ICEfaces

Developer's Guide

Copyright

ICEsoft Technologies, Inc.
Suite 300, 1717 10th Street NW
Calgary, Alberta, Canada
T2M 4S2

Toll Free: 1-877-263-3822 (USA and Canada)
Telephone: 1-403-663-3322
Fax:1-403-663-3320

For additional information, please visit the ICEfaces website: **http://www.icefaces.org**

ICEfaces 1.5

November 2006

About this Guide

The **ICEfaces Developer's Guide** is a guide to developing ICEfaces applications. This guide is applicable to both **ICEfaces** and the **ICEfaces Enterprise Production Suite** (ICEfaces EPS). By reading through this guide, you will:

- Gain a basic understanding of what ICEfaces is and what it can do for you.
- Understand key concepts related to the ICEfaces Rich Web Presentation Environment.
- Examine the details of the ICEfaces architecture.
- Access reference information for any of the following:
  - ICEfaces system configuration
  - JSF Page Markup
  - Java API reference
  - JavaScript API reference
  - Custom Component TLD
- Learn to use advanced ICEfaces development features.

For more information about ICEfaces, visit the ICEfaces Web site at:

http://www.icefaces.org

In this guide...

This guide contains the following chapters organized to assist you with developing ICEfaces applications:

Chapter 1: Introduction to ICEfaces — Provides an overview of ICEfaces describing its key features and capabilities.

Chapter 2: ICEfaces System Architecture — Describes the basic ICEfaces architecture and how it plugs into the standard JSF framework.

Chapter 3: Key Concepts — Explores some of the key concepts and mechanisms that ICEfaces brings to the application developer.

Chapter 4: ICEfaces Reference Information — Provides additional reference information for ICEfaces implementations.

Chapter 5: Advanced Topics — Introduces several ICEfaces advanced topics, such as server-initiated rendering, drag and drop, effects, and Direct-to-DOM renderers.

Prerequisites

ICEfaces applications are JSF applications, and as such, the only prerequisite to working with ICEfaces is that you must be familiar with JSF application development. A J2EE™ 1.4 Tutorial, which includes several chapters describing JSF technology and application development, is available at:

http://java.sun.com/j2ee/1.4/docs/tutorial/doc/

ICEfaces Documentation

You can find the following additional ICEfaces documentation at the ICEfaces Web site (**http://documentation.icefaces.org**):

- **ICEfaces Getting Started Guide** — Includes information to help you configure your environment to run sample applications and a tutorial designed to help you get started as quickly as possible using ICEfaces technology.
- **ICEfaces Release Notes** — Read the ICEfaces Release Notes to learn about the new features included in this release of ICEfaces.

For additional information about enterprise-scale ICEfaces development, visit the ICEsoft Web site (**http://www.icesoft.com/products/icefaces.html**) and refer to this document:

- **ICEfaces EPS Developer's Guide** — ICEfaces EPS is an extension to ICEfaces. The ICEfaces EPS Developer's Guide describes additional features related to large-scale production deployments of ICEfaces applications. This guide is available only with ICEfaces EPS.

ICEfaces Technical Support

For more information about ICEfaces, visit the ICEfaces Technical Support page at:

http://support.icefaces.org/

Contents

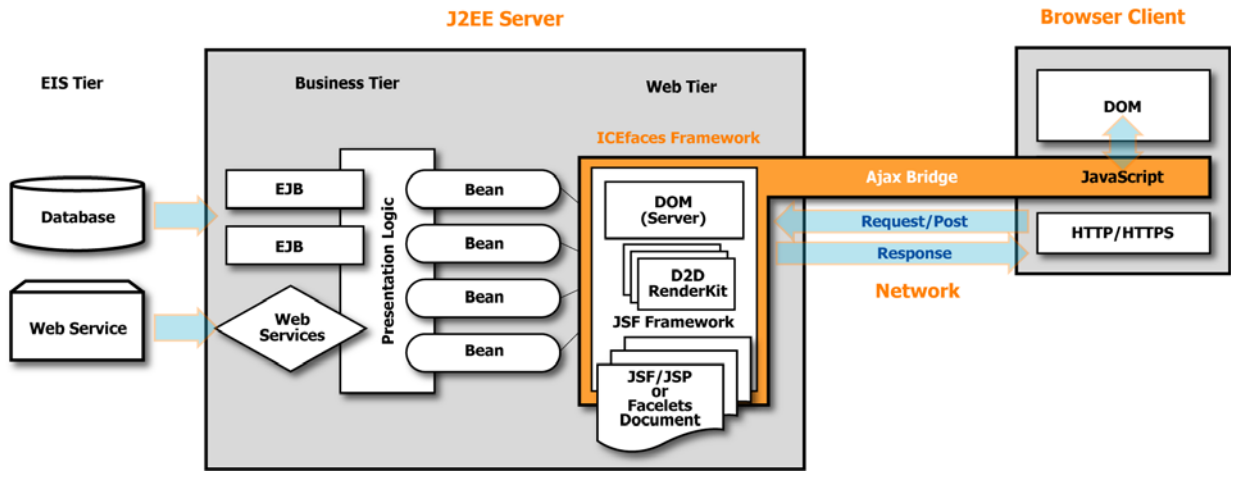
List of Figures

Chapter 1 Introduction to ICEfaces

ICEfaces™ is the industry's first standards-compliant AJAX-based solution for rapidly creating pure-Java rich web applications that are easily maintained, extended, and scaled, at very low cost.

ICEfaces provides a rich web presentation environment for JavaServer Faces (JSF) applications that enhances the standard JSF framework and lifecycle with AJAX-based interactive features. ICEfaces replaces the standard HTML-based JSF renderers with Direct-to-DOM (D2D) renderers, and introduces a lightweight AJAX bridge to deliver presentation changes to the client browser and to communicate user interaction events back to the server-resident JSF application. Additionally, ICEfaces provides an extensive AJAX-enabled component suite that facilitates rapid development of rich interactive web-based applications. The basic architecture of an ICEfaces-enabled application is shown in Figure 1 below.

Figure 1 ICEfaces-enabled JSF Application



The rich web presentation environment enabled with ICEfaces provides the following features:

- Smooth, incremental page updates that do not require a full page refresh to achieve presentation changes in the application. Only elements of the presentation that have changed are updated during the render phase.
- User context preservation during page update, including scroll position and input focus. Presentation updates do not interfere with the user's ongoing interaction with the application.

These enhanced presentation features of ICEfaces are completely transparent from the application development perspective. Any JSF application that is ICEfaces-enabled will benefit.

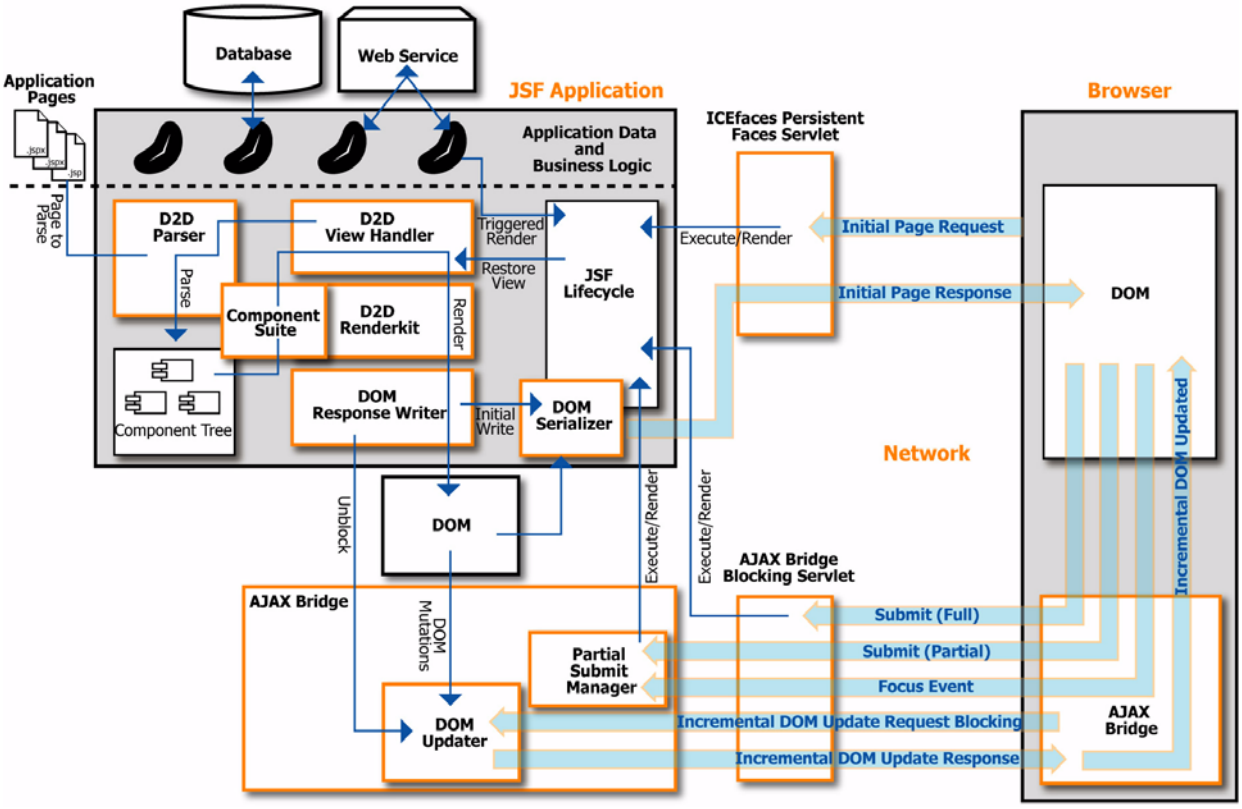
Beyond these transparent presentation features, ICEfaces introduces additional rich presentation features that the JSF developer can leverage to further enhance the user experience. Specifically, the developer can incorporate these features:

- **Intelligent form processing through a technique called Partial Submit.** Partial Submit automatically submits a form for processing based on some user-initiated event, such as tabbing between fields in a form. The automatic submission is partial, in that only partial validation of the form will occur (empty fields are marked as not required). Using this mechanism, the application can react intelligently as the user interacts with the form.
- **Server-initiated asynchronous presentation update.** Standard JSF applications can only deliver presentation changes in response to a user-initiated event, typically some type of form submit. ICEfaces introduces a trigger mechanism that allows the server-resident application logic to push presentation changes to the client browser in response to changes in the application state. This enables application developers to design systems that deliver data to the user in a near-real-time asynchronous fashion.

Chapter 2 ICEfaces System Architecture

While it is not necessary to understand the ICEfaces architecture to develop ICEfaces applications, it is generally useful for the developer to understand the basic architecture of the system. Of particular relevance is gaining an understanding of how ICEfaces plugs into the standard JSF framework. Figure 2 below illustrates the basic ICEfaces architecture.

Figure 2 ICEfaces Architecture



The major elements of the ICEfaces architecture include:

- **Persistent Faces Servlet:** URLs with the ".iface" extension are mapped to the Persistent Faces Servlet. When an initial page request into the application is made, the Persistent Faces Servlet is responsible for executing the JSF lifecycle for the associated request.
- **Blocking Servlet:** Responsible for managing all blocking and non-blocking requests after initial page rendering.

- **D2D ViewHandler:** Responsible for establishing the Direct-to-DOM rendering environment, including initialization of the DOM Response Writer. The ViewHandler also invokes the Parser for initial page parsing into a JSF component tree.
- **D2D Parser:** Responsible for assembling a component tree from a JSP Document. The Parser executes the JSP tag processing lifecycle in order to create the tree, but does this once only for each page. The standard JSP compilation and parsing process is not supported under ICEfaces.
- **D2D RenderKit:** Responsible for rendering a component tree into the DOM via the DOM Response Writer during a standard JSF render pass.
- **DOM Response Writer:** Responsible for writing into the DOM. Also initiates DOM serialization for first rendering, and unblocks the DOM Updater for incremental DOM updates.
- **DOM Serializer:** Responsible for serializing the DOM for initial page response.
- **DOM Updater:** Responsible for assembling DOM mutations into a single incremental DOM update. Updater blocks on incremental DOM update requests until render pass is complete, and DOM Response Writer performs an unblock.
- **Component Suite:** Provides a comprehensive set of rich JSF components that leverage AJAX features of the bridge and provide the basic building blocks for ICEfaces applications.
- **Client-side AJAX Bridge:** Responsible for ongoing DOM update request generation and response processing. Also responsible for focus management and submit processing.

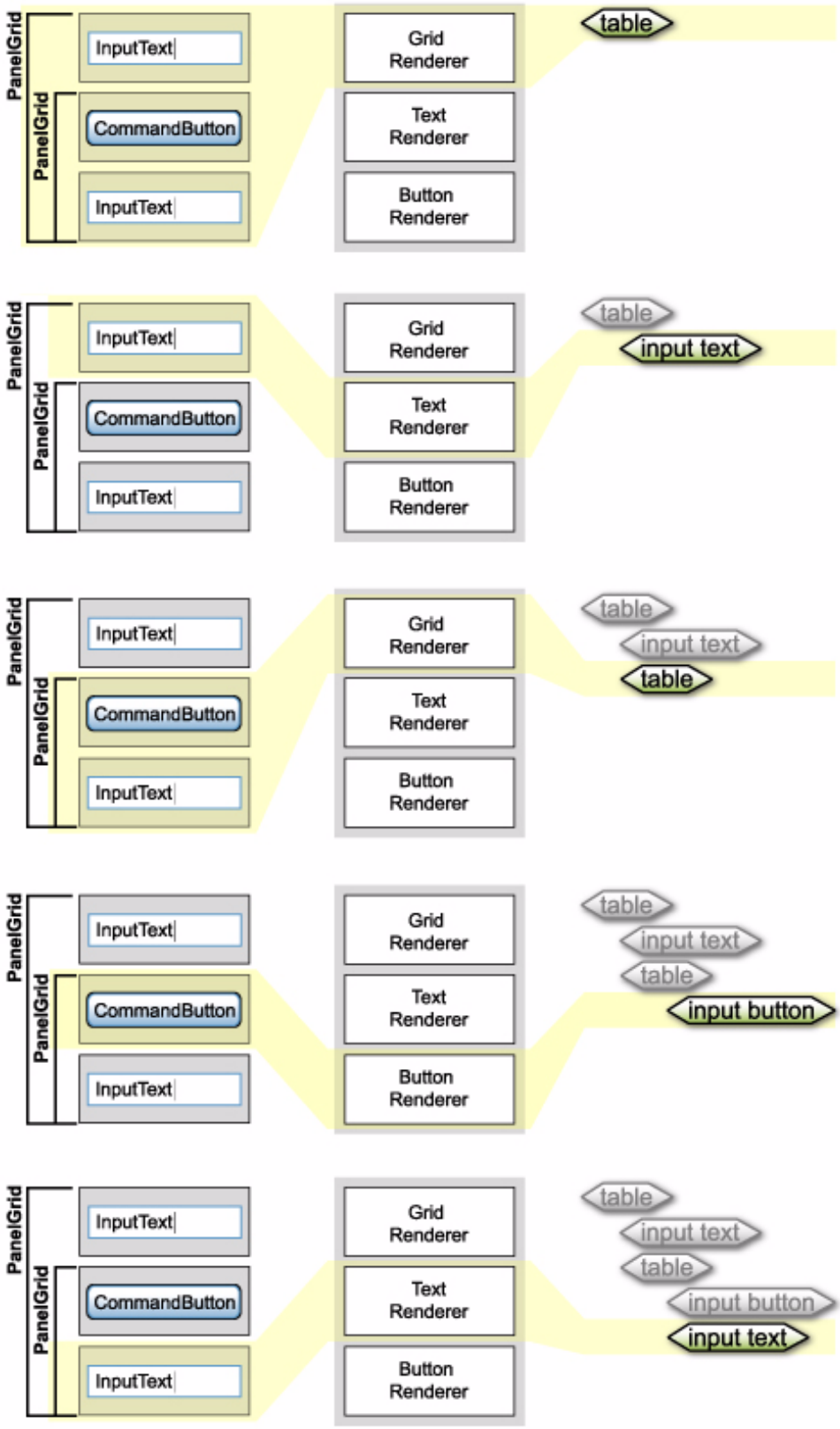
Chapter 3 Key Concepts

The JSF application framework provides the foundation for any ICEfaces application. As such, an ICEfaces application page is composed of a JSF component tree that represents the presentation for that page, and the backing beans that contain the application data model and business logic. All standard JSF mechanisms such as validation, conversion, and event processing are available to the ICEfaces application developer, and the standard JSF lifecycle applies. The following sections explore some of the key concepts and mechanisms that ICEfaces brings to the application developer.

Direct-to-DOM Rendering

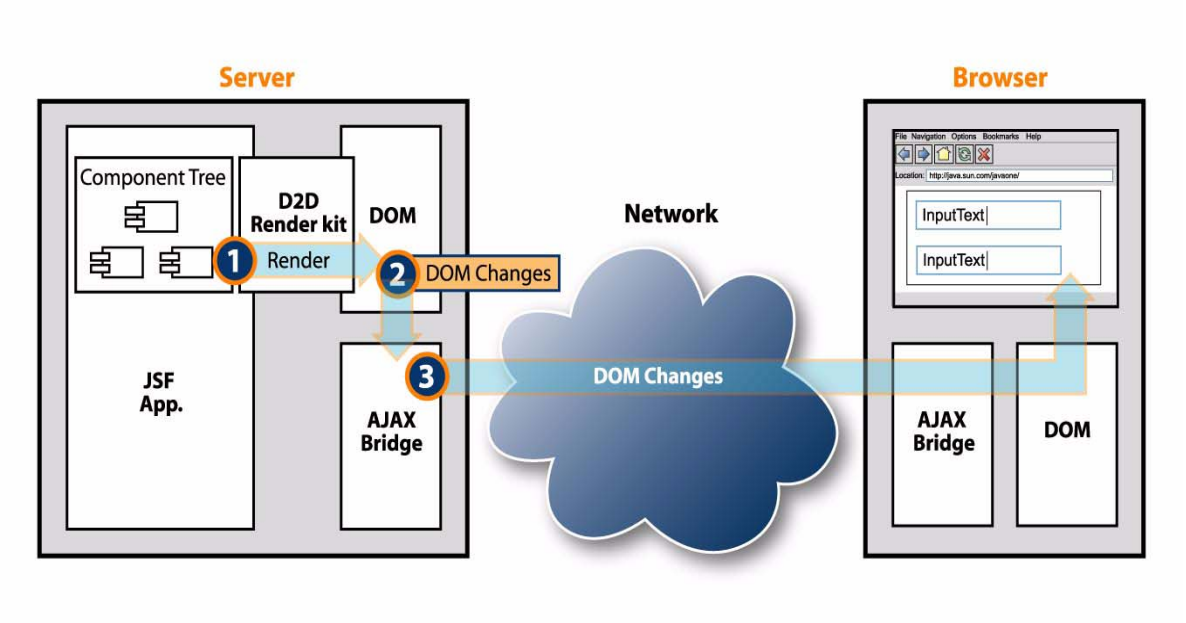
Direct-to-DOM (D2D) rendering is just what it sounds like—the ability to render a JSF component tree directly into a W3C standard DOM data structure. ICEfaces provides a Direct-to-DOM RenderKit for the standard HTML basic components available in JSF. The act of rendering a component tree into a DOM via the ICEfaces Direct-to-DOM RenderKit is illustrated in Figure 3 on page 6.

Figure 3 Direct-to-DOM Rendering



The way this basic Direct-to-DOM mechanism is deployed in an ICEfaces application involves server-side caching of the DOM and an AJAX bridge that transmits DOM changes across the network to the client browser where the changes are reassembled in the browser DOM. This process is illustrated in Figure 4.

Figure 4 Direct-to-DOM Rendering Via AJAX Bridge

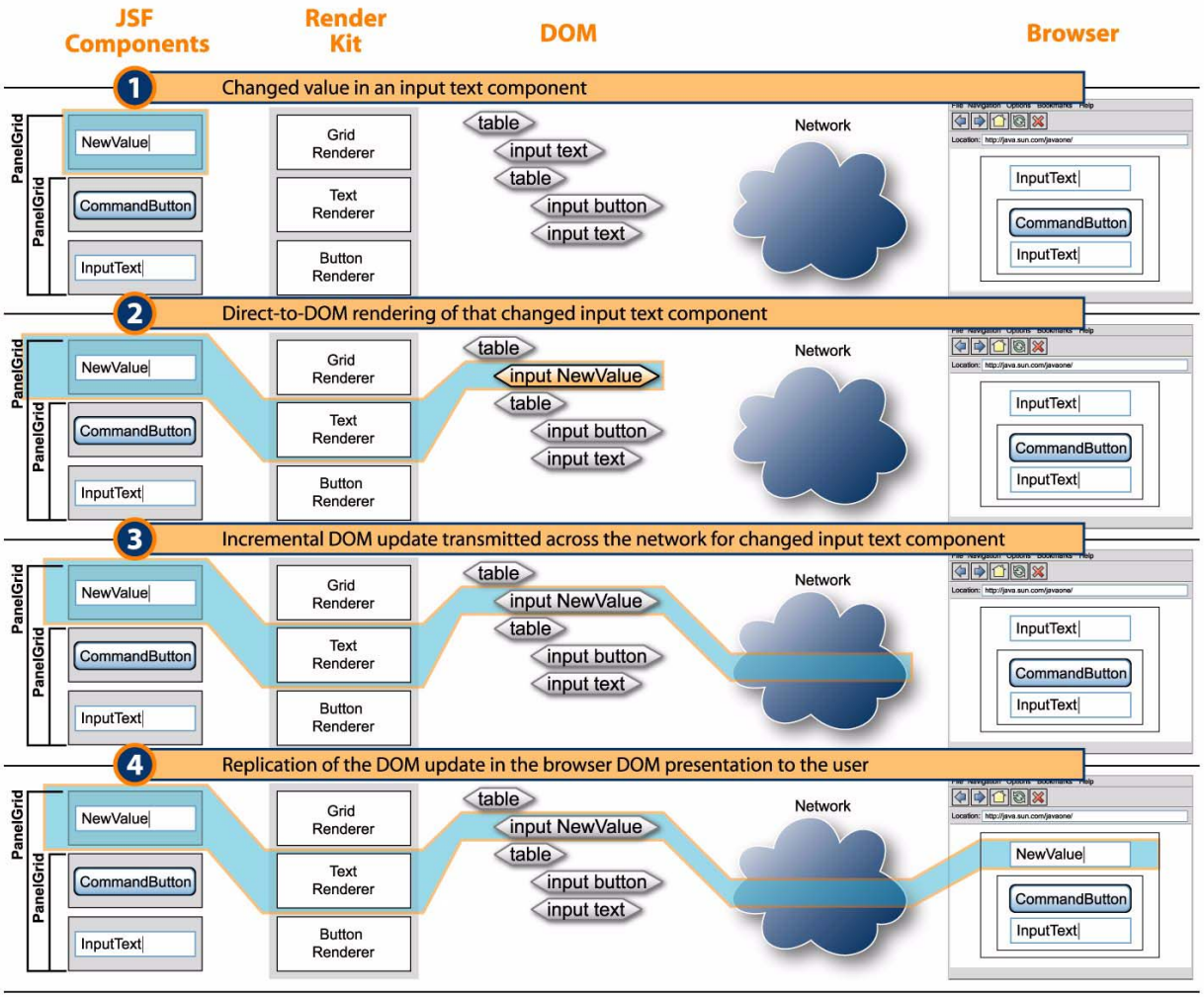


When the ICEfaces JAR is included in your JSF application, the Direct-to-DOM RenderKit is automatically configured into the application. There are no other considerations from the developer perspective. Direct-to-DOM rendering is completely transparent in the development process.

Incremental, In-place Page Updates

One of the key features of Direct-to-DOM rendering is the ability to perform incremental changes to the DOM that translate into in-place editing of the page and result in smooth, flicker-free page updates without the need for a full page refresh. This basic concept is illustrated in Figure 5.

Figure 5 Incremental Update with Direct-to-DOM Rendering



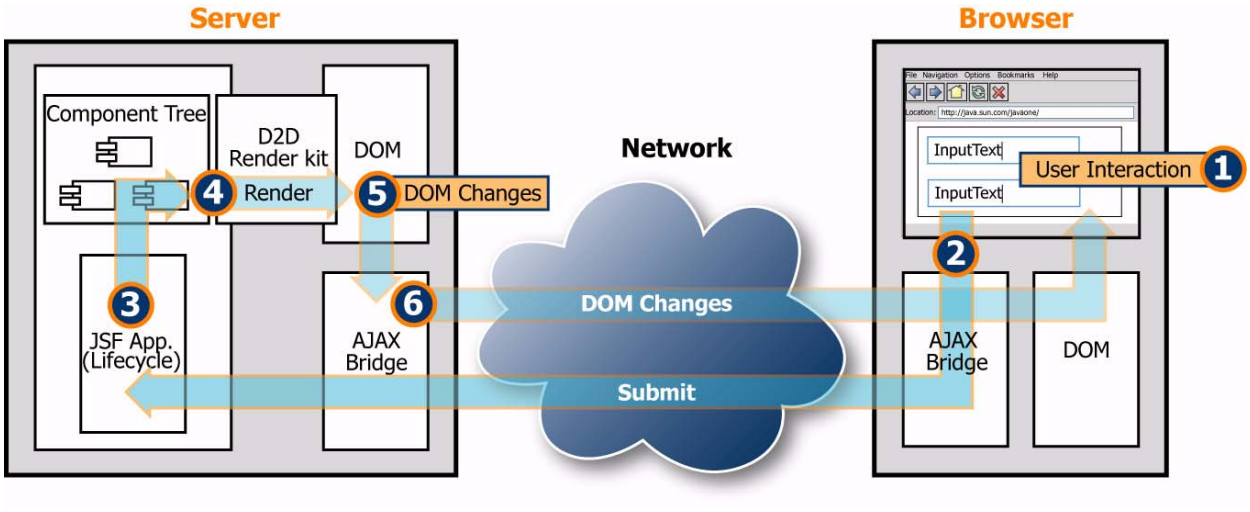
Again, incremental updates are transparent from the development perspective. As the presentation layer changes during a render pass, those changes are seamlessly realized in the client browser.

Armed with incremental Direct-to-DOM rendering, you can begin to imagine a more dynamic presentation environment for the user. You no longer have to design pages around the full page refresh model. Instead, you can consider fine-grained manipulation of the page to achieve rich effects in the application. For example, selective content presentation, based on application state, becomes easy to implement. Components can simply include value bindings on their isRendered attribute to programmatically control what elements of the presentation are rendered for any given application state. ICEfaces incremental Direct-to-DOM update will ensure smooth transition within the presentation of that data.

Synchronous and Asynchronous Updates

Normally, JSF applications update the presentation as part of the standard request/response cycle. From the perspective of the server-resident application, we refer to this as a *synchronous* update. The update is initiated from the client and is handled synchronously at the server while the presentation is updated in the response. A synchronous update for ICEfaces is illustrated in Figure 6 below.

Figure 6 Synchronous Updates

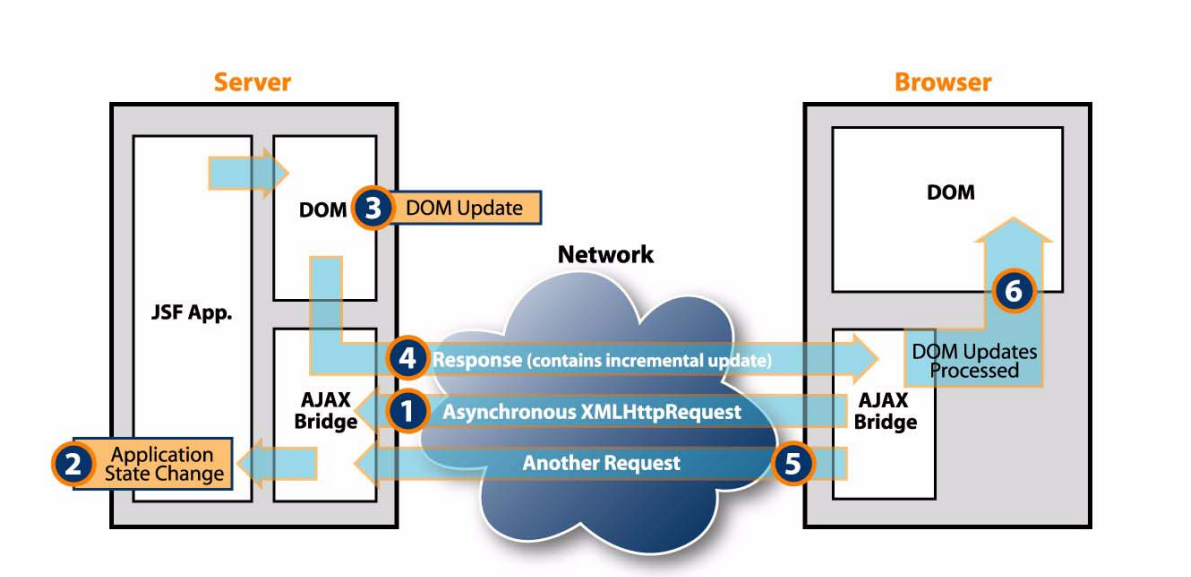


One serious deficiency with synchronous updates is that the application requires a client-generated request before it can affect presentation layer changes. If an application state change occurs during a period of client inactivity, there is no means to present changing information to the user. ICEfaces overcomes this deficiency with an *asynchronous* update mode that facilitates driving asynchronous presentation changes to the client, based on server-side application state changes. The ICEfaces application developer is not restricted to the standard request/response cycle of a normal JSF application. Again, the AJAX bridge facilitates ongoing asynchronous updates through the use of asynchronous XMLHttpRequests that are fulfilled when DOM updates become available due to a render pass. Because the process leverages incremental Direct-to-DOM updates for asynchronous presentation changes, you can expect these changes to occur in a smooth, flicker-free manner. Figure 7 illustrates this process.

The primary consideration from the developer's perspective is to identify and implement the triggers that cause the presentation updates to happen. Trigger mechanisms are entirely under developer control and can include standard JSF mechanisms like ValueChangeEvents, or any other outside stimulus.

Because it is important to manage the asynchronous rendering process in a scalable and performant manner, ICEfaces provides a Server-initiated Rendering API and implementation that does. See Server-initiated Rendering on page 11 for additional discussion.

Figure 7 Asynchronous Update with Direct-to-DOM Rendering



Asynchronous mode is the default for ICEfaces, but in cases where asynchronous updates are not required, ICEfaces can be configured to support synchronous mode only. Running in synchronous mode reduces the connection resource requirements for an application deployment. See Synchronous Updates on page 21 to specify the mode of operation.

When ICEfaces is running in asynchronous mode, it is possible for an outstanding request to remain open for an extended period of time. Depending on the deployment environment, it is possible for a long-lived connection to be lost, resulting in the loss of asynchronous updates. ICEfaces provides connection management facilities that allow the application to react to connection-related errors. See Connection Management on page 10 for additional information.

Connection Management

Client/server connectivity is a key requirement for ICEfaces applications to function. For this reason, ICEfaces provides connection status monitoring facilities in the client-side AJAX bridge, and a Connection Status component to convey connection status information to the user interface. Connection monitoring in ICEfaces can detect the following states of a client connection:

- Idle
- Waiting
- Lost

If a Connection Status component is present in the page, that component will be updated as connection status changes. See ICEfaces Custom Components on page 27 for further details on using the **Connection Status** component.

The connection lost status can be caused by the following conditions.

- Session expired
- Empty response (not detected in Internet Explorer)

- Response timeout

If connection lost status is determined, and no connection status component is present, ICEfaces displays a modal overlay indicating connection lost. A reload of the application is required at this point.

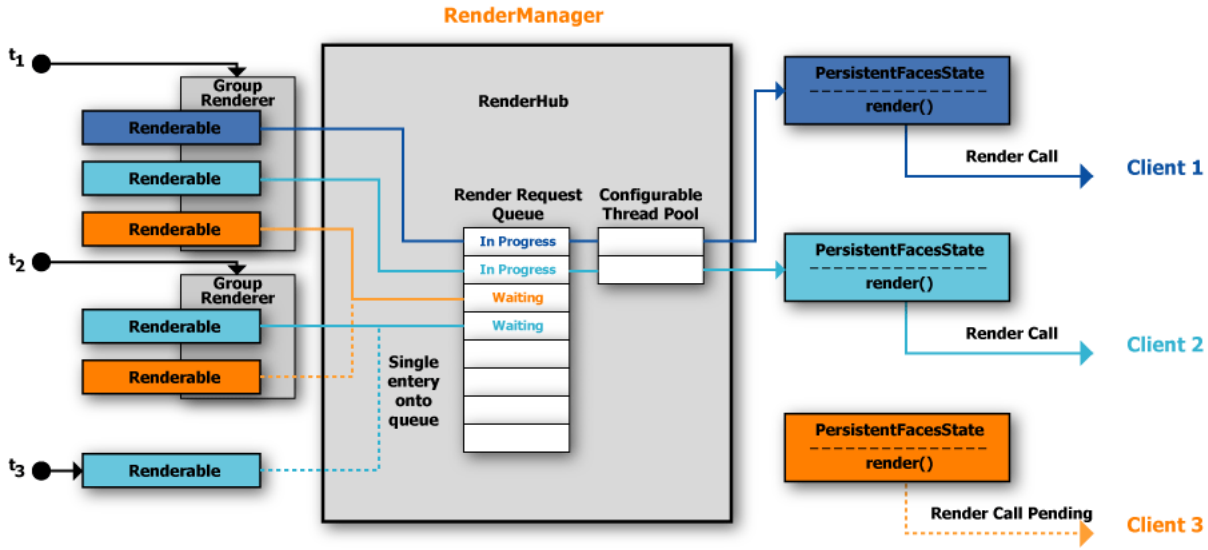
> **Note:** **ICEfaces EPS** provides more advanced connection management facilities including connection heart beating. See the **ICEfaces EPS Developer's Guide** for further details.

Server-initiated Rendering

Asynchronous update mode in ICEfaces supports server-initiated presentation updates driven from application logic. In ICEfaces, this is achieved by causing the JSF lifecycle render phase to execute in reaction to some state change within the application. The session-scoped PersistentFacesState provides this API, and facilitates low-level server-initiated rendering on a per-client basis. While this low-level rendering mechanism looks simple to use, there are a number of potential pitfalls associated with it related to concurrency/deadlock, performance, and scalability. In order to overcome these potential pitfalls, ICEfaces provides a high-performance, scalable Server-initiated Rendering API, and strongly discourages the use of the low-level render call.

The server-initiated rendering architecture is illustrated in Figure 8 below.

Figure 8 Server-initiated Rendering Architecture



The key elements of the architecture are:

Renderable: A session-scoped bean that implements the Renderable interface and associates the bean with a specific PersistentFacesState. Typically, there will be a single Renderable per client.

RenderManager: An application-scoped bean that maintains a RenderHub, and a set of named group Renderers.

RenderHub: Implements coalesced rendering via a configurable thread pool, ensuring that the number of render calls is minimized, and thread consumption is bounded.

Group Renderer: Supports rendering of a group of Renderables. Group Renderers can support on-demand, interval, and delayed rendering of a group.

For detailed information, see Server-initiated Rendering API on page 31.

Partial Submit – Intelligent Form Processing

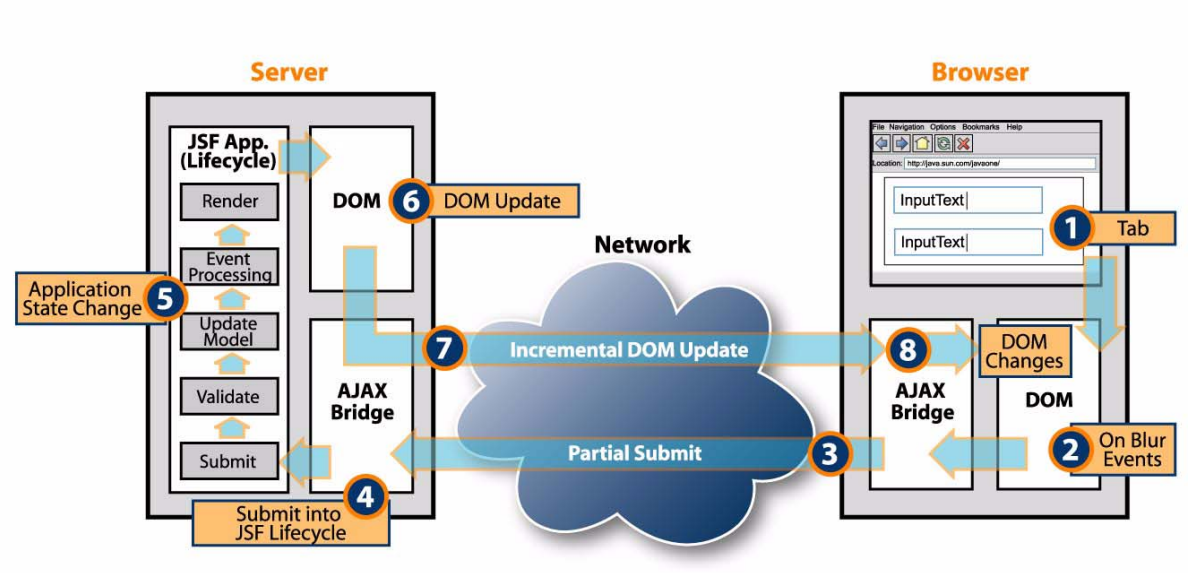
ICEfaces introduces a fine-grained user interaction model for intelligent form processing within an ICEfaces application. In JSF, the normal submit mechanism initiates the JSF application lifecycle, and as such, capabilities like client-side validation are not supported. Partial submit overcomes these limitations by tying the JavaScript event mechanism back into the JSF application lifecycle via an automatic submit. This automatic submit is partial in the sense that only partial validation of the form will occur.

The AJAX bridge does intelligent focus monitoring to identify the control associated with the partial submit, and turns off the *required* property for all other controls in the form. From here a normal JSF lifecycle is performed, after which the *required* properties are restored to their previous state. The net effect of a partial submit is that the full validation process executes, but empty fields in the form are not flagged as invalid. Figure 9 below illustrates partial submit based on an onBlur JavaScript event that occurs as the user tabs between controls in a form.

The client-side AJAX bridge provides a convenience function for tying JavaScript events to the partial submit mechanism. The API details can be found in Configuration Reference on page 20, but the mechanism relies on only a small snippet of JavaScript being defined in the specific JavaScript attribute for the JSF component instance that is intended to cause the partial submit.

The granularity at which partial submits occur is entirely under developer control. In certain cases, it may be appropriate to evaluate and react to user input on a per-keystroke-basis, and in other cases, it may be appropriate as focus moves between controls. In still other cases, only specific controls in the form would initiate a partial submit.

Figure 9 Partial Submit Based on OnBlur



The backing application logic associated with a partial submit is also entirely under developer control. The standard JSF validator mechanism can be leveraged, or any other arbitrarily complex or simple evaluation logic can be applied. If standard JSF validators are used, it is important to design these validators to facilitate partial submits.

The Address Form demo from the ICEfaces samples illustrates a couple of different mechanisms that can be leveraged under a partial submit. Standard validators are attached to City, State, and ZIP input fields to catch invalid entries, but inter-field evaluation on the {City:State:ZIP}-tuple is also performed. Using valueChangedEvents associated with these input controls, it is possible to do inter-field analysis and morph the form based on current input. For example, entering a valid City will cause the State input control to change from an input text control to a select-one-of-many controls containing only the States that have a matching City. The possibilities are endless when you apply your imagination.

Components and Styling

JSF is a component-based architecture, and as such, JSF application User Interfaces are constructed from a set of nested components. The JSF specification includes a number of standard components, but also provides for adding custom components to the JSF runtime environment. This extensible component architecture is leveraged in ICEfaces to support the standard components as well as several collections of custom components.

From the developer's perspective, a component is represented with a tag in a JSF page, and the tag library descriptor (TLD) for that tag defines a component class, and a renderer class for the component. At runtime, TLDs configured into the web application are parsed, and assembled into a RenderKit, the default for JSF being the html_basic RenderKit. ICEfaces utilizes the html_basic RenderKit but replaces

standard HTML renderers with Direct-to-DOM renderers. The following table identifies the component name spaces that ICEfaces supports with Direct-to-DOM renderers.

Name Space	Description	ICEfaces Features
www.icesoft.com/icefaces/component	• ICEfaces Component Suite	• Direct-to-DOM renderers • Automated partial submit • Automated CSS styling
java.sun.com/jsf/html	• Standard JSF Components	• Direct-to-DOM renderers

Refer to ICEfaces Component Suite on page 24 for detailed reference information for all ICEfaces supported components.

Cascading Style Sheets (CSS) Styling

The purpose of Cascading Style Sheets (CSS) is to separate style from markup. ICEfaces encourages and supports this approach in the ICEfaces Component Suite by supporting automated component styling based on a common CSS class definitions. This means that when the ICEfaces Component Suite is used to develop applications, those applications can be quickly and consistently re-skinned with a different look by replacing the CSS with a new CSS. More information about styling the ICEfaces Component Suite can be found in Styling the ICEfaces Component Suite on page 27.

Other Custom Components

ICEfaces adheres to the extensible component architecture of JSF, and as such, supports inclusion of other custom components. Most existing custom components that use HTML-based renderers should integrate seamlessly into an ICEfaces application. However, if the component renderer incorporates any significant JavaScript in its implementation, the likelihood of a conflict between the component JavaScript, and the ICEfaces Bridge JavaScript is high, and can result in unpredictable behavior.

The most effective way to incorporate a new component into an ICEfaces application is to develop a Direct-to-DOM renderer for that component. The renderer can leverage features of the ICEfaces architecture to incorporate rich interactive behavior into a component with no need to include complex JavaScript in the rendering code. See ICEfaces Tutorial: Creating Direct-to-DOM Renderers for Custom Components on page 43 for details on how to write Direct-to-DOM renderers and configure them into the ICEfaces environment.

Drag and Drop

ICEfaces includes support for dragging and dropping components using the script.aculo.us library. Any ice:panelGroup instance can be set to be draggable or a drop target.

For example, to make a panelGroup draggable, set the draggable attribute to true.

```
<ice:panelGroup draggable="true">
```

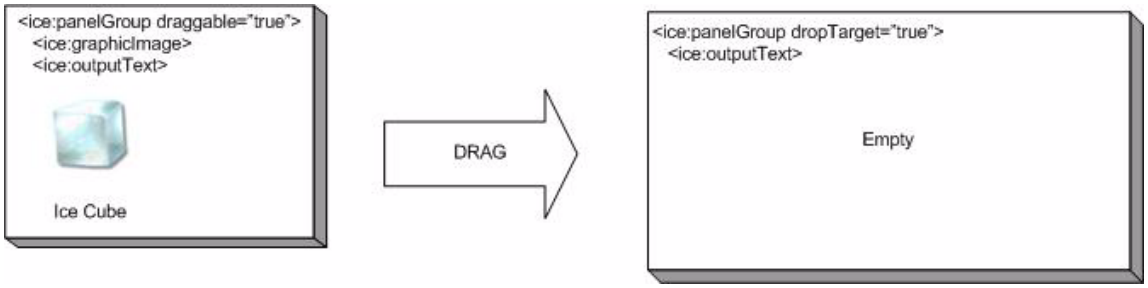
The panel group (and all of its child components) can now be dragged around the screen.

Any panelGroup can be set to a drop target as well by setting the dropTarget attribute to true.

```
<ice:panelGroup dropTarget="true">
```

When a draggable panel is moved over or dropped on the drop target panel, events can be fired to the backing beans. These events include Drag Start, Drag Cancel, Hover Start, Hover End, and Dropped.

Figure 10 Drag and Drop Concept



Draggable panels can optionally have animation effects that modify their behavior and appearance panels. The following table lists these optional effects and their behavior.

Effect	Behavior
revert	When a panel is dropped, it moves back to its starting position.
ghosting	A ghost copy of the drag panel remains at its original location during dragging.
solid	No transparency is set during dragging.

See Creating Drag and Drop Features on page 38 for details on how to build drag and drop applications.

Effects

ICEfaces uses the script.aculo.us library to provide animation effects. Effects can be easily invoked on components using the effect attribute. The value of the effect attribute is a value binding expression to a backing bean which returns the effect to invoke.

```
<ice:outputText effect="#{bean.messageEffect}"/>
```

Effects can be customized by modifying the properties of the effect object being used. For example, the effect duration could be changed. For more information, refer to Adding and Customizing Effects on page 41.

Browser-Invoked Effects

Effects can also be tied to browser events, such as onmouseover.

```
<ice:outputText onmouseovereffect="#{bean.mouseOverEffect}"/>
```

These effects will be invoked each time the mouse moves over the outputText component.

See Adding and Customizing Effects on page 41 for details on how to add effects in your ICEfaces application.

Concurrent DOM Views

By default, each ICEfaces user can have only one dynamically updated page per web application. In this configuration, a single DOM is maintained for each user session. Reloading an ICEfaces page synchronizes the browser to the server-side DOM. Opening a new browser window into the same application, however, leads to page corruption as DOM updates may be applied unpredictably to either window.

To allow multiple windows for a single application, concurrent DOM views must be enabled. See Configuring web.xml on page 20 to configure the web.xml file for concurrent DOM views.

With concurrent DOM views enabled, each browser window is distinctly identified with a *view number* and DOM updates will be correctly applied to the appropriate window. This introduces some important considerations for the application data model. Managed beans in session scope can now be shared across multiple views simultaneously. This may be the desired scope for some states, but typically, presentation-related state is more appropriately kept in request scope. For example:

```
<managed-bean>
  <managed-bean-name>BoxesCheckedBean</managed-bean-name>
  <managed-bean-class>com.mycompany.BoxesCheckedBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

Note: The ICEfaces request scope is typically longer lived than the request scope for non-dynamic applications. An ICEfaces request begins with the initial page request and remains active through user interactions with that page (such user interactions would normally each require a new request). One consideration is that new browser windows and page reloads of the same browser window are both regarded as new requests. Therefore, it is important to implement the dynamic aspects of the ICEfaces application so that asynchronous notifications are applied to all active requests and not just the initial one.

For applications that do not make use of Concurrent DOM views and require request scope to last only for the duration of a single user event, "standard request scope" must be enabled. See Configuring web.xml on page 20 to configure the web.xml file for "standard request scope".

The following table shows a summary of the Managed Bean Scope.

State	Managed Bean Scope
none	For transient data.
request	For typical view-related state, request-scope beans will persist through most user interaction but not through view changes. This is the recommended scope for ICEfaces applications that make use of multiple windows.
session	For state that must be shared across views.
application	For state that must be shared across users.

Note: Most browsers place limitations on the number of concurrent open HTTP requests. Therefore, applications with concurrent views must have regular asynchronous update, such as from a ticking clock in the page, so that the network connection is continually refreshed. This may be mitigated in the future either by improvements in browser HTTP handling or by a synchronous mode for ICEfaces.

Integrating ICEfaces With Existing Applications

JSP Inclusion

ICEfaces can be easily integrated with existing JSP-based applications through the dynamic JSP inclusion mechanism. Once you have developed a standalone page with the desired ICEfaces dynamic content, simply include that page into your existing JSP page with the following statement:

```
<jsp:include page="/content.iface" />
```

Note: Navigation (such as hyperlinks) from the embedded ICEfaces content will cause the entire page to be reloaded, but other interaction with the application will be incrementally updated in-place on the page.

Included ICEfaces pages **must** contain a <body> tag as the current implementation uses this tag as a marker. The contents of the <body> tag is precisely the content that is dynamically inserted into the including page. This restriction will be removed in a future release.

JSF Integration

In most cases, the goal of adding ICEfaces to a JSF application will be to transform the entire application into an AJAX application, but there may be reasons for converting only parts over to ICEfaces. To handle some pages with ICEfaces and other pages with the JSF default mechanism, add the ICEfaces Servlet mappings for the ".iface" extension (as described in Configuration Reference on page 20 of this document) but do not remove the "Faces Servlet" mapping or Servlet initialization. Pages served through the default Faces Servlet are to be handled without ICEfaces. Then, to ensure that Direct-to-DOM renderers are applied to ICEfaces pages only, include "just-ice.jar" rather than "icefaces.jar" in your web application libraries. This .jar file contains a version of ICEfaces configured with a ViewHandler that will process only those pages with a ".iface" extension and a RenderKit that will not override the standard JSF components (such as <h:commandLink />) with Direct-to-DOM renderers. In this configuration, ICEfaces pages must contain only standard core JSF tags ("f:" tags) and ICEfaces tags ("ice:" tags).

Facelets

Facelets is an emerging standard and open source implementation, which provides a templating environment for JSF and eliminates the need for JSP processing. ICEfaces has been integrated with Facelets so the ICEfaces developers can take advantage of its capabilities. You can learn more about Facelet development at:

https://facelets.dev.java.net/

Refer to Steps 6 and 7 in **Chapter 4, ICEfaces Tutorial: The TimeZone Application**, of the **ICEfaces Getting Started Guide** for details on how to port an existing ICEfaces application to run with Facelets, and how to leverage Facelet templating to implement ICEfaces applications.

Chapter 4 ICEfaces Reference Information

This chapter includes additional information for the following:

Markup Reference

ICEfaces supports the JavaServer Faces JSP Document syntax, but does not process these documents via the standard JSP compilation/execution cycle. Instead, ICEfaces parses input documents directly, and assembles a JSF component tree by executing the given tags. This approach allows precise adherence to the ordering of JSF and XHTML content in the input document, thereby making the JSP Document syntax more suitable to web designers. Since Direct-to-DOM rendering makes use of a server-side DOM containing the output of the current page, it is necessary that the input document be readily represented as a DOM. This requires that the input be well-formed XML, which is the expected case for JSP Documents, but may not be adhered to in certain JSP pages. To handle JSP pages, ICEfaces converts them into JSP documents on the fly, performing a small set of transformations aimed at well-formedness (such as converting "
" to "
"), before passing the document to the parser.

While parsing documents directly does resolve ordering problems currently present with JSP and JSF, some restrictions are introduced:

- Well-formed XHTML markup is recommended; the conversion process can only repair simple well-formedness errors. In particular, the JSP namespace must be declared (as shown below) in order for a JSP document to parse properly.

  ```
  xmlns:jsp="http://java/sun.com/JSP/Page"
  ```

- JSP directives, JSP expressions and inline Java code are all ignored.
- Custom JSP tags are not supported in general as ICEfaces only executes JSP tags initially to assemble the JSF component tree (JSF tags with complex processing may also not function as expected).

- ICEfaces supports the following JSP inclusion mechanisms:
  - **<%@ include %> and <jsp:directive.include />**: The contents of the given file will be included in the input prior to parsing. This is currently the recommended inclusion mechanism with this ICEfaces release.
  - **<jsp:include />**: The ICEfaces parser initiates a local HTTP request to perform dynamic inclusion. The current session is preserved, but otherwise the inclusion shares no state with the including page. If possible, use static inclusion with the current ICEfaces Release.
- Deprecated XHTML tags may create unexpected results in certain browsers, and should be avoided. Page authors should use style sheets to achieve stylistic and formatting effects rather than using deprecated XHTML presentation elements. In particular, the <font> tag is known to cause issues in certain browsers.
- To produce a non-breaking space in output use "** **" instead of " ".

The above JSP restrictions in ICEfaces are expected to be resolved in conjunction with the release of JSP 2.1/JSF 1.2 as this release is expected to address JSF/JSP ordering problems.

Java API Reference

Refer to the **ICEfaces SDK API** included with this release. The API can also be found at:

http://documentation.icefaces.org/

Refer to the ICEfaces components API included with this release. The API can also be found at:

http://documentation.icefaces.org/

The javadoc can also be found in the docs directory of this release.

Configuration Reference

Configuring faces-config.xml

The icefaces.jar file contains a faces-config.xml file that configures the ICEfaces extensions. Specifically, the configuration file registers the Direct-to-DOM renderers. There is no need for the developer to modify the ICEfaces faces-config.xml.

Configuring web.xml

Servlet Registration and Mappings

The application's web.xml file must include necessary Servlet registration and mappings.

The ICEfaces Servlets are registered as follows:

```
<servlet>
  <servlet-name>Persistent Faces Servlet</servlet-name>
```

```
  <servlet-class>
    com.icesoft.faces.webapp.xmlhttp.PersistentFacesServlet
  </servlet-class>
  <load-on-startup> 1 </load-on-startup>
</servlet>

<servlet>
  <servlet-name>Blocking Servlet</servlet-name>
  <servlet-class>com.icesoft.faces.webapp.xmlhttp.BlockingServlet</servlet-class>
  <load-on-startup> 1 </load-on-startup>
</servlet>
```

The Servlet mappings are established as follows:

```
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.jspx</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>Persistent Faces Servlet</servlet-name>
  <url-pattern>*.iface</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>Persistent Faces Servlet</servlet-name>
  <url-pattern>/xmlhttp/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>Blocking Servlet</servlet-name>
  <url-pattern>/block/*</url-pattern>
</servlet-mapping>
```

The context listener is established as follows:

```
<listener>
  <listener-class>
    com.icesoft.faces.util.event.servlet.ContextEventRepeater
  </listener-class>
</listener>
```

Synchronous Updates

Synchronous or asynchronous updates can be turned off/on application-wide using the ICEfaces context parameter, **com.icesoft.faces.synchronousUpdate**. Typically this is set in the web.xml file of your web application.

```
  <context-param>
    <param-name>com.icesoft.faces.synchronousUpdate</param-name>
    <param-value>true/false</param-value>
  </context-param>
```

If not specified, the parameter value is *false*; thus, by default, the application will run in asynchronous mode.

Concurrent Views

To allow multiple windows for a single application, concurrent DOM views must be enabled. This is set through the ICEfaces context parameter, **com.icesoft.faces.concurrentDOMViews**. Typically, this is set in the web.xml file of your web application:

```
<context-param>
  <param-name>com.icesoft.faces.concurrentDOMViews</param-name>
  <param-value>true</param-value>
</context-param>
```

Standard Request Scope

To cause request scope to last only for the duration of a single user event, "standard request scope" must be enabled. This is set through the ICEfaces context parameter:

```
com.icesoft.faces.standardRequestScope
```

Typically this is set in the web.xml file of your web application:

```
<context-param>
  <param-name>com.icesoft.faces.standardRequestScope</param-name>
  <param-value>true</param-value>
</context-param>
```

Redirect on JavaScript Blocked

Some browsers are configured to block JavaScript interpretation or some browsers cannot interpret JavaScript content. For these instances, ICEFaces can be configured to redirect the browser to a custom error page.

This feature can be turned on application-wide using the ICEfaces context parameter, **com.icesoft.faces.javascriptBlockedRedirectURI**.

```
<context-param>
   <param-name>com.icesoft.faces.javascriptBlockedRedirectURI</param-name>
   <param-value>...custom error page URL....</param-value>
</context-param>
```

If not specified, by default the server will send an HTTP error code '403 - Javascript not enabled'. This is to avoid any ambiguity, since the accessed page would be rendered but any interaction with it would be impossible.

Compressing Resources

Resources such as JavaScript and CSS files can be compressed when sent to the browser. This can improve application load time in certain deployments. This configuration works independently from the web-server configuration.

The feature can be turned on application-wide using the ICEfaces context parameter, **com.icesoft.faces.compressResources**.

```
<context-param>
  <param-name>com.icesoft.faces.compressResources</param-name>
  <param-value>true/false</param-value>
</context-param>
```

If not specified, by default the application will not compress the resources.

File Upload

The maximum file upload size can be specified in the web.xml file of your web application as follows:

```
<context-param>
    <param-name>com.icesoft.faces.uploadMaxFileSize</param-name>
    <param-value>1048576</param-value>
</context-param>
```

If not specified the default value for file upload is 10485760 bytes (10 megabytes).

To specify the directory location where uploaded files are stored, the following parameter is used:

```
<context-param>
    <param-name>com.icesoft.faces.uploadDirectory</param-name>
    <param-value>images/upload</param-value>
</context-param>
```

This parameter works in conjunction with the ice:inputFile component attribute "uniqueFolder" with four possible combinations as illustrated in the table below:

uniqueFolder	com.icesoft.faces.uploadDirectory	
	Set	**Not Set**
True	/application-context/uploadDirectory/sessionid/	/application-context/sessionid/
False	/application-context/uploadDirectory/	/application-context/

Components Reference

ICEfaces provides render kits for the following JSF component suites.

Name Space	Description	ICEfaces Features
www.icesoft.com/icefaces/component	• ICEfaces Component Suite	• Direct-to-DOM renderers • Automated partial submit • Automated CSS styling
java.sun.com/jsf/html	• Standard JSF Components	• Direct-to-DOM renderers

The ICEfaces Component Suite classes and renderers are contained in the icefaces-comps.jar. The Sun Standard JSF Component renderers are contained in the icefaces.jar.

ICEfaces Component Suite

The ICEfaces Component Suite provides a complete set of the most commonly required components. These components feature additional benefits over other JSF component sets, such as:

- Optimized to fully leverage ICEfaces Direct-to-DOM rendering technology for seamless incremental UI updates for all components without full-page refreshes.

- Support for additional attributes for ICEfaces-specific features, such as partialSubmit, effects, visibleOnUserRole, etc.
- Support for comprehensive component styling via predefined component style sheets that are easily customized.

For more details, see ICEfaces Component Suite on page 24.

Standard JSF Components

The standard JSF components as defined in the JSF specification are supported with Direct-to-DOM renderers. No additional ICEfaces-specific attributes are associated with the standard JSF component tags. ICEfaces-specific capabilities such as partial submit can be configured into the tag through the standard JavaScript-specific pass through attributes (e.g., onblur="iceSubmitPartial(form.this.event)"). The standard JSF component renderers do not support ICEfaces automated CSS styling.

ICEfaces Component Suite

The ICEfaces Component Suite includes enhanced implementations of the JSF standard components and additional custom components that fully leverage the ICEfaces Direct-to-DOM rendering technology and provide additional ICEfaces-specific features, such as automated partial submit, incremental page updates, and easily configurable component look-and-feel.

Common Attributes

The following are descriptions of the common attributes that apply to the ICEfaces components.

renderedOnUserRole The visibleOnUserRole attribute has been re-named to renderedOnUserRole. If user is in given role, this component will be rendered normally. If not, nothing is rendered and the body of this tag will be skipped.

enabledOnUserRole If user is in given role, this component will be rendered normally. If not, then the component will be rendered in the disabled state.

visible The visible attribute has been added to all the relevant standard extended components. Used to render the visibility style attribute on the root element. visible values: true || false. Note: If the visible attribute is not defined, the default is visible.

disabled The disabled attribute is a Flag indicating that this element must never receive focus or be included in a subsequent submit. Unlike the readOnly which is included in a submit but cannot receive focus.

partialSubmit The partialSubmit attribute enables a component to perform a partial submit in the appropriate event for the component. The partialSubmit attribute only applies to custom and extended components.

The following table shows which attributes are applicable to each ICEfaces component.

ICEfaces Components	renderedOnUserRole	enabledOnUserRole	visible	disabled	partialSubmit
commandButton	*	*	*	*	*
commandLink	*	*	*	*	*
commandSortHeader	*	*		*	
dataPaginator	*	*		*	
dataTable	*				
form	*				*
graphicImage	*		*		
inputFile	*	*		*	
inputSecret	*	*	*	*	*
inputText	*	*	*	*	*
inputTextarea	*	*	*	*	*
menuBar	*				
menuItem	*				
menuItems	*				
menuItemSeparator	*				
message	*		*		
messages	*		*		
outputChart	*				
outputConnectionStatus	*				
outputDeclaration	*				
outputLabel	*		*		
outputLink	*	*	*	*	
outputProgress	*				
outputText	*		*		
panelBorder	*				
panelGrid	*		*		
panelGroup	*		*		
panelPositioned					
panelPopup	*		*		
panelSeries	*				
panelStack	*				
panelTabSet	*				
panelTab	*	*		*	
selectBooleanCheckbox	*	*	*	*	*
selectInputDate	*	*		*	
selectInputText	*	*		*	
selectManyCheckbox	*	*	*	*	*

ICEfaces Components	renderedOnUserRole	enabledOnUserRole	visible	disabled	partialSubmit
selectManyListbox	*	*	*	*	*
selectManyMenu	*	*	*	*	*
selectOneListbox	*	*	*	*	*
selectOneMenu	*	*	*	*	*
selectOneRadio	*	*	*	*	*

Enhanced Standard Components

The standard JSF components have been enhanced to support ICEfaces partial page rendering, partialSubmit of editable components, and the ability to enable or disable and show or hide components based on user role.

The enhanced standard components included in the ICEfaces Component Suite are:

- commandButton
- commandLink
- dataTable
  - column
- form
- graphicImage
- inputHidden
- inputSecret
- inputText
- inputTextarea
- message
- messages
- outputFormat
- outputLabel
- outputLink
- outputText
- panelGrid
- panelGroup
- selectBooleanCheckbox
- selectManyCheckbox
- selectManyListbox
- selectManyMenu
- selectOneListbox
- selectOneMenu
- selectOneRadio

ICEfaces Custom Components

The ICEfaces Component Suite also includes a set of custom components that fully leverage the ICEfaces Direct-to-DOM rendering technology.

The following custom components are provided:

- commandSortHeader
- dataPaginator
- dataTable
- inputFile
- menuBar
  - menuItem
  - menuItems
  - menuItemSeparator
- outputChart
- outputConnectionStatus
- outputDeclaration
- outputProgress
- panelBorder
- panelGroup
- panelPositioned
- panelPopup
- panelSeries
- panelStack
- panelTabSet
  - panelTab
- selectInputDate
- selectInputText
- tree
- outputStyle

For details about these components, visit the ICEfaces Online Reference page at:

http://www.icesoft.com/support/icefaces_docs.html

Styling the ICEfaces Component Suite

The ICEfaces Component Suite fully supports consistent component styling via a set of predefined CSS style classes and associated images. Changing the component styles for a web application developed with the ICEfaces Component Suite is as simple as changing the CSS used.

A set of predefined style sheets are available to be used as-is, or customized to meet the specific requirements of the application. There are two predefined ICEfaces style sheets included:

- xp.css
- royale.css

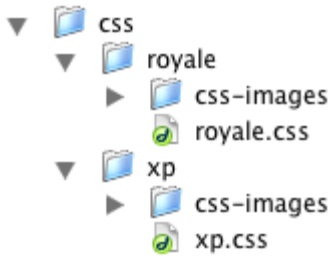
Both of these style sheets provide definitions for all style classes used in the ICEfaces Component Suite. The ICEfaces Component Suite renderers will render default style class attributes based on the style classes defined in the active style sheets. In order to use these style classes, page developers must specify a style sheet in their page.

Developers may also create their own custom style sheet based on a predefined ICEfaces style sheet. If the style class names match those defined in the ICEfaces style sheets, the ICEfaces components will use the specified styles by default, without the need to specify the style class names explicitly on each component.

> **Note:** The default CSS class names associated with each component are listed in the component's TLD (taglib) description.

The two predefined styles included with ICEfaces each consist of a style sheet and an image directory. Figure 11 shows an example of the CSS directory structure.

Figure 11 CSS Directory Structure



To use a predefined style sheet with an ICEfaces application, all the page developer needs to do is add the desired CSS link to the page:

XP

```
<link rel="stylesheet" type="text/css" href="./xmlhttp/css/xp/xp.css" />
```

Royale

```
<link rel="stylesheet" type="text/css" href="./wxmlhttp/css/xp/royale.css" />
```

The **`xmlhttp/css/xp/`** path is automatically resolved by ICEfaces and all needed resources are loaded from the ICEfaces.jar. If a developer would like to modify a particular style sheet or create a customized style sheet, the XP and Royale styles source code can be found in the "resources" directory of the ICEfaces bundle.

Using the ICEfaces Focus Management API

The ICEfaces Focus Management API consists of a single method, requestFocus(). This method is used to communicate a component focus request from the application to the client browser.

The following ICEfaces extended components have implemented the requestFocus method:

- **com.icesoft.faces.component.ext.HtmlCommandButton**
- **com.icesoft.faces.component.ext.HtmlCommandLink**
- **com.icesoft.faces.component.ext.HtmlInputSecret**
- **com.icesoft.faces.component.ext.HtmlInputText**
- **com.icesoft.faces.component.ext.HtmlInputTextarea**
- **com.icesoft.faces.component.ext.HtmlSelectBooleanCheckbox**
- **com.icesoft.faces.component.ext.HtmlSelectManyCheckbox**
- **com.icesoft.faces.component.ext.HtmlSelectManyListbox**
- **com.icesoft.faces.component.ext.HtmlSelectManyMenu**
- **com.icesoft.faces.component.ext.HtmlSelectOneListbox**
- **com.icesoft.faces.component.ext.HtmlSelectOneMenu**
- **com.icesoft.faces.component.ext.HtmlSelectOneRadio**

To use the ICEfaces Focus Management API in your application, you must use component bindings in your applications web pages.

In the following example code snippets, the ice:inputText is bound to an HtmlInputText instance named westText in the application backing bean.

Example application page:

```
<ice:inputText value="West"
    binding="#{focusBean.westText}"
/>
```

Example application backing bean:

```
import com.icesoft.faces.component.ext.HtmlInputText;

private HtmlInputText westText = null;

public HtmlInputText getWestText() {
    return westText;
}

public void setWestText(HtmlInputText westText) {
    this.westText = westText;
}
```

In the following example code snippets, the requestFocus calls are made in a valueChangeListener which is also implemented in the application's backing bean.

> **Note:** The component bindings must be visible to the valueChangeListener.

Example application page:

```
<ice:selectOneRadio value="#{focusBean.selectedText}"
   styleClass="selectOneMenu"
   valueChangeListener="#{focusBean.selectedTextChanged}"
   partialSubmit="true">
      <f:selectItem itemValue="#{focusBean.NORTH}"   itemLabel="North" />
      <f:selectItem itemValue="#{focusBean.WEST}"   itemLabel="West" />
      <f:selectItem itemValue="#{focusBean.CENTER}"   itemLabel="Center" />
      <f:selectItem itemValue="#{focusBean.EAST}"   itemLabel="East" />
      <f:selectItem itemValue="#{focusBean.SOUTH}" itemLabel="South" />
</ice:selectOneRadio>
```

Example application backing bean:

```
public void selectedTextChanged(ValueChangeEvent event) {
    selectedText = event.getNewValue().toString();

    if (selectedText.equalsIgnoreCase(NORTH)) {
        this.northText.requestFocus();
    } else if (selectedText.equalsIgnoreCase(WEST)){
        this.westText.requestFocus();
    } else if (selectedText.equalsIgnoreCase(CENTER)) {
        this.centerText.requestFocus();
    } else if (selectedText.equalsIgnoreCase(EAST)){
        this.eastText.requestFocus();
    } else if (selectedText.equalsIgnoreCase(SOUTH)) {
        this.southText.requestFocus();
    }

}
```

Chapter 5 Advanced Topics

This chapter describes several advanced features using ICEfaces. You can read or use these sections as a guide to help you advance your use of ICEfaces:

- **Server-initiated Rendering API**
- **Creating Drag and Drop Features**
- **Adding and Customizing Effects**
- **ICEfaces Tutorial: Creating Direct-to-DOM Renderers for Custom Components**

Server-initiated Rendering API

One of the unique, powerful features of ICEfaces is the ability to trigger updates to the client's user interface based on dynamic state changes within the application. It is possible for the application to request updates for one or more clients based on a very simple low-level rendering API.

However, even though the low-level API is simple, it is fairly easy to use incorrectly, which can result in:

- bad user experience
- unexpected application behavior
- poor performance
- inability to scale the application

The purpose of the Server-initiated Rendering API is to provide an effective way for developers to leverage the power of the server-side rendering feature of ICEfaces, without exposure to any of the potential pitfalls of using the low-level rendering API.

PersistentFacesState.render()

At the most basic level, server-initiated rendering relies on the PersistentFacesState. Each client that interacts with an ICEfaces application can be referenced with a unique PersistentFacesState.

Note: From within an application, it is mandatory to call the static getInstance() method from a managed-bean constructor and use the returned reference for future render calls.

For example, if you had a managed bean named User, you would do something like this:

```
public class User {

  private PersistentFacesState state;

  public User() {
    state = PersistentFacesState.getInstance();
  }
}
```

Once you have the reference to the PersistentFacesState, you can then use it to initiate a render call whenever the state of the application requires it.

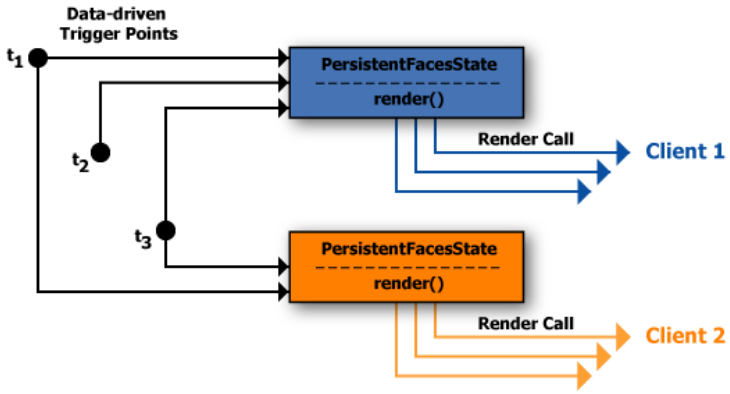
```
  state.render();
```

The render() method kicks off the render phase of the JSF lifecycle. When this is done, the ICEfaces framework detects any changes that should be sent to the client, packages them up, and sends them on their way.

Figure 12 below illustrates the use of the low-level render() API.

> **Note:** It is important to note that the ICEfaces framework synchronizes operations during a render call to ensure that the server-side DOM remains uncorrupted. While a render is in progress, subsequent calls will block waiting for the current render pass to complete.

Figure 12 Low-level Server-initiated Rendering



Rendering Considerations

The Server-initiated Rendering API is designed to avoid potential pitfalls that can occur when using the PersistentFacesState.render() call. Specifically, the implementation addresses the following characteristics of dynamic server-initiated rendering.

Concurrency

It is highly recommended to only call the render() method from a separate thread to avoid deadlock and other potential problems. For example, client updates can be induced from regular interaction with

the user interface. This type of interaction goes through the normal JSF life cycle, including the render phase. Calling a server-initiated render from the same thread that is currently calling a render (based on user interaction) can lead to unexpected application behavior. The Server-initiated Rendering implementation in ICEfaces uses a configurable thread pool to address concurrency issues, and to provide bounded thread usage in large-scale deployments.

Performance

Calling the render() method is relatively expensive so you want to ensure that you only call it when required. This is an important consideration if your application can update its state in several different places. You may find yourself sprinkling render calls throughout the code. Done incorrectly, this can lead to render calls being queued up unnecessarily and more render calls executing than actually needed. The issue is compounded with the number of clients, as application state changes may require the render() method to be called on multiple users—potentially all the currently active users of the application. In these cases, it is additionally imperative that only the minimum number of render calls be executed. The Server-initiated Rendering implementation in ICEfaces coalesces render requests to ensure that the minimum number of render calls are executed despite multiple concurrent render requests being generated in the application.

Scalability

Concurrency and performance issues both directly influence scalability. As mentioned, server-side render calls should be called in a separate thread within the web application. However, creating one or more separate threads for every potential user of the system can adversely affect scalability. As the number of users goes up, thread context switching can adversely affect performance. And since rendering is expensive, too many/frequent render calls can overwhelm the server CPU(s), reducing the number of users that your application can support. The Server-initiated Rendering implementation in ICEfaces uses a configurable thread pool for rendering, bounds thread usage in the application, and facilitates application performance tuning for large-scale deployments.

Rendering Exceptions

Server-initiated rendering does not always succeed, and can fail for a variety of reasons including recoverable causes, such as a slow client failing to accept recent page updates, and unrecoverable causes, such as an attempt to render a view with no valid session. Rendering failure is reported to the application by the following exceptions:

RenderingException

The RenderingException exception is thrown whenever rendering does not succeed. In this state, the client has not received the recent set of updates, but may or may not be able to receive updates in the future. The application should consider different strategies for TransientRenderingException and FatalRenderingException subclasses.

TransientRenderingException

The TransientRenderingException exception is thrown whenever rendering does not succeed, but may succeed in the future. This is typically due to the client being heavily loaded or on a slow connection.

In this state, the client will not be able to receive updates until it refreshes the page, and the application should consider a back-off strategy on rendering requests with the particular client.

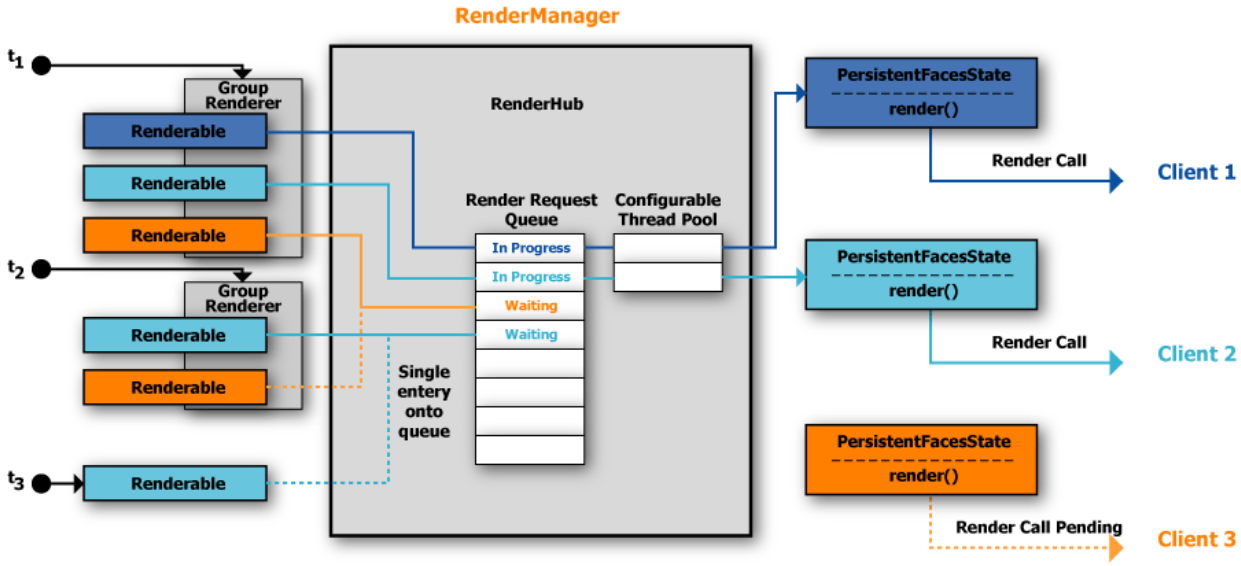
FatalRenderingException

The FatalRenderingException exception is thrown whenever rendering does not succeed and is typically due to the client no longer being connected (such as the session being expired). In this state, the client will not be able to receive updates until it reconnects, and the server should clean up any resources associated with the particular client.

Server-initiated Rendering Architecture

The server-initiated rendering architecture is illustrated in Figure 13 below.

Figure 13 Group Renderers



The key elements of the architecture are:

- **Renderable**: A session-scoped bean that implements the Renderable interface and associates the bean with a specific PersistentFacesState. Typically, there will be a single Renderable per client.
- **RenderManager**: An application-scoped bean that maintains a RenderHub, and a set of named GroupAsyncRenderers.
- **RenderHub**: Implements coalesced rendering via a configurable thread pool, ensuring that the number of render calls is minimized, and thread consumption is bounded.
- **GroupAsyncRenderer**: Supports rendering of a group of Renderables. GroupAsyncRenderers can support on-demand, interval, and delayed rendering of a group.

The following sections examine each of these elements in detail.

Renderable Interface

The Renderable interface is very simple:

```
public interface Renderable {

    public PersistentFacesState getState();

    public void renderingException(RenderingException renderingException);
}
```

The typical usage is that a session-scoped, managed-bean implements the interface and provides a getter for accessing the reference to the PersistentFacesState that was retrieved in the constructor. Since the rendering is all done via a thread pool, the interface also defines a callback for any RenderingExceptions that occur during the render call. Modifying our earlier example of a User class, it now looks like this:

```
public class User implements Renderable {

    private PersistentFacesState state;

    public User() {
        state = PersistentFacesState.getInstance();
    }

    public PersistentFacesState getState(){
        return state;
    }

    public void renderingException(RenderingException renderingException){
        //Logic for handling rendering exceptions.  The application
        //is responsible for implementing the policy for the different
        //types of RenderingExceptions.
    }
}
```

Now that the User can be referred to as a Renderable, you can use instances of User with the RenderManager and/or the various implementations of GroupAsyncRenderers.

RenderManager Class

There should only be a single RenderManager per ICEfaces application. The best and easiest way to ensure this is to create an application-scoped, managed-bean in the faces-config.xml configuration file and pass the reference into one or more of your managed beans. To continue our example, you could create a RenderManager and provide a reference to each User by setting up the following in the faces-config.xml file.

```
<managed-bean>
  <managed-bean-name>renderMgr</managed-bean-name>
  <managed-bean-class>com.icesoft.faces.async.render.RenderManager
     </managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>user</managed-bean-name>
  <managed-bean-class>com.icesoft.app.User</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>renderManager</property-name>
```

```
    <value>#{renderMgr}</value>
  </managed-property>
</managed-bean>
```

The User class needs a setter method to accommodate this:

```
public class User implements Renderable {

    private PersistentFacesState state;
    private RenderManager renderManager;

    public User() {
        state = PersistentFacesState.getInstance();
    }

    public PersistentFacesState getState(){
        return state;
    }

    public void setRenderManager( RenderManager renderManager ){
        this.renderManager = renderManager;
    }
    public void renderingException(RenderingException renderingException){
        //Logic for handling rendering exceptions.  The application
        //is responsible for implementing the policy for the different
        //types of RenderingExceptions.
    }
}
```

Once you have a reference to the RenderManager, you can request a render to be directly performed on instances that implement the Renderable interface:

```
renderManager.requestRender( aRenderable );
```

RenderHub

The heart of the RenderManager is something called the RenderHub. The RenderHub basically consists of a queue for storing Renderables and a thread pool for executing render calls on the Renderables in the queue. Each call to RenderManager.requestRender(Renderable aRenderable) puts the provided Renderable on the queue and threads from the thread pool are used to execute the render calls.

By using a configurable thread pool, thread resources can be managed on an application basis. Instead of each session bean creating its own threads, all render calls are managed through the RenderHub ensuring that thread resources are managed in a scalable manner. The other important characteristic of the RenderHub is that renders are coalesced. This is easiest to explain with an example.

Suppose that a user has opened a page that contains information that is updated via multiple server-initiated render calls: a ticking clock, a stock update, and a chat log. One possible scenario that can happen is that a render call is made due to the clock ticking. While this render call is being processed, another render call comes in because the stock price has been updated. Since the current render call may not include this change, a render request is added to the queue. Now let's suppose that the chat log gets updated and another render is requested while the initial request is still processing and the second request is still on the queue. This third request is unnecessary since the request on the queue will update all outstanding changes to the DOM. So the RenderHub ignores this third request and does not add it to the queue. Doing this makes it much safer to insert render requests throughout your application knowing that they won't get queued up unnecessarily.

GroupAsyncRenderer Implementations

Being able to render individual users in a safe and scalable manner is useful for many types of applications. However, what if you want to request a render for a group or all users of an application? As an example, consider a chat application or a chat component in your application. There could be many users in the same chat group and any update to the chat transcript should result in all users getting notified.

To handle group rendering requests in a scalable fashion, the Rendering API provides implementations of the GroupAsyncRenderer base class. There are currently three implementations of GroupAsyncRenderer you can use. Each implementation allows you to add and remove Renderable instances from their collection.

- **OnDemandRenderer** - When requestRender() method is called, goes through each Renderable in its collection and requests an immediate render call.
- **IntervalRenderer** - Once you get the initial reference to the IntervalRenderer, you set the desired interval and then call the requestRender() method which requests a render call on all the Renderables at the specified interval.
- **DelayRenderer** - Calls a single render on all the members of the collection at some future point in time.

The best way to get one of the GroupAsyncRenderer implementations is to use one of the methods provided by the RenderManager:

```
RenderManager.getOnDemandRenderer(String name);
RenderManager.getIntervalRenderer(String name);
RenderManager.getDelayRenderer(String name);
```

As you can see, each GroupAsyncRenderer has a name, which the RenderManager uses to track each GroupAsyncRenderer so that each request using the unique name returns the same instance. That way, it is easy to get a reference to the same renderer from different parts of your application.

To expand our previous example of a User, we can augment our code to use a named GroupAsyncRenderer that can be called on demand when a stock update occurs. In the example code, we are using a fictitious stockEventListener method to listen for events that indicate a stock has changed. When that occurs, we'll call requestRender() on the GroupAsyncRenderer. Every user that is a member of that group will have a render call executed.

```
public class User implements Renderable {

    private PersistentFacesState state;
    private RenderManager renderManager;
    private OnDemandRenderer stockGroup;

    public User() {
        state = PersistentFacesState.getInstance();
    }

    public PersistentFacesState getState(){
        return state;
    }

    public void setRenderManager( RenderManager renderManager ){
        this.renderManager = renderManager;
        OnDemandRenderer stockGroup;
        stockGroup = renderManager.getOnDemandRenderer( "stockGroup" );
```

```
    }

    public void stockEventListener( StockEvent event ){
        if( event instanceof StockValueChangedEvent ){
            stockGroup.requestRender();
        }
    }
}
```

Creating Drag and Drop Features

This tutorial guides you though creating an application that uses Drag and Drop.

For more information about building and deploying your project, refer to the tutorials in **Chapter 4** of the **ICEfaces Getting Started Guide**.

Creating a Draggable Panel

The dragdrop1 directory contains an empty project, with a single .jspx file in the dragdrop1/web folder. dragDrop.jsp is empty aside from the basic code needed for an ICEFaces application.

1. Begin by adding a form.

```
<ice:form>
```

> **Note:** Note: All drag and drop panels must be in a form.

2. Inside the form, add a group panel and set the draggable attribute to true.The following shows example code used to do this. It also has some added style information to give the panel a default size.

```
<ice:form>
       <ice:panelGroup style="z-index:10;width:200px;height:60px;background:
                              #ddd;border:2px solid black;
                              cursor:move;" draggable="true">
       </ice:panelGroup>
</ice:form>
```

3. Deploy the application.

```
http://localhost:8080/dragdrop1
```

To display the drag feature, click on the grey box to drag it around the screen.

Adding Drag Events

In the dragdrop1/src/com/icesoft/tutorial/ directory you will find the class DragDropBean, which is an empty class. The class is mapped to the name dragDropBean in faces-config.xml.

1. Add a listener that will receive Drag events from the panel and a String property to display the value to the DragDropBean class.

```
private String dragPanelMessage = "Test";

    public void setDragPanelMessage(String s){
        dragPanelMessage = s;
    }

    public String getDragPanelMessage(){
        return dragPanelMessage;
    }

    public void dragPanelListener(DragEvent dragEvent){
        dragPanelMessage =
         "DragEvent = " + DragEvent.getEventName(dragEvent.getEventType());
    }
```

2. In the dragDrop.jspx, set the dragListener attribute to point to dragPanelListener and output the message.

```
<ice:panelGroup style="z-index:10;width:200px;height:60px;
             background:#ddd;border:2px solid black;cursor:move;"
             draggable="true" dragListener="#{dragDropBean.dragPanelListener}">

      <ice:outputText value="#{dragDropBean.dragPanelMessage}"/>
</ice:panelGroup>
```

3. Deploy the application.

```
http://localhost:8080/dragdrop1
```

Now when you drag the panel around, the message changes. When you start to drag, the event is dragging, and when released, it is drag_cancel.

There are five Drag and Drop Event types.

Event Type	Effect
Dragging	A panel is being dragged.
Drag Cancel	Panel has been dropped, but not on a drop target.
Dropped	Panel has been dropped on a drop target.
Hover Start	Panel is hovering over a drop target.
Hover End	Panel has stopped hovering over a drop target.

4. To see the other messages, add a new group panel with the dropTarget attribute set to true. Add the panel below the draggable panel, but within the same form.

```
<ice:panelGroup style="z-index:0;width:250px;height:100px;
                       background:#FFF;border:2px solid black;"
                       dropTarget="true">
  </ice:panelGroup>
```

5. Deploy the application.

```
http://localhost:8080/dragdrop1
```

Now when you drag the panel over the dropTarget panel, the messages will change.

Setting the Event dragValue and dropValue

Panels have a dragValue and a dropValue, which can be passed in Drag and Drop events.

1. In the dropTarget panel, set the dropValue attribute to "One", and then add a second panel group with a dropValue of "Two".

```
<ice:panelGroup style="z-index:0;width:250px;height:
                    100px;background:#FFF;border:2px solid black;"
                    dropTarget="true" dropValue="One">
       <ice:outputText value="One"/>
</ice:panelGroup>
<ice:panelGroup style="z-index:0;width:250px;height:100px;background:
                    #FFF;border:2px solid black;"
                    dropTarget="true" dropValue="Two">
       <ice:outputText value="Two"/>
</ice:panelGroup>
```

2. In the Drag Event, change the listener to extract the drop value from the drop targets.

```
public void dragPanelListener(DragEvent dragEvent){
        dragPanelMessage =
            "DragEvent = " + DragEvent.getEventName(dragEvent.getEventType())
        + " DropValue:" + dragEvent.getTargetDropValue();
}
```

3. Deploy the application.

```
http://localhost:8080/dragdrop1
```

Now when you hover or drop the drag panel on one of the drop targets, the message will change.

Event Masking

Event masks prevent unwanted events from being fired, thereby improving performance. panelGroup contains two attributes for event masking: dragMask and dropMask.

1. Change the dragMask in the drag panel to filter all events except dropped.

```
<ice:panelGroup style="z-index:10;width:200px;height:60px;
                    background:#ddd;border:2px solid black; cursor:move;"
                    draggable="true"
                    dragListener="#{dragDropBean.dragPanelListener}"
                    dragMask="dragging,drag_cancel,hover_start,hover_end">
```

2. Deploy the application.

Now when you drag the panel around, the message will only change when dropped on the “One” or “Two” panels. All of the other events are masked.

Adding and Customizing Effects

This tutorial demonstrates how to add and customize effects in ICEfaces applications. Use the effects1 project located in your ICEfaces tutorial directory.

For more information about building and deploying your project, refer to the tutorials in **Chapter 4** of the **ICEfaces Getting Started Guide**.

Creating a Simple Effect

1. Open the effects1/web/effect.jsp and add the following code:

```
<ice:form>
  <ice:commandButton value="Invoke" action="#{effectBean.invokeEffect}"/>
  <ice:outputText value="Effect Test" effect="#{effectBean.textEffect}"/>
</ice:form>
```

2. In the EffectBean, add the following code:

```
private Effect textEffect;

    public Effect getTextEffect(){
        return textEffect;
    }

    public void setTextEffect(Effect effect){
        textEffect = effect;
    }

    public String invokeEffect(){
        textEffect = new Highlight();
        return null;
    }
```

3. Build and deploy the application. Open your browser to:

```
http://localhost:8080/effects1
```

4. Click the command button.

   The Output Text will be highlighted in yellow and then return to its original color.

Modifying the Effect

The following effects can be used in your ICEfaces application:

- Highlight
- Appear
- Fade
- Move
- Pulsate

The following steps demonstrate how you can modify several different effects.

1. In the EffectBean, add a Boolean flag, and then toggle the color from yellow to red on each click of the command button.

```
private boolean flag;
   public String invokeEffect(){
        if(flag){
            textEffect = new Highlight("#FF0000");
        }else{
            textEffect = new Highlight("#FFFF00");
        }
        flag = !flag;
        return null;
    }
```

2. Build and deploy the application. Open your browser to:

```
http://localhost:8080/effects1
```

   Now each time the effect is clicked, the highlight color switches between yellow and red.

> **Note:** Each effect is fired once per instance. Each time you want to fire an effect, it must be with a new effect instance object or you can set the fired flag to false. For example, `Effect.setFired(false);`

3. To change the action handler to switch from pulsate to highlight.

```
public String invokeEffect(){
        if(flag){
            textEffect = new Highlight();
        }else{
            textEffect = new Pulsate();
        }
        flag = !flag;
        return null;
    }
```

4. Build and deploy the application. Open your browser to:

```
http://localhost:8080/effects1
```

5. Effects can also be invoked locally. Change the effect attribute from effect to onmouseovereffect.

```
<ice:outputText value="Effect Test"
onmouseovereffect="#{effectBean.textEffect}"/>
```

6. Build and deploy the application. Open your browser to:

```
http://localhost:8080/effects1
```

7. Click the invoke button and then move the mouse over the text. The text will highlight immediately.
8. Click the invoke button again to change the local effect. Now when the mouse moves over the text, it will pulsate.

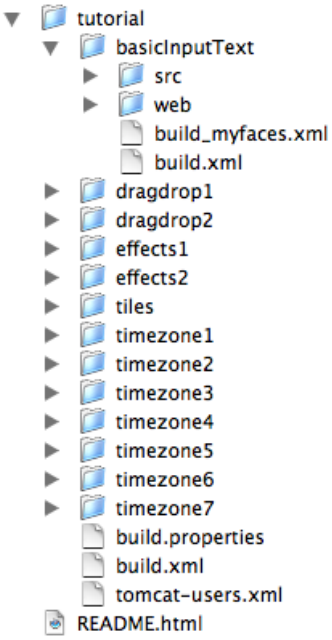
ICEfaces Tutorial: Creating Direct-to-DOM Renderers for Custom Components

This tutorial guides you through creating a Direct-to-DOM (D2D) custom renderer using ICEfaces. For more information on Direct-to-DOM rendering, refer to Chapter 3, Key Concepts.

> **Note:** This is an advanced topic and we recommend that you only read this tutorial if you are building custom components or porting existing component renderers.

The files used in this tutorial are organized in the directory structure shown in Figure 14. Before starting this tutorial, we recommend that you ensure that ICEfaces is installed properly and that your environment is configured to deploy and execute ICEfaces applications on your J2EE application server. Refer to Chapter 2, Configuring Your Environment for ICEfaces, in the **ICEfaces Getting Started Guide**.

Figure 14 Direct-to-DOM Rendering Tutorial Directory Structure



Creating a Direct-to-DOM Renderer for a Standard UIInput Component

This tutorial guides you through the process of creating a Direct-to-DOM renderer for a standard component. The source code for this tutorial is included in the basicInputText sample tutorial application.

The renderer is responsible for processing the form data for the component. This is the decoding of the component which is done in the decode method of the renderer. The renderer is also responsible for translating a component into a W3C DOM Element. This is the encoding of the component which is done in the encode methods of the renderer. With a standard component, such as a UIInput with no children,

you need only implement the encodeEnd method. The encodeBegin and encodeChildren methods do not need to be implemented for this renderer.

The first method to look at is decode(FacesContext, UIComponent). This method is responsible for taking any parameters that were passed in from a form post and setting the value on the component. The first thing the method does is to validate the context and component parameters, checking to ensure that they are not null. Next, the method ignores all but UIInput components. If the component is a UIInput, then any new value is extracted from the request and put on the component as the submittedValue.

```
public void decode(FacesContext facesContext, UIComponent uiComponent) {
    validateParameters(facesContext, uiComponent, null);
    // only need to decode input components
    if (!(uiComponent instanceof UIInput)) {
        return;
    }
    // only need to decode enabled, writable components
    if (isStatic(uiComponent)) {
        return;
    }
    // extract component value from the request map
    String clientId = uiComponent.getClientId(facesContext);
    if (clientId == null) {
        System.out.println("Client id is not defined for decoding");
    }
    Map requestMap = facesContext.getExternalContext().getRequestParameterMap();
    if (requestMap.containsKey(clientId)) {
        String decodedValue = (String) requestMap.get(clientId);
        // setSubmittedValue is a method in the superclass DomBasicInputRenderer
        setSubmittedValue(uiComponent, decodedValue);
    }
} // end decode
```

The next method to analyze is the encodeEnd(FacesContext, UIComponent). This method is responsible for building the DOM Element to represent the component in the browser. The encodeEnd method uses the DOM methods exposed in the ICEfaces DOMContext and the W3C DOM API. The DOMContext.attachDOMContext method is used to provide a DOMContext to the encodeEnd method. The DOMContext will be initialized if required. As part of the initialization, a root node is created using the DOMContext createRootElement method. The DOMContext API will be used to create new Elements and TextNodes. To set Element attributes and append child nodes to Elements, the W3C DOM API will be used.

```
public void encodeEnd(FacesContext facesContext, UIComponent uiComponent)
  throws IOException {
    DOMContext domContext = DOMContext.attachDOMContext(facesContext, uiComponent);
    if (!domContext.isInitialized()) {
        Element root = domContext.createRootElement("input");
        setRootElementId(facesContext, root, uiComponent);
        root.setAttribute("type", "text");
        root.setAttribute("name", uiComponent.getClientId(facesContext));
    }

    Element root = (Element) domContext.getRootNode() ;
    root.setAttribute("onkeydown",this.ICESUBMIT) ;

    // create a new String array to hold attributes we will exclude
    // for the PassThruAttributeRenderer
    String[] excludesArray = new String[1] ;
```

```
    excludesArray[0] = "onkeydown" ;

    // the renderAttributes method will not overwrite any attributes
    // contained in the excludesArray
    PassThruAttributeRenderer.renderAttributes(facesContext, uiComponent,
                                               excludesArray);
  } // end encodeEnd
```

The following is the content for the faces-config.xml file used in this tutorial. The faces config is used to configure the Renderer for the UIInput Text Component as well as any managed-beans that are required.

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC "-//Sun Microsystems, Inc.//DTD JavaServer Faces
Config 1.0//EN" "http://java.sun.com/dtd/web-facesconfig_1_0.dtd" >
<faces-config>
  <render-kit> description>Tutorial Basic Renderer</description>
    <renderer>
       <component-family>javax.faces.Input</component-family>
       <renderer-type>javax.faces.Text</renderer-type>
       <renderer-class>
         com.icesoft.tutorial.TutorialInputTextRenderer
       </renderer-class>
    </renderer>
  /render-kit>
<managed-bean>
    <managed-bean-name>tutorial</managed-bean-name>
    <managed-bean-class>com.icesoft.tutorial.TutorialBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
</faces-config>
```

Understanding Iterative Renderers

Iterative renderers are renderers that iterate over a collection of data and render markup for each data point.

A common example of an iterative renderer is the TableRenderer. It renders a UIData component which can be bound to a model bean that provides the collection of data. The renderer iterates over the collection of data and renders a table row for each element in the collection.

There are a few principles to keep in mind when authoring an iterative renderer.

An iterative renderer will render its own children. It is possible, but more complicated to do otherwise; so it is recommended that you override the method public boolean getRendersChildren() from the abstract class javax.faces.render.Renderer. The overriding method in your iterative renderer should return true:

```
public boolean getRendersChildren() { return true; }
```

In order to properly render the IDs of the DOM Elements, the iterative renderer must maintain the current index of the data set that is being rendered. This state (the current index) exists in the UIComponent class but will be maintained by the renderer since it knows when rendering of the current data point has completed and it is time to move on to the next data point. Each time the renderer finishes rendering a data point, it increments the data index in the UIComponent class. The index is accessed by the renderer class in order to formulate the IDs of the DOM Elements that it is rendering.

Index

G

I

J

K

M

P

R

S

T

V

W

X