) throws IOException;

public void encodeEnd(FacesContext context, UIComponent component)
throws IOException;
```

For components utilizing the *delegated implementation* programming model, these methods will be called during the *Render Response* phase of the request processing lifecycle. These methods have the same responsibilities as the corresponding `encodeBegin()`, `encodeChildren()`, and `encodeEnd()` methods of `UIComponent` (described in Section 3.1.13 “Component Specialization Methods” and the corresponding Javadocs) when the component implements the *direct implementation* programming model.

```
public String convertClientId(FacesContext context, String
clientId);
```

Converts a component-generated client identifier into one suitable for transmission to the client.

```
public boolean getRendersChildren();
```

Return a flag indicating whether this Renderer is responsible for rendering the children of the component it is asked to render.

```
 public Object getConvertedValue(FacesContext context,
UIComponent  component, Object submittedValue) throws
ConverterException;
```

Attempt to convert previously stored state information into an object of the type required for this component (optionally using the registered `Converter` for this component, if there is one). If conversion is successful, the new value should be returned from this method; if not, a `ConverterException` should be thrown.

A `Renderer` may listen for events using the `ListenerFor` annotation. Refer to the Javadocs for the `ListenerFor` class for more details.

8.3 ClientBehaviorRenderer

A `ClientBehaviorRenderer` instance produces client side behavior for components in the form of script content. It also participates in decoding and as such has the ability to enqueue server side `BehaviorEvents`..

```
public String getScript(ClientBehaviorContext behaviorContext,
ClientBehavior behavior);
```

Produce the script content that performs the client side `behavior. This method is called during the` *Render Response* phase of the request processing lifecycle.

```
public void decode(FacesContext context, UIComponent component,
ClientBehavior behavior);
```

This method will be called during the *apply request values* phase of the request processing lifecycle, for the primary purpose of enqueuing `BehaviorEvents.` All client behavior renderer implementations must extend from the `ClientBehaviorRenderer` interface.

8.3.1 ClientBehaviorRenderer Registration

ClientBehaviorRenderer implementations may be registered in the JSF faces-config.xml or with an annotation.

XML Registration

```
<renderkit>
 <renderkit-id>HTML_BASIC</renderkit-id>
 <client-behavior-renderer>
  <client-behavior-renderer-type>custom.behavior.Greet</client-
behavior-renderer-type>
  <client-behavior-renderer-class>greet.GreetRenderer</client-
behavior-renderer-class>
 </client-behavior-renderer>
...
```

Registration By Annotation

JSF provides the `javax.faces.render.FacesBehaviorRenderer annotation.`

```
@FacesClientBehaviorRenderer(value="Hello")
public class MyRenderer extends ClientBehaviorRenderer {
...
}
```

8.4 ResponseStateManager

`ResponseStateManager` is the helper class to `javax.faces.application.StateManager` that knows the specific rendering technology being used to generate the response. It is a singleton abstract class. This class knows the mechanics of saving state, whether it be in hidden fields, session, or some combination of the two.

```
public Object getState(FacesContext context);
```

[P1-start-getState]The implementation must inspect the current request and return the component tree state Object passed to it on a previous invocation of writeState()[P1-end]

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

Take the argument `state` and write it into the output using the current `ResponseWriter`, which must be correctly positioned already.

If the state is to be written out to hidden fields, the implementation must take care to make all necessary character replacements to make the Strings suitable for inclusion as an HTTP request parameter.

If the state saving method is *client* the implementation may encrypt the state to be saved to the client. We recommend that the state be unreadable by the client, and also be tamper evident.

Write out the render kit identifier associated with this `ResponseStateManager` implementation with the name as the value of the String constant `ResponseStateManager.RENDER_KIT_ID_PARAM.` [P1-start-renderkitid]This render kit identifier must not be written if:

- it is the default render kit identifier as returned by `Application.getDefaultRenderKitId()` or
- the render kit identifier is the value of `RenderKitFactory.HTML_BASIC_RENDER_KIT` and `Application.getDefaultRenderKitId()` returns `null`.[P1-end]

`ResponseStateManager.RENDER_KIT_ID_PARAM` is the name of the request parameter used by the default implementation of `ViewHandler.calculateRenderKitId()`to derive a render kit identifier..

```
public boolean isPostback(FacesContext context);
```

Return `true` if the current request is a postback. The default implementation returns `true` if this `ResponseStateManager` instance wrote out state on a previous request to which this request is a postback. Return false otherwise.

Please see *Section 7.9.7 "ResponseStateManager"* for deprecated methods in `ResponseStateManager`.

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

Return the view state as a `String` without any markup related to the rendering technology supported by this `ResponseStateManager.`

8.5 RenderKitFactory

[P1-start-renderkitFactory]A single instance of `javax.faces.render.RenderKitFactory` must be made available to each JSF-based web application running in a servlet or portlet container.[P1-end] The factory instance can be acquired by JSF implementations, or by application code, by executing

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

The `RenderKitFactory` implementation class supports the following methods:

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

Return a `RenderKit` instance for the specified render kit identifier, possibly customized based on the dynamic characteristics of the specified, (yet possibly null) `FacesContext`. For example, an implementation might choose a different `RenderKit` based on the "User-Agent" header included in the request, or the `Locale` that has been established for the response view. Note that applications which depend on this feature are not guaranteed to be portable across JSF implementations.

[P1-start-renderkitDefault]Every JSF implementation must provide a `RenderKit` instance for a default render kit identifier that is designated by the `String` constant `RenderKitFactory.HTML_BASIC_RENDER_KIT`.[P1-end] Additional render kit identifiers, and corresponding instances, can also be made available.

```
public Iterator<String> getRenderKitIds();
```

This method returns an `Iterator` over the set of render kit identifiers supported by this factory. [P1-start-renderkitIds]This set must include the value specified by `RenderKitFactory.HTML_BASIC_RENDER_KIT`.[P1-

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

end]

Register a `RenderKit` instance for the specified render kit identifier, replacing any previous RenderKit registered for that identifier.

8.6 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 required behavior of the standard HTML RenderKit is specified in a set of external HTML pages that accompany this specification, entitled “The Standard HTML RenderKit”. The behavior described in these pages is normative, and are required to be fulfilled by all implementations of JSF.

8.7 The Concrete HTML Component Classes

For each valid combination of `UIComponent` subclass and standard renderer given in the previous section, there is a concrete class in the package `javax.faces.component.html` package. Each class in this package is a subclass of an corresponding class in the `javax.faces.component` package, and adds strongly typed JavaBeans properties for all of the renderer-dependent properties. These classes also implement the `BehaviorHolder` interface, enabling them to have `Behavior attached to them. Refer to` Section 3.7 “Component Behavior Model” for additional details..

TABLE 8-1 Concrete HTML Component Classes

javax.faces.component class	renderer-type	javax.faces.component.html class
UICommand	javax.faces.Button	HtmlCommandButton
UICommand	javax.faces.Link	HtmlCommandLink
UIData	javax.faces.Table	HtmlDataTable
UIForm	javax.faces.Form	HtmlForm
UIGraphic	javax.faces.Image	HtmlGraphicImage
UIInput	javax.faces.Hidden	HtmlInputHidden
UIInput	javax.faces.Secret	HtmlInputSecret
UIInput	javax.faces.Text	HtmlInputText
UIInput	javax.faces.Textarea	HtmlInputTextarea
UIMessage	javax.faces.Message	HtmlMessage
UIMessages	javax.faces.Messages	HtmlMessages
UIOutput	javax.faces.Format	HtmlOutputFormat
UIOutput	javax.faces.Label	HtmlOutputLabel
UIOutput	javax.faces.Link	HtmlOutputLink
UIOutput	javax.faces.Text	HtmlOutputText
UIOutcomeTarget	javax.faces.Link	HtmlOutcomeTargetLink
UIOutcomeTarget	javax.faces.Button	HtmlOutcomeTargetButton
UIPanel	javax.faces.Grid	HtmlPanelGrid
UIPanel	javax.faces.Group	HtmlPanelGroup
UISelectBoolean	javax.faces.Checkbox	HtmlSelectBooleanCheckbox
UISelectMany	javax.faces.Checkbox	HtmlSelectManyCheckbox
UISelectMany	javax.faces.Listbox	HtmlSelectManyListbox
UISelectMany	javax.faces.Menu	HtmlSelectManyMenu

TABLE 8-1 Concrete HTML Component Classes

javax.faces.component class	renderer-type	javax.faces.component.html class
UISelectOne	javax.faces.Listbox	HtmlSelectOneListbox
UISelectOne	javax.faces.Menu	HtmlSelectOneMenu
UISelectOne	javax.faces.Radio	HtmlSelectOneRadio

[P1-start-htmlComponent]As with the standard components in the `javax.faces.component` package, each HTML component implementation class must define a static public final String constant named `COMPONENT_TYPE`, whose value is "`javax.faces.`" concatenated with the class name. HTML components, however, must not define a `COMPONENT_FAMILY` constant, or override the `getFamily()` method they inherit from their superclass.[P1-end]

9

Integration with JSP

Any JavaServer Faces implementations that claims compliance with this specification must include a complete JavaServer Pages implementation, and expose this implementation to the runtime of any JSF application. JSF applications, however, need not use JSP as their View Declaration Language (VDL). In fact, a JSF application is free to use whatever technology it likes for its VDL, as long as that VDL itself complies with the JSF specification.

This chapter describes the JSP support required by JavaServer Faces. This JSP support is enabled by providing custom actions so that a JSF user interface can be easy defined in a JSP page by adding tags corresponding to JSF UI components. Custom actions provided by a JSF implementation may be mixed with standard JSP actions and custom actions from other libraries, as well as template text for layout, in the same JSP page.

For JSP version 2.0 and onward, the file extension "`.jsf`" is reserved, and may optionally be used (typically by authoring tools) to represent JSP pages containing JSF content[1]. When running in a JSP 1.2 environment, JSP authors must give their JSP pages that contain JSF content a filename ending in "`.jsp`".

9.1 UIComponent Custom Actions

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

TABLE 9-1 Example Renderer Types

RendererType	Render-Dependent Attributes
"Text"	"size"
"Secret"	"size", "secretChar"
"Textarea"	"size", "rows"

1. If this extension is used, it must be declared in the web application deployment descriptor, as described in the JSP 2.0 (or later) specification.

The tag library descriptor (TLD) file for the corresponding tag library, then, would define three custom actions—one per `Renderer`. Below is an example of a portion of the custom action definition for the `inputText` tag[2]:

```
<tag>
   <name>inputText</name>
   <tag-class>acme.html.tags.InputTag</tag-class>
   <bodycontent>JSP</bodycontent>
   <attribute>
      <name>id</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
   </attribute>
   <attribute>
      <name>value</name>
      <required>false</required>
      <deferred-value>
        <type>java.lang.Object</type>
      <deferred-value>
   </attribute>
   <attribute>
      <name>size</name>
      <required>false</required>
      <deferred-value>
        <type>java.lang.Integer</type>
      <deferred-value>
   </attribute>
   ...
</tag>
```

Note that the `size` attribute is derived from the `Renderer` of type "Text", while the `id` and `value` attributes are derived from the `UIInput` component class itself. Also note that the `id` attribute has `rtexprvalue` set to `true`. This is to allow `${}` expressions in the `id` attribute so that `<c:forEach>` can include faces components that incorporate the index into their id. `RenderKit` implementors will generally provide a JSP tag library which includes component custom actions corresponding to each of the component classes (or types) supported by each of the `RenderKit`'s `Renderers`. See Section 8.1 "RenderKit" and Section 8.2 "Renderer" for details on the `RenderKit` and `Renderer` APIs. JSF implementations must provide such a tag library for the standard HTML RenderKit (see Section 9.5 "Standard HTML RenderKit Tag Library").

9.2 Using UIComponent Custom Actions in JSP Pages

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

9.2.1 Declaring the Tag Libraries

This specification hereby reserves the following Uniform Resource Identifier (URI) values to refer to the standard tag libraries for the custom actions defined by JavaServer Faces:

- **http://java.sun.com/jsf/core** -- URI for the *JavaServer Faces Core Tag Library*

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

- **http://java.sun.com/jsf/html** -- URI for the *JavaServer Faces Standard HTML RenderKit Tag Library*

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

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

declares the unique resource identifiers of the tag libraries being used, as well as the prefixes to be used within the current page for referencing actions from these libraries[3].

9.2.2 Including Components in a Page

A JSF `UIComponent` custom action can be placed at any desired position in a JSP page that contains the `taglib` directive for the corresponding tag library, subject to the following restrictions:

- When using a single JSP page to create the entire view, JSF component custom actions must be nested inside the `<f:view>` custom action from the JSF Core Tag Library.

The following example illustrates the general use of a UIComponent custom action in a JSP page. In this scenario:

```
<h:inputText id="username" value="#{logonBean.username}"/>
```

represents a `UIInput` field, to be rendered with the "Text" renderer type, and points to the username property of a backing bean for the actual value. The `id` attribute specifies the *component id* of a `UIComponent` instance, from within the component tree, to which this custom action corresponds. If no `id` is specified, one will be automatically generated by the custom action implementation.

Custom actions that correspond to JSF `UIComponent` instances must subclass `javax.faces.webapp.UIComponentELTag` (see Section 11.2.6.3 "UIComponentELTag")

During the *Render Response* phase of the request processing lifecycle, the appropriate encoding methods of the component (or its associated `Renderer`) will be utilized to generate the representation of this component in the response page. In addition, the first time a particular page is rendered, the component tree may also be dynamically constructed.

All markup other than `UIComponent` custom actions is processed by the JSP container, in the usual way. Therefore, you can use such markup to perform layout control, or include non-JSF content, in conjunction with the actions that represent UI components.

9.2.3 Creating Components and Overriding Attributes

As `UIComponent` custom actions are encountered during the processing of a JSP page, the custom action implementation must check the component tree for the existence of a corresponding `UIComponent`, and (if not found) create and configure a new component instance corresponding to this custom action. The details of this process (as implemented in the findComponent() method of UIComponentClassicTagBase, for easy reuse) are as follows:

- If the component associated with this component custom action has been identified already, return it unchanged.
- Identify the *component identifier* for the component related to this UIComponent custom action, as follows:
  - If the page author has specified a value for the `id` attribute, use that value.

3. Consistent with the way that namespace prefixes work in XML, the actual prefix used is totally up to the page author, and has no semantic meaning. However, the values shown above are the suggested defaults, which are used consistently in tag library examples throughout this specification.

- Otherwise, call the `createUniqueId()` method of the `UIViewRoot` at the root of the component tree for this view, and use that value.

- If this `UIComponent` custom action is creating a *facet* (that is, we are nested inside an `<f:facet>` custom action), determine if there is a facet of the component associated with our parent `UIComponent` custom action, with the specified facet name, and proceed as follows:
  - If such a facet already exists, take no additional action.
  - If no such facet already exists, create a new `UIComponent` (by calling the `createComponent()` method on the `Application` instance for this web application, passing the value returned by `getComponentType()`, set the component identifier to the specified value, call `setProperties()` passing the new component instance, and add the new component as a facet of the component associated with our parent `UIComponent` custom action, under the specified facet name.
- If this `UIComponent` custom action is not creating a facet (that is, we are not nested inside an `<f:facet>` custom action), determine if there is a child component of the component associated with our parent `UIComponent` custom action, with the specified component identifier, and proceed as follows:
  - If such a child already exists, take no additional action.
  - If no such child already exists, create a new `UIComponent` (by calling the `createComponent()` method on the `Application` instance for this web application, passing the value returned by `getComponentType()`, set the component identifier to the specified value, call `setProperties()` passing the new component instance, and add the new component as a child of the component associated with our parent `UIComponent` custom action.

9.2.4 Deleting Components on Redisplay

In addition to the support for dynamically creating new components, as described above, UIComponent custom actions will also *delete* child components (and facets) that are already present in the component tree, but are not rendered on this display of the page. For example, consider a UIComponent custom action that is nested inside a JSTL `<c:if>` custom action whose condition is true when the page is initially rendered. As described in this section, a new UIComponent will have been created and added as a child of the `UIComponent` corresponding to our parent `UIComponent` custom action. If the page is re-rendered, but this time the `<c:if>` condition is `false`, the previous child component will be removed.

9.2.5 Representing Component Hierarchies

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

```
<h:form id="logonForm">
  <h:panelGrid columns="2">
    <h:outputLabel for="username">
      <h:outputText value="Username:"/>
    </h:outputLabel>
    <h:inputText id="username"
       value="#{logonBean.username}"/>
    <h:outputLabel for="password">
      <h:outputText value="Password:"/>
    </h:outputLabel>
    <h:inputSecret id="password"
       value="#{logonBean.password}"/>
    <h:commandButton id="submitButton" type="SUBMIT"
       action="#{logonBean.logon}"/>
    <h:commandButton id="resetButton" type="RESET"/>
  </h:panelGrid>
</h:form>
```

9.2.6 Registering Converters, Event Listeners, and Validators

Each JSF implementation is required to provide the core tag library (see *Section 9.4 "JSF Core Tag Library"*), which includes custom actions that (when executed) create instances of a specified `Converter`, `ValueChangeListener`, `ActionListener` or `Validator` implementation class, and register the created instance with the `UIComponent` associated with the most immediately surrounding `UIComponent` custom action.

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

```
<h:inputText id="username" value="#{logonBean.username}">
   <f:validateLength minimum="6"/>
</h:inputText>
```

associates a validation check (that the value entered by the user must contain at least six characters) with the username `UIInput` component being described.

Following are usage examples for the `valueChangeListener` and `actionListener` custom actions.

```
<h:inputText id="maxUsers">
   <f:convertNumber integerOnly="true"/>
   <f:valueChangeListener
      type="custom.MyValueChangeListener"/>
</h:inputText>
<h:commandButton label="Login">
   <f:actionListener type="custom.MyActionListener"/>
</h:commandButton>
```

This example causes a `Converter` and a `ValueChangeListener` of the user specified type to be instantiated and added as to the enclosing `UIInput` component, and an `ActionListener` is instantiated and added to the enclosing `UICommand` component. If the user specified type does not implement the proper listener interface a `JSPException` must be thrown.

9.2.7 Using Facets

A *Facet* is a subordinate UIComponent that has a special relationship to its parent `UIComponent`, as described in Section 3.1.9 "Facet Management". Facets can be defined in a JSP page using the `<f:facet>` custom action. Each facet action must have one and only one child UIComponent custom action[4]. For example:

```
<h:dataTable ...>
   <f:facet name="header">
      <h:outputText value="Customer List"/>
   </f:facet>
   <h:column>
      <f:facet name="header">
         <h:outputText value="Account Id"/>
      </f:facet>
      <h:outputText id="accountId" value=
         "#{customer.accountId}"/>
   </h:column>
   ...
</h:dataTable>
```

9.2.8 Interoperability with JSP Template Text and Other Tag Libraries

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

- JSF component custom actions nested inside a custom action that conditionally renders its body (such as JSTL's `<c:if>` or `<c:choose>`) must contain a manually assigned `id` attribute.
- Interoperation with the JSTL Internationalization-Capable Formatting library (typically used with the "`fmt`" prefix) is restricted as follows:
  - The `<fmt:parseDate>` and `<fmt:parseNumber>` custom actions should not be used. The corresponding JSF facility is to use an `<h:inputText>` component custom action with an appropriate `DateTimeConverter` or `NumberConverter`.
  - The `<fmt:requestEncoding>` custom action should not be used. By the time it is executed, the request parameters will have already been parsed, so any change in the setting here will have no impact. JSF handles character set issues automatically in most cases. To use a fixed character set in exceptional circumstances, use the a "`<%@ page contentType="[content-type];[charset]" %>`" directive.
  - The `<fmt:setLocale/>` custom action should not be used. Even though it might work in some circumstances, it would result in JSF and JSTL assuming different locales. If the two locales use different character sets, the results will be undefined. Applications should use JSF facilities for setting the `locale` property on the `UIViewRoot` component to change locales for a particular user.

4. If you need multiple components in a facet, nest them inside a <h:panelGroup> custom action that is the value of the facet.

9.2.9 Composing Pages from Multiple Sources

JSP pages can be composed from multiple sources using several mechanisms:

- The `<%@include%>` directive performs a compile-time inclusion of a specified source file into the page being compiled[5]. From the perspective of JSF, such inclusions are transparent—the page is compiled as if the inclusions had been performed before compilation was initiated.
- Several mechanisms (including the `<jsp:include>` standard action, the JSTL `<c:import>` custom action when referencing a resource in the same webapp, and a call to `RequestDispatcher.include()` for a resource in the same webapp) perform a runtime dynamic inclusion of the results of including the response content of the requested page resource in place of the include action. Any JSF components created by execution of JSF component custom actions in the included resource will be grafted onto the component tree, just as if the source text of the included page had appeared in the calling page at the position of the include action.
- For mechanisms that aggregate content by other means (such as use of an `HttpURLConnection`, a `RequestDispatcher.include()` on a resource from a different web application, or accessing an external resource with the JSTL `<c:import>` custom action on a resource from a different web application, only the response content of the aggregation request is available. Therefore, any use of JSF components in the generation of such a response are not combined with the component tree for the current page.

9.3 UIComponent Custom Action Implementation Requirements

The custom action implementation classes for `UIComponent` custom actions must conform to all of the requirements defined in the JavaServer Pages Specification. In addition, they must meet the following JSF-specific requirements:

- Extend the `UIComponentELTag` or `UIComponentELBodyTag` base class, so that JSF implementations can recognize `UIComponent` custom actions versus others.
- Provide a public `getComponentType()` method that returns a String-valued component type registered with the `Application` instance for this web application. The value returned by this method will be passed to `Application.createComponent()` when a new `UIComponent` instance associated with this custom action is to be created.
- Provide a public `getRendererType()` method that returns a String-valued renderer type registered with the `RenderKit` instance for the currently selected `RenderKit`, or `null` if there should be no associated `Renderer`. The value returned by this method will be used to set the `rendererType` property of any UIComponent created by this custom action.
- Provide setter methods taking a `javax.el.ValueExpression` or `javax.el.MethodExpression` parameter for all set-able (from a custom action) properties of the corresponding `UIComponent` class, and all additional set-able (from a custom action) attributes supported by the corresponding `Renderer`.
- On the method that causes a `UIComponent` instance to be added to the tree, verify that the component id of that `UIComponent` is unique within the scope of the closest ancestor component that is a `NamingContainer`. If this constraint is not met, throw `JspException`.
- Provide a protected `setProperties()` method of type `void` that takes a `UIComponent` instance as parameter. The implementation of this method must perform the following tasks:
  - Call `super.setProperties()`, passing the same `UIComponent` instance received as a parameter.

5. In a JSP 2.0 or later environment, the same effect can be accomplished by using <include-prelude> and <include-coda> elements in the <jsp-config> element in the web application deployment descriptor.

- For each non-null custom action attribute that corresponds to a property based attribute to be set on the underlying component, call either `setValueExpression()` or `getAttributes().put()`, depending on whether or not a value expression was specified as the custom action attribute value (performing any required type conversion). For example, assume that title is the name of a render-dependent attribute for this component:

```
public void setTitle(javax.el.ValueExpression title) {
  this.title = title;
}

protected void setProperties(UIComponent component) throws
JspException {
  super.setProperties(component);
  if (title != null) {
    try {
        component.setValueExpression(“title”, title);
    }
    catch (ELException e) {
      throw new JspException(e);
    }
  ...
}
```

- For each non-null custom action attribute that corresponds to a method based attribute to be set on the underlying component, the value of the attribute must be a method reference expression. We have a number of wrapper classes to turn a MethodExpression into the appropriate listener. For example, assume that `valueChangeListener` is the name of an attribute for this component:

```
public void setValueChangeListener(javax.el.MethodExpression me)
{
  valueChangeListener = me;
}

protected void setProperties(UIComponent component) {
  super.setProperties(component);
  MethodExpressionValueChangeListener listener =
    new MethodExpressionValueChangeListener(valueChangeListener);
  input.addValueChangeListener(listener);
 ...
}
```

- Non-null custom action attributes that correspond to a writable property to be set on the underlying component are handled in a similar fashion. For example, assume a custom action for the `UIData` component is being created that needs to deal with the `rows` property (which is of type `int`):

```
public void setRows(javax.el.ValueExpression rows) {
  this.rows = rows;
}

protected void setProperties(UIComponent component) {
  super.setProperties(component);
  if (rows != null) {
    try {
        component.setValueExpression("rows", rows);
    } catch (ELException e) {
      throw new JspException(e);
    }
  }
  ...
}
```

- Optionally, provide a public `release()` method of type `void`, taking no parameters, to be called when the JSP page handler releases this custom action instance. If implemented, the method must perform the following tasks:
  - Call `super.release()` to invoke the superclass's release functionality.
  - Clear the instance variables representing the values for set-able custom action attributes (for example, by setting String values to null).
- Optionally provide overridden implementations for the following method to fine tune the behavior of your `UIComponent` custom action implementation class: `encodeComponent()`.

It is technically possible to override other public and protected methods of the `UIComponentELTag` or `UIComponentBodyELTag` base class; however, it is likely that overriding these methods will interfere with the functionality that other portions of the JSF implementation are assuming to be present, so overriding these methods is strongly discouraged.

The definition of each `UIComponent` custom action in the corresponding tag library descriptor (TLD) must conform to the following requirements:

- The `<body-content>` element for the custom action itself must specify `JSP`.
- For each attribute that is intended to be passed on to the underlying faces component:
  - The attribute may not be named `id`. This name is reserved for Faces use.
  - If the attribute represents a method expression, it must have a `<deferred-method>` element containing a `<method-signature>` element that describes the signature of the method pointed to by the expression, as described in section JSP.C.1 in the JSP 2.1 specification.
  - Otherwise, the attribute must be a value based attribute, and must have a `<deferred-value>` element containing a <type> element which describes the expected type to which the expression will evaluate. Please see section JSP.C.1 in the JSP 2.1 specification for details.

9.3.1 Considerations for Custom Actions written for JavaServer Faces 1.1 and 1.0

Versions 1.0 and 1.1 of the JavaServer Faces spec included their own EL that happend to have similar semantics to the JSP EL, but the implementation was bundled into the Faces implementation. This version leverages a new Unified EL facility provided by JSP. This change has necessitated deprecating some methods and classes, including the classes Custom Actions as their base class for tags that expose Faces components to the JSP page. This section explains how custom actions built for Faces 1.0 and 1.1 can continue to run Faces 1.2.

9.3.1.1 Past and Present Tag constraints

Faces 1.0 and 1.1 were targeted at JSP version 1.2 and Servlet version 2.3. This decision brought about several constraints for faces tag attributes:

- all tag attributes had to declare `rtexprvalue` to be `false.`
- all tag attributes had to take the type `java.lang.String`.
- Faces had to choose a new expression delimiter, `#{}`, to prevent the JSP container from prematurely evaluating the expression. This became known as deferred evaluation.
- Because Faces had introduced its own version of the EL, the custom tag action layer had to do a lot of extra work to "value binding enable" its attributes, calling Faces EL APIs to turn the String attribute value into an instance of `ValueBinding` or `MethodBinding`.
- Faces provided the `UIComponentTag` and `UIComponentBodyTag` base classes that were designed to adhere to the above rules.

Tags that use the Unified EL have the following constraints:

- all tag attributes must not have an `rtexprvalue` attribute
- all tag attributes must accept `javax.el.ValueExpression` or `javax.el.MethodExpression` as their type (depending on if the attribute refers to a method or a value).
- all tag attributes (except for `id`) must have a `<deferred-value>` or `<deferred-method>` element. See *Section 9.4 "JSF Core Tag Library"* in the description for the **Attributes** column.
- The JSP Container will hand the tag setter a `javax.el.ValueExpression` or `javax.el.MethodExpression` directly, so there is no need to use the Faces API to create them.
- The `UIComponentTag` and `UIComponentBodyTag` classes are deprecated and Faces provides new base class, UIComponentELTag to the new rules for taglibs in Faces.

It's very important to note that we still are using #{} as the delimiters for expressions that appear in a JSP page in the value of a tag attribute, but when the Java API is used, either ${} or #{} may be used for delimiters.

9.3.1.2 Faces 1.0 and 1.1 Taglib migration story

It is imperitive that applications written for Faces 1.0 and 1.1 continue to run on Faces 1.2. From the JSP perspective, this means

1. that JSP pages using the standard h: and f: tags must work without change
2. that JSP pages using custom faces taglibs must work without change

The first item is enabled by re-writing the h: and f: taglibs which must be provided by the Faces implementor.

The second item is enabled as follows. For discussion the term `jsp-version` is used to denote the `jsp-version` element in a JSP 1.2 (and earlier) TLD, as well as the `version` element in a JSP 2.0 (and later) TLD. The JSP container must examine the `jsp-version` element of the TLD for a taglib. If the `jsp-version` is less than 2.1, the taglib is deemed to be a Faces 1.0 or 1.1 taglib and the container must ignore all expressions that use #{} as delimiters, except for those appearing in tag attribute with a property setter that takes a `javax.el.ValueExpression` or `javax.el.MethodExpression`. If the `jsp-version` is 2.1 or greater, the taglib is deemed to be a Faces 1.2 or later taglib and the JSP container is aware of #{} expressions.

9.4 JSF Core Tag Library

[P1-start jsf_core taglib requirements] 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 `1.2`.
- 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 2.1). [P1-end]

[P1-start no javascript in jsf_core taglib] The tags in the implementation of this tag library must not cause JavaScript to be rendered to the client. Doing so would break the requirement that the JSF Core Tag library is independent of any specific RenderKit. [P1-end]

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

- **Name**—The name of this custom action, as used in a JSP page.
- **Short Description**—A summary of the behavior implemented by this custom action.
- **Syntax**—One or more examples of using this custom action, with the required and optional sets of attributes that may be used together. If the tag may have an *`id`* attribute, its value may be a literal string, or an immediate, non-defferd expression, such as "`userName`" or "`user${i}`" without the quotes.
- **Body Content**—The type of nested content for this custom action, using one of the standard values `empty`, `JSP`, or `tagdependent` as described in the JSP specification. This section also describes restrictions on the types of content (template text, JSF core custom actions, JSF `UIComponent` custom actions, and/or other custom actions) that can be nested in the body of this custom action.
- **Attributes**—A table containing one row for each defined attribute for this custom action. The following columns provide descriptive information about each attribute:
  - *Name*—Name of this attribute, as it must be used in the page. If the name of the attribute is in *italics*, it is required.
  - *Expr*—The type of dynamic expression (if any) that can be used in this attribute value. Legal values are VE (this may be a literal or a value expression), ME (this may be a method expression), or NONE (this attribute accepts literal values only). If the *Expr* column is VE, the corresponding `<attribute>` declaration in the TLD must contain a `<deferred-value>` element, optionally containing a `<type>` element that contains the fully qualified java class name of the expected type of the expression. If `<type>` is omitted, Object.class is assumed. If the *Expr* column is ME, the corresponding `<attribute>` declaration in the TLD must contain a `<deferred-method>` element, containing a `<method-signature>` element that describes the exact method signature for the method. In this case, the *Description* column the description column contains the method signature.
  - *Type*—Fully qualified Java class or primitive type of this attribute.
  - *Description*—The functional meaning of this attribute's value.
- **Constraints**—Additional constraints enforced by this action, such as combinations of attributes that may be used together.
- **Description**—Details about the functionality provided by this custom action.

9.4.1 <f:actionListener>

Register an `ActionListener` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

```
<f:actionListener type="fully-qualified-classname" binding="value Expression"/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
type	VE	`String`	Fully qualified Java class name of an `ActionListener` to be created and registered
binding	VE	`ValueExpression`	A `ValueExpression` expression that evaluates to an object that implements `javax.faces.event.ActionListener`

Constraints

- Must be nested inside a `UIComponent` custom action.
- The corresponding `UIComponent` implementation class must implement `ActionSource`, and therefore define a public `addActionListener()` method that accepts an `ActionListener` parameter.
- The specified listener class must implement `javax.faces.event.ActionListener`.
- *type* and/or *binding* must be specified.

[P1-start f:actionListener constraints] If this tag is not nested inside a `UIComponent` custom action, or the `UIComponent` implementation class does not correctly implement `ActionSource`, or the specified listener class does not implement `javax.faces.event.ActionListener`, throw a `JspException`. [P1-end]

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentClassicTagBase.getParentUIComponentClassicTagBase()`. If the `getCreated()` method of this instance returns `true`, check the *binding* attribute.

If *binding* is set, create a `ValueExpression` by invoking `Application.createValueExpression()` with *binding* as the *expression* argument, and `Object.class` as the *expectedType* argument. Use the `ValueExpression` to obtain a reference to the `ActionListener` instance. If there is no exception thrown, and `ValueExpression.getValue()` returned a non-null object that implements `javax.faces.event.ActionListener`, register it by calling `addActionListener()`. If there was an exception thrown, rethrow the exception as a `JspException`.

If the listener instance could not be created, check the `type` attribute. If the `type` attribute is set, instantiate an instance of the specified class, and register it by calling `addActionListener()`. If the *binding* attribute was also set, evaluate the expression into a `ValueExpression` and store the listener instance by calling `setValue()` on the `ValueExpression`. If there was an exception thrown, rethrow the exception as a `JspException`.

As an alternative to using the *binding* and/or *type* attributes, you may also register a method in a backing bean class to receive `ActionEvent` notifications, by using the `actionListener` attribute on the corresponding `UIComponent` custom action.

9.4.2 <f:attribute>

Add an attribute or `ValueExpression` on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

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

Body Content

empty.

Attributes

Name	Expr	Type	Description
name	VE	`String`	Name of the component attribute to be set
value	VE	`Object`	Value of the component attribute to be set

Constraints

- Must be nested inside a `UIComponent` custom action.

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentClassicTagBase.getParentUIComponentClassicTagBase()`. Call the `getValue()` method on the argument `name` to obtain the name of the attribute. If the associated component already has a component attribute with that name, take no action. Otherwise, call the `isLiteralText()` method on the argument `value`. If it returns `true`, store the value in the component's attribute Map under the name derived above. If it returns `false`, store the `ValueExpression` in the component's `ValueExpression` Map under the name derived above.

There is no standard implementation class for this action. It must be provided by the implementation.

9.4.3 <f:convertDateTime>

Register a `DateTimeConverter` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

```
<f:convertDateTime
  [dateStyle="{default|short|medium|long|full}"]
  [locale="{locale" | string}]
  [pattern="pattern"]
  [timeStyle="{default|short|medium|long|full}"]
  [timeZone="{timeZone| string}"]
  [type="{date|time|both}"]
  [binding="Value Expression"]/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
dateStyle	VE	String	Predefined formatting style which determines how the date component of a date string is to be formatted and parsed. Applied only if type is “date” or “both”.
locale	VE	Locale or String	Locale whose predefined styles for dates and times are used during formatting or parsing. If not specified, the Locale returned by FacesContext.getViewRoot().getLocale() will be used. Value must be either a VE expression that evaluates to a java.util.Locale instance, or a String that is valid to pass as the first argument to the constructor java.util.Locale(String language, String country). The empty string is passed as the second argument.
pattern	VE	String	Custom formatting pattern which determines how the date/time string should be formatted and parsed.
timeStyle	VE	String	Predefined formatting style which determines how the time component of a date string is to be formatted and parsed. Applied only if type is “time” or “both”.
timeZone	VE	timezone or String	Time zone in which to interpret any time information in the date string. Value must be either a VE expression that evaluates to a java.util.TimeZone instance, or a String that is a timezone ID as described in the javadocs for java.util.TimeZone.getTimeZone().
type	VE	String	Specifies whether the string value will contain a date, time, or both.
binding	VE	ValueExpression	A ValueExpression expression that evaluates to an object that implements javax.faces.convert.Converter

Constraints

- Must be nested inside a `UIComponent` custom action whose component class implements `ValueHolder`, and whose value is a `java.util.Date` (or appropriate subclass).
- If `pattern` is specified, the pattern syntax must use the pattern syntax specified by `java.text.SimpleDateFormat`.
- If `pattern` is not specified, formatted strings will contain a date value, a time value, or both depending on the specified `type`. When date or time values are included, they will be formatted according to the specified `dateStyle` and `timeStyle`, respectively.
- if `type` is not specified:
  - if `dateStyle` is set and `timeStyle` is not, `type` defaults to `date`
  - if `timeStyle` is set and `dateStyle` is not, `type` defaults to `time`
  - if both `dateStyle` and `timeStyle` are set, `type` defaults to `both`

[P1-start f:convertDateTime constraints] If this tag is not nested inside a `UIComponent` custom action, or the `UIComponent` implementation class does not correctly implement `ValueHolder`, throw a `JspException` [P1-end]

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentClassicTagBase.getParentUIComponentClassicTagBase()`. If the `getCreated()` method of this instance returns `true`, create, call `createConverter()` and register the returned Converter instance on the associated UIComponent.

[P1-start f:convertDateTime implementation requirements]The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.webapp.ConverterELTag`.
- The `createConverter()` method must:
  - If `binding` is non-null, call `getValue()` on it to obtain a reference to the `Converter` instance. If there is no exception thrown, and `binding.getValue()` returned a non-null object that implements `javax.faces.convert.Converter`, it must then cast the returned instance to `javax.faces.convert.DateTimeConverter` and configure its properties based on the specified attributes for this custom action, and return the configured instance. If there was an exception thrown, rethrow the exception as a `JspException.`
  - use the `converterId` if the converter instance could not be created from the `binding` attribute. Call the `createConverter()` method of the `Application` instance for this application, passing converter id "javax.faces.DateTime". If the *`binding`* attribute was also set, store the converter instance by calling `binding.setValue()`. It must then cast the returned instance to `javax.faces.convert.DateTimeConverter` and configure its properties based on the specified attributes for this custom action, and return the configured instance. If there was an exception thrown, rethrow the exception as a `JspException.`
- If the type attribute is not specified, it defaults as follows:
  - If dateStyle is specified but timeStyle is not specified, default to date.
  - If dateStyle is not specified but timeStyle is specified, default to time.
  - If both dateStyle and timeStyle are specified, default to both. [P1-end]

9.4.4 <f:convertNumber>

Register a `NumberConverter` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

```
<f:convertNumber
  [currencyCode="currencyCode"]
  [currencySymbol="currencySymbol"]
  [groupingUsed="{true|false}"]
  [integerOnly="{true|false}"]
  [locale="locale"]
  [maxFractionDigits="maxFractionDigits"]
  [maxIntegerDigits="maxIntegerDigits"]
  [minFractionDigits="minFractionDigits"]
  [minIntegerDigits="minIntegerDigits"]
  [pattern="pattern"]
  [type="{number|currency|percent}"]
  [binding="Value Expression"]/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
currencyCode	VE	String	ISO 4217 currency code, applied only when formatting currencies.
currencySymbol	VE	String	Currency symbol, applied only when formatting currencies.
groupingUsed	VE	boolean	Specifies whether formatted output will contain grouping separators.
integerOnly	VE	boolean	Specifies whether only the integer part of the value will be parsed.
locale	VE	java.util.Locale	Locale whose predefined styles for numbers are used during formatting or parsing. If not specified, the Locale returned by FacesContext.getViewRoot().getLocale() will be used.
maxFractionDigits	VE	int	Maximum number of digits that will be formatted in the fractional portion of the output.
maxIntegerDigits	VE	int	Maximum number of digits that will be formatted in the integer portion of the output
minFractionDigits	VE	int	Minimum number of digits that will be formatted in the fractional portion of the output.
minIntegerDigits	VE	int	Minimum number of digits that will be formatted in the integer portion of the output.
pattern	VE	String	Custom formatting pattern which determines how the number string should be formatted and parsed.
type	VE	String	Specifies whether the value will be parsed and formatted as a number, currency, or percentage.
binding	VE	ValueExpression	A ValueExpression expression that evaluates to an object that implements javax.faces.convert.Converter

Constraints

- Must be nested inside a `UIComponent` custom action whose component class implements `ValueHolder`, and whose value is a numeric wrapper class or primitive.
- If `pattern` is specified, the pattern syntax must use the pattern syntax specified by `java.text.DecimalFormat`.
- If `pattern` is not specified, formatting and parsing will be based on the specified `type`.

[P1-start f:convertNumber constraints] If this tag is not nested inside a `UIComponent` custom action, or the `UIComponent` implementation class does not correctly implement `ValueHolder`, throw a `JspException`. [P1-end]

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentClassicTagBase.getParentUIComponentClassicTagBase()`. If the `getCreated()` method of this instance returns `true`, create, call `createConverter()` and register the returned Converter instance on the associated UIComponent.

[P1-start f:convertNumber implementation] The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.webapp.ConverterELTag`.
- The `createConverter()` method must:
- If `binding` is non-null, call `binding.getValue()` to obtain a reference to the `Converter` instance. If there is no exception thrown, and `binding.getValue()` returned a non-null object that implements `javax.faces.convert.Converter`, it must then cast the returned instance to `javax.faces.convert.NumberConverter` and configure its properties based on the specified attributes for this custom action, and return the configured instance. If there was an exception thrown, rethrow the exception as a `JspException.`
- use the `converterId` if the converter instance could not be created from the `binding` attribute. Call the `createConverter()` method of the `Application` instance for this application, passing converter id "javax.faces.Number". If the *`binding`* attribute was also set, store the converter instance by calling `binding.setValue()`. It must then cast the returned instance to `javax.faces.convert.NumberConverter` and configure its properties based on the specified attributes for this custom action, and return the configured instance. If there was an exception thrown, rethrow the exception as a `JspException`. [P1-end]

9.4.5 <f:converter>

Register a named `Converter` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

```
<f:converter converterId="converterId" binding="Value Expression"/>
```

Body Content

empty

Attributes

Name	Expr	Type	Description
converterId	VE	String	Converter identifier of the converter to be created.
binding	VE	ValueExpression	A `ValueExpression` expression that evaluates to an object that implements `javax.faces.convert.Converter`

Constraints

- Must be nested inside a `UIComponent` custom action whose component class implements `ValueHolder`.
- *converterId* and/or *binding* must be specified.

[P1-start f:converter constraints] If this tag is not nested inside a `UIComponent` custom action, or the `UIComponent` implementation class does not correctly implement `ValueHolder`, throw a `JspException`. [P1-end]

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentClassicTagBase.getParentUIComponentClassicTagBase()`. If the `getCreated()` method of this instance returns `true`, create, call `createConverter()` and register the returned Converter instance on the associated UIComponent.

[P1-start f:converter implementation] The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.webapp.ConverterJspTag`.
- The `createConverter()` method must:
  - If `binding` is non-null, call `binding.getValue()` to obtain a reference to the `Converter` instance. If there is no exception thrown, and `binding.getValue()` returned a non-null object that implements `javax.faces.convert.Converter`, register it by calling `setConverter()`. If there was an exception thrown, rethrow the exception as a `JspException`. Use the `converterId` attribute if the converter instance could not be created from the `binding` attribute. If the `converterId` attribute is set, call the `createConverter()` method of the `Application` instance for this application, passing converter id specified by their converterId attribute. If the *binding* attribute was also set, store the converter instance by calling `binding.setValue()`. Register the converter instance by calling `setConverter()`. If there was an exception thrown, rethrow the exception as a `JspException`. [P1-end]

9.4.6 <f:facet>

Register a named facet (see Section 3.1.9 “Facet Management”) on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

```
<f:facet name="facet-name"/>
```

Body Content

JSP. However, only a single UIComponent custom action (and any related nested JSF custom actions) is allowed; no template text or other custom actions may be present.

Attributes

Name	Expr	Type	Description
name	NONE	String	Name of the facet to be created

Constraints

- [P1-start f:facet constraints] Must be nested inside a `UIComponent` custom action.
- Exactly one `UIComponent` custom action must be nested inside this custom action (although the nested component custom action could itself have nested children). [P1-end]

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentClassicTagBase.getParentUIComponentClassicTagBase()`. If the associated component does not already have a facet with a name specified by this custom action’s `name` attribute, create a facet with this name from the `UIComponent` custom action that is nested within this custom action.

[P1-start f:facet implementation] The implementation class must be, or extend, `javax.faces.webapp.FacetTag`. [P1-end]

9.4.7 <f:loadBundle>

Load a resource bundle localized for the locale of the current view, and expose it (as a Map) in the request attributes for the current request.

Syntax

```
<f:loadBundle basename="resource-bundle-name" var="attributeKey"/>
```

Body Content

empty

Attributes

Name	Expr	Type	Description
basename	VE	`String`	Base name of the resource bundle to be loaded.
var	NONE	String	Name of a request scope attribute under which the resource bundle will be exposed as a Map.

Constraints

- [P1-start f:loadBundle constraints] Must be nested inside an `<f:view>` custom action. [P1-end]

Description

Load the resource bundle specified by the `basename` attribute, localized for the Locale of the `UIViewRoot` component of the current view, and expose its key-values pairs as a `Map` under the attribute key specified by the `var` attribute. In this way, value binding expressions may be used to conveniently retrieve localized values. If the named bundle is not found, throw `JspException`.

If the `get()` method for the `Map` instance exposed by this custom action is passed a key value that is not present (that is, there is no underlying resource value for that key), the literal string "???foo???" (where "foo" is replaced by the key the String representation of the key that was requested) must be returned, rather than the standard `Map` contract return value of `null`.

9.4.8 <f:param>

Add a child `UIParameter` component to the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

Syntax 1: Unnamed value

```
<f:param [id="componentIdOrImmediateExpression"] value="parameter-value"
  [binding="componentReference"]/>
```

Syntax 2: Named value

```
<f:param [id="componentIdOrImmediateExpression"]
  [binding="componentReference"]
  name="parameter-name" value="parameter-value"/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
binding	VE	ValueExpression	ValueExpression expression to a backing bean property bound to the component instance for the UIComponent created by this custom action
id	NONE	String	Component identifier of a `UIParameter` component
name	VE	String	Name of the parameter to be set
value	VE	String	Value of the parameter to be set

Constraints

- [P1-start f:param constraints] Must be nested inside a `UIComponent` custom action. [P1-end]

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentClassicTagBase.getParentUIComponentClassicTagBase()`. If the `getCreated()` method of this instance returns `true`, create a new `UIParameter` component, and attach it as a child of the associated `UIComponent`. It is up to the parent `UIComponent` to determine how it will handle its `UIParameter` children.

[P1-start f:param implementation] The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.UIComponentELTag`.
- The `getComponentType()` method must return "`Parameter`".
- The `getRendererType()` method must return `null`. [P1-end]

9.4.9 <f:phaseListener>

Register a `PhaseListener` instance on the `UIViewRoot` associated with the closest parent `UIViewRoot` custom action.

Syntax

```
<f:phaseListener type="fully-qualified-classname"
binding="Value expression"/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
type	VE	`String`	Fully qualified Java class name of an `PhaseListener` to be created and registered
binding	VE	`ValueExpression`	A `ValueExpression` expression that evaluates to an object that implements `javax.faces.event.PhaseListener`

Constraints

- [P1-start f:phaseListener constraints] Must be nested inside a `UIViewRoot` custom action.
- The specified listener class must implement `javax.faces.event.PhaseListener`.
- *type* and/or *binding* must be specified. [P1-end]

Description

Locate the one and only `UIViewRoot` custom action instance by walking up the tag tree until you find a `UIComponentTagBase` instance that has no parent. If the `getCreated()` method of this instance returns `true`, check the *binding* attribute.

If *binding* is set, call *binding.getValue()* to obtain a reference to the `PhaseListener` instance. If there is no exception thrown, and `binding.getValue()` returned a non-null object that implements `javax.faces.event.PhaseListener`, register it by calling `addPhaseListener()`. If there was an exception thrown, rethrow the exception as a `JspException`.

If the listener instance could not be created, check the `type` attribute. If the `type` attribute is set, instantiate an instance of the specified class, and register it by calling `addPhaseListener()`. If the *binding* attribute was also set, store the listener instance by calling `binding.setValue()`. If there was an exception thrown, rethrow the exception as a `JspException`.

9.4.10 <f:selectItem>

Add a child `UISelectItem` component to the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

Syntax 1: Directly Specified Value

```
<f:selectItem [id="componentIdOrImmediateExpression"]
  [binding="componentReference"]
  [itemDisabled="{true|false}"]
  itemValue="itemValue"
  itemLabel="itemLabel"
  [itemDescription="itemDescription"]/>
```

Syntax 2: Indirectly Specified Value

```
<f:selectItem [id="componentIdOrImmediateExpression"]
  [binding="componentReference"]
  value="selectItemValue"/>
```

Body Content

empty

Attributes

Name	Expr	Type	Description
binding	VE	`ValueExpression`	`ValueExpression` expression to a backing bean property bound to the component instance for the UIComponent created by this custom action.
`id`	`NONE`	`String`	Component identifier of a `UISelectItem` component.
`itemDescription`	VE	`String`	Description of this option (for use in development tools).
itemDisabled	VE	boolean	Flag indicating whether the option created by this component is disabled.
itemLabel	VE	String	Label to be displayed to the user for this option.
itemValue	VE	Object	Value to be returned to the server if this option is selected by the user.
value	VE	javax.faces.model.SelectItem	Value binding pointing at a SelectItem instance containing the information for this option.
escape	VE	boolean	ValueExpression pointing to a boolean that tells whether or not the label of this selectItem should be escaped per HTML rules. Default is true.

Constraints

- [P1-start f:selectItem constraints] Must be nested inside a `UIComponent` custom action that creates a `UISelectMany` or `UISelectOne` component instance.[P1-end]

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentClassicTagBase.getParentUIComponentClassicTagBase()`. If the `getCreated()` method of this instance returns `true`, create a new `UISelectItem` component, and attach it as a child of the associated `UIComponent`.

[P1-start f:selectItem implementation] The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.UIComponentELTag`.
- The `getComponentType()` method must return "`SelectItem`".
- The `getRendererType()` method must return `null`.[P1-end]

9.4.11 <f:selectItems>

Add a child `UISelectItems` component to the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

```
<f:selectItems [id="componentIdOrImmediateExpression"]
  [binding="componentReference"]
  value="selectItemsValue"/>
```

Body Content

empty

Attributes

Name	Expr	Type	Description
binding	VE	`ValueExpression`	`ValueExpression` expression to a backing bean property bound to the component instance for the UIComponent created by this custom action.
`id`	`NONE`	`String`	Component identifier of a `UISelectItem` component.
`value`	VE	javax.faces.model.SelectItem, see description for specific details	Value binding expression pointing at one of the following instances: 1. an individual javax.faces.model.SelectItem 2. a java language array of javax.faces.model.SelectItem 3. a java.util.Collection of javax.faces.model.SeleccItem 4. A java.util.Map where the keys are converted to Strings and used as labels, and the corresponding values are converted to Strings and used as values for newly created javax.faces.model.SelectItem instances. The instances are created in the order of the iterator over the keys provided by the Map.

Constraints

- Must be nested inside a `UIComponent` custom action that creates a `UISelectMany` or `UISelectOne` component instance.

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentClassicTagBase.getParentUIComponentClassicTagBase()`. If the `getCreated()` method of this instance returns `true`, create a new `UISelectItems` component, and attach it as a child of the associated `UIComponent`.

[P1-start f:selectItems implementation]The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.UIComponentELTag`.
- The `getComponentType()` method must return "`javax.faces.SelectItems`".
- The `getRendererType()` method must return `null`. [P1-end]

9.4.12 <f:setPropertyActionListener>

Tag implementation that creates a special `ActionListener` instance and registers it on the `ActionSource` associated with our most immediate surrounding instance of a tag whose implementation class is a subclass of `UIComponentTag`. This tag creates no output to the page currently being created. This tag is useful for pushing a specific value into a managed bean on page submit.

Syntax

```
<f:setPropertyActionListener target="Value Expression" value="value Expression"/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
value	VE	ValueExpression	The `ValueExpression` from which the value is taken.
target	VE	ValueExpression	The `ValueExpression` into which the evaluated value from the "`value`" attribute is stored when the listener executes.

Constraints

- Must be nested inside a `UIComponent` custom action.
- The corresponding `UIComponent` implementation class must implement `ActionSource`, and therefore define a public `addActionListener()` method that accepts an `ActionListener` parameter.
- The tag implementation must only create and register the `ActionListener` instance the first time the component for this tag is created
- When the listener executes:
  - Call `getValue()` on the "`value`" `ValueExpression`.
  - If value of the "value" expression is null, call `setValue()`on the "target" `ValueExpression` with the null value.
  - If the value of the "value" expression is not null, call `getType()`on the "value" and "target" `ValueExpressions` to determine their property types.
  - Coerce the value of the "value" expression to the "target" expression value type following the Expression Language coercion rules. Call `setValue()`on the "target" `ValueExpression` with the resulting value.
  - If either conversion or the execution of setValue() fails throw an AbortProcessingException.
- This tag creates no output to the page currently being created. It is used solely for the side effect of `ActionListener` creation and addition.

[P1-start f:setPropertyActionListener constraints]If this tag is not nested inside a `UIComponent` custom action, or the `UIComponent` implementation class does not correctly implement `ActionSource`, or the specified listener class does not implement `javax.faces.event.ActionListener`, throw a `JspException`. [P1-end]

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentClassicTagBase.getParentUIComponentClassicTagBase()`. If the `getCreated()` method of this instance returns `true` return `SKIP_BODY`.

Create an instance of `ActionListener` that implements `StateHolder` and stores the `target` and `value` `ValueExpression` instances as instance variables included in the state saving contract. The `processAction()` method of the listener must call `getValue()` on the `value` `ValueExpression` and convert the value before passing the result to a call to `setValue()` on the `target` `ValueExpression`.

9.4.13 <f:subview>

Container action for all JSF core and component custom actions used on a nested page included via `<jsp:include>` or any custom action that dynamically includes another page from the same web application, such as JSTL's `<c:import>`.

Syntax

```
<f:subview id="componentIdOrImmediateExpression"
               [binding="componentReference"]
               [rendered="{true|false}"]>
   Nested template text and custom actions
</f:subview>
```

Body Content

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

Attributes

Name	Expr	Type	Description
binding	VE	`ValueExpression`	`ValueExpression` expression to a backing bean property bound to the component instance for the UIComponent created by this custom action.
id	NONE	`String`	Component identifier of a `UINamingContainer` component
`rendered`	VE	Boolean	Whether or not this subview should be rendered.

Constraints

- [P1-start f:subview constraints] Must be nested inside a `<f:view>` custom action (although this custom action might be in a page that is including the page containing the `<f:subview>` custom action.
- Must not contain an `<f:view>` custom action.
- Must have an `id` attribute whose value is unique within the scope of the parent naming container. If this constraint is not met, the action taken regarding id uniqueness in section Section 9.3 "UIComponent Custom Action Implementation Requirements" must be taken
- May be placed in a parent page (with `<jsp:include>` or `<c:import>` nested inside), or within the nested page. [P1-end]

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentClassicTagBase.getParentUIComponentClassicTagBase()`. If the `getCreated()` method of this instance returns `true`, create a new `UINamingContainer` component, and attach it as a child of the

associated `UIComponent`. Such a component provides a scope within which child component identifiers must still be unique, but allows child components to have the same simple identifier as child components nested in some other naming container. This is useful in several scenarios:

```
“main.jsp”
<f:view>
  <c:import url=”foo.jsp”/>
  <c:import url=”bar.jsp”/>
</f:view>

“foo.jsp”
<f:subview id=”aaa”>
  ... components and other content ...
</f:subview>

“bar.jsp”
<f:subview id=”bbb”>
  ... components and other content ...
</f:subview>
```

In this scenario, `<f:subview>` custom actions in imported pages establish a naming scope for components within those pages. Identifiers for `<f:subview>` custom actions nested in a single `<f:view>` custom action must be unique, but it is difficult for the page author (and impossible for the JSP page compiler) to enforce this restriction.

```
“main.jsp”
<f:view>
  <f:subview id=”aaa”>
    <c:import url=”foo.jsp”/>
  </f:subview>
  <f:subview id=”bbb”>
    <c:import url=”bar.jsp”/>
  </f:subview>
</f:view>

“foo.jsp”
... components and other content ...

“bar.jsp”
... components and other content ...
```

In this scenario, the `<f:subview>` custom actions are in the including page, rather than the included page. As in the previous scenario, the “id” values of the two subviews must be unique; but it is much easier to verify using this style.

It is also possible to use this approach to include the same page more than once, but maintain unique identifiers:

```
“main.jsp”
<f:view>
  <f:subview id=”aaa”>
    <c:import url=”foo.jsp”/>
  </f:subview>
  <f:subview id=”bbb”>
    <c:import url=”foo.jsp”/>
  </f:subview>
</f:view>

“foo.jsp”
... components and other content ...
```

In all of the above examples, note that `foo.jsp` and `bar.jsp` may not contain `<f:view>`.

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

- [P1-start f:subview implementation] Must extend `javax.faces.UIComponentELTag`.
- The `getComponentType()` method must return “`NamingContainer`”.
- The `getRendererType()` method must return `null`. [P1-end]

9.4.14 <f:validateDoubleRange>

Register a `DoubleRangeValidator` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

Syntax 1: Maximum only specified

```
<f:validateDoubleRange maximum="543.21" binding="VB Expression"/>
```

Syntax 2: Minimum only specified

```
<f:validateDoubleRange minimum="123.45" binding="VB Expression"/>
```

Syntax 3: Both maximum and minimum are specified

```
<f:validateDoubleRange maximum="543.21" minimum="123.45" binding="VB Expression"/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
`maximum`	VE	`double`	Maximum value allowed for this component
`minimum`	VE	`double`	Minimum value allowed for this component
`binding`	VE	`ValueExpression`	A `ValueExpression` expression that evaluates to an object that implements `javax.faces.convert.Validator`

Constraints

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

[P1-start f:validateDoubleRange constraints] If this tag is not nested inside a `UIComponent` custom action, or the `UIComponent` implementation class does not correctly implement `EditableValueHolder` throw a `JspException`. [P1-end]

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentClassicTagBase.getParentUIComponentClassicTagBase()`. If the `getCreated()` method of this instance returns `true`, create, call `createValidator()` and register the returned `Validator` instance on the associated `UIComponent`.

[P1-start f:validateDoubleRange implementation] The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.webapp.ValidatorELTag`.
- The `createValidator()` method must:

- If `binding` is non-null, create a `ValueBinding` by invoking `Application.createValueExpression()` with *`binding`* as the *`expression`* argument, and `Validator.class` as the *`expectedType`* argument.use the `ValueBinding` to obtain a reference to the `Validator` instance. If there is no exception thrown, and `ValueExpression.getValue()` returned a non-null object that implements `javax.faces.validator.Validator`, it must then cast the returned instance to `javax.faces.validator.DoubleRangeValidator` and configure its properties based on the specified attributes for this custom action, and return the configured instance. If there was an exception thrown, rethrow the exception as a `JspException.`

- use the `validatorId` if the validator instance could not be created from the `binding` attribute. Call the `createValidator()` method of the `Application` instance for this application, passing validator id "javax.faces.DoubleRange". If the *`binding`* attribute was also set, evaluate the expression into a `ValueExpression` and store the validator instance by calling `setValue()` on the `ValueExpression`. It must then cast the returned instance to `javax.faces.validator.DoubleRangeValidator` and configure its properties based on the specified attributes for this custom action, and return the configured instance. If there was an exception thrown, rethrow the exception as a `JspException`. [P1-end]

9.4.15 <f:validateDoubleRange>

Register a `DoubleRangeValidator` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

Syntax 1: Maximum only specified

```
<f:validateDoubleRange maximum="543.21" binding="VB Expression"/>
```

Syntax 2: Minimum only specified

```
<f:validateDoubleRange minimum="123.45" binding="VB Expression"/>
```

Syntax 3: Both maximum and minimum are specified

```
<f:validateDoubleRange maximum="543.21" minimum="123.45" binding="VB Expression"/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
`maximum`	VE	`double`	Maximum value allowed for this component
`minimum`	VE	`double`	Minimum value allowed for this component
`binding`	VE	`ValueExpression`	A `ValueExpression` expression that evaluates to an object that implements `javax.faces.convert.Validator`

Constraints

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

[P1-start f:validateDoubleRange constraints] If this tag is not nested inside a `UIComponent` custom action, or the `UIComponent` implementation class does not correctly implement `EditableValueHolder` throw a `JspException`. [P1-end]

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentClassicTagBase.getParentUIComponentClassicTagBase()`. If the `getCreated()` method of this instance returns `true`, create, call `createValidator()` and register the returned `Validator` instance on the associated `UIComponent`.

[P1-start f:validateDoubleRange implementation] The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.webapp.ValidatorELTag`.
- The `createValidator()` method must:

- If `binding` is non-null, create a `ValueBinding` by invoking `Application.createValueExpression()` with *`binding`* as the *`expression`* argument, and `Validator.class` as the *`expectedType`* argument.use the `ValueBinding` to obtain a reference to the `Validator` instance. If there is no exception thrown, and `ValueExpression.getValue()` returned a non-null object that implements `javax.faces.validator.Validator`, it must then cast the returned instance to `javax.faces.validator.DoubleRangeValidator` and configure its properties based on the specified attributes for this custom action, and return the configured instance. If there was an exception thrown, rethrow the exception as a `JspException.`
- use the `validatorId` if the validator instance could not be created from the `binding` attribute. Call the `createValidator()` method of the `Application` instance for this application, passing validator id "javax.faces.DoubleRange". If the *`binding`* attribute was also set, evaluate the expression into a `ValueExpression` and store the validator instance by calling `setValue()` on the `ValueExpression`. It must then cast the returned instance to `javax.faces.validator.DoubleRangeValidator` and configure its properties based on the specified attributes for this custom action, and return the configured instance. If there was an exception thrown, rethrow the exception as a `JspException`. [P1-end]

9.4.16 <f:validateRegex>

Register a `RegexValidator` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

```
<f:validateRegex pattern="a*b"/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
`pattern`	`VE`	`String`	The string to be interpreted as a `java.util.regex.Pattern`
`binding`	`VE`	`ValueExpression`	A `ValueExpression` expression that evaluates to an object that implements `javax.faces.convert.Validator`

Constraints

- Must be nested inside a `EditableValueHolder` custom action whose value is a `String`.
- Must specify either the `pattern` attribute.

[P1-start f:validateLength constraints] If this tag is not nested inside a `UIComponent` custom action, or the `UIComponent` implementation class does not correctly implement `EditableValueHolder`, throw a `JspException`. [P1-end]

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentClassicTagBase.getParentUIComponentClassicTagBase()`. If the `getCreated()` method of this instance returns `true`, create, call `createValidator()` and register the returned `Validator` instance on the associated `UIComponent`.

[P1-start f:validateLength implementation] The implementation class for this action must meet the following requirements:

- Must extend `javax.faces.webapp.ValidatorELTag`.
- The `createValidator()` method must:
  - If `binding` is non-null, create a `ValueExpression` by invoking `Application.createValueExpression()` with *`binding`* as the *`expression`* argument, and `Validator.class` as the *`expectedType`* argument.use the `ValueExpression` to obtain a reference to the `Validator` instance. If there is no exception thrown, and `ValueExpression.getValue()` returned a non-null object that implements `javax.faces.validator.Validator`, it must then cast the returned instance to `javax.faces.validator.RegexValidator` and configure its properties based on the specified attributes for this custom action, and return the configured instance. If there was an exception thrown, rethrow the exception as a `JspException.`
  - use the `validatorId` if the validator instance could not be created from the `binding` attribute. Call the `createValidator()` method of the `Application` instance for this application, passing validator id "javax.faces.RegularExpression". If the *`binding`* attribute was also set, evaluate the expression into a `ValueExpression` and store the validator instance by calling `setValue()` on the `ValueExpression`. It must then cast the returned instance to `javax.faces.validator.RegexValidator` and configure its properties based on the specified attributes for this custom action, and return the configured instance. If there was an exception thrown, rethrow the exception as a `JspException`.[P1-end]

9.4.17 <f:validateLongRange>

Register a `LongRangeValidator` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

Syntax 1: Maximum only specified

```
<f:validateLongRange maximum="543" binding="VB Expression"/>
```

Syntax 2: Minimum only specified

```
<f:validateLongRange minimum="123" binding="VB Expression"/>
```

Syntax 3: Both maximum and minimum are specified

```
<f:validateLongRange maximum="543" minimum="123" binding="VB Expression"/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
`maximum`	VE	`long`	Maximum value allowed for this component
`minimum`	VE	`long`	Minimum value allowed for this component
`binding`	VE	`ValueExpression`	A `ValueExpression` expression that evaluates to an object that implements `javax.faces.convert.Validator`

Constraints

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

[P1-start f:validateLongeRange constraints] If this tag is not nested inside a `UIComponent` custom action, or the `UIComponent` implementation class does not correctly implement `EditableValueHolder`, throw a `JspException`. [P1-end]

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentClassicTagBase.getParentUIComponentClassicTagBase()`. If the `getCreated()` method of this instance returns `true`, create, call `createValidator()` and register the returned `Validator` instance on the associated `UIComponent`.

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

- Must extend `javax.faces.webapp.ValidatorELTag`.
- The `createValidator()` method must:

- If `binding` is non-null, create a `ValueExpression` by invoking `Application.createValueExpression()` with *`binding`* as the *`expression`* argument, and `Validator.class` as the *`expectedType`* argument. Use the `ValueExpression` to obtain a reference to the `Validator` instance. If there is no exception thrown, and `ValueExpression.getValue()` returned a non-null object that implements `javax.faces.validator.Validator`, it must then cast the returned instance to `javax.faces.validator.LongRangeValidator` and configure its properties based on the specified attributes for this custom action, and return the configured instance. If there was an exception thrown, rethrow the exception as a `JspException.`
- use the `validatorId` if the validator instance could not be created from the `binding` attribute. Call the `createValidator()` method of the `Application` instance for this application, passing validator id "javax.faces.LongRange". If the *`binding`* attribute was also set, evaluate the expression into a `ValueExpression` and store the validator instance by calling `setValue()` on the `ValueExpression`. It must then cast the returned instance to `javax.faces.validator.LongRangeValidator` and configure its properties based on the specified attributes for this custom action, and return the configured instance. If there was an exception thrown, rethrow the exception as a `JspException.`

9.4.18 <f:validator>

Register a named `Validator` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

```
<f:validator validatorId="validatorId" binding="VB Expression"/>
```

Body Content

empty

Attributes

Name	Expr	Type	Description
validatorId	VE	String	Validator identifier of the validator to be created.
binding	VE	ValueExpression	A ValueExpression expression that evaluates to an object that implements javax.faces.convert.Validator

Constraints

- Must be nested inside a `UIComponent` custom action whose component class implements `EditableValueHolder`.
- *validatorId* and/or *binding* must be specified.

[P1-start f:validator constraints 2] If this tag is not nested inside a `UIComponent` custom action, or the `UIComponent` implementation class does not correctly implement `EditableValueHolder` throw a `JspException`. [P1-end]

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentClassicTagBase.getParentUIComponentClassicTagBase()`. If the `getCreated()` method of this instance returns `true`, create, call `createValidator()` and register the returned `Validator` instance on the associated `UIComponent`.

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

- Must extend `javax.faces.webapp.ValidatorJspTag`.
- The `createValidator()` method must:
  - If `binding` is non-null, call `binding.getValue()` to obtain a reference to the `Validator` instance. If there is no exception thrown, and `binding.getValue()` returned a non-null object that implements `javax.faces.validator.Validator`, register it by calling `addValidator()`. If there was an exception thrown, rethrow the exception as a `JspException`.
  - use the `validatorId` attribute if the validator instance could not be created from the `binding` attribute. If the `validatorId` attribute is set, call the `createValidator()` method of the `Application` instance for this application, passing validator id specified by their validatorId attribute. If the *binding* attribute was also set, store the validator instance by calling `binding.setValue()`. Register the validator instance by calling `addValidator()`. If there was an exception thrown, rethrow the exception as a `JspException`.

9.4.19 <f:valueChangeListener>

Register a `ValueChangeListener` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action.

Syntax

```
<f:valueChangeListener type="fully-qualified-classname" binding="VB Expression"/>
```

Body Content

empty.

Attributes

Name	Expr	Type	Description
type	VE	String	Fully qualified Java class name of a `ValueChangeListener` to be created and registered
binding	VE	ValueExpression	A `ValueExpression` expression that evaluates to an object that implements `javax.faces.event.ValueChangeListener`

Constraints

- Must be nested inside a `UIComponent` custom action.
- The corresponding `UIComponent` implementation class must implement `EditableValueHolder`, and therefore define a public `addValueChangeListener()` method that accepts an `ValueChangeListener` parameter.
- The specified listener class must implement `javax.faces.event.ValueChangeListener`.
- *type* and/or *binding* must be specified.

[P1-start f:valueChangeListener constraints] If this tag is not nested inside a `UIComponent` custom action, or the `UIComponent` implementation class does not correctly implement `EditableValueHolder`, or the specified listener class does not implement `javax.faces.event.ValueChangeListener`, throw a `JspException`. [P1-end]

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentClassicTagBase.getParentUIComponentClassicTagBase()`. If the `getCreated()` method of this instance returns `true`, check the *binding* attribute.

If binding is non-null, call `binding.getValue()` to obtain a reference to the `ValueChangeListener` instance. If there is no exception thrown, and `ValueExpression.getValue()` returned a non-null object that implements `javax.faces.event.ValueChangeListener`, register it by calling `addValueChangeListener()`. If there was an exception thrown, rethrow the exception as a `JspException`.

If the listener instance could not be created, check the `type` attribute. If the `type` attribute is set, instantiate an instance of the specified class, and register it by calling `addValueChangeListener()`. If the *binding* attribute was also set, store the listener instance by calling `binding.setValue()`. If there was an exception thrown, rethrow the exception as a `JspException`.

As an alternative to using the *binding* and/or *type* attributes, you may also register a method in a backing bean class to receive `ValueChangeEvent` notifications, by using the `valueChangeListener` attribute on the corresponding `UIComponent` custom action.instantiate an instance of the specified class, and register it by calling `addValueChangeListener()`.

9.4.20 <f:verbatim>

Register a child `UIOutput` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action which renders nested body content.

Syntax

```
<f:verbatim [escape="{true|false}" rendered="{true|false"]/>
```

Body Content

JSP. However, no UIComponent custom actions, or custom actions from the JSF Core Tag Library, may be nested inside this custom action.

Attributes

Name	Expr	Type	Description
escape	VE	boolean	If `true`, generated markup is escaped in a manner appropriate for the markup language being rendered. Default value is false.
rendered	VE	boolean	Flag indicating whether or not this component should be rendered (during Render Response Phase), or processed on any subsequent form submit. Default value is true.

Constraints

- [P1-start f:verbatim constraints] Must be implemented as a `UIComponentBodyTag`.[P1-end]

Description

Locate the closest parent `UIComponent` custom action instance by calling `UIComponentClassicTagBase.getParentUIComponentClassicTagBase()`. If the `getCreated()` method of this instance returns `true`, creates a new `UIOutput` component, and add it as a child of the `UIComponent` associated with the located instance. The `rendererType` property of this `UIOutput` component must be set to "javax.faces.Text", and the `transient` property must be set to `true`. Also, the value (or value binding, if it is an expression) of the `escape` attribute must be passed on to the renderer as the value the `escape` attribute on the `UIOutput` component.

9.4.21 <f:view>

Container for all JSF core and component custom actions used on a page.

Syntax

```
<f:view [locale="locale" renderKitId="alternate"]
        [beforePhase="methodExpression"]
        [afterPhase"methodExpression"]>
   Nested template text and custom actions
</f:view>
```

Body Content

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

Attributes

Name	Expr	Type	Description
renderKitId	VE	String	The identifier for the render kit to use for rendering this page.
locale	VE	String or Locale	Name of a Locale to use for localizing this page (such as en_uk), or value binding expression that returns a Locale instance
beforePhase	ME	String	MethodExpression expression that points to a method whose signature is that of javax.faces.event.PhaseListener.beforePhase()
afterPhase	ME	String	MethodExpression expression that points to a method whose signature is that of javax.faces.event.PhaseListener.afterPhase()

Constraints

- [P1-start f:view constraints] Any JSP-created response using actions from the JSF Core Tag Library, as well as actions extending `javax.faces.webapp.UIComponentELTag` from other tag libraries, must be nested inside an occurrence of the `<f:view>` action.
- JSP page fragments included via the standard `<%@ include %>` directive need not have their JSF actions embedded in a `<f:view>` action, because the included template text and custom actions will be processed as part of the outer page as it is compiled, and the `<f:view>` action on the outer page will meet the nesting requirement.
- If the `renderKitId` attribute is present, its value is stored in `UIViewRoot`. If the `renderKitId` attribute is not present, then the default render kit identifier as returned by `Application.getDefaultRenderKitId()` is stored in `UIViewRoot` if it is not `null`. Otherwise, the render kit identifier as specified by the constant `RenderKitFactory.HTML_BASIC_RENDER_KIT` is stored in `UIViewRoot`. Specifying a `renderKitId` for the current view also affects all subsequent views, unless overridden by another use of the `renderKitId` attribute. Please refer to Section 7.5 "ViewHandler" for more information.
- If the `locale` attribute is present, its value overrides the `Locale` stored in `UIViewRoot`, normally set by the `ViewHandler, and the doStartTag() method must store it by calling UIViewRoot.setLocale()`.

- The `doStartTag()` method must call `javax.servlet.jsp.jstl.core.Config.set()`, passing the `ServletRequest` instance for this request, the constant `javax.servlet.jsp.jstl.core.Config.FMT_LOCALE`, and the `Locale` returned by calling `UIViewRoot.getLocale()`. [P1-end]

Description

Provides the JSF implementation a convenient place to perform state saving during the *render response* phase of the request processing lifecycle, if the implementation elects to save state as part of the response.

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

- Must extend `javax.faces.UIComponentELTag`.
- The `getComponentType()` method must return "ViewRoot".
- The `getRendererType()` method must return `null`.

Please refer to the javadocs for `javax.faces.application.StateManager` for details on what the tag handler for this tag must do to implement state saving.

9.5 Standard HTML RenderKit Tag Library

All JSF implementations must provide a tag library containing actions that correspond to each valid combination of a supported component class (see Chapter 4 “Standard User Interface Components”) and a `Renderer` from the Standard HTML RenderKit (see Section 8.6 “Standard HTML RenderKit Implementation”) that supports that component type. [P1-start html_basic taglib requirements] The tag library descriptor for this tag library must meet the following requirements:

- Must declare a tag library version (`<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).[P1-end]

[P1-start html_basic return values]The custom actions defined in this tag library must specify the following return values for the `getComponentType()` and `getRendererType()` methods, respectively:.

TABLE 9-2 Standard HTML RenderKit Tag Library

getComponentType()	getRendererType()	custom action name
javax.faces.Column	(null)*	column
javax.faces.HtmlCommandButton	javax.faces.Button	commandButton
javax.faces.HtmlCommandLink	javax.faces.Link	commandLink
javax.faces.HtmlDataTable	javax.faces.Table	dataTable
javax.faces.HtmlForm	javax.faces.Form	form
javax.faces.HtmlGraphicImage	javax.faces.Image	graphicImage
javax.faces.HtmlInputHidden	javax.faces.Hidden	inputHidden
javax.faces.HtmlInputSecret	javax.faces.Secret	inputSecret
javax.faces.HtmlInputText	javax.faces.Text	inputText
javax.faces.HtmlInputTextarea	javax.faces.Textarea	inputTextarea
javax.faces.HtmlMessage	javax.faces.Message	message
javax.faces.HtmlMessages	javax.faces.Messages	messages
javax.faces.HtmlOutputFormat	javax.faces.Format	outputFormat
javax.faces.HtmlOutputLabel	javax.faces.Label	outputLabel
javax.faces.HtmlOutputLink	javax.faces.Link	outputLink
javax.faces.Output	javax.faces.Body	body
javax.faces.Output	javax.faces.Head	head
javax.faces.Output	javax.faces.resource.Script	outputScript

TABLE 9-2 Standard HTML RenderKit Tag Library

getComponentType()	getRendererType()	custom action name
javax.faces.Output	javax.faces.resource.Stylesheet	outputStylesheet
javax.faces.HtmlOutputText	javax.faces.Text	outputText
javax.faces.HtmlPanelGrid	javax.faces.Grid	panelGrid
javax.faces.HtmlPanelGroup	javax.faces.Group	panelGroup
javax.faces.HtmlSelectBooleanCheckbox	javax.faces.Checkbox	selectBooleanCheckbox
javax.faces.HtmlSelectManyCheckbox	javax.faces.Checkbox	selectManyCheckbox
javax.faces.HtmlSelectManyListbox	javax.faces.Listbox	selectManyListbox
javax.faces.HtmlSelectManyMenu	javax.faces.Menu	selectManyMenu
javax.faces.HtmlSelectOneListbox	javax.faces.Listbox	selectOneListbox
javax.faces.HtmlSelectOneMenu	javax.faces.Menu	selectOneMenu
javax.faces.HtmlSelectOneRadio	javax.faces.Radio	selectOneRadio

* This component has no associated Renderer, so the getRendererType() method must return null instead of a renderer type.

[P1-end] [P1-start html_basic taglibrary requirements 2]The tag library descriptor for this tag library (and the corresponding tag handler implementation classes) must meet the following requirements:

- The attributes for the tags, both in the TLD and in the associated tag handlers, must conform exactly to the type, name, and description given in the TLDDocs for the html_basic tag library.
- If the type of the attribute is `javax.el.ValueExpression`, the TLD for the attribute must contain a `<deferred-value>` with a nested `<type>` element, inside of which is nested the expected type, as given in the TLDDocs. The JavaBeans setter method in the tag handler for the tag must be of type `javax.el.ValueExpression`.
- If the type of the attribute is `javax.el.MethodExpression`, the TLD for the attribute must contain a `<deferred-method>` with a nested `<method-signature>`, inside of which is the method signature for that `MethodExpression`, as given in the TLDDocs. The actual name of the method in the signature declaration is immaterial and unspecified. The JavaBeans setter method in the tag handler for the tag must be of type `javax.el.MethodExpression`.
- Any attributes listed in the TLDDocs with a `request-time` value of `true` must specify an `<rtexprvalue>` of `true` in the TLD.
- The following action must be taken to handle the value of the `converter` property. If `isLiteralText()` on the `converter` property returns `true`, get the value of the property and treat it as a `converterId` by passing it as the argument to the `createConverter()` method of the `Application` instance for this webapp, then pass the created `Converter` to the `setConverter()` method of the component for this tag. If `isLiteralText()` on the `converter` property returns `false`, call `setValueExpression()` on the component, passing "converter" as the name of the `ValueExpression` and the `ValueExpression` instance as the value.
- For a non-`null` `action` attribute on custom actions related to `ActionSource2` components (`commandButton`, `commandLink`), the `setProperties()` method of the tag handler implementation class must pass the value of the action attribute, which is a `MethodExpression`, to the component's `setActionExpression()` method.

- For other non-`null` attributes that correspond to `MethodExpression` attributes on the underlying components (`actionListener`, `validator`, `valueChangeListener`), the `setProperties()` method of the tag handler implementation class must store that instance as the value of the corresponding component property.
- For any non-`null` `id`, `scope`, or `var` attribute, the `setProperties()` method of the tag handler implementation class must simply set the value of the corresponding component attribute.
- For all other non-`null` attributes, the `setProperties()` of the tag handler implementation class method must:
  - If the `attribute.isLiteralText()` returns `true`, set the corresponding attribute on the underlying component (after performing any necessary type conversion).
  - Otherwise, call the `setValueExpression()` method on the underlying component, passing the attribute name and the `ValueExpression‘` instance as parameters.[P1-end]

10

Facelets and its use in Web Applications

As of version 2 of this specification, JavaServer Faces implementations must support (although JSF-based applications need not utilize) using Facelets as the view declaration language for JSF pages. Facelets technology was created by JSR-252 EG Member Jacob Hookom.

10.1 Non-normative Background

To aid implementors in providing a spec compliant runtime for Facelets, this section provides a non-normative background to motivate the discussion of the Facelets feature. Facelets is a replacement for JSP that was designed from the outset with JSF in mind. New features introduced in version 2 and later are only exposed to page authors using Facelets. JSP is retained for backwards compatibility.

10.1.1 Differences between JSP and Facelets

Facelets was the first non-JSP page declaration language designed for JavaServer Faces. As such, Facelets was able to provide a simpler and more powerful programming model to JSF developers than that provided by JSP, largely by leveraging JSF as much as possible without carrying backwards compatibility with JSP. The following table lists some of the differences between Facelets and JSP

TABLE 10-1 Comparison of Facelets and JSP

Feature Name	JSP	Facelets
Pages are compiled to...	A Servlet that gets executed each time the page renders. The UIComponent hierarchy is built by the presence of custom tags in the page.	An abstract syntax tree that, when executed, builds a UIComponent hierarchy.
Handling of tag attributes	All tag attributes must be declared in a TLD file. Conformance instances of components in a page with the expected attributes can be enforced with a taglibrary validator.	Tag attributes are completely dynamic and automatically map to properties, attributes and ValueExpressions on UIComponent instances
Page templating	Not supported, must go outside of core JSP	Page templating is a core feature of Facelets

TABLE 10-1 Comparison of Facelets and JSP

Feature Name	JSP	Facelets
Performance	Due to the common implementation technique of compiling a JSP page to a Servlet, performance can be slow	Facelets is simpler and faster than JSP
EL Expressions	Expressions in template text cause unexpected behavior when used in JSP	Expressions in template text operate as expected.
JCP Standard	Yes, the specification is separate from the implementation for JSP	No, the specification is defined by and is one with the implementation.

10.1.2 Differences between Pre JSF 2.0 Facelets and Facelets in JSF 2.0

The work of taking a snapshot of a version of Facelets and producing the specification for Facelets in JSF 2.0 consists of extracting the parts of Facelets that are intended to be "public" and leaving the rest as implementation details. A decision was made early in this process to strive for backwards compatibility between the latest popular version of Facelets and Facelets in JSF 2.0. The sole determinant to backwards compatibility lies in the answer to the question, "is there any Java code in the application, or in libraries used by the application, that extends from or depends on any class in package `com.sun.facelets` and/or its sub-packages?"

- If the answer to this question is "yes", Facelets in JSF 2.0 is *not* backwards compatibile with Facelets and such an application *must* continue to bundle the Facelets jar file along with the application, continue to set the Facelets configuration parameters, and also set the `javax.faces.DISABLE_FACELET_JSF_VIEWHANDLER` `<context-param>` to `true`. Please see Section 11.1.3 "Application Configuration Parameters" for details on this option. Any code that extends or depends on any class in package `com.sun.facelets` and/or its sub-packages must be modified to depend on the appropriate classes in package `javax.faces.webapp.vdl` and/or its sub-packages.
- If the answer to this question is "no", Facelets in JSF 2.0 *is* backwards compatible with pre-JSF 2.0 Facelets and such an application *must not* continue to bundle the Facelets jar file along with the application, and *must not* continue to set the Facelets configuration parameters.

Thankfully, most applications that use Facelets fall into the latter category, or, if they fall in the former, their dependence will easily be migrated to the new public classes.

Facelets in JSF 2.0 provides tag libraries that are compatible with the following libraries already found in pre JSF 2.0 Facelets.

TABLE 10-2 Taglibs in pre JSF 2.0 Facelets that are available in Facelets in JSF 2.0

Common prefix	Namespace URI
h	http://java.sun.com/jsf/html
f	http://java.sun.com/jsf/core
c	http://java.sun.com/jsp/jstl/core
fn	http://java.sun.com/jsp/jstl/functions
ui	http://java.sun.com/jsf/facelets

Naturally, new features built on Facelets in JSF 2.0 are not available in pre JSF 2.0 Facelets and will only work in JSF 2.0 or later.

10.2 Java Programming Language Specification for Facelets in JSF 2.0

The subsections within this section specify the Java API requirements of a Facelets implementation. Adherence to this section and the next section, which specifies the XHTML specification for Facelets in JSF 2.0, will ensure applications and JSF component libraries that make use of Facelets are portable across different implementations of JavaServer Faces.

The original Facelet project did not separate the API and the implementation into separate jars, as is common practice with JCP specifications. Thus, a significant task for integrating Facelets into JSF 2 was deciding which classes to include in the public Java API, and which to keep as an implementation detail.

There were two guiding principles that influenced the task of integrating Facelets into JSF 2.

- The original decision in JSF 1.0 to allow the ViewHandler to be pluggable enabled the concept of a Page Declaration Language for JSF. The two most popular ones were Facelets and JSFTemplating. The new integration should preserve this pluggability, since it is still valuable to be able to replace the Page Declaration Language.
- After polling users of Facelets, the expert group decided that most of them were only using the markup based API and were not extending from the Java classes provided by the Facelet project. Therefore, we decided to keep the Java API for Facelets in JSF 2 as small as possible, only exposing classes where absolutely necessary.

The application of these principles produced the classes in the package `javax.faces.view.facelets`. Please consult the Javadocs for that package, and the classes within it, for additional normative specification.

10.2.1 Specification of the ViewDeclarationLanguage Implementation for Facelets for JSF 2.0

As normatively specified in the javadocs for `ViewDeclarationLanguageFactory.getViewDeclarationLanguage()`, a JSF implementation must guarantee that a valid and functional `ViewDeclarationLanguage` instance is returned from this method when the argument is a refrence to either a JSP view or a Facelets View. This section describes the specification for the Facelets implementation.

```
public void buildView(FacesContext context,
                              UIViewRoot root)
                      throws IOException
```

The argument `root` will have been created with a call to either `createView()` or `ViewMetadata.createMetadataView()`. If the root already has non-metadata children, this method must return immediately. Otherwise, the implementation must examine the `viewId` of the argument root, which must resolve to an entity written in Facelets for JSF 2 markup language. Because Facelets for JSF 2.0 views are written in XHTML, an XML parser is well suited to the task of processing such an entity. Each element in the XHTML view falls into one of the following categories, each of which corresponds to an instance of a Java object that implements `javax.faces.view.facelets.FaceletHandler`, or a subinterface or subclass thereof, and an instance of `javax.faces.view.facelets.TagConfig`, or a subinterface or subclass thereof, which is passed to the constructor of the object implementing `FaceletHandler`.

The mapping between the categories of elements in the XHTML view and the appropriate subinterface or subclass of `FaceletHandler` is specified below. Each `FaceletHandler` instance must be traversed and its `apply()` method called in the same depth-first order as in the other lifecycle phase methods in jsf. Each `FaceletHandler` instance must use the `getNextHandler()` method of the `TagConfig` instance passed to its constuctor to perform the traversal starting from the root `FaceletHandler`.

- Standard XHTML markup elements

  These are declared in the XHTML namespace `http://www.w3.org/1999/xhtml`. Such elements should be passed through as is to the rendered output.

  These elements correspond to instances of `javax.faces.view.facelets.TextHandler`. See the javadocs for that class for the normative specification.

- Markup elements that represent `UIComponent` instance in the view.

  These elements can come from the Standard HTML Renderkit namespace `http://java.sun.com/jsf/html`, or from the namespace of a custom tag library (including composite components) as described in Section 10.3.2 “Facelet Tag Library mechanism”.

  These elements correspond to instances of `javax.faces.view.facelets.ComponentHandler`. See the javadocs for that class for the normative specification.

- Markup elements that take action on their parent or children markup element(s). Usually these come from the JSF Core namespace `http://java.sun.com/jsf/core`, but they can also be provided by a custom tag library.

  Such elements that represent an attached object must correspond to an appropriate subclass of `javax.faces.view.facelets.FaceletsAttachedObjectHandler`. The supported subclasses are specified in the javadocs.

  Such elements that represent a facet component must correspond to an instance of `javax.faces.component.FacetHandler`.

  Such elements that represent an attribute that must be pushed into the parent `UIComponent` element must correspond to an instance of `javax.facelets.view.facelets.AttributeHandler`.

- Markup Elements that indicate facelet templating, as specified in the VDL Docs for the namespace `http://java.sun.com/jsf/facelets`.

  Such elements correspond to an instance of `javax.faces.view.facelets.TagHandler`.

- Markup elements from the Facelet version of the JSTL namespaces `http://java.sun.com/jsp/jstl/core` or `http://java.sun.com/jsp/jstl/functions`, as specified in the VDL Docs for those namespaces.

  Such elements correspond to an instance of `javax.faces.view.facelets.TagHandler`.

10.3 XHTML Specification for Facelets for JSF 2.0

10.3.1 General Requirements

[P1-start_facelet_xhtml]Facelet pages are authored in XHTML. The runtime must support all XHTML pages that confirm with the XHTML-1.0-Transitional DTD, as described at `http://www.w3.org/TR/xhtml1/#a_dtd_XHTML-1.0-Transitional`.[P1-end_facelet_xhtml]

10.3.2 Facelet Tag Library mechanism

Facelets leverages the XML namespace mechanism to support the concept of a “tag library” analogous to the same concept in JSP. However, in Facelets, the role of the tag handler java class is greatly reduced and in most cases is unnecessary. The tag library mechanism has two purposes.

- Allow page authors to access tags declared in the supplied tag libraries declared in Section 10.4 “Standard Facelet Tag Libraries”, as well as accessing third-party tag libraries developed by the application author, or any other third party

- Define a framework for component authors to group a collection of custom `UIComponents` into a tag library and expose them to page authors for use in their pages.

[P1_start_facelet_taglib_decl]The runtime must support the following syntax for making the tags in a tag library available for use in a Facelet page.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:prefix="namespace_uri">
```

Where *`prefix`* is a page author chosen arbitrary string used in the markup inside the `<html>` tag to refer to the tags declared within the tag library and *`namespace_uri`* is the string declared in the `<namespace>` element of the facelet tag library descriptor. For example, declaring `xmlns:h="http://java.sun.com/jsf/html"` within the `<html>` element in a Facelet XHTML page would cause the runtime to make all tags declared in Section 10.4.2 “Standard HTML RenderKit Tag Library” to be available for use in the page using syntax like: `<h:inputText />`. [P1_end_facelet_taglib_decl]

[P1_start_facelet_taglib_discovery]The run time must support two modes of discovery for Facelet tag library descriptors

- Via declaration in the web.xml, as specified in Section 11.1.3 “Application Configuration Parameters”
- Via auto discovery by placing the tag library discriptor file within a jar on the web application classpath, naming the file so that it ends with “`.taglib.xml`”, without the quotes, and placing the file in the `META-INF` directory in the jar file.

The discovery of tag library files must happen at application startup time and complete before the application is placed in service. Failure to parse, process and otherwise interpret any of the tag library files discovered must cause the application to fail to deploy and must cause an informative error message to be logged.[P1_end_facelet_taglib_discovery]

The specification for how to interpret a facelet tag library descriptor is included in the documentation elements of the schema for such files, see Section 1.1 “XML Schema Definition for Application Configuration Resource file”.

10.3.3 Requirements specific to composite components

The text in this section makes use of the terms defined in Section 3.6.1.6 “Composite Component Terms”. When such a term appears in this section, it will be in *emphasis font face*.

10.3.3.1 Declaring a composite component library for use in a Facelet page

[P1_start_composite_library_decl]The runtime must support the following two ways of declaring a *composite component library*.

- If a facelet taglibrary is declared in an XHTML page with a namespace starting with the string “`http://java.sun.com/jsf/composite/`” (without the quotes), the remainder of the namespace declaration is taken as the name of a resource library as described in Section 2.6.1.4 “Libraries of Localized and Versioned Resources”, as shown in the following example:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ez="http://java.sun.com/jsf/composite/ezcomp">
```

The runtime must look for a resource library named `ezcomp`. If the substring following “`http://java.sun.com/jsf/composite/`” contains a “`/`” character, or any characters not legal for a library name the following action must be taken. If `application.getProjectStage()` is `Development` an informative error message must be placed in the page and also logged. Otherwise the message must be logged only.

- As specified in Section 1.1 “XML Schema Definition for Application Configuration Resource file”, the runtime must also support the `<composite-library-name>` element. The runtime must interpret the contents of this element as the name of a resource library as described in Section 2.6.1.4 “Libraries of Localized and Versioned Resources”. If a facelet tag library descriptor file is encountered that contains this element, the runtime must examine the `<namespace>` element in that same tag library descriptor and make it available for use in an XML namespace declaration in facelet pages.[P1_end_composite_library_decl]

10.3.3.2 Creating an instance of a *top level component*

[P1_start_top_level_component_creation]If, during the process of building the view, the facelet runtime encounters an element in the page using the prefix for the namespace of a composite component library, the runtime must create a `Resource` instance with a library property equal to the library name derived in Section 10.3.3.1 “Declaring a composite component library for use in a Facelet page”and call the variant of `application.createComponent()` that takes a `Resource`.

After causing the *top level component* to be instantiated, the runtime must create a `UIComponent` with component-family of `javax.faces.Panel` and renderer-type `javax.faces.Group` to be installed as a facet of the *top level component* under the facet name `UIComponent.COMPOSITE_FACET_NAME`.[P1_end_top_level_component_creation]

10.3.3.3 Populating a *top level component* instance with children

[P1_start_top_level_component_population]As specified in Section 3.6.1.3 “How does one make a composite component?” the runtime must support the use of `composite:` tag library in the *defining page* pointed to by the `Resource` derived as specified in Section 10.3.3.2 “Creating an instance of a top level component”.
[P1_start_top_level_component_population]The runtime must ensure that all `UIComponent` children in the *composite component definition* within the *defining page* are placed as children of the `UIComponent.COMPOSITE_FACET_NAME` facet of the *top level facet*.[P1_end_top_level_component_population]

Please see the tag library documentation for the `<composite:insertChildren>` and `<composite:insertFacet>` tags for details on these two tags that are relevant to populating a *top level component* instance with children.

Special handling is required for attributes declared on the *composite component tag* instance in the *using page*.
[P1_start_composite_component_tag_attributes]The runtime must ensure that all such attributes are copied to the attributes map of the *top level component* instance in the following manner.

- Obtain a reference to the `ExpressionFactory`, for discussion called *expressionFactory*.
- Let the value of the attribute in the *using page* be *value*.
- If *value* is “id” or “binding” without the quotes, skip to the next attribute.
- If the value of the attribute starts with “#{“ (without the quotes) call `expressionFactory.createValueExpression(elContext, value, Object.class)`
- If the value of the attribute does not start with “`#{`“, call `expressionFactory.createValueExpression(value, Object.class)`
- If there already is a key in the `map` for *value*, inspect the type of the value at that key. If the type is `MethodExpression` take no action. [P1_end_composite_component_tag_attributes]

For code that handles tag attributes on `UIComponent` XHTML elements special action must be taken regarding composite components. [P1_start_composite_component_method_expression]If the type of the attribute is a `MethodExpression`, the code that takes the value of the attribute and creates an actual `MethodExpression` instance around it must take the following special action. Inspect the value of the attribute. If the EL expression string starts with the `compositeComponent` implicit object, is followed by the special string “`attrs`” (without the quotes), as specified in Section 5.6.2.2 “Composite Component Attributes ELResolver”, and is followed by a single remaining expression segment, let the value of that remaining expression segment be *attrName*. In this case, the runtime must guarantee that the actual `MethodExpression` instance that is created for the tag attribute have the following behavior in its `invoke()` method.

- Obtain a reference to the current composite component by calling `UIComponent.getCurrentCompositeComponent()`.
- Look in the attribute of the component for a key under the value *attrName*.
- There must be a value and it must be of type `MethodExpression`. If either of these conditions are `false` allow the ensuing exception to be thrown.
- Call `invoke()` on the discovered `MethodExpression`, passing the arguments passed to our `invoke()` method.[P1_end_composite_component_method_expression]

[P1_start_composite_component_retargeting]Once the composite component has been populated with children, the runtime must ensure that `ViewHandler.retargetAttachedObjects()` and then `ViewHandler.retargetMethodExpressions()` is called, passing the *top level component*.[P1_end_composite_component_retargeting] The actions taken in these methods set the stage for the tag attribute behavior and the special `MethodExpression` handling behavior described previously.

[P1_start_nested_composite_components]The runtime must support the inclusion of composite components within the *composite component definition*. [P1_end_nested_composite_components].

10.4 Standard Facelet Tag Libraries

This section specifies the tag libraries that must be provided by an implementation.

10.4.1 JSF Core Tag Library

This tag library must be equivalent to the one specified in *Section 9.4 "JSF Core Tag Library"*. The following additional tags apply to the Facelet Core Tag Library only.

10.4.1.1 <f:ajax>

This tag serves two roles depending on its placement. If this tag is nested within a single component, it will associate an Ajax action with that component. If this tag is placed around a group of components it will associate an Ajax action with all components that support the "events" attribute.

Syntax

<f:ajax [event="Literal"] [execute="Literal | Value Expression"] [render="Literal | Value Expression"] [onevent="Literal | Value Expression"] [onerror="Literal | Value Expression"] | [listener="Method Expression"] [disabled="Literal|Value Expression"] [immediate="Literal|ValueExpression]/>

Body Content

empty.

Attributes

The following optional attributes are available:

TABLE 10-3

Name	Expr	Type	Description
event	String	String	A String identifying the type of event the Ajax action will apply to. If specified, it must be one of the events supported by the component the Ajax behavior is being applied to. If not specified, the default event is determined for the component.
execute	VE	Collection<String>	If a literal is specified, it must be a space delimited String of component identifiers and/or one of the keywords outlined in Section 14.2.2 “Keywords”. If not specified, then @this is the default. If a ValueExpression is specified, it must refer to a property that returns a Collection of Strings. Each String in the Collection must not contain spaces.
render	VE	Collection<String>	If a literal is specified, it must be a space delimited String of component identifiers and/or one of the keywords outlined in Section 14.2.2 “Keywords”. If not specified, then @none is the default . If a ValueExpression is specified, it must refer to a property that returns a Collection of Strings. Each String in the Collection must not contain spaces.
onevent	VE	String	The name of a JavaScript function that will handle events
onerror	VE	String	The name of a JavaScript function that will handle errors.
disabled	VE	boolean	“false” indicates the Ajax behavior script should be rendered; “true” indicates the Ajax behavior script should not be rendered. “false” is the default.
listener	ME	MethodExpression	The listener method to execute when Ajax requests are processed on he server.
immediate	VE	boolean	If “true” behavior events generated from this behavior are broadcast during Apply Request Values phase. Otherwise, the events will be broadcast during Invoke Aplications phase.

Specifying “execute”/”render” Identifiers

The String value for identifiers specified for `execute` and `render` may be specified as a search expression as outlined in the JavaDocs for UIComponent.findComponent. [P1_start_execrenderIds]The implementation must resolve these identifiers as specified for UIComponent.findComponent.[P1_end]

Constraints

This tag may be nested within any of the standard HTML components. It may also be nested within any custom component that implements the ClientBehaviorHolder interface. Refer to Section 3.7 “Component Behavior Model” for more information about this interface. [P1_start_ajaxtag_events]A TagAttibuteException must be thrown if an “event” attribute value is specified that does not match the events supported by the component type. [P1_end_ajaxtag_events] For example:

```
<h:commandButton ..>
  <f:ajax event=”valueChange”/>
</h:commandButton id=”button1” ...>
```

An attempt is made to apply a “valueChange” Ajax event to an “action” component. This is invalid and the Ajax behavior will not be applied. [P1_start_bevent]The event attribute that is specified, must be one of the events returned from the ClientBehaviorHolder component implementation of ClientBehaviorHolder.getEventNames. If an event is not specified the value returned from the component implementation of ClientBehaviorHolder.getDefaultEventName must be used. If the event is still not determined, a TagAttributeException must be thrown.[P1_end]

This tag may also serve to “ajaxify” regions of a page by nesting a group of components within it:

```
<f:ajax>
  <h:panelGrid>
    <h:inputText id=”text1”/>
    <h:commandButton id=”button1”/>
  </h:panelGrid>
</f:ajax>
```

From this example, “text1” and “button1” will have ajax behavior applied to them. The default events for these components would cause Ajax requests to fire. For “text1” a “valueChange” event would apply and for “button1” an “action” event would apply. <h:panelGrid> has no default event so in this case a behavior would not be applied.

```
<f:ajax event=”click”>
  <h:panelGrid id=”grid1”>
    <h:inputText id=”text1”/>
    <h:commandButton id=”button1”>
      <f:ajax event=”mouseover”/>
    </h:commandButton>
  </h:panelGrid>
</f:ajax>
```

From this example, “grid1” and “text1” would have ajax behavior applied for an “onclick” event. “button1” would have ajax behavior applied for both “mouseover” and “onclick” events.

```
<f:ajax>
    <h:commandButton id=”button1”>
      <f:ajax/>
    </h:commandButton>
</f:ajax>
```

For this example, the inner <f:ajax/> would apply to “button1”. The outer (wrapping) <f:ajax> would not be applied, since it is the same type of submitting behavior (AjaxBehavior) and the same event type (action).

```
<f:ajax event=”action”>
    <h:commandButton id=”button1”>
      <b:greet event=”action”/>
    </h:commandButton>
</f:ajax>
```

Here, there is a custom behavior “greet” attached to “button1”. the outer <f:ajax> Ajax behavior will also get applied to “button1”. But it will be applied *after* the “greet” behavior.

Description

Enable one or more components in the view to perform Ajax operations. This tag handler must create an instance of javax.faces.component.behavior.AjaxBehavior instance using the tag attribute values. If this tag is nested within a single ClientBehaviorHolder component:

- If the event attribute is not specified, determine the event by calling the component’s getDefaultEventName method. If that returns null, throw an exception.
- If the event attribute is specified, ensure that it is a valid event - that is one of the events contained in the Collection returned from getEventNames method. If it does not exist in this Collection, throw an exception.
- Add the AjaxBehavior to the component by calling the addBehavior method, passing the event and AjaxBehavior instance.

If this tag is wrapped around component children add the AjaxBehavior instance to AjaxBehaviors by calling AjaxBehaviors.pushBehavior. As subsequent child components that implement the BehaviorHolder interface are evaluated, this AjaxBehavior instance must be added as a Behavior to the component. Please refer to the Javadocs for the core tag handler AjaxHandler for additional requirements.

Examples

Apply Ajax to “button1” and “text1”:

```
<f:ajax>
  <h:form>
    <h:commandButton id=”button1” ...>
    <h:inputText id=”text1” ..>
  </h:form>
</f:ajax>
```

Apply Ajax to “text1”:

```
<f:ajax event=”valueChange”>
  <h:form>
    <h:commandButton id=”button1” ...>
    <h:inputText id=”text1” ..>
  </h:form>
</f:ajax>
```

Apply Ajax to “button1”:

```
<f:ajax event=”action”>
  <h:form>
    <h:commandButton id=”button1” ...>
    <h:inputText id=”text1” ..>
  </h:form>
</f:ajax>
```

Override default Ajax action. “button1” is associated with the Ajax “execute=’cancel’” action:

```
<f:ajax event=”action” execute=”reset”>
  <h:form>
    <h:commandButton id=”button1” ...>
      <f:ajax execute=”cancel”/>
    </h:commandButton>
    <h:inputText id=”text1” ..>
  </h:form>
</f:ajax>
```

10.4.1.2 <f:event>

Allow JSF page authors to install `ComponentSystemEventListener` instances on a component in a page. Because this tag is closely tied to the event system, please see section Section 3.4.3.4 “Declarative Listener Registration” for the normative specification.

10.4.1.3 <f:metadata>

Register a facet on the parent component, which must be the `UIViewRoot`. This must be a child of the `<f:view>`. This tag must reside within the top level XHTML file for the given viewId, not in a template. The implementation must insure that the direct child of the facet is a `UIPanel`, even if there is only one child of the facet. The implementation must set the id of the `UIPanel` to be the value of the `UIViewRoot.METADATA_FACET_NAME` symbolic constant.

10.4.1.4 <f:validateBean>

Register a BeanValidator instance on the parent EditableValueHolder UIComponent or the EditableValueHolder UIComponent whose client id matches the value of the "for" attribute when used within a composite component. If neither criteria is satisfied, save the validation groups in an attribute on the parent UIComponent to be used as defaults inherited by any BeanValidator in that branch of the component tree. Don't save the validation groups string if it is null or empty string. If the validationGroups attribute is not defined on this tag when used in an EditableValueHolder, or the value of the attribute is empty string, attempt to inherit the validation groups from the nearest parent component on which a set of validation groups is stored. If no validation groups are inherited, assume the Default validation group, javax.validation.groups.Default. If the BeanValidator is one of the default validators, then this tag simply specializes the validator by providing the list of validation groups to be used.

Syntax

<f:validateBean validationGroups="javax.validation.groups.Default,app.validation.groups.Order"/>

Body Content

empty.

Attributes

Name	Exp	Type	Description
binding	VE	ValueExpression	A ValueExpression that evaluates to an object that implements javax.faces.validate.BeanValidator
disabled	VE	Boolean	A flag which indicates whether this validator, or a default validator with the id "javax.faces.Bean", should be permitted to be added to this component
validationGroups	VE	String	A comma-delimited of type-safe validation groups that are passed to the Bean Validation API when validating the value

Constraints

Must be nested in an EditableValueHolder or nested in a composite component and have a for attribute (Facelets only). Otherwise, it simply defines enables or disables the validator as a default for the branch of the component tree under the parent component and/or sets the validation group defaults for the branch. No exception is thrown if one of the first two conditions are not met, unlike other standard validators.

JSR 303 allows the user to validate a graph of objects. This version of the JSF specification does not support graph validation.

Description

- Must extend the ValidateHandler class.
- If not within an EditableValueHolder or composite component, store the validation groups as defaults for the current branch of the component tree, but only if the value is a non-empty string.
- If the disabled attribute is true, the validator should not be added. In addition, the validatorId, if present, should be added to an exclusion list on the parent component to prevent a default validator with the same id from being registered on the component.
- The createValidator() method must:
  - If binding is non-null, create a ValueExpression by invoking Application.createValueExpression() with binding as the expression argument, and Validator.class as the expectedType argument. Use the ValueExpression to obtain a reference to the Validator instance. If there is no exception thrown, and ValueExpression.getValue() returned a non-null object that implements javax.faces.validator.Validator, it must then cast the returned instance to javax.faces.validator.BeanValidator, configure its properties based on the specified attributes, and return the configured instance. If there was an exception thrown, rethrow the exception as a TagException.
  - Use the validatorId if the validator instance could not be created from the binding attribute. Call the createValidator() method of the Application instance for this application, passing validator id "javax.faces.Bean". If the binding attribute was also set, evaluate the expression into a ValueExpression and store the validator instance by calling setValue() on the ValueExpression. It must then cast the returned instance to javax.faces.validator.BeanValidator, configure its properties based on the specified attributes, and return the configured instance. If there was an exception thrown, rethrow the exception as a TagException.

10.4.1.5 <f:validateRequired>

Register a RequiredValidator instance on the parent EditableValueHolder UIComponent or the EditableValueHolder UIComponent whose client id matches the value of the "for" attribute when used within a composite component.

Syntax

<f:validateRequired/>

Body Content

empty

Attributes

Name	Exp	Type	Description
binding	VE	ValueExpression	A ValueExpression that evaluates to an object that implements javax.faces.validate.RequiredValidator
disabled	VE	Boolean	A flag which indicates whether this validator, or a default validator with the id "javax.faces.Required", should be permitted to be added to this component

Constraints

Must be nested in an EditableValueHolder or nested in a composite component and have a for attribute (Facelets only). Otherwise, it simply enables or disables the use of the validator as a default for the branch of the component tree under the parent. No exception is thrown if one of the first two conditions are not met, unlike other standard validators.

Description

- Must use or extend the ValidateHandler class.
- If the disabled attribute is true, the validator should not be added. In addition, the validatorId, if present, should be added to an exclusion list on the parent component to prevent a default validator with the same id from being registered on the component
- The createValidator() method must:
  - If binding is non-null, create a ValueExpression by invoking Application.createValueExpression() with binding as the expression argument, and Validator.class as the expectedType argument. Use the ValueExpression to obtain a reference to the Validator instance. If there is no exception thrown, and ValueExpression.getValue() returned a non-null object that implements javax.faces.validator.Validator, it must then cast the returned instance to javax.faces.validator.RequiredValidator, configure its properties based on the specified attributes, and return the configured instance. If there was an exception thrown, rethrow the exception as a TagException..
  - Use the validatorId if the validator instance could not be created from the binding attribute. Call the createValidator() method of the Application instance for this application, passing validator id "javax.faces.Required". If the binding attribute was also set, evaluate the expression into a ValueExpression and

store the validator instance by calling setValue() on the ValueExpression. It must then cast the returned instance to javax.faces.validator.RequiredValidator, configure its properties based on the specified attributes, and return the configured instance. If there was an exception thrown, rethrow the exception as a TagException.

10.4.2 Standard HTML RenderKit Tag Library

This tag library must be equivalent to the one specified in Section 9.5 “Standard HTML RenderKit Tag Library”.

10.4.3 Facelet Templating Tag Library

This tag library is the specified version of the ui: tag library found in pre JSF 2.0 Facelets. The specification for this library can be found in the VDLDocs for the `ui:` library.

10.4.4 Composite Component Tag Library

This tag library is used to declare composite components. The specification for this tag library can be found in the VDLDocs for the `composite:` library.

10.4.5 JSTL Core and Function Tag Libraries

Facelets exposes a subset of the JSTL Core tag library and the entirety of the JSTL Function tag library. Please see the VDLDocs for the JSTL Core and JSTL Functions tag libraries for the normative specification.

10.5 Assertions relating to the construction of the view hierarchy

[P1-start processListenerForAnnotation] When the VDL calls for the creation of a `UIComponent` instance, after calling `Application.createComponent()` to instantiate the component instance, and after calling `setRendererType()` on the newly instantiated component instance, the following action must be taken.

- Obtain the `Renderer` for this component. If no `Renderer` is present, ignore the following steps.
- Call `getClass()` on the `Renderer` instance and inspect if the `ListenerFor` annotation is present. If so, inspect if the `Renderer` instance implements `ComponentSystemEventListener`. If neither of these conditions are `true`, ignore the following steps.
- Obtain the value of the `systemEventClass()` property of the `ListenerFor` annotation on the `Renderer` instance.
- Call `subscribeToEvent()` on the `UIComponent` instance from which the `Renderer` instance was obtained, using the `systemEventClass` from the annotation as the second argument, and the `Renderer` instance as the third argument.

[P1-end]

11

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.

11.1 Web Application Deployment Descriptor

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

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

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

11.1.1 Servlet Definition

JSF implementations must provide request processing lifecycle services through a standard servlet, defined by this specification. [P1-start-servlet]This servlet must be defined, in the deployment descriptor of an application that wishes to employ this portable mechanism, as follows:

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

The servlet name, denoted as `faces-servlet-name` above, may be any desired value; however, the same value must be used in the servlet mapping (see Section 11.1.2 “Servlet Mapping”).[P1-end]

In addition to `FacesServlet`, JSF implementations may support other ways to invoke the JavaServer Faces request processing lifecycle, but applications that rely on these mechanisms will not be portable.

11.1.2 Servlet Mapping

All requests to a web application are mapped to a particular servlet based on matching a URL pattern (as defined in the *Java Servlet Specification*) against the portion of the request URL after the context path that selected this web application. [P1-start-mapping]JSF implementations must support web application that define a `<servlet-mapping>` that maps any valid `url-pattern` to the `FacesServlet`. [P1-end]Prefix or extension mapping may be used. When using prefix mapping, the following mapping is recommended, but not required:

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

When using extension mapping the following mapping is recommended, but not required:

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

In addition to `FacesServlet`, JSF implementations may support other ways to invoke the JavaServer Faces request processing lifecycle, but applications that rely on these mechanisms will not be portable.

11.1.3 Application Configuration Parameters

Servlet containers support application configuration parameters that may be customized by including `<context-param>` elements in the web application deployment descriptor. [P1-start-configParams]All JSF implementations are required to support the following application configuration parameter names:

- `javax.faces.CONFIG_FILES` -- Comma-delimited list of context-relative resource paths under which the JSF implementation will look for application configuration resources (see Section 11.4.4 “Application Configuration Resource Format”), before loading a configuration resource named “`/WEB-INF/faces-config.xml`” (if such a resource exists). If “`/WEB-INF/faces-config.xml`” is present in the list, it must be ignored.
- `javax.faces.DEFAULT_SUFFIX` -- Allow the web application to define an alternate suffix for JSP pages containing JSF content. See the javadocs for the symbolic constant `ViewHandler.DEFAULT_SUFFIX_PARAM_NAME` for the complete specification.
- `javax.faces.FACELETS_SUFFIX` -- Allow the web application to define an alternate suffix for Facelet based XHTML pages containing JSF content. See the javadocs for the symbolic constant `ViewHandler.FACELETS_SUFFIX_PARAM_NAME` for the complete specification.
- `javax.faces.LIFECYCLE_ID` -- Lifecycle identifier of the `Lifecycle` instance to be used when processing JSF requests for this web application. If not specified, the JSF default instance, identified by `LifecycleFactory.DEFAULT_LIFECYCLE`, must be used.
- `javax.faces.STATE_SAVING_METHOD` -- The location where state information is saved. Valid values are “server” (typically saved in `HttpSession`) and “client (typically saved as a hidden field in the subsequent form submit). If not specified, the default value “server” must be used.
- `javax.faces.PARTIAL_STATE_SAVING` --The ServletContext init parameter consulted by the runtime to determine if the partial state saving mechanism should be used.
  If undefined, the runtime must determine the version level of the application.

- For applications versioned at 1.2 and under, the runtime must not use the partial state saving mechanism.
- For applications versioned at 2.0 and above, the runtime must use the partial state saving mechanism.

If this parameter is defined, and the application is versioned at 1.2 and under, the runtime must not use the partial state saving mechanism. Otherwise, If this param is defined, and calling toLowerCase().equals("true") on a `String` representation of its value returns true, the runtime must use partial state mechanism. Otherwise the partial state saving mechanism must not be used.

- `javax.faces.FULL_STATE_SAVING_VIEW_IDS` -- The runtime must interpret the value of this parameter as a comma separated list of view IDs, each of which must have their state saved using the state saving mechanism specified in JSF 1.2.
- `javax.faces.PROJECT_STAGE` -- A human readable string describing where this particular JSF application is in the software development lifecycle. Valid values are "`Development`", "`UnitTest`", "`SystemTest`", or "`Production`", corresponding to the enum constants of the class `javax.faces.application.ProjectStage`. It is also possible to set this value via JNDI. See the javadocs for `Application.getProjectStage()`.
- `javax.faces.DISABLE_FACELET_JSF_VIEWHANDLER` -- If this param is set, and calling `toLowerCase().equals("true")` on a `String` representation of its value returns `true`, the default `ViewHandler` must behave as specified in the latest 1.2 version of this specification. Any behavior specified in Section 7.5 "ViewHandler" and implemented in the default `ViewHandler` that pertains to handling requests for pages authored in the JavaServer Faces Page Declaration Language must not be executed by the runtime.
- `javax.faces.FACELETS_LIBRARIES` -- If this param is set, the runtime must interpret it as a semicolon (;) separated list of paths, starting with "/" (without the quotes). The runtime must interpret each entry in the list as a path relative to the web application root and interpret the file found at that path as a facelet tag library, conforming to the schema declared in Section 1.1 "XML Schema Definition for Application Configuration Resource file"and expose the tags therein according to Section 10.3.2 "Facelet Tag Library mechanism". The runtime must also consider the `facelets.LIBRARIES` param name as an alias to this param name for backwards compatibility with existing facelets tag libraries.
- `javax.faces.FACELETS_VIEW_MAPPINGS` -- If this param is set, the runtime must interpret it as a semicolon (;) separated list of strings that is used to forcibly declare that certain pages in the application must be interpreted as using Facelets, regardless of their extension. The runtime must also consider the `facelets.VIEW_MAPPINGS` param name as an alias to this param name for backwards compatibility with existing facelets applications. See the javadocs for the symbolic constant `ViewHandler.FACELETS_VIEW_MAPPINGS_PARAM_NAME` for the complete specification.

- `javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL` -- If this param is set, and calling `toLowerCase().equals("true")` on a `String` representation of its value returns `true`, any implementation of `UIInput.validate()` must take the following additional action.

  If the javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL context parameter value is `true` (ignoring case), and UIInput.`getSubmittedValue()` returns a zero-length `String` call `UIInput.setSubmittedValue(null)` and continue processing using null as the current submitted value

- `javax.faces.DATETIMECONVERTER_DEFAULT_TIMEZONE_IS_SYSTEM_TIMEZONE` -- If this param is set, and calling `toLowerCase().equals("true")` on a `String` representation of its value returns `true`, `Application.createConverter()` must guarantee that the default for the timezone of all `javax.faces.convert.DateTimeConverter` instances must be equal to `TimeZone.getDefault()` instead of "GMT".

- `javax.faces.VALIDATE_EMPTY_FIELDS` -- If this param is set, and calling `toLowerCase().equals("true")` on a `String` representation of its value returns `true`, all submitted fields will be validated. This is necessary to allow the model validator to decide whether `null` or empty values are allowable in the current application. If the value is `false`, `null` or empty values will not be passed to the validators. If the value is the string `"auto"`, the runtime must check if JSR-303 Beans Validation is present in the current environment. If so, the runtime must proceed as if the value `"true"` had been specified. If JSR-303 Beans Validation is **not** present in the current environment, the runtime most proceed as if the value `"false"` had been specified. If the param is not set, the system must behave as if the param was set with the value `"auto"`.

- `javax.faces.validator.DISABLE_DEFAULT_BEAN_VALIDATOR` -- If this param is set, and calling `toLowerCase().equals("true")` on a `String` representation of its value returns `true`, the runtime must not automatically add the validator with validator-id equal to the value of the symbolic constant `javax.faces.validator.`VALIDATOR_ID to the list of default validators. Setting this parameter to `true` will have the effect of disabling the automatic installation of Bean Validation to every input component in every view in the application, though manual installation is still possible.

- `[P1-end]`

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.

11.2 Included Classes and Resources

A JSF-based application will rely on a combination of APIs, and corresponding implementation classes and resources, in addition to its own classes and resources. The web application archive structure identifies two standard locations for classes and resources that will be automatically made available when a web application is deployed:

- `/WEB-INF/classes` -- A directory containing unpacked class and resource files.
- `/WEB-INF/lib` -- A directory containing JAR files that themselves contain class files and resources.

In addition, servlet and portlet containers typically provide mechanisms to share classes and resources across one or more web applications, without requiring them to be included inside the web application itself.

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

11.2.1 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.

11.2.2 Servlet and JSP API Classes (javax.servlet.*)

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

11.2.3 JSP Standard Tag Library (JSTL) API Classes (javax.servlet.jsp.jstl.*)

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

11.2.4 JSP Standard Tag Library (JSTL) Implementation Classes

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

11.2.5 JavaServer Faces API Classes (javax.faces.*)

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

11.2.6 JavaServer Faces Implementation Classes

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

11.2.6.1 FactoryFinder

`javax.faces.FactoryFinder` implements the standard discovery algorithm for all factory objects specified in the JavaServer Faces APIs. For a given factory class name, a corresponding implementation class is searched for based on the following algorithm. Items are listed in order of decreasing search precedence:

1. If a default JavaServer Faces configuration file (/WEB-INF/faces-config.xml) is bundled into the `web application, and it` contains a factory entry of the given factory class name, that factory class is used.
2. If the JavaServer Faces configuration resource(s) named by the `javax.faces.CONFIG_FILES ServletContext` init parameter (if any) contain any factory entries of the given factory class name, those factories are used, with the last one taking precedence.
3. If there are any META-INF/faces-config.xml resources bundled any JAR files in the `web ServletContext's resource paths`, the factory entries of the given factory class name in those files are used, with the last one taking precedence.

4. If a `META-INF/services/{factory-class-name}` resource is visible to the web application class loader for the calling application (typically as a result of being present in the manifest of a JAR file), its first line is read and assumed to be the name of the factory implementation class to use.

5. If none of the above steps yield a match, the JavaServer Faces implementation specific class is used.

If any of the factories found on any of the steps above happen to have a one-argument constructor, with argument the type being the abstract factory class, that constructor is invoked, and the previous match is passed to the constructor. For example, say the container vendor provided an implementation of `FacesContextFactory`, and identified it in `META-INF/services/javax.faces.context.FacesContextFactory` in a jar on the webapp ClassLoader. Also say this implementation provided by the container vendor had a one argument constructor that took a `FacesContextFactory` instance. The `FactoryFinder` system would call that one-argument constructor, passing the implementation of `FacesContextFactory` provided by the JavaServer Faces implementation.

If a Factory implementation does not provide a proper one-argument constructor, it must provide a zero-arguments constructor in order to be successfully instantiated.

Once the name of the factory implementation class is located, the web application class loader for the calling application is requested to load this class, and a corresponding instance of the class will be created. A side effect of this rule is that each web application will receive its own instance of each factory class, whether the JavaServer Faces implementation is included within the web application or is made visible through the container's facilities for shared libraries.

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

Create (if necessary) and return a per-web-application instance of the appropriate implementation class for the specified JavaServer Faces factory class, based on the discovery algorithm described above.

JSF implementations must also include implementations of the several factory classes. In order to be dynamically instantiated according to the algorithm defined above, the factory implementation class must include a public, no-arguments constructor. [P1-start-factoryNames]For each of the `public static final String` fields on the class `FactoryFinder` whose field names end with the string "`_FACTORY`" (without the quotes), the implementation must provide an implementation of the corresponding Factory class using the algorithm described earlier in this section.[P1-end]

11.2.6.2 FacesServlet

`FacesServlet` is an implementation of `javax.servlet.Servlet` that accepts incoming requests and passes them to the appropriate `Lifecycle` implementation for processing. This servlet must be declared in the web application deployment descriptor, as described in Section 11.1.1 “Servlet Definition”, and mapped to a standard URL pattern as described in Section 11.1.2 “Servlet Mapping”.

```
public void init(ServletConfig config) throws ServletException;
```

Acquire and store references to the `FacesContextFactory` and `Lifecycle` instances to be used in this web application. For the `LifecycleInstance`, first consult the `init-param` set for this `FacesServlet` instance for a parameter of the name `javax.faces.LIFECYCLE_ID`. If present, use that as the `lifecycleID` attribute to the `getLifecycle()` method of `LifecycleFactory`. If not present, consult the `context-param` set for this web application. If present, use that as the `lifecycleID` attribute to the `getLifecycle()` method of `LifecycleFactory`. If neither param set has a value for `javax.faces.LIFECYCLE_ID`, use the value `DEFAULT`. As an implementation note, please take care to ensure that all `PhaseListener` instances defined for the application are installed on all lifecycles created during this process.

```
public void destroy();
```

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

```
public void service(ServletRequest request, ServletResponse
response) throws IOException, ServletException;
```

For each incoming request, the following processing is performed:

- Using the `FacesContextFactory` instance stored during the `init()` method, call the `getFacesContext()` method to acquire a `FacesContext` instance with which to process the current request.
- Call the `execute()` method of the saved `Lifecycle` instance, passing the `FacesContext` instance for this request as a parameter. If the `execute()` method throws a `FacesException`, re-throw it as a `ServletException` with the `FacesException` as the root cause.
- Call the `render()` method of the saved `Lifecycle` instance, passing the `FacesContext` instance for this request as a parameter. If the `render()` method throws a `FacesException`, re-throw it as a `ServletException` with the `FacesException` as the root cause.
- Call the `release`() method on the `FacesContext` instance, allowing it to be returned to a pool if the JSF implementation uses one.

[P1-start-servletParams]The FacesServlet implementation class must also declare two static public final String constants whose value is a context initialization parameter that affects the behavior of the servlet:

- `CONFIG_FILES_ATTR` -- the context initialization attribute that may optionally contain a comma-delimited list of context relative resources (in addition to `/WEB-INF/faces-config.xml` which is always processed if it is present) to be processed. The value of this constant must be "`javax.faces.CONFIG_FILES`".
- `LIFECYCLE_ID_ATTR` -- the lifecycle identifier of the `Lifecycle` instance to be used for processing requests to this application, if an instance other than the default is required. The value of this constant must be "`javax.faces.LIFECYCLE_ID`".[P1-end]

11.2.6.3 UIComponentELTag

`[P1-start-uicomponenteltag]UIComponentELTag` is an implementation of `javax.servlet.jsp.tagext.BodyTag`, and must be the base class for any JSP custom action that corresponds to a JSF `UIComponent`.[P1-end] See Chapter 9 "Integration with JSP, and the Javadocs for `UIComponentELTag`, for more information about using this class as the base class for your own `UIComponent` custom action classes.

11.2.6.4 FacetTag

JSP custom action that adds a named facet (see Section 3.1.9 "Facet Management") to the UIComponent associated with the closest parent UIComponent custom action. See Section 9.4.6 "<f:facet>".

11.2.6.5 ValidatorTag

JSP custom action (and convenience base class) that creates and registers a `Validator` instance on the `UIComponent` associated with the closest parent `UIComponent` custom action. See Section 9.4.15 "<f:validateDoubleRange>", Section 9.4.16 "<f:validateRegex>", Section 9.4.17 "<f:validateLongRange>", and Section 9.4.18 "<f:validator>".

11.3 Deprecated APIs in the webapp package

Faces depends on version JSP 2.1 or later, and the JSP tags in Faces expose properties that leverage concepts specific to that release of JSP. Importantly, most Faces JSP tag attributes are either of type `javax.el.ValueExpression` or `javax.el.MethodExpression`. For backwards compatability with existing Faces component libraries that expose themselves as JSP tags, the existing classes relating to JSP have been deprecated and new ones introduced that leverage the EL API.

11.3.1 AttributeTag

[P1-start-attributetag]The faces implementation must now provide this class.[P1-end]

11.3.2 ConverterTag

This has been replaced with `ConverterELTag`

11.3.3 UIComponentBodyTag

All component tags now implement `BodyTag` by virtue of the new class `UIComponentClassicTagBase` implementing `BodyTag`. This class has been replaced by `UIComponentELTag`.

11.3.4 UIComponentTag

This component has been replaced by `UIComponentELTag`.

11.3.5 ValidatorTag

This component has been replaced by `ValidatorELTag`.

11.4 Application Configuration Resources

This section describes the JSF support for portable application configuration resources used to configure application components.

11.4.1 Overview

JSF defines a portable configuration resource format (as an XML document) for standard configuration information. One or more such application resources will be loaded automatically, at application startup time, by the JSF implementation. The information parsed from such resources will augment the information provided by the JSF implementation, as described below.

In addition to their use during the execution of a JSF-based web application, configuration resources provide information that is useful to development tools created by Tool Providers. The mechanism by which configuration resources are made available to such tools is outside the scope of this specification.

11.4.2 Application Startup Behavior

Implementations may check for the presence of a `servlet-class` definition of class `javax.faces.webapp.FacesServlet` in the web application deployment descriptor as a means to abort the configuration process and reduce startup time for applications that do not use JavaServer Faces Technology.

At application startup time, before any requests are processed, the [P1-start-startup]JSF implementation must process zero or more application configuration resources, located according as follows

Make a list of all of the application configuration resources found using the following algorithm:

- Search for all resources that match either "`META-INF/faces-config.xml`" or the end with "`.faces-config.xml`" directly in the "`META-INF`" directory. Each resource that matches that expression must be considered an application configuration resource.
- Check for the existence of a context initialization parameter named `javax.faces.CONFIG_FILES`. If it exists, treat it as a comma-delimited list of context relative resource paths (starting with a "/"), and add each of the specfied resources to the list.

Let this list be known as *applicationConfigurationResources* for discussion. Also, check for the existence of a web application configuration resource named "`/WEB-INF/faces-config.xml`", and refer to this as *applicationFacesConfig* for discussion, but do not put it in the list. When parsing the application configuration resources, the implementation must ensure that *applicationConfigurationResources* are parsed before *applicationFacesConfig*.[P1-end]

Please see Section 11.4.7 "Ordering of Artifacts" for details on the ordering in which the decoratable artifacts in the application configuration resources in *applicationConfigurationResources* and *applicationFacesConfig* must be processed.

This algorithm provides considerable flexibility for developers that are assembling the components of a JSF-based web application. For example, an application might include one or more custom `UIComponent` implementations, along with associated `Renderer`s, so it can declare them in an application resource named "`/WEB-INF/faces-config.xml`" with no need to programmatically register them with `Application` instance. In addition, the application might choose to include a component library (packaged as a JAR file) that includes a "`META-INF/faces-config.xml`" resource. The existence of this resource causes components, renderers, and other JSF implementation classes that are stored in this library JAR file to be automatically registered, with no action required by the application.

[P1-start-PostConstructApplicationEvent]The runtime must publish the `javax.faces.event.PostConstructApplicationEvent` immediately after all application configuration resources have been processed.[P1-end]

[P1-start-startupErrors]XML parsing errors detected during the loading of an application resource file are fatal to application startup, and must cause the application to not be made available by the container. JSF implementations that are part of a Java EE technology-compliant implementation are required to validate the application resource file against the XML schema for structural correctness. [P1-end]The validation is recommended, but not required for JSF implementations that are not part of a Java EE technology-compliant implementation.

11.4.3 Application Shutdown Behavior

When the JSF runtime is directed to shutdown by its container, the following actions must be taken. [p1-start-application-shutdown]

1. Ensure that calls to `FacesContext.getCurrentInstance()` that happen during application shutdown return successfully, as specified in the Javadocs for that method.
2. Publish the `javax.faces.event.PreDestroyApplicationEvent`.
3. Call `FactoryFinder.releaseFactories()`.

[p1-end]

11.4.4 Application Configuration Resource Format

[P1-start-schema]Application configuration resources that are written to run on JSF 2.0 must include the following schema declaration and must conform to the schema shown in Chapter A “Appendix A - JSF Metadata:

```
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
    version="2.0">
```

Application configuration resources that are written to run on JSF 1.2 Application configuration resources must include the following schema declaration and must conform to the schema referenced in the schemalocation URI shown below:

```
<faces-config version="1.2" xmlns=
"http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd">
```

Application configuration resources that are written to run on JSF 1.1 implementations must use the DTD declaration and include the following DOCTYPE declaration:

```
<!DOCTYPE faces-config PUBLIC
  “-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN”
  “http://java.sun.com/dtd/web-facesconfig_1_1.dtd”>
```

Application configuration resources that are written to run on JSF 1.0 implementations must use the DTD declaration for the 1.0 DTD contained in the binary download of the JSF reference implementation. They must also use the following DOCTYPE declaration:[P1-end]

```
<!DOCTYPE faces-config PUBLIC
  “-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN”
  “http://java.sun.com/dtd/web-facesconfig_1_0.dtd”>
```

11.4.5 Configuration Impact on JSF Runtime

```
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
```

The following XML elements[1] in application configuration resources cause registration of JSF objects into the corresponding factories or properties. It is an error if the value of any of these elements cannot be correctly parsed, loaded, set, or otherwise used by the implementation.

- **/faces-config/component** -- Create or replace a component type / component class pair with the `Application` instance for this web application.
- **/faces-config/converter** -- Create or replace a converter id / converter class or target class / converter class pair with the `Application` instance for this web application.
- **/faces-config/render-kit** -- Create and register a new `RenderKit` instance with the `RenderKitFactory`, if one does not already exist for the specified `render-kit-id`.
- **/faces-config/render-kit/renderer** -- Create or replace a component family + renderer id / renderer class pair with the `RenderKit` associated with the render-kit element we are nested in.
- **/faces-config/validator** -- Create or replace a validator id / validator class pair with the `Application` instance for this web application.

For components, converters, and validators, it is legal to replace the implementation class that is provided (by the JSF implementation) by default. This is accomplished by specifying the standard value for the `<component-type>`, `<converter-id>`, or `<validator-id>` that you wish to replace, and specifying your implementation class. To avoid class cast exceptions, the replacement implementation class must be a subclass of the standard class being replaced. For example, if you declare a custom `Converter` implementation class for the standard converter identifier `javax.faces.Integer`, then your replacement class must be a subclass of `javax.faces.convert.IntegerConverter`.

For replacement `Renderer`s, your implementation class must extend `javax.faces.render.Renderer`. However, to avoid unexpected behavior, your implementation should recognize all of the render-dependent attributes supported by the Renderer class you are replacing, and provide equivalent decode and encode behavior.

The following XML elements cause the replacement of the default implementation class for the corresponding functionality, provided by the JSF implementation. See Section 11.4.6 “Delegating Implementation Support” for more information about the classes referenced by these elements:

1. Identified by XPath selection expressions.

- **/faces-config/application/action-listener** -- Replace the default `ActionListener` used to process `ActionEvent events with an` instance with the class specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is an `ActionListener`.
- **/faces-config/application/navigation-handler** -- Replace the default `NavigationHandler` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `NavigationHandler`.
- **/faces-config/application/property-resolver** -- Replace the default `PropertyResolver` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `PropertyResolver`.
- **/faces-config/application/state-manager** -- Replace the default `StateManager` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `StateManager`.
- **/faces-config/application/system-event-listener** -- Instantiate a new instance of the class specified as the content within a nested **system-event-listener-class** element, which must implement `SystemEventListener`. This instance is referred to as **systemEventListener** for discussion. If a **system-event-class** is specified as a nested element within **system-event-listener**, it must be a class that extends `SystemEvent` and has a public zero-arguments constructor. The `Class` object for **system-event-class** is obtained and is referred to as **systemEventClass** for discussion. If **system-event-class** is not specified, `SystemEvent.class` must be used as the value of **systemEventClass**. If **source-class** is specified as a nested element within **system-event-listener**, it must be a fully qualified class name. The `Class` object for **source-class** is obtained and is referred to as **sourceClass** for discussion. If **source-class** is not specified, let **sourceClass** be `null`. Obtain a reference to the `Application` instance and call `subscribeForEvent(`**`facesEventClass, sourceClass, systemEventListener`**`)`, passing the arguments as assigned in the discussion.
- **/faces-config/application/variable-resolver** -- Replace the default `VariableResolver` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `VariableResolver`.
- **/faces-config/application/view-handler** -- Replace the default `ViewHandler` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `ViewHandler`.
- **/faces-config/application/resource-handler** -- Replace the default `ResourceHandler` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `ResourceHandler`.

The following XML elements cause the replacement of the default implementation class for the corresponding functionality, provided by the JSF implementation. Each of the referenced classes must have a public zero-arguments constructor:

- **/faces-config/factory/application-factory** -- Replace the default `ApplicationFactory` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is an `ApplicationFactory`.
- **/faces-config/factory/exception-handler-factory** -- Replace the default `ExceptionHandlerFactory` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `ExceptionHandlerFactory`.
- **/faces-config/factory/faces-context-factory** -- Replace the default `FacesContextFactory` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `FacesContextFactory`.
- **/faces-config/factory/lifecycle-factory** -- Replace the default `LifecycleFactory` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `LifecycleFactory`.
- **/faces-config/factory/page-declaration-language-factory** -- Replace the default `ViewDeclarationLanguageFactory` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `ViewDeclarationLanguageFactory`.
- **/faces-config/factory/render-kit-factory** -- Replace the default `RenderKitFactory` instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a `RenderKitFactory`.

The following XML elements cause the addition of event listeners to standard JSF implementation objects, as follows. Each of the referenced classes must have a public zero-arguments constructor.

- **/faces-config/lifecycle/phase-listener** -- Instantiate a new instance of the specified class, which must implement `PhaseListener`, and register it with the `Lifecycle` instance for the current web application.

In addition, the following XML elements influence the runtime behavior of the JSF implementation, even though they do not cause registration of objects that are visible to a JSF-based application.

- **/faces-config/managed-bean** -- Make the characteristics of a managed bean with the specified `managed-bean-name` available to the default `VariableResolver` implementation.
- **/faces-config/navigation-rule** -- Make the characteristics of a navigation rule available to the default `NavigationHandler` implementation.

11.4.6 Delegating Implementation Support

[P1-decoratable_artifacts]The runtime must support the decorator design pattern as specified below for the following artifacts.

- `ActionListener`
- `ApplicationFactory`
- `FacesContextFactory`
- `LifecycleFactory`
- `NavigationHandler`
- `PropertyResolver`
- `RenderKit`
- `RenderKitFactory`
- `ResourceHandler`
- `StateManager`
- `VariableResolver`
- `ViewHandler`

[P1_end_decoratable_artifacts]For all of these artifacts, the decorator design pattern is leveraged, so that if one provides a constructor that takes a single argument of the appropriate type, the custom implementation receives a reference to the implementation that was previously fulfilling the role. In this way, the custom implementation is able to override just a subset of the functionality (or provide only some additional functionality) and delegate the rest to the existing implementation.

The implementation must also support decoration of a `RenderKit` instance. At the point in time of when the `<render-kit>` element is processed in an application configuration resources, if the current `RenderKitFactory` already has a `RenderKit` instance for the `<render-kit-id>` within the `<render-kit>` element, and the Class whose fully qualified java class name is given as the value of the `<render-kit-class>` element within the `<render-kit>` element has a constructor that takes an `RenderKit` instance, the existing `RenderKit` for that `<render-kit-id>` must be passed to that constructor, and the RenderKit resulting from the executing of that constructor must be passed to `RenderKitFactory.addRenderKit()`.

For example, say you wanted to provide a custom `ViewHandler` that was the same as the default one, but provided a different implementation of the `calculateLocale()` method. Consider this code excerpt from a custom `ViewHandler`:

```
public class MyViewHandler extends ViewHandler {

  public MyViewHandler() { }

  public MyViewHandler(ViewHandler handler) {
    super();
    oldViewHandler = handler;
  }

  private ViewHandler oldViewHandler = null;

  // Delegate the renderView() method to the old handler
  public void renderView(FacesContext context, UIViewRoot view)
      throws IOException, FacesException {
    oldViewHandler.renderView(context, view);
  }

  // Delegate other methods in the same manner

  // Overridden version of calculateLocale()
  public Locale calculateLocale(FacesContext context) {
    Locale locale = ... // Custom calculation
    return locale;
  }

}
```

The second constructor will get called as the application is initially configured by the JSF implementation, and the previously registered `ViewHandler` will get passed to it.

In version 1.2, we added new wrapper classes to make it easier to override a subset of the total methods of the class and delegate the rest to the previous instance. We provide wrappers for `javax.faces.application.ViewHandler`, `javax.faces.application.StateManager`, and `javax.faces.context.ResponseWriter`. For example, you could have a `faces-config.xml` file that contains the following:

```
<application>
   <view-handler>com.foo.NewViewHandler</view-handler>
   <state-manager>com.foo.NewStateManager</state-manager>
 </application>
```

Where your implementations for these classes are simply:

```
package com.foo;

import javax.faces.application.ViewHandler;
import javax.faces.application.ViewHandlerWrapper;

public class NewViewHandler extends ViewHandlerWrapper {

    private ViewHandler oldViewHandler = null;

    public NewViewHandler(ViewHandler oldViewHandler) {
       this.oldViewHandler = oldViewHandler;
    }

    public ViewHandler getWrapped() {
       return oldViewHandler;
    }
}

package com.foo;

import javax.faces.application.StateManager;
import javax.faces.application.StateManagerWrapper;

public class NewStateManager extends StateManagerWrapper {

    private StateManager oldStateManager = null;

    public NewStateManager(StateManager oldStateManager) {
       this.oldStateManager = oldStateManager;
    }

    public StateManager getWrapped() {
       return oldStateManager;
    }
}
```

This allows you to override as many or as few methods as you'd like.

11.4.7 Ordering of Artifacts

Because the specification allows the application configuration resources to be composed of multiple files, discovered and loaded from several different places in the application, the question of ordering must be addressed. This section specifies how application configuration resource authors may declare the ordering requirements of their artifacts.

Section 11.4.2 "Application Startup Behavior" defines two concepts: *applicationConfigurationResources* and *applicationFacesConfig*. The former is an ordered list of all the application configuration resources **except** the one at "`WEB-INF/faces-config.xml`", and the latter is a list containing **only** the one at "`WEB-INF/faces-config.xml`".

An application configuration resource may have a top level `<name>` element of type `javaee:java-identifierType`. [P1-facesConfigIdStart]If an `<name>` element is present, it must be considered for the ordering of decoratable artifacts (unless the *duplicate name exception* applies, as described below).

Two cases must be considered to allow application configuration resources to express their ordering preferences.

1. Absolute ordering: an `<absolute-ordering>` element in the *applicationFacesConfig*

   In this case, ordering preferences that would have been handled by case 2 below must be ignored.

   Any `<name>` element direct children of the `<absolute-ordering>` must be interpreted as indicating the absolute ordering in which those named application configuration resources, which may or may not be present in *applicationConfigurationResources,* must be processed.

   The `<absolute-ordering>` element may contain zero or one `<others />` element. The required action for this element is described below. If the `<absolute-ordering>` element does not contain an `<others />` element, any application configuration resources not specifically mentioned within `<name />` elements must be ignored.

   Duplicate name exception: if, when traversing the children of `<absolute-ordering>`, multiple children with the same `<name>` element are encountered, only the first such occurrence must be considered.

   If an `<ordering>` element appears in the *applicationFacesConfig*, an informative message must be logged and the element must be ignored.

2. Relative ordering: an `<ordering>` element within a file in the *applicationConfigurationResources*

   An entry in *applicationConfigurationResources* may have an `<ordering>` element. If so, this element must contain zero or one `<before>` element and zero or one `<after>` element. The meaning of these elements is explained below.

Duplicate name exception: if, when traversing the constituent members of *applicationConfigurationResources,* multiple members with the same `<name>` element are encountered, the application must log an informative error message including information to help fix the problem, and must fail to deploy. For example, one way to fix this problem is for the user to use absolute ordering, in which case relative ordering is ignored.

If an `<absolute-ordering>` element appears in an entry in *applicationConfigurationResources*, an informative message must be logged and the element must be ignored.

Consider this abbreviated but illustrative example. faces-configA, faces-configB and faces-configC are found in *applicationConfigurationResources*, while my-faces-config is the *applicationFacesConfig*. The principles that explain the ordering result follow the example code.

faces-configA:.

```
<faces-config>
  <name>A</name>
  <ordering><after><name>B</name></after></ordering>
  <application>
    <view-handler>com.a.ViewHandlerImpl</view-handler>
  </application>
  <lifecycle>
    <phase-listener>com.a.PhaseListenerImpl</phase-listener>
  </lifecycle>
</faces-config>
```

faces-configB:.

```
<faces-config>
  <name>B</name>
  <application>
    <view-handler>com.b.ViewHandlerImpl</view-handler>
  </application>
  <lifecycle>
    <phase-listener>com.b.PhaseListenerImpl</phase-listener>
  </lifecycle>
</faces-config>
```

faces-configC:.

```
<faces-config>
  <name>C</name>
  <ordering><before><others /></before></ordering>
  <application>
    <view-handler>com.c.ViewHandlerImpl</view-handler>
  </application>
  <lifecycle>
    <phase-listener>com.c.PhaseListenerImpl</phase-listener>
  </lifecycle>
</faces-config>
```

my-faces-config:.

```
<faces-config>
  <name>my</name>
  <application>
    <view-handler>com.my.ViewHandlerImpl</view-handler>
  </application>
  <lifecycle>
    <phase-listener>com.my.PhaseListenerImpl</phase-listener>
  </lifecycle>
</faces-config>
```

In this example, the processing order for the *applicationConfigurationResources* and *applicationFacesConfig* will be.

```
Implementation Specific Config
C
B
A
my
```

The preceding example illustrates some, but not all, of the following principles.[P1-start-decoratableOrdering]

- `<before>` means the document must be ordered **before** the document with the **name** matching the **name** specified within the nested `<name>` element.
- `<after>` means the document must be ordered **after** the document with the **name** matching the **name** specified within the nested `<name>` element.

- There is a special element <others /> which may be included zero or one time within the <before> or <after> element, or zero or one time directly within the <absolute-ordering> element. The <others /> element must be handled as follows.
  - If the <before> element contains a nested <others />, the document will be moved to the beginning of the list of sorted documents. If there are multiple documents stating <before><others />, they will all be at the beginning of the list of sorted documents, but the ordering within the group of such documents is unspecified.
  - If the <after> element contains a nested <others />, the document will be moved to the end of the list of sorted documents. If there are multiple documents requiring <after><others />, they will all be at the end of the list of sorted documents, but the ordering within the group of such documents is unspecified.
  - Within a <before> or <after> element, if an <others /> element is present, but is not the only <name> element within its parent element, the other elements within that parent must be considered in the ordering process.
  - If the <others /> element appears directly within the <absolute-ordering> element, the runtime must ensure that any application configuration resources in *applicationConfigurationResources* not explicitly named in the <absolute-ordering> section are included at that point in the processing order.
- If a faces-config file does not have an <ordering> or <absolute-ordering> element the artifacts are assumed to not have any ordering dependency.
- If the runtime discovers circular references, an informative message must be logged, and the application must fail to deploy. Again, one course of action the user may take is to use absolute ordering in the *applicationFacesConfig*.

The previous example can be extended to illustrate the case when *applicationFacesConfig* contains an ordering section.

my-faces-config:.

```
<faces-config>
  <name>my</name>
  <absolute-ordering>
    <name>C</name>
    <name>A</name>
  </absolute-ordering>
  <application>
    <view-handler>com.my.ViewHandlerImpl</view-handler>
  </application>
  <lifecycle>
    <phase-listener>com.my.PhaseListenerImpl</phase-listener>
  </lifecycle>
</faces-config>
```

In this example, the constructor decorator ordering for `ViewHandler` would be C, A, my.

Some additional example scenarios are included below. All of these apply to the *applicationConfigurationResources* relative ordering case, not to the *applicationFacesConfig* absolute ordering case.

```
Document A - <after><others/><name>C</name></after>
Document B - <before><others/></before>
Document C - <after><others/></after>
Document D - no ordering
Document E - no ordering
Document F - <before><others/><name>B</name></before>
```

The valid parse order is F, B, D/E, C, A, where D/E may appear as D, E or E, D

```
Document <no id> - <after><others/></after>
                   <before><name>C</name></before>
Document B - <before><others/></before>
Document C - no ordering
Document D - <after><others/></after>
Document E - <before><others/></before>
Document F - no ordering
```

The complete list of parse order solutions for the above example is

B,E,F,<no id>,C,D

B,E,F,<no_id>,D,C

E,B,F,<no id>,C,D

E,B,F,<no_id>,D,C

B,E,F,D,<no id>,C

E,B,F,D,<no id>,C

```
Document A - <after><name>B</name></after>
Document B - no ordering
Document C - <before><others/></before>
Doucment D - no ordering
```

Resulting parse order: C, B, D, A. The parse order could also be: C, D, B, A.

[P1-endDecoratableOrdering]

11.4.8 Example Application Configuration Resource

The following example application resource file defines a custom `UIComponent` of type `Date`, plus a number of `Renderer`s that know how to decode and encode such a component:

```
<?xml version=”1.0”?>
<faces-config version="1.2" xmlns=
"http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd">
<!-- Define our custom component -->
<component>
  <description>
    A custom component for rendering user-selectable dates in
    various formats.
  </description>
  <display-name>My Custom Date</display-name>
  <component-type>Date</component-type>
  <component-class>
    com.example.components.DateComponent
  </component-class>
</component>

<!-- Define two renderers that know how to deal with dates -->
<render-kit>
  <!-- No render-kit-id, so add them to default RenderKit -->
  <renderer>
    <display-name>Calendar Widget</display-name>
    <component-family>MyComponent</component-family>
    <renderer-type>MyCalendar</renderer-type>
    <renderer-class>
      com.example.renderers.MyCalendarRenderer
    </renderer-class>
  </renderer>
  <renderer>
    <display-name>Month/Day/Year</display-name>
    <renderer-type>MonthDayYear</renderer-type>
    <renderer-class>
      com.example.renderers.MonthDayYearRenderer
    </renderer-class>
  </renderer>
</render-kit>

</faces-config>
```

Additional examples of configuration elements that might be found in application configuration resources are in *Section 5.3.1 “Managed Bean Configuration Example”* and Section 7.4.3 “Example NavigationHandler Configuration”.

11.5 Annotations that correspond to and may take the place of entries in the Application Configuration Resources

An implementation must support several annotation types that take may take the place of entries in the Application Configuration Resources. The implementation requirements are specified in this section.

11.5.1 Requirements for scanning of classes for annotations

- [P1_start-annotation-discovery]If the `<faces-config>` element in the `WEB-INF/faces-config.xml` file contains `metadata-complete` attribute whose value is "`true`", the implementation must not perform annotation scanning on any classes except for those classes provided by the implementation itself. Otherwise, continue as follows.
- If the runtime discovers a conflict between an entry in the Application Configuration Resources and an annotation, the entry in the Application Configuration Resources takes precedence.
- All classes in `WEB-INF/classes` must be scanned.
- For every jar in the application's `WEB-INF/lib` directory, if the jar contains a "`META-INF/faces-config.xml`" file or a file that matches the regular expression "`META-INF/.*faces-config.xml`" (even an empty one), all classes in that jar must be scanned.[P1_end-annotation-discovery]

The following table lists the annations that the implementation must support. The normative specification for each annotation is found in its corresponding javadoc.

TABLE 11-1 Annotations that relate to artifacts that reside in a page

Fully Qualifiied Class Name	Description
`javax.faces.component.FacesComponent`	Class level annotation on a class that has `UIComponent` in its inheritance hierarchy.
`javax.faces.convert.FacesConverter`	Class level annotation on a class that has `Converter` in its inheritance hierarchy.
`javax.faces.render.FacesRenderer`	Class level annotation on a class that has `Renderer` in its inheritance hierarchy.
`javax.faces.render.FacesRenderKit`	Class level annotation on a class that has `RenderKit` in its inheritance hierarchy.
`javax.faces.validator.FacesValidator`	Class level annotation on a class that has `Validator` in its inheritance hierarchy.

12

Lifecycle Management

In Chapter 2 “Request Processing Lifecycle,” the required functionality of each phase of the request processing lifecycle was described. This chapter describes the standard APIs used by JSF implementations to manage and execute the lifecycle. Each of these classes and interfaces is part of the `javax.faces.lifecycle` package.

Page authors, component writers, and application developers, in general, will not need to be aware of the lifecycle management APIs—they are primarily of interest to tool providers and JSF implementors.

12.1 Lifecycle

Upon receipt of each JSF-destined request to this web application, the JSF implementation must acquire a reference to the `Lifecycle` instance for this web application, and call its `execute()` and `render()` methods to perform the request processing lifecycle. The `Lifecycle` instance invokes appropriate processing logic to implement the required functionality for each phase of the request processing lifecycle, as described in Section 2.2 “Standard Request Processing Lifecycle Phases”.

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

public void render(FacesContext context) throws FacesException;
```

The `execute()` method performs phases up to, but not including, the *Render Response* phase. The `render()` method performs the Render Response phase. This division of responsibility makes it easy to support JavaServer Faces processing in a portlet-based environment.

As each phase is processed, registered `PhaseListener` instances are also notified. The general processing for each phase is as follows:

- From the set of registered `PhaseListener` instances, select the relevant ones for the current phase, where "relevant" means that calling `getPhaseId()` on the `PhaseListener` instance returns the phase identifier of the current phase, or the special value `PhaseId.ANY_PHASE`.
- Call the `beforePhase()` method of each relevant listener, in the order that the listeners were registered.
- If no called listener called the `FacesContext.renderResponse()` or `FacesContext.responseComplete()` method, execute the functionality required for the current phase.
- Call the `afterPhase()` method of each relevant listener, in the reverse of the order that the listeners were registered.
- If the `FacesContext.responseComplete()` method has been called during the processing of the current request, or we have just completed the *Render Response* phase, perform no further phases of the request processing lifecycle.
- If the `FacesContext.renderResponse()` method has been called during the processing of the current request, and we have not yet executed the *Render Response* phase of the request processing lifecycle, ensure that the next executed phase will be *Render Response*

```
public void addPhaseListener(PhaseListener listener);

public void removePhaseListener(PhaseListener listener);
```

These methods register or deregister a `PhaseListener` that wishes to be notified before and after the processing of each standard phase of the request processing lifecycle. Implementations should prevent duplicate `PhaseListener` registrations and log an exception if an attempt is made. The webapp author can declare a `PhaseListener` to be added using the `phase-listener` element of the application configuration resources file. Please see *Section 12.3 "PhaseListener"*.

12.2 PhaseEvent

This class represents the beginning or ending of processing for a particular phase of the request processing lifecycle, for the request encapsulated by the `FacesContext` instance passed to our constructor.

```
public PhaseEvent(FacesContext context, PhaseId phaseId, Lifecycle
lifecycle);
```

Construct a new `PhaseEvent` representing the execution of the specified phase of the request processing lifecycle, on the request encapsulated by the specified `FacesContext` instance. The `Lifecycle` instance must be the lifecycle used by the current `FacesServlet` that is processing the request. It will serve as the `source` of the `java.util.EventObject` from which `PhaseEvent` inherits.

```
public FacesContext getFacesContext();

public PhaseId getPhaseId();
```

Return the properties of this event instance. The specified `FacesContext` instance will also be returned if `getSource()` (inherited from the base `EventObject` class) is called.

12.3 PhaseListener

This interface must be implemented by objects that wish to be notified before and after the processing for a particular phase of the request processing lifecycle, on a particular request. Implementations of `PhaseListener` must be programmed in a thread-safe manner.

```
public PhaseId getPhaseId();
```

The `PhaseListener` instance indicates for which phase of the request processing lifecycle this listener wishes to be notified. If `PhaseId.ANY_PHASE` is returned, this listener will be notified for all standard phases of the request processing lifecycle.

```
public void beforePhase(PhaseEvent event);

public void afterPhase(PhaseEvent event);
```

The beforePhase() method is called before the standard processing for a particular phase is performed, while the afterPhase() method is called after the standard processing has been completed. The JSF implementation must guarantee that, if `beforePhase()` has been called on a particular instance, then `afterPhase()` will also be called, regardless of any Exceptions that may have been thrown during the actual execution of the lifecycle phase. For example, let's say there are three PhaseListeners attached to the lifecycle: `A`, `B`, and `C`, in that order. `A.beforePhase()` is called, and executes successfully. `B.beforePhase()` is called and throws an exception. [P1-start_publishExceptionBefore] Any exceptions thrown during the `beforePhase()` listeners must be caught and published to the `ExceptionHandler`, as described below.[P1-end_publishExceptionBefore] In this

example, `C.beforePhase()` must not be called. Then the actual lifecycle phase executes. Any exceptions thrown during the execution of the actual phase, that reach the runtime code that implements the JSF lifecycle phase, [P1-start_publishExceptionDuring] must be caught and and published to the `ExceptionHandler`, as described below[P1-end_publishExceptionDuring]. When the lifecycle phase exits, due to an exeception or normal termination, the `afterPhase()` listeners must be called in reverse order from the `beforePhase()` listeners in the following manner. `C.afterPhase()` must not be called, since `C.beforePhase()` was not called. `B.afterPhase()` must not be called, since `B.beforePhase()`did not execute successfully. `A.afterPhase()` must be called. [P1-start_publishExceptionAfter]Any exceptions thrown during the `afterPhase()` liseteners must be caught and published to the `ExceptionHandler`, as described below.[P1-start_publishExceptionAfter]

The previous paragraph detailed several cases where exceptions should be published to the `Exception` handler. [P1-start_publishExceptionSpec] The following action must be taken by the runtime to implement this requirement as well as an additional requirent to cause the `ExceptionHandler` to take action on the published `Exception`(s). The specification is shown in pseudocode. This code does not implement the before/after matching guarantees specified above and is only intended to describe the specification for publishing and handling

`ExceptionEvent` instances that arise from exceptions being thrown during the execution of a lifecycle phase. Methods shown in *`thisTypeface()`* are not a part of the API and are just included for discussion.

```
FacesContext facesContext = FacesContext.getCurrentInstance();
Application app = facesContext.getApplication();
ExceptionHandler handler = facesContext.getExceptionHandler();

try {
  callBeforePhaseListeners();
} catch (Throwable thrownException) {
  javax.faces.event.ExceptionEventContext eventContext =
    new ExceptionEventContext(thrownException, null,
                              facesContext.getPhaseId());
  eventContext.getAttributes().put(EventContext.IN_BEFORE_PHASE,
                                   Boolean.TRUE);
  app.publishEvent(ExceptionEvent.class, eventContext);
}

try {
  doCurrentPhase();
} catch (Throwable thrownException) {
  javax.faces.event.ExceptionEventContext eventContext =
    new ExceptionEventContext(thrownException, null,
                              facesContext.getPhaseId());
  app.publishEvent(ExceptionEvent.class, eventContext);
} finally {
  try {
    callAfterPhaseListeners();
  } catch (Throwable thrownException) {
    javax.faces.event.ExceptionEventContext eventContext =
      new ExceptionEventContext(thrownException, null,
                              facesContext.getPhaseId());
  eventContext.getAttributes().put(EventContext.IN_AFTER_PHASE,
                                   Boolean.TRUE);
    app.publishEvent(ExceptionEvent.class, eventContext);
  }
  handler.handle();
}
```

body text.

[P1-end_publishExceptionSpec]

`PhaseListener` implementations may affect the remainder of the request processing lifecycle in several ways, including:

- Calling `renderResponse()` on the `FacesContext` instance for the current request, which will cause control to transfer to the *Render Response* phase of the request processing lifecycle, once processing of the current phase is complete.
- Calling responseComplete() on the FacesContext instance for the current request, which causes processing of the request processing lifecycle to terminate once the current phase is complete.

12.4 LifecycleFactory

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

```
LifecycleFactory factory = (LifecycleFactory)
   FactoryFinder.getFactory(FactoryFinder.LIFECYCLE_FACTORY);
```

The `LifecycleFactory` implementation class supports the following methods:

```
public void addLifecycle(String lifecycleId, Lifecycle lifecycle);
```

Register a new `Lifecycle` instance under the specified lifecycle identifier, and make it available via calls to the `getLifecycle` method for the remainder of the current web application's lifetime.

```
public Lifecycle getLifecycle(String lifecycleId);
```

The `LifecycleFactory` implementation class provides this method to create (if necessary) and return a `Lifecycle` instance. All requests for the same lifecycle identifier from within the same web application will return the same `Lifecycle` instance, which must be programmed in a thread-safe manner.

Every JSF implementation must provide a `Lifecycle` instance for a default lifecycle identifier that is designated by the `String` constant `LifecycleFactory.DEFAULT_LIFECYCLE`. For advanced uses, a JSF implementation may support additional lifecycle instances, named with unique lifecycle identifiers.

```
public Iterator<String> 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`.

13

Ajax Integration

This chapter of the specification describes how Ajax integrates with the JavaServer Faces framework to create dynamic web applications. JavaServer Faces 1.2 standardized portions of the architecture to facilitate building Web 2.0 applications with Ajax. This chapter describes the resources and JavaScript APIs that are used to expose the Ajax capabilities of JavaServer Faces to page authors and component authors. It also describes the necessary ingredients of a JavaServer Faces Ajax framework, namely, a resource delivery mechanism, partial tree traversal, partial page update.

13.1 JavaScript Resource

There must be a single JavaScript resource that exists with the resource identifier `jsf.js` and it must exist under the resource library `javax.faces,` following the conventions in `Section 2.6 “Resource Handling”.` This resource contains the JavaScript APIs that facilitate Ajax interaction with JavaServer Faces.

13.1.1 JavaScript Resource Loading

The JavaScript resource can become available to a JavaServer Faces application using a number of different approaches.

13.1.1.1 The Annotation Approach

Component authors can specify that a custom component or renderer requires the Ajax resource with the use of the `ResourceDependency` annotation.

```
@ResourceDependency (name="jsf.js", library="javax.faces",
target="head")
public class MyComponent extends UIOutput...
```

For more information on this approach refer to Section 2.6.2.1 “Relocatable Resources” and Section 2.6.2.2 “Resource Rendering Using Annotations”.

13.1.1.2 The Resource API Approach

Component authors can also specify that a custom component or renderer requires the JavaScript resource by using the resource APIs. For example, a component or renderer's encode method may contain:

```
Resource resource = context.getApplication().getResourceHandler()
             .createResource(“jsf.js”, “javax.faces”);
...
writer.startElement("script", component);
writer.writeAttribute("type", "text/javascript", "type");
writer.writeAttribute("src",((resource != null)?
resource.getRequestPath(): "RES_NOT_FOUND"),”src”);
writer.endElement(“script”);
```

Script resources are relocatable resources (see Section 2.6.2.1 “Relocatable Resources”) which means you can control the rendering location for these resources by setting the “target” attribute on the resource component:

```
public class MyComponent extends UIOutput {
...
   getAttributes().put(“target”, “head”);
...
}
```

This attribute must be set before the component is added to the view. The component or renderer must also implement the event processing method:

```
public void processEvent(SystemEvent event) throws
AbortProcessingException {
   UIComponent component = (UIComponent) event.getSource();
   FacesContext context = FacesContext.getCurrentInstance();
   if (component.getAttributes().get("target") != null) {
       context.getViewRoot().addComponentResource(context,
       component);
   }
}
```

When the component is added to the view, an event will be published. This event handling method will add the component resource to one of the resource location facets under the view root so it will be in place before rendering.

13.1.1.3 The Page Declaration Language Approach

Page authors can make the Ajax resource available to the current view using the `outputScript` tag. For example:

```
<f:view contentType="text/html"/>
 <h:head>
   <meta...
   <title...
 </h:head>
 <h:body>
...
 <h:outputScript name="jsf.js" library="javax.faces"
    target="body"/>
...
 </h:body>
...
```

13.2 JavaScript Namespacing

JavaScript objects that are not enclosed within a namespace are global, which means they run the risk of interfering, overriding and/or clobbering previously defined JavaScript objects. This section defines the requirements for implementations intending to use the JavaServer Faces 2.0 JavaScript API.

The Open Ajax Alliance is an organization of leading vendors, open source projects, and companies using Ajax. Their prime objective is to accelerate customer success with Ajax, through the use of open standards. The Open Ajax Registry is an industry-wide Ajax registration authority managed by the OpenAjax Alliance. The Registry maintains industry-wide lists of Ajax runtime libraries to help prevent object collisions.

There is a top level namespace `jsf` that is registered with the Open Ajax Alliance:

```
Java Ajax: {
 namespaceURI:"http://www.sun.com",
version:"1.0",
globals_to_approve:["jsf"],
comments: "Used in the JSF 2.0 specification.",
specificationURI:"http://www.jcp.org/en/jsr/detail?id=316",
email: "jsfaces@sun.com"
}
```

[P1-start openajax registration]If the OpenAjax library is available, libraries must register themselves using `OpenAjax.registerLibrary()` at the time when the JavaScript files are fetched and parsed by the browser's JavaScript engine.

```
if (typeof OpenAjax != "undefined" &&
    typeof OpenAjax.hub.registerLibrary != "undefined") {
    OpenAjax.hub.registerLibrary("jsf", "www.sun.com", "1.0",
    null);
}
```

[P1-end]

[P1-start javascript namespace]Any implementation that intends to use the JavaServer Faces 2.0 JavaScript API must define a top level JavaScript object name `jsf,` whose type is a JavaScript associative array. Within that top level JavaScript object, found in the OpenAjax Hub, there must be a property named `ajax.`.

```
if (jsf == null || typeof jsf == "undefined") {
    var jsf = new Object();
}
if (jsf.ajax == null || typeof jsf.ajax == "undefined") {
    jsf["ajax"] = new Object();
}
```

[P1-end]

13.3 Ajax Interaction

This section of the specification outlines the Ajax JavaScript APIs that are used to initiate client side interactions with the JavaServer Faces framework including partial tree traversal and partial page update. All of the functions in this JavaScript API will be exposed on a page scoped JavaScript object. Refer to Chapter 14 “JavaScript APIfor details about the individual API functions.

13.3.1 Sending an Ajax Request

The JavaScript function `jsf.ajax.request` is used to send information to the server to control partial view processing (Section 13.4.2 “Partial View Processing”) and partial view rendering (Section 13.4.3 “Partial View Rendering”). All requests using the `jsf.ajax.request` function will be made asynchronously to the server. Refer to Section 14.2 “Initiating an Ajax Request”.

13.3.2 Ajax Request Queueing

[P1-start-ajaxrequest-queue] All Ajax requests must be put into a client side request queue before they are sent to the server to ensure Ajax requests are processed in the order they are sent. The request that has been waiting in the queue the longest is the next request to be sent. After a request is sent, the Ajax request callback function must remove the request from the queue (also known as dequeuing). If the request completed successfully, it must be removed from the queue. If there was an error, the client must be notified, but the request must still be removed from the queue so the next request can be sent. The next request (the oldest request in the queue) must be sent. Refer to the `jsf.ajax.request` JavaScript documentation for more specifics about the Ajax request queue.[P1-end]

13.3.3 Request Callback Function

The Ajax request callback function is called when the Ajax request/response interaction is complete. [P1-start-callback]This function must perform the following actions:

- If the return status is >= 200 and < 300, send a “complete” event following Section 13.3.5.3 “Sending Events”. Call `jsf.ajax.response` passing the Ajax `request` object (for example the XMLHttpRequest instance) and the request `context` (containing the `source` DOM element, `onevent` event function callback and `onerror` error function callback).

- If the return status is outside the range mentioned above, send a “complete” event following Section 13.3.5.3 “Sending Events”. Send an “httpError” error following Section 13.3.6.3 “Signaling Errors”.
- Regardless of whether the request completed successfully or not:
  - remove the completed requests (Ajax readystate 4) from the request queue (dequeue) - specifically the requests that have been on the queue the longest.
  - find the next oldest unprocessed (Ajax readystate 0) request on the queue, and send it. The implementation must ensure that the request that is sent does not enter the queue again.[P1-end]

Refer to Section 13.3.4 “Receiving The Ajax Response”. Also refer to the `jsf.ajax.request` JavaScript documentation for more specifics about the request callback function.

13.3.4 Receiving The Ajax Response

The `jsf.ajax.response` function is responsible for examining the markup that is returned from the server and updating the client side DOM. The Ajax request callback function should call this function when a request completes successfully. [P1-start-ajaxresponse]The implementation of `jsf.ajax.response` must handle the response as outlined in the JavaScript documentation for `jsf.ajax.response.` The elements in the response must be processed in the order they appear in the response.[P1-end]

13.3.5 Monitoring Events On The Client

JavaScript functions can be registered to be notified during various stages of the Ajax request/response cycle. Functions can be set up to monitor individual Ajax requests, and functions can also be set up to monitor all Ajax requests.

13.3.5.1 Monitoring Events For An Ajax Request

There are two ways to monitor events for a single Ajax request by registering an event callback function:

- By using the `<f:ajax>` tag with the `onevent` attribute.
- By using the JavaScript API function `jsf.ajax.request` with `onevent` as an option.

Refer to Section 10.4.1.1 “<f:ajax>” for details on the use of the `<f:ajax>` tag approach. Refer to Section 14.2 “Initiating an Ajax Request” for details about using the `jsf.ajax.request` function approach. [P1-start-event-request]The implementation must ensure the JavaScript function that is registered for an Ajax request must be called in accordance with the events outlined in Section TABLE 14-3 “Events”.[P1-end]

13.3.5.2 Monitoring Events For All Ajax Requests

The JavaScript API provides the `jsf.ajax.addOnEvent` function that can be used to register a JavaScript function that will be notified when any Ajax request/response event occurs. Refer to Section 14.4 “Registering Callback Functions” for more details. The `jsf.ajax.addOnEvent` function accepts a JavaScript function argument that will be notified when events occur during any Ajax request/response event cycle. [P1-start-event] The implementation must ensure the JavaScript function that is registered must be called in accordance with the events outlined in Section TABLE 14-3 “Events”.[P1-end]

13.3.5.3 Sending Events

[P1-start-event-send]The implementation must send events to the runtime as follows:

- Construct a data payload for events using the properties described in Section TABLE 14-4 “Event Data Payload”.

- If an event handler function was registered with the “onevent” attribute (Section 13.3.5.1 “Monitoring Events For An Ajax Request”) call it passing the data payload.
- If any event handling functions were registered with the “addOnEvent” function (Section 13.3.5.2 “Monitoring Events For All Ajax Requests”) call them passing the data payload.[P1-end]

13.3.6 Handling Errors On the Client

JavaScript functions can be registered to be notified when Ajax requests complete with error status codes from the server to give implementations a chance to handle the errors. Functions can be set up to handle errors from individual Ajax requests and functions can be setup to handle errors for all Ajax requests.

13.3.6.1 Handling Errors For An Ajax Request

There are two ways to handle errors for a single Ajax request by registering an error callback function:

- By using the `<f:ajax>` tag with the `onerror` attribute.
- By using the JavaScript API function `jsf.ajax.request` with `onerror` as an option.

Refer to Section 10.4.1.1 “<f:ajax>” for details on the use of the `<f:ajax>` tag approach. Refer to Section 14.2 “Initiating an Ajax Request” for details about using the `jsf.ajax.request` function approach. [P1-start-event-request]The implementation must ensure the JavaScript function that is registered for an Ajax request must be called in accordance when the request status code from the server is as outlined in Section TABLE 14-5 “Errors”.[P1-end]

13.3.6.2 Handling Errors For All Ajax Requests

The JavaScript API provides the `jsf.ajax.addOnError` function that can be used to register a JavaScript function that will be notified when an error occurs for any Ajax request/response. Refer to Section 14.4 “Registering Callback Functions” for more details. The `jsf.ajax.addOnError` function accepts a JavaScript function argument that will be notified when errors occur during any Ajax request/response cycle. [P1-start-event] The implementation must ensure the JavaScript function that is registered must be called in accordance with the errors outlined in Section TABLE 14-5 “Errors”.[P1-end]

13.3.6.3 Signaling Errors

[P1-start-error-signal]The implementation must signal errors to the runtime as follows:

- Construct a data payload for errors using the properties described in Section TABLE 14-6 “Error Data Payload”.
- If an error handler function was registered with the “onerror” attribute (Section 13.3.6.1 “Handling Errors For An Ajax Request”) call it passing the data payload.
- If any error handling functions were registered with the “addOnError” function (Section 13.3.6.2 “Handling Errors For All Ajax Requests”) call them passing the data payload.
- If the project stage is “development” (see Section 14.5 “Determining An Application’s Project Stage”) use JavaScript “alert” to signal the error(s).[P1-end]

13.3.7 Handling Errors On The Server

JavaServer Faces handles exceptions on the server as outlined in Section 6.2 “ExceptionHandler”. [P1-start-error-server]JavaServer Faces Ajax frameworks must ensure exception information is written to the response in the format:

```
<partial-response>
  <error>
    <error-name>...</error-name>
    <error-message>...</error-message>
  </error>
</partial-response>
```

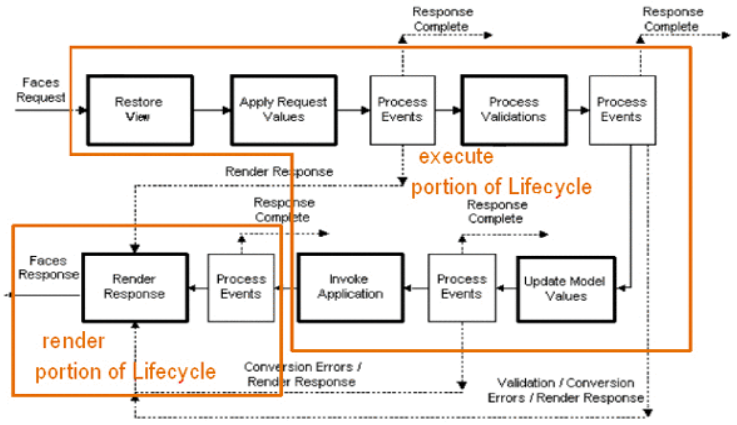
- Extract the “class” from the “Throwable” object and write that as the contents of `error-name` in the response.
- Extract the “cause” from the “Throwable” object if it is available and write that as the contents of `error-message` in the response. If “cause” is not available, write the string returned from “Throwable.getMessage()”.

Implementations must ensure that an ExceptionHandler suitable for writing exceptions to the partial response is installed if the current request required an Ajax response (PartialViewContext.isAjaxRequest() returns true).[P1-end]

Implementations may choose to include a specialized ExceptionHandler for Ajax that extends from `javax.faces.context.ExceptionHandlerWrapper`, and have the `javax.faces.context.ExceptionHandlerFactory` implementation install it if the environment requires it.

13.4 Partial View Traversal

The JavaServer Faces lifecycle, can be viewed as consisting of an `execute` phase and a `render` phase.



Partial traversal is the technique that can be used to “visit” one or more components in the view, potentially to have them pass through the “execute” and/or “render” phases of the request processing lifecycle. This is a key feature for JSF and Ajax frameworks and it allows selected components in the view to be processed and/or rendered. There are a variety of JSF Ajax frameworks available, and they all perform some variation of partial traversal.

13.4.1 Partial Traversal Strategy

Frameworks use a partial traversal strategy to perform partial view processing and partial view rendering. This specification does not dictate the use of a specific partial traversal strategy. However, frameworks must implement their desired strategy by implementing the PartialViewContext.processPartial method. Refer to the JavaDocs for details about this method.

13.4.2 Partial View Processing

Partial view processing allows selected components to be processed through the "execute" portion of the lifecycle. Although the diagram in Section 13.4 "Partial View Traversal" depicts the "execute" portion as encompassing everything except the "Render Response Phase", it really is the "Apply Request Values Phase", "Update Model Values Phase" and "Process Validations Phase". Partial view processing on the server is triggered by a request from the client. The request does not have to be an Ajax request. The request contains special parameters that indicate the request is a partial `execute` request or a partial `execute` request that was triggered using Ajax. The client also sends a set of client ids of the components that must be processed through the `execute` phase of the request processing lifecycle. Refer to Section 13.3.1 "Sending an Ajax Request" about the request sending details. The `FacesContext` has methods for retrieving the `PartialViewContext` instance for the request. The `PartialViewContext` has properties and methods that indicate the request is a partial request based on the values of these special request parameters. Refer to the JavaDocs for javax.faces.context.PartialViewContext and Section 6.1.11 "Partial View Context" for the specifics of the `PartialViewContext` constants and methods that facilitate partial processing. [P1-start-partialExec]The `UIViewRoot processDecodes, processValidators and processUpdates` methods must determine if the request is a partial request using the `FacesContext.isPartialRequest()` method. If `FacesContext.isPartialRequest()` returns `true`, then the implementation of these methods must retrieve a `PartialViewContext` instance and invoke `PartialViewcontext.processPartial.` Refer to `Section 2.2.2 "Apply Request Values", Section 2.2.2.1 "Apply Request Values Partial Processing", Section 2.2.3 "Process Validations", Section 2.2.3.1 "Partial Validations Partial Processing", Section 2.2.4 "Update Model Values", Section 2.2.4.1 "Update Model Values Partial Processing".`[P1-end]

13.4.3 Partial View Rendering

Partial view rendering on the server is triggered by a request from the client. It allows one or more components in the view to perform the encoding process. The request contains special parameters that indicate the request is a partial `render` request. The client also sends a set of client ids of the components that must be processed by the `render` phase of the request processing lifecycle. Refer to Section 13.3.1 "Sending an Ajax Request" about the request sending details. The `FacesContext` has methods that indicate the request is a partial request based on the values of these special request parameters. Refer to Section 6.1.10 "Partial Processing Methods" for the specifics of the `FacesContext` constants and methods that facilitate partial processing. [P1-start-partialRender]The `UIViewRoot getRendersChildren` and `encodeChildren` methods must determine if the request is an Ajax request using the `FacesContext.isAjaxRequest()` method. If `FacesContext.isAjaxRequest()` returns `true`, then the `getRendersChildren` method must return `true` and the `encodeChildren` method must perform partial rendering using the `PartialViewContext.processPartial` implementation. Refer to the JavaDocs for `UIViewRoot.encodeChildren` for specific details.[P1-end]

13.4.4 Sending The Response to The Client

The Ajax response (also known as partial response) is formulated and sent to the client during the Render Response phase of the request processing lifecycle. The partial response consists of markup rendered by one or more components. The response should be in a common format so JavaScript clients can interpret the markup in a consistent way - an important requirement for component compatability. The agreed upon format and content type for the partial response is XML. This means there should be a `ResponseWriter` suitable for writing the response in XML. The

`UIViewRoot.encodeChildren` method delegates to a partial traversal strategy. The partial traversal strategy implementation produces the partial response. The markup that is sent to the client must contain elements that the client can recognize. In addition to the markup produced by server side components, the response must contain "instructions" for the client to interpret, so the client will know, for example, that it is to add new markup to the client DOM, or update existing areas of the DOM. When the response is sent back to the client, it must contain the view state. [P1-start-sending-response]Implementations must adhere to the response format as specified in the JavaScript docs for jsf.ajax.response.[P1-end] Refer to the XML schema definition in the Section 1.2 "XML Schema Definition for Ajax Response" section. This XML schema is another important area for component library compatability.

13.4.4.1 Writing The Partial Response

JavaServer Faces provides `javax.faces.context.PartialResponseWriter` to ensure the Ajax response that is written follows the standard format as specified in Section 1.2 "XML Schema Definition for Ajax Response". Implementations must take care to properly handle nested CDATA sections when writing the response. `PartialResponseWriter` decorates an existing ResponseWriter implementation by extending `javax.faces.context.ResponseWriterWrapper.` Refer to the `javax.faces.context.PartialResponseWriter` JavaDocs, and the JavaScript documentation for the `jsf.ajax.response` function for more specifics.

14

JavaScript API

This chapter of the specification describes the JavaScript functions that are used to facilitate Ajax operations in a JavaServer Faces framework. All of these functions are contained in the canonical `jsf.js` file.

14.1 Collecting and Encoding View State

In JavaServer Faces 1.2 the `javax.faces.ViewState` parameter was standardized to facilitate "postback" requests to the server in a JavaServer Faces application. Implementations must use this parameter to save the view state between requests. Refer to the Javadocs for `javax.faces.render.ResponseStateManager.`

Collecting and encoding view state that will be sent to the server is a common operation used by most JavaServer Faces Ajax frameworks. When a JavaServer Faces view is rendered, it will contain a hidden field with the identifier `javax.faces.ViewState` whose value contains the state for the current view. JSF Ajax clients collect additional view state, combine it with the current view state and send it's encoded form to the server.

```
jsf.getViewState(FORM_ELEMENT)
```

Collect and encode element data for the given `FORM_ELEMENT` and return it as the view state that will be sent to the server. `FORM_ELEMENT` is the identifier for a DOM form element. All input elements of type "hidden" should be included in the collection and encoding process.

- Encode the name and value for each input element of `FORM_ELEMENT.` Only select elements that have at least one of their options selected must be included. only checkbox elements that are checked must be included.
- Find the element identified as `javax.faces.ViewState` in the specified `FORM_ELEMENT` and encode the name and value.
- Return a concatenated String of the encoded input elements and `javax.faces.ViewState` element.

14.1.1 Use Case

Collect and Encode Elements Of a Form

```
var viewState = jsf.getViewState(form);
```

14.2 Initiating an Ajax Request

```
jsf.ajax.request(source, |event|, { |OPTIONS| });
```

The `jsf.ajax.request` function is responsible for sending an Ajax request to the server. [P1-start-ajaxrequest]The requirements for this function are as follows:

- The request must be sent asynchronously
- The request must be sent with method type POST
- The request URL will be the `form action` attribute
- All requests will be queued with the use of a client side request queue to help ensure request ordering
- [P1-end]

14.2.1 Usage

Typically, this function is attached as a JavaScript event handler (such as "onclick").

```
<ANY_HTML_OR_JSF_ELEMENT
on|EVENT|="jsf.ajax.request(source, event,
{ |OPTIONS| });" />
```

The function arguments are as follows:

`source` is the DOM element that triggered this Ajax request. [P1-start-source]It must be a DOM element object or a string identifier for a DOM element. [P1-end]The `event` argument is the JavaScript `event` object. The optional `|OPTIONS|` argument is a JavaScript associative object array that may contain the following name/value pairs:

TABLE 14-1 request `OPTIONS`

Name	Value
execute	A space delimited list of client identifiers or one of the keywords (Section 14.2.2 "Keywords"). These reference the components that will be processed during the "execute" phase of the request processing lifecycle.
render	A space delimited list of client identifiers or one of the keywords (Section 14.2.2 "Keywords"). These reference the components that will be processed during the "render" phase of the request processing lifecycle.
onevent	A String that is the name of the JavaScript function to call when an event occurs.
onerror	A String that is the name of the JvaScript function to call when an error occurs.
params	An object that may include additional parameters to include in the request.

14.2.2 Keywords

The following keywords can be used for the value of the “execute” and “render” attributes:

TABLE 14-2 Execute / Render Keywords

Keyword	Description
@all	All component identifiers
@none	No identifiers
@this	The element that triggered the request
@form	The enosing form

14.2.3 Default Values

Values for the `execute` **`and`** `render` attributes are not required. When using the JavaScript API, the default values for `execute is @this. The default value for render` is `@none.`

```
<h:commandButton id=”button1” value=”submit”>
onclick="jsf.ajax.request(this,event);" />
is the same as:
<h:commandButton id=”button1” value=”submit”>
onclick="jsf.ajax.request(this,event,
{execute:’@this’,render:’@this’});" />
```

```
<h:commandButton id=”button1” value=”submit”>
onclick="jsf.ajax.request(this,event, {execute:’@this’});" />
is the same as:
<h:commandButton id=”button1” value=”submit”>
onclick="jsf.ajax.request(this,event, {execute:’button1’});" />
```

Refer to Section 10.4.1.1 “<f:ajax>” for the default values for the `execute` **`and`** `render` attributes when they are used with the core “<f:ajax>” tag.

14.2.4 Request Sending Specifics

The mechanics of sending an Ajax request becomes very important to promote component compatability. Even more important, is standardizing on the post data that is sent to server implementations, so they all can expect the same arguments. [P1-start-ajaxrequest-send]The request header must be set with the name `Faces-Request` and the value `partial/ajax.` Specifics of formulating post data and sending the request must be followed as outlined in the JavaScript documentation for the `jsf.ajax.request` function. The post data arguments that must be sent are:

Name	**Value**
javax..faces.ViewState	The value of the javax.faces.ViewState hidden field. This is included when using the jsf.getViewState function.
javax.faces.partial.ajax	true
javax.faces.source	The identifier of the element that is the source of this request

- [P1-end]

14.2.5 Use Case

```
<h:commandbutton id="submit" value="submit"
   onclick="jsf.ajax.request(this, event,
   {execute:'submit',render:'outtext'}); return false;" />
```

This use case assumes there is another component in the view with the identifier `outtext.`

14.3 Processing The Ajax Response

```
jsf.ajax.response(request, context);
```

The `jsf.ajax.response` function is called when a request completes successfully. This typically means that returned status code is >= 200 and < 300. The `jsf.ajax.response` function must extract the XML response from the `request` argument. The XML response is expected to follow the format that is outlined in the JavaScript documentation for this function. The response format is an "instruction set" telling this function how it should update the DOM. The `context` argument contains properties that facilitate event and error processing such as the `source` DOM element (the DOM element that triggered the Ajax request), `onevent` (the event handling callback for the request) and `onerror` (the error handling callback for the request). [P1-start-ajaxresponse] The specifics details of this function's operation must follow the `jsf.ajax.response` JavaScript documentation.[P1-end]

14.4 Registering Callback Functions

The JavaScript API allows you to register callback functions for Ajax request/response event monitoring and error handling. The event callbacks become very useful when monitoring request connection status. The error callback provides a convenient way for implementions to trap errors. The handling of the errors is left up to the implementation. These callback function names can also be set using the JavaScript API (Section TABLE 14-1 "request OPTIONS"), and the core <f:ajax> tag (Section 10.4.1.1 "<f:ajax>").

14.4.1 Request/Response Event Handling

```
jsf.ajax.addOnEvent(callback);
```

The `callback` argument must be a reference to an existing JavaScript function that will handle the events. The events that can be handled are:

TABLE 14-3 Events

Event Name	Description
begin	Occurs immediately before the request is sent.
success	Occurs immediately after jsf.ajax.response has completed.
complete	Occurs immediately after the request has completed. For successful requests, this is immediately before javax.faces.response is called. For unsuccessful requests, this is immediately before the error handling callback is invoked.

The callback function has access to the following “data payload”:.

TABLE 14-4 Event Data Payload

Name	Description/Value
type	“event”
name	Callback function name
source	The DOM element that triggered the Ajax request.
responseCode	Ajax request object ‘status’ (XMLHttpRequest.status); Not present for “begin” event;
responseXML	The XML response (XMLHttpRequest.responseXML); Not present for “begin” event;
responseTxt	The text response (XMLHttpResponse.responseTxt) Not present for “begin” event;

14.4.1.1 Use Case

```
jsf.ajax.addOnEvent(statusUpdate);
...
var statusUpdate = function statusUpdate(data) {
... do something with “data payload” ...
}
```

14.4.2 Error Handling

```
jsf.ajax.addOnError(callback);
```

The `callback` argument must be a reference to an existing JavaScript function that will handle errors from the server.

TABLE 14-5 Errors

Error Name	Description
httpError	request status==null or request.status==undefined or request.status<200 or request.status >=300
serverError	The Ajax response contains an “error” element.
malformedXML	The Ajax response does not follow the proper format. See Section 1.2 “XML Schema Definition for Ajax Response”
emptyResponse	There was no Ajax response from the server.

The callback function has access to the following “data payload”:.

TABLE 14-6 Error Data Payload

Name	Description/Value
type	“error”
name	One of error names defined TABLE 14-5
source	The DOM element that triggered the Ajax request.
responseCode	Ajax request object ‘status’ (XMLHttpRequest.status);
responseXML	The XML response (XMLHttpRequest.responseXML)
responseTxt	The text response (XMLHttpResponse.responseTxt)
errorName	The error name taken from the Ajax response “error” element.
errorMessage	The error messages taken from the Ajax response “error” element.

14.4.2.1 Use Case

```
jsf.ajax.addOnError(handleError);
...
var handleError = function handleError(data) {
... do something with “data payload” ...
}
```

14.5 Determining An Application's Project Stage

```
jsf.getProjectStage();
```

[P1-start-projStage]This function must return the constant representing the current state of the running application in a typical product development lifecycle. The returned value must be the value returned from the server side method `javax.faces.application.Application.getProjectStage();` Refer to Section 7.1.8 "ProjectStage Property" for more details about this property.[P1-end]

14.5.1 Use Case

```
var projectStage = javax.faces.Ajax.getProjectStage();
if (projectStage == "Production") {
.... throw exception
else if (projectStage == "Development") {
.... send an alert for debugging
}
```

14.6 Script Chaining

```
jsf.util.chain(source, event, |<script>, <script>,...|)
```

This utility function invokes an arbitrary number of scripts in sequence. If any of the scripts return false, subsequent scripst will not be executed. The arguments are:

- source - The DOM element that triggered this Ajax request, or an id string of the element to use as the triggering element.
- event - The DOM event that triggered this Ajax request. A value does not have to be specified for this argument.

The variable number of script arguments follow the source and event arguments. Refer to the JavaScript API documentation in the source for more details.

A

Appendix A - JSF Metadata

This chapter lists the latest XML Schema definition and Document Type Definition for JSF metadata and config files.

1.1 XML Schema Definition for Application Configuration Resource file

```
<xsd:schema
      targetNamespace="http://java.sun.com/xml/ns/javaee"
      xmlns:javaee="http://java.sun.com/xml/ns/javaee"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:xml="http://www.w3.org/XML/1998/namespace"
      elementFormDefault="qualified"
      attributeFormDefault="unqualified"
      version="2.0">

    <xsd:annotation>
        <xsd:documentation>
            $Id: web-facesconfig_2_0.xsd,v 1.1.8.2 2008/03/20 21:12:50 edburns Exp $
        </xsd:documentation>
    </xsd:annotation>

    <xsd:annotation>
        <xsd:documentation>

            Copyright 2007 Sun Microsystems, Inc.,
            901 San Antonio Road,
            Palo Alto, California 94303, U.S.A.
```

```
    </xsd:documentation>
</xsd:annotation>

<xsd:annotation>
    <xsd:documentation>

        <![CDATA[
```

```
          The XML Schema for the JavaServer Faces Application
          Configuration File (Version 2.0).

          All JavaServer Faces configuration files must indicate
          the JavaServer Faces schema by indicating the JavaServer
          Faces namespace:

          http://java.sun.com/xml/ns/javaee

          and by indicating the version of the schema by
          using the version element as shown below:

          <faces-config xmlns="http://java.sun.com/xml/ns/javaee"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:schemaLocation="..."
                    version="2.0">
               ...
          </faces-config>

          The instance documents may indicate the published
          version of the schema using xsi:schemaLocation attribute
          for javaee namespace with the following location:

          http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd

          ]]>

     </xsd:documentation>
</xsd:annotation>

<xsd:include schemaLocation="javaee_5.xsd"/>

<!-- ************************************************** -->
```

```
<xsd:element name = "faces-config" type="javaee:faces-configType">
     <xsd:annotation>
          <xsd:documentation>

               The "faces-config" element is the root of the configuration
               information hierarchy, and contains nested elements for all
               of the other configuration settings.

          </xsd:documentation>
     </xsd:annotation>

     <xsd:unique name="faces-config-behavior-ID-uniqueness">
          <xsd:annotation>
               <xsd:documentation>

                    Behavior IDs must be unique within a document.

               </xsd:documentation>
          </xsd:annotation>

          <xsd:selector xpath="javaee:behavior"/>
          <xsd:field    xpath="javaee:behavior-id"/>
     </xsd:unique>

     <xsd:unique name="faces-config-converter-ID-uniqueness">
          <xsd:annotation>
               <xsd:documentation>

                    Converter IDs must be unique within a document.

               </xsd:documentation>
          </xsd:annotation>
```

```
     <xsd:selector xpath="javaee:converter"/>
     <xsd:field    xpath="javaee:converter-id"/>
</xsd:unique>

<xsd:unique name="faces-config-converter-for-class-uniqueness">
     <xsd:annotation>
          <xsd:documentation>

               'converter-for-class' element values must be unique
               within a document.

          </xsd:documentation>
     </xsd:annotation>

     <xsd:selector xpath="javaee:converter"/>
     <xsd:field    xpath="javaee:converter-for-class"/>
</xsd:unique>

<xsd:unique name="faces-config-validator-ID-uniqueness">
     <xsd:annotation>
          <xsd:documentation>

               Validator IDs must be unique within a document.

          </xsd:documentation>
     </xsd:annotation>

     <xsd:selector xpath="javaee:validator"/>
     <xsd:field    xpath="javaee:validator-id"/>
</xsd:unique>

<xsd:unique name="faces-config-managed-bean-name-uniqueness">
     <xsd:annotation>
          <xsd:documentation>
```

```
                    Managed bean names must be unique within a document.

                </xsd:documentation>
            </xsd:annotation>

            <xsd:selector xpath="javaee:managed-bean"/>
            <xsd:field    xpath="javaee:managed-bean-name"/>
        </xsd:unique>
    </xsd:element>

    <!-- ************************************************** -->

    <xsd:complexType name="faces-configType">
        <xsd:annotation>
            <xsd:documentation>

                The "faces-config" element is the root of the configuration
                information hierarchy, and contains nested elements for all
                of the other configuration settings.

            </xsd:documentation>
        </xsd:annotation>
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="application"
                         type="javaee:faces-config-applicationType"/>
            <xsd:element name="ordering"
                         type="javaee:faces-config-orderingType"/>
            <xsd:element name="absolute-ordering"
                         type="javaee:faces-config-absoluteOrderingType"/>
            <xsd:element name="factory"
                         type="javaee:faces-config-factoryType"/>
            <xsd:element name="component"
                         type="javaee:faces-config-componentType"/>
```

```
<xsd:element name="converter"
               type="javaee:faces-config-converterType"/>
<xsd:element name="managed-bean"
               type="javaee:faces-config-managed-beanType"/>
<xsd:element name="name"
               type="javaee:java-identifierType"
               minOccurs="0"
               maxOccurs="1">
  <xsd:annotation>
       <xsd:documentation>

            The "name" element within the top level "faces-config"
            element declares the name of this application
            configuration resource.  Such names are used
            in the document ordering scheme specified in section
            JSF.11.4.6.

       </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<xsd:element name="navigation-rule"
               type="javaee:faces-config-navigation-ruleType"/>
<xsd:element name="referenced-bean"
               type="javaee:faces-config-referenced-beanType"/>
<xsd:element name="render-kit"
               type="javaee:faces-config-render-kitType"/>
<xsd:element name="lifecycle"
               type="javaee:faces-config-lifecycleType"/>
<xsd:element name="validator"
               type="javaee:faces-config-validatorType"/>
<xsd:element name="behavior"
               type="javaee:faces-config-behaviorType"/>
<xsd:element name="faces-config-extension"
```

```
                    type="javaee:faces-config-extensionType"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
</xsd:choice>
<xsd:attribute name="metadata-complete"
                  type="xsd:boolean"
                  use="optional">
    <xsd:annotation>
        <xsd:documentation>

            The metadata-complete attribute defines whether this
            JavaServer Faces application is complete, or whether
            the class files available to this module and packaged with
            this application should be examined for annotations
            that specify configuration information.

            This attribute is only inspected on the application
            configuration resource file located at "WEB-INF/faces-config.xml".
            The presence of this attribute on any application configuration
            resource other than the one located at "WEB-INF/faces-config.xml",
            including any files named using the javax.faces.CONFIG_FILES
            attribute, must be ignored.

            If metadata-complete is set to "true", the JavaServer Faces
            runtime must ignore any annotations that specify configuration
            information, which might be present in the class files
            of the application.

            If metadata-complete is not specified or is set to
            "false", the JavaServer Faces runtime must examine the class
            files of the application for annotations, as specified by
            the specification.

            If "WEB-INF/faces-config.xml" is not present, the JavaServer
```

```
                    Faces runtime will assume metadata-complete to be "false".

                    The value of this attribute will have no impact on
                    runtime annotations such as @ResourceDependency or
                    @ListenerFor.
                </xsd:documentation>
            </xsd:annotation>
        </xsd:attribute>
        <xsd:attribute name="id" type="xsd:ID" />
        <xsd:attribute name="version"
                       type="javaee:faces-config-versionType"
                       use="required"/>
</xsd:complexType>

<!-- ********************************************** -->

<xsd:complexType name = "faces-config-extensionType">
        <xsd:annotation>
            <xsd:documentation>

                Extension element for faces-config.  It may contain
                implementation specific content.

            </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
            <xsd:any namespace="##any"
                      processContents="lax"
                      minOccurs="0"
                      maxOccurs="unbounded" />
         </xsd:sequence>
         <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
```

```
<!-- ************************************************** -->


<xsd:complexType name="faces-config-orderingType">
     <xsd:annotation>
          <xsd:documentation>

            Please see section JSF.11.4.6 for the specification of this element.

          </xsd:documentation>
     </xsd:annotation>
     <xsd:sequence>
          <xsd:element name="after"
                         type="javaee:faces-config-ordering-orderingType"
                         minOccurs="0"
                         maxOccurs="1"/>
          <xsd:element name="before"
                         type="javaee:faces-config-ordering-orderingType"
                         minOccurs="0"
                         maxOccurs="1"/>
     </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="faces-config-ordering-orderingType">
     <xsd:annotation>
          <xsd:documentation>

               RELEASE_PENDING (edburns,rogerk) review docs
               This element contains a sequence of "id" elements, each of which
               refers to an application configuration resource by the "id"
               declared on its faces-config element.  This element can also contain
               a single "others" element which specifies that this document comes
```

```
                before or after other documents within the application.


            </xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
           <xsd:element name="name" type="javaee:java-identifierType" minOccurs="0" maxOccurs=
"unbounded"/>
            <xsd:element name="others" type="javaee:faces-config-ordering-othersType"
minOccurs="0" maxOccurs="1" />
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="faces-config-ordering-othersType">
        <xsd:annotation>
            <xsd:documentation>

                This element indicates that the ordering sub-element in which
                it was placed should take special action regarding the ordering
                of this application resource relative to other
                application configuration resources.  See section JSF.11.4.6
                for the complete specification.

            </xsd:documentation>
        </xsd:annotation>
        <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>


    <!-- ********************************************** -->


    <xsd:complexType name="faces-config-absoluteOrderingType">
        <xsd:annotation>
            <xsd:documentation>
```

```
                Only relevant if this is placed within the /WEB-INF/faces-config.xml.
                Please see section JSF.11.4.6 for the specification for details.

            </xsd:documentation>
        </xsd:annotation>
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
           <xsd:element name="name" type="javaee:java-identifierType" minOccurs="0" maxOccurs=
"unbounded"/>
            <xsd:element name="others" type="javaee:faces-config-ordering-othersType"
minOccurs="0" maxOccurs="1" />
        </xsd:choice>
    </xsd:complexType>



    <!-- ********************************************** -->



    <xsd:complexType name="faces-config-applicationType">
        <xsd:annotation>
            <xsd:documentation>

                The "application" element provides a mechanism to define the
                various per-application-singleton implementation artifacts for
                a particular web application that is utilizing
                JavaServer Faces.  For nested elements that are not specified,
                the JSF implementation must provide a suitable default.

            </xsd:documentation>
        </xsd:annotation>

        <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="action-listener"
                         type="javaee:fully-qualified-classType">
                <xsd:annotation>
                    <xsd:documentation>
```

```
                    The "action-listener" element contains the fully
                    qualified class name of the concrete
                    ActionListener implementation class that will be
                    called during the Invoke Application phase of the
                    request processing lifecycle.

          </xsd:documentation>
     </xsd:annotation>
</xsd:element>
<xsd:element name="default-render-kit-id"
               type="javaee:string">
     <xsd:annotation>
          <xsd:documentation>

                    The "default-render-kit-id" element allows the
                    application to define a renderkit to be used other
                    than the standard one.

          </xsd:documentation>
     </xsd:annotation>
</xsd:element>
<xsd:element name="message-bundle"
               type="javaee:string">
     <xsd:annotation>
          <xsd:documentation>

                    The base name of a resource bundle representing
                    the message resources for this application.  See
                    the JavaDocs for the "java.util.ResourceBundle"
                    class for more information on the syntax of
                    resource bundle names.

          </xsd:documentation>
```

```
        </xsd:annotation>
    </xsd:element>
    <xsd:element name="navigation-handler"
                 type="javaee:fully-qualified-classType">
        <xsd:annotation>
            <xsd:documentation>

                The "navigation-handler" element contains the
                fully qualified class name of the concrete
                NavigationHandler implementation class that will
                be called during the Invoke Application phase
                of the request processing lifecycle, if the
                default ActionListener (provided by the JSF
                implementation) is used.

            </xsd:documentation>
        </xsd:annotation>
    </xsd:element>
    <xsd:element name="partial-traversal"
                 type="javaee:fully-qualified-classType">
        <xsd:annotation>
            <xsd:documentation>

                The "partial-traversal" element contains the fully
                qualified class name of the concrete
                PartialTraversal implementation class that will be
                called during the "execute" and "render" phases of the
                request processing lifecycle.

            </xsd:documentation>
        </xsd:annotation>
    </xsd:element>
    <xsd:element name="view-handler"
                 type="javaee:fully-qualified-classType">
```

```
        <xsd:annotation>
            <xsd:documentation>

                The "view-handler" element contains the fully
                qualified class name of the concrete ViewHandler
                implementation class that will be called during
                the Restore View and Render Response phases of the
                request processing lifecycle.  The faces
                implementation must provide a default
                implementation of this class.

            </xsd:documentation>
        </xsd:annotation>
    </xsd:element>
    <xsd:element name="state-manager"
                 type="javaee:fully-qualified-classType">
        <xsd:annotation>
            <xsd:documentation>

                The "state-manager" element contains the fully
                qualified class name of the concrete StateManager
                implementation class that will be called during
                the Restore View and Render Response phases of the
                request processing lifecycle.  The faces
                implementation must provide a default
                implementation of this class.

            </xsd:documentation>
        </xsd:annotation>
    </xsd:element>
    <xsd:element name="el-resolver"
                 type="javaee:fully-qualified-classType">
        <xsd:annotation>
            <xsd:documentation>
```

```
                    The "el-resolver" element contains the fully
                    qualified class name of the concrete
                    javax.el.ELResolver implementation class
                    that will be used during the processing of
                    EL expressions.

            </xsd:documentation>
        </xsd:annotation>
    </xsd:element>
    <xsd:element name="property-resolver"
                 type="javaee:fully-qualified-classType">
        <xsd:annotation>
            <xsd:documentation>

                    The "property-resolver" element contains the fully
                    qualified class name of the concrete
                    PropertyResolver implementation class that will
                    be used during the processing of value binding
                    expressions.

            </xsd:documentation>
        </xsd:annotation>
    </xsd:element>
    <xsd:element name="variable-resolver"
                 type="javaee:fully-qualified-classType">
        <xsd:annotation>
            <xsd:documentation>

                    The "variable-resolver" element contains the fully
                    qualified class name of the concrete
                    VariableResolver implementation class that will
                    be used during the processing of value binding
                    expressions.
```

```
          </xsd:documentation>
     </xsd:annotation>
</xsd:element>
<xsd:element name="resource-handler"
               type="javaee:fully-qualified-classType">
     <xsd:annotation>
          <xsd:documentation>
               <![CDATA[

                    The "resource-handler" element contains the
                    fully qualified class name of the concrete
                    ResourceHandler implementation class that
                    will be used during rendering and decoding
                    of resource requests The standard
                    constructor based decorator pattern used for
                    other application singletons will be
                    honored.

               ]]>

          </xsd:documentation>
     </xsd:annotation>
</xsd:element>
<xsd:element name="system-event-listener"
               type="javaee:faces-config-system-event-listenerType"
               minOccurs="0"
               maxOccurs="unbounded">
</xsd:element>
<xsd:element
     name="locale-config"
     type="javaee:faces-config-locale-configType"/>
<xsd:element
     name="resource-bundle"
```

```
            type="javaee:faces-config-application-resource-bundleType"/>
        <xsd:element name="application-extension"
                        type="javaee:faces-config-application-extensionType"
                        minOccurs="0"
                        maxOccurs="unbounded"/>
        <xsd:element
            name="default-validators"
            type="javaee:faces-config-default-validatorsType"/>
        </xsd:choice>
    <xsd:attribute name = "id" type = "xsd:ID"/>
</xsd:complexType>

<xsd:complexType name="faces-config-application-resource-bundleType">
    <xsd:annotation>
        <xsd:documentation>

          The resource-bundle element inside the application element
          references a java.util.ResourceBundle instance by name
          using the var element.  ResourceBundles referenced in this
          manner may be returned by a call to
          Application.getResourceBundle() passing the current
          FacesContext for this request and the value of the var
          element below.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
        <xsd:group ref="javaee:descriptionGroup"/>
        <xsd:element name="base-name"
                        type="javaee:fully-qualified-classType">
            <xsd:annotation>
                <xsd:documentation>
```

```
                    The fully qualified class name of the
                    java.util.ResourceBundle instance.

                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="var"
                     type="javaee:string">
            <xsd:annotation>
                <xsd:documentation>

                    The name by which this ResourceBundle instance
                    is retrieved by a call to
                    Application.getResourceBundle().

                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ********************************************** -->

<xsd:complexType name = "faces-config-application-extensionType">
    <xsd:annotation>
        <xsd:documentation>

            Extension element for application.  It may contain
            implementation specific content.

        </xsd:documentation>
    </xsd:annotation>
```

```
    <xsd:sequence>
        <xsd:any namespace="##any"
                 processContents="lax"
                 minOccurs="0"
                 maxOccurs="unbounded" />
     </xsd:sequence>
     <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>


<!-- ************************************************* -->

<xsd:complexType name = "faces-config-factoryType">
    <xsd:annotation>
        <xsd:documentation>

            The "factory" element provides a mechanism to define the
            various Factories that comprise parts of the implementation
            of JavaServer Faces.  For nested elements that are not
            specified, the JSF implementation must provide a suitable
            default.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="application-factory"
                      type="javaee:fully-qualified-classType">
            <xsd:annotation>
                <xsd:documentation>

                    The "application-factory" element contains the
                    fully qualified class name of the concrete
                    ApplicationFactory implementation class that will
```

```
                be called when
                FactoryFinder.getFactory(APPLICATION_FACTORY) is
                called.

            </xsd:documentation>
        </xsd:annotation>
    </xsd:element>
    <xsd:element name="exception-handler-factory"
                 type="javaee:fully-qualified-classType">
        <xsd:annotation>
            <xsd:documentation>

                The "exception-handler-factory" element contains the
                fully qualified class name of the concrete
                ExceptionHandlerFactory implementation class that will
                be called when
                FactoryFinder.getFactory(EXCEPTION_HANDLER_FACTORY)
                is called.

            </xsd:documentation>
        </xsd:annotation>
    </xsd:element>
    <xsd:element name="external-context-factory"
                 type="javaee:fully-qualified-classType">
        <xsd:annotation>
            <xsd:documentation>

                The "external-context-factory" element contains the
                fully qualified class name of the concrete
                ExternalContextFactory implementation class that will
                be called when
                FactoryFinder.getFactory(EXTERNAL_CONTEXT_FACTORY)
                is called.
```

```
          </xsd:documentation>
     </xsd:annotation>
</xsd:element>

<xsd:element name="faces-context-factory"
               type="javaee:fully-qualified-classType">
     <xsd:annotation>
          <xsd:documentation>

               The "faces-context-factory" element contains the
               fully qualified class name of the concrete
               FacesContextFactory implementation class that will
               be called when
               FactoryFinder.getFactory(FACES_CONTEXT_FACTORY)
               is called.

          </xsd:documentation>
     </xsd:annotation>
</xsd:element>

<xsd:element name="partial-view-context-factory"
               type="javaee:fully-qualified-classType">
     <xsd:annotation>
          <xsd:documentation>

               The "partial-view-context-factory" element contains the
               fully qualified class name of the concrete
               PartialViewContextFactory implementation class that will
               be called when FactoryFinder.getFactory
               (FactoryFinder.PARTIAL_VIEW_CONTEXT_FACTORY) is called.

          </xsd:documentation>
     </xsd:annotation>
</xsd:element>
```

```
<xsd:element name="lifecycle-factory"
             type="javaee:fully-qualified-classType">
     <xsd:annotation>
          <xsd:documentation>

               The "lifecycle-factory" element contains the fully
               qualified class name of the concrete LifecycleFactory
               implementation class that will be called when
               FactoryFinder.getFactory(LIFECYCLE_FACTORY) is called.

          </xsd:documentation>
     </xsd:annotation>
</xsd:element>
<xsd:element name="view-declaration-language-factory"
             type="javaee:fully-qualified-classType">
     <xsd:annotation>
          <xsd:documentation>

               The "view-declaration-language-factory" element contains
               the fully qualified class name of the concrete
               ViewDeclarationLanguageFactory
               implementation class that will be called when
               FactoryFinder.getFactory(VIEW_DECLARATION_FACTORY) is called.

          </xsd:documentation>
     </xsd:annotation>
</xsd:element>
<xsd:element name="tag-handler-delegate-factory"
             type="javaee:fully-qualified-classType">
     <xsd:annotation>
          <xsd:documentation>

               The "tag-handler-delegate-factory" element contains
```

```
                the fully qualified class name of the concrete
                PageDeclarationLanguageFactory
                implementation class that will be called when
                FactoryFinder.getFactory(TAG_HANDLER_DELEGATE_FACTORY) is called.

            </xsd:documentation>
        </xsd:annotation>
    </xsd:element>
    <xsd:element name="render-kit-factory"
                  type="javaee:fully-qualified-classType">
        <xsd:annotation>
            <xsd:documentation>

                The "render-kit-factory" element contains the fully
                qualified class name of the concrete RenderKitFactory
                implementation class that will be called when
                FactoryFinder.getFactory(RENDER_KIT_FACTORY) is
                called.

            </xsd:documentation>
        </xsd:annotation>
    </xsd:element>
    <xsd:element name="visit-context-factory"
                  type="javaee:fully-qualified-classType">
        <xsd:annotation>
            <xsd:documentation>

                The "visit-context-factory" element contains the fully
                qualified class name of the concrete VisitContextFactory
                implementation class that will be called when
                FactoryFinder.getFactory(VISIT_CONTEXT_FACTORY) is
                called.

            </xsd:documentation>
```

```
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="factory-extension"
                     type="javaee:faces-config-factory-extensionType"
                     minOccurs="0"
                     maxOccurs="unbounded"/>
    </xsd:choice>
    <xsd:attribute name = "id" type = "xsd:ID"/>
</xsd:complexType>

<!-- ************************************************** -->

<xsd:complexType name = "faces-config-factory-extensionType">
    <xsd:annotation>
        <xsd:documentation>

            Extension element for factory.  It may contain
            implementation specific content.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
        <xsd:any namespace="##any"
                  processContents="lax"
                  minOccurs="0"
                  maxOccurs="unbounded" />
     </xsd:sequence>
     <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>


<!-- ************************************************** -->
```

```
<xsd:complexType name="faces-config-attributeType">
     <xsd:annotation>
          <xsd:documentation>

               The "attribute" element represents a named, typed, value
               associated with the parent UIComponent via the generic
               attributes mechanism.

               Attribute names must be unique within the scope of the parent
               (or related) component.

          </xsd:documentation>
     </xsd:annotation>

     <xsd:sequence>
          <xsd:group ref="javaee:descriptionGroup"/>
          <xsd:element name="attribute-name"
                       type="javaee:string">
               <xsd:annotation>
                    <xsd:documentation>

                         The "attribute-name" element represents the name under
                         which the corresponding value will be stored, in the
                         generic attributes of the UIComponent we are related
                         to.

                    </xsd:documentation>
               </xsd:annotation>
          </xsd:element>
          <xsd:element name="attribute-class"
                       type="javaee:fully-qualified-classType">
               <xsd:annotation>
                    <xsd:documentation>
```

```
                        The "attribute-class" element represents the Java type
                        of the value associated with this attribute name.

                    </xsd:documentation>
                </xsd:annotation>
            </xsd:element>
            <xsd:element name="default-value"
                          type="javaee:faces-config-default-valueType"
                          minOccurs="0"/>
            <xsd:element name="suggested-value"
                          type="javaee:faces-config-suggested-valueType"
                          minOccurs="0"/>
            <xsd:element name="attribute-extension"
                          type="javaee:faces-config-attribute-extensionType"
                          minOccurs="0"
                          maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name = "id" type = "xsd:ID"/>
    </xsd:complexType>

    <!-- ********************************************** -->

    <xsd:complexType name = "faces-config-attribute-extensionType">
        <xsd:annotation>
            <xsd:documentation>

                Extension element for attribute.  It may contain
                implementation specific content.

            </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
            <xsd:any namespace="##any"
```

```
                    processContents="lax"
                    minOccurs="0"
                    maxOccurs="unbounded" />
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ************************************************* -->

<xsd:complexType name="faces-config-componentType">
     <xsd:annotation>
          <xsd:documentation>

               The "component" element represents a concrete UIComponent
               implementation class that should be registered under the
               specified type identifier, along with its associated
               properties and attributes.  Component types must be unique
               within the entire web application.

               Nested "attribute" elements identify generic attributes that
               are recognized by the implementation logic of this component.
               Nested "property" elements identify JavaBeans properties of
               the component class that may be exposed for manipulation
               via tools.

          </xsd:documentation>
     </xsd:annotation>

     <xsd:sequence>
          <xsd:group ref="javaee:descriptionGroup"/>
          <xsd:element name="component-type"
                       type="javaee:string">
               <xsd:annotation>
                    <xsd:documentation>
```

```
                The "component-type" element represents the name under
                which the corresponding UIComponent class should be
                registered.

            </xsd:documentation>
        </xsd:annotation>
    </xsd:element>
    <xsd:element name="component-class"
                  type="javaee:fully-qualified-classType">
        <xsd:annotation>
            <xsd:documentation>

                The "component-class" element represents the fully
                qualified class name of a concrete UIComponent
                implementation class.

            </xsd:documentation>
        </xsd:annotation>
    </xsd:element>
    <xsd:element name="facet"
                  type="javaee:faces-config-facetType"
                  minOccurs="0"
                  maxOccurs="unbounded"/>
    <xsd:element name="attribute"
                  type="javaee:faces-config-attributeType"
                  minOccurs="0"
                  maxOccurs="unbounded"/>
    <xsd:element name="property"
                  type="javaee:faces-config-propertyType"
                  minOccurs="0"
                  maxOccurs="unbounded"/>
    <xsd:element name="component-extension"
                  type="javaee:faces-config-component-extensionType"
```

```
                        minOccurs="0"
                        maxOccurs="unbounded"/>
     </xsd:sequence>
     <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ************************************************* -->

<xsd:complexType name="faces-config-component-extensionType">
     <xsd:annotation>
          <xsd:documentation>
               Extension element for component.  It may contain
               implementation specific content.
          </xsd:documentation>
     </xsd:annotation>

     <xsd:sequence>
          <xsd:any namespace="##any"
                    processContents="lax"
                    minOccurs="0"
                    maxOccurs="unbounded" />
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ************************************************* -->

<xsd:complexType name="faces-config-default-localeType">
     <xsd:annotation>
          <xsd:documentation>

               The "default-locale" element declares the default locale
               for this application instance.
```

```
                It must be specified as :language:[_:country:[_:variant:]]
                without the colons, for example "ja_JP_SJIS".  The
                separators between the segments may be '-' or '_'.

            </xsd:documentation>
        </xsd:annotation>
        <xsd:simpleContent>
            <xsd:extension base="javaee:faces-config-localeType">
                <xsd:attribute name="id" type="xsd:ID"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>

    <!-- ************************************************** -->


    <xsd:complexType name="faces-config-default-valueType">
        <xsd:annotation>
            <xsd:documentation>

                The "default-value" contains the value for the property or
                attribute in which this element resides.  This value differs
                from the "suggested-value" in that the property or attribute
                must take the value, whereas in "suggested-value" taking the
                value is optional.

            </xsd:documentation>
        </xsd:annotation>
        <xsd:simpleContent>
            <xsd:restriction base="javaee:string"/>
        </xsd:simpleContent>
    </xsd:complexType>

    <!-- ************************************************** -->
```

```
<xsd:simpleType name="faces-config-el-expressionType">
     <xsd:annotation>
          <xsd:documentation>

               EL expressions present within a faces config file
               must start with the character sequence of '#{' and
               end with '}'.

          </xsd:documentation>
     </xsd:annotation>
     <xsd:restriction base="xsd:string">
          <xsd:pattern value="#\{.*\}"/>
     </xsd:restriction>
</xsd:simpleType>

<!-- ********************************************** -->

<xsd:complexType name="faces-config-facetType">
     <xsd:annotation>
          <xsd:documentation>

               Define the name and other design-time information for a facet
               that is associated with a renderer or a component.

          </xsd:documentation>
     </xsd:annotation>

     <xsd:sequence>
          <xsd:group ref="javaee:descriptionGroup"/>
          <xsd:element name="facet-name"
                       type="javaee:java-identifierType">
               <xsd:annotation>
                    <xsd:documentation>
```

```
                        The "facet-name" element represents the facet name
                        under which a UIComponent will be added to its parent.
                        It must be of type "Identifier".

                    </xsd:documentation>
                </xsd:annotation>
            </xsd:element>
            <xsd:element name="facet-extension"
                         type="javaee:faces-config-facet-extensionType"
                         minOccurs="0"
                         maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

    <!-- ********************************************** -->

    <xsd:complexType name="faces-config-facet-extensionType">
        <xsd:annotation>
            <xsd:documentation>

                Extension element for facet.  It may contain implementation
                specific content.

            </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
            <xsd:any namespace="##any"
                     processContents="lax"
                     minOccurs="0"
                     maxOccurs="unbounded" />
        </xsd:sequence>
```

```
      <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ************************************************* -->

<xsd:complexType name="faces-config-from-view-idType">
     <xsd:annotation>
          <xsd:documentation>

               The value of from-view-id must contain one of the following
               values:

                  - The exact match for a view identifier that is recognized
                    by the the ViewHandler implementation being used (such as
                    "/index.jsp" if you are using the default ViewHandler).

                  - A proper prefix of a view identifier, plus a trailing
                    "*" character.  This pattern indicates that all view
                    identifiers that match the portion of the pattern up to
                    the asterisk will match the surrounding rule.  When more
                    than one match exists, the match with the longest pattern
                    is selected.

                  - An "*" character, which means that this pattern applies
                    to all view identifiers.

          </xsd:documentation>
     </xsd:annotation>
     <xsd:simpleContent>
          <xsd:restriction base="javaee:string"/>
     </xsd:simpleContent>
</xsd:complexType>

<!-- ************************************************* -->
```

```
<xsd:complexType name="faces-config-from-actionType">
     <xsd:annotation>
          <xsd:documentation>

               The "from-action" element contains an action reference
               expression that must have been executed (by the default
               ActionListener for handling application level events)
               in order to select the navigation rule.  If not specified,
               this rule will be relevant no matter which action reference
               was executed (or if no action reference was executed).

          </xsd:documentation>
     </xsd:annotation>
     <xsd:simpleContent>
          <xsd:extension base="javaee:faces-config-el-expressionType">
               <xsd:attribute name="id" type="xsd:ID"/>
          </xsd:extension>
     </xsd:simpleContent>
</xsd:complexType>

<!-- ********************************************** -->

<xsd:complexType name="faces-config-ifType">
     <xsd:annotation>
          <xsd:documentation>

               The "if" element defines a condition that must resolve
               to true in order for the navigation case on which it is
               defined to be matched, with the existing match criteria
               (action method and outcome) as a prerequiste, if present.
               The condition is defined declaratively using an value
               expression in the body of this element. The expression is
               evaluated at the time the navigation case is being matched.
```

```
                If the "from-outcome" is omitted and this element is
                present, the navigation handler will match a null outcome
                and use the condition return value to determine if the
                case should be considered a match.

            </xsd:documentation>
        </xsd:annotation>
        <xsd:simpleContent>
            <xsd:extension base="javaee:faces-config-el-expressionType">
                <xsd:attribute name="id" type="xsd:ID"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>

    <!-- ************************************************* -->

    <xsd:complexType name="faces-config-converterType">
        <xsd:annotation>
            <xsd:documentation>

                The "converter" element represents a concrete Converter
                implementation class that should be registered under the
                specified converter identifier.  Converter identifiers must
                be unique within the entire web application.

                Nested "attribute" elements identify generic attributes that
                may be configured on the corresponding UIComponent in order
                to affect the operation of the Converter.  Nested "property"
                elements identify JavaBeans properties of the Converter
                implementation class that may be configured to affect the
                operation of the Converter.  "attribute" and "property"
                elements are intended to allow component developers to
                more completely describe their components to tools and users.
                These elements have no required runtime semantics.
```

```
    </xsd:documentation>
</xsd:annotation>

<xsd:sequence>
    <xsd:group ref="javaee:descriptionGroup"/>
    <xsd:choice>
        <xsd:element name="converter-id"
                     type="javaee:string">
            <xsd:annotation>
                <xsd:documentation>

                    The "converter-id" element represents the
                    identifier under which the corresponding
                    Converter class should be registered.

                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>

        <xsd:element name="converter-for-class"
                     type="javaee:fully-qualified-classType">
            <xsd:annotation>
                <xsd:documentation>

                    The "converter-for-class" element represents the
                    fully qualified class name for which a Converter
                    class will be registered.

                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
    </xsd:choice>
```

```
<xsd:element name="converter-class"
               type="javaee:fully-qualified-classType">
     <xsd:annotation>
          <xsd:documentation>

               The "converter-class" element represents the fully
               qualified class name of a concrete Converter
               implementation class.

          </xsd:documentation>
     </xsd:annotation>
</xsd:element>
<xsd:element name="attribute"
               type="javaee:faces-config-attributeType"
               minOccurs="0"
               maxOccurs="unbounded">
     <xsd:annotation>
          <xsd:documentation>

              Nested "attribute" elements identify generic
              attributes that may be configured on the
              corresponding UIComponent in order to affect the
              operation of the Converter.  This attribute is
              primarily for design-time tools and is not
              specified to have any meaning at runtime.

          </xsd:documentation>
     </xsd:annotation>
</xsd:element>
<xsd:element name="property"
               type="javaee:faces-config-propertyType"
               minOccurs="0"
               maxOccurs="unbounded">
     <xsd:annotation>
```

```
                    <xsd:documentation>

                        Nested "property" elements identify JavaBeans
                        properties of the Converter implementation class
                        that may be configured to affect the operation of
                        the Converter.  This attribute is primarily for
                        design-time tools and is not specified to have
                        any meaning at runtime.

                    </xsd:documentation>
                </xsd:annotation>
            </xsd:element>
            <xsd:element name="converter-extension"
                           type="javaee:faces-config-converter-extensionType"
                           minOccurs="0"
                           maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

    <!-- ********************************************** -->

    <xsd:complexType name = "faces-config-converter-extensionType">
        <xsd:annotation>
            <xsd:documentation>

                Extension element for converter.  It may contain
                implementation specific content.

            </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
            <xsd:any namespace="##any"
```

```
                    processContents="lax"
                    minOccurs="0"
                    maxOccurs="unbounded" />
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>



<!-- ************************************************* -->

<xsd:complexType name="faces-config-lifecycleType">
     <xsd:annotation>
          <xsd:documentation>

               The "lifecycle" element provides a mechanism to specify
               modifications to the behaviour of the default Lifecycle
               implementation for this web application.

          </xsd:documentation>
     </xsd:annotation>

     <xsd:sequence>
          <xsd:element name="phase-listener"
                       type="javaee:fully-qualified-classType"
                       minOccurs="0"
                       maxOccurs="unbounded">

               <xsd:annotation>
                    <xsd:documentation>

                         The "phase-listener" element contains the fully
                         qualified class name of the concrete PhaseListener
                         implementation class that will be registered on
```

```
                        the Lifecycle.

                    </xsd:documentation>
            </xsd:annotation>
            </xsd:element>
            <xsd:element name="lifecycle-extension"
                          type="javaee:faces-config-lifecycle-extensionType"
                          minOccurs="0"
                          maxOccurs="unbounded"/>

        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID"/>

</xsd:complexType>

<!-- ********************************************** -->

<xsd:complexType name = "faces-config-lifecycle-extensionType">
        <xsd:annotation>
            <xsd:documentation>

                Extension element for lifecycle.  It may contain
                implementation specific content.

            </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
            <xsd:any namespace="##any"
                      processContents="lax"
                      minOccurs="0"
                      maxOccurs="unbounded" />
         </xsd:sequence>
         <xsd:attribute name="id" type="xsd:ID"/>
```

```
    </xsd:complexType>


    <!-- ************************************************* -->

    <xsd:simpleType name="faces-config-localeType">
        <xsd:annotation>
            <xsd:documentation>

                The localeType defines valid locale defined by ISO-639-1
                and ISO-3166.

            </xsd:documentation>
        </xsd:annotation>

        <xsd:restriction base="xsd:string">
            <xsd:pattern value="([a-z]{2})[_|\-]?([\p{L}]{2})?[_|\-]?(\w+)?"/>
        </xsd:restriction>
</xsd:simpleType>

    <!-- ************************************************* -->

    <xsd:complexType name="faces-config-locale-configType">
        <xsd:annotation>
            <xsd:documentation>

                The "locale-config" element allows the app developer to
                declare theÂ supported locales for this application.

            </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
            <xsd:element name="default-locale"
```

```
                        type="javaee:faces-config-default-localeType"
                        minOccurs="0">
            </xsd:element>
            <xsd:element name="supported-locale"
                        type="javaee:faces-config-supported-localeType"
                        minOccurs="0"
                        maxOccurs="unbounded">
            </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

    <!-- ********************************************** -->

    <xsd:complexType name="faces-config-default-validatorsType">
        <xsd:annotation>
            <xsd:documentation>

                The "default-validators" element allows the app developer to
                register a set of validators, referenced by identifier, that
                are automatically assigned to any EditableValueHolder component
                in the application, unless overridden or disabled locally.

            </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
            <xsd:element name="validator-id"
                        type="javaee:string"
                        minOccurs="0"
                        maxOccurs="unbounded">
                <xsd:annotation>
                    <xsd:documentation>
```

```
                    The "validator-id" element represents the identifier
                    of a registered validator.

                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ************************************************** -->

<xsd:complexType name="faces-config-managed-beanType">
    <xsd:annotation>
        <xsd:documentation>

            The "managed-bean" element represents a JavaBean, of a
            particular class, that will be dynamically instantiated
            at runtime (by the default VariableResolver implementation)
            if it is referenced as the first element of a value binding
            expression, and no corresponding bean can be identified in
            any scope.  In addition to the creation of the managed bean,
            and the optional storing of it into the specified scope,
            the nested managed-property elements can be used to
            initialize the contents of settable JavaBeans properties of
            the created instance.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
        <xsd:group ref="javaee:descriptionGroup"/>
        <xsd:element name="managed-bean-name"
                     type="javaee:java-identifierType">
```

```
      <xsd:annotation>
           <xsd:documentation>

                The "managed-bean-name" element represents the
                attribute name under which a managed bean will
                be searched for, as well as stored (unless the
                "managed-bean-scope" value is "none").

           </xsd:documentation>
      </xsd:annotation>
</xsd:element>
<xsd:element name="managed-bean-class"
                type="javaee:fully-qualified-classType">
      <xsd:annotation>
           <xsd:documentation>

                The "managed-bean-class" element represents the fully
                qualified class name of the Java class that will be
                used`to instantiate a new instance if creation of the
                specified`managed bean is requested.

                The specified class must conform to standard JavaBeans
                conventions.  In particular, it must have a public
                zero-arguments constructor, and zero or more public
                property setters.

           </xsd:documentation>
      </xsd:annotation>
</xsd:element>
<xsd:element
      name="managed-bean-scope"
      type="javaee:faces-config-managed-bean-scopeOrNoneType">
      <xsd:annotation>
           <xsd:documentation>
```

```
                    The "managed-bean-scope" element represents the scope
                    into which a newly created instance of the specified
                    managed bean will be stored (unless the value is
                    "none").

               </xsd:documentation>
          </xsd:annotation>
     </xsd:element>
     <xsd:choice>
          <xsd:element name="managed-property"
                         type="javaee:faces-config-managed-propertyType"
                         minOccurs="0"
                         maxOccurs="unbounded"/>
          <xsd:element name="map-entries"
                         type="javaee:faces-config-map-entriesType"/>
          <xsd:element name="list-entries"
                         type="javaee:faces-config-list-entriesType"/>
     </xsd:choice>
     <xsd:element name="managed-bean-extension"
                    type="javaee:faces-config-managed-bean-extensionType"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="eager"
                  type="xsd:boolean"
                  use="optional">
     <xsd:annotation>
          <xsd:documentation>

               This attribute is only considered when associated with
               an application-scoped managed bean. If the value of the eager
               attribute is true the runtime must instantiate this class
               and store the instance within the application scope when the
```

```
                    application starts.

                    If eager is unspecified or is false, the default "lazy"
                    instantiation and scoped storage of the managed bean
                    will occur.

                </xsd:documentation>
            </xsd:annotation>
        </xsd:attribute>
        <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

    <!-- ************************************************* -->

    <xsd:complexType name = "faces-config-managed-bean-extensionType">
        <xsd:annotation>
            <xsd:documentation>

                Extension element for managed-bean.  It may contain
                implementation specific content.

            </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
            <xsd:any namespace="##any"
                      processContents="lax"
                      minOccurs="0"
                      maxOccurs="unbounded" />
         </xsd:sequence>
         <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>
```

```
<!-- ************************************************** -->

<xsd:complexType name="faces-config-managed-bean-scopeOrNoneType">
     <xsd:annotation>
          <xsd:documentation>

               <![CDATA[
               Defines the legal values for the <managed-bean-scope>
               element's body content, which includes all of the scopes
               normally used in a web application, plus the "none" value
               indicating that a created bean should not be stored into
               any scope.  Alternatively, an EL expression may be used
               as the value of this element.  The result of evaluating this
               expression must by of type java.util.Map.
               ]]>

          </xsd:documentation>
     </xsd:annotation>
     <xsd:simpleContent>
          <xsd:restriction base="javaee:string">
          <xsd:pattern value="view|request|session|application|none|#\{.*\}"/>
          </xsd:restriction>
     </xsd:simpleContent>
</xsd:complexType>

<!-- ************************************************** -->

<xsd:complexType name="faces-config-managed-propertyType">
     <xsd:annotation>
          <xsd:documentation>

               The "managed-property" element represents an individual
               property of a managed bean that will be configured to the
               specified value (or value set) if the corresponding
```

```
            managed bean is automatically created.

        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:group ref="javaee:descriptionGroup"/>
        <xsd:element name="property-name"
                     type="javaee:string">
            <xsd:annotation>
                <xsd:documentation>

                    The "property-name" element represents the JavaBeans
                    property name under which the corresponding value may
                    be stored.

                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="property-class"
                     type="javaee:java-typeType"
                     minOccurs="0">
            <xsd:annotation>
                <xsd:documentation>

                    The "property-class" element represents the Java type
                    of the value associated with this property name.
                    If not specified, it can be inferred from existing
                    classes; however, this element should be specified
                    if the configuration file is going to be the source
                    for generating the corresponding classes.

                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
```

```
            <xsd:choice>
                <xsd:element name="map-entries"
                             type="javaee:faces-config-map-entriesType"/>
                <xsd:element name="null-value"
                             type="javaee:faces-config-null-valueType">
                </xsd:element>
                <xsd:element name="value"
                             type="javaee:faces-config-valueType"/>
                <xsd:element name="list-entries"
                             type="javaee:faces-config-list-entriesType"/>
            </xsd:choice>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

    <!-- ************************************************* -->

    <xsd:complexType name="faces-config-map-entryType">
        <xsd:annotation>
            <xsd:documentation>

                The "map-entry" element reprsents a single key-entry pair
                that will be added to the computed value of a managed
                property of type java.util.Map.

            </xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:element name="key"
                         type="javaee:string">
                <xsd:annotation>
                    <xsd:documentation>

                        The "key" element is the String representation of a
```

```
                        map key that will be stored in a managed property of
                        type java.util.Map.

                    </xsd:documentation>
               </xsd:annotation>
          </xsd:element>
          <xsd:choice>
               <xsd:element name="null-value"
                              type="javaee:faces-config-null-valueType"/>
               <xsd:element name="value"
                              type="javaee:faces-config-valueType"/>
          </xsd:choice>
     </xsd:sequence>
     <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ************************************************* -->

<xsd:complexType name="faces-config-map-entriesType">
     <xsd:annotation>
          <xsd:documentation>

               The "map-entries' element represents a set of key-entry pairs
               that will be added to the computed value of a managed property
               of type java.util.Map.  In addition, the Java class types
               of the key and entry values may be optionally declared.

          </xsd:documentation>
     </xsd:annotation>

     <xsd:sequence>
          <xsd:element name="key-class"
                         type="javaee:fully-qualified-classType"
                         minOccurs="0">
```

```
            <xsd:annotation>
                <xsd:documentation>

                    The "key-class" element defines the Java type to which
                    each "key" element in a set of "map-entry" elements
                    will be converted to.  If omitted, "java.lang.String"
                    is assumed.

                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="value-class"
                     type="javaee:faces-config-value-classType"
                     minOccurs="0"/>
        <xsd:element name="map-entry"
                     type="javaee:faces-config-map-entryType"
                     minOccurs="0"
                     maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ********************************************** -->

<xsd:complexType name="faces-config-navigation-caseType">
    <xsd:annotation>
        <xsd:documentation>

            The "navigation-case" element describes a particular
            combination of conditions that must match for this case to
            be executed, and the view id of the component tree that
            should be selected next.

        </xsd:documentation>
```

```
</xsd:annotation>
<xsd:sequence>
     <xsd:group ref="javaee:descriptionGroup"/>
     <xsd:element name="from-action"
                    type="javaee:faces-config-from-actionType"
                    minOccurs="0">
     </xsd:element>
     <xsd:element name="from-outcome"
                    type="javaee:string" minOccurs="0">
          <xsd:annotation>
               <xsd:documentation>

                    The "from-outcome" element contains a logical outcome
                    string returned by the execution of an application
                    action method selected via an "actionRef" property
                    (or a literal value specified by an "action" property)
                    of a UICommand component.  If specified, this rule
                    will be relevant only if the outcome value matches
                    this element's value.  If not specified, this rule
                    will be relevant if the outcome value is non-null
                    or, if the "if" element is present, will be relevant
                    for any outcome value, with the assumption that the
                    condition specified in the "if" element ultimately
                    determines if this rule is a match.

               </xsd:documentation>
          </xsd:annotation>
     </xsd:element>
     <xsd:element name="if"
                    type="javaee:faces-config-ifType"
                    minOccurs="0">
          <xsd:annotation>
               <xsd:documentation>
```

```
                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="to-view-id"
                      type="javaee:faces-config-valueType">
            <xsd:annotation>
                <xsd:documentation>

                    The "to-view-id" element contains the view identifier
                    of the next view that should be displayed if this
                    navigation rule is matched. If the contents is a
                    value expression, it should be resolved by the
                    navigation handler to obtain the view identifier.

                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element
            name="redirect"
            type="javaee:faces-config-redirectType" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ********************************************** -->

<xsd:complexType name="faces-config-navigation-ruleType">
    <xsd:annotation>
        <xsd:documentation>

            The "navigation-rule" element represents an individual
            decision rule that will be utilized by the default
```

```
            NavigationHandler implementation to make decisions on
            what view should be displayed next, based on the
            view id being processed.

        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:group ref="javaee:descriptionGroup"/>
        <xsd:element name="from-view-id"
                     type="javaee:faces-config-from-view-idType"
                     minOccurs="0"/>
        <xsd:element name="navigation-case"
                     type="javaee:faces-config-navigation-caseType"
                     minOccurs="0"
                     maxOccurs="unbounded"/>
        <xsd:element
            name="navigation-rule-extension"
            type="javaee:faces-config-navigation-rule-extensionType"
            minOccurs="0"
            maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ********************************************** -->

<xsd:complexType name = "faces-config-navigation-rule-extensionType">
    <xsd:annotation>
        <xsd:documentation>

            Extension element for navigation-rule.  It may contain
            implementation specific content.

        </xsd:documentation>
```

```
    </xsd:annotation>

    <xsd:sequence>
        <xsd:any namespace="##any"
                 processContents="lax"
                 minOccurs="0"
                 maxOccurs="unbounded" />
     </xsd:sequence>
     <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>


<!-- ************************************************* -->

<xsd:complexType name="faces-config-null-valueType">
    <xsd:annotation>
        <xsd:documentation>

            The "null-value" element indicates that the managed
            property in which we are nested will be explicitly
            set to null if our managed bean is automatically
            created.  This is different from omitting the managed
            property element entirely, which will cause no
            property setter to be called for this property.

            The "null-value" element can only be used when the
            associated "property-class" identifies a Java class,
            not a Java primitive.

        </xsd:documentation>
    </xsd:annotation>
    <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
```

```
<!-- ************************************************** -->

<xsd:complexType name="faces-config-propertyType">
     <xsd:annotation>
          <xsd:documentation>

               The "property" element represents a JavaBean property of the
               Java class represented by our parent element.

               Property names must be unique within the scope of the Java
               class that is represented by the parent element, and must
               correspond to property names that will be recognized when
               performing introspection against that class via
               java.beans.Introspector.

          </xsd:documentation>
     </xsd:annotation>
     <xsd:sequence>
          <xsd:group ref="javaee:descriptionGroup"/>
          <xsd:element name="property-name"
                       type="javaee:string">
               <xsd:annotation>
                    <xsd:documentation>

                         The "property-name" element represents the JavaBeans
                         property name under which the corresponding value
                         may be stored.

                    </xsd:documentation>
               </xsd:annotation>
          </xsd:element>
          <xsd:element name="property-class"
                       type="javaee:java-typeType">
               <xsd:annotation>
```

```
                    <xsd:documentation>

                         The "property-class" element represents the Java type
                         of the value associated with this property name.
                         If not specified, it can be inferred from existing
                         classes; however, this element should be specified if
                         the configuration file is going to be the source for
                         generating the corresponding classes.

                    </xsd:documentation>
               </xsd:annotation>
          </xsd:element>
          <xsd:element name="default-value"
                         type="javaee:faces-config-default-valueType"
                         minOccurs="0"/>
          <xsd:element name="suggested-value"
                         type="javaee:faces-config-suggested-valueType"
                         minOccurs="0"/>
          <xsd:element name="property-extension"
                         type="javaee:faces-config-property-extensionType"
                         minOccurs="0"
                         maxOccurs="unbounded"/>
     </xsd:sequence>
     <xsd:attribute name="id" type="xsd:ID"/>

</xsd:complexType>

<!-- ********************************************** -->

<xsd:complexType name="faces-config-property-extensionType">
     <xsd:annotation>
          <xsd:documentation>

               Extension element for property.  It may contain
```

```
                implementation specific content.

            </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
            <xsd:any namespace="##any"
                      processContents="lax"
                      minOccurs="0"
                      maxOccurs="unbounded" />
         </xsd:sequence>
         <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ************************************************* -->

<xsd:complexType name="faces-config-redirectType">
     <xsd:annotation>
          <xsd:documentation>

                    The "redirect" element indicates that navigation to the
                    specified "to-view-id" should be accomplished by
                    performing an HTTP redirect rather than the usual
                    ViewHandler mechanisms.

          </xsd:documentation>
     </xsd:annotation>
     <xsd:sequence>
          <xsd:element name="view-param"
                         type="javaee:faces-config-redirect-viewParamType"
                         minOccurs="0" maxOccurs="unbounded"/>
     </xsd:sequence>
     <xsd:attribute name="id" type="xsd:ID"/>
     <xsd:attribute name="include-view-params" type="xsd:boolean" use="optional"/>
```

```
</xsd:complexType>

<!-- ************************************************** -->

<xsd:complexType name="faces-config-redirect-viewParamType">
     <xsd:annotation>
          <xsd:documentation>

                    The "view-param" element, only valid within
                    a "redirect" element, contains child "name"
                    and "value" elements that must be included in the
                    redirect url when the redirect is performed.

          </xsd:documentation>
     </xsd:annotation>
     <xsd:sequence>
          <xsd:element name="name"
                         type="javaee:string"
                         minOccurs="1" maxOccurs="1"/>
          <xsd:element name="value"
                         type="javaee:string"
                         minOccurs="1" maxOccurs="1"/>
     </xsd:sequence>
     <xsd:attribute name="id" type="xsd:ID"/>

</xsd:complexType>

<!-- ************************************************** -->

<xsd:complexType name="faces-config-referenced-beanType">
     <xsd:annotation>
          <xsd:documentation>
```

```
            The "referenced-bean" element represents at design time the
            promise that a Java object of the specified type will exist at
            runtime in some scope, under the specified key.  This can be
            used by design time tools to construct user interface dialogs
            based on the properties of the specified class.  The presence
            or absence of a referenced bean element has no impact on the
            JavaServer Faces runtime environment inside a web application.

        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:group ref="javaee:descriptionGroup"/>
        <xsd:element name="referenced-bean-name"
                     type="javaee:java-identifierType">
            <xsd:annotation>
                <xsd:documentation>

                    The "referenced-bean-name" element represents the
                    attribute name under which the corresponding
                    referenced bean may be assumed to be stored, in one
                    of 'request', 'session', or 'application' scopes.

                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="referenced-bean-class"
                     type="javaee:fully-qualified-classType">
            <xsd:annotation>
                <xsd:documentation>

                    The "referenced-bean-class" element represents the
                    fully qualified class name of the Java class
                    (either abstract or concrete) or Java interface
                    implemented by the corresponding referenced bean.
```

```
                    </xsd:documentation>
                </xsd:annotation>
            </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ********************************************** -->

<xsd:complexType name="faces-config-render-kitType">
        <xsd:annotation>
            <xsd:documentation>

                The "render-kit" element represents a concrete RenderKit
                implementation that should be registered under the specified
                render-kit-id.  If no render-kit-id is specified, the
                identifier of the default RenderKit
                (RenderKitFactory.DEFAULT_RENDER_KIT) is assumed.

            </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
            <xsd:group ref="javaee:descriptionGroup"/>
            <xsd:element name="render-kit-id"
                         type="javaee:string"
                         minOccurs="0">
                <xsd:annotation>
                    <xsd:documentation>

                        The "render-kit-id" element represents an identifier
                        for the RenderKit represented by the parent
                        "render-kit" element.
```

```
                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="render-kit-class"
                      type="javaee:fully-qualified-classType"
                      minOccurs="0">
            <xsd:annotation>
                <xsd:documentation>

                    The "render-kit-class" element represents the fully
                    qualified class name of a concrete RenderKit
                    implementation class.

                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="renderer"
                      type="javaee:faces-config-rendererType"
                      minOccurs="0"
                      maxOccurs="unbounded"/>
        <xsd:element name="client-behavior-renderer"
                      type="javaee:faces-config-client-behavior-rendererType"
                      minOccurs="0"
                      maxOccurs="unbounded"/>
        <xsd:element name="render-kit-extension"
                      type="javaee:faces-config-render-kit-extensionType"
                      minOccurs="0"
                      maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ********************************************** -->
```

```
<xsd:complexType name="faces-config-client-behavior-rendererType">
     <xsd:annotation>
          <xsd:documentation>

               The "client-behavior-renderer" element represents a concrete
               ClientBehaviorRenderer implementation class that should be
               registered under the specified behavior renderer type identifier,
               in the RenderKit associated with the parent "render-kit"
               element.  Client Behavior renderer type must be unique within the RenderKit
               associated with the parent "render-kit" element.

               Nested "attribute" elements identify generic component
               attributes that are recognized by this renderer.

          </xsd:documentation>
     </xsd:annotation>

     <xsd:sequence>
          <xsd:element name="client-behavior-renderer-type"
                       type="javaee:string">
               <xsd:annotation>
                    <xsd:documentation>

                      The "client-behavior-renderer-type" element represents a renderer type
                       identifier for the Client Behavior Renderer represented by the parent
                          "client-behavior-renderer" element.

                    </xsd:documentation>
               </xsd:annotation>
          </xsd:element>
          <xsd:element name="client-behavior-renderer-class"
                       type="javaee:fully-qualified-classType">
               <xsd:annotation>
```

```
                    <xsd:documentation>

                         The "client-behavior-renderer-class" element represents the fully
                         qualified class name of a concrete Client Behavior Renderer
                         implementation class.

                    </xsd:documentation>
               </xsd:annotation>
          </xsd:element>
     </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="faces-config-rendererType">
     <xsd:annotation>
          <xsd:documentation>

               The "renderer" element represents a concrete Renderer
               implementation class that should be registered under the
               specified component family and renderer type identifiers,
               in the RenderKit associated with the parent "render-kit"
               element.  Combinations of component family and
               renderer type must be unique within the RenderKit
               associated with the parent "render-kit" element.

               Nested "attribute" elements identify generic component
               attributes that are recognized by this renderer.

          </xsd:documentation>
     </xsd:annotation>

     <xsd:sequence>
          <xsd:group ref="javaee:descriptionGroup"/>
          <xsd:element name="component-family"
                         type="javaee:string">
```

```
     <xsd:annotation>
          <xsd:documentation>

               The "component-family" element represents the
               component family for which the Renderer represented
               by the parent "renderer" element will be used.

          </xsd:documentation>
     </xsd:annotation>
</xsd:element>
<xsd:element name="renderer-type"
               type="javaee:string">
     <xsd:annotation>
          <xsd:documentation>

               The "renderer-type" element represents a renderer type
               identifier for the Renderer represented by the parent
               "renderer" element.

          </xsd:documentation>
     </xsd:annotation>
</xsd:element>
<xsd:element name="renderer-class"
               type="javaee:fully-qualified-classType">
     <xsd:annotation>
          <xsd:documentation>

               The "renderer-class" element represents the fully
               qualified class name of a concrete Renderer
               implementation class.

          </xsd:documentation>
     </xsd:annotation>
</xsd:element>
```

```
            <xsd:element name="facet"
                         type="javaee:faces-config-facetType"
                         minOccurs="0"
                         maxOccurs="unbounded"/>
            <xsd:element name="attribute"
                         type="javaee:faces-config-attributeType"
                         minOccurs="0"
                         maxOccurs="unbounded"/>
            <xsd:element name="renderer-extension"
                         type="javaee:faces-config-renderer-extensionType"
                         minOccurs="0"
                         maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

    <!-- ********************************************** -->

    <xsd:complexType name="faces-config-renderer-extensionType">
        <xsd:annotation>
            <xsd:documentation>

                Extension element for renderer.  It may contain implementation
                specific content.

            </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
            <xsd:any namespace="##any"
                     processContents="lax"
                     minOccurs="0"
                     maxOccurs="unbounded" />
         </xsd:sequence>
```

```
      <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ************************************************** -->

<xsd:complexType name="faces-config-render-kit-extensionType">
     <xsd:annotation>
          <xsd:documentation>

               Extension element for render-kit.  It may contain
               implementation specific content.

          </xsd:documentation>
     </xsd:annotation>

     <xsd:sequence>
          <xsd:any namespace="##any"
                     processContents="lax"
                     minOccurs="0"
                     maxOccurs="unbounded" />
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>


<!-- ************************************************** -->

<xsd:complexType name="faces-config-suggested-valueType">
     <xsd:annotation>
          <xsd:documentation>

               The "suggested-value" contains the value for the property or
               attribute in which this element resides.  This value is
               advisory only and is intended for tools to use when
```

```
            populating pallettes.

        </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
        <xsd:restriction base="javaee:string"/>
    </xsd:simpleContent>
</xsd:complexType>

<!-- ************************************************** -->

<xsd:complexType name="faces-config-supported-localeType">
    <xsd:annotation>
        <xsd:documentation>

            The "supported-locale" element allows authors to declare
            which locales are supported in this application instance.

            It must be specified as :language:[_:country:[_:variant:]]
            without the colons, for example "ja_JP_SJIS".  The
            separators between the segments may be '-' or '_'.

        </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
        <xsd:extension base="javaee:faces-config-localeType">
            <xsd:attribute name="id" type="xsd:ID"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>

<!-- ************************************************** -->

<xsd:complexType name="faces-config-behaviorType">
```

```
<xsd:annotation>
     <xsd:documentation>

          The "behavior" element represents a concrete Behavior
          implementation class that should be registered under the
          specified behavior identifier.  Behavior identifiers must
          be unique within the entire web application.

          Nested "attribute" elements identify generic attributes that
          may be configured on the corresponding UIComponent in order
          to affect the operation of the Behavior.  Nested "property"
          elements identify JavaBeans properties of the Behavior
          implementation class that may be configured to affect the
          operation of the Behavior.  "attribute" and "property"
          elements are intended to allow component developers to
          more completely describe their components to tools and users.
          These elements have no required runtime semantics.

     </xsd:documentation>
</xsd:annotation>
<xsd:sequence>
     <xsd:group ref="javaee:descriptionGroup"/>
     <xsd:element name="behavior-id"
                   type="javaee:string">
          <xsd:annotation>
               <xsd:documentation>

                    The "behavior-id" element represents the identifier
                    under which the corresponding Behavior class should
                    be registered.

               </xsd:documentation>
          </xsd:annotation>
     </xsd:element>
```

```
<xsd:element name="behavior-class"
               type="javaee:fully-qualified-classType">
     <xsd:annotation>
          <xsd:documentation>

               The "behavior-class" element represents the fully
               qualified class name of a concrete Behavior
               implementation class.

          </xsd:documentation>
     </xsd:annotation>
</xsd:element>
<xsd:element name="attribute"
               type="javaee:faces-config-attributeType"
               minOccurs="0"
               maxOccurs="unbounded">
     <xsd:annotation>
          <xsd:documentation>

              Nested "attribute" elements identify generic
              attributes that may be configured on the
              corresponding UIComponent in order to affect the
              operation of the Behavior.  This attribute is
              primarily for design-time tools and is not
              specified to have any meaning at runtime.

          </xsd:documentation>
     </xsd:annotation>

</xsd:element>
<xsd:element name="property"
               type="javaee:faces-config-propertyType"
               minOccurs="0"
               maxOccurs="unbounded">
```

```
            <xsd:annotation>
                <xsd:documentation>

                    Nested "property" elements identify JavaBeans
                    properties of the Behavior implementation class
                    that may be configured to affect the operation of
                    the Behavior.  This attribute is primarily for
                    design-time tools and is not specified to have
                    any meaning at runtime.

                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="behavior-extension"
                     type="javaee:faces-config-behavior-extensionType"
                     minOccurs="0"
                     maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<!-- ********************************************** -->

<xsd:complexType name = "faces-config-behavior-extensionType">
    <xsd:annotation>
        <xsd:documentation>

            Extension element for behavior.  It may contain
            implementation specific content.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
        <xsd:any namespace="##any"
```

```
                    processContents="lax"
                    minOccurs="0"
                    maxOccurs="unbounded" />
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ************************************************** -->

<xsd:complexType name="faces-config-validatorType">
     <xsd:annotation>
          <xsd:documentation>

               The "validator" element represents a concrete Validator
               implementation class that should be registered under the
               specified validator identifier.  Validator identifiers must
               be unique within the entire web application.

               Nested "attribute" elements identify generic attributes that
               may be configured on the corresponding UIComponent in order
               to affect the operation of the Validator.  Nested "property"
               elements identify JavaBeans properties of the Validator
               implementation class that may be configured to affect the
               operation of the Validator.  "attribute" and "property"
               elements are intended to allow component developers to
               more completely describe their components to tools and users.
               These elements have no required runtime semantics.

          </xsd:documentation>
     </xsd:annotation>
     <xsd:sequence>
          <xsd:group ref="javaee:descriptionGroup"/>
          <xsd:element name="validator-id"
                       type="javaee:string">
```

```
     <xsd:annotation>
          <xsd:documentation>

               The "validator-id" element represents the identifier
               under which the corresponding Validator class should
               be registered.

          </xsd:documentation>
     </xsd:annotation>
</xsd:element>
<xsd:element name="validator-class"
               type="javaee:fully-qualified-classType">
     <xsd:annotation>
          <xsd:documentation>

               The "validator-class" element represents the fully
               qualified class name of a concrete Validator
               implementation class.

          </xsd:documentation>
     </xsd:annotation>
</xsd:element>
<xsd:element name="attribute"
               type="javaee:faces-config-attributeType"
               minOccurs="0"
               maxOccurs="unbounded">
     <xsd:annotation>
          <xsd:documentation>

              Nested "attribute" elements identify generic
              attributes that may be configured on the
              corresponding UIComponent in order to affect the
              operation of the Validator.  This attribute is
              primarily for design-time tools and is not
```

```
                    specified to have any meaning at runtime.

                </xsd:documentation>
            </xsd:annotation>

        </xsd:element>
        <xsd:element name="property"
                        type="javaee:faces-config-propertyType"
                        minOccurs="0"
                        maxOccurs="unbounded">
            <xsd:annotation>
                <xsd:documentation>

                    Nested "property" elements identify JavaBeans
                    properties of the Validator implementation class
                    that may be configured to affect the operation of
                    the Validator.  This attribute is primarily for
                    design-time tools and is not specified to have
                    any meaning at runtime.

                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="validator-extension"
                        type="javaee:faces-config-validator-extensionType"
                        minOccurs="0"
                        maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ********************************************** -->

<xsd:complexType name = "faces-config-validator-extensionType">
```

```
      <xsd:annotation>
           <xsd:documentation>

                Extension element for validator.  It may contain
                implementation specific content.

           </xsd:documentation>
      </xsd:annotation>

      <xsd:sequence>
           <xsd:any namespace="##any"
                      processContents="lax"
                      minOccurs="0"
                      maxOccurs="unbounded" />
       </xsd:sequence>
       <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ************************************************** -->

<xsd:simpleType name="faces-config-valueType">
      <xsd:annotation>
           <xsd:documentation>

                The "value" element is the String representation of
                a literal value to which a scalar managed property
                will be set, or a value binding expression ("#{...}")
                that will be used to calculate the required value.
                It will be converted as specified for the actual
                property type.

           </xsd:documentation>
      </xsd:annotation>
      <xsd:union
```

```
          memberTypes="javaee:faces-config-el-expressionType xsd:string"/>
</xsd:simpleType>

<!-- ********************************************** -->

<xsd:complexType name="faces-config-value-classType">
     <xsd:annotation>
          <xsd:documentation>

               The "value-class" element defines the Java type to which each
               "value" element's value will be converted to, prior to adding
               it to the "list-entries" list for a managed property that is
               a java.util.List, or a "map-entries" map for a managed
               property that is a java.util.Map.

          </xsd:documentation>
     </xsd:annotation>
     <xsd:simpleContent>
          <xsd:restriction base="javaee:fully-qualified-classType"/>
     </xsd:simpleContent>
</xsd:complexType>

<!-- ********************************************** -->

<xsd:complexType name="faces-config-list-entriesType">
     <xsd:annotation>
          <xsd:documentation>

               The "list-entries" element represents a set of initialization
               elements for a managed property that is a java.util.List or an
               array.  In the former case, the "value-class" element can
               optionally be used to declare the Java type to which each
               value should be converted before adding it to the Collection.
```

```
            </xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:element name="value-class"
                         type="javaee:faces-config-value-classType"
                         minOccurs="0"/>
            <xsd:choice minOccurs="0" maxOccurs="unbounded">
                <xsd:element name="null-value"
                             type="javaee:faces-config-null-valueType"/>
                <xsd:element name="value"
                             type="javaee:faces-config-valueType"/>
            </xsd:choice>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

    <!-- ********************************************************** -->

    <xsd:complexType name="faces-config-system-event-listenerType">
        <xsd:annotation>
            <xsd:documentation>

                The presence of this element within the "application" element in
                an application configuration resource file indicates the
                developer wants to add an SystemEventListener to this
                application instance.  Elements nested within this element allow
                selecting the kinds of events that will be delivered to the
                listener instance, and allow selecting the kinds of classes that
                can be the source of events that are delivered to the listener
                instance.

            </xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
```

```
<xsd:element name="system-event-listener-class"
               type="javaee:fully-qualified-classType">
     <xsd:annotation>
          <xsd:documentation>

               The "system-event-listener-class" element contains
               the fully qualified class name of the concrete
               SystemEventListener implementation class that will be
               called when events of the type specified by the
               "system-event-class" are sent by the runtime.

          </xsd:documentation>
     </xsd:annotation>
</xsd:element>
<xsd:element name="system-event-class"
               type="javaee:fully-qualified-classType">
     <xsd:annotation>
          <xsd:documentation>

               The "system-event-class" element contains the fully
               qualified class name of the SystemEvent subclass for
               which events will be delivered to the class whose fully
               qualified class name is given by the
               "system-event-listener-class" element.

          </xsd:documentation>
     </xsd:annotation>
</xsd:element>
<xsd:element name="source-class" minOccurs="0"
               type="javaee:fully-qualified-classType">
     <xsd:annotation>
          <xsd:documentation>

               The "source-class" element, if present, contains the
```

```
                        fully qualified class name of the class that will be the
                        source for the event to be delivered to the class whose
                        fully qualified class name is given by the
                        "system-event-listener-class" element.

                    </xsd:documentation>
                </xsd:annotation>
            </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

    <!-- ************************************************* -->

    <xsd:simpleType name="faces-config-versionType">
        <xsd:annotation>
            <xsd:documentation>

                This type contains the recognized versions of
                faces-config supported.

            </xsd:documentation>
        </xsd:annotation>
        <xsd:restriction base="xsd:token">
            <xsd:enumeration value="2.0"/>
        </xsd:restriction>
    </xsd:simpleType>

    <!-- ************************************************* -->

</xsd:schema>
```

1.2 XML Schema Definition for Ajax Response

```
<xsd:schema
      targetNamespace="http://java.sun.com/xml/ns/javaee"
      xmlns:javaee="http://java.sun.com/xml/ns/javaee"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:xml="http://www.w3.org/XML/1998/namespace"
      elementFormDefault="qualified"
      attributeFormDefault="unqualified"
      version="2.0">

    <xsd:annotation>
        <xsd:documentation>
            $Id: web-partialresponse_2_0.xsd,v 1.0 2008/12/04 21:12:50 rogerk Exp $
        </xsd:documentation>
    </xsd:annotation>

    <xsd:annotation>
        <xsd:documentation>

            Copyright 2007 Sun Microsystems, Inc.,
            901 San Antonio Road,
            Palo Alto, California 94303, U.S.A.
            All rights reserved.

            Sun Microsystems, Inc. has intellectual property
            rights relating to technology described in this document. In
            particular, and without limitation, these intellectual
            property rights may include one or more of the U.S. patents
            listed at http://www.sun.com/patents and one or more
            additional patents or pending patent applications in the
            U.S. and other countries.

            This document and the technology which it describes are
```

```
            distributed under licenses restricting their use, copying,
            distribution, and decompilation. No part of this document
            may be reproduced in any form by any means without prior
            written authorization of Sun and its licensors, if any.

            Third-party software, including font technology, is
            copyrighted and licensed from Sun suppliers.

            Sun, Sun Microsystems, the Sun logo, Solaris, Java, Java EE,
            JavaServer Pages, Enterprise JavaBeans and the Java Coffee
            Cup logo are trademarks or registered trademarks of Sun
            Microsystems, Inc. in the U.S. and other countries.

            Federal Acquisitions: Commercial Software - Government Users
            Subject to Standard License Terms and Conditions.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:annotation>
        <xsd:documentation>

            <![CDATA[

            The XML Schema for the JavaServer Faces (Version 2.0)
            Partial Response used in JSF Ajax frameworks.

            ]]>

        </xsd:documentation>
    </xsd:annotation>

    <xsd:include schemaLocation="javaee_5.xsd"/>
```

```
<!-- ************************************************* -->

<xsd:element name = "partial-response" type="javaee:partial-responseType">
     <xsd:annotation>
          <xsd:documentation>

               The "partial-response" element is the root of the partial
               response information hierarchy, and contains nested elements for all
               possible elements that can exist in the response.

          </xsd:documentation>
     </xsd:annotation>
</xsd:element>

<!-- ************************************************* -->

<xsd:complexType name="partial-responseType">
     <xsd:annotation>
          <xsd:documentation>

               The "partial-response" element is the root of thei partial
               response information hierarchy, and contains nested elements for all
               possible elements that can exist in the response.

          </xsd:documentation>
     </xsd:annotation>
     <xsd:choice>
          <xsd:element name="changes"
                         type="javaee:partial-response-changesType"
                         minOccurs="0"
                         maxOccurs="1"/>
          <xsd:element name="redirect"
                         type="javaee:partial-response-redirectType"
                         minOccurs="0"
```

```
                              maxOccurs="1"/>
               <xsd:element name="error"
                              type="javaee:partial-response-errorType"
                              minOccurs="0"
                              maxOccurs="1"/>
          </xsd:choice>
     </xsd:complexType>

     <xsd:complexType name="partial-response-changesType">
          <xsd:choice minOccurs="0" maxOccurs="unbounded">
               <xsd:element name="update"
                              type="javaee:partial-response-updateType"/>
               <xsd:element name="insert"
                              type="javaee:partial-response-insertType"/>
               <xsd:element name="delete"
                              type="javaee:partial-response-deleteType"/>
               <xsd:element name="attributes"
                              type="javaee:partial-response-attributesType"/>
               <xsd:element name="eval" type="xsd:string">
                    <xsd:annotation>
                         <xsd:documentation>

                              The "eval" element enables this element's contents to be executed
as JavaScript.

                         </xsd:documentation>
                    </xsd:annotation>
               </xsd:element>
               <xsd:element name="extension"
                              type="javaee:partial-response-extensionType"/>
          </xsd:choice>
     </xsd:complexType>



     <xsd:complexType name="partial-response-updateType">
```

```
        <xsd:annotation>
            <xsd:documentation>

                The "update" element enables DOM elements matching the "id"
                attribute to be updated with the contents of this element.

            </xsd:documentation>
        </xsd:annotation>
        <xsd:simpleContent>
            <xsd:extension base="xsd:string">
                <xsd:attribute name="id" type="xsd:string" use="required"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>

    <xsd:complexType name="partial-response-insertType">
        <xsd:annotation>
            <xsd:documentation>

                The "insert" element enables content to be inserted into the DOM
                before or after an existing DOM element as specified by the
                nested "before" or "after" elements.  The elements "before" and
                "after" are mutually exclusive - one of them must be specified.

            </xsd:documentation>
        </xsd:annotation>
        <xsd:choice minOccurs="1" maxOccurs="1">
            <xsd:element name="before">
                <xsd:complexType>
                    <xsd:attribute name="id" type="xsd:string" use="required"/>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="after">
                <xsd:complexType>
```

```
                    <xsd:attribute name="id" type="xsd:string" use="required"/>
               </xsd:complexType>
          </xsd:element>
     </xsd:choice>
</xsd:complexType>

<xsd:complexType name="partial-response-deleteType">
     <xsd:annotation>
          <xsd:documentation>

               The "delete" element enables DOM elements matching the "id"
               attribute to be removed.

          </xsd:documentation>
     </xsd:annotation>
     <xsd:attribute name="id" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="partial-response-attributesType">
     <xsd:annotation>
          <xsd:documentation>

             The "attributes" element enables attributes of DOM elements matching the "id"
               attribute to be updated.  If this element is used, then it must contain at
               least one "attribute" element.

          </xsd:documentation>
     </xsd:annotation>
     <xsd:sequence>
          <xsd:element name="attribute" minOccurs="1" maxOccurs="unbounded">
               <xsd:complexType>
                    <xsd:attribute name="name" type="xsd:string" use="required"/>
                    <xsd:attribute name="value" type="xsd:string" use="required"/>
               </xsd:complexType>
```

```
          </xsd:element>
     </xsd:sequence>
     <xsd:attribute name="id" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="partial-response-redirectType">
     <xsd:annotation>
          <xsd:documentation>

             The "redirect" element enables a redirect to the location as specified by the
               "url" attribute.

          </xsd:documentation>
     </xsd:annotation>
     <xsd:attribute name="url" type="xsd:anyURI" use="required"/>
</xsd:complexType>

<xsd:complexType name="partial-response-errorType">
     <xsd:annotation>
          <xsd:documentation>

               The "error" element contains error information from the server.

          </xsd:documentation>
     </xsd:annotation>
     <xsd:sequence>
          <xsd:element name="error-name" type="xsd:string" minOccurs="1" maxOccurs="1"/>
          <xsd:element name="error-message" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="partial-response-extensionType">
     <xsd:annotation>
          <xsd:documentation>
```

```
                Extension element for partial response.  It may contain
                implementation specific content.
            </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
            <xsd:any namespace="##any"
                     processContents="lax"
                     minOccurs="0"
                     maxOccurs="unbounded" />
         </xsd:sequence>
         <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

</xsd:schema>
```

1.3 XML Schema Definition For Facelet Taglib

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsd:schema
    targetNamespace="http://java.sun.com/xml/ns/javaee"
    xmlns:javaee="http://java.sun.com/xml/ns/javaee"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xml="http://www.w3.org/XML/1998/namespace"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified"
    version="2.0">
   <xsd:include schemaLocation="javaee_5.xsd"/>
   <xsd:element name="facelet-taglib" type="javaee:facelet-taglibType">
      <xsd:unique name="facelet-taglib-tagname-uniqueness">
         <xsd:annotation>
            <xsd:documentation>
```

```
                tag-names must be unique within a document.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:selector xpath="javaee:tag"/>
        <xsd:field    xpath="javaee:tag-name"/>
    </xsd:unique>
    <xsd:unique name="faces-config-behavior-ID-uniqueness">
        <xsd:annotation>
            <xsd:documentation>

                Behavior IDs must be unique within a document.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:selector xpath="javaee:behavior"/>
        <xsd:field    xpath="javaee:behavior-id"/>
    </xsd:unique>
    <xsd:unique name="faces-config-converter-ID-uniqueness">
        <xsd:annotation>
            <xsd:documentation>

                Converter IDs must be unique within a document.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:selector xpath="javaee:converter"/>
        <xsd:field    xpath="javaee:converter-id"/>
    </xsd:unique>
<xsd:unique name="faces-config-validator-ID-uniqueness">
        <xsd:annotation>
            <xsd:documentation>

                Validator IDs must be unique within a document.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:selector xpath="javaee:validator"/>
```

```
        <xsd:field    xpath="javaee:validator-id"/>
    </xsd:unique>
</xsd:element>
<xsd:complexType name="facelet-taglibType">
    <xsd:annotation>
      <xsd:documentation>
        The top level XML element in a facelet tag library XML file.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:group ref="javaee:descriptionGroup"/>
        <xsd:choice>
            <xsd:element name="library-class"
                         type="javaee:fully-qualified-classType"/>
            <xsd:sequence>
                <xsd:element name="namespace" type="javaee:string"/>
                <xsd:element minOccurs="0" maxOccurs="1"
                             name="composite-library-name"
                             type="javaee:fully-qualified-classType"/>
                <xsd:choice minOccurs="0" maxOccurs="unbounded">
                    <xsd:element name="tag"
                                 type="javaee:facelet-taglib-tagType"/>
                    <xsd:element name="function"
                                 type="javaee:facelet-taglib-functionType"/>

                </xsd:choice>
            </xsd:sequence>
        </xsd:choice>
        <xsd:element name="taglib-extension"
                     type="javaee:facelet-taglib-extensionType"
                     minOccurs="0"
                     maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
```

```
        <xsd:attribute name="version"
                      type="javaee:facelet-taglib-versionType"
                      use="required"/>
    </xsd:complexType>
<xsd:complexType name="facelet-taglib-extensionType">
        <xsd:annotation>
            <xsd:documentation>
                Extension element for facelet-taglib. It may contain
                implementation specific content.
            </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
            <xsd:any namespace="##any"
                     processContents="lax"
                     minOccurs="0"
                     maxOccurs="unbounded" />
         </xsd:sequence>
         <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

    <xsd:complexType name="facelet-taglib-tagType">
        <xsd:annotation>
          <xsd:documentation>
            If the tag library XML file contains individual tag
            declarations rather than pointing to a library-class or a
            declaring a composite-library name, the individual tags are
            enclosed in tag elements.
          </xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:group ref="javaee:descriptionGroup"/>
            <xsd:element name="tag-name" type="javaee:facelet-taglib-canonical-nameType"/>
            <xsd:choice>
```

```
            <xsd:element name="handler-class"
                    type="javaee:fully-qualified-classType"/>
            <xsd:element name="behavior"
                    type="javaee:facelet-taglib-tag-behaviorType"/>
            <xsd:element name="component"
                    type="javaee:facelet-taglib-tag-componentType"/>
            <xsd:element name="converter"
                    type="javaee:facelet-taglib-tag-converterType"/>
            <xsd:element name="validator"
                    type="javaee:facelet-taglib-tag-validatorType"/>
            <xsd:element name="source" type="javaee:string"/>
          </xsd:choice>
          <xsd:element name="attribute"
                    type="javaee:facelet-taglib-tag-attributeType"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
<xsd:element name="tag-extension"
                    type="javaee:facelet-taglib-tag-extensionType"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

<!-- ********************************************** -->

  <xsd:complexType name="facelet-taglib-tag-attributeType">
    <xsd:annotation>
      <xsd:documentation>

        The attribute element defines an attribute for the nesting
        tag.  The attribute element may have several subelements
        defining:

        description     a description of the attribute
```

```
        name            the name of the attribute

        required        whether the attribute is required or
                        optional

        type            the type of the attribute

      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
      <xsd:group ref="javaee:descriptionGroup"/>
      <xsd:element name="name"
                   type="javaee:java-identifierType"/>
      <xsd:element name="required"
                   type="javaee:generic-booleanType"
                   minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>

            Defines if the nesting attribute is required or
            optional.

            If not present then the default is "false", i.e
            the attribute is optional.

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:choice>
<xsd:sequence>
          <xsd:sequence minOccurs="0">
            <xsd:element name="type"
```

```
                type="javaee:fully-qualified-classType"
                minOccurs="0">
            <xsd:annotation>
              <xsd:documentation>

                Defines the Java type of the attributes value.
                If this element is omitted, the expected type is
                assumed to be "java.lang.Object".

              </xsd:documentation>
            </xsd:annotation>
          </xsd:element>
        </xsd:sequence>

      </xsd:sequence>

    </xsd:choice>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>


  <xsd:complexType name="facelet-taglib-tag-extensionType">
      <xsd:annotation>
          <xsd:documentation>
              Extension element for tag  It may contain
              implementation specific content.
          </xsd:documentation>
      </xsd:annotation>

      <xsd:sequence>
          <xsd:any namespace="##any"
                  processContents="lax"
                  minOccurs="0"
```

```
                maxOccurs="unbounded" />
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>
<xsd:complexType name="facelet-taglib-functionType">
        <xsd:annotation>
          <xsd:documentation>
            If the tag library XML file contains individual function
            declarations rather than pointing to a library-class or a
            declaring a composite-library name, the individual functions are
            enclosed in function elements.
          </xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:group ref="javaee:descriptionGroup"/>
            <xsd:element name="function-name" type="javaee:string"/>
            <xsd:element name="function-class"
                    type="javaee:fully-qualified-classType"/>
            <xsd:element name="function-signature" type="javaee:string"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="facelet-taglib-tag-behaviorType">
        <xsd:annotation>
          <xsd:documentation>
            Within a tag element, the behavior element encapsulates
            information specific to a JSF Behavior.
          </xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:group ref="javaee:descriptionGroup"/>
            <xsd:element minOccurs="1" maxOccurs="1"
                    name="behavior-id" type="javaee:string"/>
            <xsd:element minOccurs="0" maxOccurs="1"
                    name="handler-class" type="javaee:fully-qualified-classType"/>
```

```
            <xsd:element name="behavior-extension"
                         type="javaee:facelet-taglib-tag-behavior-extensionType"
                         minOccurs="0"
                         maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
<xsd:complexType name="facelet-taglib-tag-behavior-extensionType">
        <xsd:annotation>
            <xsd:documentation>
                Extension element for behavior. It may contain
                implementation specific content.
            </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
            <xsd:any namespace="##any"
                     processContents="lax"
                     minOccurs="0"
                     maxOccurs="unbounded" />
         </xsd:sequence>
         <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>
    <xsd:complexType name="facelet-taglib-tag-componentType">
        <xsd:annotation>
          <xsd:documentation>
            Within a tag element, the component element encapsulates
            information specific to a JSF UIComponent.
          </xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:group ref="javaee:descriptionGroup"/>
            <xsd:element name="component-type" type="javaee:string"/>
            <xsd:element minOccurs="0" maxOccurs="1"
                         name="renderer-type" type="javaee:string"/>
```

```
            <xsd:element minOccurs="0" maxOccurs="1"
                         name="handler-class"
                         type="javaee:fully-qualified-classType"/>
            <xsd:element name="component-extension"
                         type="javaee:facelet-taglib-tag-component-extensionType"
                         minOccurs="0"
                         maxOccurs="unbounded"/>

        </xsd:sequence>
    </xsd:complexType>
<xsd:complexType name="facelet-taglib-tag-component-extensionType">
        <xsd:annotation>
            <xsd:documentation>
                Extension element for component  It may contain
                implementation specific content.
            </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
            <xsd:any namespace="##any"
                     processContents="lax"
                     minOccurs="0"
                     maxOccurs="unbounded" />
         </xsd:sequence>
         <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

    <xsd:complexType name="facelet-taglib-tag-converterType">
        <xsd:annotation>
          <xsd:documentation>
            Within a tag element, the converter element encapsulates
            information specific to a JSF Converter.
          </xsd:documentation>
        </xsd:annotation>
```

```
        <xsd:sequence>
            <xsd:group ref="javaee:descriptionGroup"/>
            <xsd:element minOccurs="1" maxOccurs="1"
                    name="converter-id" type="javaee:string"/>
            <xsd:element minOccurs="0" maxOccurs="1"
                    name="handler-class" type="javaee:fully-qualified-classType"/>
            <xsd:element name="converter-extension"
                    type="javaee:facelet-taglib-tag-converter-extensionType"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
<xsd:complexType name="facelet-taglib-tag-converter-extensionType">
        <xsd:annotation>
            <xsd:documentation>
                Extension element for converter  It may contain
                implementation specific content.
            </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>
            <xsd:any namespace="##any"
                    processContents="lax"
                    minOccurs="0"
                    maxOccurs="unbounded" />
         </xsd:sequence>
         <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

    <xsd:complexType name="facelet-taglib-tag-validatorType">
        <xsd:annotation>
          <xsd:documentation>
            Within a tag element, the validator element encapsulates
            information specific to a JSF Validator.
```

```
        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:group ref="javaee:descriptionGroup"/>
        <xsd:element minOccurs="1" maxOccurs="1"
                name="validator-id" type="javaee:string"/>
        <xsd:element minOccurs="0" maxOccurs="1"
                name="handler-class" type="javaee:fully-qualified-classType"/>
        <xsd:element name="validator-extension"
                type="javaee:facelet-taglib-tag-validator-extensionType"
                minOccurs="0"
                maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
<xsd:complexType name="facelet-taglib-tag-validator-extensionType">
      <xsd:annotation>
        <xsd:documentation>
          Extension element for validator  It may contain
          implementation specific content.
        </xsd:documentation>
      </xsd:annotation>

      <xsd:sequence>
        <xsd:any namespace="##any"
                processContents="lax"
                minOccurs="0"
                maxOccurs="unbounded" />
       </xsd:sequence>
       <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

    <!-- ********************************************** -->
    <xsd:simpleType name="facelet-taglib-versionType">
      <xsd:annotation>
```

```
            <xsd:documentation>
                This type contains the recognized versions of
                facelet-taglib supported.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:restriction base="xsd:token">
            <xsd:enumeration value="2.0"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:complexType name="facelet-taglib-canonical-nameType">

      <xsd:annotation>
        <xsd:documentation>

          Defines the canonical name of a tag or attribute being
          defined.

          The name must conform to the lexical rules for an NCName

        </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
      <xsd:extension base="xsd:NCName">
        <xsd:attribute name="id" type="xsd:ID"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

</xsd:schema>
```

B

Appendix B - Change Log

2.1 Changes Between 1.1 and 1.2

Unified Expression Language (EL)

Previous versions of the JavaServer Faces included an innovative, EL tailored to the needs of Faces. The main emphasis of this version of the Faces spec, and also the focus of the JSP spec corresponding to it, is to take those innovations and expose them to JSP page authors by creating a Unified EL that leverages the combined power of the Faces and JSP ELs. The Faces EL would then be deprecated, and the deprecated implementation would be written in terms of the Unified EL to preserve backwards compatability.

2.1.0.1 Guide to Deprecated Methods Relating to the Unified EL and their Corresponding Replacements

The following classes and methods have been deprecated:

- `javax.faces.el.EvaluationException`
  - replaced by: `javax.el.ELException`
- `javax.faces.el.MethodBinding`
  - replaced by: `javax.el.MethodExpression`
- `javax.faces.el.MethodNotFoundException`
  - replaced by: `javax.el.MethodNotFoundException`
- `javax.faces.el.PropertyNotFoundException`
  - replaced by: `javax.el.PropertyNotFoundException`
- `javax.faces.el.PropertyResolver`
  - replaced by: `javax.el.ELResolver`
- `javax.faces.el.ReferenceSyntaxException`
  - replaced by: `javax.el.ELException`
- `javax.faces.el.ValueBinding`
  - replaced by: `javax.el.ValueExpression`
- `javax.faces.el.VariableResolver`
  - replaced by: `javax.el.ELResolver`

- javax.faces.application.Application.createComponent(ValueBinding componentBinding, FacesContext context, String componentType)
  - replaced by: javax.faces.application.Application.createComponent(ValueExpression componentExpression, FacesContext context, String componentType)
- javax.faces.application.Application.createMethodBinding
  - replaced by: javax.faces.application.Application.createMethodExpression
- javax.faces.application.Application.createValueBinding
  - replaced by calling: javax.faces.application.Application.getExpressionFactory then ExpressionFactory.createValueExpression
  - see Javadoc for javax.faces.application.Application.createValueBinding
- javax.faces.application.Application.getPropertyResolver
  - replaced by: javax.faces.application.Application.getELResolver
- javax.faces.application.Application.setPropertyResolver
  - see Javadoc for javax.faces.application.Application.setPropertyResolver
- javax.faces.application.Application.getVariableResolver
  - replaced by: javax.faces.application.Application.getELResolver
- javax.faces.application.Application.setVariableResolver
  - see Javadoc for javax.faces.application.Application.setVariableResolver
- javax.faces.component.ActionSource.getAction
  - replaced by: javax.faces.component.ActionSource2.getActionExpression
- javax.faces.component.ActionSource.setAction
  - replaced by: javax.faces.component.ActionSource2.setActionExpression
- javax.faces.component.ActionSource.getActionListener
  - replaced by: javax.faces.component.ActionSource.getActionListeners
  - see Javadoc for javax.faces.component.ActionSource.getActionListener
- javax.faces.component.ActionSource.setActionListener
  - replaced by: javax.faces.component.ActionSource.addActionListener
- javax.faces.component.EditableValueHolder.getValidator
  - replaced by: javax.faces.component.EditableValueHolder.getValidators
  - see Javadoc for: javax.faces.component.EditableValueHolder.getValidator
- javax.faces.component.EditableValueHolder.setValidator
  - replaced by: javax.faces.component.EditableValueHolder.addValidator
  - see Javadoc for: javax.faces.component.EditableValueHolder.setValidator
- javax.faces.component.EditableValueHolder.getValueChangeListener
  - replaced by: javax.faces.component.EditableValueHolder.getValueChangeListeners
  - see Javadoc for: javax.faces.component.EditableValueHolder.getValueChangeListener
- javax.faces.component.EditableValueHolder.setValueChangeListener
  - replaced by: javax.faces.component.EditableValueHolder.addValueChangeListener
  - see Javadoc for: javax.faces.component.EditableValueHolder.setValueChangeListener
- javax.faces.component.UICommand.getAction
  - replaced by: javax.faces.component.UICommand.getActionExpression
- javax.faces.component.UICommand.setAction
  - replaced by: javax.faces.component.UICommand.setActionExpression
- javax.faces.component.UICommand.getActionListener

    - replaced by: javax.faces.component.UICommand.getActionListeners
    - see Javadoc for: javax.faces.component.UICommand.getActionListener
- javax.faces.component.UICommand.setActionListener
    - replaced by: javax.faces.component.UICommand.addActionListener
    - see Javadoc for: javax.faces.component.UICommand.setActionListener
- javax.faces.component.UIComponentBase.getValueBinding
    - replaced by: javax.faces.component.UIComponentBase.getValueExpression
- javax.faces.component.UIComponentBase.setValueBinding
    - replaced by: javax.faces.component.UIComponentBase.setValueExpression
- javax.faces.component.UIComponent.getValueBinding
    - replaced by: javax.faces.component.UIComponent.getValueExpression
- javax.faces.component.UIComponent.setValueBinding
    - replaced by: javax.faces.component.UIComponent.setValueExpression
- javax.faces.component.UIData.setValueBinding
    - replaced by: javax.faces.component.UIData.setValueExpression
- javax.faces.component.UIGraphic.getValueBinding
    - replaced by: javax.faces.component.UIGraphic.getValueExpression
- javax.faces.component.UIGraphic.setValueBinding
    - replaced by: javax.faces.component.UIGraphic.setValueExpression
- javax.faces.component.UIInput.getValidator
    - replaced by: javax.faces.component.UIInput.getValidators
    - see Javadoc for: javax.faces.component.UIInput.getValidator
- javax.faces.component.UIInput.setValidator
    - replaced by: javax.faces.component.UIInput.addValidator
    - see Javadoc for: javax.faces.component.UIInput.setValidator
- javax.faces.component.UIInput.setValueChangeListener
    - replaced by: javax.faces.component.UIInput.addValueChangeListener
    - see Javadoc for: javax.faces.component.UIInput.setValueChangeListener
- javax.faces.component.UISelectBoolean.getValueBinding
    - replaced by: javax.faces.component.UISelectBoolean.getValueExpression
- javax.faces.component.UISelectBoolean.setValueBinding
    - replaced by: javax.faces.component.UISelectBoolean.setValueExpression
- javax.faces.component.UISelectMany.getValueBinding
    - replaced by: javax.faces.component.UISelectMany.getValueExpression
- javax.faces.component.UISelectMany.setValueBinding
    - replaced by: javax.faces.component.UISelectMany.setValueExpression

New Methods not replacing a Deprecated methods:

- javax.faces.component.UIViewRoot.getBeforePhaseListener
- javax.faces.component.UIViewRoot.setBeforePhaseListener
- javax.faces.component.UIViewRoot.getAfterPhaseListener
- javax.faces.component.UIViewRoot.setAfterPhaseListener

Guide to Deprecated Methods Relating to State Management and their Corresponding Replacements

The following classes and methods have been deprecated:

- `javax.faces.application.StateManager.SerializedView`
  - replaced by `java.lang.Object` that implements `java.io.Serializable`
- `javax.faces.application.StateManager.saveSerializedView`
  - replaced by `javax.faces.application.StateManager.saveView`
- `javax.faces.application.StateManager.getTreeStructureToSave`
  - The separation between tree structure and component state is now a recommended implementation detail.
- `javax.faces.application.StateManager.getComponentStateToSave`
  - The separation between tree structure and component state is now a recommended implementation detail.
- `javax.faces.application.StateManager.writeState that takes a SerializedView`
  - replaced by `javax.faces.application.StateManager.writeState` that takes a `java.lang.Object` that implements `Serializable`.
- `javax.faces.application.StateManager.restoreTreeStructure`
  - The separation between tree structure and component state is now a recommended implementation detail.
- `javax.faces.application.StateManager.restoreComponentState`
  - The separation between tree structure and component state is now a recommended implementation detail.
- `javax.faces.render.ResponseStateManager.writeState that takes a javax.faces.application.StateManager.SerializedView`
  - Replaced by `javax.faces.render.ResponseStateManager.writeState that takes a java.lang.Object` that implements `java.io.Serializable.`
- `javax.faces.render.ResponseStateManager.getTreeStructureToRestore`
  - The separation between tree structure and component state is now a recommended implementation detail. Semantically has been replaced by `javax.faces.render.ResponseStateManager.getState`.
- `javax.faces.render.ResponseStateManager.getComponentStateToRestore`
  - The separation between tree structure and component state is now a recommended implementation detail. Semantically has been replaced by `javax.faces.render.ResponseStateManager.getState`.

JavaServer Faces 1.2 Backwards Compatibility

- Faces 1.2 is backwards compatible with Faces 1.1. This means that a web-application that was developed to run with Faces 1.1 won't require any modification when run with Faces 1.2 except in the cases described in the following section.
- Note that Faces is a part of the Java EE platform as of Faces 1.2. A web application therefore does not need to bundle a Faces implementation anymore when it runs on a web container that is Java EE technology compliant. Should a Faces implementation be bundled with a web-application, it will simply be ignored as the Faces implementation provided by the platform always takes precedence.
- The JSP aspects of backwards compatability are described in the JSP specification in the Preface, in the section titled "Backwards Compatability with JSP 2.0".

Breakages in Backwards Compatability

- In Faces 1.1 you could override implicit objects in your custom resolvers. For example, for the following expression: `${param['x']}` you could change the meaning of `param` in your custom `VariableResolver.` In Faces 1.2, implicit objects are always recognized - so `param` will always mean a map of parameters. See Section 5.3 "The Managed Bean Facility"

- In Faces 1.1, any custom resolvers that do not honor the “decorator” pattern - that is, delegate to their parent resolver, will still work in Faces 1.2 with the following clarification: those resolvers would operate independently with regards to other resolvers in the chain. See *Section 5.6.1 “Faces ELResolver for JSP Pages”*.
- In Faces 1.1 it was valid to call `setVariableResolver()` or `setPropertyResolver()` on the `Application` at any point in the application’s lifetime. This allowed for the application to be in an indeterminate state. In Faces 1.2, neither of these methods may be called after the application has served any requests.
- In Faces 1.1, if a view couldn’t be restored due to session expiration, we’d create a new one and go to render response. In 1.2, this is not the case. We now throw a `ViewExpiredException`. 1.1-based applications may rely on the old behavior to forward to a login page when a session expired. 1.2 circumvents this.

General changes

The numbers in the text below refer to issue numbers in the issue tracker found at <https://javaserverfaces-spec-public.dev.java.net/servlets/ProjectIssues>.

- 2 - Clarify that for client side state saving, the state should be encrypted for security.
- 3 - Clarify the specification with respect to constraint violations for tags in the Core Tag Library.
- 4 - Added `headerClass` and `footerClass` attributes at the “h:column” level. Please see Section 8.6 “Standard HTML RenderKit Implementation” for more details.
- 5 - Clarified the use of a string literal for the “action” attribute on ActionSource components.
- 6 - Introduced a new optional “label” attribute for input components that will provide an association between a component, and the message that the component (indirectly) produced. Please refer to Section 8.6 “Standard HTML RenderKit Implementation” and Section 2.5.2.4 “Localized Application Messages” for more details.
- 8 - Made UViewRoot a source of PhaseEvent(s) for all phases of the request processing lifecycle except RestoreView. Provided additional “before” and “after” phase listener attributes for the <f:view> tag. Please see Section 4.1.19 “UIViewRoot” for more details.
- 9 - Clarified the behavior of PhaseListener implementations in the following way: they must guarantee that if “beforePhase()” is called, then “afterPhase()” must also be called, regardless of any thrown exceptions. Please see Section 12.3 “PhaseListener” for more specifics.
- 11 - Provide a unique window identifier (in addition to the “viewid”) to accomodate applications that have mutiple instances of the same view, but perhaps in different browser windows or frames.
- 13 - Specified “by type” converter registration for BigDecimal and BigInteger.
- 15 - Enhanced the usage of the “Decorator Pattern” for ViewHandler, StateManager and ResponseWriter classes by providing abstract wrapper classes to make it easier to override a subset of the total methods for those classes. Please see Section 11.4.6 “Delegating Implementation Support” for more details.
- 16 - Provided additional I18n attributes “dir and “lang” for the tags: <h:outputText>, <h:outputFormat>, <h:messages>, <h:message>. Please see Section 8.6 “Standard HTML RenderKit Implementation” for descriptions of these components.
- 17 - Introduced a new optional “layout” attribute on the “PanelGroup” component that controls the rendering of either a “div” or “span” HTML element. Please see Section 8.6 “Standard HTML RenderKit Implementation” for more details.
- 18 - When a resource lookup is done on the java.util.Map (loaded from <f:loadBundle>) using a key, and the key is not found in the Map, return the literal string ???KEY??? where KEY is the key being looked up in the Map (instead of throwing a MissingResourceException). Throw a JspException if the named bundle identified by <f:loadBundle> does not exist. Please see *Section 9.4.7 “<f:loadBundle>”*.
- 20 - Specify that the event queue should be cleared after each phase (except RestoreViewPhase and RenderResponse) if “responseComplete” or “renderResponse” has been set on the FacesContext.
- 21 - Provided an additional “binding” attribute for the core Converter, Listener and Validator tags that would be used as a ValueExpression to alternatively create the Converter, Listener or Validator instance. Please see *Section 9.4 “JSF Core Tag Library”* for more details.

- 27 - <h:messages> now renders HTML list elements ("<ul>","<li>") if the "layout" attribute is "list" or the "layout" attribute is not specified. If the "layout" is "table", an HTML "table" element is rendered instead of an outer "span". Please see Section 8.6 "Standard HTML RenderKit Implementation" for more details.
- 29 - Allow the use of user-defined "onclick" Javascript on CommandLink.
- 30 - Make the "commandButton" "image" attribute render the same as the "graphicImage" "img" attribute for consistency. Please see Section 8.6 "Standard HTML RenderKit Implementation" for more information.
- 35 - Provided a new facet for DataTable Renderer that allows the rendering of a table "caption" element immediately following the "table" element. Also provided style sheet attributes for this new element. Please see Section 8.6 "Standard HTML RenderKit Implementation" for a descrption of this component.
- 43 - Migrated over to using XML Schema (from DTD) for configuation file validation. Please see Section 1.1 "XML Schema Definition for Application Configuration Resource file".
- 45 - Avoided concurrent read issues by using a java.util.HashMap instead of java.util.WeakHashMap for a component's Property Descriptor Map. This also fixes the performance problem as identified in the forum. Please refer to the Property Descriptor methods and the constructor in `javax.faces.component.UIComponentBase.`
- 47 - Introduced a mechanism to detect if a request is a postback.
- 48 - Specify the algorithm used for client id generation as well as provide a way to allow the page author to specify exactly what the client Id should be, and preventing Faces from altering it.
- 50 - Allow an application to specify multiple render kits by introducing an optional "renderKitId" attribute on "<f:view>". It is no longer required to write a custom ViewHandler to incorporate a different render kit. Please refer to Section 8.4 "ResponseStateManager" and Section 9.4.21 "<f:view>" for more details.
- 51 - Clarify the specification with respect to "Application Startup Behavior". Allow implementations to check for the presence of a servlet-class definition in a web application deployment descriptor as a means to abort the configuration and save startup time.
- 54 - Added new extension elements to the Faces XML schema. Please see Section 1.1 "XML Schema Definition for Application Configuration Resource file".
- 55 - For postback requests, in the "RestoreViewPhase", during `ValueExpression` examination for each component in the component tree, specify that calling the `setValue()` method on each `ValueExpression`, should be done in a "parent-first" fashion, calling the `setValue()` method and then traversing the children.
- 58 - Enabed "protected" access to internal "DataModel" in UIData.
- 59 - Avoid EL expression evaluation for "value" attribute on "AttributeTag". "AttributeTag" now passes the expression to UIComponent for evaluation.
- 65 - Added standard converter messages. Please see Section 2.5.2.4 "Localized Application Messages" for more details.
- 66 - Specified that "FormRenderer" must render the "name" attribute with the same value as the "id" attribute. Please see Section 8.6 "Standard HTML RenderKit Implementation" for more details.
- 67 - Allow the resetting of an input component's value by introducing a `resetValue()` method on `UIInput.`
- 68 - Specify that the component tree may be manipulated throughout the request processing lifecycle, except during render. Please see Section 2.2.6 "Render Response" for more details.
- 69 - Permit the passing of `a null` value to `SelectItem.setValue().`
- 72 - Improve XHTML compliance by rendering both "lang" and "xml:lang" attributes.
- 73 - Added a new FacesException - "javax.faces.application.ViewExpiredException". Specified that implementations must throw this exception when an attempt to restore a view results in failure on postback. Please see Section 2.2.1 "Restore View" for more details.
- 74 - Added "disabled" property to "outputLink" and "commandLink". Please see Section 8.6 "Standard HTML RenderKit Implementation" for more details.
- 75 - Added "getRequestContentType" and "getResponseContentType" to ExternalContext.
- 78 - Added a more "user-friendly" default error message for UIInput "update model". Please see Section 2.5.2.4 "Localized Application Messages" for more details.

- 80 - Specify that the JSF Core Tag Library must not contain any tags that cause JavaScript to be rendered to the client.
- 81 - Enable the message displayed for “required” validation, conversion, and validation to be overridden by the page author (JSP or non-JSP)
- 82 - Added new feature, the ability to resolve ResourceBundles via the EL without the use of the <f:loadBundle> tag.
- 84 - Added `rendered` attribute to the core `f:verbatim` tag. Please see *Section 9.4 “JSF Core Tag Library”* for more details.
- 85 - Add new tag: f:setPropertyActionListener. Useful for pushing values into managed beans without allowing modification of the value.
- 86 - Specified that “OutputLinkRenderer” must render the “name” attribute with the same value as the “id” attribute. Please see Section 8.6 “Standard HTML RenderKit Implementation” for more details.
- 87 - Modified specification for the `setVariableResolver()` and `setPropertyResolver()` methods on `Application` to state that they may not be called after the application has served any requests.
- 93 - Added “escape” flag indicating the text of UISelectItem should be escaped when rendering.
- 95 - Allow multiple instances of `FacesServlet` in a single webapp, mapped with different URI mappings, to use different implementations of `Lifecycle` by allowing the lifecycle-id to be specified as an `init-param` in addition to the existing way of specifying it as a `context-param`.
- 98 - Specified that “SelectManyCheckboxListRenderer: and “RadioRenderer” must render the “label” element after the “input element for each “SelectItem. Specified that the “label” element must refer to the “input” element using the “for” attribute. Please see Section 8.6 “Standard HTML RenderKit Implementation” for more details.
- 99 - Specified Java EE 5 Generics usage where applicable.
- 105 - Specified that for commandButton rendering, the “image” attribute value must not be escaped. Specified that for graphicImage rendering, the “src” attribute value must not be escaped.
- 108 - Specified that JSF implementations that are part of a Java EE technology-compliant implementation are required to validate the application resource file against the XML schema for structural correctness. Also specified that validation is recommended, but not required for JSF implementatons that are not part of a Java EE technology compliant implementation. Please refer to Section 11.4.2 “Application Startup Behavior” for more details.
- 111 - Specified that a component must allow child components to be added to and removed from the child list of the current component, even though the child component returns null from getParent().
- 118 - Specified that an implementation of Map returned from ExternalContext.getSessionMap implement a “clear” method that calls “removeAttribute” on each attribute in the Servlet or Portlet session.
- 119 - Specified that implementations running in a JSR-250 compliant container have their managed bean methods annotated with @PostConstruct be called after the object is instantiated, and after injection is performed, but before the bean is placed into scope. Specified that methods annotated with @PreDestroy be called when the scope for the bean is ending.
- 120 - Specified in the renderkit docs that commandButton rendering can generate javascript for “onclick” attribute.
- 122 - Clarified renderkit docs with respect to what gets rendered for disabled command link attributes.
- 123 - Clarified renderkit docs with respect to dataTable attribute rendering.
- 124 - Clarified renderkit docs with repsect to graphicImage “alt” attribute.
- 131 - Specified that a compliant implementation must allow the registration of a converter for class java.lang.String and java.lang.String.TYPE that will be used to convert values for these types.
- 133 - Removed the incorrect statement: “"It is the callers responsibility to ensure that setViewId() is called on the returned view, passing the same viewId value." pertaining to ViewHandler.createView()
- 134 - Fixed backwards compatability issues.
- 135 - Support Java EE 5 enums as valid types/
- 138 - Change the required return type for action methods to be Object instead of String. This allows the usage of Enums for the return type of action methods, as long as the toString() method of the enum matches the expected value in the application configuration resources.

- 145 - Define new method on UIComponent, invokeOnComponent(). This will find a component in the tree by clientId and invoke a user specified callback on it. Please see Section 3.1.8 “Component Tree Navigation” and Section 4.1.1.3 “Methods” [of UIData] for more details.
- jsf-ri 127 - Specify that FacesContext methods getClientIdsWithMessages( ) and getMessages( ) must be implemented using order-preserving structures so the items in the iterator are returned in the order they were added with addMessage().Spec document changes
- 147 - Clarified grammer with respect to component id.
- 151 - Specified standard converter for Enums
- 152 - Specified EL coercion usage in API javadocs UISelectOne/UISelectMany (when items are compared in validation) and standard html renderkit docs during encoding of select components.
- 154 - Fixed FacesTag “name” attribute discrepency - made it a String (was ValueExpression).
- 155 - Specified “columnClasses”, “rowClasses” descriptions for panelGrid in renderkit docs.
- 160 - Added and specified ResponseWriter.writeText method that takes a UIComponent argument.
- The TLDDocs for the h: tag library are now a normative part of the spec.

Following is a section by section breakdown of the main changes since the last release of this document. This work was done mainly to support changes in tree creation and content interweaving for JSP based faces applications, as well as for fixing the following issues from the above list: 2 3 4 5 6 8 9 11 13 15 16 17 18 20 21 27 29 30 35 43 45 47 48 50 51 53 54 55 58 59 65 66 67 68 69 72 73 74 75 78 80 81 82 84 85 86 93 95 98 99 105 108 111 118 119 120 122 123 124 131 133 134 135 138 145 147 151 152 154 155 160.

Preface

- Added new section: Guide to Deprecated Methods Relating to State Management and their Corresponding Replacements.

Section 2.2.1 “Restore View”

- Do per-component actions in a "parent-first" fashion, calling the setValue() method and *then* traversing the children.
- Describe the new responsibilities of this phase with respect to the new StateSaving changes.
- Describe when ViewHandler.initView() is to be called.
- Describe how the ViewHandler.calculateRenderKitId() and ResponseStateManager.isPostback() method are to be used.
- Specify that implementations must throw `javax.faces.application.ViewExpiredException` when an attempt to restore a view results in failure on postback.

Section 2.2.6 “Render Response”

- Specify that the component tree may be manipulated throughout the request processing lifecycle, except during render.

Section 2.4.2.1 “Create A New View”

Document that multiple renderkits are supported.

Section 2.5.2.4 “Localized Application Messages”

Added updates to standard messages; Also mentioned new parameter substitution token for the generic input component attribute "label".

Section 3.1.11 “Generic Attributes”

Completely specify how attribute/property transparency works.

Section 3.1.13 “Component Specialization Methods”

Add new method, encodeAll(), which is now the preferred method for developers to call to render a child or facet().

Section 4.1.4 “UIForm”

Document the new `prependId` property and `getContainerClientId()` method.

UIData Section 4.1.3.2 “Properties”

Added `protected` property for `DataModel`.

UIInput Section 4.1.6 “UIInput”

- Document the behavior of the `requiredMessage`, `converterMessage` and `validatorMessage` properties.

UIInput Section 4.1.6.3 “Methods”

- Add mention of resetValue() to the "Methods" section for UIInput.

Section 4.1.19 “UIViewRoot”

- Change callsite for saveSerializedView and writeState().
- Change to clear the event queue after each phase if skipping to rendering response.
- JSP tag no longer deals with state saving.

Section 5.1.2 and 5.1.3 “ValueExpression Syntax” and “ValueExpression Semantics”

Removed and made reference to EL spec.

Section 5.2.1 “MethodExpression Syntax and Semantics”

Make reference to EL spec.

Section 5.4 “Leveraging Java EE 5 Annotations in Managed Beans”

- This new section covers the modifications necessary to allow managed beans to be the target of container managed dependency injection using the `@Resource` and `@EJB` annotations.
- Added section 5.4.1 that specifies how the `@PostConstruct` and `@PreDestroy` annotations must be handled.

Section 5.5.3 “ExpressionFactory”

- Update signature of createValueExpresion() and createMethodExpression() to include ELContext as the first argument.

Section 5.6.1.4 “ResourceBundle ELResolver for JSP Pages”

- This resolver, when coupled with the `javax.el.ResourceBundleELResolver`, allows the resolution of ResourceBundles and entries therein via the EL. See also Section 5.6.2.6 “el.ResourceBundleELResolver” and Section 5.6.2.7 “ResourceBundle ELResolver for Programmatic Access”.

Section 7.5.1 “Overview” ViewHandler

- Document new methods initView() and calculateCharacterEncoding();

Section 7.5.2 “Default ViewHandler Implementation”

- modify createView() to reflect current reality:
  - Do the viewId discovery algorithm formerly in restoreView().
  - Redirect to the context root if no viewId can be discerned.
  - Do the existing processing.
- modify restoreView() to reflect current reality:
  - Do the existing processing.
  - Do the viewId discovery algorithm now in createView().
  - If no viewId can be discovered, return null.
  - Always call StateManager.restoreView().
  - no longer set the character encoding, this has moved out to the Restore View phase implementation
- change callsite for saveSerializedView() to be saveView().
- added ViewHandler.calculateRenderKitId responsibility of returning the request parameter named ResponseStateManager.RENDER_KIT_ID_PARAM if not null.

State Saving Section 7.7.1 “Overview”

- Soften the wording about the separation between tree structure and component state, say it's only a recommendation to keep these two separate.

Section 7.7.2 “State Saving Alternatives and Implications”

- Modified client state saving text to add "It is advisable that this information be encrypted and tamper evident, since it is being sent down to the client, where it may persist for some time."

- Modified server state saving text to add "Implementations that wish to enable their saved state to fail over to a different container instance must keep this in mind when implementing their server side state saving strategy. The default implementation Serializes the view in both the client and server modes. In the server mode, this serialized view is stored in the session and a unique key to retrieve the view is sent down to the client. By storing the serialized view in the session, failover may happen using the usual mechanisms provided by the container."
- The values of all component attributes and properties must implement `Serializable`.
- New section 7.7.6 StateManager in the "Deprecated APIs" section (7.7)
- New section 7.7.7 ResponseStateManager in the "Deprecated APIs" section (7.7)

Section 8.4 “ResponseStateManager”

Describe the non-deprecated methods.

- Added verbiage about ResponseStateManager implementation's responsibility of writing out render kit identifier.
- Describe the isPostback() method.

Section 9.1 “UIComponent Custom Actions”

- Specify that `id` is now `rtexprvalue` `true`.

Section 9.2.8 “Interoperability with JSP Template Text and Other Tag Libraries”

- Changes in the current version of the EL allow Faces applications to use JSTL `<c:forEach>` tags with Faces components as long as the `items` attribute points to a deferred EL expression (ie, a `#{}` expression, as opposed to an immediate `${}` expression).
- Also, remove the requirements that `<f:verbatim>` be used, and that components added to the tree programmatically will only be rendered if they are the children of a `rendersChildren==true` component

Section “Integration with JSP”

- Changes to account for moving from `UIComponentTag`/`UIComponentBodyTag` to `UIComponentELTag.`

Section 9.3.1.2 “Faces 1.0 and 1.1 Taglib migration story”

- Describe the new `jsp-version` TLD based migration story.

Section 9.4 “JSF Core Tag Library”

- For listener/converter/validator tags, clarified that exceptions would be rethrown as JspException. Also specfy JspException should be thrown if certain constraints are not met.
- Specify that tags may have non-deferred expressions as the value of their `id` attribute.
- Added `binding` attribute to listener/converter/validator tags.
- Added `rendered` attribute to verbatim tag.
- Specify that none of the tags in the JSF Core Tag Library may cause JavaScript to be rendered to the client.

Section 9.4.2 “<f:attribute>”

- Specify that the argument value must be interrogated to see if it isLiteralText(). If so, store in the attributes set, If not, store in the ValueExpression set.

Section 9.4.12 “<f:setPropertyActionListener>”

- New Section, document this new tag.

Section 9.4.21 “<f:view>”

- Added renderKitId attribute description to f:view;

Section 9.5 “Standard HTML RenderKit Tag Library”

- Specify how to handle action attributes that are string literals.
- Call out to TLDDocs for parts of the requirements. TLDDocs are now normative.

Section 11.2.6.2 “FacesServlet”

- Describe how the `init-param` then the `context-param` must be consulted for the lifecycleID for this `FacesServlet` instance.

Section 11.3 “Deprecated APIs in the webapp package”

New section describing deprecated APIs. Previous section at this address moved to next section number.

Section 11.4.2 “Application Startup Behavior”

- Implementations may check for the presence of a servlet-class definition of class javax.faces.webapp.FacesServlet in the web application deployment descriptor as a means to abort the configuration process and reduce startup time for applications that do not use JavaServer Faces Technology.

Chapter A “XML Schema Definition for Application Configuration Resource file

- New appendix for XML Schema and DTD