JAKARTA EE

Jakarta Faces

Jakarta Faces Team, https://projects.eclipse.org/projects/ee4j.faces

4.1, May 07, 2024: Final

Table of Contents

```
Specification: Jakarta Faces

Version: 4.1

Status: Final

Release: May 07, 2024
```

Eclipse Foundation Specification License - v1.1

By using and/or copying this document, or the Eclipse Foundation document from which this statement is linked or incorporated by reference, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to copy, and distribute the contents of this document, or the Eclipse Foundation document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the document, or portions thereof, that you use:

- link or URL to the original Eclipse Foundation document.
- All existing copyright notices, or if one does not exist, a notice (hypertext is preferred, but a textual representation is permitted) of the form: "Copyright (c) [$date-of-document] Eclipse Foundation AISBL <<url to this license>> "

Inclusion of the full text of this NOTICE must be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

No right to create modifications or derivatives of Eclipse Foundation documents is granted pursuant to this license, except anyone may prepare and distribute derivative works and portions of this document in software that implements the specification, in supporting materials accompanying such software, and in documentation of such software, PROVIDED that all such works include the notice below. HOWEVER, the publication of derivative works of this document for use as a technical specification is expressly prohibited.

The notice is:

"Copyright (c) [$date-of-document] Eclipse Foundation AISBL. This software or document includes material copied from or derived from [title and URI of the Eclipse Foundation specification document]."

Disclaimers

Preface

This is the Jakarta Faces Specification Document, version 4.1.

Related Technologies

Other Jakarta Platform Specifications

Jakarta Faces 4.1 is based on the following Jakarta specifications:

Jakarta Servlet, version 6.1 https://jakarta.ee/specifications/servlet/

Jakarta Expression Language, version 6.0 https://jakarta.ee/specifications/expression-language/

Jakarta Contexts and Dependency Injection, version 4.1 https://jakarta.ee/specifications/cdi/

Jakarta Bean Validation, version 3.1 https://jakarta.ee/specifications/bean-validation/

Jakarta Tags, version 3.0 https://jakarta.ee/specifications/tags/

Jakarta WebSocket, version 2.2, optional https://jakarta.ee/specifications/websocket/

Jakarta JSON Processing, version 2.1, optional https://jakarta.ee/specifications/jsonp/

JavaBeans™ Specification, version 1.01 https://www.oracle.com/java/technologies/javase/javabeans-spec.html

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

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

Portlet Specification 1.0, JSR-168 https://jcp.org/en/jsr/detail?id=168

Portlet Specification 2.0, JSR-286 https://jcp.org/en/jsr/detail?id=286

Portlet Specification 3.0, JSR-362 https://jcp.org/en/jsr/detail?id=362

Portlet Bridge Specification, JSR-301 https://jcp.org/en/jsr/detail?id=301

Related Documents and Specifications

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

Hypertext Markup Language (HTML), Living Standard https://html.spec.whatwg.org/

Extensible HyperText Markup Language (XHTML), version 1.0 https://www.w3.org/TR/xhtml1/

Extensible Markup Language (XML), version 1.0 (Second Edition) https://www.w3.org/TR/REC-xml/

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

The Facelet tag library for the HTML_BASIC standard RenderKit is specified in the VDLDocs 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) https://www.rfc-editor.org/rfc/rfc2119.txt

Providing Feedback

We welcome any and all feedback about this specification. Please email your comments to <faces-dev@eclipse.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 authors would like to thank the Jakarta Faces 4.1 contributors.

- Arjan Tijms
- Bauke Scholtz
- Thomas Andraschko
- Manfred Riem
- Emil Lefkof
- Paolo B
- Nicola Isotta
- Emil Sierżęga
- Jason Lee
- Volodymyr Siedlecki
- Hantsy Bai
- Paul Nicolucci

The editors would like to recognize the following individuals who have contributed to the success of Jakarta Faces over the years.

Ed Burns	Frank Caputo	Cagatay Civici
Ken Fyten	Neil Griffin	Josh Juneau
Brian Leatham	Kito Mann	Michael Müller
Paul Nicolucci	Leonardo Uribe	Dan Allen
Thomas Asel	Jennifer Ball	Lincoln Baxter III
Hans Bergsten	Shawn Bayern	Joseph Berkovitz
Dennis Byrne	Pete Carapetyan	Ryan DeLaplante
Keith Donald	Jim Driscoll	Hanspeter Duennenberger
Ken Finnigan	Amy Fowler	Mike Freedman
David Geary	Manfred Geiler	Ted Goddard
Juan Gonzalez	Jeremy Grelle	Rick Hightower
Jacob Hookom	Justyna Horwat	Alexander Jesse
Max Katz	Roger Keays	Gavin King
Roger Kitain	Eric Lazarus	Felipe Leme
Cody Lerum	Alberto Lemos	Ryan Lubke
Barbara Louis	Martin Marinschek	Kumar Mettu
Craig McClanahan	Pete Muir	Bernd Müller
Michael Müller	Hans Muller	Brendan Murray
Michael Nash	Imre Osswald	Joe Ottinger
Ken Paulsen	Dhiru Pandey	Raj Premkumar
Werner Punz	Matt Raible	Hazem Saleh
Andy Schwartz	Yara Senger	Stan Silvert
Vernon Singleton	Bernhard Slominski	Alexander Smirnov
Thomas Spiegel	Kyle Stiemann	James Strachan
Jayashri Visvanathan	Ana von Klopp	Matthias Wessendorf
Adam Winer	Mike Youngstrom	John Zukowski
Christoph Straßer		

Chapter 1. Overview

Jakarta Faces 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. Jakarta Faces 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, Jakarta Faces 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, Jakarta Faces 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

Jakarta Faces’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 Jakarta Servlet and Jakarta Server Pages 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. Jakarta Faces 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 *Jakarta 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 Jakarta Faces, 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 Facelets) 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 Jakarta Faces implementors. The components themselves will be represented as configurable objects that utilize the dynamic markup capabilities of the underlying rendering technology. When Facelets are in use, for example, Jakarta Faces components will be represented as namespaced tags, which will support configuring the attributes of those components as tag attributes in the page. In addition, the pages produced by a page author will be used by the Jakarta Faces 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 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 Jakarta Faces 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 Jakarta Faces implementations
- Tool providers whose tools can leverage the specific capabilities of those libraries in development of Jakarta Faces-based applications
- Jakarta Faces implementors who provide implementation-specific component libraries as part of their Jakarta Faces product suite

Within Jakarta Faces, 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 Jakarta Faces 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 Jakarta Faces-based web applications (such as creating schemas in a relational database management system)
- Create a Java object representation of the persistent information, such as Jakarta Persistence entities, 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 CDI beans.
- Expose the data representation and functional logic objects for use via Jakarta Faces, as would be done for any Jakarta Servlet- or Jakarta Server Pages-based application.

Only the latter responsibility is directly related to Jakarta 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 Jakarta Servlet API), via *value reference expressions*, as described in Standard User Interface Components."
- Provide application-level event handlers for the events that are enqueued by Jakarta Faces components during the request processing lifecycle, as described in Invoke Application.

Application modules interact with Jakarta Faces 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 Jakarta Faces-based applications, rather than creating such applications directly. Jakarta Faces APIs support the creation of a rich variety of development tools, which can create applications that are portable across multiple Jakarta Faces 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 Jakarta Faces
- Web application frameworks (such as MVC-based and workflow management systems) that facilitate the use of Jakarta Faces 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 Jakarta Faces 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. Jakarta Faces Implementors

Finally, *Jakarta Faces implementors* will provide runtime environments that implement all of the requirements described in this specification. Typically, a Jakarta Faces implementor will be the provider of a Jakarta EE application server, although it is also possible to provide a Jakarta Faces implementation that is portable across Jakarta EE servers.

Advanced features of the Jakarta Faces APIs allow Jakarta Faces implementors, as well as application developers, to customize and extend the basic functionality of Jakarta Faces 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 Jakarta Faces-based applications across different Jakarta Faces implementations.

1.3. Introduction to Jakarta Faces APIs

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

1.3.1. package *jakarta.faces*

This package contains top level classes for the Jakarta 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 FactoryFinder.

1.3.2. package *jakarta.faces.application*

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

Please see Application.

1.3.3. package *jakarta.faces.component*

This package contains fundamental APIs for user interface components.

Please see User Interface Component Model.

1.3.4. package *jakarta.faces.component.html*

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

1.3.5. package *jakarta.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 FacesContext.

1.3.6. package *jakarta.faces.convert*

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

Please see Conversion Model.

1.3.7. package *jakarta.faces.el*

This package contains an interface for the Composite Component Attributes ELResolver.

Please see Composite Component Attributes ELResolver.

1.3.8. package *jakarta.faces.flow* and *jakarta.faces.flow.builder*

The runtime API for Faces Flows.

Please see FlowHandler.

1.3.9. package *jakarta.faces.lifecycle*

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

Please see Request Processing Lifecycle.

1.3.10. package *jakarta.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 Event and Listener Model.

1.3.11. package *jakarta.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 Rendering Model.

1.3.12. package *jakarta.faces.validator*

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

Please see Validation Model

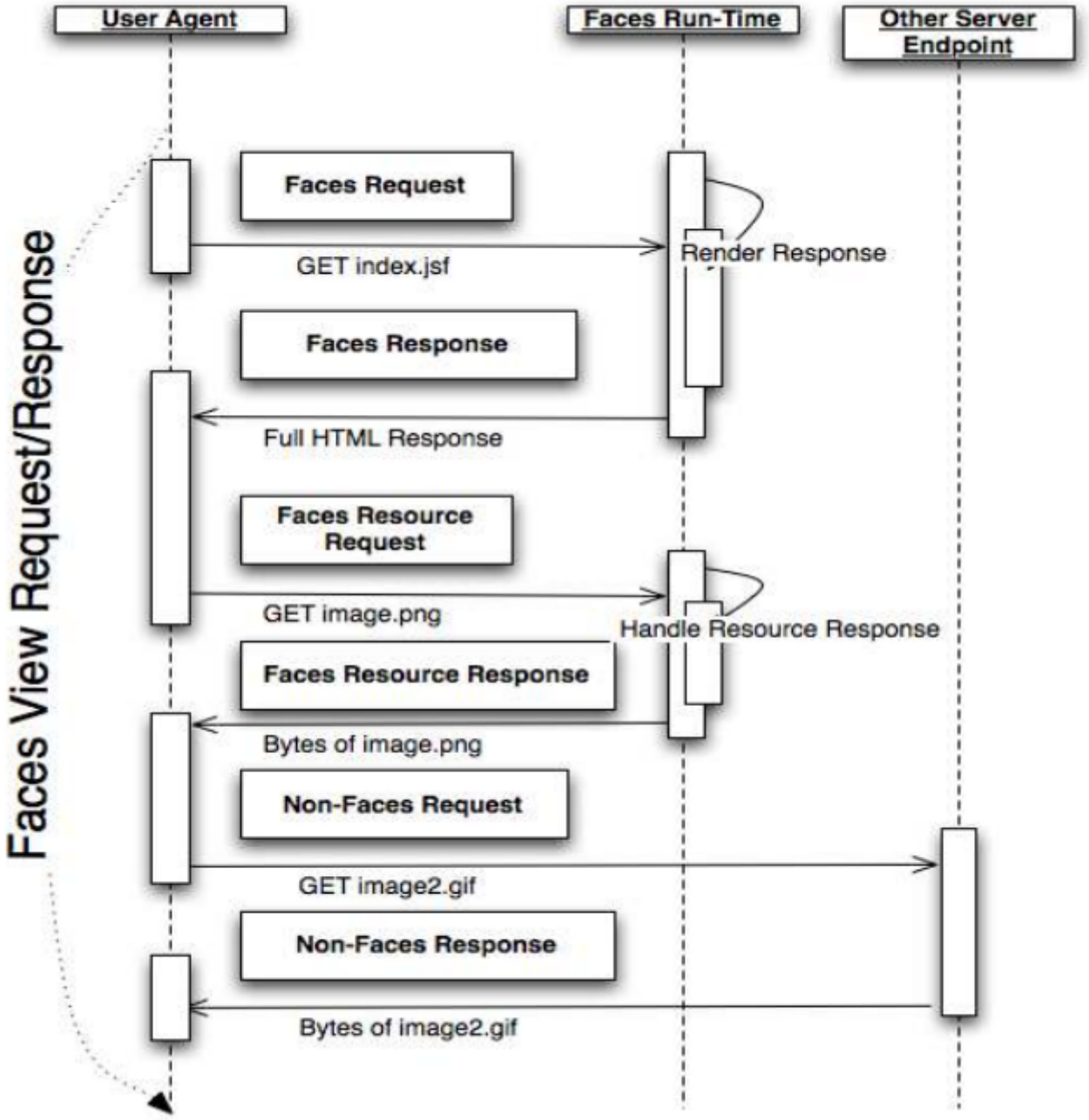
1.3.13. package *jakarta.faces.webapp*

A standard Jakarta Servlet class required for integration of Jakarta Faces into web applications.

Please see Using Jakarta Faces in Web Applications.

Chapter 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 Jakarta Faces 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 Jakarta Servlet-generated or Jakarta Server Pages-rendered response that does not incorporate Jakarta Faces 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 Jakarta Servlet or Jakarta Server Pages 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 Jakarta 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 Jakarta Faces 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.

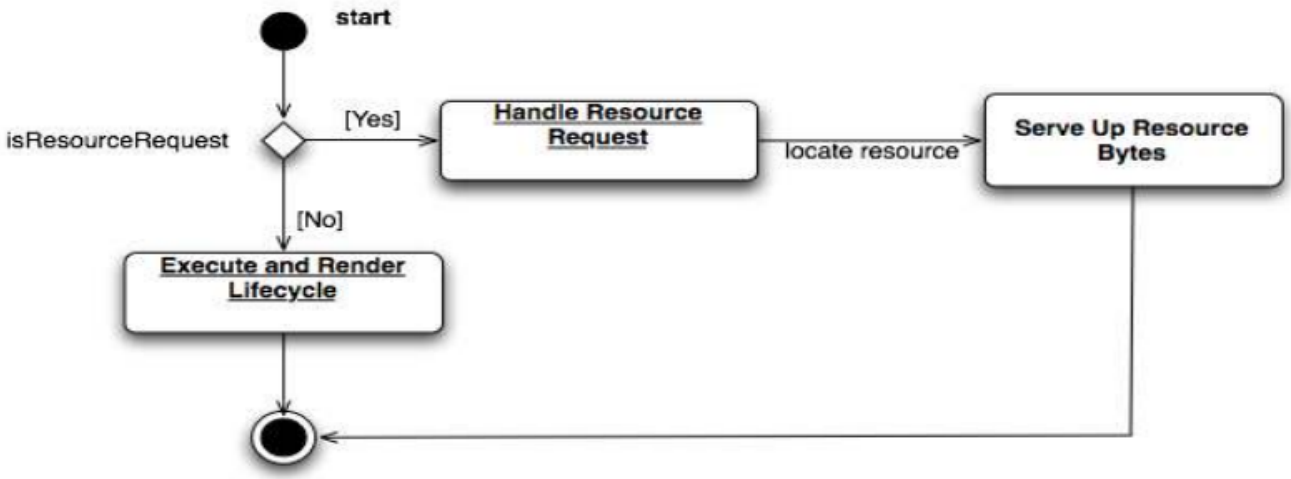
2.1.1. Non-Faces Request Generates Faces Response

An application that is processing a non-Faces request may use Jakarta Faces 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:

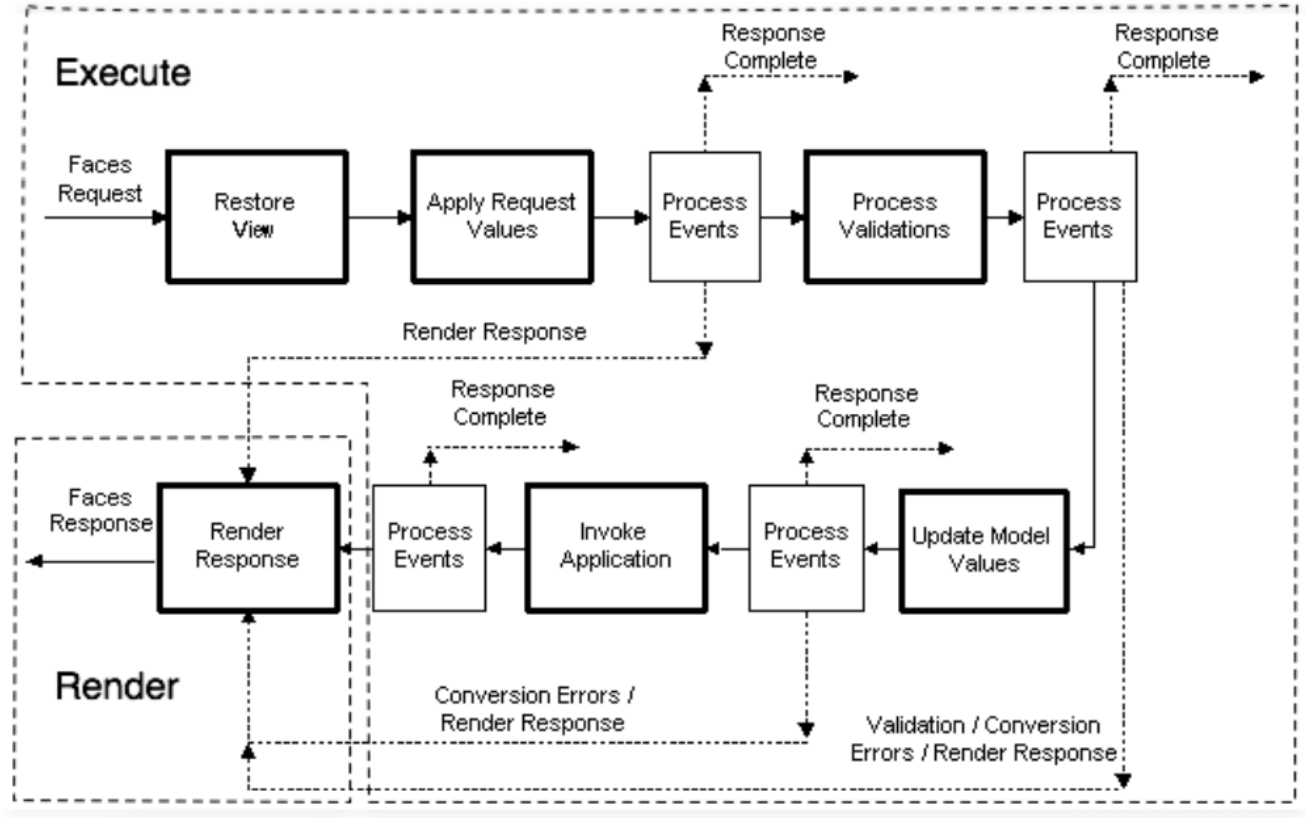
- Acquire Faces object references, as described in Acquire Faces Object References, below.
- Create a new view, as described in Create And Configure A New View, below.
- Store the view into the *FacesContext* by calling the *setViewRoot()* method on the *FacesContext*.

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 Jakarta Faces implementation's controller, as described in Servlet Mapping. Because such a request will be initially handled by the Jakarta Faces 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 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 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 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 Jakarta Faces-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 Jakarta Servlet, or a Jakarta Server Pages page not containing Jakarta Faces 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 Jakarta Servlet or Portlet API to create the response headers and content. It is then necessary to tell the Jakarta Faces 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.

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.

2.2.1. Restore View

The Jakarta Faces 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.
  - 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* according to the following algorithm, or one semantically equivalent to it.
  - Look in the request map for a value under the key *jakarta.servlet.include.path_info*. If found, let it be the *viewId*.
  - Call *getRequestPathInfo()* on the current *ExternalContext*. If this value is non-null, let this be the *viewId* .
  - Look in the request map for a value under the key *jakarta.servlet.include.servlet_path*. If found, let it be the *viewId*.
  - If none of these steps yields a non-*null* viewId, throw a *FacesException* with an appropriate localized message.
- 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 current request is an attempt by the servlet container to display a servlet error page, do not interpret the request as a postback, even if it is indeed a postback.
- If the request is a postback, call *setProcessingEvents(false)* on the current *FacesContext*. Then 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. *jakarta.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*.

- Call *setProcessingEvents(true)* on the current *FacesContext*.
- If the request is not a postback, try to obtain the *ViewDeclarationLanguage* from the *ViewHandler*, for the current *viewId* by calling *ViewHandler.deriveLogicalViewId()* and passing the result to *ViewHandler.getViewDeclarationLanguage()*. If no such instance can be obtained, call *facesContext.renderResponse()*. Otherwise, call *getViewMetadata()* on the *ViewDeclarationLanguage* instance. If the result is non-*null*, call *createMetadataView()* on the *ViewMetadata* instance. Call *ViewMetadata.hasMetadata()*, passing the newly created *viewRoot*. If this method returns false, call *facesContext.renderResponse()*. If it turns out that the previous call to *createViewMetadata()* did not create a *UIViewRoot* instance, call *createView()* on the *ViewHandler*.

View Protection

- Call ViewHandler.getProtectedViewsUnmodifiable() to determine if the view for this viewId is protected. If not, assume the requested view is not protected and take no additional view protection steps. Obtain the value of the value of the request parameter whose name is given by the value of ResponseStateManager.NON_POSTBACK_VIEW_TOKEN_PARAM. If there is no value, throw ProtectedViewException. If the value is present, compare it to the return from ResponseStateManager.getCryptographicallyStrongTokenFromSession(). If the values do not match, throw ProtectedViewException. If the values do match, look for a Referer [sic] request header. If the header is present, use the protected view API to determine if any of the declared protected views match the value of the Referer header. If so, conclude that the previously visited page is also a protected view and it is therefore safe to continue. Otherwise, try to determine if the value of the Referer header corresponds to any of the views in the current web application. If not, throw a ProtectedViewException. If the Origin header is present, additionally perform the same steps as with the Referer header.
- Call *renderResponse()* on the *FacesContext* .

Obtain a reference to the *FlowHandler* from the *Application*. Call its *clientWindowTransition()* method. This ensures that navigation that happened as a result of the renderer for the *jakarta.faces.OutcomeTarget* component-family is correctly handled with respect to flows. For example, this enables <*h:button*> to work correctly with flows.

Using *Application.publishEvent()*, publish a *PostAddToViewEvent* with the created *UIViewRoot* as the event source.

In all cases, the implementation must ensure that the restored tree is traversed and the *PostRestoreStateEvent* is published for every node in the tree.

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".

During the *Apply Request Values* phase, the Jakarta Faces implementation must call the *processDecodes()* method of the *UIViewRoot* of the component tree. This will normally cause the *processDecodes()* method of each component in the tree to be called recursively, as described in the Javadocs for the *UIComponent.processDecodes()* method. The *processDecodes()* method must determine if the current request is a "partial request" by calling *FacesContext.getCurrentInstance().getPartialViewContext().isPartialRequest()* . If *FacesContext.getCurrentInstance().getPartialViewContext().isPartialRequest()* returns *true*, perform the sequence of steps as outlined in Apply Request Values Partial Processing. Details of the decoding process follow.

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 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*, as described in the *UIInput* Javadocs. 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. 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.

2.2.2.1. Apply Request Values Partial Processing

Call FacesContext.getPartialViewContext(). Call PartialViewContext.processPartial() passing the FacesContext, PhaseID.APPLY_REQUEST_VALUES as arguments.

2.2.3. Process Validations

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

During the *Process Validations* phase of the request processing lifecycle, the Jakarta Faces implementation must call the *processValidators()* method of the *UIViewRoot* of the tree. This will normally cause the *processValidators()* method of each component in the tree to be called recursively, as described in the API reference for the *UIComponent.processValidators()* method. The *processValidators()* method must determine if the current request is a "partial request" by calling *FacesContext.getCurrentInstance().getPartialViewContext().isPartialRequest()* . If *FacesContext.getCurrentInstance().getPartialViewContext().isPartialRequest()* returns *true*, perform the sequence of steps as outlined in Partial Validations Partial Processing. 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 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. 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.

2.2.3.1. Partial Validations Partial Processing

Call FacesContext.getPartialViewContext(). Call PartialViewContext.processPartial() passing the FacesContext, PhaseID.PROCESS_VALIDATIONS as arguments.

2.2.4. Update Model Values

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

During the *Update Model Values* phase, the Jakarta Faces implementation must call the

processUpdates() method of the *UIViewRoot* component of the tree. This will normally cause the *processUpdates()* method of each component in the tree to be called recursively, as described in the API reference for the *UIComponent.processUpdates()* method. The *processUpdates()* method must determine if the current request is a "partial request" by calling *FacesContext.getCurrentInstance().getPartialViewContext().isPartialRequest()* . If *FacesContext.getCurrentInstance().getPartialViewContext().isPartialRequest()* returns *true*, perform the sequence of steps as outlined in Update Model Values Partial Processing. 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 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. 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.

2.2.4.1. Update Model Values Partial Processing

Call FacesContext.getPartialViewContext(). Call PartialViewContext.processPartial() passing the FacesContext, PhaseID.UPDATE_MODEL_VALUES as arguments.

2.2.5. Invoke Application

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

2.2.6. Render Response

This phase accomplishes two things:

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

Jakarta Faces supports a range of approaches that Jakarta Faces 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.

Because of the number of possible options, the mechanism for implementing the *Render Response* phase cannot be specified precisely. However, all Jakarta Faces 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 *jakarta.faces.event.PreRenderViewEvent*.
- Jakarta Faces implementations must provide a default *ViewHandler* implementation that is capable of handling views written in the Faces View Declaration Language (VDL).
- 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.
- During the rendering process, additional components may be added to the component tree based on information available to the *ViewHandler* implementation. 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 Facelets) 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*.
- If running on a container that supports Jakarta Servlet 4.0 or later, after any dynamic component manipulations have been completed, any resources that have been added to the UIViewRoot, such as scripts, images, or stylesheets, and any inline images, must be pushed to the client using the Jakarta Servlet Server Push API. All of the pushes must be started before any of the HTML of the response is rendered to the client.
- 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*.

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 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, but before state saving the Jakarta Faces runtime must publish a *jakarta.faces.event.PostRenderViewEvent*. After doing so the Jakarta Faces runtime must save the completed state using the methods of the class *StateManager*. This state information must be made accessible on a subsequent request, so that the *Restore View* can access it. For more on *StateManager*, see State Saving Methods.

2.2.6.1. Render Response Partial Processing

According to *UIViewRoot.encodeChildren()*, *FacesContext.processPartial(PhaseId.RENDER_RESPONSE)*, will be called if and only if the current request is an Ajax request. Take these actions in this case.

On the *ExternalContext* for the request, call *setResponseContentType("text/xml")* and *addResponseHeader("Cache-control", "no-cache")*. Call *startDocument()* on the *PartialResponseWriter*.

Call *writePreamble("<?xml version='1.0' encoding='currentEncoding'?>\n")* on the *PartialResponseWriter*, where *encoding* is the return from the *getCharacterEncoding()* on the *PartialResponseWriter*, or UTF-8 if that method returns *null*.

If *isResetValues()* returns *true*, call *getRenderIds()* and pass the result to *UIViewRoot.resetValues()*.

If *isRenderAll()* returns *true* and the view root is not an instance of *NamingContainer*, call *startUpdate(PartialResponseWriter.RENDER_ALL_MARKER)* on the *PartialResponseWriter*. For each child of the *UIViewRoot*, call *encodeAll()*. Call *endUpdate()* on the *PartialResponseWriter*. Render the state using the algorithm described below in Partial State Rendering, call *endDocument()* on the *PartialResponseWriter* and return. If *isRenderAll()* returns *true* and this *UIViewRoot* is a *NamingContainer*, treat this as a case where *isRenderAll()* returned *false*, but use the *UIViewRoot* itself as the one and only component from which the tree visit must start.

If *isRenderAll()* returns *false*, if there are ids to render, visit the subset of components in the tree to be rendered in similar fashion as for other phases, but for each *UIComponent* in the traversal, call *startUpdate(id)* on the *PartialResponseWriter*, where *id* is the client id of the component. Call *encodeAll()* on the component, and then *endUpdate()* on the *PartialResponseWriter*. If there are no ids to render, this step is un-necessary. After the subset of components (if any) have been rendered, Render the state using the algorithm described below in Partial State Rendering, call *endDocument()* on the *PartialResponseWriter* and return.

Partial State Rendering

This section describes the requirements for rendering the <*update*> elements pertaining to view state and window id in the case of partial response rendering.

If the view root is marked transient, take no action and return.

Obtain a unique id for the view state, as described in the JavaDocs for the constant field *ResponseStateManager.VIEW_STATE_PARAM*. Pass this id to a call to *startUpdate()* on the *PartialResponseWriter*. Obtain the view state to render by calling *getViewState()* on the application's *StateManager*. Write the state by calling *write()* on the *PartialResponseWriter*, passing the state as the argument. Call *endUpdate()* on the *PartialResponseWriter*.

If *getClientWindow()* on the *ExternalContext*, returns non-*null*, obtain an id for the <*update*> element for the window id as described in the JavaDocs for the constant *ResponseStateManager.WINDOW_ID_PARAM*. Pass this id to a call to *startUpdate()* on the *PartialResponseWriter*. Call *write()* on that same writer, passing the result of calling *getId()* on the *ClientWindow*. Call *endUpdate()* on the *PartialResponseWriter*.

2.3. Common Event Processing

For a complete description of the event processing model for Jakarta Faces components, see Event and Listener Model.

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

For each queued event, the *broadcast()* method of the source *UIComponent* must be called to broadcast the event to all event listeners who have registered an interest, on this source component

for events of the specified type, after which the event is removed from the event queue. See the API reference for the *UIComponent.broadcast()* method for the detailed functional requirements.

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

2.4. Common Application Activities

The following subsections describe common activities that may be undertaken by an application that is using Jakarta Faces to process an incoming request and/or create an outgoing response. Their use is described in 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 Jakarta Faces implementation, as described below.

2.4.1.1. Acquire and Configure Lifecycle Reference

As described in Lifecycle, the Jakarta Faces implementation must provide an instance of *jakarta.faces.lifecycle.Lifecycle* that may be utilized to manage the remainder of the request processing lifecycle. An application may acquire a reference to this instance in a portable manner, as follows:

```
LifecycleFactory lFactory = (LifecycleFactory)
    FactoryFinder.getFactory(FactoryFinder.LIFECYCLE_FACTORY);
Lifescycle 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 Jakarta Faces implementation you are using. However, using a non-default lifecycle identifier will generally not be portable to any other Jakarta Faces implementation.

2.4.1.2. Acquire and Configure FacesContext Reference

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

```
FacesContextFactory fcFactory = (FacesContextFactory)
```

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

where the *context*, *request*, and *response* objects represent the corresponding instances for the application environment. For example, in a Jakarta 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 *jakarta.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. The *ViewHandler* provides a factory method that may be utilized to construct new component trees, as follows:

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

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

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

2.4.2.2. Configure the Desired RenderKit

The *UIViewRoot* instance provided by the *ViewHandler*, as described in the previous subsection, must automatically be configured to utilize the default *jakarta.faces.render.RenderKit* implementation provided by the Jakarta Faces implementation, as described in RenderKit. This *RenderKit* must support the standard components and *Renderer*s described later in this specification, to maximize the portability of your application.

However, a different *RenderKit* instance provided by your Jakarta Faces 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.

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

2.4.2.3. Configure The View's Components

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

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

2.4.2.4. Store the new View in the FacesContext

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

2.5. Concepts that impact several lifecycle phases

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

2.5.1. Value Handling

At a fundamental level, Jakarta 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 Jakarta Servlet/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 Jakarta Faces 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)

Jakarta Faces is fully internationalized. The I18N capability in Jakarta Faces builds on the I18N concepts offered in the Jakarta Servlet and Jakarta Tags 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*

Jakarta Faces 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 Jakarta Faces what locales the application supports in the applications' *WEB-INF/faces-config.xml* file. For example:

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

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

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

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

To facilitate BCP 47 support, the Locale parsing mentioned above is done only if the JDK Locale.languageForTag method does not return a Locale with a language in it. The additional format of the Locale string is as specified by that method.

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 Jakarta Servlet or Portlet request) to parse request parameter correctly.

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

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

On a subsequent postback, before any of the ExternalContext methods for accessing request parameters are invoked, the ViewHandler must examine the Content-Type header to read the

charset attribute and use its value to set it as the request encoding for the underlying request object. If the Content-Type header doesn't contain a charset attribute, the encoding previously stored in the session (if and only if a session already exists), must be used to set the encoding for the underlying request object. If no character encoding is found, the request encoding must be left unmodified.

The above algorithm allows an application to use the mechanisms of the underlying technologies to adjust both the request and response encoding in an application-specific manner. Note, though, that the character encoding rules prior to Jakarta Servlet 2.4 are imprecise and special care must be taken for portability between containers.

2.5.2.3. Localized Text

Since most Jakarta Faces 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, Jakarta Faces 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 Jakarta Faces handles localized error and informational messages that occur as a result of conversion, validation, or other application actions during the request processing lifecycle. The Jakarta Faces class *jakarta.faces.application.FacesMessage* is provided to encapsulate summary, detail, and severity information for a message. A Jakarta Faces implementation must provide a *jakarta.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:

- jakarta.faces.component.UIInput.CONVERSION={0}: Conversion error occurred
- jakarta.faces.component.UIInput.REQUIRED={0}: Validation Error: Value is required
- jakarta.faces.component.UIInput.UPDATE= {0}: An error occurred when processing your submitted information
- jakarta.faces.component.UISelectOne.INVALID={0}: Validation Error: Value is not valid
- jakarta.faces.component.UISelectMany.INVALID={0}: Validation Error: Value is not valid
- jakarta.faces.converter.BigDecimalConverter.DECIMAL={2}: "{0}" must be a signed decimal number.
- jakarta.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}

- jakarta.faces.converter.BigIntegerConverter.BIGINTEGER={2}: "{0}" must be a number consisting of one or more digits.
- jakarta.faces.converter.BigIntegerConverter.BIGINTEGER_detail={2}: "{0}" must be a number consisting of one or more digits. Example: {1}
- jakarta.faces.converter.BooleanConverter.BOOLEAN={1}: "{0}" must be 'true' or 'false'.
- jakarta.faces.converter.BooleanConverter.BOOLEAN_detail={1}: "{0}" must be 'true' or 'false'. Any value other than 'true' will evaluate to 'false'.
- jakarta.faces.converter.ByteConverter.BYTE={2}: "{0}" must be a number between -128 and 127.
- jakarta.faces.converter.ByteConverter.BYTE_detail={2}: "{0}" must be a number between -128 and 127. Example: {1}
- jakarta.faces.converter.CharacterConverter.CHARACTER={1}: "{0}" must be a valid character.
- jakarta.faces.converter.CharacterConverter.CHARACTER_detail={1}: "{0}" must be a valid ASCII character.
- jakarta.faces.converter.DateTimeConverter.DATE={2}: "{0}" could not be understood as a date.
- jakarta.faces.converter.DateTimeConverter.DATE_detail={2}: "{0}" could not be understood as a date. Example: {1}
- jakarta.faces.converter.DateTimeConverter.TIME={2}: "{0}" could not be understood as a time.
- jakarta.faces.converter.DateTimeConverter.TIME_detail={2}: "{0}" could not be understood as a time. Example: {1}
- jakarta.faces.converter.DateTimeConverter.DATETIME={2}: "{0}" could not be understood as a date and time.
- jakarta.faces.converter.DateTimeConverter.DATETIME_detail={2}: '{0}" could not be understood as a date and time. Example: {1}
- jakarta.faces.converter.DateTimeConverter.PATTERN_TYPE={1}: A 'pattern' or 'type' attribute must be specified to convert the value "{0}".
- jakarta.faces.converter.DoubleConverter.DOUBLE={2}: "{0}" must be a number consisting of one or more digits.
- jakarta.faces.converter.DoubleConverter.DOUBLE_detail={2}: "{0}" must be a number between 4.9E-324 and 1.7976931348623157E308 Example: {1}
- jakarta.faces.converter.EnumConverter.ENUM={2}: "{0}" must be convertible to an enum.
- jakarta.faces.converter.EnumConverter.ENUM_detail={2}: "{0}" must be convertible to an enum from the enum that contains the constant "{1}".
- jakarta.faces.converter.EnumConverter.ENUM_NO_CLASS={1}: "{0}" must be convertible to an enum from the enum, but no enum class provided.
- jakarta.faces.converter.EnumConverter.ENUM_NO_CLASS_detail={1}: "{0}" must be convertible to an enum from the enum, but no enum class provided.
- jakarta.faces.converter.FloatConverter.FLOAT={2}: "{0}" must be a number consisting of one or

more digits.

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

- jakarta.faces.validator.DoubleRangeValidator.TYPE={0}: Validation Error: Value is not of the correct type
- jakarta.faces.validator.LengthValidator.MAXIMUM={1}: Validation Error: Length is greater than allowable maximum of "{0}"
- jakarta.faces.validator.LengthValidator.MINIMUM={1}: Validation Error: Length is less than allowable minimum of "{0}"
- jakarta.faces.validator.LongRangeValidator.MAXIMUM={1}: Validation Error: Value is greater than allowable maximum of "{0}"
- jakarta.faces.validator.LongRangeValidator.MINIMUM={1}: Validation Error Value is less than allowable minimum of "{0}"
- jakarta.faces.validator.LongRangeValidator.NOT_IN_RANGE={2}: Validation Error: Specified attribute is not between the expected values of {0} and {1}.
- jakarta.faces.validator.LongRangeValidator.TYPE={0}: Validation Error: Value is not of the correct type

A Jakarta Faces 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, Jakarta Faces adopts the policy that *ResourceBundle* key for the message looks up the message summary. The detail is stored under the same key as the summary, with *detail* appended. These *ResourceBundle* keys must be used to look up the necessary values to create a localized *FacesMessage* instance. Note that the value of the summary and detail keys in the *ResourceBundle* may contain parameter substitution tokens, which must be substituted with the appropriate values using *java.text.MessageFormat*. 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 *Generic Attributes* and *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*>.

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 *jakarta.faces.Messages* resource bundle.
- In either case, if a message is found, use the above conventions to create a *FacesMessage* instance.

2.5.3. State Management

Jakarta 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 *jakarta.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 The jakarta.faces.component package, all Jakarta Faces components implement the *StateHolder* interface. As a consequence the standard components provide implementations of *PartialStateHolder* to suit their needs. A custom component that extends *UIComponent* directly, and does not extend any of the standard components, must implement *PartialStateHolder* (or its older super-interface, *StateHolder*), manually. The helper class *StateHelper* exists to simplify this process for the custom component author. Please see *PartialStateHolder* or 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* or *PartialStateHolder* correctly. Notably, calls to *saveState()* must not alter the state in any way. The subclass is responsible for saving and restoring the state of the superclass. Consider this example. My custom component represents a “slider” ui widget. As such, it needs to keep track of the maximum value, minimum value, and current values as part of its state.

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

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

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

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

2.5.3.2. State Management Considerations for the Jakarta Faces 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 Jakarta Faces 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

- *StateHelper* the helper class that defines a *Map* -like contract that makes it easier for components to implement *PartialStateHolder*.
- *ViewHandler* the entry point to the state management system. Uses a helper class, *StateManager*, to do the actual work.
- *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. 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 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 *jakarta.faces.resource.Script* and *jakarta.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 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 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. The normative specification for this feature is spread out across several places, including the View 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 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, 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

Jakarta Faces has a bookmarking capability with the use of two Standard HTML RenderKit additions.

Provided is a 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. Jakarta Bean Validation

Jakarta Faces supports Jakarta Bean Validation. A Jakarta Faces implementation must support Jakarta Bean Validation if the environment in which the Jakarta Faces runtime is included requires Jakarta Bean Validation. Currently the only such environment is when Jakarta Faces is included in a Jakarta EE runtime.

A detailed description of the usage of Jakarta Bean Validation with Jakarta Faces 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, that has several text fields inside of a form. This application is running in a Jakarta Faces runtime in an environment that does require Jakarta Bean Validation, and therefore this feature is available. Assume that every text field is bound to a managed bean property that has at least one Jakarta Bean Validation constraint annotation attached to it.

During the render response phase that always precedes a postback, due to the specification requirements in Validation Registration, every *UIInput* in this application has an instance of *Validator* with id *jakarta.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 *jakarta.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.

See also Bean Validation Integration.

2.5.8. Ajax

Jakarta Faces supports Ajax. The specification contains 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 compatibility. The specification provides two ways of adding Ajax to Jakarta Faces web applications. Page authors may use the JavaScript library directly in their pages by attaching the Ajax request call to a Jakarta Faces component via a JavaScript event (such as onclick). They may also take a more declarative approach and use a core Facelets tag (<f:ajax/>) that they can nest within Jakarta Faces 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 Jakarta Faces Ajax frameworks work with the standard Jakarta Faces 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 Jakarta Faces view can be processed (known as partial processing) and selected components can be rendered to the client (known as partial rendering).

2.5.9. Component Behaviors

The Jakarta Faces specification contains a 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 Jakarta Faces 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 provided by the Jakarta Faces 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 Jakarta Faces 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 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 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 Faces Request Generates Faces Response, the Jakarta Faces run-time must determine if the current Faces Request is a *Faces Resource Request* or a *View Request*. This must be accomplished by calling *Application.getResourceHandler().isResourceRequest()*. 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

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*.

2.6.1.2. Packaging Resources into the Classpath

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*.

2.6.1.3. Resource Identifiers

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

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

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

- The set of characters that are valid for use in the *localePrefix*, *libraryName*, *libraryVerison*, *resourceName* and *resourceVersion* segments of the resource identifier is specififed as XML NameChar excluding the path separator and ':' characters. The specification for XML NameChar may be seen at *https://www.w3.org/TR/REC-xml/#NT-NameChar*.
- A further restriction applies to *libraryName*. A *libraryName* must not be an underscore separated sequence of non-negative integers or a locale string. More rigorously, a *libraryName* must not match either of the following regular expressions:

```
[0-9]+(_[0-9]+)*
[A-Za-z]{2}(_[A-Za-z]{2}(_[A-Za-z]+)*)?
```

- 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.

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

localePrefx	libraryNam e	library Version [optional]	resourceNa me	resource Version [optional]	Description	actual resourceIde ntifier
__	__	__	*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 Jakarta Faces runtime.

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 Packaging Resources into the Web Application Root, Packaging Resources into the Classpath, and 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 the 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 resource = null;
    var resourceId = null;
    for (var contract : getLibraryContracts()) {
        resourceId = deriveResourceIdConsideringResourceLoaders(contract,
                resourceName, libraryName)
        if (null != resourceId) {
            resource = create the resource using the resourceId;
            return resource;
        }
    }

    // try without a contract
    resourceId = deriveResourceIdConsideringResourceLoaders(null,
            resourceName, libraryName)
    if (null != resourceId) {
        resource = create the resource using the resourceId;
    }
    return resource;
}

function deriveResourceIdConsideringResourceLoaders(contract,
        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 resourceId = deriveResourceIdConsideringLocalePrefix(contract,
            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 = deriveResourceIdConsideringLocalePrefix(contract,
                prefix, resourceLoader, resourceName, libraryName);
    }
    return resourceId;
}

function deriveResourceIdConsideringLocalePrefix(contract, prefix,
        resourceLoader, resourceName, libraryName) {
    var localePrefix = getLocalePrefix();
    var result = deriveResourceId(contract, prefix, resourceLoader,
            resourceName, libraryName, localePrefix);
    // If the application has been configured to have a localePrefix,
    // and the resource is not found, try to find it again,
    // without the localePrefix.
    if (null == result && null != localePrefix) {
        result = deriveResourceId(contract, prefix, resourceLoader,
                resourceName, libraryName, null);
    }
    return result;
}

function deriveResourceId(contract, prefix, resourceLoader,
        resourceName, libraryName, localePrefix) {
    var resourceVersion = null;
    var libraryVersion = null;
    var resourceId;
    if (null != localePrefix) {
        prefix = localePrefix + '/' + prefix;
    }
    if (null != contract) {
        prefix = contract + '/' + prefix;
    }

    if (null != libraryName) {
        // actual argument is
        // resourcesInContractInJar/resources/resourcesInContractInJar
        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 =
```

```
            // If there is a viewRoot on the current facesContext,
            // use its locale.
            // Otherwise, use the locale of the application's ViewHandler
        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;
}
```

2.6.2. Rendering Resources

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

```
<h:graphicImage library="common" name="images/planets.png" />
<h:graphicImage value="#{resource['common:images/planets.png']}" />
```

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 view declaration language as follows:

```
<!DOCTYPE html>
<html xmlns:h="jakarta.faces.html">
  <h:head>
    <title>Example View</title>
  </h:head>
  <h:body>
    <h:form>
      <h:outputScript library="jakarta.faces" name="faces.js" target="head" />
    </h:form>
  </h:body>
</html>
```

The example tag *<h:outputScript>* which extends from *UIOutput* refers to the example renderer, *ScriptRenderer*, implementing *ComponentSystemEventListener*, that listens for *PostAddToViewEvent* event types:

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

Refer to 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 Methods.

The <h:head> and <h:body> tags refer to the renderers HeadRenderer and BodyRenderer respectively. They are described in the Standard HTML Renderkit documentation referred to in Standard HTML RenderKit Implementation. During the rendering phase, the encode methods for these renderers render the HTML “head” and “body” elements respectively. Then they render all component resources under the facet child (named by target) under the UIViewRoot using the UIViewRoot component resource methods as described in 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 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 Component Tree Manipulation for detailed information.

2.7. Resource Library Contracts

A resource library contract is a resource library, as specified in the preceding section, except that instead of residing in the *resources* directory of the web-app root, or in the *META-INF/resources* JAR entry name in a JAR file, it resides in the *contracts* directory of the web-app root, or in the *META-INF/contracts* JAR entry name in a JAR file. When packaged in a JAR file, there is one additional packaging requirement: each resource library contract in the JAR must have a marker file. The name of the file is given by the value of the symbolic constant *jakarta.faces.application.ResourceHandler.RESOURCE_CONTRACT_XML*. This may be a zero length file, though future versions of the specification may use the file to declare the usage contract. The requirement to have a marker file enables implementations to optimize for faster deployment while still enabling automatic discovery of the available contracts.

Following is a listing of the entries in a JAR file containing two resource library contracts.

```
META-INF/contracts/
                siteLayout/
                          jakarta.faces.contract.xml
```

```
                        topNav_template.xhtml
                        leftNav_foo.xhtml
                        styles.css
                        script.js
                        background.png
                subSiteLayout/
                            jakarta.faces.contract.xml
                            sub_template.xhtml
```

All of the other packaging, encoding and decoding requirements are the same as for resource libraries.

See Resource Library Contracts Background for a non-normative overview of the feature, including a brief usage example.

Chapter 3. User Interface Component Model

A Jakarta Faces *user interface component* is the basic building block for creating a Jakarta Faces 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*.

Jakarta Faces 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 Jakarta Faces-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 *jakarta.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 Facelets. A standard set of components (described in 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 Jakarta Faces implementors can also create additional *UIComponent* implementations for use within a particular application. To assist such developers, a convenience subclass, *jakarta.faces.component.UIComponentBase*, is provided as part of Jakarta Faces. 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 Jakarta Faces user interface components.

3.1.1. Component Identifiers

```
public String getId();
public void setId(String componentId);
```

Every component may be named by a *component identifier* 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 ('_').

To minimize the size of responses generated by Jakarta 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 Object Factories for more on *component-type*.

Component types starting with "jakarta.faces." are reserved for use by the Jakarta Faces 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 "jakarta.faces." are reserved for use by the Jakarta Faces 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 ValueExpression) 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. 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. For the standard component classes defined by this specification, all attributes, and all properties other than *id*, *parent*, *action*, *listener*, *actionListener*, *valueChangeListener*, and *validator* are value expression enabled. The *action*, *listener*, *actionListener*, *valueChangeListener*, and *validator* attributes are method expression enabled.

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 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:

- When a component instance is first created (typically by virtue of being referenced by a tag in a page), the Jakarta Faces 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).

- 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.

Component bindings are often used in conjunction with JavaBeans that are dynamically instantiated via the Managed Bean Creation facility. If application developers place managed beans that are pointed at by component binding expressions in any scope other than request scope, the system cannot behave correctly. This is because placing it in a scope wider than request scope would require thread safety, since UIComponent instances depend on running inside of a single thread. There are also potentially negative impacts on memory management when placing a component binding in "session" or "view" scopes.

3.1.6. Client Identifiers

Client identifiers are used by Jakarta Faces 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 Jakarta Faces-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. 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. 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 *jakarta.faces.component.visit* for the normative specification.

3.1.9. Facet Management

Jakarta 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 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, Jakarta 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 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 *jakarta.faces.component.behavior.BehaviorHolder* interface. *UIComponentBase* does not implement the *jakarta.faces.component.behavior.BehaviorHolder* interface, but it provides the default implementations to simplify subclass implemenations. Refer to 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()*. For 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 Jakarta Bean component properties with getter and setter methods (see 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 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 *jakarta.faces* are reserved for use by the Jakarta Faces specification. Names that begin with *jakarta* are reserved for definition through the Eclipse Foundation Process. Implementations are not allowed to define names that begin with *jakarta*.

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.
- 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.

3.1.11.1. Special Attributes

UIComponent Constants

```
public static final String BEANINFO_KEY =
    "jakarta.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 =
    "jakarta.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 =
    "jakarta.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 =
    "jakarta.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 jakarta.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 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 Jakarta Faces 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 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 Jakarta Faces 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 Common Event Processing) at the end of several request processing lifecycle phases. For more information about the event and listener model, see 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. The default implementation in UIComponentBase *encodeChildren()* must iterate over its children and call *encodeAll()* for each child component. *encodeBegin()* must publish a *PreRenderComponentEvent*.

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 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 Application Actions) via the default *ActionListener* facility (see ActionListener Property).

The *ActionSource* interface also provides a JavaBeans *actionExpression* property analogous to the *action* property. This allows the *ActionSource* concept to leverage the Jakarta Expression Language API.

3.2.1.1. Properties

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

Name	Access	Type	Description
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*.
actionExpression	RW	*jakarta.el.MethodExpression*	A *MethodExpression* (see MethodExpression) that must (if non-*null*) point at an action method (see 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 Invoke Application.

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 Event and Listener Model for more details on the event and listener model provided by Jakarta Faces.

```
public void addActionListener(ActionListener listener);
public void removeActionListener(ActionListener listener);
```

In addition to manually registered listeners, the Jakarta Faces 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 Invoke Application for more information.

3.2.2. NamingContainer

NamingContainer is a marker interface. Components that implement *NamingContainer* have the property that, for all of their children that have non-*null* component identifiers, all of those identifiers are unique. This property is enforced by the *renderView()* method on *ViewHandler*. Since this is just a marker interface, there are no properties, methods, or events. Among the standard components, *UIForm* and *UIData* implement *NamingContainer*. See UIForm and *Section* Methods "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 as default value to separate components of client identifiers, as well as the components of search expressions used by the *findComponent()* method see (Component Tree Navigation). The value of this constant must be a colon character (":"), which is according to the HTML standard among the allowable values for the *id* and *name* attribute.

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 *styleClass* attribute to select CSS styles by class rather than by identifier, or by wrapping the section in a plain HTML <*div*> or <*span*> element containing the desired client identifier, or by escaping the colon character with a backslash in the CSS selector.

This separator character can be overridden to another character by the context initialization parameter named *jakarta.faces.SEPARATOR_CHAR*, for example an underscore character ("_"), but the page authors must then ensure to not use this exact character in the component identifiers themselves.

Component authors should never use the final character constant *SEPARATOR_CHAR* directly, but they should instead use *UINamingContainer.getSeparatorChar(FacesContext)* method to obtain the desired separator character, which can be either the overridden character or the default character of (":").

3.2.3. StateHolder

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

3.2.3.1. Properties

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

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

3.2.3.2. Methods

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

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

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

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

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

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

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

Since all of the standard user interface components listed in Standard User Interface Components”

extend from *UIComponent*, they all implement the *StateHolder* interface. In addition, the standard *Converter* and *Validator* classes that require state to be saved and restored also implement *StateHolder*.

3.2.3.3. Events

StateHolder does not originate any standard events.

3.2.4. PartialStateHolder

PartialStateHolder extends *StateHolder* and adds a usage contract for components that wish to take part in the partial state saving mechanism introduced in version 2.0. Implementations of this interface should use the *jakarta.faces.component.StateHelper* instance returned from *UIComponent.getStateHelper()* to store stateful component information that otherwise would have been stored as instance variables on the class implementing *PartialStateHolder*.

3.2.4.1. Properties

PartialStateHolder adds no properties to the *StateHolder* contract

3.2.4.2. Methods

The following methods support the partial state saving feature:

```
void clearInitialState();
boolean initialStateMarked();
void markInitialState();
```

These methods allow the state saving feature to determine if the component is in its initial state or not, and to set the flag indicating this condition of existence. The Javadocs for these methods specify the conditions under which these methods are invoked.

3.2.4.3. Events

PartialStateHolder does not originate any standard events.

3.2.5. 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.

3.2.5.1. 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 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.

3.2.5.2. Methods

ValueHolder adds no methods.

3.2.5.3. Events

ValueHolder does not originate any standard events.

3.2.6. EditableValueHolder

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

3.2.6.1. 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.
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).

3.2.6.2. 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.

3.2.6.3. 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 Event and Listener Model for more details on the event and listener model provided by Jakarta Faces.

```
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.

3.2.7. 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.

3.2.7.1. Properties

This interface contains no JavaBeans properties

3.2.7.2. Methods

The following method gives the Jakarta Faces 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()*.

3.2.7.3. 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 Event and Listener Model for more details on the event and listener model provided by Jakarta Faces.

3.2.8. ClientBehaviorHolder

Components must implement the *ClientBehaviorHolder* interface to add the ability for attaching

ClientBehavior instances (see 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. The concrete HTML component classes that come with Jakarta Faces 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. The Collection must be non null and unmodifiable.

```
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.

Each key value in this Map must be one of the event names in the Collection returned from getEventNames().

```
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 Jakarta 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, Jakarta 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 Object Factories) for your application.

3.3.2. Converter

Jakarta Faces provides the *jakarta.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 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()*. Otherwise, the Jakarta Faces implementation will automatically create new instances based on the data type being converted, if such Converter classes have been registered. In either case, Converter implementations need not be threadsafe, because they will be used only in the context of a single request processing thread.

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

```
public Object getAsObject(FacesContext context,
```

```
    UIComponent component, String value) throws ConverterException;
```

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

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

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.

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.

3.3.3. Standard Converter Implementations

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

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

- *jakarta.faces.BigDecimal* —An instance of *jakarta.faces.convert.BigDecimalConverter* (or a subclass of this class).
- *jakarta.faces.BigInteger* —An instance of *jakarta.faces.convert.BigIntegerConverter* (or a subclass of this class).
- *jakarta.faces.Boolean* —An instance of *jakarta.faces.convert.BooleanConverter* (or a subclass of this class).
- *jakarta.faces.Byte* —An instance of *jakarta.faces.convert.ByteConverter* (or a subclass of this class).
- *jakarta.faces.Character* —An instance of *jakarta.faces.convert.CharacterConverter* (or a subclass of this class).
- *jakarta.faces.DateTime* —An instance of *jakarta.faces.convert.DateTimeConverter* (or a subclass of this class).
- *jakarta.faces.Double* —An instance of *jakarta.faces.convert.DoubleConverter* (or a subclass of this class).
- *jakarta.faces.Float* —An instance of *jakarta.faces.convert.FloatConverter* (or a subclass of this

class).

- *jakarta.faces.Integer* —An instance of *jakarta.faces.convert.IntegerConverter* (or a subclass of this class).
- *jakarta.faces.Long* —An instance of *jakarta.faces.convert.LongConverter* (or a subclass of this class).
- *jakarta.faces.Number* —An instance of *jakarta.faces.convert.NumberConverter* (or a subclass of this class).
- *jakarta.faces.Short* —An instance of *jakarta.faces.convert.ShortConverter* (or a subclass of this class).
- *jakarta.faces.UUID* —An instance of *jakarta.faces.convert.UUIDConverter* (or a subclass of this class).

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

A Jakarta Faces 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 *jakarta.faces.convert.BigDecimalConverter* (or a subclass of this class).
- *java.math.BigInteger*, and *java.math.BigInteger.TYPE* —An instance of *jakarta.faces.convert.BigIntegerConverter* (or a subclass of this class).
- *java.lang.Boolean*, and *java.lang.Boolean.TYPE* —An instance of *jakarta.faces.convert.BooleanConverter* (or a subclass of this class).
- *java.lang.Byte*, and *java.lang.Byte.TYPE* — An instance of *jakarta.faces.convert.ByteConverter* (or a subclass of this class).
- *java.lang.Character*, and *java.lang.Character.TYPE* —An instance of *jakarta.faces.convert.CharacterConverter* (or a subclass of this class).
- *java.lang.Double*, and *java.lang.Double.TYPE* —An instance of *jakarta.faces.convert.DoubleConverter* (or a subclass of this class).
- *java.lang.Float*, and *java.lang.Float.TYPE* —An instance of *jakarta.faces.convert.FloatConverter* (or a subclass of this class).
- *java.lang.Integer*, and *java.lang.Integer.TYPE* —An instance of *jakarta.faces.convert.IntegerConverter* (or a subclass of this class).
- *java.lang.Long*, and *java.lang.Long.TYPE* — An instance of *jakarta.faces.convert.LongConverter* (or a subclass of this class).
- *java.lang.Short*, and *java.lang.Short.TYPE* —An instance of *jakarta.faces.convert.ShortConverter* (or a subclass of this class).
- *java.lang.Enum*, and *java.lang.Enum.TYPE* — An instance of *jakarta.faces.convert.EnumConverter* (or a subclass of this class).
- *java.lang.UUID*, and *java.lang.UUID.TYPE* — An instance of *jakarta.faces.convert.UUIDConverter* (or a subclass of this class).

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

A 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.

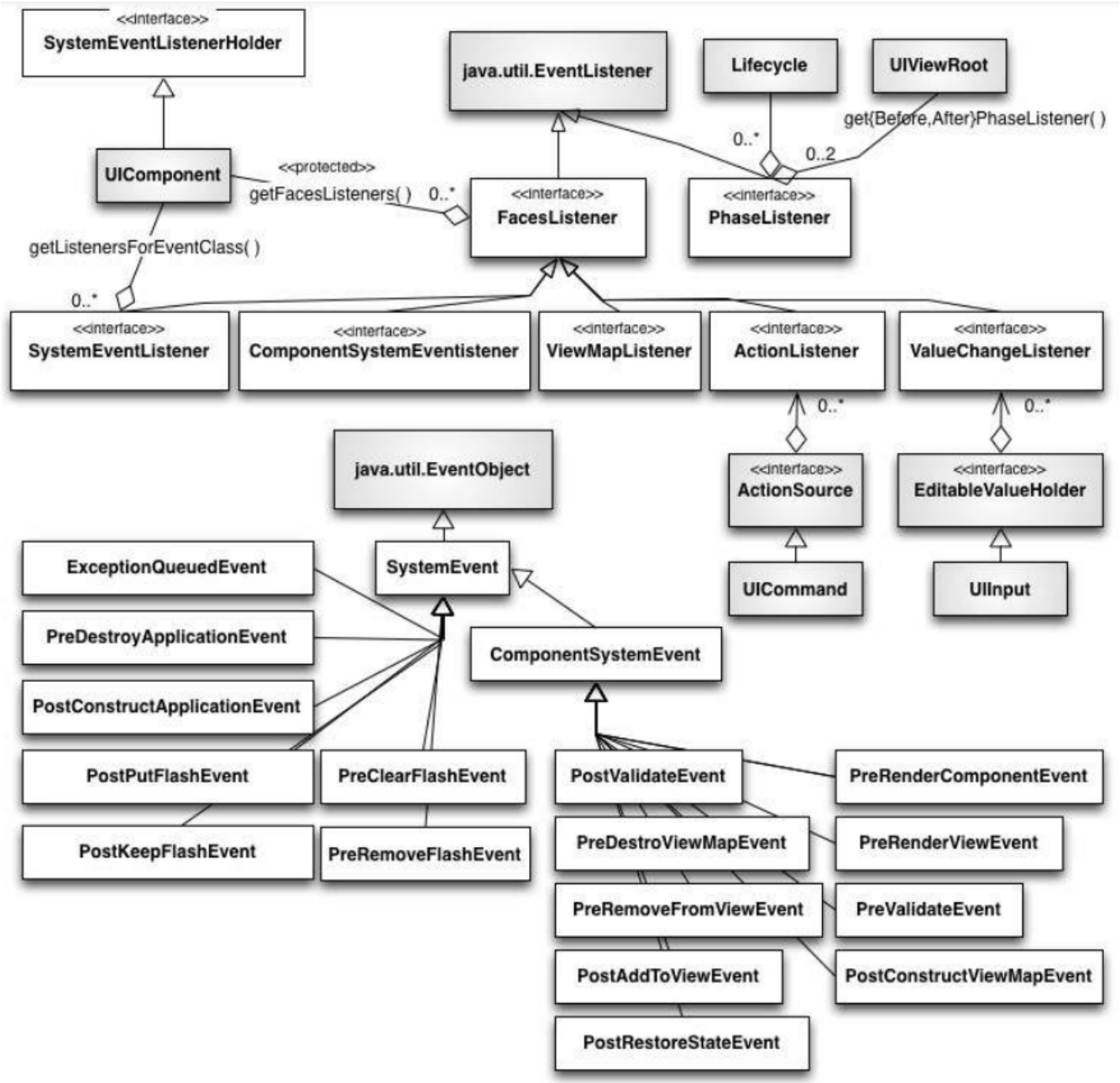
3.4. Event and Listener Model

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

3.4.1. Overview

Jakarta Faces 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 Jakarta Faces implementation will broadcast all of the events that have been queued to interested listeners. Jakarta Faces 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 Jakarta Faces 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 *jakarta.faces.event* package.



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 Jakarta Faces from the beginning.

3.4.2.1. Event Classes

All events that are broadcast by Jakarta Faces user interface components must extend the *jakarta.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 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 Jakarta Faces 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.

Jakarta Faces 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 *jakarta.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, Jakarta Faces 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 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 *jakarta.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. In order to be recognized by development tools, these listener methods must follow the naming patterns defined in the *JavaBeans Specification*. For example, for a component that emits *FooEvent* events, to be received by listeners that implement the *FooListener* interface, the method signatures (on the component class) must be:

```
public void addFooListener(FooListener listener);
public FooListener[] getFooListeners();
public void removeFooListener(FooListener listener);
```

The application (or other components) may register listener instances at any time, by calling the appropriate add method. The set of listeners associated with a component is part of the state information that Jakarta Faces saves and restores. Therefore, listener implementation classes must have a public zero-argument constructor, and may implement *StateHolder* (see 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 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 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 Jakarta Faces implementation that no further broadcast of this event should take place.
- Call *renderResponse()* on the *FacesContext* instance for the current request. This tells the Jakarta Faces 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 Jakarta Faces 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 represent specific points in time for a Jakarta Faces application. *PhaseEvent*s also represent specific points in time in a Jakarta Faces application, but the granularity they offer is not as precise as System Events. For more on *PhaseEvent*s, please see 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 Event Classes.

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 Event Classes.

The specification defines the following *SystemEvent* subclasses, all in package *jakarta.faces.event*.

- *ExceptionQueuedEvent* indicates a non-expected *Exception* has been thrown. Please see ExceptionHandler for the normative specification.
- *PostConstructApplicationEvent* must be published immediately after application startup. Please see Application Startup Behavior for the normative specification.
- *PreDestroyApplicationEvent* must be published as immediately before application shutdown. Please see Application Shutdown Behavior for the normative specification
- *PostKeepFlashEvent* This event must be published by a call to *Application.publishEvent()* when a value is kept in the flash.
- *PostPutFlashEvent* This event must be published by a call to *Application.publishEvent()* when a value is stored in the flash.
- *PreClearFlashEvent* This event must be published by a call to *Application.publishEvent()* when a before the flash is cleared.
- *PreRemoveFlashEvent* This event must be published by a call to *Application.publishEvent()* when a value is removed from the flash.

The specification defines the following *ComponentSystemEvent* classes, all in package *jakarta.faces.event*.

- *PostAddToViewEvent* indicates that the *source* component has just been added to the view. Please see 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 Events for a reference to the normative specification.
- PostRenderViewEvent indicates that the UIViewRoot source component has just been rendered. Please see Render Response for the normative specification.
- PostRestoreStateEvent indicates that an individual component instance has just had its state restored. Please see the *UIViewRoot* Events for a reference to the normative specification.

- PostValidateEvent indicates that an individual component instance has just been validated. Please see the *EditableValueHolder* Events for the normative specification.
- *PreDestroyViewMapEvent* indicates that the *Map* that is the view scope is about to be destroyed. Please see, the UIViewRoot Properties for the normative specification.
- *PreRenderComponentEvent* indicates that the *source* component is about to be rendered. Please see Component Tree Manipulation for a reference to the normative specification.
- *PreRenderViewEvent* indicates that the *UIViewRoot* source component is about to be rendered. Please see Render Response for the normative specification.
- PreValidateEvent indicates that an individual component instance is about to be validated. Please see the *EditableValueHolder* 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 *jakarta.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 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. One example of the tag usage is as follows:

```
<h:form>
  <f:event type="postAddToView"
           listener="#{dynamicFormBacking.populateFields}" />
</h:form>
```

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 VDLDocs 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 *jakarta.faces.event.ComponentSystemEventListener.processEvent()*, which is:

```
public void processEvent(jakarta.faces.event.ComponentSystemEvent event)
    throws AbortProcessingException
```

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 System Event Methods for the normative specification of *publishEvent()*.

3.5. Validation Model

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

3.5.1. Overview

Jakarta Faces 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 *jakarta.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.

Jakarta Faces includes implementations of several standard validators, as described in 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. Please see the javadocs for *EditableValueHolder.addValidator()*.

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 Jakarta Faces saves and restores. Validators that wish to have configuration properties saved and restored must also implement *StateHolder* (see 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. The default validators are appended after any locally defined validators once the *EditableValueHolder* is populated and added to the component tree. 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>jakarta.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 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 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 *jakarta.faces.validator.DISABLE_DEFAULT_BEAN_VALIDATOR* <*context-param*> exists and its value is *true*, the following step must be skipped:

- The runtime must guarantee that the validator id *jakarta.faces.Bean* is included in the result from a call to *Application.getDefaultValidatorInfo()* (see Default Validator Ids), regardless of any configuration found in the application configuration resources or via the *@FacesValidator* annotation.

3.5.4. Validation Processing

During the *Process Validations* phase of the request processing lifecycle (as described in Process Validations), the Jakarta Faces 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, there is an entry in the request payload corresponding to this component, and the component has no value, the component is marked invalid and a message is added to the *FacesContext* instance. See Localized Application Messages for details on the message.

3.5.5. Standard Validator Implementations

Jakarta 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 Jakarta Faces implementation must register this instance (see below).

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

The following standard *Validator* implementations (in the *jakarta.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 "jakarta.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 "jakarta.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 "jakarta.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 "jakarta.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 "jakarta.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. Standard identifier for this validator is "jakarta.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.

3.5.6. Bean Validation Integration

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

As stated in the specification goals of Jakarta Bean Validation, validation often gets spread out across the application, from user interface components to persistent objects. Jakarta 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, Jakarta Faces 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

If Bean Validation is present in the runtime environment, the system must ensure that the standard validator with validator-id *jakarta.faces.Bean* is added with a call to *Application.addDefaultValidatorId()*. See Standard Validator Implementations for the description of the standard *BeanValidator*, and *<f:validateBean>* for the Facelet tag that exposes this validator to the page author. This ensures Bean Validation will be called for every field in the application.

If Bean Validation is present, and the *jakarta.faces.VALIDATE_EMPTY_FIELDS <context-param>* is not explicitly set to *false*, Jakarta Faces 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. A ValidatorFactory is retrieved using the following algorithm:

- If the servlet context contains a ValidatorFactory instance under the attribute named jakarta.faces.validator.beanValidator.ValidatorFactory, this instance is used by Jakarta Faces 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, Jakarta Faces 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 Jakarta Faces is running in a Jakarta EE environment, Jakarta Bean Validation will be available, as defined by the Jakarta EE specification, and thus activated in Jakarta Faces. The EE container will be responsible for making the ValidatorFactory available as an attribute in the ServletContext as mentioned above.

3.5.6.3. Class-Level Validation

Jakarta Faces conversion and validation as described in this chapter operates on the principle that all conversion and validation is performed before values are pushed into the model. This principle allows one to safely assume that if a value is pushed into the model, it is of the proper type and has been validated. This validation is done on a "field level" basis, as mentioned in Bean Validation Integration. This approach poses challenges for higher level validation that needs to take the value of several fields together into account to decide if they are valid or not. For example, consider the common case of a user account creation page with two fields for the password. The page can only be considered valid if both password fields are themselves individually valid based on the specified password constraints and also are both the same value. Jakarta Faces provides for this case by providing a facility for performing Class-Level Validation using Bean Validation. Please see the VDLDoc for the *<f:validateWholeBean />* tag for the normative specification of this feature as well as a usage example showing the password validation scenario.

3.5.6.4. Localization of Bean Validation Messages

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

```
public class FacesMessageInterpolator implements MessageInterpolator {
  private final MessageInterpolator delegate;

  public FacesMessageInterpolator(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 Obtaining a ValidatorFactory, Jakarta Faces receives a Validator instance by providing the custom message interpolator to the validator state.

```
//could be cached
MessageInterpolator facesMessageInterpolator = new FacesMessageInterpolator(
    validatorFactory.getMessageInterpolator() );

//...

Validator validator = validatorFactory
    .usingContext()
    .messageInterpolator(facesMessageInterpolator)
    .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 jakarta.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:

```
jakarta.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 Jakarta Faces 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}).

```
jakarta.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, Jakarta Faces UI components without any Java code or configuration XML.

3.6.1.1. What does it mean to be a Jakarta Faces User Interface component?

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

A Jakarta Faces UI Component is represented at runtime by an instance of a Java class that includes *jakarta.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 Jakarta Faces and to conform to user's expectations of what a Jakarta Faces UI Component is. For example, users expect a Jakarta Faces 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 view, such as a Facelet view. 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 Jakarta Faces User Interface component?

To satisfy a user's expectations for a Jakarta Faces 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 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 Jakarta Faces 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 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, Jakarta Faces 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 in Jakarta Faces: resources (Resource Handling) and Facelets (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="jakarta.faces.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 Facelet Templating Tag Library. In addition, the tag library specified in Composite Component Tag Library must be used to declare the metadata for the composite component. Future versions of the Jakarta Faces specification may relax this requirement, but for now at least the *<cc:interface>* and *<cc: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>
<html xmlns:f="jakarta.faces.core"
      xmlns:h="jakarta.faces.html"
      xmlns:ez="jakarta.faces.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="#{loginBacking.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 "jakarta.faces.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 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.

```
<ui:component xmlns:ui="jakarta.faces.facelets"
              xmlns:f="jakarta.faces.core"
              xmlns:h="jakarta.faces.html"
              xmlns:cc="jakarta.faces.composite">
  <cc:interface>
    <cc:actionSource name="loginEvent" />
  </cc:interface>
  <cc:implementation>
    <p>Username: <h:inputText id="usernameInput" /></p>
    <p>Password: <h:inputSecret id="passwordInput" /></p>
    <p><h:commandButton id="loginEvent" value="login"/></p>
  </cc:implementation>
</ui:component>
```

The *<cc: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 *ActionSource* (see ActionSource), and therefore anything one can do with an *ActionSource* in a Facelet markup page you one do with the composite component. (See Component Behavioral Interfaces for more on *ActionSource* and other behavioral interfaces). The <*cc: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 example 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 Composite Component Terms.

1. The user-agent requests the *index.html* from A simple composite component example. This page contains the *xmlns:ez="jakarta.faces.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 “jakarta.faces.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 *jakarta.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 *jakarta.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 <*cc:interface*> section and its children.

Composite Component Definition

The section of markup within the *composite component defining page* that includes the <*cc: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 Renderer

A 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 1. References to Composite Component Requirements in Context

Section	Feature
Implicit Object ELResolver for Facelets and Programmatic Access	Ability for the *composite component author* to refer to the *top level component* from a Jakarta Expression Language expression, such as *#{cc.children[3]}*.
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 *#{cc.attrs.usernameLabel}*
Object Factories	Methods called by the VDL page to create a new instance of a *top level component* for eventual inclusion in the view
Requirements specific to composite components	Requirements of the Facelet implementation relating to Facelets.
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 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 *<cc: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 “name” attributes of the *<cc:interface/ >* element is exposed using the corresponding method on the composite component *BeanDescriptor*. If *ProjectStage* is *Development*, The “displayName”, “shortDescription”, “expert”, “hidden”, and “preferred” attributes of the *<cc:interface/ >* element are exposed using the corresponding methods on the composite component *BeanDescriptor*. Any additional attributes on *<cc: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 *jakarta.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

```
<cc:interface>
  <cc:editableValueHolder name="username" />
  <cc:actionSource name="loginEvent" />
  <cc:actionSource name="allEvents"
                   targets="loginEvent cancelEvent" />
</cc:interface>
```

The list of attached object targets would consist of instances of implementations of the following interfaces from the package *jakarta.faces.view*.

1. EditableValueHolderAttachedObjectTarget
2. ActionSourceAttachedObjectTarget
3. ActionSourceAttachedObjectTarget
4. BehaviorHolderAttachedObjectTarget

- A *ValueExpression* that evaluates to the component type of the composite component. By default this is "*jakarta.faces.NamingContainer*" but the composite component page author can change this, or provide a Java or 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 *<cc: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 *<cc:attribute/ >* element, those properties must be exposed as properties on the *PropertyDescriptor* for that markup element. Any additional attributes on *<cc:attribute/ >* are exposed as attributes accessible from the *getValue()* and *attributeNames()* methods on *PropertyDescriptor*. The return type from getValue() must be a *ValueExpression* with the exception of the *getValue("type")*. The return type from *getValue("type")* must be *Class*. If the value specified for the *type* attribute of *<cc:attribute/>* cannot be converted to an actual *Class*, a *TagAttributeException* must be thrown, including the *Tag* and *TagAttribute* instances in the constructor.

The *composite component BeanDescriptor* must return a *Collection<String>* when its *getValue()* method is called with an argument equal to the value of the symbolic constant *UIComponent.ATTRS_WITH_DECLARED_DEFAULT_VALUES*. The *Collection<String>* must contain the names of any *<cc:attribute>* elements for which the *default* attribute was specified, or *null* if none of the attributes have been given a default value.

3.7. Component Behavior Model

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

3.7.1. Overview

Jakarta Faces 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 Jakarta Faces request processing lifecycle. Jakarta Faces'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 Jakarta Faces 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 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. ClientBehaviorHolder

Components that support client behaviors must implement the ClientBehaviorHolder interface. Refer to 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 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, String 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 Jakarta Faces event / listener model as described in 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 *jakarta.faces.event.FacesEvent*. All events that are broadcast by Jakarta Faces behaviors must extend the *jakarta.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 *jakarta.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 *jakarta.faces.event.BehaviorListener* interface. *BehaviorListener* extends from *jakarta.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 Listener Registration.

3.7.11. Ajax Behavior

3.7.11.1. AjaxBehavior

The specification defines a single concrete *ClientBehavior* implementation: *jakarta.faces.component.behavior.AjaxBehavior*. This class extends *jakarta.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 *jakarta.faces.event.AjaxBehaviorEvents* to registered *jakarta.faces.event.AjaxBehaviorListener* implementations. Refer to the javadocs for more details about *AjaxBehavior*. This behavior must define the behavior id “jakarta.faces.behavior.Ajax”. The renderer type must also be “jakarta.faces.behavior.Ajax”.

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.

jakarta.faces.event.AjaxBehaviorEvent

This event type extends from *jakarta.faces.event.BehaviorEvent* and it is broadcast from an AjaxBehavior. This class follows the standard Jakarta Faces event / listener model, incorporating the usual methods as outlined in Event and Listener Model. This class is responsible for invoking

the method implementation of *jakarta.faces.event.AjaxBehaviorListener.processAjaxBehavior*. Refer to the javadocs for more complete details about this class.

jakarta.faces.event.AjaxBehaviorListener

This listener type extends from *jakarta.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 *AjaxBehavior* Events. See the javadocs for more details about this class.

3.7.12. Adding Behavior To Components

Using the ClientBehaviorHolder interface (see 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
    jakarta.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

Jakarta Faces provides methods for registering *Behavior* implementations and these methods are similar to the methods used to register converters and validators. Refer to Object Factories for the specifics about these methods.

3.7.13.1. XML Registration

Jakarta Faces 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

Jakarta Faces provides the @FacesBehavior annotation for registering custom behavior implementations.

```
@FacesBehavior(value="custom.behavior.Greet")
public class GreetBehavior extends BehaviorBase implements Serializable {
...
}
```

Chapter 4. Standard User Interface Components

In addition to the abstract base class *UIComponent* and the abstract base class *UIComponentBase*, described in the previous chapter, Jakarta Faces 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 Jakarta 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 Jakarta Faces.

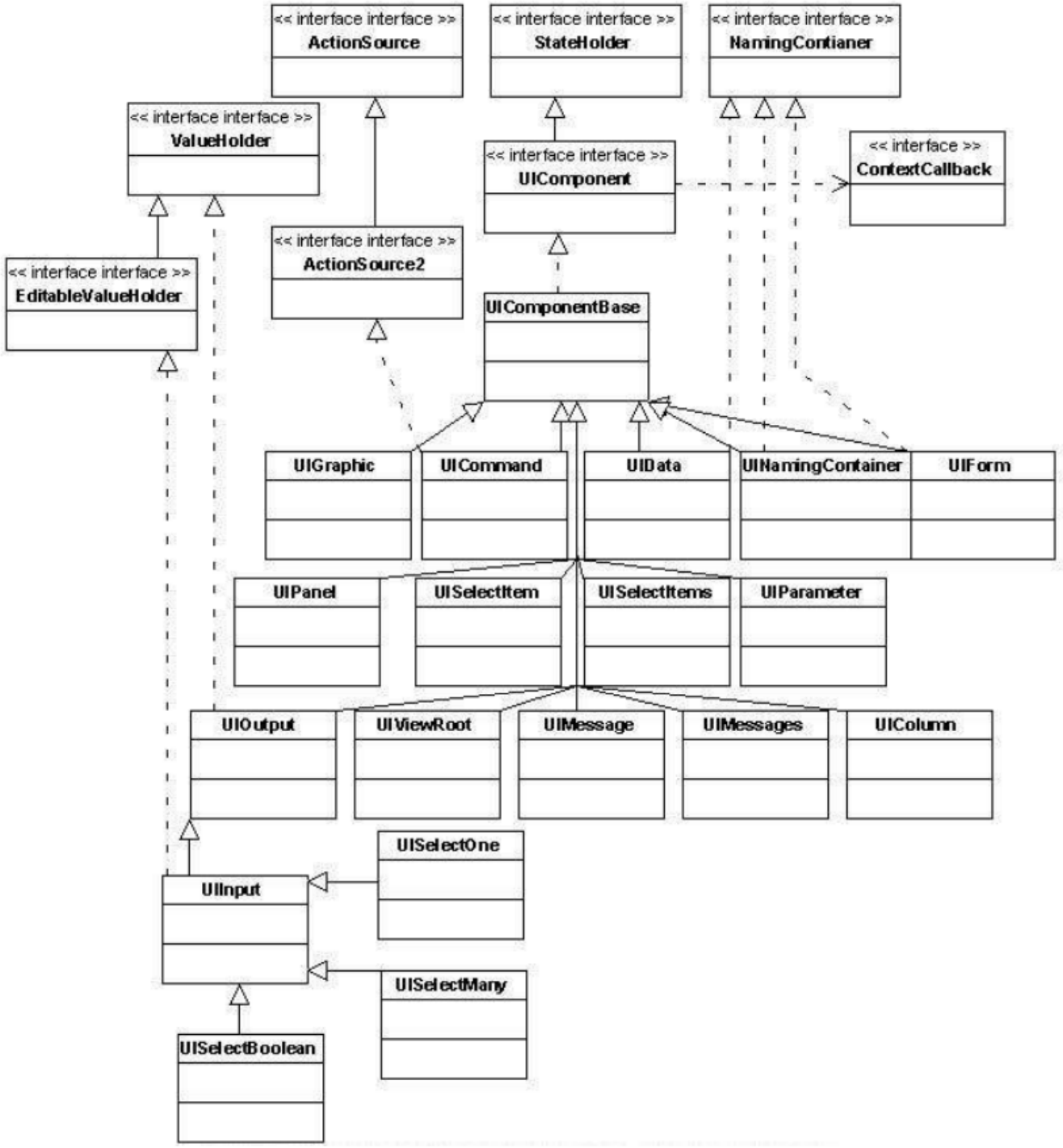
The implementation for each standard *UIComponent* class must specify two public static final String constant values:

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

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

The following UML class diagram shows the classes and interfaces in the package *jakarta.faces.component*.

The jakarta.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 “jakarta.faces.Column”.

4.1.1.2. Properties

UIColumn adds the following render-independent properties:

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

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

- The default value of the *family* property must be set to “jakarta.faces.Column”.
- The default value of the *rendererType* property must be set to *null*.

4.1.1.3. Methods

UIColumn adds no new processing methods.

4.1.1.4. Events

UIColumn adds no new event handling methods.

4.1.2. UICommand

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

4.1.2.1. Component Type

The standard component type for *UICommand* components is “*jakarta.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 ActionSource for information about properties introduced by the implemented classes.

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

- The default value of the *family* property must be set to “jakarta.faces.Command”.
- The default value of the *rendererType* property must be set to “jakarta.faces.Button”.

4.1.2.3. Methods

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

4.1.2.4. Events

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

4.1.3. UIData

UIData (extends *UIComponentBase*; implements *NamingContainer*) is a component that represents a data binding to a collection of data objects represented by a DataModel instance (see 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 “jakarta.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.

Name	Access	Type	Description
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 NamingContainer for information about properties introduced by the implemented classes.

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

- The default value of the *family* property must be set to "jakarta.faces.Data".
- The default value of the *rendererType* property must be set to "*jakarta.faces.Table*".

The current value identified by the *value* property is normally of type *DataModel*. However, a *DataModel* wrapper instance must automatically be provided by the Jakarta Faces 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*)
- java.util.Map (uses the wrapper for java.lang.Iterable by providing access to java.util.Map#entrySet())
- Any other Java object is wrapped by a *DataModel* instance with a single row.

Convenience implementations of *DataModel* are provided in the *jakarta.faces.model* package for each of the above (see 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 NamingContainer for information about methods introduced by the implemented classes.

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

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

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

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

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

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. See NamingContainer for information about event handling introduced by the implemented classes.

4.1.4. UIForm

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

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

4.1.4.1. Component Type

The standard component type for *UIForm* components is "*jakarta.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*.

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

4.1.4.3. Methods.

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

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

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

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

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

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

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

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

The *saveState()* method of UIForm must call *setSubmitted(false)* before calling *super.saveState()* as an extra precaution to ensure the submitted state is not persisted across requests..

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

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*.

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 "*jakarta.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.

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 "jakarta.faces.Graphic".
- The default value of the *rendererType* property must be set to "*jakarta.faces.Image*".

4.1.5.3. Methods

UIGraphic adds no new processing methods.

4.1.5.4. Events

UIGraphic does not originate any standard events.

4.1.6. UIInput

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

4.1.6.1. Component Type

The standard component type for *UIInput* components is "*jakarta.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 EditableValueHolder for information about properties introduced by the implemented interfaces.

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

- The default value of the *family* property must be set to "*jakarta.faces.Input*".
- The default value of the *rendererType* property must be set to "*jakarta.faces.Text*".

- The *Converter* specified by the *converter* property (if any) must also be used to perform String→Object conversions during decoding.
- If the *value* property has an associated *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(jakarta.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 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 "*jakarta.faces.Message*".

4.1.7.2. Properties

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

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

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

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

4.1.7.3. Methods.

UIMessage adds no new processing methods.

4.1.7.4. Events

UIMessage adds no new event handling methods.

4.1.8. UIMessages

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

4.1.8.1. Component Type

The standard component type for *UIMessages* components is "*jakarta.faces.Messages*".

4.1.8.2. Properties

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

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

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

- The default value of the *family* property must be set to “*jakarta.faces.Messages*”.
- The default value of the *rendererType* property must be set to “*jakarta.faces.Messages*”.

4.1.8.3. Methods.

UIMessages adds no new processing methods.

4.1.8.4. Events

UIMessages adds no new event handling methods.

4.1.9. 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 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 "jakarta.faces.OutcomeTarget".

4.1.9.2. Properties

The following render-independent properties are added by thec component:

Name	Access	Type	Description
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".

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 "jakarta.faces.UIOutcomeTarget"
- The default value of the rendererType property must be set to "jakarta.faces.Link"

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 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 "*jakarta.faces.Output*".

4.1.10.2. Properties

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

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

- The default value of the *family* property must be set to “jakarta.faces.Output”.
- The default value of the *rendererType* property must be set to “jakarta.faces.Text”.

4.1.10.3. Methods

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

4.1.10.4. Events

UIOutput does not originate any standard events. See 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 “*jakarta.faces.Panel*”.

4.1.11.2. Properties

UIPanel adds no new render-independent properties.

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

- The default value of the *family* property must be set to “*jakarta.faces.Panel*”.
- The default value of the *rendererType* property must be set to *null*.

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 “*jakarta.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.

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 “jakarta.faces.Parameter”.
- The default value of the *rendererType* property must be set to *null*.

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 “*jakarta.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.

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 “*jakarta.faces.SelectBoolean*”.
- The default value of the *rendererType* property must be set to “*jakarta.faces.Checkbox*”.

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 "*jakarta.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	jakarta.faces.model.SelectItem	The *SelectItem* instance associated with this component.

UISelectItem specializes the behavior of render-independent properties inherited

- The default value of the *family* property must be set to "jakarta.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.

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 "*jakarta.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.

UISelectItems specializes the behavior of render-independent properties inherited

- The default value of the *family* property must be set to "*jakarta.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).

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 "*jakarta.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.

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

4.1.16.3. Methods

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

4.1.16.4. 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 "*jakarta.faces.SelectOne*".

4.1.17.2. Properties

UISelectOne adds no new render-independent properties.

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

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

4.1.17.3. Methods

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

4.1.17.4. 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 "*jakarta.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).

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.

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 "*jakarta.faces.ViewRoot*".
- The default value of the *rendererType* property must be set to *null*.

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. The resource must be added using the algorithm outlined in this method's Javadocs.

```
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. The List must be formulated in accordance with this method's Javadocs.

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 Apply Request Values, Process Validations], Update Model Values and 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 PhaseEvent and PhaseListener for more details on the event and listener class.

```
public void addPhaseListener(PhaseListener listener);
public void removePhaseListener(VPhaseListener listener);
public List<PhaseListener> getPhaseListeners();
```

UIViewRoot must listen for the top level *PostAddToViewEvent* event sent by the *Restore View* phase. Refer to *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.

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.

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 Partial View Traversal, Partial View Processing, Partial View Rendering for an overview of partial processing. *UIViewRoot* must perform partial processing as outlined in the Javadocs for the "processXXX" and "encodeXXX" methods if the current request is a partial request.

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 Jakarta 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 *ELResolver* 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

The Jakarta Faces implementation must provide concrete implementations of DataModel (in the jakarta.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.
- *ResultSetDataModel* —Wrap an object of type *java.sql.ResultSet* (which therefore means that *javax.sql.RowSet* instances are also supported).
- *ScalarDataModel* —Wrap a single Java object in what appears to be a one-row data set.

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

4.2.2. SelectItem

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

4.2.2.1. Properties

An instance of *SelectItem* supports the following properties:

Name	Access	Type	Description
description	RW	String	A description of this selection item, for use in development tools.
disabled	RW	boolean	Flag indicating that this option should be rendered in a fashion that disables selection by the user. Default value is *false*.
label	RW	*String*	Label of this selection item that should be rendered to the user.
value	RW	*Object*	The server-side value of this item, of the same basic data type as the parent component's value. If the parent component type's value is a value 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 Living Standard does not allow an <*optgroup*> to be nested inside another <*optgroup*> within a <*select*> control.

4.2.3.2. Methods

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

4.2.3.3. Events

An instance of *SelectItemGroup* supports no events.

Chapter 5. Expression Language 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 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 Jakarta Faces utilizes in order to evaluate value expressions and method expressions.

Jakarta Faces relies on Jakarta Expression Language as described by version 4.0 of the Jakarta Expression Language specification. Please consult that document for complete details about the Expression Language.

This chapter will focus exclusively on how Jakarta Faces leverages and integrates with Jakarta Expression Language. It does not describe how Jakarta Expression Language 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.

The Jakarta Expression Language has a powerful set of coercion rules that automatically convert the type of the value to the appropriate type. These rules occasionally rely on the JavaBeans *PropertyEditor* facility to perform this conversion. Note that this conversion is entirely separate from normal Jakarta Faces Conversion.

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 Jakarta Expression Language Specification, Version 4.0 or higher 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 Jakarta Faces 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 *ActionSource* behavioral interface (see ActionSource) utilize *MethodExpressions* as follows:
  - If the *actionExpression* property is specified, it must be a *MethodExpression* expression that identifies an Application Action method (see 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 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 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 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 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 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 Listener Classes) that was built in to a separate Java class.

5.2.1. MethodExpression Syntax and Semantics

The exact syntax and semantics of MethodExpression are the domain of the Jakarta Expression Language. Please see Section 1.2.1.2 of the Jakarta Expression Language Specification, Version 4.0 or higher.

5.2.2. Jakarta Faces Managed Classes and Jakarta EE Annotations

The following Jakarta Faces artifacts must be injectable.

Jakarta Faces Artifacts Eligible for Injection

- *jakarta.faces.application.ApplicationFactory*
- *jakarta.faces.application.NavigationHandler*
- *jakarta.faces.application.ResourceHandler*
- *jakarta.faces.application.StateManager*
- *jakarta.faces.component.visit.VisitContextFactory*
- *jakarta.faces.context.ExceptionHandlerFactory*
- *jakarta.faces.context.ExternalContextFactory*
- *jakarta.faces.context.FacesContextFactory*
- *jakarta.faces.context.PartialViewContextFactory*
- *jakarta.faces.event.ActionListener*
- *jakarta.faces.event.SystemEventListener*
- *jakarta.faces.lifecycle.ClientWindowFactory*
- *jakarta.faces.lifecycle.LifecycleFactory*
- *jakarta.faces.event.PhaseListener*
- *jakarta.faces.render.RenderKitFactory*
- *jakarta.faces.view.ViewDeclarationLanguageFactory*
- *jakarta.faces.view.facelets.FaceletCacheFactory*
- *jakarta.faces.view.facelets.TagHandlerDelegateFactory*

Please consult the Jakarta EE Specification for complete details of this feature. Here is a summary of

the Jakarta EE annotations one may use in a artifact from the preceding table.

- *@jakarta.inject.Inject*
- *@jakarta.inject.Named*
- *@jakarta.inject.Qualifier*
- *@jakarta.inject.Scope*
- *@jakarta.inject.Singleton*
- *@jakarta.enterprise.context.ApplicationScoped*
- *@jakarta.enterprise.context.ConversationScoped*
- *@jakarta.enterprise.context.Dependent*
- *@jakarta.enterprise.context.RequestScoped*
- *@jakarta.enterprise.context.SessionScoped*
- *@jakarta.annotation.Resource*
- *@jakarta.annotation.Resources*
- *@jakarta.ejb.EJB*
- *@jakarta.ejb.EJBs*
- *@jakarta.xml.ws.WebServiceRef*
- *@jakarta.xml.ws.WebServiceRefs*
- *@jakarta.persistence.PersistenceContext*
- *@jakarta.persistence.PersistenceContexts*
- *@jakarta.persistence.PersistenceUnit*
- *@jakarta.persistence.PersistenceUnits*

5.3. How Faces Leverages the Expression Language

This section is non-normative and covers the major players in the Jakarta Expression Language and how they relate to Jakarta Faces. The number one goal in this version of the Jakarta Faces specification is to export the concepts behind the Jakarta Faces EL into the Jakarta Expression Language, and then rely on those facilities to get the work done. Readers interested in how to implement the Jakarta Expression Language itself must consult the Jakarta Expression Language Spec document.

5.3.1. ELContext

The ELContext is a handy little “holder” object that gets passed all around the Jakarta Expression Language API. It has two purposes.

- To allow technologies that use the Jakarta Expression Language, such as Jakarta Faces, Jakarta Server Pages and Jakarta Tags, 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 Jakarta Faces. It means, “find the *UIViewRoot* instance for the current view, and

return its *viewId*". The Jakarta Expression Language doesn't know about the "view" implicit object or what a UIViewRoot is, but Jakarta Faces does. The Jakarta Expression Language has plugin points that will get called to resolve "view", but to do so, Jakarta Faces needs access to the *FacesContext* from within the callstack of Expression Language 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 Jakarta Expression Language 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 Jakarta Expression Language Specification, Version 4.0.

5.3.1.1. Lifetime, Ownership and Cardinality

An ELContext instance is created the first time *getELContext()* is called on the *FacesContext* for this request. Please see *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*.

5.3.1.2. Properties

Name	Access	Type	Description
ELResolver	RO	*jakarta.el.ELResolver*	Return the ELResolver instance described in *Faces ELResolver for Facelets and Programmatic Access*
propertyResolved	RW	boolean	Set by an ELResolver implementation if it successfully resolved a property. See *ELResolver* for how this property is used.

5.3.1.3. Methods

Here is a subset of the methods that are relevant to Jakarta Faces.

```
public Object getContext(Class key);
void putContext(Class key, Object contextInstance);
...
```

As mentioned in *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.

5.3.1.4. Events

The creation of an ELContext instance precipitates the emission of an *ELContextEvent* from the *FacesContext* that created it. Please see *ELContext* for details.

5.3.2. 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 Jakarta Expression Language. 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 Jakarta Expression Language Specification, Version 4.0 or higher.

As described in more detail in *ELResolver Instance Provided by Faces*, Faces must provide an implementation of *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. 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.

5.3.2.1. Lifetime, Ownership, and Cardinality

ELResolver instances have application lifetime and scope. The CDI container maintains one top level ELResolver (into which a Faces specific ELResolver is added) accessible from *BeanManager.getELResolver()*. This ELResolver instance is also used from the Jakarta Faces VDL. Faces maintains one *ELResolver* accessible from *FacesContext.getELContext().getELResolver()* and *Application.getELResolver()*.

5.3.2.2. Properties

ELResolver has no proper JavaBeans properties

5.3.2.3. 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 Jakarta Expression Language Specification.

5.3.2.4. Events

ELResolver precipitates no events.

5.3.3. ExpressionFactory

The Jakarta Expression Language owns the *ExpressionFactory* class. It is a factory for *ValueExpression* and *MethodExpression* instances.

5.3.3.1. Lifetime, Ownership, and Cardinality

ExpressionFactory instances are application scoped. The *Application* object maintains the *ExpressionFactory* instance used by Faces (See *Acquiring ExpressionFactory Instance)*. The *ELManager* object maintains the *ExpressionFactory* used by the Jakarta Expression Language (and therefore by the Jakarta Faces VDL). It is permissible for both of these access methods to yield the same java object instance.

5.3.3.2. Properties

ExpressionFactory has no properties.

5.3.3.3. 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 Programmatically Evaluating Expressions .

5.3.3.4. Events

ExpressionFactory precipitates no events.

5.4. ELResolver Instance Provided by Faces

This section provides details on what an implementation of the Jakarta Faces specification must do to support the Jakarta Expression Language for usage in a Jakarta Faces application.

ELResolver mentions that a Faces implementation must provide an implementation of *ELResolver*. This *ELResolver*, let's call it the *Faces 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 *jakarta.el.ELResolver* for the specification and method semantics for each method in *ELResolver*. The remainder of this section lists the implementation requirements for this resolver.

5.4.1. ELResolvers from 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 *jakarta.el.ELResolver*. These are added to the *Faces ELResolver for Facelets and Programmatic Access* in the order in which they occur in the application configuration resources.

5.4.2. ELResolvers from Application.addELResolver()

Any such resolvers are considered at this point in the *Faces ELResolver for Facelets and Programmatic Access* in the order in which they were added.

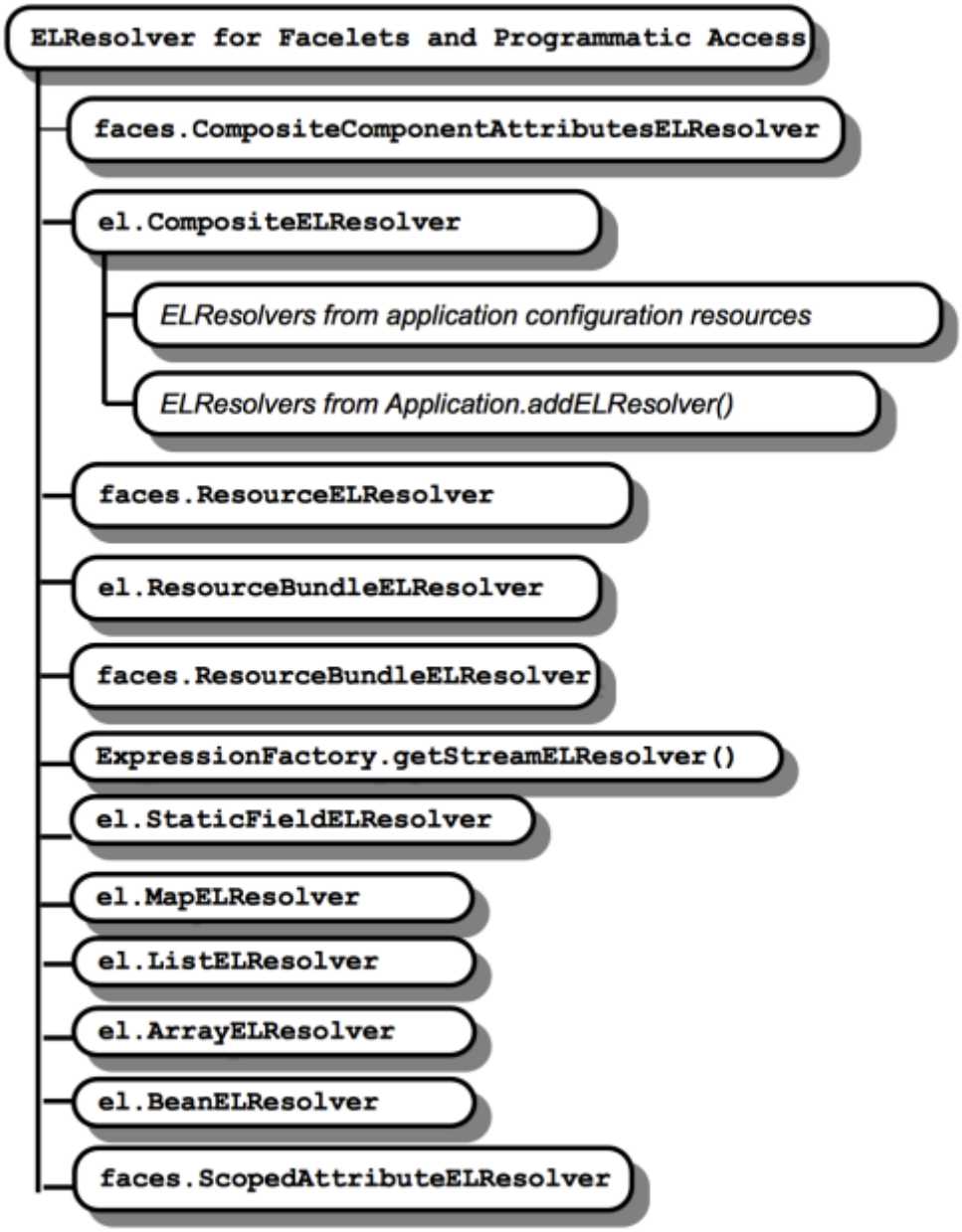
5.4.3. Faces ELResolver for Facelets and Programmatic Access

This section documents the requirements for the second *ELResolver* mentioned in *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.

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.

5.4.3.1. faces.CompositeComponentAttributesELResolver

This ELResolver makes it so expressions that refer to the attributes of a composite component get correctly evaluated. For example, the expression *#{cc.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 2. 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(), put(), and containsKey() are required to be supported. get(): if the result of calling get() on the component attributes map is null, and a default value was declared in the composite component metadata, the value will be a ValueExpression. Evaluate it and return it. Otherwise, simply return the value from the component attributes map. put(): Call getValueExpression() on the component. If this returns non-null, call setValue() on it, passing the value argument as the last argument. Otherwise, simply call through to put on the component attributes map. containsKey(): If the attributes map contains the key, return true. Otherwise, if a default value has been declared for the attribute, return true. Otherwise, return false. The Map implementation must also implement the interface jakarta.faces.el.CompositeComponentExpressionHolder. Otherwise, take no action.
getType	If the base argument to getType() is not an instance of the composite component attributes map or the property argument to getType() is not an instance of java.lang.String, return null. Otherwise, check the top level component’s ValueExpression collection for an expression under the name given by the property argument to getType(). If the expression exists, call getType() on the expression. If the property argument to getType() is not empty, search the composite component’s metadata for a declared type on a <cc:attribute> whose name matches the property argument to getType(). If the expression and the metadata both yield results, the metadata takes precedence ONLY if it provides a narrower result than does the expression, i.e. expression type is assignable from metadata type. If the metadata result does take precedence, call ELContext.setPropertyResolved(true). Otherwise, return whichever result was available, or null.
setValue	Take no action.
isReadOnly	Take no action and return true.
getFeatureDescriptors	Take no action.

ELResolver method	implementation requirements
getCommonPropertyType	Return String.class

5.4.3.2. el.CompositeELResolver

As indicated in *ELResolver for Facelets and Programmatic Access*, following the *faces.CompositeComponentAttributesELResolver*, the semantics obtained by adding a *CompositeELResolver* must be inserted here. This *ELResolver* contains the following *ELResolvers*, described in the referenced sections.

1. *ELResolvers from application configuration resources*
2. *ELResolvers from Application.addELResolver()*

5.4.3.3. faces.ResourceELResolver

This Resource ELResolver for Facelets and Programmatic Access 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 *Resource Handling*.

Table 3. 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['jakarta.faces:faces.js']}*, perform the following. (Note: This is possible due to the ImplicitObjectELResolver returning the ResourceHandler, see Implicit Objects 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 the value of *libraryName* is the literal string "this" (without the quotes), discover the library name of the current resource (or the contract name of the current resource, the two are mutually exclusive) and replace "this" with that library name (or contract name) before calling *ResourceHandler.createResource()*. In the case of resource library contracts, *libraryName* will actually be the contract name. • 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). Call the getRequestPath() method on the Resource instance, pass the result through ExternalContext.encodeResourceUrl() and return the result.
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.

5.4.3.4. el.ResourceBundleELResolver

This entry in the chain must have the semantics the same as the class *jakarta.el.ResourceBundleELResolver*. The default implementation just includes an instance of this resolver in the chain.

5.4.3.5. faces.ResourceBundleELResolver

This Resource Bundle ELResolver for Facelets and Programmatic Access is the means by which resource bundles defined in the application configuration resources are called into play during

Expression Language resolution.

Table 4. 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 jakarta.el.PropertyNotWriteable, since ResourceBundles are read-only.
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;

ELResorver method	implementation requirements
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.

5.4.3.6. Stream, StaticField, Map, List, Array, and Bean ELResolvers

These ELResolver instances are provided by the Jakarta Expression Language API and must be added in the following order:

1. The return from *ExpressionFactory.getStreamELResolver()*
2. *jakarta.el.StaticFieldELResolver*
3. *jakarta.el.MapELResolver*
4. *jakarta.el.ListELResolver*
5. *jakarta.el.ArrayELResolver*
6. *jakarta.el.BeanELResolver*

These actual ELResolver instances must be added. It is not compliant to simply add other resolvers that preserve these semantics.

5.4.3.7. faces.ScopedAttributeELResolver

This Scoped Attribute ELResolver for Facelets and Programmatic Access 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. 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 facesContext.getViewRoot().getViewMap().get() (accounting for the potential for null returns safely). 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;
isReadOnly	If base is false, setPropertyResolved(true) return false; Otherwise, return false;

ELResorver method	implementation requirements
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.

5.5. Current Expression Evaluation APIs

5.5.1. ELResolver

Please see *ELResolver* for more details.

5.5.2. ValueExpression

It is the main object oriented abstraction for an Expression Language expression that results in a value either being retrieved or set. Please see Chapter 2 of the Jakarta Expression Language Specification, Version 4.0 or higher.

5.5.3. MethodExpression

It is the main object oriented abstraction for an Expression Language expression that results in a method being invoked. Please see Chapter 2 of the Jakarta Expression Language Specification, Version 4.0 or higher.

5.5.4. Expression Evaluation Exceptions

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

- *jakarta.el.ELException* (which extends *java.lang.Exception*)—used to report a problem evaluating a value exception dynamically.
- *MethodNotFoundException* (which extends *jakarta.el.ELException*)—used to report that a requested public method does not exist in the context of evaluation of a method expression.
- *jakarta.el.PropertyNotFoundException* (which extends *jakarta.el.ELException*)—used to report that a requested property does not exist in the context of evaluation of a value expression.
- *jakarta.el.PropertyNotWriteableException* (which extends *jakarta.el.ELException*)—used to indicate that the requested property could not be written to when evaluating the expression.

5.6. CDI Integration

Jakarta Faces must run in a container that supports CDI version 3.0 or higher. This requirement allows CDI to provide all the functionality of the managed bean facility in a better integrated way with the rest of the Jakarta EE platform. Delegating these features to CDI allows them to evolve independently of Jakarta Faces. The remainder of this section specifies some details of CDI integration pertinent to Jakarta Faces.

5.6.1. Jakarta Faces Objects Valid for @Inject Injection

It must be possible to inject the following Jakarta Faces objects into other objects using *@Inject*.

Maps Returned by Various Jakarta Faces Accessors

The annotations in package *jakarta.faces.annotation* are used to cause *@Inject* injection of the corresponding *Map* into a field. Generics may be used.

Jakarta Faces Objects

It must be possible to *@Inject* the following Jakarta Faces and Jakarta EE objects into CDI beans.

- *jakarta.faces.application.ResourceHandler*
- *jakarta.faces.context.ExternalContext*
- *jakarta.faces.context.FacesContext*
- *jakarta.faces.context.Flash*
- *jakarta.servlet.http.HttpSession*

Support for Injection into Jakarta Faces Managed Objects

It must be possible to use *@Inject* when specifying the following kinds of Jakarta Faces managed objects.

- Validators declared with *@jakarta.faces.validator.FacesValidator(managed=true)*
- Converters declared with *@jakarta.faces.convert.FacesConverter(managed=true)*
- FacesBehaviors declared with *@jakarta.faces.component.behavior.FacesBehavior(managed=true)*

5.6.2. Expression Language Resolution

The Implicit Objects for Facelets and Programmatic must be resolved using CDI

5.6.2.1. Implicit Objects for Facelets and Programmatic Access

The following Implicit Objects for Facelets and Programmatic Access must be resolved using CDI.

Table 6. Implicit Objects for Programmatic Access

implicitObject	*source*	*scope*
facesContext	FacesContext.getCurrentInstance()	request
externalContext	facesContext.getExternalContext()	request

implicitObject	*source*	*scope*
application	externalContext.getContext()	application
applicationScope	externalContext.getApplicationMap()	application
cc	UIComponent.getCurrentCompositeComponent(facesContext)	dependent
cookie	externalContext.getRequestCookieMap()	request
component	UIComponent.getCurrentComponent(facesContext)	dependent
flash	externalContext.getFlash()	request
flow	facesContext.getApplication().getFlowHandler().getCurrentFlow()	flow
flowScope	facesContext.getApplication().getFlowHandler().getCurrentFlowScope()	flow
header	externalContext.getRequestHeaderMap()	request
headerValues	externalContext.getRequestHeaderValuesMap()	request
initParam	externalContext.getInitParameterMap()	application
param	externalContext.getRequestParameterMap()	request
paramValues	externalContext.getRequestParameterValuesMap()	request
request	externalContext.getRequest()	request
requestScope	externalContext.getRequestMap()	request
resource	facesContext.getApplication().getResourceHandler()	request
session	externalContext.getSession()	session
sessionScope	externalContext.getSessionMap()	session
view	facesContext.getViewRoot()	request
viewScope	facesContext.getViewRoot().getViewMap()	request

Chapter 6. Per-Request State Information

During request processing for a Jakarta Faces 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

Jakarta Faces defines the *jakarta.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 Jakarta Faces implementation, prior to beginning the request processing lifecycle, by a call to the *getFacesContext* method of *FacesContextFactory*, as described in FacesContextFactory. When the request processing lifecycle has been completed, the Jakarta Faces implementation will call the *release* method, which gives Jakarta Faces implementations the opportunity to release any acquired resources, as well as to pool and recycle *FacesContext* instances rather than creating new ones for each request.

6.1.1. Application

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

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

6.1.2. Attributes

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

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.

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. 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 *jakarta.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(jakarta.el.ELContextEvent)* method.

6.1.4. ExternalContext

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

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

The 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.

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

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

Name	Access	Type	Description
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 [1].
userPrincipal	RO	*java.security.Principal*	The Principal object containing the name of the currently logged on user (if any).

See the JavaDocs for the normative specification.

6.1.4.1. 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.

6.1.5. 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 Jakarta Faces 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();
```

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()*. 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 *Message*s, 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 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);
```

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

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

The *getRenderResponse()* and *getResponseComplete()* methods allow a Jakarta Faces-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();
```

The getPartialViewContext()method must return an instance of PartialViewContext either by creating a new instance, or returning an existing instance from the FacesContext.

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 Partial View Processing and Partial View Rendering. Refer to the JavaDocs for the jakarta.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, Jakarta Faces components, and application objects that access them, are passed a reference to the *FacesContext* instance for the current request. However, in some cases, no such reference is available. The *getCurrentInstance()* method may be called by any Java class in the current web application to retrieve an instance of the *FacesContext* for this request. The Jakarta Faces 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.

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 .

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 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 PhaseListener for the circumstances under which *ExceptionHandler* is used.

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. 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 *@PreDestroy* annotation on a managed bean.
- All classes when an *AbortProcessingException* is thrown.

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. At that point, all *Exceptions* are passed to the *ExceptionHandler* as described in 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 ExceptionQueuedEvent getHandledExceptionEvent();
```

Return the first “handled” *ExceptionQueuedEvent*, that is, the one that was actually re-thrown.

```
public Iterable<ExceptionQueuedEvent> 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 *jakarta.el.ELException.class*. If there is no root cause, *null* is returned.

```
public Iterable<ExceptionQueuedEvent> 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 *ExceptionQueuedEvent* instances in the order in which they were queued by calls to *Application.publishEvent(ExceptionQueuedEvent.class, eventContext)*.

For each *ExceptionQueuedEvent* in the list, call its *getContext()* method and call *getException()* on the returned result. Upon encountering the first such *Exception* the corresponding *ExceptionQueuedEvent* must be set so that a subsequent call to *getHandledExceptionEvent()* or *getHandledExceptionEvents()* returns that *ExceptionQueuedEvent* instance. The implementation must also ensure that subsequent calls to *getUnhandledExceptionEvents()* do not include that *ExceptionQueuedEvent* 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 ExceptionQueuedEvent)
    throws AbortProcessingException;
```

The default implementation must store the argument *ExceptionQueuedEvent* in a strongly ordered queue for later processing by the *handle()* method.

6.2.2. 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 *ExceptionQueuedEvent* was handled.
- All scoped variables in request, view, session and application scope.
- If the error happens during the execution of the view 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 *ExceptionQueuedEvent* 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 “*jakarta.faces.error.xhtml*”.

6.3. FacesMessage

Each message queued within a *FacesContext* is an instance of the *jakarta.faces.application.FacesMessage* class. The presence of one or more *FacesMessage* instances on the *FacesContext* indicates a failure of some kind during the lifecycle. In particular, a validation or conversion failure is required to cause a *FacesMessage* to be added to the *FacesContext*.

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 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 startCDATA();
public void endCDATA();
```

Start and end an XML CDATA Section..

```
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

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

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

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

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

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

6.7. ExceptionHandlerFactory

A single instance of *jakarta.faces.context.ExceptionHandlerFactory* must be made available to each Jakarta Faces-based web application running in a servlet or portlet container. The factory instance can be acquired, by Jakarta Faces implementations or by application code, by executing:

```
ExceptionHandlerFactory factory = (ExceptionHandlerFactory)
    FactoryFinder.getFactory(FactoryFinder.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

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

```
ExternalContextFactory factory = (ExternalContextFactory)
    FactoryFinder.getFactory(FactoryFinder.EXTERNAL_CONTEXT_FACTORY);
```

The *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*.

Chapter 7. Application Integration

Previous chapters of this specification have described the component model, request state information, and the next chapter describes the rendering model for Jakarta 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 Jakarta Faces. These classes are in the *jakarta.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 Jakarta 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 Jakarta 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 ApplicationFactory), and provides default implementations of features that determine how application logic interacts with the Jakarta Faces implementation. Advanced applications (or application frameworks) can install replacements for these default implementations, which will be used from that point on. Access to several integration objects is available via JavaBeans property getters and setters, as described in the following subsections.

7.1.1. ActionListener Property

```
public ActionListener getActionListener();
public void setActionListener(ActionListener listener);
```

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

- The *processAction()* method must first 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 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. If the originating component has an attribute whose name is equal to the value of the symbolic constant *ActionListener.TO_FLOW_DOCUMENT_ID_ATTR_NAME*, invoke *handleNavigation(FacesContext, String, String, String)* passing the value of the attribute as the last parameter. Otherwise, invoke *handleNavigation(FacesContext, String, String)*. In either case, the first *String* argument is the expression string of the *fromAction* and the second *String* argument is the logical outcome.

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.

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

7.1.3. FlowHandler Property

```
public FlowHandler getFlowHandler();
public void setFlowHandler(FlowHandler handler);
```

Return or replace the *FlowHandler* that will be used by the *NavigationHandler* to make decisions about navigating application flow. See FlowHandler for an overview of the flow feature.

setFlowHandler() may only be called at application startup, before any Faces requests have been processed. This is a limitation of the current Specification, and may be lifted in a future release. *getFlowHandler()* may be called at any time after application startup.

7.1.4. NavigationHandler Property

```
public NavigationHandler getNavigationHandler();
public void setNavigationHandler(NavigationHandler handler);
```

Return or replace the *NavigationHandler* instance (see 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 Default NavigationHandler

Algorithm:

7.1.5. StateManager Property

```
public StateManager getStateManager();
public void setStateManager(StateManager manager);
```

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

7.1.6. ELResolver Property

```
public ELResolver getELResolver();
public void addELResolver(ELResolver resolver);
```

Return the ELResolver instance to be used for all Expression Language resolution. This is actually an instance of jakarta.el.CompositeELResolver that must contain the ELResolver instances as specified in *ELResolver for Facelets and Programmatic Access*.

addELResolver must cause the argument resolver to be added at the end of the list in the jakarta.el.CompositeELResolver returned from getELResolver(). See the diagram in *ELResolver for Facelets and Programmatic Access*

7.1.7. 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 ELContextListeners. 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().

7.1.8. ViewHandler Property

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

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

7.1.9. ProjectStage Property

```
public ProjectStage getProjectStage();
```

This method must return the enum constant from the class *jakarta.faces.application.ProjectStage* as specified in the corresponding application init parameter, JNDI entry, or default Value. See Application Configuration Parameters.

7.1.10. Acquiring ExpressionFactory Instance

```
public ExpressionFactory getExpressionFactory();
```

Return the ExpressionFactory instance for this application. This instance is used by the evaluateExpressionGet (*See Programmatically Evaluating Expressions*) convenience method.

The default implementation simply returns the ExpressionFactory from the Jakarta Expression Language by calling ELManager.getExpressionFactory().

7.1.11. 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()*.

7.1.12. Object Factories

The *Application* instance for a web application also acts as an object factory for the creation of new Jakarta Faces 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 [2], based on the requested identifier. The names of the implementation class used for each identifier is normally provided by the Jakarta Faces implementation automatically (for standard classes described in this Specification), or in one or more application configuration resources (see Application Configuration Resources) included with a Jakarta Faces 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);
```

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.

```
public UIComponent createComponent(
    FacesContext context, Resource componentResource);
```

All variants *createComponent()* must take some action to inspect the component for *@ResourceDependency* and *@ListenerFor* annotations. Please see the JavaDocs and 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);
```

Jakarta Faces-based applications can register additional mappings of identifiers to a corresponding

fully qualified class name, or replace mappings provided by the Jakarta Faces implementation in order to customize the behavior of standard Jakarta Faces features. These methods are also used by the Jakarta Faces 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();
```

Jakarta Faces-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.

7.1.12.1. 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 Validation Registration.

7.1.13. 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 Jakarta Faces objects.

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

Jakarta Faces 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 Jakarta Faces

implementation will acquire message strings that correspond to standard message keys See Localized Application Messages for a list of the standard message keys recognized by Jakarta Faces.

7.1.14. System Event Methods

System events are described in System Events. This section describes the methods defined on *Application* that support system events

7.1.14.1. 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()*. *NOTE*: The implementation must not support subclasses for the *systemEventClass* and/or *sourceClass* arguments to *subscribeToEvent()* or *publishEvent()*. 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 Jakarta Faces application. The specification for when *publishEvent()* is called is given in the javadoc for the event classes that are listed in Event Classes. See the javadoc for *publishEvent()* for the complete specification.

7.1.14.2. 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 *jakarta.faces.application.ApplicationFactory* must be made available to each Jakarta Faces-based web application running in a servlet or portlet container. The factory instance can be acquired by Jakarta Faces 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 Jakarta Faces implementation must provide a default *Application* instance whose behavior is described in 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 Jakarta Faces 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 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

Most Jakarta Faces applications can be thought of as a directed graph of views, each node of which roughly corresponds to the user’s perception of “location” within the application. Applications that use the Faces Flows feature have additional kinds of nodes in the directed graph. In any case, navigating the nodes of this graph is the responsibility of the *NavigationHandler*. 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* or the empty string, 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 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 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.

Jakarta Faces also contains the *ConfigurableNavigationHandler* interface, which extends the contract of the *NavigationHandler* to include two additional methods that accommodate 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 *getNavigationCases* 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 NavigationCase getNavigationCase(FacesContext context,
    String fromAction, String outcome);

public Map<String, Set<NavigationCase>> getNavigationCases();
```

A Jakarta Faces 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.

7.4.2. Default NavigationHandler Algorithm

Jakarta Faces 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 or flow node 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 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.xhtml” 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.

Version 2.2 of the specification introduced the Faces Flows feature. With respect to the navigation algorithm, any text that references a *view identifier*, such as <*from-view-id*> or <*to-view-id*>, can also refer to a flow node, subject to these constraints.

- When outside of a flow, *view identifier* has the additional possibility of being a flow id.
- When inside a flow, a *view identifier* has the additional possibility of being the id of any node within the current flow.

If the specification needs to refer to a *view identifier* that is an actual VDL view (and not a VDL view or a flow, or flow node), the term *vdl view identifier* will be used.

Nested within each <*navigation-rule*> element are zero or more <*navigation-case*> elements that contain additional matching criteria based on the action reference expression value used to select an application action to be invoked (if any), and the logical outcome returned by calling the *invoke()* method of that application action [3]. 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) [4]. When the <if> element is present, the value expression it contains must evaluate to true when the navigation case is being consulted in order for the navigation case to match [5]. 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.

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

- If no navigation case is matched by a call to the handleNavigation() method, this is an indication that the current view should be redisplayed. 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 [6], 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, proceed as follows.
  - If the <*to-view-id*> element is the id of a flow, discover that flow's start node and resolve it to a *vdl view identifier* by following the algorithm in Requirements for Explicit Navigation in Faces Flow Call Nodes other than ViewNodes
  - If the <*to-view-id*> element is a non-view flow node, resolve it to a *vdl view identifier* by following the algorithm in Requirements for Explicit Navigation in Faces Flow Call Nodes other than ViewNodes.
  - If *UIViewAction.isProcessingBroadcast()* returns *true*, call *getFlash().setKeepMessages(true)* on the current *FacesContext*. Compare the viewId of the current viewRoot with the <*to-view-id*> of the matching <*navigation-case*>. If they differ, take any necessary actions to effectively restart the Jakarta Faces lifecycle on the <*to-view-id*> of the matching <*navigation-case*>. Care must be taken to preserve any view parameters or navigation case parameters, clear the view map of the *UIViewRoot*, and call *setRenderAll(true)* on the *PartialViewContext*. Implementations may choose to meet this requirement by treating this case as if a <*redirect* /> was specified on the matching <*navigation-case*>. If the viewIds do not differ, continue on to the next bullet point.
  - Clear the view map if the viewId of the new *UIViewRoot* differs from the viewId of the current *UIViewRoot*.
  - If 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. Call *transition()* on the *FlowHandler*, passing the current *FacesContext*, the current flow, the new flow and the *facesFlowCallNode* corresponding to this faces flow call, if any. Pass the new *UIViewRoot* 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 the <*redirect/*> element *was* specified in this <*navigation-case*>, or this invocation of *handleNavigation()* was due to a *UIViewAction* broadcast event where the new *viewId* is different from the current *viewId*, resolve the <*to-view-id*> to a view identifier, using the algorithm in *Requirements for Explicit Navigation in Faces Flow Call Nodes other than ViewNodes*. 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. If this navigation is a flow transition (where current flow is not the same as the new flow), include the relevant flow metadata as entries in the *parameters* .
    - If current flow is not null and new flow is null, include the following entries: *FlowHandler.TO_FLOW_DOCUMENT_ID_REQUEST_PARAM_NAME*: *FlowHandler.NULL_FLOW*

FlowHandler.FLOW_ID_REQUEST_PARAM_NAME: "" (the empty string)

- If current flow is null and new flow is not null, include the following entries:
  FlowHandler.TO_FLOW_DOCUMENT_ID_REQUEST_PARAM_NAME: The to flow document id
  FlowHandler.FLOW_ID_REQUEST_PARAM_NAME: the flow id for the flow that is the destination of the transition.
- If the *parameters* map has entries for either of these keys, both of the entries must be replaced with the new values. This allows the call to *FlowHandler.clientWindowTransition()* to perform correctly when the GET request after the redirect happens.

The return from *getRedirectURL()* is the value to be sent to the client to which the redirect will occur. 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*. This implicit matching algorithm also includes navigating within the current faces flow, and returning from the current faces flow.
  - Let *outcome* be *viewIdToTest*.
  - 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*" or "*faces-include-view-params=true*". If either are 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 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, the implementation must ensure the algorithm described for *ViewHandler.deriveViewId()* specified in *Default ViewHandler Implementation* is performed. Let the result be *implicitViewId*.
  - If *implicitViewId* is non-*null*, discover if *fromOutcome* is equal to the flow-id of an existing flow in the *FlowHandler*. If so find the start node of the flow. If the start node is a *ViewNode*, let *viewIdToTest* be the *vdlDocumentId* value of the *ViewNode*. Call *deriveViewId* as in the

preceding step and let the result be *implicitViewId*. If *fromOutcome* is not equal to the flow-id of an existing flow in the *FlowHandler*, and we are currently in a flow, discover if this is call to a *faces-flow-return* node. If so, obtain the *fromOutcome* of the *faces-flow-return* node, re-apply this algorithm to derive the value of the *implicitViewId* and continue.

    - If the *implicitViewId* is non-*null*, take the following action. If *isRedirect* is *true*, append the *queryString* to *implicitViewId*. Let *implicitNavigationCase* 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 *implicitNavigationCase* as a matching navigation case and return to the first step above that starts with "If a matching <*navigation-case*> element was located…".

- If *UIViewAction.isProcessingBroadcast()* returns *true*, call *getFlash().setKeepMessages(true)* on the current *FacesContext*. Compare the viewId of the current viewRoot with the effective <*to-view-id*> of the matching <*navigation-case*>. If they differ, take any necessary actions effectively restart the Jakarta Faces lifecycle on the effective <*to-view-id*> of the matching <*navigation-case*>. Care must be taken to preserve any view parameters or navigation case parameters, clear the view map of the *UIViewRoot*, and call *setRenderAll(true)* on the *PartialViewContext*.
- If none of the above steps found a matching <*navigation-case*>, perform the steps in Requirements for Explicit Navigation in Faces Flow Call Nodes other than ViewNodes to find a matching <*navigation-case*>.
- 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. This includes clearing out the view map.

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 N*avigationHandler* 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).

7.4.2.1. Requirements for Explicit Navigation in Faces Flow Call Nodes other than ViewNodes

These steps must be performed in this order to determine the *vdl view identifier* when navigating to a flow node that is not a view node.

Algorithm for resolving a *nodeId* to a *vdl view identifier*.

- If *nodeId* is a view node, let *vdl view identifier* be the value of *nodeId* and exit the algorithm.
- If the node is a *SwitchNode*, iterate over the *NavigationCase* instances returned from its *getCases()* method. For each, one call *getCondition()*. If the result is *true*, let *nodeId* be the value of its *fromOutcome* property.
- If the node is a *MethodCallNode*, let *nodeId* be the value invoking the value of its *methodExpression* property. If the result is *null*, let *nodeId* be the value of the *MethodCallNode*'s *outcome* property.
- If the node is a *FlowCallNode*, save it aside as *facesFlowCallNode*. Let *flowId* be the value of its *calledFlowId* property and *flowDocumentId* be the value of its *calledFlowDocumentId* property. If no *flowDocumentId* exists for the node, let it be the string resulting from *flowId* + *"/"* + *flowId* *".xhtml"*. Ask the *FlowHandler* for a *Flow* for this *flowId*, *flowDocumentId* pair. Obtain a reference to the start node and execute this algorithm again, on that start node.
- If the node is a *ReturnNode* obtain its navigation case and call *FlowHandler.pushReturnMode()*. This enables the navigation to proceed with respect to the calling flow's navigation rules, or the application's navigation rules if there is no calling flow. Start the navigation algorithm over using it as the basis but pass the value of the symbolic constant *jakarta.faces.flow.FlowHandler.NULL_FLOW* as the value of the *toFlowDocumentId* argument. If this does not yield a navigation case, call *FlowHandler.getLastDisplayedViewId()*, which will return the last displayed view id of the calling flow, or *null* if there is no such flow. In a *finally* block, when the re-invocation of the navigation algorithms completes, call *FlowHandler.popReturnMode()*.

7.4.2.2. Requirements for Entering a Flow

If any of the preceding navigation steps cause a flow to be entered, the implementation must perform the following steps, in this order, before continuing with navigation.

- Make it so any *@FlowScoped* beans for this flow are able to be activated when an Expression Language expression that references them is evaluated.
- Call the initializer for the flow, if any.
- Proceed to the start node of the flow, which may be any flow node type.

An attempt to navigate into a flow other than via the identified start node of that throw should cause a *FacesException*.

7.4.2.3. Requirements for Exiting a Flow

If any of the preceding navigation steps cause a flow to be exited, the implementation must perform the following steps, in this order, before continuing with navigation.

- Call the finalizer for the flow, if any.
- De-activate any *@FlowScoped* beans for the current flow.
- If exiting via a return node ensure the return parameters are correctly passed back to the caller.

7.4.2.4. Requirements for Calling A Flow from the Current Flow

If any of the preceding navigation steps cause a flow to be called from another flow, the *transition()* method on *FlowHandler* will ensure parameters are correctly passed.

7.4.3. Example NavigationHandler Configuration

The following <*navigation-rule*> elements might appear in one or more application configuration resources (see 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.xhtml</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.xhtml</to-view-id>
  </navigation-case>

  <navigation-case>
    <description>
      Illustrate paramaters
    </description>
    <from-outcome>redirectPasswordStrength</from-outcome>
    <redirect>
      <view-param>
        <name>userId</name>
        <value>someValue</value>
      </view-param>
      <include-view-params>true</include-view-params>
    </redirect>
  </navigation-case>
```

```
</navigation-rule>
```

```
<navigation-rule>
  <description>
    LOGIN PAGE NAVIGATION HANDLING
  </description>
  <from-view-id> /login.xhtml </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.xhtml</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.xhtml</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.xhtml</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.xhtml</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.xhtml</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.xhtml</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.xhtml</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>
```

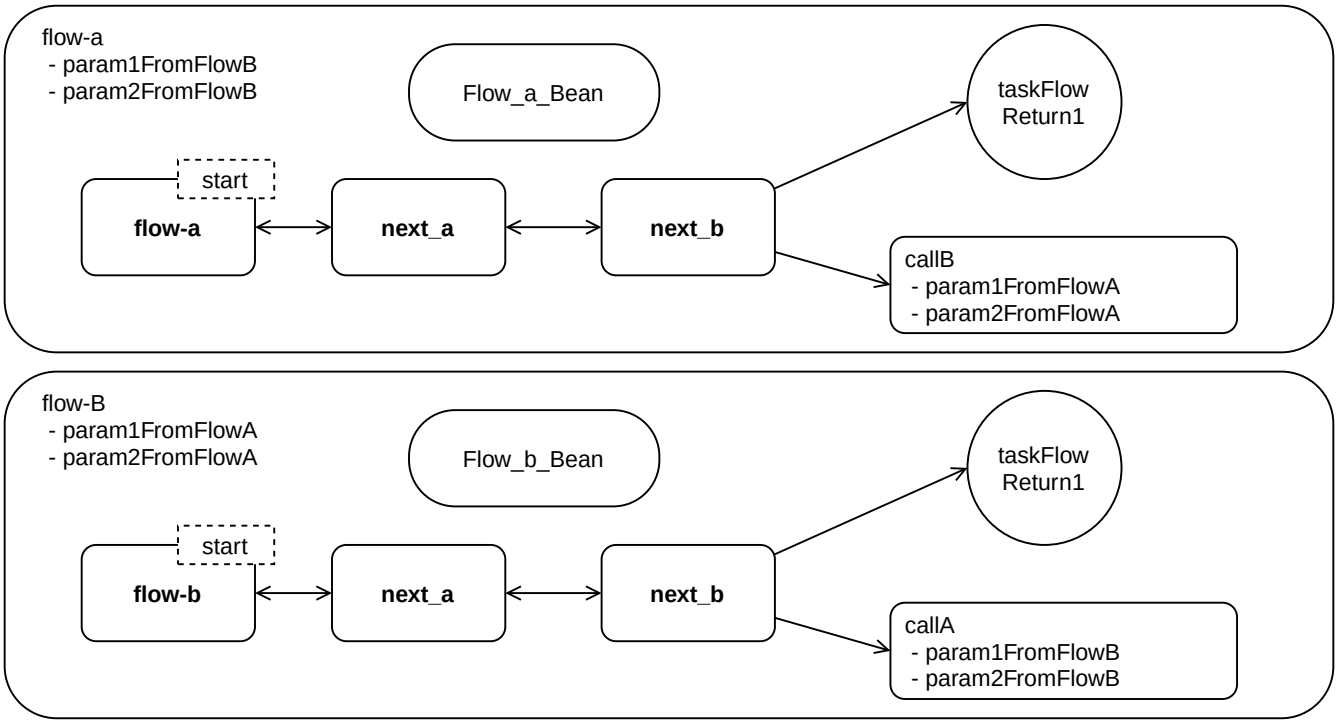
```
    <from-action>#{pizzaBuilder.placeOrder}</from-action>
    <if>#{not account.oneClickDelivery}</if>
    <to-view-id>/delivery.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

7.5. FlowHandler

Any Jakarta Faces application can be modeled as a directed graph where the nodes are views and the edges are transitions between the views. Faces Flows introduces several other kinds of nodes to this directed graph, providing support for encapsulating related views and edges together. Applications can be created as composites of modules of functionality, with each module consisting of well defined entry and exit conditions, and the ability to share state among the nodes within each module. This feature is heavily influenced by the design of ADF Task Flows in Oracle's Fusion Middleware and also by Spring Web Flow and Apache MyFaces CODI. The normative specification for this feature proceeds from the Javadoc for the class *jakarta.faces.flow.FlowHandler*, and also from related requirements in NavigationHandler. This section provides a non-normative usage example and walkthrough of feature so that all the other parts of the specification that intersect with this feature can be discovered.

7.5.1. Non-normative example

Here is a simple example to introduce the feature. It does not touch on all aspects of the feature. The example has two flows, each of which calls the other, passing parameters. Any view outside of a flow may navigate to either of the flows, named flow-a and flow-b.



This diagram uses the following conventions.

- view nodes are boxes

- faces flow return nodes are circles
- faces flow call nodes are boxes with the corners chopped off
- *@FlowScoped* beans are rectangles semi-circular short sides
- the start node is marked “start”
- inbound and outbound parameters are listed by name
- arrows show valid traversals among the nodes.

These flows are identical, except for the names of their constituents, and each has the following properties.

- Three view nodes, one of which is the implicit start node
- One faces flow return node, each of which returns the outcome “return1”
- One flow call node, which calls the other flow, with two outbound parameters, named to match up with the other flow
- Two inbound parameters, named to match up with the other flow

The different kinds of nodes mentioned in the preceding discussion are defined in the javadoc for class *jakarta.faces.flow.FlowHandler*.

Consider this simple web app, called *basic_faces_flow_call.war*, containing the above mentioned flows. The file layout for of the app is shown next. The example is shown using maven war packaging

```
basic_faces_flow_call/
  pom.xml
  src/main/webapp/
             index.xhtml
             return1.xhtml
             WEB-INF/beans.xml
             flow-a/
               flow-a.xhtml
               next_a.xhtml
               next_b.xhtml
             flow-b/
               flow-b-flow.xml
               next_a.xhtml
               next_b.xhtml
  src/main/java/com/sun/faces/basic_faces_flow_call/
                                FlowA.java
                                Flow_a_Bean.java
                                Flow_b_Bean.java
```

To complete the example, the execution of the flows is examined. When the application containing these flows is deployed, the runtime discovers the flow definitions and adds them to the internal flow data structure. One flow is defined in *flow-b-flow.xml*. This is an XML file conforming to the Application Configuration Resources syntax described in Application Configuration Resources. The

other flow is defined in *FlowA.java*, a class with a method with the *@FlowDefinition* annotation. When the flow discovery is complete, an application scoped, thread safe data structure containing the flow definitions is available from the *jakarta.faces.flow.FlowHandler* singleton. This data structure is navigable by the runtime via the *jakarta.faces.flow.Flow* API.

When the user agent visits *http://localhost:8080/basic_faces_flow_call/faces/index.xhtml*, they see a page with two buttons, the actions of which are *flow-a*, and *flow-b*, respectively. Clicking either button causes entry to the corresponding flow. In this case, the user clicks the *flow-a* button. The *@FlowScoped* bean *Flow_a_Bean* is instantiated by the container and navigation proceeds immediately to the start node, in this case *flow-a.xhtml*. The user proceeds directly to click a button taking them to *next_a.xhtml*, and then to *next_b.xhtml*. On that page there is a button whose action is *callB*. Clicking this button activates the correspondingly named faces flow call node, which prepares the specified outbound parameters, de-activates *Flow_a_Bean* and calls *flow-b*.

Upon entry to *flow-b*, the *@FlowScoped* bean *Flow_b_Bean* is instantiated by the container, the outbound parameters from *flow-a* are matched up with corresponding inbound parameters on *flow-b* and navigation proceeds immediately to the start node, in this case *flow-b.xhtml*. The user proceeds directly to click a button taking them to *next_a.xhtml*, and then to *next_b.xhtml*. On that page there is a button whose action is *taskFlowReturn1*. Clicking this button causes *Flow_b_Bean* to be deactivated and navigation to the view named *return1* to be performed.

7.5.2. Non-normative Feature Overview

The normative requirements of the feature are stated in the context of the part of the specification impacted. This section gives the reader a non-normative overview of the feature that touches on all the parts of the specification that intersect with this feature.

Startup Time

At startup time, the runtime will discover flows available for this application. *This behavior is normatively specified in Faces Flows and in the XML schema for the application configuration resources.*

Invoke Application Time

The default *ActionListener* may need to take special action when calling into a flow. *This behavior is normatively specified in ActionListener Property.*

The default *NavigationHandler* implementation must use the *FlowHandler* during its operation. *This behavior is normatively specified in Default NavigationHandler Algorithm.*

7.6. ViewHandler

ViewHandler is the pluggability mechanism for allowing implementations of or applications using the Jakarta 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 Jakarta Faces implementation must provide a default implementation of the *ViewHandler*

interface. See ViewHandler Property for information on replacing this default implementation with another implementation.

7.6.1. 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 Jakarta Faces request processing lifecycle for the specified *viewId* to be executed

```
public String getBookmarkableURL(FacesContext context, String viewId,
    Map<String, List<String>> parameters, boolean includeViewParams);
```

Return a Jakarta Faces action URL derived from the viewId argument that is suitable to be used as the target of a link in a Jakarta Faces response. The URL, if activated, would cause the browser to issue an initial request to the specified viewId

```
public String getRedirectURL(FacesContext context, String viewId,
    Map<String, List<String>> parameters, boolean includeViewParams);
```

Return a Jakarta Faces action URL derived from the *viewId* argument that is suitable to be used by the *NavigationHandler* to issue a redirect request to the URL using an initial request.

```
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 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 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.

```
public Set<String> getProtectedViewsUnmodifiable();
public void addProtectedView(String urlPattern);
public boolean removeProtectedView(String urlPattern)
```

See the javadocs for these methods for the specification.

7.6.2. 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 Facelets page. In the Facelets case, this is a context relative path to the XHTML page representing the view, such as */foo.xhtml*.

Jakarta Faces implementations must provide a default *ViewHandler* implementation, along with a default *ViewDeclarationLanguageFactory* implementation that vends *ViewDeclarationLanguage* implementation designed to support the rendering of Facelets pages containing Jakarta Faces components. The default *ViewHandler* is specified in this section and the default *ViewDeclarationLanguage* implementation is specified in the following section.

7.6.2.1. ViewHandler Methods that Derive Information From the Incoming Request

The *deriveViewId()* method must fulfill the following responsibilities:

- If the argument input is *null*, return *null*.
- If prefix mapping (such as “/faces/*”) is used for *FacesServlet*, normalize the *viewId* according to the following algorithm, or its semantic equivalent, and return it.
  - Remove any number of occurrences of the prefix mapping from the viewId. For example, if the incoming value was */faces/faces/faces/view.xhtml* the result would be simply *view.xhtml*.
- If suffix mapping (such as “*.faces”) is used for *FacesServlet*, the *viewId* is set using following algorithm.
  - Let *requestViewId* be the value of argument *input*.
  - 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.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 *faceletsDefaultSuffixes*.

    - For each entry in the list from *faceletsDefaultSuffixes*, replace the suffix of *requestViewId* with that entry. 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, if a physical resource exists with the name *requestViewId* let that value be *viewId*.
    - Otherwise return *null*.
- If an exact mapping (such as /foo) is used for FacesServlet, the *viewId* is set using following algorithm.
    - Let *requestViewId* be the value of the argument input.
    - 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 *faceletsDefaultSuffixes*.
    - For each entry in the list from *faceletsDefaultSuffixes*, add that current entry to the end of *requestViewId*. For discussion, call this *candidateViewId*. Look for a physical resource with the name *candidateViewId*. If such a resource exists, consider the algorithm complete with the result being *candidateViewId*.
    - Otherwise, if a physical resource exists with the name *requestViewId* let that value be *viewId*.
    - Otherwise return *null*.
- The getViewDeclarationLanguage() must fulfill the following responsibilites.
- See the javadocs for the normative specification for this method.

The *deriveLogicalViewId()* method is identical to *deriveViewId()* except that it does not check for the existence of the resource.

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 *jakarta.faces.application.ViewHandler.CHARACTER_ENCODING_KEY*. If a value is found, convert it to a String and return it.

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 8.3.2 "Resource Bundle Determination Algorithm" of the Jakarta Standard Tag Library Specification Document. If a match is found, return the corresponding *Locale* object.

- Otherwise, if the application has specified a default locale in the application configuration resources, return the corresponding *Locale* object.
- Otherwise, return the value returned by calling *Locale.getDefault()*.

The *calculateRenderKitId()* method must fulfill the following responsibilities:

- Return the value 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*.

7.6.2.2. ViewHandler Methods that are Called to Fill a Specific Role in the Lifecycle

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.

The *initView()* method must fulfill the following responsibilities:

- See the javadocs for this method for the specification.

The *renderView()* method must obtain a reference to the *ViewDeclarationLanguage* for the *viewId* of the argument *viewToRender* and call its *ViewDeclarationLanguage.restoreView()* method, returning the result and not swallowing any exceptions thrown by that method.

The *restoreView()* method must obtain a reference to the *ViewDeclarationLanguage* for the *viewId* of the argument *viewToRender* and call its *ViewDeclarationLanguage.restoreView()* method, returning the result and not swallowing any exceptions thrown by that method.

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.

7.6.2.3. ViewHandler Methods Relating to Navigation

The *getActionURL()* method must fulfill the following responsibilities:

- If the specified *viewId* does not start with a “/”, throw *IllegalArgumentException*.
- If exact mapping (such as /foo) is used for FacesServlet, the following algorithm must be followed to derive the result.
  - Retrieve the collection of existing mappings of the FacesServlet, e.g. using *ServletRegistration#getMappings()*. Let this be *facesServletMappings*. If the argument *viewId*

has an extension, then 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 *faceletsDefaultSuffixes*.

    - For each entry in the list from *faceletsDefaultSuffixes*, if the extension of the argument *viewId* is equal to this entry, remove the extension from *viewId*. For discussion, call this *candidateViewId*.
    - Look if the *candidateViewId* is present in *facesServletMappings*. If so,the result is *contextPath candidateViewId*.
    - If the argument *viewId* has no extension, then look if the *viewId* is present in *facesServletMappings*. If so, the result is *contextPath* + *viewId*.
    - If no result has been obtained, pick any prefix mapping or extension mapping from *facesServletMappings*. If no such mapping is found, throw an *IllegalStateException*.
    - If such mapping is found remove the "*" character from that mapping, take that as the new mapping and continue with evaluating this mapping as specified below for "if prefix mapping [...] is used" and for "if suffix mapping [...] is used

- 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.xhtml*".
- 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*"
- If the current view is one of the views to which view protection must be applied, the returned URL must contain the parameter with a name equal to the value of the constant defined by *ResponseStateManager.NON_POSTBACK_VIEW_TOKEN_PARAM*. The value of this parameter must be the return value from a call to *ResponseStateManager.getCryptographicallyStrongTokenFromSession()*. This parameter is inspected during the restore view phase (see Restore View).

The *getBookmarkableURL()* method must fulfill the following responsibilities:

- If argument *includeViewParams* is *true*, obtain the view paramaters corresponding to the argument *viewId* and append them to the *Map* given in argument *parameters*. Let the resultant *Map* be called *paramsToEncode*.
    - If the *viewId* of the current *FacesContext* is not equal to the argument *viewId*, get the *ViewDeclarationLanguage* for the argument *viewId*, obtain its *ViewMetadata*, call

createMetadataView() on it, then call *ViewMetadata.getViewParameters()* passing the return from *createMetadataView()*. Let the result of this method be *toViewParams*.

- If the *viewId* of the current *FacesContext* is equal to the argument *viewId*, call *ViewMetadata.getViewParameters()* passing the current *UIViewRoot*. Let the result of this method be *toViewParams*.
- If *toViewParams* is empty, take no further action to add view parameters to this URL. Iterate over each *UIViewParameter* element in *toViewParams* and take the following actions on each element.
- If the *Map* given by *parameters* has a key equal to the *name* property of the current element, take no action on the current element and continue iterating.
- If the current *UIViewParameter* has a *ValueExpression* under the key *"value"* (without the quotes), let *value* be the result of calling *getStringValueFromModel()* on the current *UIViewParameter*.
- Otherwise, if the current *viewId* is the same as the argument *viewId*, let *value* be the result of calling *getStringValue()* on the current *UIViewParameter*.
- Otherwise, if the current *viewId* is different from the argument *viewId*, locate the *UIViewParameter* instance in the current view whose name is equivalent to the current element and let *value* be the result of calling *getStringValue()* on the located *UIViewParameter*.
- If the above steps yielded a non-*null value*, find the *List<String>* value in the *parameters* map under the key given by the *name* property of the current *UIViewParameter* element. If such a *List* exists, add *value* to it. Otherwise create a *List<String>*, add *value* to it, and add it to the *parameters* map under the appropriate key.

- If argument *includeViewParams* is *false*, take no action to add additional entries to *paramaters*. Let *paramsToEncode* be *parameters*.
- Call *getActionURL()* on the argument *viewId*. Let the result be *actionEncodedViewId*.
- Call *encodeBookmarkableURL()* on the current *ExternalContext*, passing *actionEncodedViewId* as the first argument and *paramsToEncode* as the second. Let the result be *bookmarkEncodedURL*.
- Pass *bookmarkEncodedURL* to *ExternalContext.encodeActionURL()* and return the result.

The *getRedirectURL()* method must fulfill the following responsibilities:

- Take exactly the same action as in *getBookmarkableURL()* up to and including the call to *getActionURL()*. Thereafter take the following actions.
- Call *encodeRedirectURL()* on the current *ExternalContext*, passing *actionEncodedViewId* as the first argument and *paramsToEncode* as the second. Let the result be *redirectEncodedURL*.
- Pass *redirectEncodedURL* to *ExternalContext.encodeActionURL()* and return the result.

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.

7.6.2.4. ViewHandler Methods that relate to View Protection

- See the javadocs for *addProtectedView()* for the normative specification.
- See the javadocs for *removeProtectedView()* for the normative specification.
- See the javadocs for *getProtectedViewsUnmodifiable()* for the normative specification.

See the *View Protection* section within Restore View for the normative specification of this feature.

7.7. ViewDeclarationLanguage

To support the introduction of Facelets into the version 2 of the core specification, whilst preserving backwards compatibility with Jakarta Server Pages applications used with version 1 of the specification, 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 Jakarta Faces *UIComponent*s. Under this definition, Facelets is an example of an implementation of a VDL. Historically, Jakarta Server Pages was another example of an implementation of a VDL, but this has been deprecated in version 2 of the specification and removed in version 4 of the specification. 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.

The Facelets specific implementation is further detailed in Specification of the ViewDeclarationLanguage Implementation for Facelets for Jakarta Faces.

7.7.1. 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 FactoryFinder and the decoration mechanism specified in 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. The default implementation must return a valid *ViewDeclarationLanguage* instance for views written in Facelets. Whether the instance returned is the same for a Facelet view or another VDL is an implementation detail.

7.7.2. 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 views

Any implementation strategy is valid as long as these requirements are met.

7.7.2.1. ViewDeclarationLanguage.createView()

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

The *createView()* method must fulfill the following responsibilities.

- 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*, properties, 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)*.
- Pass the argument *viewId* to the *setViewId()* method on the new *UIViewRoot* instance.
- 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.
- Call *calculateResourceLibraryContracts()*, passing the argument *viewId*, and unconditionally set the result as the *resourceLibraryContracts* property on the *FacesContext*. The implementation must obtain the *ViewDeclarationLanguage* reference on which to invoke *calculateResourceLibraryContracts()* from the *ViewHandler*. This ensures the methods can be correctly decorated.
- Return the newly created *UIViewRoot*.

7.7.2.2. ViewDeclarationLanguage.calculateResourceLibraryContracts()

```
public List<String> calculateResourceLibraryContracts(
    FacesContext context, String viewId)
```

The implementation must examine the resource library contracts data structure, which was populated as specified in Resource Library Contracts, and find the <*contract-mapping*> element that matches the argument *viewId*. When processing the nested <*url-pattern*> matches must be made using the following rules in this order.

1. An exact match.
2. The longest match
3. The value * matches all incoming viewIds

The value returned from this method is the list whose contents are taken from the *contracts* attribute of the matching <*contract-mapping*> element.

7.7.2.3. ViewDeclarationLanguage.buildView()

```
public void buildView(FacesContext context, UIComponent root)
```

The *buildView()* method must fulfill the following responsibilities.

- 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.
- The runtime must guarantee that the view must be fully populated before the *afterPhase()* method of any *PhaseListeners* attached to the application or to the *UIViewRoot* (via *UIViewRoot.setAfterPhaseListener()* or *UIViewRoot.addPhaseListener()*) are called.
- The implementation must guarantee the markup comprising the view is executed with the UIComponent instances in the view being encountered in the same depth-first order as in other lifecycle methods defined on *UIComponent*, and added to the view (but not rendered at this time), during the traversal. .

7.7.2.4. ViewDeclarationLanguage.getComponentMetadata()

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

The *getComponentMetadata()* method must fulfill the following responsibilities:

- 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.
- Support argument *componentResource* being a Facelet markup file that is to be interpreted as a composite component as specified in Composite Component Metadata.

7.7.2.5. ViewDeclarationLanguage.getViewMetadata() and getViewParameters()

```
public ViewMetadata getViewMetadata(FacesContext context, String viewId)
```

The *getViewMetadata()* method must fulfill the following responsibilities:

- 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 implementation must support argument *viewId* being a Facelet markup file from which the view metadata should be extracted.

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>* or *<f:viewAction>* children. Each such element is the metadata will cause a *UIViewParameter* or *UIViewAction* (respectively) 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. Because *UIViewAction* implements *ActionSource*, it is valid to attach any of the kinds of attached objects to an *<f:viewAction>* that are valid for any element that represents any other kind of *ActionSource* in the view.

```
public Collection<UIViewParameter> getViewParameters(UIViewRoot root)
```

Convenience method that uses the view metadata specification above to obtain the *List<UIViewParameter>* for the argument viewId.

7.7.2.6. ViewDeclarationLanguage.getScriptComponentResource()

```
public Resource getScriptComponentResource(
    FacesContext context, Resource componentResource)
```

The *getScriptComponentResource()* method must fulfill the following responsibilities:

- 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*.

7.7.2.7. ViewDeclarationLanguage.renderView()

```
public void renderView(FacesContext context, String viewId)
```

The *renderView()* method must fulfill the following responsibilities:

- Return immediately if calling *isRendered()* on the argument *UIViewRoot* returns *false*.
- 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*.
- Call *encodeAll()* on the *UIViewRoot*.
- Call *endDocument()* on the *ResponseWriter*.
- Close the writer used to write the response.

7.7.2.8. ViewDeclarationLanguage.restoreView()

```
public UIViewRoot restoreView(FacesContext context, String viewId)
```

The *restoreView()* method must fulfill the following responsibilities:

- Call *ResponseStateManager.isStateless()*. If the result is *false*,
  - 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*.
- Otherwise, take the following steps and return.
  - Obtain a reference to the *ViewDeclarationLanguage* from the *ViewDeclarationLanguageFactory*. This is necessary to allow for proper decoration. It is not acceptable to simply use the java language *this* keyword.
  - Call *createView()* on the *ViewDeclarationLanguage* instance, passing the *context* and *viewId* arguments. Let *viewRoot* be the result.
  - Call *FacesContext.setViewRoot(viewRoot)*.
  - Call *buildView()* on the *ViewDeclarationLanguage*, passing the *context* and *viewRoot*.
  - Return the *viewRoot*.

7.8. 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 ResponseStateManager), that is provided by the *RenderKit* implementation, and is therefore aware of the markup language details. The Jakarta Faces implementation must provide a default *StateManager* implementation that supports the behavior described below.

7.8.1. 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 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 [7].

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.

7.8.1.1. Stateless Views

Version 2.2 of the specification adds support for stateless views. In such a view, the *UIComponent* state for the components is not saved. This feature must be used with full awareness of the statefulness requirements of the components in the view. If a component requires state to operate correctly, it must not be used in a stateless view. Furthermore, it is not required that *@ViewScoped* managed beans work at all with stateless views.

To mark a view as stateless, the existing *transient* property from *UIComponent* is exposed to the view author by means of the *transient* attribute on the *<f:view>* tag from the Faces Core tag library. The following spec sections contain more normative requirements for stateless views.

- The VDLDocs for the *<f:view>* tag.
- The javadocs for *ResponseStateManager.writeState(FacesContext, Object)*
- The javadocs for *ResponseStateManager.isStateless(FacesContext)*
- The specification of ViewDeclarationLanguage.restoreView()
- The javadocs for *jakarta.faces.view.ViewScoped*
- The javadocs for *jakarta.faces.bean.ViewScoped*

7.8.2. State Saving Alternatives and Implications

Jakarta Faces implementations support two primary mechanisms for saving state, based on the value of the *jakarta.faces.STATE_SAVING_METHOD* initialization parameter (see Application Configuration Parameters). The possible values for this parameter give a general indication of the approach to be used, while allowing Jakarta Faces 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 Jakarta Faces 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.The default implementation Serializes the view in *client* mode.
- *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. Serializing the view in server mode is optional but must be possible by setting the *context-param*

jakarta.faces.SERIALIZE_SERVER_STATE to *true*. 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.

Serializable in the preceding text means the values of all component attributes and properties (as well as the saved state of attached objects) must implement *java.io.Serializable* such that if the aggregate saved state were written to an *ObjectOutputStream*, a *NotSerializableException* would not be thrown.

7.8.3. State Saving Methods.

```
public Object saveView(FacesContext context);
```

This method causes the tree structure and component state of the view contained in the argument *FacesContext* to be collected, stored, and returned in a *java.lang.Object* instance that must implement *java.io.Serializable*. If *null* is returned from this method, there is no state to save.

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.

7.8.4. 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()*.

7.8.5. Convenience Methods

```
public boolean isSavingStateInClient(FacesContext context);
```

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.

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

Return the current view state as a String. This method must call ResposeStateManger.getViewState. Refer to ResponseStateManager for more details.

7.9. ResourceHandler

The normative specification for this class is in the javadoc for *jakarta.faces.application.ResourceHandler*. See also Resource Handling.

```
public ResourceHandler getResourceHandler();
public void setResourceHandler(ResourceHandler impl);
```

Chapter 8. Rendering Model

Jakarta 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 Jakarta Faces 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 Component Behavior Model for more details about this feature. Each Jakarta Faces implementation must provide a default *RenderKit* instance (named by the render kit identifier associated with the String constant *RenderKitFactory.HTML_BASIC_RENDER_KIT* as described below) that is utilized if no other *RenderKit* is selected.

```
public Renderer getRenderer(String family, String rendererType);
```

Return the *Renderer* instance corresponding to the specified component *family* and *rendererType* (if any), which will typically be the value of the *rendererType* property of a *UIComponent* about to be decoded or encoded

```
public 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 Jakarta Faces implementation may either choose to create their own *RenderKit* instances and register them with the *RenderKitFactory* instance (see 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 Jakarta Faces implementation, at application startup time See Example Application Configuration Resource for an example of a *faces-config.xml* configuration resource that defines two additional Renderer instances to be registered in the default *RenderKit*.

```
public ResponseWriter createResponseWriter(Writer writer,
    String contentTypeList, String characterEncoding);
```

Use the provided *Writer* to create a new *ResponseWriter* instance for the specified character encoding.

The *contentTypeList* parameter is an "Accept header style" list of content types for this response, or *null* if the *RenderKit* should choose the best fit. The *RenderKit* must support a value for the *contentTypeList* argument that comes straight from the *Accept* HTTP header, and therefore requires parsing according to the specification of the *Accept* header. Please see Section 14.1 of RFC 2616 (the HTTP 1.1 RFC) for the specification of the *Accept* header.

Implementors are advised to consult the *getCharacterEncoding()* method of class *jakarta.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 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 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 Jakarta Faces faces-config.xml or with an annotation.

XML Registration

```
<render-kit>
  <render-kit-id>HTML_BASIC</render-kit-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

Jakarta Faces provides the *jakarta.faces.render.FacesBehaviorRenderer annotation.*

```
@FacesClientBehaviorRenderer(value="Hello")
public class YourRenderer extends ClientBehaviorRenderer {
  ...
}
```

8.4. ResponseStateManager

ResponseStateManager is the helper class to *jakarta.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);
```

Please see the javadoc for this method for the normative specification.

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

Please see the javadoc for this method for the normative specification.

```
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.

```
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

A single instance of *jakarta.faces.render.RenderKitFactory* must be made available to each Jakarta Faces-based web application running in a servlet or portlet container. The factory instance can be acquired by Jakarta Faces 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 Jakarta Faces implementations.

Every Jakarta Faces implementation must provide a *RenderKit* instance for a default render kit identifier that is designated by the *String* constant *RenderKitFactory.HTML_BASIC_RENDER_KIT*. Additional render kit identifiers, and corresponding instances, can also be made available.

```
public Iterator<String> getRenderKitIds();
```

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

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

Register a *RenderKit* instance for the specified render kit identifier, replacing any previous RenderKit registered for that identifier.

8.6. Standard HTML RenderKit Implementation

To ensure application portability, all Jakarta Faces 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 Living Standard. Jakarta Faces implementors, and other parties, may also provide additional *RenderKit* libraries, or additional *Renderer*s that are added to the standard *RenderKit* at application startup time, but applications must ensure that the standard *Renderer*s are made available for the web application to utilize them.

The 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 Jakarta Faces.

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 *jakarta.faces.component.html* package. Each class in this package is a subclass of an corresponding class in the *jakarta.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* Component Behavior Model for additional details.

Table 7. Concrete HTML Component Classes

jakarta.faces.component class	renderer-type	jakarta.faces.component.html class
UICommand	jakarta.faces.Button	HtmlCommandButton
UICommand	jakarta.faces.Link	HtmlCommandLink
UIData	jakarta.faces.Table	HtmlDataTable
UIForm	jakarta.faces.Form	HtmlForm
UIGraphic	jakarta.faces.Image	HtmlGraphicImage
UIInput	jakarta.faces.Hidden	HtmlInputHidden
UIInput	jakarta.faces.Secret	HtmlInputSecret
UIInput	jakarta.faces.Text	HtmlInputText
UIInput	jakarta.faces.Textarea	HtmlInputTextarea
UIMessage	jakarta.faces.Message	HtmlMessage
UIMessages	jakarta.faces.Messages	HtmlMessages
UIOutput	jakarta.faces.Format	HtmlOutputFormat
UIOutput	jakarta.faces.Label	HtmlOutputLabel
UIOutput	jakarta.faces.Link	HtmlOutputLink
UIOutput	jakarta.faces.Text	HtmlOutputText
UIOutcomeTarget	jakarta.faces.Link	HtmlOutcomeTargetLink
UIOutcomeTarget	jakarta.faces.Button	HtmlOutcomeTargetButton
UIPanel	jakarta.faces.Grid	HtmlPanelGrid
UIPanel	jakarta.faces.Group	HtmlPanelGroup
UISelectBoolean	jakarta.faces.Checkbox	HtmlSelectBooleanCheckbox
UISelectMany	jakarta.faces.Checkbox	HtmlSelectManyCheckbox
UISelectMany	jakarta.faces.Listbox	HtmlSelectManyListbox
UISelectMany	jakarta.faces.Menu	HtmlSelectManyMenu
UISelectOne	jakarta.faces.Listbox	HtmlSelectOneListbox
UISelectOne	jakarta.faces.Menu	HtmlSelectOneMenu
UISelectOne	jakarta.faces.Radio	HtmlSelectOneRadio

As with the standard components in the *jakarta.faces.component* package, each HTML component implementation class must define a static public final String constant named *COMPONENT_TYPE*, whose value is "*jakarta.faces*". concatenated with the class name. HTML components, however, must not define a *COMPONENT_FAMILY* constant, or override the *getFamily()* method they inherit from their superclass.

Chapter 9. Standard Tag Libraries

This chapter describes the Standard Tag Libraries required by Jakarta Faces. This support is enabled by providing standard tags so that a Jakarta Faces user interface can be easy defined in a view by adding tags corresponding to Jakarta Faces UI components. Standard tags provided by a Jakarta Faces implementation may be mixed with and tags from other libraries, as well as template text for layout, in the same view.

Any Jakarta Faces implementation that claims compliance with this specification must include a complete Facelets implementation of the Standard Tag Libraries, and expose this implementation to the runtime of any Jakarta Faces application. Jakarta Faces applications, however, need not use Facelets as their View Declaration Language (VDL). In fact, a Jakarta Faces application is free to use whatever technology it likes for its VDL, as long as that VDL itself complies with the Jakarta Faces specification.

Facelets is specified in Facelets and its use in Web Applications.

The exact meaning of the term “page” is dependent on the VDL being used. In case of Facelets, that is usually the physical XHTML file representing the source code containing the declaration of the tags, or a class extending from *jakarta.faces.view.facelets.Facelet* having the *@View* annotation.

The term “view” represents the Jakarta Faces component tree hierarchy, as available by *FacesContext.getViewRoot()*, that represent the components created by the tags declared on the pages.

9.1. UIComponent Tags

A tag for a Jakarta Faces *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 8. Example Renderer Types

RendererType	Render-Dependent Attributes
“Text”	“size”
“Secret”	“size”, “secretChar”
“Textarea”	“size”, “rows”

The tag library descriptor file for the corresponding tag library, then, would define three tags—one per *Renderer* type. Below is an example of a portion of the tag definition for the *inputText* tag in the Facelet Taglib syntax [8]:

```
<tag>
  <tag-name>inputText</tag-name>
  <component>
    <component-type>HtmlInputText</component-type>
```

```
    <renderer-type>Text</renderer-type>
  </component>

  <attribute>
    <name>id</name>
    <required>false</required>
    <type>java.lang.String</type>
  </attribute>

  <attribute>
    <name>id</name>
    <required>false</required>
    <type>java.lang.Object</type>
  </attribute>

  <attribute>
    <name>size</name>
    <required>false</required>
    <type>java.lang.Integer</type>
  </attribute>
</tag>
```

Note that the *size* attribute is derived from the *Renderer* of type “Text”, while the *id* and *value* attributes are derived from the *UIInput* component class itself. *RenderKit* implementors will generally provide a Facelets tag library which includes component tags corresponding to each of the component classes (or types) supported by each of the *RenderKit*’s *Renderers*. See RenderKit and Renderer for details on the *RenderKit* and *Renderer* APIs. Jakarta Faces implementations must provide such a tag library for the standard HTML RenderKit (see Standard HTML RenderKit Tag Library).

9.2. Using UIComponent Tags

The following subsections define how a page author utilizes the tags provided by the *RenderKit* implementor in the views that create the user interface of a Jakarta Faces-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 defined by Jakarta Faces:

- *jakarta.faces.core* —URI for the *Jakarta Faces Core Tag Library*
- *jakarta.faces.html* —URI for the *Jakarta Faces Standard HTML RenderKit Tag Library*

The page author must declare the URI of each tag library to be utilized, as well as the prefix used to identify tags from this library. For example in Facelets XHTML syntax,

```
<!DOCTYPE html>
<html xmlns:f="jakarta.faces.core"
```

```
      xmlns:h="jakarta.faces.html">
  ...
</html>
```

declares the unique resource identifiers of the tag libraries being used, as well as the prefixes to be used within the root element for referencing tags from these libraries [9].

9.2.2. Including Components in a Page

A Jakarta Faces *UIComponent* tag can be placed at any desired position in a root element that contains the declaration for the corresponding tag library.

The following example illustrates the general use of a UIComponent tag in a page. In this scenario:

```
<h:inputText id="username" value="#{loginBacking.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 tag corresponds. If no *id* is specified, one may be automatically generated by the tag implementation if the tag implementation or functionality requires so.

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* tags is processed in the usual way. Therefore, you can use such markup to perform layout control, or include non-Jakarta Faces content, in conjunction with the tags that represent UI components.

9.2.3. Creating Components and Overriding Attributes

As *UIComponent* tags are encountered during the processing of a page, the tag 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 tag. The details of this process are as follows:

- If the component associated with this component tag has been identified already, return it unchanged.
- Identify the *component identifier* for the component related to this UIComponent tag, as follows:
  - If the page author has specified a value for the *id* attribute, use that value.
  - Otherwise, call the *createUniqueId()* method of the *UIViewRoot* at the root of the component tree for this view, and use that value.
- If this *UIComponent* tag is creating a *facet* (that is, we are nested inside an <*f:facet*> tag), determine if there is a facet of the component associated with our parent *UIComponent* tag, 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, and add the new component as a facet of the component associated with our parent *UIComponent* tag, under the specified facet name.

- If this *UIComponent* tag is not creating a facet (that is, we are not nested inside an *<f:facet>* tag), determine if there is a child component of the component associated with our parent *UIComponent* tag, 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, and add the new component as a child of the component associated with our parent *UIComponent* tag.

9.2.4. Deleting Components on Redisplay

In addition to the support for dynamically creating new components, as described above, *UIComponent* tags 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* tag that is nested inside a Jakarta Tags *<c:if>* tag 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* tag. 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* tags will generally mirror the hierarchical relationships of the corresponding *UIComponent* instances in the view that is associated with each page. For example, assume that a *UIForm* component (whose component id is *loginForm*) contains a *UIPanel* component used to manage the layout. You might specify the contents of the form like this in Facelets XHTML syntax:

```
<h:form id="loginForm">
  <fieldset>
    <legend>Login</legend>
    <p>
      <h:outputLabel for="username" value="Username:"/>
      <h:inputText id="username" value="#{loginBacking.username}"/>
    </p>
    <p>
      <h:outputLabel for="password" value="Password:"/>
      <h:inputSecret id="password" value="#{loginBacking.password}"/>
    </p>
```

```
    <p>
      <h:commandButton id="submit" action="#{loginBacking.login}"/>
      <h:commandButton id="reset" type="reset"/>
    </p>
  </fieldset>
</h:form>
```

9.2.6. Registering Converters, Event Listeners, and Validators

Each Jakarta Faces implementation is required to provide the core tag library (see Jakarta Faces Core Tag Library), which includes tags 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* tag.

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="#{loginBacking.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* tags.

```
<h:inputText id="maxUsers">
  <f:convertNumber integerOnly="true"/>
  <f:valueChangeListener type="com.example.YourValueChangeListener"/>
</h:inputText>
<h:commandButton value="Login">
  <f:actionListener type="com.example.YourActionListener"/>
</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* of the user specified type is instantiated and added to the enclosing *UICommand* component. If the user specified type could not be instantiated or does not implement the proper listener interface a *AbortProcessingException* 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 Facet Management. Facets can be defined in a page using the *<f:facet>* tag. Each facet tag must have one and only one child UIComponent tag [10]. 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 Jakarta Tags

It is permissible to use other tag libraries, such as the Jakarta Tags in the same view with *UIComponent* tags that correspond to Jakarta Faces components, subject to certain restrictions. When Jakarta Faces component tags are nested inside tags from other libraries, or combined with template text, the following behaviors must be supported:

- Jakarta Faces component tags nested inside a tag that conditionally renders its body (such as Jakarta Tags <*c:if*> or <*c:choose*>) must contain a manually assigned *id* attribute.
- Interoperation with the Jakarta Tag's Internationalization-Capable Formatting library (typically used with the "*fmt*" prefix) is restricted as follows:
  - The <*fmt:parseDate*> and <*fmt:parseNumber*> tags should not be used. The corresponding Jakarta Faces facility is to use an <*h:inputText*> component tag with an appropriate *DateTimeConverter* or *NumberConverter*.
  - The <*fmt:requestEncoding*> tag 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. Jakarta Faces handles character set issues automatically in most cases.
  - The <*fmt:setLocale/*> tag should not be used. Even though it might work in some circumstances, it would result in Jakarta Faces and Jakarta Tags assuming different locales. If the two locales use different character sets, the results will be undefined. Applications should use Jakarta Faces facilities for setting the *locale* property on the *UIViewRoot* component to change locales for a particular user.

9.3. UIComponent tag Implementation Requirements

The tag implementation classes for *UIComponent* tags must conform to all of the requirements defined in the specification specific to the VDL. In addition, they must meet the following Jakarta Faces-specific requirement:

- 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 *IllegalStateException*.

9.4. Jakarta Faces Core Tag Library

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

- Must declare a URI (<*uri*>) value of *jakarta.faces.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 specific to the VDL (see Facelet Tag Library mechanism for the Facelets Taglib example).

The tags in the implementation of this tag library must not cause JavaScript of CSS to be rendered to the client. Doing so would break the requirement that the Jakarta Faces Core Tag library is independent of any specific RenderKit.

Each tag included in the Jakarta Faces Core Tag Library is documented in the VDLDocs for the *f:* tag library.

9.5. Standard HTML RenderKit Tag Library

All Jakarta Faces implementations must provide a tag library containing tags that correspond to each valid combination of a supported component class (see Standard User Interface Components") and a *Renderer* from the Standard HTML RenderKit (see Standard HTML RenderKit Implementation) that supports that component type. The tag library descriptor for this tag library must meet the following requirements:

- Must declare a URI (<*uri*>) value of *jakarta.faces.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 specific to the VDL (see Facelet Tag Library mechanism for the Facelets Taglib example).

The tags defined in this tag library must specify the following return values for the *getComponentType()* and *getRendererType()* methods, respectively:

Table 9. Standard HTML RenderKit Tag Library

getComponentType()	getRendererType()	tag name
jakarta.faces.Column	*null* [11]	column
jakarta.faces.HtmlCommandButton	jakarta.faces.Button	commandButton
jakarta.faces.HtmlCommandLink	jakarta.faces.Link	commandLink
jakarta.faces.HtmlCommandScript	jakarta.faces.Script	commandScript
jakarta.faces.HtmlDataTable	jakarta.faces.Table	dataTable
jakarta.faces.HtmlForm	jakarta.faces.Form	form

getComponentType()	getRendererType()	tag name
jakarta.faces.HtmlGraphicImage	jakarta.faces.Image	graphicImage
jakarta.faces.HtmlInputHidden	jakarta.faces.Hidden	inputHidden
jakarta.faces.HtmlInputSecret	jakarta.faces.Secret	inputSecret
jakarta.faces.HtmlInputText	jakarta.faces.Text	inputText
jakarta.faces.HtmlInputTextarea	jakarta.faces.Textarea	inputTextarea
jakarta.faces.HtmlMessage	jakarta.faces.Message	message
jakarta.faces.HtmlMessages	jakarta.faces.Messages	messages
jakarta.faces.HtmlOutputFormat	jakarta.faces.Format	outputFormat
jakarta.faces.HtmlOutputLabel	jakarta.faces.Label	outputLabel
jakarta.faces.HtmlOutputLink	jakarta.faces.Link	outputLink
jakarta.faces.Output	jakarta.faces.Body	body
jakarta.faces.Output	jakarta.faces.Head	head
jakarta.faces.Output	jakarta.faces.resource.Script	outputScript
jakarta.faces.Output	jakarta.faces.resource.Stylesheet	outputStylesheet
jakarta.faces.HtmlOutputText	jakarta.faces.Text	outputText
jakarta.faces.HtmlPanelGrid	jakarta.faces.Grid	panelGrid
jakarta.faces.HtmlPanelGroup	jakarta.faces.Group	panelGroup
jakarta.faces.HtmlSelectBooleanCheckbox	jakarta.faces.Checkbox	selectBooleanCheckbox
jakarta.faces.HtmlSelectManyCheckbox	jakarta.faces.Checkbox	selectManyCheckbox
jakarta.faces.HtmlSelectManyListbox	jakarta.faces.Listbox	selectManyListbox
jakarta.faces.HtmlSelectManyMenu	jakarta.faces.Menu	selectManyMenu
jakarta.faces.HtmlSelectOneListbox	jakarta.faces.Listbox	selectOneListbox
jakarta.faces.HtmlSelectOneMenu	jakarta.faces.Menu	selectOneMenu
jakarta.faces.HtmlSelectOneRadio	jakarta.faces.Radio	selectOneRadio

Each tag included in the Standard HTML RenderKit Tag Library is documented in the VDLDocs for the *h:* tag library.

Chapter 10. Facelets and its use in Web Applications

Jakarta Faces implementations must support (although Jakarta Faces-based applications need not utilize) using Facelets as the view declaration language for Jakarta Faces 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 Jakarta Server Pages that was designed from the outset with Jakarta Faces in mind. See ViewDeclarationLanguage.

10.1.1. Differences between Jakarta Server Pages and Facelets

Facelets was the first non-Jakarta Server Pages view declaration language designed for Jakarta Faces. As such, Facelets was able to provide a simpler and more powerful programming model to Jakarta Faces developers than that provided by Jakarta Server Pages, largely by leveraging Jakarta Faces as much as possible without carrying backwards compatibility with Jakarta Server Pages. The following table lists some of the differences between Facelets and Jakarta Server Pages

Table 10. Comparison of Facelets and Jakarta Server Pages

Feature Name	Jakarta Server Pages	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 Jakarta Server Pages	Page templating is a core feature of Facelets
Performance	Due to the common implementation technique of compiling a Jakarta Server Pages page to a Servlet, performance can be slow	Facelets is simpler and faster than Jakarta Server Pages
Expression Language Expressions	Expressions in template text cause unexpected behavior when used in Jakarta Server Pages	Expressions in template text operate as expected.

Feature Name	Jakarta Server Pages	Facelets
Jakarta Standard	Yes, the specification is separate from the implementation for Jakarta Server Pages	No, the specification is defined by and is one with the implementation.

10.1.2. Resource Library Contracts Background

Jakarta Faces defines a system called “resource library contracts” for applying facelet templates to an entire application in a re-usable and interchangeable manner. The feature is built on top of the resource library facility described in Libraries of Localized and Versioned Resources. A configurable set of Facelet VDL views in the application will be able to declare themselves to be template-clients of any template in a resource library contract. Facelet VDL views in the application can also make use of resources contained in a resource library contract, but the feature has ample value when only used with templates.

10.1.2.1. Non-normative Example

Consider this resource library contract, called *siteLayout*.

```
siteLayout/
  topNav_template.xhtml
  leftNav_foo.xhtml
  styles.css
  script.js
  background.png
```

This simple example takes advantage of several conventions built into the feature, most notably the default application of all available contracts in the application to all views in the application. It is possible to customize how resource library contracts are applied to the application, including using several different contracts in the same or different parts of the application. Such customizing is accomplished by including a <*resource-library-contracts*> element within the <*application*> element of the *faces-config.xml* (or similar) file. Because this example is designed with the convention in mind, it does not need a *faces-config.xml* file.

The *siteLayout* contract offers two templates: *topNav_template.xhtml* and *leftNav_foo.xhtml*. For discussion, these are known as “declared templates”. When used by a template client, they will lay out the template client’s contents with a navigation menu on the top or the left side of the page, respectively. In *siteLayout*, each of the templates has <*ui:insert*> tags named “title”, “content”, and “nav”. For discussion, these are knows as “declared insertion points”. Furthermore, each of the templates uses the CSS styles declared in *styles.css*, some scripts defined in *script.js*, and the background image *background.png*. For discussion, these are known as “declared resources”. In order to use a resource library contract, one must know its declared templates, their declared insertion points, and, optionally, their declared resources. No constraint is placed on the naming and arrangement of declared templates, insertion points, or resources, but all three concepts together can informally be thought of as the declaration of the resource library contract. The contract declaration of *siteLayout* can be stated as follows.

siteLayout provides two declared templates, *topNav_template.xhtml* and *leftNav_foo.xhtml*. Each templates offers declared insertion points “title”, “content”, and “nav”.

In this case, the css, script, and image are left out of the contract declaration but this distinction is completely arbitrary. The important content of *topNav_template.xhtml* is shown next.

```
<!DOCTYPE html>
<html xmlns:ui="jakarta.faces.facelets"
      xmlns:h="jakarta.faces.html">
  <h:head>
    <h:outputStylesheet id="default" name="default.css" />
    <h:outputStylesheet name="cssLayout.css" />
    <title><ui:insert name="title"></ui:insert></title>
  </h:head>
  <h:body>
    <div id="top" class="top">
      <p>Top Navigation Menu</p>
      <ui:insert name="nav">Nav content</ui:insert>
    </div>
    <div id="content" class="center_content">
      <ui:insert name="content">Content</ui:insert>
    </div>
  </h:body>
</html>
```

This example packages the entire *siteLayout* directory and its contents into the *META-INF/contracts* entry of a JAR file named *siteLayout.jar*. The simplest possible way to use *siteLayout* is to drop *siteLayout.jar* into *WEB-INF/lib* and apply the knowledge of the resource library contract declaration to the facelet views in the app.

Consider this simple web app, called *useContract*, the file layout for which is shown next. The example is shown using a simplified maven war packaging.

```
useContract/
  pom.xml
  src/main/webapp/
             /WEB-INF/lib/siteLayout.jar
             index.xhtml
             page2.xhtml
```

Notice the absence of a *faces-config.xml* file. Because this example is content to let all the contracts in *siteLayout.jar* be applied to all views in the app, this file is not necessary. The two pages are shown next.

index.xhtml.

```
<ui:composition template="/topNav_template.xhtml"
                xmlns:ui="jakarta.faces.facelets"
```

```
                xmlns:h="jakarta.faces.html">
  <ui:define name="title">#{msgs.contactsWindowTitle}</ui:define>
  <ui:define name="content">
    <h:commandButton value="next" action="page2" />
  </ui:define>
  <ui:define name="nav">#{msgs.contactsNavMessage}</ui:define>
</ui:composition>
```

page2.xhtml

```
<ui:composition template="/leftNav_foo.xhtml"
                xmlns:ui="jakarta.faces.facelets"
                xmlns:h="jakarta.faces.html">
  <ui:define name="title">Hard coded title</ui:define>
  <ui:define name="content">
    <h:commandButton value="back" action="index" />
  </ui:define>
  <ui:define name="nav">Hard coded nav</ui:define>
</ui:composition>
```

To complete the example, the execution of the *useContract* app is illustrated.

When *useContract.war* is deployed, the runtime will discover that *siteLayout.jar* is a resource library contract and make its contents available for template clients.

When the user agent visits *http://localhost:8080/useContract/faces/index.xhtml*, because the *siteLayout* resource library contract provides */topNav_template.xhtml* , that file will be loaded as the template. Likewise, when the *next* button is pressed, */leftNav_foo.xhtml*, also from *siteLayout*, will be loaded as the template.

Now, consider there is an alternate implementation of the *siteLayout* contract, packaged as *newSiteLayout.jar*. This implementation doesn't change the contract declaration, but completely changes the arrangement and style of the views. As long as the contract declaration does not change, *useContract* can take advantage of *newSiteLayout* simply by replacing one JAR in *WEB-INF/lib*.

10.1.2.2. Non-normative Feature Overview

The normative requirements of the feature are stated in the context of the part of the specification impacted. This section gives the reader a non-normative overview of the feature that touches on all the parts of the specification that intersect with this feature.

Design Time

At design time, the developer has packaged any resource library contracts to be used in the application in the right place in the web application, or JAR file classpath. *This behavior is normatively specified in Resource Library Contracts*.

Startup Time

At startup time, the runtime will discover the set of resource library contracts available for this

application. If there is one or more *<resource-library-contracts>* element, only those contracts explicitly named will be made available for use in the application. If there is no such element, all of the discovered contracts are made available for use in the application. *This behavior is normatively specified in Resource Library Contracts and in the XML schema for the application configuration resources.*

Facelet Processing Time

The specification for *ViewDeclarationLanguage.createView()* requires a call to *ViewDeclarationLanguage.calculateResourceLibraryContracts()*, passing the current *viewId*. This method will examine the data structure assembled at startup and return a *List<String>* representing the resource library contracts eligible for use in this view. This value is set as the value of the *resourceLibraryContracts* property on the *FacesContext*. *This behavior is normatively specified in ViewDeclarationLanguage.createView()*.

The specification of the tag handler for *<f:view>* is the one other place where the *resourceLibraryContracts* property may be set. *This behavior is normatively specified in the tag handler for <f:view>*.

In any *<ui:composition>* or *<ui:decorate>* tag reached from that view, it is valid to use any of the templates in any of the listed contracts as the value of the *template* attribute. This behavior happens naturally as a side effect of the requirements of *ResourceHandler.createViewResource()*, where the implementation of that method is required to first consult the *resourceLibraryContracts* property of the current *FacesContext*. If the value of the property is non-*null* and non empty, the implementation must first look for the named view resource within each of the contracts in the list, and return the first matching one found. Otherwise, the implementation just returns the matching resource, if found. *This behavior is normatively specified in the javadoc for ResourceHandler.createViewResource()*.

View Rendering Time

When the view is being rendered, any resources that reside in a resource library contract will have additional metadata so that a subsequent request from the user agent is able to quickly find the resource inside the named contract. *This behavior is normatively specified in the javadoc for Resource.getRequestPath()*.

User-Agent Rendering Time

By the point in time that the User-Agent is rendering the view, all of the work related to resource library contracts will have been completed, but it is worth mentioning that any resources in the page that originate from within resource library contracts will be correctly fetched.

10.1.3. HTML5 Friendly Markup

Without the HTML5 Friendly Markup feature the view authoring model relies entirely on the concept of a Jakarta Faces UI component in a view as a means to encapsulate arbitrarily complex web user interface code behind a simple UI component tag in a page. For example, the act of including *<my:datePicker value="#{user.dob}">* in a view could cause a large amount of HTML, CSS, JavaScript, and images to be delivered to the user agent. This abstraction is very appropriate when the view author is content to delegate the work of designing the user experience for such components to a component author. As web designer skills have become more widespread, the

need has arisen to expose the hitherto hidden complexity so the view author has near total control on the user experience of each individual element in the view. The HTML5 Friendly Markup feature addresses this requirement, as well as providing access to the loosened attribute syntax also present in HTML5.

10.1.3.1. Non-normative Feature Overview

The normative requirements of the feature are stated in the context of the part of the specification impacted. This section gives the reader a non-normative overview of the feature that touches on all the parts of the specification that intersect with this feature. There are two main aspects to the feature, pass through attributes and pass through elements.

Pass Through Attributes

For any given Jakarta Faces component tag in a view, the set of available attributes that component supports is determined by a combination of the *UIComponent* and *Renderer* for that tag. In some cases the value of the attribute is interpreted by the *UIComponent* or *Renderer* (for example, the *columns* attribute of *h:panelGrid*) and in others the value is passed straight through to the user agent (for example, the *lang* attribute of *h:inputText*). In both cases, the *UIComponent/Renderer* has a priori knowledge of the set of allowable attributes. *Pass Through Attributes* allows the view author to list arbitrary name value pairs that are passed straight through to the user agent without interpretation by the *UIComponent/Renderer*. *This behavior is normatively specified in the "Rendering Pass Through Attributes" section of the overview of the standard HTML_BASIC render kit.*

The view author may specify pass through attributes in three ways.

- Nesting the <*f:passThroughAttribute*> tag within a *UIComponent* tag. For example,

```
<h:inputText value="#{user.name}">
  <f:passThroughAttribute name="autofocus" value="#{config.autofocus}" />
</h:inputText>
```

- Nesting the <*f:passThroughAttributes*> tag within a *UIComponent* tag, For example,

```
<h:inputText value="#{user.name}">
  <f:passThroughAttributes value="#{config.inputTextAttributes}" />
</h:inputText>
```

  The Jakarta Expression Language expression must point to a *Map<String, Object>*. If the value is a *ValueExpresison* call *getValue()* the value first. Whether the value is a *ValueExpression* or not, the value must have its *toString()* called on it.

- Prefixing the attribute with the shortname assigned to the *jakarta.faces.passthrough* XML namespace. For example

```
<!DOCTYPE html>
<html xmlns:h="jakarta.faces.html"
      xmlns:pt="jakarta.faces.passthrough">
```

```
  <h:body>
    <h:form>
      <h:inputText pt:autofocus="#{config.autofocus}" value="#{user.name}" />
    </h:form>
  </h:body>
</html>
```

This behavior is normatively specified in the VDLdoc for <f:passthroughAttribute>, <f:passThroughAttributes> tags in the "Faces Core" tag library, and the "Pass Through Attributes" tag library.

Pass Through Elements

This feature circumvents the traditional component abstraction model of Jakarta Faces, allowing the page author nearly complete control of the rendered markup, without sacrificing any of the server side lifecycle offered by Jakarta Faces. This is accomplished by means of enhancements to the Facelet *TagDecorator* API. This API describes a mapping from the common markup elements to target tags in the HTML_BASIC RenderKit such that the actual markup specified by the view author is what gets rendered, but the server side component is an actual component from the HTML_BASIC RenderKit. A special *Renderer* is provided to cover cases when none of the mappings specified in *TagDecorator* fit the incoming markup. To allow further flexibility, the existing Facelets TagDecorator mechanism allows complete control of the mapping process. *This behavior is normatively specified in the javadocs for class jakarta.faces.view.facelets.TagDecorator and in the section "Rendering Pass Through Attributes" in the "General Notes On Encoding" in the Standard HTML_BASIC RenderKit.*

An example will illustrate the mapping process.

```
<!DOCTYPE HTML>
<html xmlns:faces="jakarta.faces"
      xmlns:h="jakarta.faces.html">
  <h:body>
    <h:form>
      <input type="number" pattern="[0-9]*" faces:value="#{user.age}" />
    </h:form>
  </h:body>
</html>
```

As required in Specification of the ViewDeclarationLanguage Implementation for Facelets for Jakarta Faces *TagDecorator* is called during the facelet processing. Because the <*input*> element has an attribute from the *jakarta.faces* namespace, the system treats the element as a pass through element. The table listed in the javadocs for *TagDecorator* is consulted and it is determined that this component should act as an <*h:inputText*> component for the purposes of postback processing. However, the rendering is entirely taken from the markup in the facelet view. Another example illustrates the special *Renderer* that is used when no mapping can be found in the table in the javadocs for *TagDecorator*.

```
<!DOCTYPE HTML>
```

```
<html xmlns:faces="jakarta.faces"
      xmlns:h="jakarta.faces.html">
  <h:body>
    <meter faces:id="meter" min="#{bean.min}" max="#{bean.max}" value="350">
      350 degrees
    </meter>
  </h:body>
</html>
```

As in the preceding example, the *TagDecorator* mechanism is activated but it is determined that this component should act as a *<faces:element>* component for the purposes of postback processing. *The behavior of the <faces:element> is normatively specified in the VDLdoc for that tag. The behavior of the jakarta.faces.passthrough.Element renderer is normatively specified in the RenderKitDoc for that renderer*.

10.2. Java Programming Language Specification for Facelets in Jakarta Faces

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 Jakarta Faces, will ensure applications and Jakarta Faces component libraries that make use of Facelets are portable across different implementations of Jakarta Faces.

The original Facelet project did not separate the API and the implementation into separate jars, as is common practice with specifications. Thus, a significant task for integrating Facelets into Jakarta Faces 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 Jakarta Faces.

- The original decision in pre-Jakarta Faces JSF 1.0 (under the JCP) to allow the ViewHandler to be pluggable enabled the concept of a View Declaration Language for Jakarta Faces. 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 View Declaration Language.
- After polling users of Facelets, the JCP 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 Jakarta Faces as small as possible, only exposing classes where absolutely necessary.

The application of these principles produced the classes in the package *jakarta.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 Jakarta Faces

As normatively specified in the javadocs for *ViewDeclarationLanguageFactory.getViewDeclarationLanguage()*, a Jakarta Faces implementation must guarantee that a valid and functional *ViewDeclarationLanguage* instance is returned from this method when the argument is a reference to 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, the view must still be re-built, but care must be taken to ensure that the existing components are correctly paired up with their VDL counterparts in the VDL page. The implementation must examine the *viewId* of the argument root, which must resolve to an entity written in Facelets for Jakarta Faces markup language. Because Facelets 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 *jakarta.faces.view.facelets.FaceletHandler*, or a subinterface or subclass thereof, and an instance of *jakarta.faces.view.facelets.TagConfig*, or a subinterface or subclass thereof, which is passed to the constructor of the object implementing *FaceletHandler*.

When constructing the *TagConfig* implementation to be passed to the *FaceletHandler* implementation, the runtime must ensure that the instance returned from *TagConfig.getTag()* has been passed through the tag decoration process as described in the javadocs for *jakarta.faces.view.facelets.TagDecorator* prior to the *TagConfig* being passed to the *FaceletHandler* implementation.

The mapping between the categories of elements in the XHTML view and the appropriate sub-interface 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 Jakarta Faces. Each *FaceletHandler* instance must use the *getNextHandler()* method of the *TagConfig* instance passed to its constructor 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* which must be considered the default namespace. I.e. the Facelets page authors should not have the need to explicitly declare this namespace. Such elements should be passed through as is to the rendered output.
    - These elements correspond to instances of *jakarta.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 *jakarta.faces.html*,

or from the namespace of a custom tag library (including composite components) as described in Facelet Tag Library mechanism.

    - These elements correspond to instances of *jakarta.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 Jakarta Faces Core namespace *jakarta.faces.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 *jakarta.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 *jakarta.faces.component.FacetHandler*.

    - Such elements that represent an attribute that must be pushed into the parent *UIComponent* element must correspond to an instance of *jakarta.facelets.view.facelets.AttributeHandler*.

- Markup Elements that indicate facelet templating, as specified in the VDL Docs for the namespace *jakarta.faces.facelets*.

    - Such elements correspond to an instance of *jakarta.faces.view.facelets.TagHandler*.

- Markup elements from the Facelet version of the Jakarta Tags namespaces *jakarta.tags.core* or *jakarta.tags.functions*, as specified in the VDL Docs for those namespaces.

    - Such elements correspond to an instance of *jakarta.faces.view.facelets.TagHandler*.

10.3. XHTML Specification for Facelets for Jakarta Faces

10.3.1. General Requirements

Facelet pages are authored in XHTML. The runtime must support all XHTML pages that conform to the XHTML-1.0-Transitional DTD, as described at *http://www.w3.org/TR/xhtml1/#a_dtd_XHTML-1.0-Transitional*.

The runtime must ensure that Jakarta Expression Language expressions that appear in the page without being the right-hand-side of a tag attribute are treated as if they appeared on the right-hand-side of the *value* attribute of an *<h:outputText>* element in the *jakarta.faces.html* namespace. This behavior must happen regardless of whether or not the *jakarta.faces.html* namespace has been declared in the page.

10.3.1.1. DOCTYPE and XML Declaration

When processing Facelet VDL files, the system must ensure that at most one XML declaration and at most one DOCTYPE declaration appear in the rendered markup, if and only if there is corresponding markup in the Facelet VDL files for those elements. If multiple occurrences of XML declaration and DOCTYPE declaration are encountered when processing Facelet VDL files, the “outer-most” occurrence is the one that must be rendered. If an XML declaration is present, it must be the very first markup rendered, and it must precede any DOCTYPE declaration (if present). The

output of the XML and DOCTYPE declarations are subject to the configuration options listed in the table titled “Valid <process-as> values and their implications on the processing of Facelet VDL files” in The facelets-processing element.

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 Jakarta Server Pages. 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 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 *UIComponent*s into a tag library and expose them to page authors for use in their pages.

The runtime must support the following syntax for making the tags in a tag library available for use in a Facelet page.

```
<html 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="jakarta.faces.html"* within the <*html*> element in a Facelet XHTML page would cause the runtime to make all tags declared in Standard HTML RenderKit Tag Library to be available for use in the page using syntax like: <*h:inputText*>.

The unprefixed namespace, also known as the root namespace, must be passed through without modification or check for validity. The passing through of the root namespace must occur on any non-prefixed element in a facelet page. For example, the following markup declaration:.

```
<!DOCTYPE html>
<html xmlns:h="jakarta.faces.html">
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <msup>
      <msqrt>
        <mrow>
          <mi>a</mi>
          <mo>+</mo>

          <mi>b</mi>
        </mrow>
      </msqrt>
    <mn>27</mn>
  </msup>
```

```
</math>
```

would be rendered as

```
<!DOCTYPE html>
<html>
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <msup>
      <msqrt>
        <mrow>
          <mi>a</mi>
          <mo>+</mo>

          <mi>b</mi>
        </mrow>
      </msqrt>
    <mn>27</mn>
  </msup>
</math>
```

The run time must support two modes of discovery for Facelet tag library descriptors

- Via declaration in the web.xml, as specified in 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.

The specification for how to interpret a facelet tag library descriptor is included in the documentation elements of the schema for such files, see XML Schema Definition For Facelet Taglib.

10.3.3. Requirements specific to composite components

The text in this section makes use of the terms defined in 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

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 "*jakarta.faces.composite/*" (without the quotes), the remainder of the namespace declaration is taken as the name of a resource library as described in Libraries of Localized and Versioned Resources, as shown in the following example:

```
<html xmlns:ez="jakarta.faces.composite/ezcomp">
```

The runtime must look for a resource library named *ezcomp*. If the substring following "*jakarta.faces.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 facelet taglibrary schema, 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 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.

10.3.3.2. Creating an instance of a *top level component*

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 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 *jakarta.faces.Panel* and renderer-type *jakarta.faces.Group* to be installed as a facet of the *top level component* under the facet name *UIComponent.COMPOSITE_FACET_NAME*.

10.3.3.3. Populating a *top level component* instance with children

As specified in How does one make a composite component? the runtime must support the use of *cc:* tag library in the *defining page* pointed to by the *Resource* derived as specified in Creating an instance of a top level component. 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*.

Please see the tag library documentation for the <*cc:insertChildren*> and <*cc: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*. 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.

For code that handles tag attributes on *UIComponent* XHTML elements special action must be taken regarding composite components. 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 Jakarta Expression Language expression string starts with the *cc* implicit object, is followed by the special string "*attrs*" (without the quotes), as specified in 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.

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*. The actions taken in these methods set the stage for the tag attribute behavior and the special *MethodExpression* handling behavior described previously.

The runtime must support the inclusion of composite components within the *composite component definition*. .

10.4. Standard Facelet Tag Libraries

This section specifies the tag libraries that must be provided by an implementation.

10.4.1. Jakarta Faces Core Tag Library

This tag library must be equivalent to the one specified in Jakarta Faces Core Tag Library. The specification for this library can be found in the VDLDocs for the *f:* tag library.

10.4.2. Standard HTML RenderKit Tag Library

This tag library must be equivalent to the one specified in Standard HTML RenderKit Tag Library. The specification for this library can be found in the VDLDocs for the *h:* tag library.

10.4.3. Facelet Templating Tag Library

This tag library is the specified version of the *ui:* tag library found in Facelets. The specification for this library can be found in the VDLDocs for the *ui:* tag 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 *cc:* tag library.

10.4.5. Jakarta Tags Core and Function Tag Libraries

Facelets exposes a subset of the Jakarta Tags Core tag library and the entirety of the Jakarta Tags Function tag library. Please see the VDLDocs for the *jstl-core:* and *jstl-fn:* tag libraries for the normative specification.

10.5. Assertions relating to the construction of the view

hierarchy

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.

Chapter 11. Using Jakarta Faces in Web Applications

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

11.1. Web Application Deployment Descriptor

Jakarta Faces-based applications are *web applications* that conform to the requirements of the *Jakarta Servlet Specification* (version 5.0 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 Jakarta Faces-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 Jakarta Faces implementations must support the functionality described in this chapter to ensure application portability.

11.1.1. Servlet Definition

Jakarta Faces implementations must provide request processing lifecycle services through a standard servlet, defined by this specification. This servlet must be defined, in the deployment descriptor of an application that wishes to employ this portable mechanism, as follows:

```
<servlet>

 <servlet-name> faces-servlet-name </servlet-name>

 <servlet-class> jakarta.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 Servlet Mapping).

In addition to *FacesServlet*, Jakarta Faces implementations may support other ways to invoke the Jakarta 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 *Jakarta Servlet Specification*) against the portion of the request URL after the context path that selected this web application. Jakarta Faces implementations must support web application that define a <*servlet-mapping*> that maps any valid *url-pattern* to the *FacesServlet*. Prefix or extension mapping may be used. When using prefix mapping, the following mapping is recommended, but not required:

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

When using extension mapping the following mapping is recommended, but not required:

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

In addition to *FacesServlet*, Jakarta Faces implementations may support other ways to invoke the Jakarta 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. All Jakarta Faces implementations are required to support the following application configuration parameter names:

- *jakarta.faces.ALWAYS_PERFORM_VALIDATION_WHEN_REQUIRED_IS_TRUE* —See the javadocs for the constant *jakarta.faces.component.UIInput.ALWAYS_PERFORM_VALIDATION_WHEN_REQUIRED_IS_TRUE* for the specification of this feature.
- *jakarta.faces.CLIENT_WINDOW_MODE* —The context-param that controls the operation of the

ClientWindow feature. See the javadocs for the constant *jakarta.faces.lifecycle.ClientWindow.CLIENT_WINDOW_MODE_PARAM_NAME* for the specification of this feature.

- *jakarta.faces.CONFIG_FILES* — Comma-delimited list of context-relative resource paths under which the Jakarta Faces implementation will look for application configuration resources (see 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.
- *jakarta.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 *jakarta.faces.convert.DateTimeConverter* instances must be equal to *TimeZone.getDefault()* instead of "GMT".
- *jakarta.faces.DISABLE_FACESSERVLET_TO_XHTML* — If this param is set, and calling *toLowerCase().equals("true")* on a *String* representation of its value returns *true*, the default mapping of the *FacesServlet* to **.xhtml* must not take effect.
- *jakarta.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 facelet taglibrary schema and expose the tags therein according to Facelet Tag Library mechanism.
- *jakarta.faces.FACELETS_BUFFER_SIZE* —The buffer size to set on the response when the *ResponseWriter* is generated. By default the value is 1024. A value of -1 will not assign a buffer size on the response. This should be increased if you are using development mode in order to guarantee that the response isn't partially rendered when an error is generated.
- *jakarta.faces.FACELETS_DECORATORS* —A semicolon (;) delimitted list of class names of type *jakarta.faces.view.facelets.TagDecorator*, with a no-argument constructor. These decorators will be loaded when the first request for a Facelets VDL view hits the *ViewHandler* for page compilation.
- *jakarta.faces.FACELETS_REFRESH_PERIOD* — When a page is requested, what interval in seconds should the compiler check for changes. If you don't want the compiler to check for changes once the page is compiled, then use a value of -1. Setting a low refresh period helps during development to be able to edit pages in a running application. If this value is not specified, then the default depends on *Application.getProjectStage()*. If it is *Production*, then runtime must act as if it is set to -1, else the runtime must act as if it is set to 0.
- *jakarta.faces.FACELETS_RESOURCE_RESOLVER* — If this param is set, the runtime must interpret its value as a fully qualified classname of a java class that extends *jakarta.faces.view.facelets.ResourceResolver* and has a zero argument public constructor or a one argument public constructor where the type of the argument is *ResourceResolver*. If this param is set and its value does not conform to those requirements, the runtime must log a message and continue. If it does conform to these requirements and has a one-argument constructor, the default *ResourceResolver* must be passed to the constructor. If it has a zero argument constructor it is invoked directly. In either case, the new *ResourceResolver* replaces the old one.

Related to this param is the corresponding annotation, *jakarta.faces.view.facelets.FaceletsResourceResolver*. The presence of this annotation must be ignored if the corresponding param has been specified. If present, this annotation must be attached to a class that extends *jakarta.faces.view.facelets.ResourceResolver*. If more than one class in the application has this annotation, an informative error message with logging level SEVERE must be logged indicating this case. Exactly one of the classes with the annotation must be taken to be the *ResourceResolver* for the application and any other classes with the annotation must be ignored. See Ordering of Artifacts for the means to put application configuration resources in order such that the chosen class can be defined. The same rules regarding decoration of the instance as listed above must apply to the annotated class.

- *jakarta.faces.FACELETS_SKIP_COMMENTS* —If this param is set, and calling *toLowerCase().equals("true")* on a *String* representation of its value returns *true*, the runtime must ensure that any XML comments in the Facelets source page are not delivered to the client.
- *jakarta.faces.FACELETS_SUFFIX* —Allow the web application to define alternate suffixes for Facelet based XHTML pages containing Jakarta Faces content. See the javadocs for the symbolic constant *ViewHandler.FACELETS_SUFFIX_PARAM_NAME* for the complete specification.
- *jakarta.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. See the javadocs for the symbolic constant *ViewHandler.FACELETS_VIEW_MAPPINGS_PARAM_NAME* for the complete specification.
- ~~*jakarta.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 pre-Jakarta Faces JSF 1.2 (under the JCP).~~ **Deprecated since 4.1:** Full state saving will be removed in favor of partial state saving in order to keep the spec simple. Therefore specifying full state saving view IDs via this context parameter will not be an option anymore.
- *jakarta.faces.HONOR_CURRENT_COMPONENT_ATTRIBUTES* — The ServletContext init parameter consulted by the UIComponent to tell whether or not the CURRENT_COMPONENT and CURRENT_COMPOSITE_COMPONENT attribute keys should be honored as specified. If this parameter is not specified, or is set to false, the contract specified by the CURRENT_COMPONENT and CURRENT_COMPOSITE_COMPONENT method is not honored. If this parameter is set to true, the contract is honored.
- *jakarta.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 *jakarta.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

- *jakarta.faces.LIFECYCLE_ID* —Lifecycle identifier of the *Lifecycle* instance to be used when processing Jakarta Faces requests for this web application. If not specified, the Jakarta Faces default instance, identified by *LifecycleFactory.DEFAULT_LIFECYCLE*, must be used.

- ~~*jakarta.faces.PARTIAL_STATE_SAVING* --The ServletContext init parameter consulted by the runtime to determine if the partial state saving mechanism should be used.~~ **Deprecated since 4.1:** Full state saving will be removed in favor of partial state saving in order to keep the spec simple. Therefore disabling partial state saving via this context parameter will not be an option anymore.
- *jakarta.faces.PROJECT_STAGE* —A human readable string describing where this particular Jakarta Faces application is in the software development lifecycle. Valid values are "*Development*", "*UnitTest*", "*SystemTest*", or "*Production*", corresponding to the enum constants of the class *jakarta.faces.application.ProjectStage*. It is also possible to set this value via JNDI. See the javadocs for *Application.getProjectStage()*.
- *jakarta.faces.RESOURCE_EXCLUDES* — The ServletContext init parameter consulted by the handleResourceRequest to tell which kinds of resources must never be served up in response to a resource request. The value of this parameter is a single space separated list of file extensions, including the leading '.' character (without the quotes). If not specified, the default value `.class .jsp .jspx .properties .xhtml .groovy` is used. If manually specified, the given value entirely overrides the default one and does not supplement it.
- *jakarta.faces.SEPARATOR_CHAR* --The context param that allows the character used to separate segments in a *UIComponent* clientId to be set on a per-application basis.
- *jakarta.faces.SERIALIZE_SERVER_STATE* --If this param is set, and calling *toLowerCase().equals("true")* on a *String* representation of its value returns *true*, and the *jakarta.faces.STATE_SAVING_METHOD* is set to "server" (as indicated below), the server state must be guaranteed to be Serializable such that the aggregate state implements *java.io.Serializable*. The intent of this parameter is to ensure that the act of writing out the state to an *ObjectOutputStream* would not throw a *NotSerializableException*, but the runtime is not required verify this before saving the state.
- *jakarta.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. When examining the parameter value, the runtime must ignore case.
- *jakarta.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"*.
- *jakarta.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 *jakarta.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.

- *jakarta.faces.validator.ENABLE_VALIDATE_WHOLE_BEAN* —If this param is set, and calling *toLowerCase().equals("true")* on a *String* representation of its value returns *true*, the *<f:validateWholeBean />* tag is enabled. If not set or set to false, this tag is a no-op.
- *jakarta.faces.VIEWROOT_PHASE_LISTENER_QUEUES_EXCEPTIONS* —If this param is set, and calling *toLowerCase().equals("true")* on a *String* representation of its value returns *true*, exceptions thrown by *PhaseListeners* installed on the *UIViewRoot* are queued to the *ExceptionHandler* instead of being logged and swallowed. If this param is not set or is set to false, the old behavior prevails.
- *jakarta.faces.ENABLE_WEBSOCKET_ENDPOINT* — Enable WebSocket support. See the javadoc for *jakarta.faces.component.UIWebsocket*.
- *jakarta.faces.WEBSOCKET_ENDPOINT_PORT* — The integer context parameter name to specify the websocket endpoint port when it's different from HTTP port. See the javadoc for *jakarta.faces.component.UIWebsocket*.
- *jakarta.faces.WEBAPP_RESOURCES_DIRECTORY*

  If this param is set, the runtime must interpret its value as a path, relative to the web app root, where resources are to be located. This param value must not start with a "/", though it may contain "/" characters. If no such param exists, or its value is invalid, the value "resources", without the quotes, must be used by the runtime as the value.

- *jakarta.faces.WEBAPP_CONTRACTS_DIRECTORY*

  If this param is set, the runtime must interpret its value as a path, relative to the web app root, where resource library contracts are to be located. This param value must not start with a "/", though it may contain "/" characters. If no such param exists, or its value is invalid, the value "contracts", without the quotes, must be used by the runtime as the value.

Jakarta Faces implementations may choose to support additional configuration parameters, as well as additional mechanisms to customize the Jakarta Faces implementation; however, applications that rely on these facilities will not be portable to other Jakarta Faces implementations.

11.2. Included Classes and Resources

A Jakarta Faces-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. Jakarta Servlet API Classes (jakarta.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. Jakarta Tags API Classes (jakarta.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. Jakarta Tags 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. Jakarta Faces API Classes (jakarta.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. Jakarta 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

jakarta.faces.FactoryFinder implements the standard discovery algorithm for all factory objects specified in the Jakarta 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 Jakarta 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 Jakarta Faces configuration resource(s) named by the *jakarta.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 Jakarta 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/jakarta.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 Jakarta 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 Jakarta 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 Jakarta Faces factory class, based on the discovery algorithm described above.

Jakarta Faces 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. 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.

11.2.6.2. FacesServlet

FacesServlet is an implementation of *jakarta.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 Servlet Definition, and mapped to a standard URL pattern as described in 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 *jakarta.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 *jakarta.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 Jakarta Faces implementation uses one.

The FacesServlet implementation class must also declare two static public final String constants whose value is a context initialization parameter that affects the behavior of the servlet:

- *CONFIG_FILES_ATTR* —the context initialization attribute that may optionally contain a comma-delimited list of context relative resources (in addition to */WEB-INF/faces-config.xml* which is always processed if it is present) to be processed. The value of this constant must be "*jakarta.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 "*jakarta.faces.LIFECYCLE_ID*".

11.3. Application Configuration Resources

This section describes the Jakarta Faces support for portable application configuration resources used to configure application components.

11.3.1. Overview

Jakarta Faces defines a portable configuration resource format (as an XML document) for standard configuration information. Please see the Javadoc overview for a link, titled “faces-config XML Schema Documentation” to the XML Schema Definition for such documents.

One or more such application resources will be loaded automatically, at application startup time, by the Jakarta Faces implementation. The information parsed from such resources will augment the information provided by the Jakarta Faces implementation, as described below.

In addition to their use during the execution of a Jakarta Faces-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.

11.3.2. Application Startup Behavior

Implementations may check for the presence of a *servlet-class* definition of class *jakarta.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 Jakarta Faces Technology.

At application startup time, before any requests are processed, the Jakarta Faces implementation must process zero or more application configuration resources, located as follows

Make a list of all of the application configuration resources found using the following algorithm:

- Check for the existence of a context initialization parameter named *jakarta.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 specified resources to the list. If this parameter exists, skip the searching specified in the next bullet item in this list.
- Search for all resources that match either “*META-INF/faces-config.xml*” or end with “*.faces-config.xml*” directly in the “*META-INF*” directory. Each resource that matches that expression must be considered an application configuration resource.
- Using the *java.util.ServiceLoader*, locate all implementations of the *jakarta.faces.ApplicationConfigurationResourceDocumentPopulator* service. For each implementation, create a fresh *org.w3c.dom.Document* instance, configured to be in the XML namespace of the application configuration resource format, and invoke the implementation’s *populateApplicationConfigurationResource()* method. If no exception is thrown, add the document to the list, otherwise log a message and continue.

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*.

Please see 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 Jakarta Faces-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 Jakarta Faces implementation classes that are stored in this library JAR file to be automatically registered, with no action required by the application.

Perform the actions specified in Faces Flows.

Perform the actions specified in Resource Library Contracts.

The runtime must publish the *jakarta.faces.event.PostConstructApplicationEvent* immediately after all application configuration resources have been processed.

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. Jakarta Faces implementations that are part of a Jakarta EE technology-compliant implementation are required to validate the application resource file against the XML schema for structural correctness. The validation is recommended, but not required for Jakarta Faces implementations that are not part of a Jakarta EE technology-compliant implementation.

11.3.2.1. Resource Library Contracts

If the parsing of the application configuration resources completed successfully, scan the application for resource library contracts. Any resource library contract as described in Resource Library Contracts must be discovered at application startup time. The complete set of discovered contracts has no ordering semantics and effectively is represented as a *Set*<*String*> where the values are just the names of the resource libraries. If multiple sources in the application configuration resources contained <*resource-library-contracts*>, they are all merged into one element. Duplicates are resolved in as specified in Ordering of Artifacts. If the application configuration resources produced a <*resource-library-contracts*> element, create an implementation private data structure (called the "resource library contracts data structure") containing the mappings between viewId patterns and resource library contracts as listed by the contents of that element.

The <*resource-library-contracts*> element is contained with in the <*application*> element and contains one or more <*contract-mapping*> elements. Each <*contract-mapping*> element must one or more <*url-pattern*> elements and one or more <*contract*> elements.

The value of the <*url-pattern*> element may be any of the following.

- The literal string *, meaning all views should have these contracts applied.
- An absolute prefix mapping, relative to the web app root, such as */directoryName/** meaning only views matching that prefix should have these contracts applied.
- An exact fully qualified file path, relative to the web app root, such as */directoryName/fileName.xhtml*, meaning exactly that view should have the contracts applied.

  See ViewDeclarationLanguage.calculateResourceLibraryContracts() for the specification of how the values of the <*url-pattern*> are to be processed.

The value of the <*contracts*> element is a comma separated list of resource library contract names. A resource library contract name is the name of a directory within the *contracts* directory of the web app root, or the *contracts* directory within the *META-INF/contracts* JAR entry.

Only the contracts explicitly mentioned in the <*resource-library-contracts*> element are included in the data structure. If the information from the application configuration resources refers to a contract that is not available to the application, an informative error message must be logged.

If the application configuration resources did not produce a <*resource-library-contracts*> element, the data structure should be populated as if this were the contents of the <*resource-library-contracts*> element:

```
<resource-library-contracts>
  <contract-mapping>
    <url-pattern>*</url-pattern>
    <contracts>all available contracts</contracts>
  </contract-mapping>
</resource-library-contracts>
```

Where “all available contracts” is replaced with a comma separated list of all the contracts discovered in the startup scan. In the case where there is no <*resource-library-contracts*> element in the application configuration resources, ordering of contracts is unspecified, which may lead to unexpected behavior in the case of multiple contracts that have the same contract declaration.

11.3.3. Faces Flows

If the parsing of the application configuration resources completed successfully, any XML based flow definitions in the application configuration resources will have been successfully discovered as well. The discovered flows must be exposed as thread safe immutable application scoped instances of *jakarta.faces.flow.Flow*, and made accessible to the runtime via the *FlowHandler*. If flows exist in the application, but the *jakarta.faces.CLIENT_WINDOW_MODE* context-param was not specified, the runtime must behave as if the value “url” (without the quotes) was specified for this context-param.

11.3.3.1. Defining Flows

Flows are defined using the <*flow-definition*> element. This element must have an *id* attribute which

uniquely identifies the flow within the scope of the Application Configuration Resource file in which the element appears. To enable multiple flows with the same *id* to exist in an application, the *<faces-config><name>* element is taken to be the *definingDocumentId* of the flow. If no *<name>* element is specified, the empty string is taken as the value for *definingDocumentId*. Please see FlowHandler for an overview of the flow feature. Note that a number of conventions exist to make defining flows simpler. These conventions are specified in Packaging Flows in Directories.

11.3.3.2. Packaging Faces Flows in JAR Files

The runtime must support packaging Faces Flows in JAR files as specified in this section. Any flows packaged in a jar file must have its flow definition included in a *faces-config.xml* file located at the *META-INF/faces-config.xml* JAR entry. This ensures that such flow definitions are included in the application configuration resources. Any view nodes included in the jar must be located within sub entries of the *META-INF/flows/<flowName>* JAR entry, where *<flowName>* is a JAR directory entry whose name is identical to that of a flow id in the corresponding *faces-config.xml* file. If there are *@FlowScoped* beans or beans with *@FlowDefinition* in the JAR, there must be a JAR entry named *META-INF/beans.xml*. This ensures that such beans and definitions are discovered by the runtime at startup. None of the flow definition conventions specified in Packaging Flows in Directories apply when a flow is packaged in a JAR file. In other words, the flow must be explicitly declared in the JAR file's *faces-config.xml*.

11.3.3.3. Packaging Flows in Directories

The view nodes of a flow need not be collected in any specific directory structure, but there is a benefit in doing so: flow definition conventions. If the *jakarta.faces.CONFIG_FILES* context parameter includes references to files of the form */<flowName>/<flowName>-flow.xml* or */WEB-INF/<flow-Name>/<flowName>-flow.xml*, and if such files exist in the current application (even if they are zero length), they are treated as flow definitions. Flow definitions defined in this way must not be nested any deeper in the directory structure than one level deep from the web app root or the *WEB-INF* directory.

The following conventions apply to flows defined in this manner. Any flow definition in the corresponding *-flow.xml* file will override any of the conventions in the case of a conflict.

- Every vdl file in that directory is a view node of that flow.
- The start node of the flow is the view whose name is the same as the name of the flow.
- Navigation among any of the views in the directory is considered to be within the flow.
- The flow defining document id is the empty string.

In the case of a zero length flow definition file, the following also applies:

- There is one return node in the flow, whose id is the id of the flow with the string "*-return*" (without the quotes) appended to it. For example, if *flowId* is *shopping*, the return node id is *shopping-return*.
- The from-outcome of the return node is a string created with the following formula: *"/" + flowId + "-return"*.

For each directory packaged flow definition, the runtime must synthesize an instance of

jakarta.faces.flow.Flow that represents the union of the flow definition from the */<flowName>/<flowName>-flow.xml* file for that directory, and any of the preceding naming conventions, with precedence being given to the *-flow.xml* file. Such *Flow* instances must be added to the *FlowHandler* before the *PostConstructApplicationEvent* is published.

11.3.4. Application Shutdown Behavior

When the Jakarta Faces runtime is directed to shutdown by its container, the following actions must be taken.

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 *jakarta.faces.event.PreDestroyApplicationEvent*.
3. Call *FactoryFinder.releaseFactories()*.

11.3.5. Application Configuration Resource Format

Application configuration resources that are written to run on Jakarta Faces 4.1 must include the following schema declaration and must conform to the schema shown in Appendix A - Jakarta Faces Metadata

```
<faces-config
    xmlns="https://jakarta.ee/xml/ns/jakartaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
        https://jakarta.ee/xml/ns/jakartaee/web-facesconfig_4_1.xsd"
    version="4.1">
```

Application configuration resources that are written to run on Jakarta Faces 4.0 must include the following schema declaration:

```
<faces-config
    xmlns="https://jakarta.ee/xml/ns/jakartaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
        https://jakarta.ee/xml/ns/jakartaee/web-facesconfig_4_0.xsd"
    version="4.0">
```

Application configuration resources that are written to run on Jakarta Faces 3.0 must include the following schema declaration:

```
<faces-config
    xmlns="https://jakarta.ee/xml/ns/jakartaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
        https://jakarta.ee/xml/ns/jakartaee/web-facesconfig_3_0.xsd"
```

```
    version="3.0">
```

Note that the “hostname” of the *xmlns* and *xsi:schemaLocation* attributes has changed from “xmlns.jcp.org” to “jakarta.ee”. The “jakarta.ee” hostname must be used when using *version="3.0"* and *web-facesconfig_3_0.xsd*. It is not valid to use this hostname with versions prior to 3.0. Likewise, it is not valid to use the “xmlns.jcp.org” hostname when using *version="3.0"* and *web-facesconfig_3_0.xsd*.

Application configuration resources that are written to run on pre-Jakarta Faces JSF 2.3 must include the following schema declaration:

```
<faces-config
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_3.xsd"
    version="2.3">
```

Application configuration resources that are written to run on pre-Jakarta Faces JSF 2.2 must include the following schema declaration:

```
<faces-config
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd"
    version="2.2">
```

Note that the “hostname” of the *xmlns* and *xsi:schemaLocation* attributes has changed from “java.sun.com” to “xmlns.jcp.org”. The “xmlns.jcp.org” hostname must be used when using *version="2.2"* and *web-facesconfig_2_2.xsd*. It is not valid to use this hostname with versions prior to 2.2. Likewise, it is not valid to use the “java.sun.com” hostname when using *version="2.2"* and *web-facesconfig_2_2.xsd*.

Application configuration resources that are written to run on pre-Jakarta Faces JSF 2.1 must include the following schema declaration:

```
<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_1.xsd"
    version="2.1">
```

Application configuration resources that are written to run on pre-Jakarta Faces JSF 2.0 must include the following schema declaration:

```
<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 pre-Jakarta Faces 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
    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"
    version="1.2">
```

Application configuration resources that are written to run on pre-Jakarta Faces 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 pre-Jakarta Faces JSF 1.0 implementations must use the DTD declaration for the 1.0 DTD contained in the binary download of the pre-Jakarta Faces JSF 1.0 reference implementation. They must also use the following DOCTYPE declaration:

```
<!DOCTYPE faces-config PUBLIC
    "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
    "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
```

11.3.6. Configuration Impact on Jakarta Faces Runtime

The following XML elements [12] in application configuration resources cause registration of Jakarta Faces 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 Jakarta Faces 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 *jakarta.faces.Integer*, then your replacement class must be a subclass of *jakarta.faces.convert.IntegerConverter*.

For replacement *Renderer*s, your implementation class must extend *jakarta.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 Jakarta Faces implementation. See Delegating Implementation Support for more information about the classes referenced by these elements:

- */faces-config/application/action-listener* —Replace the default *ActionListener* used to process *ActionEvent events with an* instance with the class specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is an *ActionListener*.
- */faces-config/application/navigation-handler* —Replace the default *NavigationHandler* instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a *NavigationHandler*.
- /faces-config/application/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*.
- */faces-config/application/search-expression-handler* —This element contains the fully qualified class name of the concrete *jakarta.faces.component.search.SearchExpressionHandler* implementation class that will be used for processing of a search expression.
- /faces-config/application/search-keyword-resolver — This element contains the fully qualified class name of the concrete jakarta.faces.component.search.SearchKeywordResolver implementation class that will be used during the processing of a search expression keyword.
- */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 system *EventClass*. 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/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*.

The following XML elements cause the replacement of the default implementation class for the corresponding functionality, provided by the Jakarta Faces 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/client-window-factory* —Replace the default *ClientWindowFactory* instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a *ClientWindowFactory*.
- */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/flash-factory* —Replace the default *FlashFactory* instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a *FlashFactory*.
- */faces-config/factory/flow-handler-factory* —Replace the default *FlowHandlerFactory* instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a *FlowHandlerFactory*.
- */faces-config/factory/lifecycle-factory* —Replace the default *LifecycleFactory* instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a *LifecycleFactory*.
- */faces-config/factory/render-kit-factory* —Replace the default *RenderKitFactory* instance with the one specified. The contents of this element must be a fully qualified Java class name that, when instantiated, is a *RenderKitFactory*.
- */faces-config/factory/search-expression-context-kit-factory* —This element contains the fully qualified class name of the concrete *SearchExpressionContextFactory* implementation class that will be called when *FactoryFinder.getFactory(SEARCH_EXPRESSION_CONTEXT_FACTORY)* is

called.

- */faces-config/factory/view-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*.

The following XML elements cause the addition of event listeners to standard Jakarta Faces 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 Jakarta Faces implementation, even though they do not cause registration of objects that are visible to a Jakarta Faces-based application.

- */faces-config/navigation-rule* —Make the characteristics of a navigation rule available to the default *NavigationHandler* implementation.

11.3.7. Delegating Implementation Support

The runtime must support the decorator design pattern as specified below for all classes of the *jakarta.faces* package implementing the *FacesWrapper* interface, among others:

- *Application* via *ApplicationWrapper*
- *ApplicationFactory*
- *ClientWindow* via *ClientWindowWrapper*
- *ClientWindowFactory*
- *ConfigurableNavigationHandler* via *ConfigurableNavigationHandlerWrapper*
- *ExceptionHandler* via *ExceptionHandlerWrapper*
- *ExceptionHandlerFactory*
- *ExternalContext* via *ExternalContextWrapper*
- *ExternalContextFactory*
- *FaceletCacheFactory*
- *FacesContext* via *FacesContextWrapper*
- *FacesContextFactory*
- *Flash* via *FlashWrapper*
- *FlashFactory*
- *FlowHandlerFactory* via *FlowHandlerFactoryWrapper*
- *Lifecycle* via *LifecycleWrapper*

- *LifecycleFactory*
- *NavigationCase* via *NavigationCaseWrapper*
- *NavigationHandler* via *NavigationHandlerWrapper*
- *PartialViewContext* via *PartialViewContextWrapper*
- *PartialViewContextFactory*
- *Renderer* via *RendererWrapper*
- *RenderKit* via *RenderKitWrapper*
- *RenderKitFactory*
- *Resource* via *ResourceWrapper*
- *ResourceHandler* via *ResourceHandlerWrapper*
- *ResponseWriter* via *ResponseWriterWrapper*
- *SearchExpressionContextFactory*
- *SearchExpressionHandler* via *SearchExpressionHandlerWrapper*
- *StateManager* via *StateManagerWrapper*
- *TagHandlerDelegateFactory*
- *ViewDeclarationLanguage* via *ViewDeclarationLanguageWrapper*
- *ViewDeclarationLanguageFactory*
- *ViewHandler* via *ViewHandlerWrapper*
- *VisitContext* via *VisitContextWrapper*
- *VisitContextFactory*

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 via *FacesWrapper.getWrapped()* method a reference to the implementation that was previously fulfilling the role. In this way, the custom implementation is able to override just a subset of the functionality (or provide only some additional functionality) and delegate the rest to the existing implementation.

For example, say you wanted to provide a custom *ViewHandler* that was the same as the default one, but provided a different implementation of the *calculateLocale()* method. Consider this code excerpt from a custom *ViewHandler*:

```
public class YourViewHandler extends ViewHandlerWrapper {

  // Always implement the constructor taking the wrapped instance
  public YourViewHandler(ViewHandler wrapped) {
    super(wrapped);
  }

  // Overridden version of calculateLocale()
  public Locale calculateLocale(FacesContext context) {
```

```
    // You can if necessary obtain the original locale as follows
    Locale originalLocale = getWrapped().calculateLocale(context);

    // You can perform custom calculation of the locale here
    Locale customLocale = ...

    return customLocale;
  }
}
```

This can then be configured in the *faces-config.xml* file as follows:

```
<application>
  <view-handler>com.example.YourViewHandler</view-handler>
</application>
```

The constructor taking the wrapped instance will get called as the application is initially configured by the Jakarta Faces implementation, and the previously registered *ViewHandler* will get passed to it.

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()*.

11.3.8. 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.

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*. If a <*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* /> elements. 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*> elements and zero or one <*after*> elements. 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.

- <*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*> elements, or zero or one time directly within the <*absolute-ordering*> elements. The <*others* /> element must be handled as follows.
- The <*others* /> element represents a set of application configuration resources. This set is described as the set of all application configuration resources discovered in the application, minus the one being currently processed, minus the application configuration resources mentioned by name in the <*ordering*/> section. If this set is the empty set, at the time the application configuration resources are being processed, the <*others* /> element must be ignored.
    - 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.

11.3.9. 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" encoding="UTF-8"?>
<faces-config
    xmlns="https://jakarta.ee/xml/ns/jakartaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
        https://jakarta.ee/xml/ns/jakartaee/web-facesconfig_4_1.xsd"
    version="4.1">

  <!-- 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 Example NavigationHandler Configuration.

11.4. 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.

11.4.1. Requirements for scanning of classes for annotations

- 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 "*.*\.faces-config.xml*" (even an empty one), all classes in that jar must be scanned.

Chapter 12. Lifecycle Management

In Request Processing Lifecycle, the required functionality of each phase of the request processing lifecycle was described. This chapter describes the standard APIs used by Jakarta Faces implementations to manage and execute the lifecycle. Each of these classes and interfaces is part of the *jakarta.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 Jakarta Faces implementors.

12.1. Lifecycle

Upon receipt of each Jakarta Faces-destined request to this web application, the Jakarta Faces 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 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 Jakarta 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 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 Jakarta Faces 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. Any exceptions thrown during the *beforePhase()* listeners must be caught and published to the *ExceptionHandler*, as described below. 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 Jakarta Faces lifecycle phase, must be caught and and published to the *ExceptionHandler*, as described below. 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. Any exceptions thrown during the *afterPhase()* liseteners must be caught and published to the *ExceptionHandler*, as described below.

The previous paragraph detailed several cases where exceptions should be published to the *Exception* handler. 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 *ExceptionQueuedEvent* 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) {
  jakarta.faces.event.ExceptionEventContext eventContext =
      new ExceptionEventContext(
          thrownException, null, facesContext.getPhaseId());
  eventContext.getAttributes()
      .put(EventContext.IN_BEFORE_PHASE, Boolean.TRUE);
  app.publishEvent(ExceptionQueuedEvent.class, eventContext);
}

try {
  doCurrentPhase();
} catch (Throwable thrownException) {
  jakarta.faces.event.ExceptionEventContext eventContext =
      new ExceptionEventContext(
          thrownException, null, facesContext.getPhaseId());
  app.publishEvent(ExceptionQueuedEvent.class, eventContext);
```

```
} finally {
  try {
    callAfterPhaseListeners();
  } catch (Throwable thrownException) {
    jakarta.faces.event.ExceptionEventContext eventContext =
        new ExceptionEventContext(
            thrownException, null, facesContext.getPhaseId());
    eventContext.getAttributes()
        .put(EventContext.IN_AFTER_PHASE, Boolean.TRUE);
    app.publishEvent(ExceptionQueuedEvent.class, eventContext);
  }
  handler.handle();
}
```

body text.

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 *jakarta.faces.lifecycle.LifecycleFactory* must be made available to each Jakarta Faces-based web application running in a servlet or portlet container. The factory instance can be acquired by Jakarta Faces 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 Jakarta Faces 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 Jakarta Faces 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*.

Chapter 13. Ajax Integration

This chapter of the specification describes how Ajax integrates with the Jakarta Faces framework to create dynamic web applications. This chapter describes the resources and JavaScript APIs that are used to expose the Ajax capabilities of Jakarta Faces to page authors and component authors. It also describes the necessary ingredients of a Jakarta 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 given by the value of the constant *jakarta.faces.application.ResourceHandler.FACES_SCRIPT_RESOURCE_NAME* and it must exist under the resource library given by the value of the constant *jakarta.faces.application.ResourceHandler.FACES_SCRIPT_LIBRARY_NAME*, following the conventions in *Resource Handling*. This resource contains the JavaScript APIs that facilitate Ajax interaction with Jakarta Faces.

13.1.1. JavaScript Resource Loading

The JavaScript resource can become available to a Jakarta 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(
  library=ResourceHandler.FACES_SCRIPT_LIBRARY_NAME,
  name=ResourceHandler.FACES_SCRIPT_RESOURCE_NAME,
  target="head")
public class YourComponent extends UIOutput...
```

For more information on this approach refer to Relocatable Resources and 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("faces.js", "jakarta.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 Relocatable Resources) which means you can control the rendering location for these resources by setting the "target" attribute on the resource component:

```
public class YourComponent 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 explicitly available to the current view using the <*h:outputScript*> tag referencing resource library *jakarta.faces* and resource name *faces.js*. For example:

```
<h:outputScript library="jakarta.faces" name="faces.js" />
```

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 Jakarta Faces 2.0 JavaScript API.

Any implementation that intends to use the Jakarta Faces JavaScript API must define a top level JavaScript object name faces, whose type is a JavaScript associative array. Within that top level

JavaScript object, there must be a property named ajax..

```
if (faces == null || typeof faces == "undefined") {
    var faces = new Object();
}
if (faces.ajax == null || typeof faces.ajax == "undefined") {
    faces["ajax"] = new Object();
}
```

13.3. Ajax Interaction

This section of the specification outlines the Ajax JavaScript APIs that are used to initiate client side interactions with the Jakarta 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 JavaScript API for details about the individual API functions.

13.3.1. Sending an Ajax Request

The JavaScript function faces.ajax.request is used to send information to the server to control partial view processing (Partial View Processing) and partial view rendering (Partial View Rendering). All requests using the faces.ajax.request function will be made asynchronously to the server. Refer to Initiating an Ajax Request.

13.3.2. Ajax Request Queueing

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 faces.ajax.request JavaScript documentation for more specifics about the Ajax request queue.

13.3.3. Request Callback Function

The Ajax request callback function is called when the Ajax request/response interaction is complete. This function must perform the following actions:

- If the return status is >= 200 and < 300, send a “complete” event following Sending Events. Call faces.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 Sending Events. Send an “httpError” error following 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.

Refer to Receiving The Ajax Response. Also refer to the faces.ajax.request JavaScript documentation for more specifics about the request callback function.

13.3.4. Receiving The Ajax Response

The faces.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. The implementation of faces.ajax.response must handle the response as outlined in the JavaScript documentation for faces.ajax.response. The elements in the response must be processed in the order they appear in the response.

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 faces.ajax.request with onevent as an option.

Refer to the VDLDocs on the *f:* tag library for details on the use of the <f:ajax> tag approach. Refer to Initiating an Ajax Request for details about using the faces.ajax.request function approach. The implementation must ensure the JavaScript function that is registered for an Ajax request must be called in accordance with the events outlined in Events.

13.3.5.2. Monitoring Events For All Ajax Requests

The JavaScript API provides the faces.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 Registering Callback Functions for more details. The faces.ajax.addOnEvent function accepts a JavaScript function argument that will be notified when events occur during any Ajax request/response event cycle. The implementation must ensure the JavaScript function that is registered must be called in accordance with the events outlined in Events.

13.3.5.3. Sending Events

The implementation must send events to the runtime as follows:

- Construct a data payload for events using the properties described in Event Data Payload.

- If an event handler function was registered with the “onevent” attribute (Monitoring Events For An Ajax Request) call it passing the data payload.
- If any event handling functions were registered with the “addOnEvent” function (Monitoring Events For All Ajax Requests) call them passing the data payload.

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 faces.ajax.request with onerror as an option.

Refer to the VDLDocs on the *f:* tag library for details on the use of the <f:ajax> tag approach. Refer to Initiating an Ajax Request for details about using the faces.ajax.request function approach. 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 Errors.

13.3.6.2. Handling Errors For All Ajax Requests

The JavaScript API provides the faces.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 Registering Callback Functions for more details. The faces.ajax.addOnError function accepts a JavaScript function argument that will be notified when errors occur during any Ajax request/response cycle. The implementation must ensure the JavaScript function that is registered must be called in accordance with the errors outlined in Errors.

13.3.6.3. Signaling Errors

The implementation must signal errors to the runtime as follows:

- Construct a data payload for errors using the properties described in Error Data Payload.
- If an error handler function was registered with the “onerror” attribute (Handling Errors For An Ajax Request) call it passing the data payload.
- If any error handling functions were registered with the “addOnError” function (Handling Errors For All Ajax Requests) call them passing the data payload.
- If the project stage is “development” (see Determining An Application’s Project Stage) use JavaScript “alert” to signal the error(s).

13.3.7. Handling Errors On The Server

Jakarta Faces handles exceptions on the server as outlined in ExceptionHandler. Jakarta Faces Ajax frameworks must ensure exception information is written to the response in the format:

```
<partial-response id="j_id1">
  <error>
    <error-name>...</error-name>
    <error-message>...</error-message>
  </error>
</partial-response>
```

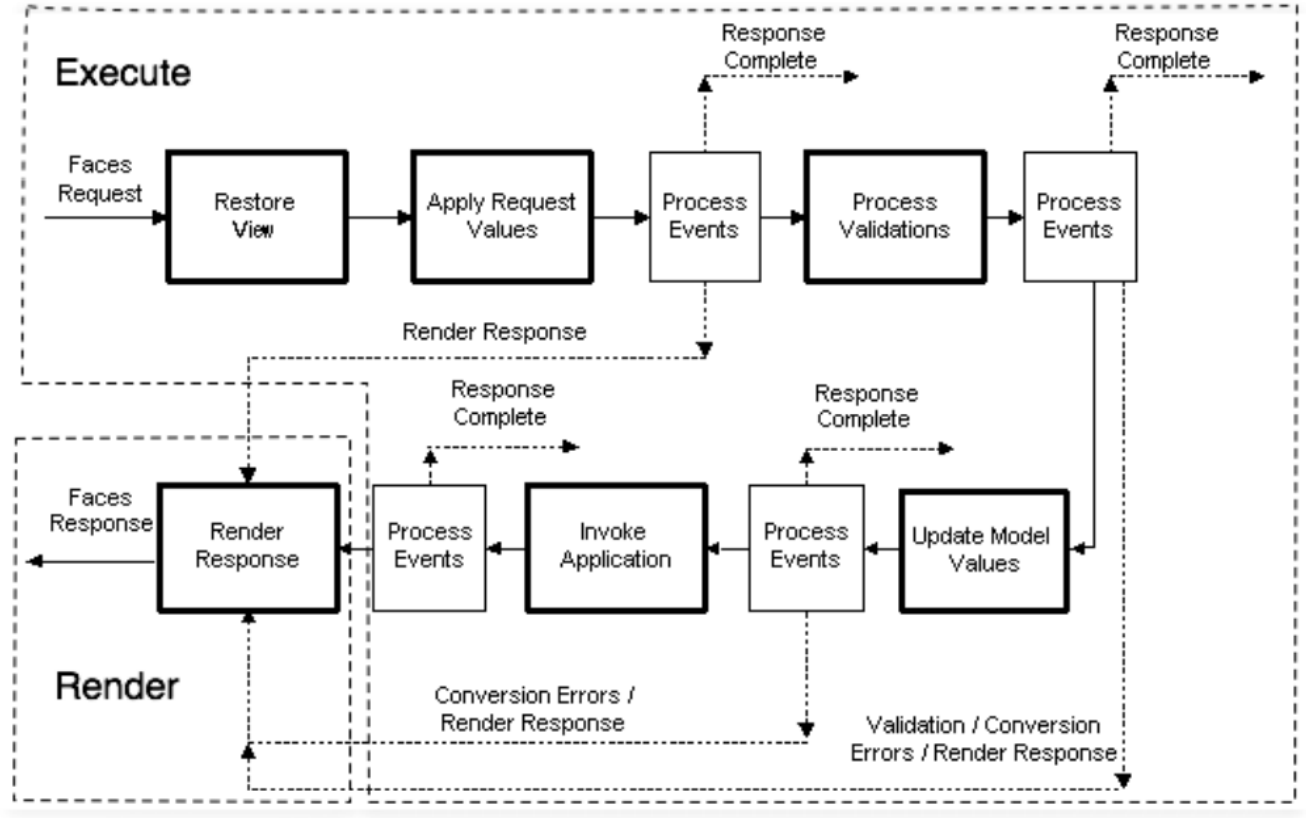
- Extract the “class” from the “Throwable” object and write that as the contents of error-name in the response.
- Extract the “cause” from the “Throwable” object if it is available and write that as the contents of error-message in the response. If “cause” is not available, write the string returned 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).

Implementations may choose to include a specialized ExceptionHandler for Ajax that extends from jakarta.faces.context.ExceptionHandlerWrapper, and have the jakarta.faces.context.ExceptionHandlerFactory implementation install it if the environment requires it.

13.4. Partial View Traversal

The Jakarta 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 Faces and Ajax frameworks and it allows selected components in the view to be processed and/or rendered. There are a variety of Jakarta Faces 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 Partial View Traversal depicts the “execute” portion as encompassing everything except the “Render Response Phase”, for the purposas of an ajax request, the execute portion of the lifecycle 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 (not triggered by Ajax) 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 Sending an Ajax Request about the request sending details. The FacesContext has methods for retrieving the PartialViewContext instance for the request. The PartialViewContext may also be retrieved by using the PartialViewContextFactory class. The XML schema allows for the definition

of a PartialViewContextFactory using the “partial-view-context-factory” element. Refer to the partial response schema in the Javadoc section of the spec for more information on this element. 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 jakarta.faces.context.PartialViewContext and Partial View Context for the specifics of the PartialViewContext constants and methods that facilitate partial processing. The UIViewRoot processDecodes, processValidators and processUpdates methods must determine if the request is a partial request using the FacesContext.getCurrentInstance().getPartialViewContext().isPartialRequest() method. If FacesContext.getCurrentInstance().getPartialViewContext().isPartialRequest() returns true, then the implementation of these methods must retrieve a PartialViewContext instance and invoke PartialViewContext.processPartial. Refer to Apply Request Values, Apply Request Values Partial Processing, Process Validations, Partial Validations Partial Processing, Update Model Values, Update Model Values Partial Processing.

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 Sending an Ajax Request about the request sending details. The PartialViewContext has methods that indicate the request is a partial request based on the values of these special request parameters. Refer to Partial Processing Methods for the specifics of the FacesContext constants and methods that facilitate partial processing. The UIViewRoot getRendersChildren and encodeChildren methods must determine if the request is an Ajax request using the FacesContext.getCurrentInstance().getPartialViewContext().isAjaxRequest() method. If PartialViewContext.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.

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. Implementations must adhere to the response format as specified in the JavaScript docs for faces.ajax.response.

13.4.4.1. Writing The Partial Response

Jakarta Faces provides jakarta.faces.context.PartialResponseWriter to ensure the Ajax response that is written follows the standard format. Implementations must take care to properly handle nested CDATA sections when writing the response. PartialResponseWriter decorates an existing ResponseWriter implementation by extending jakarta.faces.context.ResponseWriterWrapper. Refer to the jakarta.faces.context.PartialResponseWriter JavaDocs, and the JavaScript documentation for the faces.ajax.response function for more specifics.

Chapter 14. JavaScript API

This chapter of the specification describes the JavaScript functions that are used to facilitate Ajax operations in a Jakarta Faces framework. All of these functions are contained in the canonical *faces.js* file.

14.1. Collecting and Encoding View State

In Jakarta Faces the jakarta.faces.ViewState parameter was standardized to facilitate “postback” requests to the server in a Jakarta Faces application. Implementations must use this parameter to save the view state between requests. Refer to the Javadocs for jakarta.faces.render.ResponseStateManager.

Collecting and encoding view state that will be sent to the server is a common operation used by most Jakarta Faces Ajax frameworks. When a Jakarta Faces view is rendered, it will contain a hidden field with the identifier jakarta.faces.ViewState whose value contains the state for the current view. Jakarta Faces Ajax clients collect additional view state, combine it with the current view state and send it’s encoded form to the server.

```
faces.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 jakarta.faces.ViewState in the specified FORM_ELEMENT and encode the name and value.
- Return a concatenated String of the encoded input elements and jakarta.faces.ViewState element.

14.1.1. Use Case

Collect and Encode Elements Of a Form

```
var viewState = faces.getViewState(form);
```

14.2. Initiating an Ajax Request

```
faces.ajax.request(source, |event|, { |OPTIONS| });
```

The faces.ajax.request function is responsible for sending an Ajax request to the server. 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

14.2.1. Usage

Typically, this function is attached as a JavaScript event handler (such as “onclick”).

```
<ANY_HTML_OR_FACES_ELEMENT
    on|EVENT|="faces.ajax.request(source, event, { |OPTIONS| });" />
```

The function arguments are as follows:

source is the DOM element that triggered this Ajax request. It must be a DOM element object or a string identifier for a DOM element. 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 11. request OPTIONS

Name	Value
execute	A space delimited list of client identifiers or one of the keywords (Keywords). These reference the components that will be processed during the “execute” portion of the request processing lifecycle.
render	A space delimited list of client identifiers or one of the keywords (Keywords). These reference the components that will be processed during the “render” portion 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 JavaScript 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 12. Execute / Render Keywords

Keyword	Description
@all	All component identifiers
@none	No identifiers
@this	The element that triggered the request
@form	The enclosing 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="faces.ajax.request(this,event);" />

is the same as:

<h:commandButton id="button1" value="submit">
    onclick="faces.ajax.request(this, event,
        {execute:'@this', render:'@this'});" />
```

```
<h:commandButton id="button1" value="submit"
    onclick="faces.ajax.request(this,event, {execute:'@this'});" />

is the same as:

<h:commandButton id="button1" value="submit">
    onclick="faces.ajax.request(this,event, {execute:'button1'});" />
```

Refer to the VDLDocs on the *f:* tag library 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. 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 faces.ajax.request function. The post data arguments that must be sent are:

Name	Value
jakarta.faces.ViewState	The value of the jakarta.faces.ViewState hidden field. This is included when using the faces.getViewState function.
jakarta.faces.partial.ajax	true

Name	Value
jakarta.faces.source	The identifier of the element that is the source of this request

14.2.5. Use Case

```
<h:commandbutton id="submit" value="submit"
    onclick="faces.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

```
faces.ajax.response(request, context);
```

The faces.ajax.response function is called when a request completes successfully. This typically means that returned status code is >= 200 and < 300. The faces.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). The specifics details of this function’s operation must follow the faces.ajax.response JavaScript documentation.

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 implementations 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 (request OPTIONS), and the core <f:ajax> tag.

14.4.1. Request/Response Event Handling

```
faces.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 13. Events

Event Name	Description
begin	Occurs immediately before the request is sent.

Event Name	Description
complete	Occurs immediately after the request has completed. For successful requests, this is immediately before jakarta.faces.response is called. For unsuccessful requests, this is immediately before the error handling callback is invoked.
success	Occurs immediately after faces.ajax.response has completed.

The callback function has access to the following “data payload”:.

Table 14. Event Data Payload

Name	Description/Value
type	“event”
status	One of the events specified in Events
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;
responseText	The text response (XMLHttpResponse.responseText) Not present for “begin” event;

14.4.1.1. Use Case

An event listener can be installed from JavaScript in this manner.

```
function statusUpdate(data) {
  // do something with data.status or other parts of data payload
}
...
faces.ajax.addOnEvent(statusUpdate);
```

An event listener can be installed from markup in this manner.

```
<f:ajax ... onevent="statusUpdate" />
```

14.4.2. Error Handling

```
faces.ajax.addOnError(callback);
```

The callback argument must be a reference to an existing JavaScript function that will handle errors from the server.

Table 15. 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.
emptyResponse	There was no Ajax response from the server.

The callback function has access to the following “data payload”:.

Table 16. Error Data Payload

Name	Description/Value
type	“error”
status	One of error names defined Errors
description	Text describing the error
source	The DOM element that triggered the Ajax request.
responseCode	Ajax request object ‘status’ (XMLHttpRequest.status);
responseXML	The XML response (XMLHttpRequest.responseXML)
responseText	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

```
faces.ajax.addOnError(handleError);
...
var handleError = function handleError(data) {
  ... do something with data payload ...
}
```

14.5. Determining An Application’s Project Stage

```
faces.getProjectStage();
```

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 jakarta.faces.application.Application.getProjectStage(); Refer to ProjectStage Property for more details about this property.

14.5.1. Use Case

```
var projectStage = faces.getProjectStage();
if (projectStage == "Production") {
  .... throw exception
} else if (projectStage == "Development") {
  .... send an alert for debugging
}
```

14.6. Script Chaining

```
faces.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.

Appendix A: Jakarta Faces Metadata

The XML Schema Definition for Application Configuration Resource *web-facesconfig_4_1.xsd* is included in a web browser optimized format along with the Javadoc. That is the canonical location of the schemas in the specification.

A.1. Required Handling of **-extension* elements in the application configuration resources files

As specified in the XML Schema for Application Configuration Resources, many of the elements in the file have **-extension* elements declared in a similar fashion to this one for the *faces-config-extension*:

```
<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>
```

This section specifies the required handling of such elements.

Note that it is normal for an application to have several application configuration resources files. If multiple such resource files have conflicting **-extension* elements, the last element processed takes precedence over preceding elements. Processing order can be controlled as specified in Ordering of Artifacts.

A.1.1. *faces-config-extension* handling

If an application configuration resource contains a *faces-config-extension* element, the children of the element must be inspected for containing well-formed XML conforming to the syntax specified in the following subsection(s). DTD syntax is used for convenience since the content of a **-extension* element may not be constrained due to its declaration as containing *<xsd:any>*.

A.1.1.1. The *facelets-processing* element

DTD syntax..

```
<!ELEMENT facelets-processing (file-extension, process-as) >
<!ELEMENT file-extension ANY>
```

```
<!ELEMENT process-as ANY>
```

The *<facelets-processing>* element is used to affect the processing of Facelets VDL files. Therefore, this setting only applies to those requests that reach the Facelets *ViewDeclarationLanguage* implementation, as specified to the runtime via the *jakarta.faces.FACELETS_VIEW_MAPPINGS* and *jakarta.faces.FACELETS_SUFFIX* *<context-param>* entries. The specification defines three processing modes for Facelets files: Facelets XHTML syntax, XML View syntax, and Facelets JSPX syntax. This last syntax is intended to ease the migration to Facelets for applications already using the Jakarta Server Pages XML document syntax (also known as JSPX syntax). The affect on the processing of files in each of these three modes is specified in the following table.

Table 17. Valid <process-as> values and their implications on the processing of Facelet VDL files.

	<process-as>html5</process-as> *HTML 5 (default)*	*<process-as>xhtml</process-as>* *Facelets XHTML*	**<process-as>xml</process-as>** **XML View**	**<process-as>jspx</process-as>** **Facelets JSPX**
XML Doctype	Simplified to *<!DOCTYPE html>*	passed through	consumed	consumed
XML declaration	passed through	passed through	consumed	consumed
Processing instructions	passed through	passed through	consumed	consumed
CDATA section start and end tags	passed through	passed through	consumed	consumed
Escaping of inline text	escaped	escaped	escaped	not escaped
XML Comments	passed through	passed through	consumed	consumed

In the preceding table, “passed through” means that the content is passed through unmodified to the user agent. “consumed” means the content is silently consumed on the server. Note that for CDATA sections, the content of the CDATA section itself is passed through, even if the start and end tags should be consumed. “escaped” means that sensivite content in the response is automatically escaped: *&* becomes *&*, for example. “not escaped” means that such content is not escaped.

The content of the *<file-extension>* element is particular to the file extension of the physical resource for the Facelets VDL content, as specified in the *jakarta.faces.FACELETS_VIEW_MAPPINGS* and *jakarta.faces.FACELETS_SUFFIX* *<context-param>* elements. Consider the following example *faces-config.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config
    xmlns="https://jakarta.ee/xml/ns/jakartaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
        https://jakarta.ee/xml/ns/jakartaee/web-facesconfig_4_1.xsd"
    version="4.1">
```

```
  <faces-config-extension>
    <facelets-processing>
      <file-extension>.jspx</file-extension>
      <process-as>jspx</process-as>
    </facelets-processing>
    <facelets-processing>
      <file-extension>.view.xml</file-extension>
      <process-as>xml</process-as>
    </facelets-processing>
  </faces-config-extension>
</faces-config>
```

And *web.xml* content

```
<context-param>
  <param-name>jakarta.faces.FACELETS_VIEW_MAPPINGS</param-name>
  <param-value>*.xhtml;*.view.xml;*.jspx</param-value>
</context-param>
<context-param>
  <param-name>jakarta.faces.FACELETS_SUFFIX</param-name>
  <param-value>.xhtml .view.xml .jspx</param-value>
</context-param>
```

This configuration states that .xhtml, .view.xml, and .jspx files must be treated as Facelets VDL files declares the processing mode of .jspx files to be jspx and declares the processing mode of .view.xml files to be xml.

A.2. XML Schema Definition For Facelet Taglib

The XML Schema Definition for Facelet Taglib *web-facelettaglibrary_4_1.xsd* is included in a web browser optimized format along with the Javadoc. That is the canonical location of the schemas in the specification.

A.3. XML Schema Definition For Partial Response

The XML Schema Definition for Partial Repsonse *web-partialresponse_4_1.xsd* is included in a web browser optimized format along with the Javadoc. That is the canonical location of the schemas in the specification.

Appendix B: Change Log

Note: this refers to historic issues and titles using the term "JSF". This refers to the pre-Jakarta Faces specification under the JCP.

B.1. Changes between 4.1 and 4.0

This release contains the following changes:

- Issue ID 1832
  Unify ActionSource and ActionSource2
- Issue ID 1829
  Faces 4.1: Deprecate full state saving (FSS)
- Issue ID 1828
  Remove mentions in javadoc wrt backwards compatibility of facelets.* context params
- Issue ID 1823
  FacesMessage: implement equals(), hashcode(), toString()
- Issue ID 1821
  Spec: jakarta.faces.FACELETS_REFRESH_PERIOD default when ProjectStage is Development
- Issue ID 1819
  Add UUIDConverter
- Issue ID 1811
  Missing "if" attribute of <f:viewAction>
- Issue ID 1789
  Make FacesMessage#VALUES / VALUES_MAP generic
- Issue ID 1764
  Add ExternalContext.setResponseContentLengthLong
- Issue ID 1760
  id attribute is missing in vdl of h:head and h:body
- Issue ID 1739
  Require firing events for @Initialized, @BeforeDestroyed, @Destroyed for build-in scopes.
- Issue ID 1712
  Javadoc for Application.subscribeToEvent methods has a reference to context that is incorrect
- Issue ID 1708
  Add missing generics to API that were missed in Faces 4.0
- Issue ID 1707
  Deprecate unused PreDestroyCustomScopeEvent and PostConstructCustomScopeEvent
- Issue ID 1549
  Deprecate unused composite:extension
- Issue ID 1342
  Support @Inject of current flow like "@Inject Flow currentFlow"

- Issue ID 1263
  Add rowStatePreserved property to UIRepeat, exactly the same as UIData
- Issue ID 1864
  dataTable rowStatePreserved attribute is missing in vdl
- Issue ID 1838
  Remove references to the Java SecurityManager and associated APIs

This release now requires Java SE 17 or newer (aligned with Jakarta EE 11).

B.2. Changes between 4.0 and 3.0

This release contains the following changes:

- Issue ID 1581
  New API to programmatically create Facelets
- Issue ID 1508
  New automatic extensionless mapping
- Issue ID 1509
  New annotation @ClientWindowScoped
- Issue ID 1570
  Support custom cookie attributes such as SameSite in ExternalContext#addResponseCookie()
- Issue ID 1555
  New attribute <h:inputFile multiple="...">
- Issue ID 1556
  New attribute <h:inputFile accept="...">
- Issue ID 1557
  New method FacesContext#getLifecycle()
- Issue ID 1559
  New tag <f:selectItemGroups>
- Issue ID 1560
  New attribute <h:inputText type="...">
- Issue ID 1563
  New tag <f:selectItemGroup>
- Issue ID 1568
  New method UIViewRoot#getDoctype()
- Issue ID 1573
  New attribute <f:websocket onerror="...">
- Issue ID 1574
  New layout="list" for <h:selectManyCheckbox> and <h:selectOneRadio>
- Issue ID 1582
  New annotation literals for all @Qualifiers
- Issue ID 1558

Make UIComponent#subscribeToEvent() more convenient

- Issue ID 1565
  Skip type attribute from <link> and <script> when doctype is HTML5
- Issue ID 1567
  Improve <f:ajax> behavior in composite components

This release now requires Java SE 11 or newer (aligned with Jakarta EE 10).

B.2.1. Backward Compatibility with Previous Versions

This release contains the following backwards incompatible changes:

- Issue ID 1552
  Rename "JSF" to "Faces" over all place
- Issue ID 1553
  Rename "http://xmlns.jcp.org/jsf/**" URL to "jakarta.faces.**" URN
- Issue ID 1546
  Remove all JSP support
- Issue ID 1547
  Remove native Managed Beans (@ManagedBean and related)
- Issue ID 1548
  Remove MethodBinding, ValueBinding and friends
- Issue ID 1571
  Remove CURRENT_COMPONENT constants from UIComponent class
- Issue ID 1578
  Remove deprecated methods of StateManager class
- Issue ID 1583
  Remove entire ResourceResolver class

B.3. Changes between 3.0 and 2.3

Namespace has been changed from `javax.faces` to `jakarta.faces`.

B.3.1. Backward Compatibility with Previous Versions

Jakarta Faces 3.0 has a breaking change due to the namespace change from `javax.faces` to `jakarta.faces`.

B.4. Changes between 2.2 and 2.3

This section gives the reader a survey of the changes between version 2.3 of the specification and version 2.2, using the categories from the issue tracker at https://github.com/jakartaee/faces/issues/.

B.4.1. Big Ticket Features

- WebSocket Integration
  Issue ID 1396
  See the VDLDocs for <f:websocket>.
- Multi-field Validation
  Issue ID 1
  See the VDLDocs for <f:validateWholeBean>.
- Java Time Support
  Issue ID 1370
  See the VDLDocs for <f:convertDateTime>.
- Use CDI for evaluation of JSF specific EL implicit Objects
  Issue ID 1311, 1322, 1325, 1327, 1328, 1334, 1332, 1331, 1384, 1385, 1383, 1386 - 1394
  See Expression Language Resolution
- Issue ID 1417
  Deprecate javax.faces.bean.
- Support @Inject on JSF specific artifacts
  Issue ID 1316, 527, 1309, 1323, 1283, 1353, 1335, 1333, 1349, 1351, 1350, 1345
  See Jakarta Faces Objects Valid for @Inject Injection
- Issue ID 1364
  UIData and <ui:repeat> supports Map and Iterable
- Issue ID 1102
  ui:repeat condition check.
- Issue ID 1418
  CDI Replacement for @ManagedProperty.
  See javadocs for javax.faces.annotation.ManagedProperty.
- Issue ID 1103, 1364
  See the Javadoc for javax.faces.component.UIData, and javax.faces.model.IterableDataModel.
- DataModel implementations can be registered
  Issue ID 1078
  See the javadoc for annotation javax.faces.model.FacesDataModel.
- Issue ID 1412
  See the javadoc for annotation javax.faces.partialViewContext.getEvalScripts().
- Issue ID 613
  Ajax Method Invocation. See vdldoc for <h:commandScript>.
- Issue ID 1238
  Enhanced component search facility. See the javadoc for package javax.faces.component.search.

B.4.2. Other Features, by Functional Area

B.4.2.1. Components/Renderers

Larger Changes

- Issue ID 217
  styleClass attribute added to h:column
- Issue ID 329
  Add “group” attribute to <h:selectOneRadio>.
- Issue ID 1423, 1404
  ResourceHandler.markResourceRendered(), and isResourceRendered(), UIViewRoot.getComponentResources() enable the discovery of dynamically added resources, even within Ajax requests.
- Issue ID 1404
  Add API to check if a resource has already been rendered. See Javadoc for javax.faces.application.ResourceHandler.markResourceRendered() and isResourceRendered().
- Issue ID 1436
  In Render Response, specify how Server Push is utilized.

Smaller Changes

- Issue ID 1422
  UISelectMany detects converter based on first item.
- Issue ID 1007
  Explicit support for dynamic component manipulation
- Issue ID 819
  Add “disabled” attribute for h:button
- Issue ID 1300
  UIViewRoot.getViewMap() and publishEvent().
- Issue ID 1229
  Document UIData.setRowStatePreserved() in VDLDoc and RenderKit Doc.
- Issue ID 1135
  Add PostRenderViewEvent.
- Issue ID 1258
  Clarify text escaping for <h:outputText> or equivalent EL expressions.
- Issue ID 807
  Pass FacesContext to system event listeners.
- Issue ID 1113
  Remove onselect attribute from SELECT components.
- Issue ID 1433
  Add a context-param to enable forcing validation to happen even when there is no parameter corresponding to the current component.

B.4.2.2. Lifecycle

Larger Changes

- Issue ID 790
  javax.faces.ViewState and ajax with cross form submit.

Smaller Changes

- Issue ID 473
  FacesEvent.getFacesContext().
- Issue ID 1241
  faces-config supports client-window-factory.
- Issue ID 1346
  Simplify decoration of FaceletCacheFactory.
- Issue ID 1361
  Correct oversight regarding re-entrancy of flow scoped beans.
- Issue ID 821
  Implement ExternalContext.getRealPath() on startup and shutdown.
- Issue ID 1401
  Explicitly prohibit using NavigationHandler from within ExceptionHandler invoked during RENDER RESPONSE.
- Issue ID 1306
  @FlowScoped should be @NormalScope(passivating=true).
- Issue ID 1382
  Generify return from ExternalContext.getInitParameterMap().
- Issue ID 1329
  @NotNull and <f:viewParam>.
- Issue ID 1403
  Allow entry into flow via <f:viewAction>.
- Issue ID 1216
  Improve consistency in handling PhaseListener instances registered on UIViewRoot components.
- Issue ID 1435
  Add ResourceHandler.getViewResources() method.

B.4.2.3. Platform Integration

- Issue ID 1379
  ExternalContext.getResponseCharacterEncoding() and Portlet 3.0.

B.4.2.4. Facelets/VDL

Larger Changes

- Issue ID 1424
  Add tag <f:importConstants>, see VDLDoc for that tag.

Smaller Changes

- Issue ID 1362

Revisit some cardinality rules regarding <tag> and <component> elements.

- Issue ID 936
  Set FACELETS_REFRESH_PERIOD to -1 if ProjectStage is Production.

B.4.2.5. Spec Clarifications

- Issue ID 1254
  Loosen language regarding the contracts attribute on <f:view>.
- Issue ID 1338
  Clarify pseudocode for resource libraries.
- Issue ID 1279
  Specify UIInput.isEmpty()
- Issue ID 1242
  Remove mentione of OpenAjax hub.
- Issue ID 1215
  Additional warning on DelegatingMetaTagHandler.getTagHandlerDelegate.
- Issue ID 1131
  “name” attribute not required.
- Issue ID 1270
  TagDecorator spec namespace modifications.
- Issue ID 1401
  Advisory text for ExceptionHandler.
- Issue ID 1402
  Explicitly declare that flow eagerness not supported.
- Issue ID 677
  Document automatic UIPanel behavior for f:facet.
- Issue ID 1095
  Description for “rendered” attribute for repeat and fragment.
- Issue ID 1066
  Application.getNavigationHandler() and application element.
- Issue ID 803
  VisitHint.EXECUTE_LIFECYCLE clarifications.
- Issue ID 1217
  EnumConverter.getAsString() clarifications.
- Issue ID 1356
  UIInput.processValidators() clarifications.
- Issue ID 1424
  Public constants for source, behavior, and partial.event. See the Javadocs for javax.faces.component.behavior.ClientBehaviorContext, and javax.faces.conetxt.PartialViewContext.
- Issue ID 1428

API constants for jsf.js and javax.faces in JavaScript.

- Issue ID 1260
  Support for exact mapping of FacesServlet. See ViewHandler Methods that Derive Information From the Incoming Request and ViewHandler Methods Relating to Navigation.

Issue ID 1250
Fix entries in table Jakarta Faces Artifacts Eligible for Injection.

B.4.2.6. Resources

B.4.2.7. Expression Language

B.4.2.8. Configuration and Bootstrapping

B.4.2.9. Miscellaneous

Smaller Changes

- Issue ID 1225
  Clarify requirements to support BCP-47 regarding localization. See Determining the active *Locale*
- Issue ID 1429
  Add constructor to make wrapping easier.
- Issue ID 1430
  Leverage Java SE 8 repeatable annotations where appropriate.

B.4.3. Backward Compatibility with Previous Versions

JSF 2.3 is fully backward compatible with previous releases of JSF, unless any of the following context-parameter values are specified. See Application Configuration Parameters for details.

javax.faces.ALWAYS_PERFORM_VALIDATION_WHEN_REQUIRED_IS_TRUE

javax.faces.DISABLE_FACESSERVLET_TO_XHTML

javax.faces.VIEWROOT_PHASE_LISTENER_QUEUES_EXCEPTIONS.

JSF 2.3 is fully backward compatible with previous releases of JSF unless a CDI managed bean is included in the application with the annotation @javax.faces.annotation.FacesConfig. See the javadocs for that annotation for details.

B.4.4. Breakages in Backward Compatibility

B.5. Changes between 2.1 and 2.2

This section gives the reader a survey of the changes between this version of the specification and the previous version, using the categories from the issue tracker at < https://github.com/jakartaee/faces/issues/ >.

B.5.1. Big Ticket Features

- HTML5 Friendly Markup
  Issue ID 1090
  Start with HTML5 Friendly Markup
- Resource Library Contracts
  Issue ID 1142
  Start with Resource Library Contracts Background.
- Faces Flows
  Issue ID 730
  Start with FlowHandler.
- Stateless Views
  Issue ID 1055
  Start with Stateless Views.

B.5.2. Other Features, by Functional Area

B.5.2.1. Components/Renderers

Larger Changes

- Issue ID 479
  UIData supports the Collection Interface rather than List.
- Issue ID 1134
  Add the "role" pass through attribute.

Smaller Changes

- Issue ID 1080
  Warn about some important corner cases when *UIComponent.findComponent()* may not provide the expected results.
- Issue ID 1068
  New section describing what happens with respect to partial processing during render response. See Render Response Partial Processing.
- Issue ID 1067
  Spec clarifications. See the VDLDoc for *cc:insertChildren*, *cc:insertFacet*
- Issue ID 1061
  Clarify that both *Application.publishEvent()* and the manual traversal based delivery are required for publishing the *PostRestoreStateEvent*.
- Issue ID 1030
  Clarify docs for h:message h:messages
- Issue ID 1023
  Modify JavaDoc to relax requirements for PostAddToViewEvent publishing
- Issue ID 1019
  Modify spec for ResponseWriter.writeURIAttribute() to explicitly require adherence to the W3C URI spec

- Issue ID 997
  javax.faces.component.ComponentSystemEvent: Override isAppropriateListener so that it first asks the listener, "are you a ComponentSystemEventListener", then, if not, asks super.isAppropriateListener()
- Issue ID 984
  Component Context Manager, see *javax.faces.component.visit.ComponentModificationManager*.
- Issue ID 943
  See *javax.faces.view.ViewDeclarationLanguageWrapper*
- Issue ID 784
  Deprecate the CURRENT_COMPONENT and CURRENT_COMPOSITE_COMPONENT attributes
- Issue ID 599
  Make it possible to programmatically create components in the same way as they are created by Facelets. See *javax.faces.application.Application.createComponent(FacesContext, String taglibUri, String tagName, Map attrs)*
- Issue ID 703
  Make "value" optional for @FacesComponent.
- Issue ID 585
  outputText and inputText do not render children by default
- Issue ID 550
  OutputStylesheet "media" attribute
- Issue ID 1125
  javax.faces.application.Application event subscription clarifications.

B.5.2.2. Lifecycle

Larger Changes

- Issue ID 949, 947
  Give JSF the ability to correctly handle browsing context (tab, browser window, pop-up, etc). See *javax.faces.lifecycle.ClientWindow*.
- Issue ID 758 and 1042

  A jsf-api/src/main/java/javax/faces/component/UIViewAction.java

  > The heart of this changebundle, this class came over from the JBoss Seam Faces Module, but I've rewritten most of the javadoc.

  M jsf-api/src/main/java/javax/faces/event/PhaseId.java

  > new methods
  >
  > public String getName()
  >
  > public static PhaseId phaseIdValueOf(String phase)

  Change Default NavigationHandler Algorithm to account for UIViewAction
- Issue ID 1062 and 802

File Upload

- Issue ID 766
  Events from the flash
- Issue ID 1050
  Add support for delay value in options for Ajax requests

Smaller Changes

- Issue ID 1129
  In validate(), clarify that setSubmittedValue() null must be called if validation succeeds.
- Issue ID 1071 Add FlashFactory. See Delegating Implementation Support.
- Issue ID 1065
  When calculating the locale for the resource library prefix, if there is a UIViewRoot, use its locale first, otherwise, just use the Applications's ViewHandler's calculateLocale() method. See Libraries of Localized and Versioned Resources
- Issue ID 1039
  In ApplicationWrapper, mark things as deprecated
- Issue ID 1028
  Deprecate StateManager, point to StateManagementStrategy. In StateManagementStrategy, require the use of the visit API to perform the saving.
- Issue ID 993
  Wrapper for ActionListener
- Tweak circumstances for skipping intervening lifecycle phases in the case of view metadata
  Issue ID 762

  Section 2.2.1. Now has this text.

  > Otherwise, call getViewMetadata() on the ViewDeclarationLanguage instance. If the result is non-null, call createMetadataView() on the ViewMetadata instance. Call ViewMetadata.getViewParameters(). If the result is a non-empty Collection, do not call facesContext.renderResponse(). If the result is an empty collection, try to obtain the metadata facet of the UIViewRoot by asking the UIViewRoot for the facet named UIViewRoot.METADATA_FACET_NAME. This facet must exist. If the facet has no children, call facesContext.renderResponse(). Otherwise, none of the previous steps have yielded the discovery any of metadata, so call facesContext.renderResponse().

- Issue ID 566
  UIOutput.getValue() value returns.
- Issue ID 220
  In web-partialresponse_2_2.xsd, require that the <partial-response> element has an "id" attribute whose value is the return from *UIViewRoot.getContainerClientId()*.

B.5.2.3. Platform Integration

- Issue ID 763
  Change Managed Bean Annotations to account for new injectability requirements.

- Issue ID 976
  In Javadoc for “Faces Managed Bean Annotation Specification For Containers Conforming to Servlet 2.5 and Beyond”, indicate that *javax.faces.bean* will be deprecated in the next version.
- Issue ID 1087
  Introduce CDI based @ViewScoped annotation.

B.5.2.4. Facelets/VDL

Larger Changes

- Issue ID 1001
  Allow cc and non-cc components in one taglib

  A jsf-api/doc/web-facelettaglibrary_2_2.xsd

  First change to the facelet taglib schema in 2.2: introduce the ability to declare a resource which will be the composite component for a tag. Now, before you get all excited about what conventions we can use to make this easier, let me stop you right there. Here is a summary of the ease of use story regarding taglib files.

  > The 80/20 rule says we should make taglib files optional most of the time. Here are the 80% cases.
  >
  > Employs the cc naming convention http://java.sun.com/jsf/composite/<libraryName>
  >
  > The user employs a java component has a @FacesComponent on it that declares the necessary metadata. Issue ID 594

  Here are some of the cases where you must have a taglib file, the 20% cases.

  > If you want to employ a cc with a namespace other than http://java.sun.com/jsf/composite/<libraryName> you need to have a taglib file that declares <composite-library-name>. Currently you must not detlare any <tag> elements in such a taglib file. All the tags in such a library must come from the same resource library.
  >
  > If the user is not employs a java component but is not using @FacesComponent.

  This patch introduces the following syntax.

```
<?xml version="1.0" encoding="UTF-8"?>
<facelet-taglib 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-facelettaglibary_2_2.xsd"
    version="2.2">
  <namespace>test</namespace>
  <tag>
    <tag-name>layout</tag-name>
    <resource-id>myCC/whatever.xhtml</resource-id>
  </tag>
</facelet-taglib>
```

Where <*resource-id*> is a valid resource identifier as specified in section 2.6.1.3.

- Issue ID 548
  Require additional check to handle the case where, in one resource library, there are both localized and non-localized resources. See Libraries of Localized and Versioned Resources.

Smaller Changes

- Issue ID 1038
  Declare an annotation corresponding to the *javax.faces.FACELETS_RESOURCE_RESOLVER* application configuration parameter. See Requirements for scanning of classes for annotations and Application Configuration Parameters.
- Issue ID 1082
  Account for DOCTYPE discrepancy between server side representation of Facelet VDL files, which are proper XHTML, and processed files which are sent to the client, which now, by default, are HTML5 HTML syntax files. See The facelets-processing element.
- Fix incorrect VDLDoc Issue ID 967 f:selectItems itemValue description incorrect.
- Issue ID 922
  Remove maxlength from f:viewParam
- Issue ID 998
  Document that ui:fragment should not be bound to a bean with wider than request scope, document that the type of the property must extend from UIComponent.
- Issue ID 999
  Changes to "template" attribute requiredness for ui:decorate and ui:composition
- Issue ID 901
  Deprecate “targets” concept.
- Issue ID 1088
  add short-name to schema.
- Issue ID 746
  Missing *method-signature* element in taglib xsd.

B.5.2.5. Spec Clarifications

- Issue ID 1136
  In *ExternalContext.dispatch()*, clarify what is to be done if *getRequestDispatcher()* returns *null*.
- Issue ID 1132
  Replace literal strings with symbolic constants where possible.
- Issue ID 1127
  State Saving Serializability concerns.
- Issue ID 1114
  javax.faces.view.facelets.Facelet.apply() spec version reference error.
- Issue ID 1100, 1097
  HTML5 id attribute sensitivity
- Issue ID 1064

In The facelets-processing element, clarify that in XML and JSPX modes, only the CDATA section start and end tags, not the entire CDATA section and contents, should be consumed.

- Issue ID 1063
javax.faces.component.UIViewParameter.getSubmittedValue() return value.
- Issue ID 1058
ui:repeat tag size attribute
- Issue ID 1036
In ValueExpression properties, clarify which attributes are expression enabled,
- Issue ID 1035
Change section FacesMessage to clarify the meaning of having *FacesMessage* instances on the *FacesContext*.
- Issue ID 1026
f:ajax event attribute is String (not VE)
- Issue ID 1014

12.1.3

> The specification for the context-param that declares the list of TagDecorator implementations to the runtime should have always been javax.faces.FACELETS_DECORATORS. Prior to this revision, the name of this context param was incorrectly specified as javax.faces.DECORATORS. The reference implementation has always used the correct name, however.

- Issue ID 1010
Check existing usages of the state saving method parameter to ensure case insensitivity.
- Issue ID 1004

M usingFacesInWebapps.fm

> 12.1.3 Set default for javax.faces.FACELETS_BUFFER_SIZE to be 1024.

- Issue ID 998
Additional clarification about binding attribute in VDLDocs
- Issue ID 915, 1015
Non-normative text about http methods and prefix mapping
- Issue ID 2740
In Component Bindings, clarify a corner case regarding bean scope and component binding.

B.5.2.6. Resources

Larger Changes

- Issue ID 809
This commit introduces a cleaner contract for allowing views to be loaded from the Filesystem (including inside of JAR files). All VDL resources must be loaded using ResourceHandler.createViewResource().

Smaller Changes

- Issue ID 996
  Enable configuring the web app resources directory. See Application Configuration Parameters.
- Issue ID 719
  Method to map a viewId to a resourcePath
- Issue ID 1130
  Modify State Saving Alternatives and Implications to clarify there is no requirement to serialize server state by default when state saving method is server. Introduce a context parameter to this effect in Application Configuration Parameters

B.5.2.7. Expression Language

- Issue ID 1092

  Remove text from MethodExpressionValueChangeListener and MethodExpressionActionListener regarding wrapping any exception thrown by the invoked method being wrapped in an AbortProcessingException. Such a requirement is incorrect and should not have been introduced.

  In section ExceptionHandler, add *AbortProcessingException*, to the list of exceptions that do not get to the *ExceptionHandler*.

- Issue ID 1043
  When publishing a ComponentSystemEvent, ensure the EL current component is pushed correctly
- Issue ID 1164
  If running in a container that supports EL 3.0, add the necessary ELResolvers. See Stream, StaticField, Map, List, Array, and Bean ELResolvers.

B.5.2.8. Configuration and Bootstrapping

- Issue ID 533
  Modify Application Startup Behavior to account for requirement to call new API when starting up.

B.5.2.9. Miscellaneous

- Issue ID 1169

  New XML Namespace for XSD files introduced in JSF 2.2, and also for facelet tag libraries.

  The following XSD files are new for JSF 2.2, and each will be in the XML namespace *http://xmlns.jcp.org/xml/ns/javaee*.

  web-facelettaglibrary_2_2.xsd

  web-facesconfig_2_2.xsd

  web-partialresponse_2_2.xsd

Facelet Tag Libraries will now respond to the following URIs

Library	Old URI	New URI
Composite Components	http://java.sun.com/jsf/composite	http://xmlns.jcp.org/jsf/composite
Faces Core	http://java.sun.com/jsf/core	http://xmlns.jcp.org/jsf/core
HTML_BASIC	http://java.sun.com/jsf/html	http://xmlns.jcp.org/jsf/html
JSTL Core	http://java.sun.com/jsp/jstl/core	http://xmlns.jcp.org/jsp/jstl/core
JSTL Functions	http://java.sun.com/jsp/jstl/functions	http://xmlns.jcp.org/jsp/jstl/functions
Facelets Templating	http://java.sun.com/jsf/facelets	http://xmlns.jcp.org/jsf/facelets
Pass Through Attributes	http://java.sun.com/jsf/passthrough	http://xmlns.jcp.org/jsf/passthrough
Pass Through Elements	http://java.sun.com/jsf	http://xmlns.jcp.org/jsf

Developers are requested to use the values from the New URI column, though both columns will work.

- Issue ID 997

M jsf-api/src/main/java/javax/faces/event/ComponentSystemEvent.java

> Override isAppropriateListener so that it first asks the listener, "are you a ComponentSystemEventListener", then, if not, asks super.isAppropriateListener()

M jsf-api/src/main/java/javax/faces/event/SystemEvent.java

> in isAppropriateListener(), document the default implementation.

M jsf-api/src/main/java/javax/faces/component/UIComponent.java

> Make inner class ComponentSystemEventListenerAdapter implement ComponentSystemEventListener.

- Issue ID 917

javax.faces.application.ResourceWrapper:

> getContentType()
>
> getLibraryName()
>
> getResourceName()
>
> setContentType(String)
>
> setLibraryName(String)
>
> setResourceName(String)

javax.faces.context.ExternalContextWrapper:

getSessionMaxInactiveInterval()

isSecure()

setSessionMaxInactiveInterval()

javax.faces.context.PartialViewContextWrapper

setPartialRequest(boolean)

- 12.1.3 add this text to the javax.faces.STATE_SAVING_METHOD spec. When examining the value, the runtime must ignore the case.
- Add ExternalContext.getApplicationContextPath() Issue ID 1012
- Issue ID 787
  restore ViewScope before templates are processed with buildView()
- 7.6.2.6 typo in spec for renderView(). Should be
  Return immediately if calling isRendered() on the argument UIViewRoot returns false.
- Per Matt Benson, remove duplicate descriptions of the cc and component implicit object from the getValue() specification for the composite component ELResolver in section 5.6.2.1.
- Issue ID 869

  Specify Cross Site Request Forgery protection.

  Add text in Restore View to describe how non-postback requests are inspected for protection, if necessary.

  Remove text for writeState() in ResponseStateManager, point to the JavaDocs.

  Add View Protection methods to Overview for ViewHandler.

  Introduce subsections to Default ViewHandler Implementation that group the methods by their purpose. Add a new section ViewHandler Methods that relate to View Protection, which points to the javadocs. In ViewHandler Methods Relating to Navigation, in the spec for *getActionUrl()*, specify how view protection is affected.

- Remove tables in section Requirements for scanning of classes for annotations
  The Javadoc tool lists annotations in a separate section, making the tables that list JSF related annotations redundant.
- Issue ID 1082
  Add new section DOCTYPE and XML Declaration.
- Issue ID 1141
  In Resource Identifiers, declare that resourceName and resourceVersion, in addition to the already listed other segments, are subject to the same constraints.
- Issue ID 1129
  In Render Response Partial Processing, require calling *UIViewRoot.resetValues()* if necessary.

B.5.3. Backward Compatibility with Previous Versions

Faces 2.2 is backwards compatible with Faces 2.1 and 2.0. This means that a web-application that was developed to run with Faces 2.1 or 2.0 won't require any modification when run with Faces 2.2 except in the cases described in the following section.

B.5.4. Breakages in Backward Compatibility

- Issue ID 1092
  Due to an error in previous versions of the specification, exceptions were silently being swallowed that now will bubble up to the exception handler. Code that was relying on this incorrect behavior may need to be modified to account for fixing this problem.
- Issue ID 745

  5.6.2.2 Table 5-11. Make the following changes to the spec for Composite Component Attribute ELResoler

  > Modify getType() according to new specification language.
  >
  > Require the implementation of containsKey() on the Map.
  >
  > - containsKey(): If the attributes map contains the key, return true. Otherwise, if a default value has been declared for the attribute, return true. Otherwise, return false.

  Composite Component Metadata

  New text dealing with <cc:attribute>

  > Any additional attributes on *<cc:attribute/ >* are exposed as attributes accessible from the *getValue()* and *attributeNames()* methods on *PropertyDescriptor*. The return type from getValue() must be a *ValueExpression* with the exception of the *getValue("type")*. The return type from *getValue("type")* must be *Class*. If the value specified for the *type* attribute of *<cc:attribute/>* cannot be converted to an actual *Class*, a *TagAttributeException* must be thrown, including the *Tag* and *TagAttribute* instances in the constructor.

Yes, this is a backwards incompatible change, but because the usecase is so specific, and the performance benefit so substantial, it was judged to be worth the cost.

B.6. Changes between 2.0 Rev a and 2.1

B.6.1. Facelet Tag Library mechanism

Document that the unprefixed namespace must pass through.

B.6.2. New feature: <facelets-processing>

See Required Handling of *-extension elements in the application configuration resources files

B.6.3. Update schema for 2.1

The only new element is <facelet-cache-factory>. See the full schema in the Javadoc section of the documents.

B.6.4. Change Restore View Phase

Change Restore View to require a call to *ViewHandler.deriveLogicalViewId()* before trying to find the *ViewDeclarationLanguage*.

B.6.5. Default ViewHandler Implementation

Document *deriveLogicalViewId()*.

B.7. Changes between 2.0 Final and 2.0 Rev a

B.7.1. Global changes

B.7.1.1. ExceptionQueuedEvent

The specification incorrectly refers to the ExceptionQueuedEvent as the ExceptionEvent. All instances should be replaced, as there is no such class ExceptionEvent in JSF.

B.7.1.2. Usage of the term "page" in the JSF 2.0 spec

This might be kind of nit-picky, but there are several occurrences of the term "Facelet page" in the JSF 2.0 spec, but I'd like to set forth the argument that the term "Facelet document" or "Facelet view" would be more appropriate, depending on context. Similarly, the spec uses the term "Composite component markup page" which isn't always appropriate either. Better to call it a "Composite component markup document" or something like that.

All Facelet XHTML files are documents, but not all Facelet XHTML files are pages. There is a built-in bias to the word "page" that assumes the markup output will be rendered as a "web page" which is not always the case. In the case of portlets, the rendered output is a fragment of markup (normally a ...). In the case of a custom renderkit, the rendered output might be some mobile device. In the case of ICEfaces, the rendered markup is a server-side DOM tree. In the case of a composite component, a Facelet XHTML file is not a page, but a "Composite Component markup document" (or definition) file.

For example.. Instead of a "Facelet Page", I think the following should be called a "Facelet Document" or a "Facelet View" (since the f:view tag is optional, but implied)

```
<!DOCTYPE html>
<html xmlns:f="jakarta.faces.core"
      xmlns:h="jakarta.faces.html">
  <h:head>
    ...
  </h:head>
  <h:body>
```

```
    ...
  </h:body>
</html>
```

But in the case of Portlets, the <html> , <head>, and <body> tags are forbidden. The equivalent "Facelet Document" or "Facelet View" for portlets would look like this:

```
<f:view xmlns:f="jakarta.faces.core"
        xmlns:h="jakarta.faces.html">
  ...
</f:view>
```

B.7.2. Front Matter

Change Sun logo to Oracle Logo. Sun postal address to Oracle postal address, and Sun phone number to Oracle phone number

Update spec license.

B.7.3. Chapter 2

B.7.3.1. Restore View

Per Andy Schwartz recommendation, loosen the spec requirements for the delivery of the PostRestoreStateEvent to be "somewhere during RestoreView".

B.7.3.2. Localized Application Messages

Suggestion: Change to

Validation Error: Length is less than allowable minimum of 5.

B.7.3.3. JSR 303 Bean Validation

Change "leas" to "least"

B.7.3.4. JSR 303 Bean Validation needs to reference "Bean Validation Integration" section

While reading section 2.5.7, one becomes very disappointed with the limited about of information that it provides. But section 3.5.6 provides more information, so the recommendation is that these sections reference each other, or perhaps are combined in some way if that makes sense.

B.7.3.5. Resource Identifiers

Tighten spec for the localePrefix, libraryName, and resourceVersion segments of the resource identifier.

B.7.4. Chapter 3

B.7.4.1. Clarify meaning of "javax.faces.bean" in Bean Validator Activation

Section 3.5.6.1 says:

"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()"

But the reader of the Spec has no idea what the "javax.faces.Bean" standard validator is, within the context of this paragraph. Recommend adding some verbiage that says that javax.faces.Bean is the validatorId of the standard JSR 303 validator of the JSF 2.0 API, which equates to the javax.faces.validator.BeanValidator class as mentioned in Section 10.4.1.4.

B.7.4.2. Need to be consistent between Declarative Listener Registration of the JSF 2.0 Spec and the VDLDoc for f:event

Section 3.4.3.4 of the JSF 2.0 Spec reads:

The method signature for the MethodExpression pointed to by the listener attribute must match the signature of

javax.faces.event.ComponentSystemEventListener.processEvent().

And the VDLDocs for f:event read:

(signature must match public void listener(javax.faces.event.ComponentSystemEvent event) throws javax.faces.event.AbortProcessingException)

Both of these are true, and indeed saying the same thing. But I think it would be helpful to say BOTH things, in BOTH documents.

B.7.4.3. Typo in Declarative Listener Registration of the JSF 2.0 Spec regarding "beforeRender"

Section 3.4.3.4 of the JSF 2.0 Spec has this example:

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

But "beforeRender" is a typo. The correct value is "preRenderComponent" as stated in the f:event VDLDocs.

B.7.4.4. Validation Registration, What does it mean to be a JSF User Interface component?

Remove references to UInput.encodeEnd(). Not for a very long time has this method been used to instigate validation.

B.7.4.5. Composite Component Metadata

Section 3.6.2.1 of the Spec indicates that the "hidden" property of the javax.bean.FeatureDescriptor is to appear as an attribute for tags like cc:actionSource, cc:attribute, cc:facet, and cc:interface but the VDLDocs do not declare that the "hidden" property is available.

B.7.5. Chapter 4

B.7.5.1. Events

Remove text pertaining to PostRestoreStateEvent, rely instead on text in setion 2.2.1.

B.7.6. Chapter 7

B.7.6.1. Overview

getNavigationCase should return NavigationCase and not void.

B.7.6.2. Default NavigationHandler Algorithm

Add faces-include-view-params

B.7.6.3. Default ViewHandler Implementation

Fix typo the specified createView() should be called in renderView() and restoreView().

B.7.7. Chapter 10

Need to change "confirm with" to "conform with".

Confusing verbiage in table 10-1 of JSF 2.0 spec regarding the word "template"

In table 10-1, it correctly says that "page templating" is not a feature of JSP.

But later on in the table, it says "Expressions in template text cause unexpected behavior when used in JSP."

Somehow there needs to be an explanation of the distinction of "page templating" and "template text" here. Right now it kind of reads as a contradiction.

B.7.7.1. General Requirements

Add an assertion to section 10.3.1 stating that EL expressions that appear in the facelet XHTML page must appear in the rendered view as if they were the right hand side of the value attribute of an at the same point in the page as the EL expression

B.7.7.2. Facelet Tag Library mechanism

Section 10.3.2. Correct xref to point to section in appendix that includes the facelet taglib schema.

Correct xref to point to section in appendix that includes the facelet taglib schema.

B.7.7.3. VDLDocs

See Facelet Templating Tag Library, and Jakarta Tags Core and Function Tag Libraries. Refer the reader to the "VDLDocs" (View Declaration Language Documentations).

B.7.8. Chapter 13

B.7.8.1. Redundancy in Partial View Processing of the JSF 2.0 Spec

Section 13.4.2 of the JSF 2.0 spec has this sentence:

The request contains special parameters that indicate the request is a partial execute request or a partial execute request that was triggered using Ajax

This needs clarification—does this mean to say:

partial execute request (not triggered by Ajax) or a partial execute request (that was triggered using Ajax)

B.7.8.2. "Execute portions" of the JSF request processing lifecycle in the JSF 2.0 Spec

Section 13.4.2 reads:

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.

Why does the diagram include the INVOKE_APPLICATION phase if it's not "really" considered to be part of the execute portions?

B.7.9. Chapter 14

B.7.9.1. Initiating an Ajax Request Typo in table 14.2.2 of the JSF 2.0 Spec

Table 14.2.2 reads:

"execute" phase

But in order to be consistent with the rest of the spec, it should read:

"execute" portion

Also, the same goes for "render" in that the word "portion" should be used instead of "phase".

B.7.9.2. Request/Response Event Handling Table 14.4.1

Change responseTxt to responseText.

Table 14.3: Reorder rows

B.7.10. Appendix A Metadata

Update schema to remove partial-traversal, as well as fixing 768.

B.7.11. VDLDoc changes

B.7.11.1. Typo in f:selectItems VDLDocs

Change "mest" to "must"

B.7.11.2. Need clarification on execute attribute of f:ajax

The VDLDocs for f:ajax say "Identifiers of components" but, for some reason, it wasn't obvious to me that this term referred to the "id" attributes of components. The recommendation isthat this be clarified to say "list of id attribute values" instead. Also, the value of an id attribute like "mycomponent" or something should be added to an example that includes a keyword… something like this: "@this componentone componenttwo"

B.7.11.3. Spelling error in VDLDocs for f:ajax

This one from Lincoln:

See the "onerror" attribute

There is an extra 'e' → "oneerror"

B.7.11.4. Need clarification on required attribute in VDLDocs for tags that got a new "for" attribute in JSF 2.0

The VDLDocs correctly have green for the new "for" attribute, but the "required" column says false, when that's not always the case.

For example, with f:actionListener the VDLDocs say that that it is not required. However, when the tag is used as a child of a Composite Component, then the for attribute is indeed required. This would be true of all tags like that, such as f:convertDateTime, f:convertNumber, etc.

B.7.11.5. Uppercase typo in VDLDocs for f:event

Change uppercase "P" to lowercase for: PostAddToView for the f:event VDLDocs

B.7.11.6. Need to change "JSP" to "Facelets" in "Body Content" of VDLDocs

Search for "JSP" on the f:event VDLDocs. My suspicion is that this is a problem across the board.

B.7.11.7. Need clarification in VDLDocs for f:metadata

In the VDLDocs for f:metadata, recommend changing:

"This tag must reside within the top level XHTML file for the given viewId, not in a template"

to this:

"This tag must reside within the top level Facelet view, or in a template-client, but not in a template"

Also, it needs to be clarified that the page01.xhtml example is a template-client. So recommend changing this:

"viewId XHTML page, page01.xhtml"

to this:

"template-client XHTML page, page01.xhtml"

B.7.11.8. Missing description in VDLDocs for name attribute of f:viewParam

The VDLDocs for f:viewParam are missing documentation of the "name" attribute, which is pretty important since it is required.

B.7.11.9. VLDDocs on "for" attribute of f:viewParam claim it can be used in a CC

The VDLDocs for f:viewParam claim that the "for" attribute is supported. I just checked Mojarra's jsf-api and UIViewParameter.java does not support the "for" attribute, since it does not have a getter/setter for "for" like HtmlOutputLabel does. There are restrictions on f:viewParam such that it may only be used inside of f:metadata, and f:metadata may only be used inside of f:view. So that disqualifies the f:viewParam tag from being able to be used inside of a Composite Component. Therefore I recommend that the documentation of the "for" attribute be totally removed.

B.7.11.10. Miscellaneous VDLDoc items

- VDLDocs for "execute" attribute of f:ajax say (must evaluate to java.lang.Object) but then say "Evaluates to Collection"
- VDLDocs f:selectItem lists the new JSF 2.0 "noSelectionOption" but is not colored green to indicate "new in JSF 2.0" and the link for f:selectItem in the navigation frame needs to be orange
- VDLDocs for f:validateBean should have all of its attributes in green to indicate "new in JSF 2.0" since it is a new tag.
- VDLDocs for f:validateRegex has a typo in the description which reads "RegexpValidator" rather than "RegexValidator"
- In spec, the "Changes between 1.2 and EDR2" section refers the reader to section 3.5.2 for the addition of "javax.faces.RegularExpressionValidator" but actually it should be section 3.5.5
- VDLDocs for h:button say that the outcome attribute is not required, but really it should be required otherwise there is no purpose of f:button—you would end up navigating back to the current view. The whole point of f:button is to perform navigation to a different view that potentially contains view parameters. Why have a bookmarkable URL back to itself?
- VDLDocs for h:button don't mention a disabled attribute, but the h:link one does have the disabled attribute. My guess would be that both should have this attribute?
- VDLDocs for h:outputScript and h:outputStylesheet should indicate that even though the UIOutput class implements the ValueHolder interface, the coverter and value attributes are basically ignored by the renderers, since the value attribute has no meaning. This is basically a design flaw—a new class named UIOutputResource should have been created instead of

UIOutput being reused.

- VDLDocs for h:outputScript and h:outputStylesheet should indicate that the "name" attribute is required, since section 2.6.1.3 implies that this is the case with the following resource pattern:
- VDLDocs for h:outputScript needs to have all the possible values for the target attribute documented. I think the only valid values are "head", "body", and "form"
- VDLDocs for h:graphicImage has a dead hyperlink to "Common Algorithm for Obtaining A Resource to Render
- VDLDocs for h:selectManyCheckbox indicate orange for the collectionType and hideNoSelectionOption attributes but they should be green to indicate "new in JSF 2.0"
- VDLDocs for h:selectManyCheckbox says that the return type must evaluate to a String, but that's not entirely true. It can also evaluate to a concrete class that implements java.util.Collection
- VDLDocs for ui:param have two "name" attributes specified. The second one should be the "value" attribute

B.7.11.11. Should TLDDocs now be VDLDocs?

The Spec introduces this term VDLDocs (which as I said in the other email, was formerly PDLDocs), but it also refers the reader to the TLDDocs. Should we just settle on VDLDocs as the standard term throughout the Spec?

B.7.11.12. Typo in VDLDocs for f:event.

The VDLDocs for f:event specify a "name" attribute, but the Description column of the page talks about a "type" column (not "name"), which would be consistent with Section 3.4.3.4 of the Spec which talks about a "type" column.

Jim Driscoll verified that there is a doc bug in the VDLDocs for f:event and that the "name" attribute is actually "type"

B.7.12. Accepted Changes from JCP Change Log for JSF 2.0 Rev a

The referenced spec public issue number can be found in the issue tracker at https://github.com/jakartaee/faces/issues/

ID	Category	Description	Fixed in Source Code Repository of Specification	Issue
C002	Errata	Section 5.6.2.2 is out of sync with the current resolver implementation.	yes	*Issue ID:848*
C004	Errata	RenderKitDoc for *OutcomeTarget* Renderers are incorrect with respect to intended design. Refer to *appendixB-changelog.html#UNKNOWNChangeC006* in the footnotes section below.	yes	*Issue ID:823*

C007	Errata	Section 11.4.7 "Ordering". After the sentence "The <others /> element must be handled as follows" add a bullet point: "The <others /> element represents a set of application configuration resources. This set is described as the set of all application configuration resources discovered in the application minus the one currently being processed, minus the application configuration resources mentioned by name in the <ordering /> section. If this set is the empty set, at the time the application configuration resources are being processed, the <others > element must be ignored."	yes	*Issue ID:824*
C008	Errata	taglib docs for cc:interface are missing documentation for *componentType* attribute.	yes	*Issue ID:849*
C011	Errata	Section 3.6.2.1 "Composite Component Metadata". Add *BehaviorHolderAttachedObjectTarget* after iii. *ActionSource2AttachedObjectTarget*	yes	*Issue ID:825*
C012	Errata	Javadocs for *ResourceHandler.createResource(String resourceId)* need to me ammended to state that if there is an error in argument resourceId, null must be returned by this method.	yes	*Issue ID:851*
C013	Errata	PDL DOCS: f:event listener attribute clarification:Change description to: "A method expression that JSF invokes when an event occurs. That event is specified with the name attribute."	yes	*Issue ID:586*
C015	Errata	*UIViewRoot.setBeforePhaseListeners()* removed the statement that all phases including *RestoreView* will have their *beforePhaseListeners* called. Reverted to the way it was in 1.2	yes	*Issue ID:826*
C016	Errata	Section 2.6.2.1 Relocatable Resources: code snippet: <f:view..../> should be <f:view...>	yes	*Issue ID:565*
C017	Errata	*UISelectItem* doesn't mention *itemEscaped*.	yes	*Issue ID:430*
C018	Errata	*ViewDeclarationLanguage.retargetAttachedObjects()* misses talking about Behaviors	yes	*Issue ID:827*
C021	Errata	ui:insert missing existing "name" attribute, implemented, tested, but not documented	yes	*Issue ID:667*
C022	Errata	*f:valueChangeListener* missing "for" attribute. Implemented, tested, but not documented	yes	*Issue ID:828*
C023	Change	in facelets VDLdoc, mark f:verbatim and f:subview as deprecated	no	*Issue ID:852*

C024	Errata	Add an assertion to section 10.3.1 stating that EL expressions that appear in the facelet XHTML page must appear in the rendered view as if they were the right hand side of the value attribute of an <*h:outputText*> at the same point in the page as the EL expression	yes	*Issue ID:829*
C027	Errata	web-facelettaglibrary_2_0.xsd type incorrect for composite-library-name. Should be **javae:string**	no	*Issue ID:854*
C028	Errata	*ui:remove* VDLDoc has attribute with no name	no	*Issue ID:842*
C029	Errata	*ui:param* has attribute duplicated. One of them should be "value"	yes	*Issue ID:855*
C030	Errata	RenderKit Docs *javax.faces.CompositeFacet* change " The implementation of *encodeBegin()*, must obtain " to be " The implementation of *encodeChildren()*, must obtain "	no	*Issue ID:843*
C031	Errata	VDL docs state that *cc:attribute* has a target attribute with *required="true"*. This attribute is not really required on *cc:attribute*.	yes	*Issue ID:644*
C032	Errata	Mention in spec that Objects put in view scope may need to be Serializable	yes	*Issue ID:830*
C033	Errata	Modify the javadoc for *ResourceHandler* to state that for resources residing at *META-INF/resources/*<*resourceidentifier*>. The implementation is not required to support the optional *libraryVersion* and *resourceVersion* segments	yes	*Issue ID:844*
C034	Change	Modify table 5-10 to state that implicit object cc returns the current composite component, relative to the composite component markup page in which the expression appears	yes	*Issue ID:831*
C035	Errata	3.6.2.1 Modify composite component metadata specification to state that, within the *cc:interface* element, the following attributes are not available unless *ProjectStage* is *Development*: displayName, expert, hidden, preferred, shortDescription	yes	*Issue ID:832*
C037	Errata	*UIComponent.restoreState()* javadocs must be changed to **say** NPE is thrown if context is null, but no action is taken if state argument is null	yes	*Issue ID:845*
C040	Errata	VDLDocs for *f:metadata*. Don't mention *f:view*. State, "This tag must reside within the top level facelet page whose filename corresponds ot the *viewid* being loaded."	yes	*Issue ID:856*

C043	Errata	Document *SEPARATOR_CHAR* in section 11.1.3 where all the other context-params are documented	yes	*Issue ID:833*
C044	Errata	Section 11.4.6 doesn't include *ViewDeclarationLanguage*, *VisitContextFactory*, *ExceptionHandlerFactory*, *PartialViewContext*, *TagHandlerDelegateFactory* as decoratable	yes	*Issue ID:834*
C046	Errata	Section 10.4.1.1 specifies the use of *AjaxBehaviors pushBehavior* but *AjaxBehaviors* is an implementation detail (the class exists under com package)	yes	*Issue ID:836*
C047	Errata	Add "defaults" for "execute", "render" *AjaxBehavior* in VDLDocs.	yes	*Issue ID:568*
C048	Errata	JSP should not have *f:viewParam*. Facelets *f:viewParam* must have *name* attribute.	yes	*Issue ID:656*
C049	Errata	Spec section 3.2.5 is empty. Fix that	yes	*Issue ID:835*
C050	Errata	Spec for *UIComponent.setParent()* incomplete	yes	*Issue ID:837*
C054	Errata	*f:event name* attribute should be type.	yes	*Issue ID:639*
C058	Errata	Section 14.4.1: Table 14-4: responseTxt should be responseText. Table 14-4: Add *status* property; Table 14-4: There is no "name" property. Table 14-3: reorder "status" values to be in chronological order. Section 14.4.1.1: Fix use case.	yes	*Issue ID:642*
C060	Change	Replace the last sentence in the javadoc for *FacesServlet.service()* to say "The implementation must make it so *FacesContext.release()* is called within a finally block as late as possible in the processing for the JSF related portion of this request".	yes	*Issue ID:846*
C061	Change	Non-normatively document that JavaBeans PropertyEditors will be used for EL Coercion. Mention this in the context of JSF converters	yes	*Issue ID:838*
C062	Change	In 3.1.5, explicitly mention not to use view scope	yes	*Issue ID:839*
C063	Change	7.4.1 Clarify that, in the case of navigation actions, an empty string should be treated the same way as null: stay on the same page.	yes	*Issue ID:747*
C064	Change	Correct *StateHolder.setTransient* JavaDoc (specified backwards)	yes	*Issue ID:840*

C065	Change	Correct typos in *Composite.tld* (for pdldocs). Specifically, quotes around *actionListener*, *method-signature* (spelling). Also clarify the default value "false" for "required" attribute.	yes	*Issue ID:841*
C066	Change	Specify that the Component Resource container facet must be marked transient. Specifically, the JavaDocs for *UIViewRoot.getComponentResources* should include: "Set the transient property of the facet to true."	yes	*Issue ID:800*
C068	Change	Modify the facelet taglib xsd so that older versions of taglibs are acceptable.	yes	*Issue ID:744*
C069	Change	Make sure VDLDocs for *f:event* list event all possible event types	yes	*Issue ID:712*
C072	Errata	Neither applyNextHandler of DelegatingMetaTagHandler or nextHandler of TagHandler are documented.	yes	*Issue ID:780*
C073	Errata	Specify f:ajax execute/render id behavior in VDLDocs (as outlined in Section 10.4.1.1 of the spec).	yes	*Issue ID:567*
C074	Errata	<view-param> has no business being a child of <redirect> and should be renamed to <redirect-param>.	yes	*Issue ID:698*
C075	Errata	includeViewParams implicit navigation flag should be faces-include-view-params.	yes	*Issue ID:699*
C077	Errata	Event broadcasting should apply to Behaviors (not just ClientBehaviors).	yes	*Issue ID:798*
C078	Errata	PostAddToViewEvent delivery specification needs clarification. Clarify UIComponent.getParent and getChildren for consistency.	yes	*Issue ID:805*
C079	Errata	RenderKit Docs - TableRenderer:Clarification - the docs say to render the footer the same as the header which causes the problem.	yes	*Issue ID:255*
C080	Errata	RenderKit Docs - ButtonRenderer Encode behavior w/r/t onclick attribute - should not be passthrough attribute.	yes	*Issue ID:257*
C081	Errata	h:message "for" attribute is mis-specified:"for" attribute should be relative id (not clientid).	yes	*Issue ID:266*
C082	Errata	clarify whether expression of binding-attribute of f:xxxxListener should be evaluated on postback.	yes	*Issue ID:320*
C083	Errata	Option rendering, specifically when dealing with SelectItemGroups, is too generic.	yes	*Issue ID:420*

C084	Errata	submittedValue get/set methods underspecified	yes	*Issue ID:434*
C085	Errata	Current wording in renderkit docs leads to double encoding of query parameters	yes	*Issue ID:436*
C086	Errata	SelectManyCheckBox Clarification	yes	*Issue ID:466*
C087	Errata	PDL document for JSTL(Facelets) has the incorrect URI for the NameSpace.	yes	*Issue ID:509*
C088	Errata	API docs missing for ExceptionEventContext.	yes	*Issue ID:515*
C089	Errata	cc:attribute component documentation for the attribute type should be for attribute method-signature.	yes	*Issue ID:524*
C090	Errata	The UML Diagram for javax.faces.event is out of date.	yes	*Issue ID:525*
C091	Errata	Minor typo in the Interface BehaviorHolder API.	yes	*Issue ID:534*
C092	Errata	Two references to the itemLabelEscaped attribute.	yes	*Issue ID:536*
C093	Errata	Missing class description for javax.faces.event.PostValidateEvent and javax.faces.event.PreValidateEvent.	yes	*Issue ID:537*
C094	Errata	Section 3.7.5 typo - ClientBehavorHolder should be ClientBehaviorHolder.	yes	*Issue ID:540*
C095	Errata	Section 4.1.3 typo - NamingContaier should be NamingContainer.	yes	*Issue ID:541*
C096	Errata	API Docs: Application.publishEvent: Docs say to throw NPE if any of the arguments is null. However, sourceBaseType arg can be null.	yes	*Issue ID:553*
C097	Errata	Facelets TLD Docs: Missing "for" attribute for "message" and "messages" tags.	yes	*Issue ID:558*
C099	Errata	ResourceHandler docs: Clarify that relative paths are disallowed in library names.	yes	*Issue ID:577*
C100	Errata	Renderkit Docs: h:link - Formatting - add paragraphs	yes	*Issue ID:588*
C101	Errata	Spec Section 2.5.9: Fix Grammar: "The first client behavior to provided by the JSF specification is the AjaxBehavior." should be: "The first client behavior provided by the JSF specification is the AjaxBehavior."	yes	*Issue ID:590*
C102	Errata	Spec Section 9.4 doesn't list all the validation tags and it lists the validateDoubleRange tag twice.	no	*Issue ID:591*

C103	Errata	VDLDocs and Spec section 3.6.2.1 have component:actionSource target attribute with commas as delimiters - should be "space" as delimiter.	yes	*Issue ID:592*
C104	Errata	Spec Section 7.4.1: getNavigationCase should return NavigationCase and not void.	yes	*Issue ID:605*
C105	Errata	Spec Section 10.4.1.4 says: f:validateBean should extend validateHandler. Should be ValidatorHandler.	yes	*Issue ID:615*
C106	Errata	Typo: Pages in the TLD docs says "JSF 2.0 Page Decraration Language". Should be "Declaration".	yes	*Issue ID:617*
C107	Errata	Typos: Table 14-1, 14-2, page 14-3.	yes	*Issue ID:629*
C108	Errata	UIData.invokeOnComponent docs need to be updated to include handling of column level facets.	yes	*Issue ID:632*
C109	Errata	Spec Section 3.5.6.1 needs to be corrected to state that default validators are added during tag execution time.	yes	*Issue ID:635*
C110	Errata	validateBean and validateRequired tags need to be removed from the Jakarta Server Pages PDL documentation	yes	*Issue ID:645*
C111	Errata	faces.ajax.response update element clarification needed in JavaScript docs.	yes	*Issue ID:646*
C112	Errata	Spec Section 10.4.1.1: Clarify what happens when nesting and wrapping f:ajax tags collide.	yes	*Issue ID:652*
C113	Errata	Typo: Spec Section 8.3.1: "renderkit-id" should be "render-kit-id" and "renderkit" should be "render-kit".	yes	*Issue ID:660*
C114	Errata	Add "rendered" attribute to VDL docs for ui:component and ui:fragment.	yes	*Issue ID:661*
C115	Errata	JavaDocs for UIComponent.processValidators is incomplete. It should mention *popComponentFromEL*.	yes	*Issue ID:664*
C116	Errata	Dead link in VDL docs.	yes	*Issue ID:666*
C117	Errata	Spec Section 2.5.2.4: Standard messages for LengthValidator are confusing.	yes	*Issue ID:668*
C118	Errata	Spec/pdldocs don’t say what the default is for "target" in h:outputScript.	yes	*Issue ID:673*
C119	Errata	partial-view-context-factory is only mentioned in the schema part of the spec. Houls be added to Spec Section 13.4.2.	yes	*Issue ID:705*
C120	Errata	Specification edits needed	no	*Issue ID:714*

C121	Errata	Typo - Spec Section 7.5.2: "ViewHanlder" should be "ViewHandler"; "renderView" and "restoreView" methods should call "ViewDeclarationLanguage.renderView" and "ViewDeclarationLanguage.restoreView".	yes	*Issue ID:729*
C122	Errata	Spec Section 2.6.1.3: Specify that a libraryName or resourceName contains only XML NameChar, but not a colon; a libraryName or resourceName does not match the regex "[0-9]+(_[0-9]+)* or [A-Za-z]{2}(_[A-Za-z]{2}(_[A- Za-z]+)*)?"	yes	*Issue ID:740*
C123	Errata	Typos in PDLDocs for ui:repeat	yes	*Issue ID:743*
C124	Errata	Remove "partial-traversal" application element from the spec as it does not exist in the schema.	yes	*Issue ID:767*
C125	Errata	Add mssing ID attributes to schema for: faces-config-orderingType,faces-config-ordering-orderingType,faces-config-absoluteOrderingType,faces-config-default-valueType,faces-config-from-view-idType,faces-config-client-behavior-rendererType,faces-config-behaviorType,faces-config-value-classType,faces-config-rendererType	yes	*Issue ID:768*
C126	Errata	UIInput JavaDocs: Specify the handling of conversion failures.	yes	*Issue ID:775*
C127	Errata	EditableValueHodler JavaDocs: Missing "@Since 2.0" for "resetValue" method.	yes	*Issue ID:779*
C128	Errata	VDL documentation for f:selectItem references the "escape" attribute. It should be "itemEscaped".	yes	*Issue ID:788*
C129	Errata	Specify description for "f:param" "disabled" attribute.	yes	*Issue ID:794*
C130	Errata	Simplify PostRestoreStateEvent delivery requirements.	yes	*Issue ID:806*

B.8. Changes in versions below 2.0 Final

See the "JavaServer™ Faces Specification Version 2.3" for these changes. The change log items are removed from this document in Jakarta Faces 3.0.

---

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

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

[3] It is an error to specify more than one <navigation-case>, nested within one or more <navigation-rule> elements with the same <from-view-id> matching pattern, that have exactly the same combination of <from-xxx>, unless each is discriminated by a unique <if> element.

[4] The presence of the <if> element in the absense of the <from-outcome> element is characterized as an alternate, contextual means of obtaining a logical outcome and thus the navigation case is checked even when the application action returns a a null (or void) outcome value.

[5] Note that multiple conditions can be checked using the built-in operators and grouping provided by the Expression Language (e.g., and, or, not).

[6] Or, equivalently, with no <*from-view-id*> element at all.

[7] The implementation classes for attached object must include a public zero-arguments constructor.

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

[9] 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.

[10] If you need multiple components in a facet, nest them inside a <h:panelGroup> tag that is the value of the facet.

[11] This component has no associated Renderer, so the getRendererType() method must return null instead of a renderer type.

[12] Identified by XPath selection expressions.