ICEfaces

Developer's Guide

Version 1.7 Beta

Copyright

ICEsoft Technologies, Inc.
Suite 200, 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 web site: **http://www.icefaces.org**

ICEfaces Developer's Guide v1.7 Beta

February 2008

About this Guide

The **ICEfaces® Developer's Guide** is your manual to developing ICEfaces applications. 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 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, Portlets, Asynchronous HTTP Server, Seam integration,Spring Framework integration, 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.

ICEfaces Technical Support

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

http://support.icefaces.org/

Contents

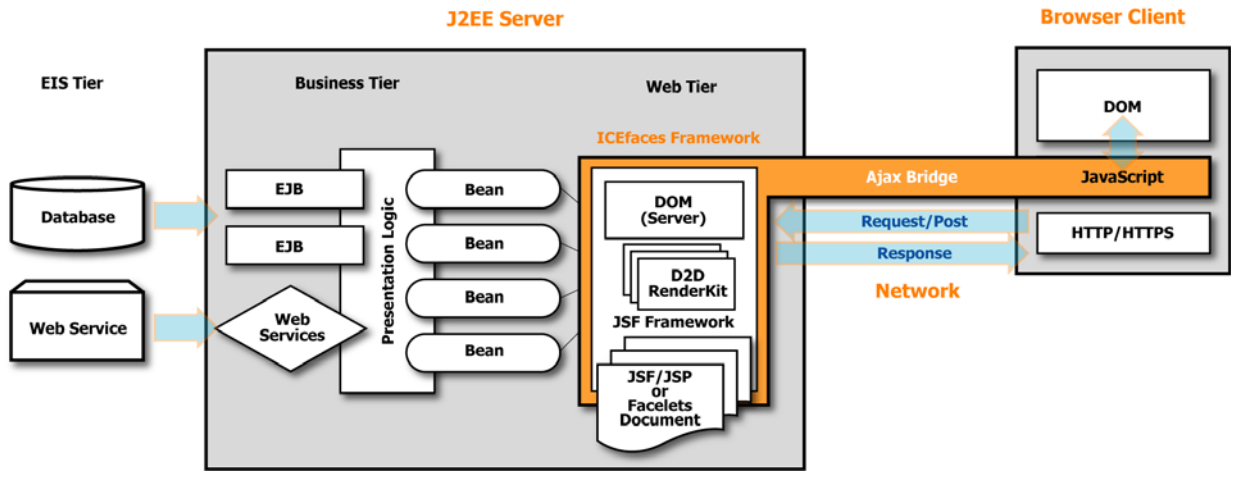
List of Figures

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.

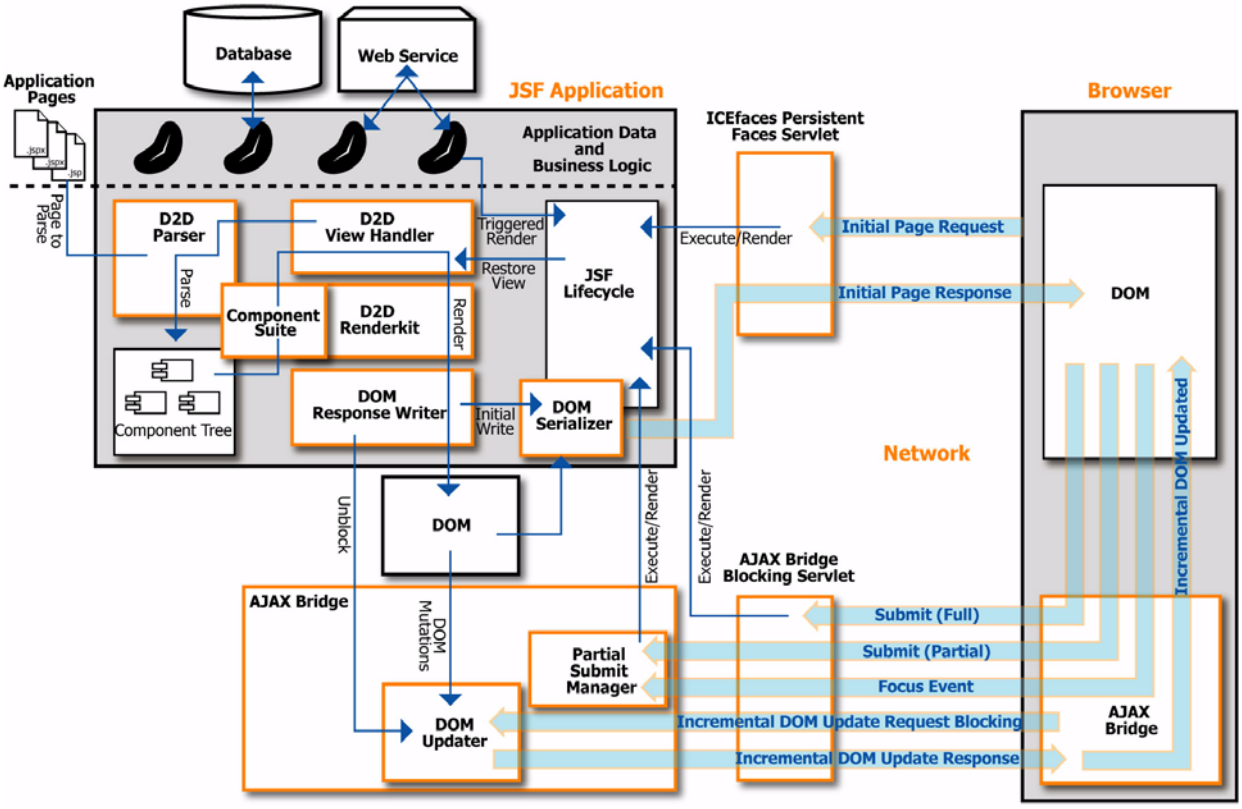
Beyond these transparent presentation features, ICEfaces introduces additional rich presentation features that the JSF developer can leverage to further enhance the user experience. Specifically, the developer can incorporate these features:

- **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.

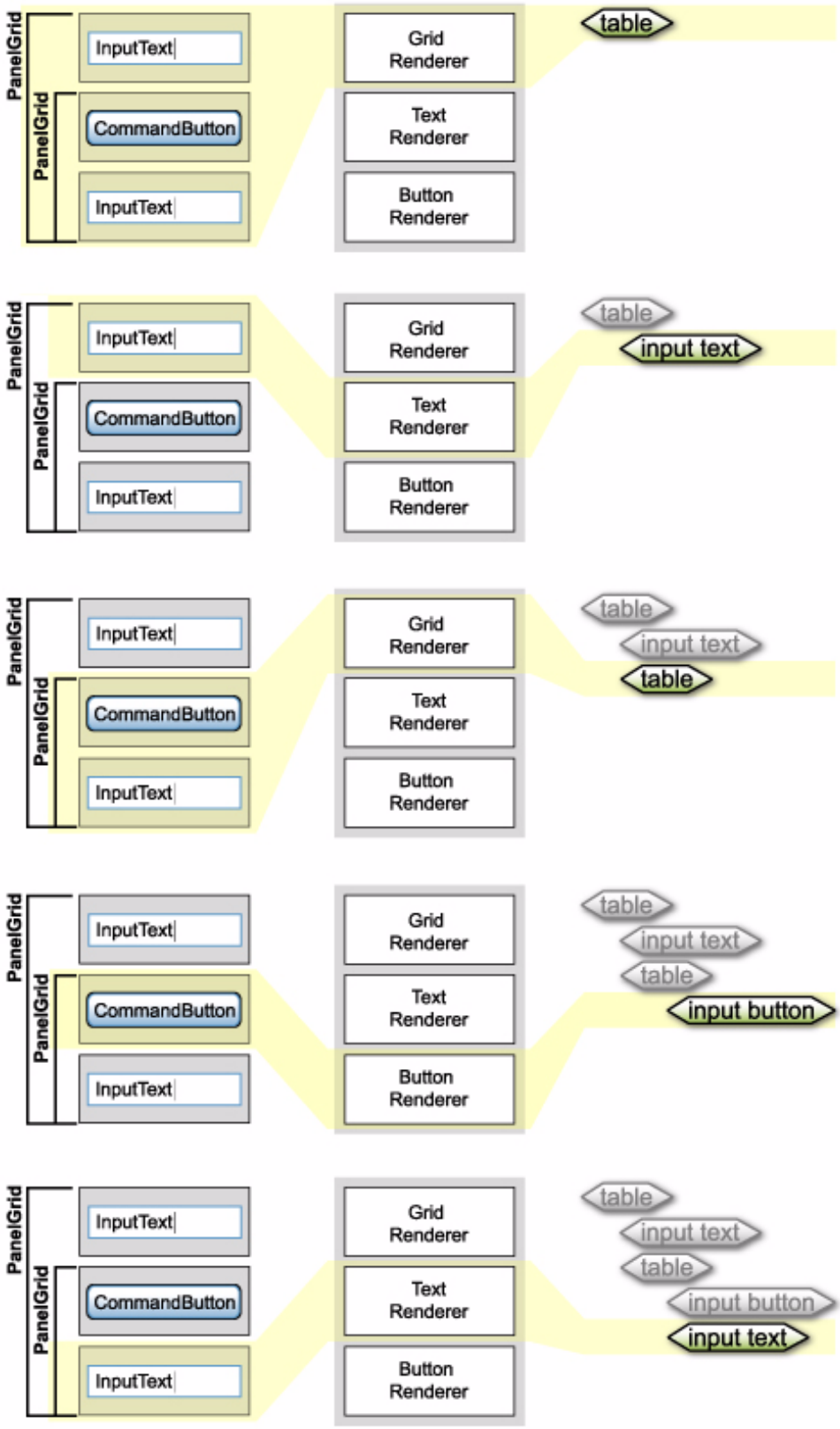
Chapter 3 Key Concepts

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

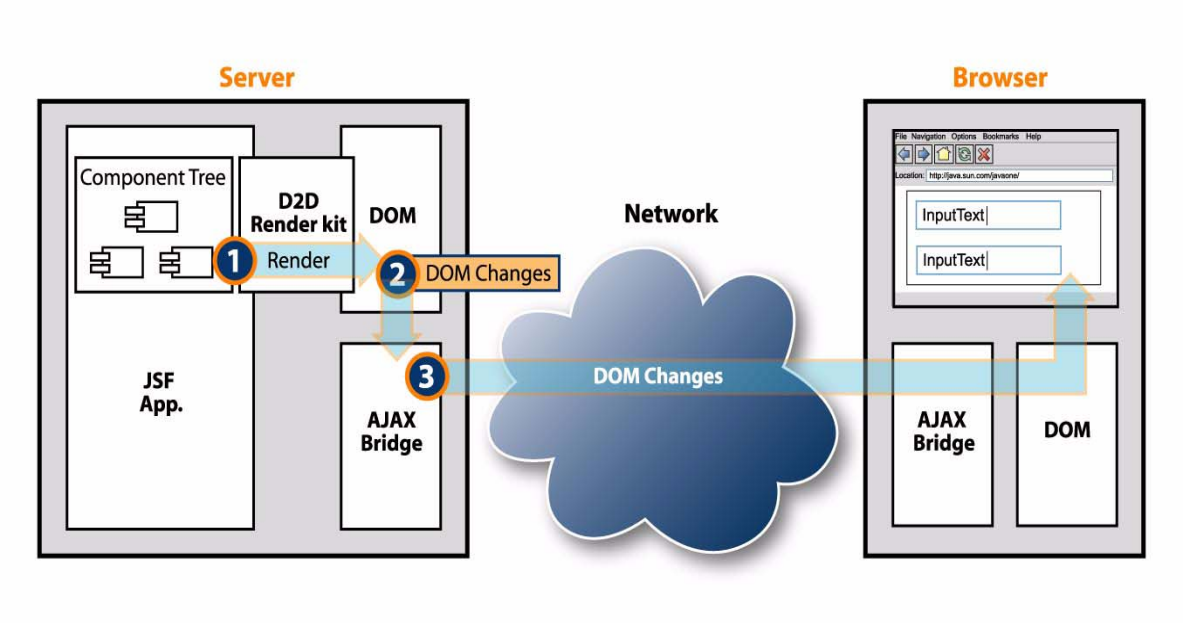
Direct-to-DOM (D2D) rendering is just what it sounds like—the ability to render a JSF component tree directly into a W3C standard DOM data structure. ICEfaces provides a Direct-to-DOM RenderKit for the standard HTML basic components available in JSF. The act of rendering a component tree into a DOM via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

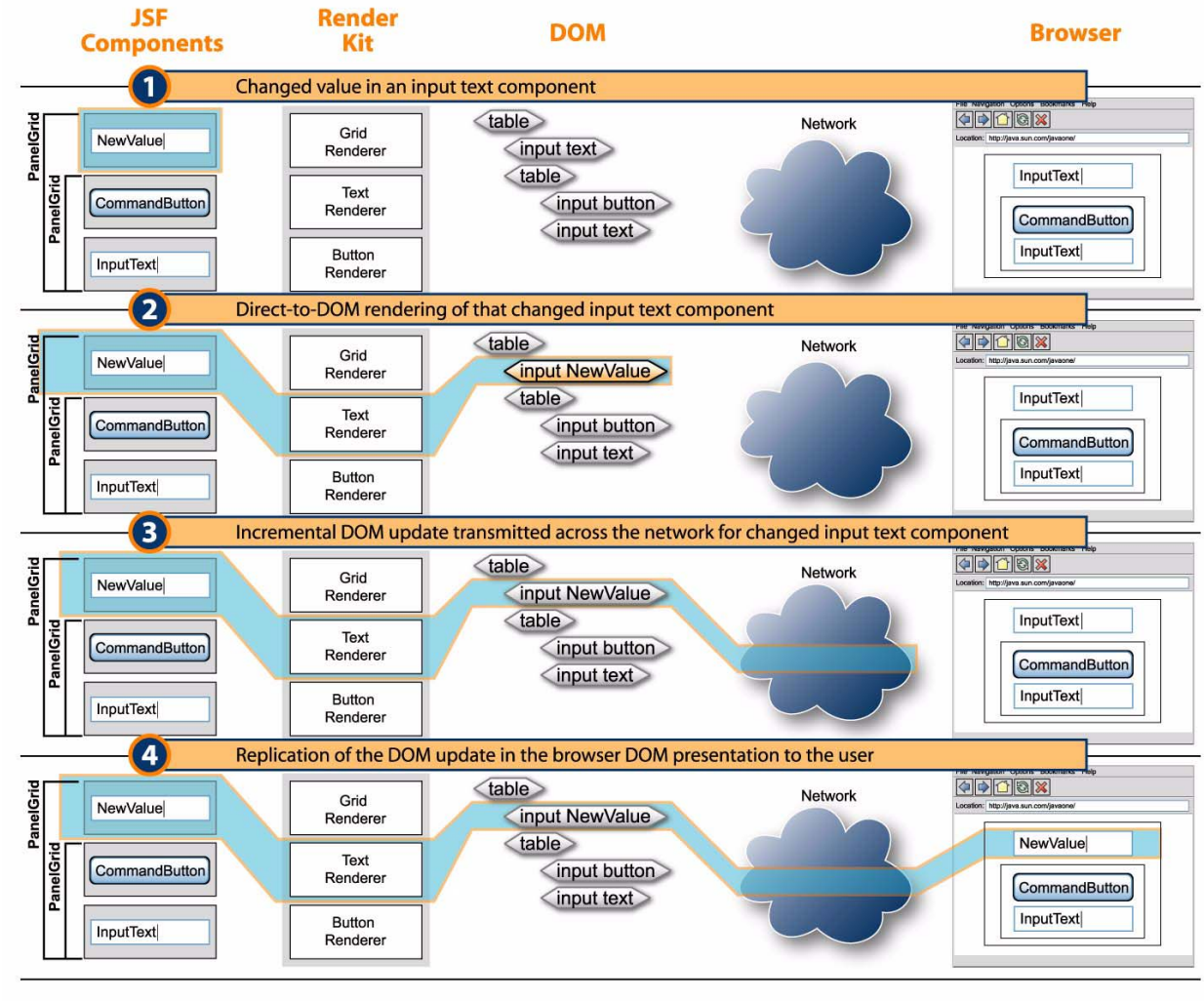


When the ICEfaces JAR is included in your JSF application, the Direct-to-DOM RenderKit is automatically configured into the application. There are no other considerations from the developer perspective. Direct-to-DOM rendering is completely transparent in the development process.

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



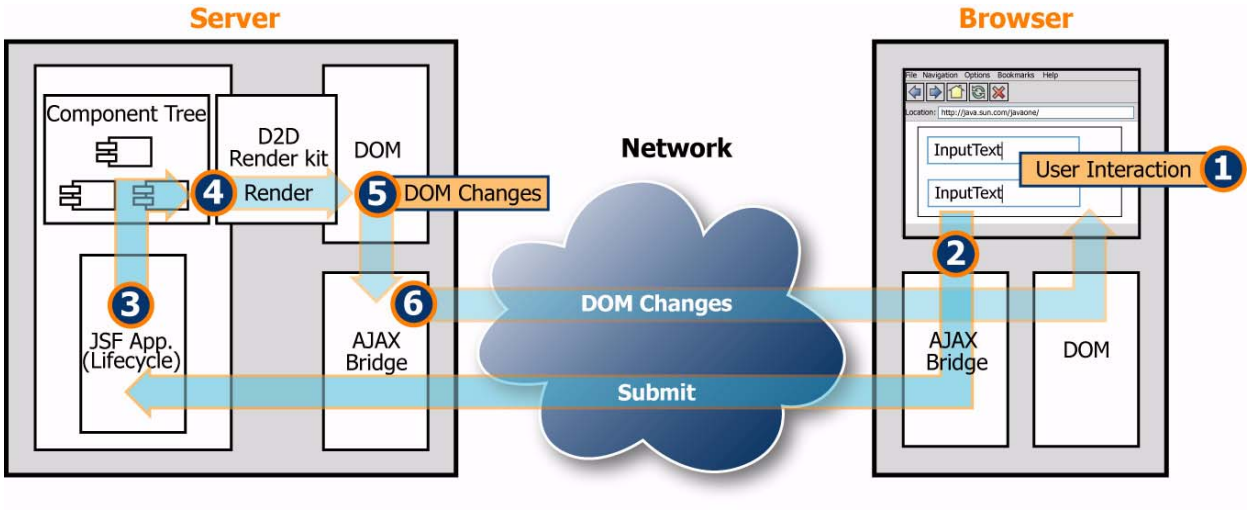
Again, incremental updates are transparent from the development perspective. As the presentation layer changes during a render pass, those changes are seamlessly realized in the client browser.

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

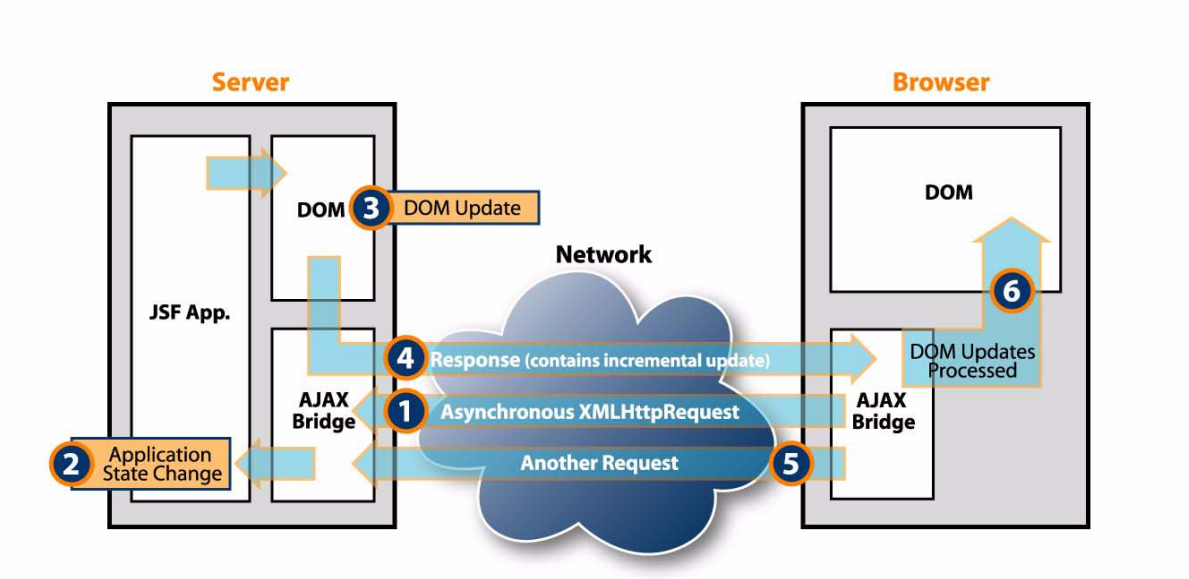


One serious deficiency with synchronous updates is that the application requires a client-generated request before it can affect presentation layer changes. If an application state change occurs during a period of client inactivity, there is no means to present changing information to the user. ICEfaces 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

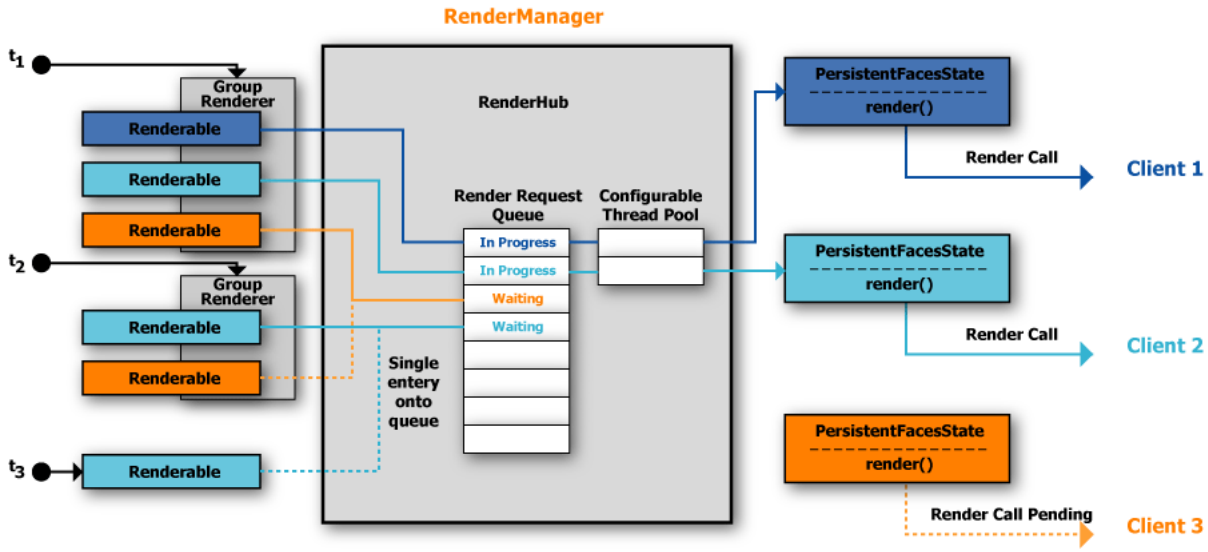
Client/server connectivity is a key requirement for ICEfaces applications to function. For this reason, ICEfaces provides connection heartbeating and status monitoring facilities in the client-side AJAX bridge, and a Connection Status component to convey connection status information to the user interface. Additionally, ICEfaces provides the ability to automatically redirect to an error page when the connection is lost. See Connection Management on page 33 for details on configuring connection management, and ICEfaces Custom Components on page 28 for additional information on the Connection Status component.

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 35.

Partial Submit – Intelligent Form Processing

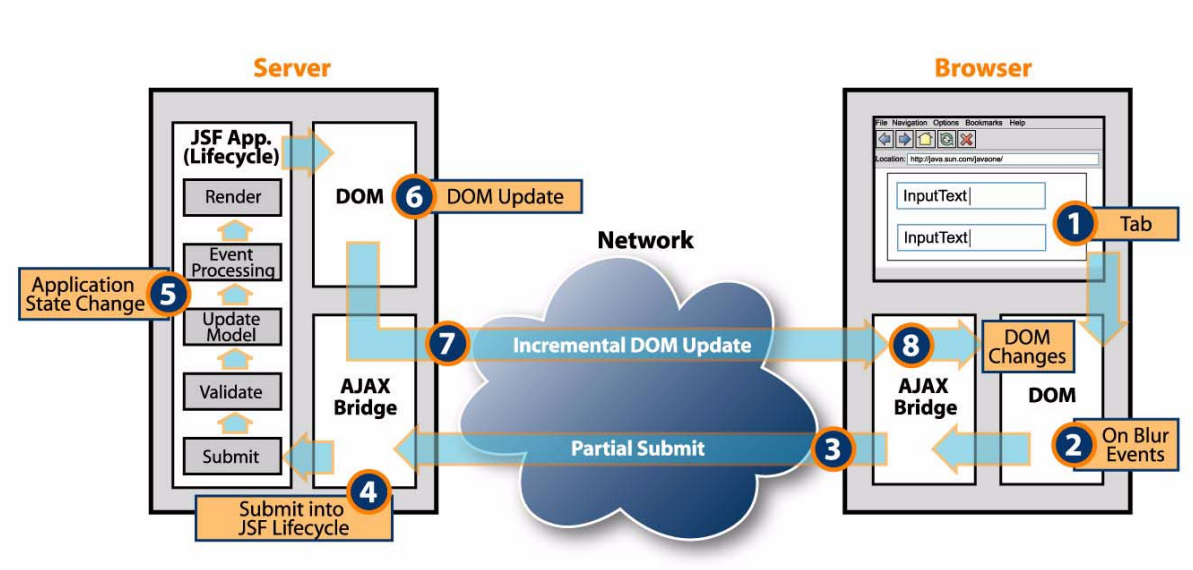
ICEfaces introduces a fine-grained user interaction model for intelligent form processing within an ICEfaces application. In JSF, the normal submit mechanism initiates the JSF application lifecycle, and as such, capabilities like client-side validation are not supported. Partial submit overcomes these 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.

Table 1 identifies the component libraries (name spaces) that ICEfaces supports

Table 1 ICEfaces-supported Component Libraries.

Name Space	Description	ICEfaces Features
www.icesoft.com/icefaces/component	ICEfaces Component Suite	• Comprehensive set of rich components • Incremental page update • Automated partial submit • Automated CSS styling/ themes • Client-side effects and animations
java.sun.com/jsf/html	Sun Standard JSF Components	• Incremental page update
myfaces.apache.org/tomahawk	Apache MyFaces Tomahawk Components	• Incremental page update

See Components Reference on page 24 for more information on using the component libraries with ICEfaces.

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 29.

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 101 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">
```

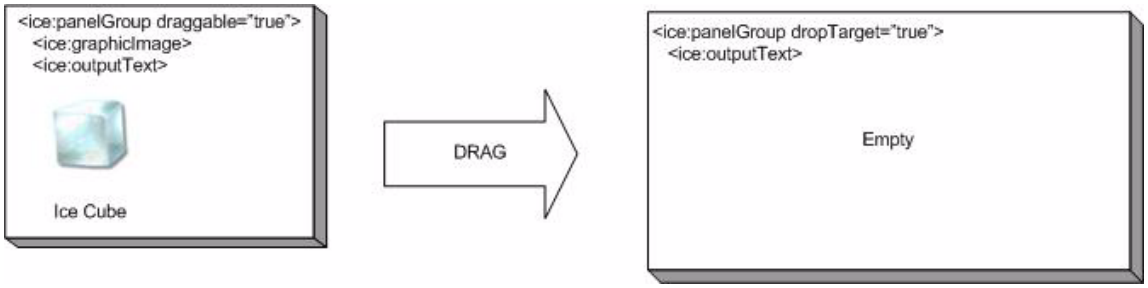
The panel group (and all of its child components) can now be dragged around the screen.

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. Table 2 lists these optional effects and their behavior.

Table 2 Drag and Drop Effects

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 42 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 45.

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 45 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".

Table 3 shows a summary of the Managed Bean Scope.

Table 3 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.

If concurrent DOM views is configured, and multiple windows are created by the user, ICEfaces uses a single blocking connection to deliver asynchronous updates to all views in the session. This ensures that all views remain current without exceeding the maximum connection limit implemented in the browser. When views are destroyed by the user, it may be necessary to release view-specific resources such as threads or database connections. A listener mechanism has been added to support this.

See **com.icesoft.faces.context.ViewListener** and **com.icesoft.faces.webapp.xmlhttpPersistenFacesState.addViewListener()** in the ICEfaces Java API for details.

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**
- **Java API Reference**
- **Configuration Reference**
- **Components Reference**
- **ICEfaces Component Suite**

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 supports the following JSF component libraries.

Table 4 JSF Component Libraries

Name Space	Description	ICEfaces Features
www.icesoft.com/icefaces/component	ICEfaces Component Suite	• Comprehensive set of rich components • Incremental page update • Automated partial submit • Automated CSS styling/ themes • Client-side effects and animations
java.sun.com/jsf/html	Sun Standard JSF Components	• Incremental page update
myfaces.apache.org/tomahawk	Apache MyFaces Tomahawk Components	• Incremental page update

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.
- Support for rich client-side component effects, such as fading, expand, collapse, etc.

For more details, see ICEfaces Component Suite on page 25.

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.

Apache MyFaces Tomahawk Components

It is possible to integrate MyFaces Tomahawk components with ICEfaces and ICEfaces Component Suite Components on the same page. Any Tomahawk components used with ICEfaces will benefit from incremental UI updates. All other aspects of using the Tomahawk components will remain the same. For detailed information on using the MyFaces Tomahawk components with ICEfaces, please see this related ICEfaces Knowledge Base article: **Status of ICEfaces Support for MyFaces Tomahawk Components**.

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.

Table 5 shows which attributes are applicable to each ICEfaces component.

Table 5 ICEfaces Component Attributes

ICEfaces Components	renderedOnUserRole	enabledOnUserRole	visible	disabled	partialSubmit
column					
columns					
commandButton	*	*	*	*	*
commandLink	*	*	*	*	*
commandSortHeader	*	*		*	
dataPaginator	*	*		*	
dataTable	*				
effect					
form	*				*
graphicImage	*		*		
inputFile	*	*		*	
inputHidden					
inputSecret	*	*	*	*	*
inputText	*	*	*	*	*
inputTextarea	*	*	*	*	*
menuBar	*				
menuItem	*				
menuItems	*				
menuItemSeparator	*				
message	*		*		
messages	*		*		
outputChart	*				
outputConnectionStatus	*				
outputDeclaration	*				
outputLabel	*		*		
outputLink	*	*	*	*	
outputProgress	*				
outputStyle					
outputText	*		*		
panelBorder	*				
panelCollapsible					
panelGrid	*		*		
panelGroup	*		*		
panelPositioned					
panelPopup	*		*		
panelSeries	*				

ICEfaces Components	renderedOnUserRole	enabledOnUserRole	visible	disabled	partialSubmit
panelStack	*				
panelTabSet	*				
panelTab	*	*		*	
rowSelector					
selectBooleanCheckbox	*	*	*	*	*
selectInputDate	*	*		*	
selectInputText	*	*		*	
selectManyCheckbox	*	*	*	*	*
selectManyListbox	*	*	*	*	*
selectManyMenu	*	*	*	*	*
selectOneListbox	*	*	*	*	*
selectOneMenu	*	*	*	*	*
selectOneRadio	*	*	*	*	*
tabChangeListener					
tree					
treeNode					

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:

- columns
- commandSortHeader
- dataPaginator
- dataTable
- effect
- inputFile
- menuBar
  - menuItem
  - menuItems
  - menuItemSeparator
- outputChart
- outputConnectionStatus
- outputDeclaration
- outputFormat
- outputProgress
- outputStyle
- panelBorder
- panelCollapsible
- panelGroup
- panelPositioned
- panelPopup
- panelSeries

- panelStack
- panelTabSet
  - panelTab
- rowSelector
- selectInputDate
- selectInputText
- tree
  - treeNode

For more information about the ICEfaces Component Suite, see the following documents on the **ICEfaces Documentation** page:

- **Component Suite Overview**
- **ICEfaces Component Suite TLD (taglib)**

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

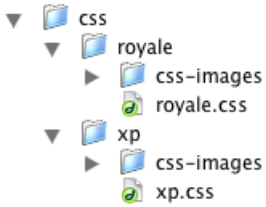
Both of these style sheets provide definitions for all style classes used in the ICEfaces Component Suite. The ICEfaces Component Suite renderers will render default style class attributes based on the style classes defined in the active style sheets. In order to use these style classes, page developers 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 themes 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 theme style sheet with an ICEfaces application, all the page developer needs to do is add the desired CSS link to the page. This can be accomplished in one of two ways:

1. Add a standard HTML link tag to the document:
   - XP

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

   - Royale

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

2. Use the ICEfaces ice:outputStyle component (recommended).
   - XP

```
<ice:outputStyle href="./xmlhttp/css/xp/xp.css" rel="stylesheet"
                                                type="text/css" /
```

   - Royale

```
<ice:outputStyle href="./xmlhttp/css/royale/royale.css" rel="stylesheet"
                                                        type="text/css" />
```

The ice:outputStyle component has the following advantages over the HTML link tag:

- Automatically includes browser-specific variants of the main style sheet in the page to adapt the theme styling to account for differences in each browsers' CSS support. See the TLD (taglib) documentation for the ice:outputStyle component for details.
- Provides the ability to dynamically change the theme style sheet at runtime via a value binding on the component's href attribute.

```
<ice:outputStyle href="#{styleBean.activeTheme}" rel="stylesheet"
                                                 type="text/css" />
```

Note: In the examples above, the **`xmlhttp/css/xp/`** path is automatically resolved by ICEfaces and all needed resources are loaded from the ICEfaces.jar.

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 use these sections as a guide to help you advance your use of ICEfaces:

- **Connection Management**
- **Server-initiated Rendering API**
- **Creating Drag and Drop Features**
- **Adding and Customizing Effects**
- **Developing Portlets with ICEfaces**
- **Introduction to the Asynchronous HTTP Server**
- **JBoss Seam Integration**
- **Spring Framework Integration**
- **ICEfaces Tutorial: Creating Direct-to-DOM Renderers for Custom Components**

Connection Management

ICEfaces provides the following advanced connection management features:

- **Asynchronous Heartbeating**
- **Managing Connection Status**
- **Managing Redirection**
- **Compressing Static Resources**

Asynchronous Heartbeating

At the core of EPS connection management is a configurable connection heartbeat mechanism. Heartbeating improves the long-term health of the connection by keeping it active, and closely monitors the connection status based on heartbeat responses. Heartbeating is configured in the application **web.xml** file using the context parameters in the code examples below.

The following code example defines the time in milliseconds between heartbeat messages. The default value is 20000 (20 seconds).

```
<context-param>
```

```
    <param-name>com.icesoft.faces.heartbeatInterval</param-name>
    <param-value>20000</param-value>
</context-param>
```

For debugging purposes, heartbeating can be toggled on and off from the client browser using the following: **CTR+SHIFT+.** (that is, press the Control and Shift keys and type “.”).

The following code example defines how long, in milliseconds, that heartbeat monitor will wait for a response prior to timing out. The default value is 3000 (3 second).

```
<context-param>
    <param-name>com.icesoft.faces.heartbeatTimeout</param-name>
    <param-value>3000</param-value>
</context-param>
```

The following code example defines the number of timeout/retries allowed before the connection is considered failed. The default value is 3.

```
<context-param>
    <param-name>com.icesoft.faces.heartbeatRetries</param-name>
    <param-value>3</param-value>
</context-param>
```

Managing Connection Status

Heartbeating enables an additional state for connections, so the available states are:

- idle
- waiting
- caution
- lost

The caution state occurs if heartbeats go missing, but retries are in progress. If the retries fail, the connection state will transition to lost, and if a retry succeeds the connection state will return to idle.

The outputConnectionStatus component in the ICEfaces Component Suite recognizes all four states, so applications can incorporate visual indicators for connection status. 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.

Managing Redirection

Redirect on Connection Lost

When a connection is lost, 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.connectionLostRedirectURI**.

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

```
</context-param>
```

If not specified, the default 'Connection Lost' overlay will be rendered or if the status component is present, the 'Connection Lost' icon.

Compressing Static Resources

By default, ICEfaces gzip compresses static resource files such as .js, and .css before sending them to the browser. The gzip compression can be disabled via the following configuration parameter:

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

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.

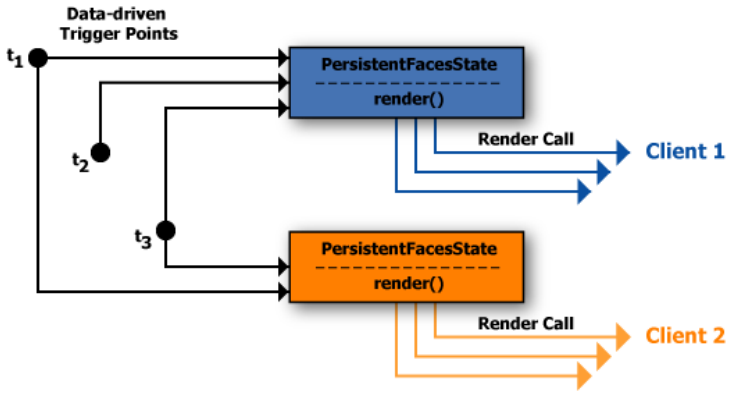
```
  state.render();
```

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.

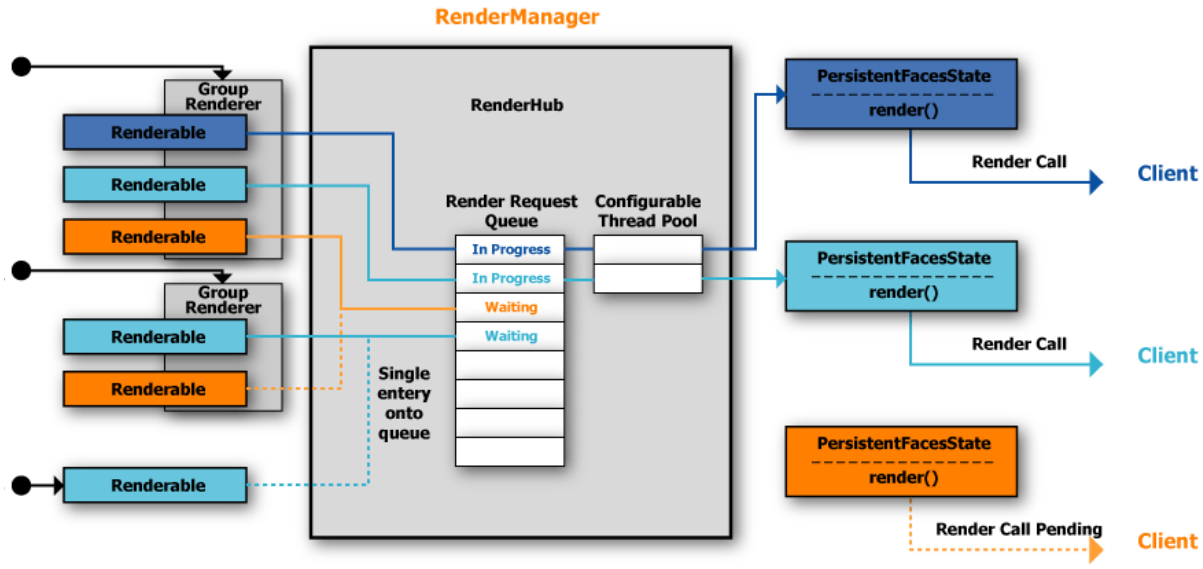
FatalRenderingException

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.

Event Type	Effect
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:

Effect	Description
scaleContent	Scale content.
puff	Grow and fade an element.
blindup	Remove an element by rolling it up.
blinddown	Show an element by rolling it down.
switchoff	Flash and then fold the element, removing it from the display.
dropout	Move the element down and fade, removing it from the display.
shake	Shake an element from left to right.
slidedown	Slide an element down from the top.
slideup	Slide an element up to the top, removing it from the display.
squish	Squish an element off the display.
grow	Grow an element from hidden to its normal size.
shrink	Shrink an element off the display.
fold	Fold an element into smaller pieces, until it is removed from the display.

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.

Developing Portlets with ICEfaces

ICEfaces provides the same benefits to portlets as it does to web applications. This section describes how to use ICEfaces for portlet development.

ICEfaces Portlet Configuration

The current specification for enterprise Java portlets is JSR 168, which details the portlet-specific APIs that are supported by the portal container as well as the configuration artifacts that are required. In addition, vendors implementing JSR 168 typically have their own custom configuration options. The instructions in this section outline both the general and vendor-specific requirements for developing and deploying ICEfaces-powered portlets.

JSF Portlet Bridge

Because ICEfaces is an extension to JavaServer Faces (JSF), the typical way to run standard JSF-powered portlets in a compliant portal is to use a bridge. There are currently several different bridges to choose from as well as a group working to standardize an interface for JSF portlet bridges to follow (JSR 301).

The problem with the current bridge implementations is that they vary in how they adapt to the JSF implementation as well as the portal implementations they work with. This makes the portability of JSF portlets a challenge. Each bridge also hooks into the JSF implementation in ways that are currently incompatible with ICEfaces. If you are developing ICEfaces portlets, do not include any JSF portlet bridge library.

The portlet.xml

With portlets, the main configuration document is the portlet.xml file where one or more portlets are declared along with their specific configuration information. For a comprehensive discussion of the contents of the portlet.xml file, refer to the portlet specification. For developing ICEfaces applications, you only need to be concerned with a couple of important settings: <portlet-class> and <init-param>.

> **Note:** In the following example, boldface text is used only to call attention to the relevant settings in the code example.

The following is a sample code snippet of a portlet declaration with the specific ICEfaces information added:

```
...
<portlet>
  <portlet-name>toggleMessage</portlet-name>
  <display-name>Toggle Message</display-name>
  <portlet-class>
     com.icesoft.faces.webapp.http.portlet.MainPortlet
  </portlet-class>
  <init-param>
    <name>com.icesoft.faces.VIEW</name>
    <value>/toggle.iface</value>
  </init-param>
</portlet>
...
```

The <portlet class> is the fully qualified class name of the class that implements the Portlet interface. ICEfaces has its own portlet, which could be considered a bridge of sorts, that handles incoming calls from the portal container and passes them on to the ICEfaces framework. You must use this portlet or a subclass of this portlet as the <portlet-class> value.

The <init-param> setting uses a key of com.icesoft.faces.VIEW. The value of this parameter is the initial view that the portlet displays on the initial render pass. It is important to use the .iface extension to ensure that the request is properly handled by the ICEfaces framework.

The <ice:portlet> Component

ICEfaces provides a portlet component that you should use to wrap around the entire content of your portlet. It is implemented as a NamingContainer so that it can apply the portlet namespace as the top level of the JSF ID hierarchy. Doing this makes the ID hierarchy more efficient and helps the ICEfaces framework uniquely identify components on the page, which is important when more than one ICEfaces portlet is running.

To use the portlet component, add it as the top-level component of your content. For example, the following is an ICEfaces page that toggles a message on and off. You can see that the portlet component is wrapped around the actual content of what we want to see in the portlet.

```
<f:view xmlns:h="http://java.sun.com/jsf/html"
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns:ice="http://www.icesoft.com/icefaces/component">

  <ice:portlet>
    <ice:form>
      <ice:panelGrid columns="2">
        <ice:outputText value="Message:" />
        <ice:outputText value="#{test.message}" />
        <ice:commandButton value="Toggle" actionListener="#{test.toggle}"/>
      </ice:panelGrid>
    </ice:form>
  </ice:portlet>

</f:view>
```

The ICEfaces portlet component is designed to behave unobtrusively in a regular web application. It is fairly common practice for developers to run their portlets as a web application to speed development and to check for portlet specific issues. In a portal container, the <ice:portlet> is rendered as a element with an ID attribute set to the portlet instance's unique namespace. In a web application, the

portlet namespace is not available so the component simply renders out as a with an ID attribute that is automatically generated by the JSF framework. Because it is a NamingContainer, the ID of the <ice:portlet> component will be prefixed to the client ID of the nested sub-components.

Setting com.icesoft.faces.concurrentDOMViews

This guide documents a context parameter called, com.icesoft.faces.concurrentDOMViews. It is set in the **web.xml** file as follows:

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

In a normal web application, setting this to true indicates that ICEfaces should support multiple views for a single web application and tells the ICEfaces framework to treat each view separately. Typically this is enabled when you want to use multiple windows of a single browser instance to concurrently view an ICEfaces application. In a portlet environment, the framework needs to treat the separate portlets on a single portal page as distinct views so it is almost always necessary (and therefore safest) to have this parameter set to true.

Using the Portlet API

Configuring your portlet is half the battle but, as a developer, you'll likely want to access the Portlet API to do a few things.

ExternalContext

The JSF API was designed to support access to both the Servlet API (for web applications) as well as the Portlet API by exposing an abstract class called the ExternalContext. In your code, you get access to the ExternalContext as follows:

```
FacesContext facesContext = FacesContext.getCurrentInstance();
ExternalContext externalContext = facesContext.getExternalContext();
```

Attributes

Once you have a reference to the ExternalContext, you can access information from the Portlet API. The ExternalContext API provides methods to get information in a way that is independent of the environment that it is running in. For example, to access request attributes, you can do the following:

```
Map requestMap = ec.getRequestMap();
String uri = (String)requestMap.get("javax.servlet.include.request_uri");
```

The requestMap contains all the attributes associated with the request. Check the ExternalContext JavaDoc to see what is provided by the rest of the API as some methods state specifically what they do differently in a portlet environment as compared to a servlet environment.

PortletConfig

If you need to access the PortletConfig, you can use the requestMap. From there you can retrieve information specific to the current portlet's configuration:

```
PortletConfig pc = (PortletConfig)requestMap.get("javax.portlet.config");
String portletName = portletConfig.getPortletName();
String view = portletConfig.getInitParameter("com.icesoft.faces.VIEW");
```

PortletRequest, PortletResponse

You can directly access copies of the PortletRequest and PortletResponse objects using the ExternalContext:

```
PortletRequest pReq = (PortletRequest)ec.getRequest();
PortletResponse pRes = (PortletResponse)ec.getResponse();
```

PortletPreferences

Once you have the PortletRequest, you can access the PortletPreferences which can be used to get and set preferences:

```
PortletPreferences prefs = portletReq.getPreferences();
try {
    prefs.setValue("prefKey", "prefVal");
    prefs.store();
} catch (ReadOnlyException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} catch (ValidatorException e) {
    e.printStackTrace();
```

Portlet Styles

The JSR 168 specification also documents a base set of common styles that should be applied to specific page elements. By documenting these style names, portlets can be developed that adhere to a portal container's overall “theme-ing” strategy. For example, the style portlet-form-button is a style name that is used to determine the style of the text on a button. Where appropriate, ICEfaces automatically renders out the proper style names for its components. However, since the portlet specification does not cover every rich component that is offered by ICEfaces (e.g., calendar), it is possible that some components may not match the current theme of the portal page.

Additionally, ICEfaces provides themes of its own (see **Styling the ICEfaces Component Suite** on page 29). These stylesheets were designed with web applications in mind. For portlet developers, ICEfaces also provides a portlet-friendly version of the xp.css stylesheet. The xp-portlet.css stylesheet can be easily added to your portlet by including the following component.

```
<ice:outputStyle href="/xmlhttp/css/xp/xp-portlet.css" />
```

Supported Portal Implementations

ICEfaces currently supports the following portal implementations:

- **Liferay Portal**
- **JBoss Portal**
- **WebLogic Portal**
- **Apache Jetspeed-2**
- **Apache Pluto**

The process of developing and deploying portlets to the various portal containers is vendor-specific so you should be familiar with the platform you are using. If you have questions, please consult the portal vendor's documentation for additional help. Issues that are specific to ICEfaces running on a specific portal implementation are covered in the following sections.

Liferay Portal

Liferay uses JavaScript to enhance both the developer and user experience. Portlet users can drag and drop portlets on the page without requiring a full page refresh. For the developer, Liferay makes it easy to hot deploy and load portlets dynamically, which can be a big advantage in speeding up the development cycle. However, deploying ICEfaces portlets in this manner can be problematic because the JavaScript that ICEfaces relies on may not get executed properly. Since the portal container is in control of the page, the ICEfaces JavaScript bridge (the client portion of the ICEfaces framework) is not the only JavaScript code on the page. Once the portal page has been loaded, the bridge's window.onload() logic won't get executed unless there is a full page refresh.

Fortunately, Liferay provides configuration parameters that allow the developer to specify that a full render pass is required. Doing this ensures that the ICEfaces bridge is properly initiated. The required parameters, render-weight and ajaxable, are specified in the liferay-portlet.xml configuration file. They are added to the portlet section as shown in this snippet:

```
<portlet>
  <portlet-name>clock</portlet-name>
  <instanceable>true</instanceable>
  <render-weight>1</render-weight>
  <ajaxable>false</ajaxable>
</portlet>
```

> **Note:** In the above example, boldface text is used only to call attention to the relevant parameters in the code example.

By setting these parameters, you ensure that a full-page refresh is done when the portlet is added to the portal page which, in turn, ensures that the ICEfaces JavaScript bridge is initialized correctly.

JBoss Portal

No specific parameters are required to run ICEfaces portlets in JBoss Portal.

WebLogic Portal

No specific parameters are required to run ICEfaces portlets in WebLogic Portal.

Apache Jetspeed-2

There is a browser-specific issue when running ICEfaces portals on Jetspeed-2 which is related to parsing cookie paths. The problem prevents Firefox from working properly. To solve it, you should modify **`[jetspeed-root]/conf/server.xml`** and add the emptySessionPath="true" attribute to the <Connector> element. It should look something like this:

```
<Connector port="8080" maxHttpHeaderSize="8192" maxThreads="150"
           minSpareThreads="25" maxSpareThreads="75" enableLookups="false"
           redirectPort="8443" acceptCount="100" connectionTimeout="20000"
           disableUploadTimeout="true" emptySessionPath="true"/>
```

This attribute is set by default in other Tomcat-based portals like JBoss Portal and Apache Pluto.

Apache Pluto

No specific parameters are required to run ICEfaces portlets in Apache Pluto.

Using AJAX Push in Portlets

By default, AJAX Push is active in ICEfaces. The configuration parameter that controls this is com.icesoft.faces.synchronousUpdate. If you don't need server-initiated rendering, then you should set this parameter to false by adding the following to your **web.xml** file:

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

If you are using AJAX Push then you can set the parameter to true or just leave it out altogether.

Development and Deployment Considerations

To maximize the successful development and deployment of ICEfaces portlets, consider these tips and suggestions:

Mixing Portlet Technologies

The sheer size and complexity of the matrix of portlet technology combinations makes it difficult to test and/or document every combination. While it is possible to deploy ICEfaces portlets on the same portal page as portlets built without ICEfaces, you should consider the following sections.

Static Portlets

We define standard portlets as portlets that are built using more traditional technologies and without AJAX. These portlets follow the usual portlet lifecycle in that, when an action is taken on one portlet, a render is performed on all of the portlets on the page and a full-page refresh is triggered. If you mix ICEfaces portlets on a portal page with these traditional portlets, you'll get the behavior of the lowest common denominator. What this means is that if you interact with a static portlet, the regular lifecycle is engaged and all the portlets will re-render which triggers a page refresh. This behavior can undermine the benefits of using ICEfaces in your portlet development. If possible, you should consider porting the static portlets to use ICEfaces.

Dynamic Portlets

Several web development products, libraries, and component suites allow you to build rich, interactive portlets. Other products that use AJAX techniques (client-side, server-side, JSF, etc.) will have their own JavaScript libraries. JSF-based solutions can hook into the implementation in incompatible ways.

While running portlets built with these other technologies may work, there is a definite possibility of conflict with ICEfaces. While ICEfaces strives to co-exist with these other offerings, you can increase your chances of successfully building your project by reducing the complexity of your architecture. This can mean going with a single technology throughout or perhaps constraining a single portal page to a single technology.

ICEfaces With and Without AJAX Push

It is possible to combine synchronous and asynchronous ICEfaces portlets on a single portal page without any kind of additional configuration, as long as the asynchronous ICEfaces portlets are deployed from the same .war file. However, if the asynchronous ICEfaces portlets running on a single page are deployed in different .war files, you will need to configure the Asynchronous HTTP Server on the portal as well. We recommend deploying it as a servlet in this case to minimize the additional configuration required. Each .war file that contains one or more asynchronous ICEfaces portlet needs to be configured to use the Asynchronous HTTP Server. Refer to **Deployment Configurations** on page 61 for more information on the various options. The required configuration depends on the portal implementation being used.

> **Note:** All ICEfaces portlets on the same portal page are required to use the same version of ICEfaces.

Portlet Deployment – Same or Separate Archive

It is possible to deploy individual ICEfaces portlets in separate web archives (.war files) as well as bundling several ICEfaces portlets into a single archive. There are considerations for each. If multiple ICEfaces portlets are bundled together in a single archive, they gain the ability to share some common state and, using AJAX Push, benefit from a form of inter-portlet communication. However, these portlets also share a single set of configuration files (web.xml, portlet.xml, etc.) so it's important to understand the implications.

Shared faces-config.xml

The faces-config.xml file is the JSF configuration file used to describe managed beans, navigation, and other JSF-related features. Deploying multiple JSF portlets in a single archive means that all the portlets will share the same configuration. However, each portlet will, in essence get a copy of that configuration. This is an important distinction for the scope of managed beans and bean inter-dependencies.

Application-scoped beans are visible across all the portlets in the archive (as you would expect) and this can be useful if all the portlets need to share some common state. Application-scoped beans, together with AJAX Push, can be used to do a form of inter-portlet communication.

Session-scoped beans are scoped to the portlet session (rather than the user session) which means that each portlet gets its own instance. You can store information in the global user session so that it is available to all the portlets in a single user's session by using the Portal API and specifying the application scope.

Request-scoped beans are scoped to the requests for the individual portlet. Since ICEfaces directly handles all its own AJAX traffic directly, bypassing the portal container, it can be easy to make incorrect assumptions about the number of beans being created and the relationship between them.

For example, consider the following set of managed JSF beans:

```
<managed-bean>
    <managed-bean-name>app</managed-bean-name>
    <managed-bean-class>com.icesoft.example.AppBean</managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
</managed-bean>

<managed-bean>
    <managed-bean-name>sess</managed-bean-name>
    <managed-bean-class>com.icesoft.example.SessionBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
        <property-name>appBean</property-name>
        <value>#{app}</value>
    </managed-property>
</managed-bean>

<managed-bean>
    <managed-bean-name>req</managed-bean-name>
    <managed-bean-class>com.icesoft.example.RequestBean</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
    <managed-property>
        <property-name>sessionBean</property-name>
        <value>#{sess}</value>
    </managed-property>
</managed-bean>
```

Now let's assume we have two portlets, A and B, declared in the .war file and actively deployed on the page. The application-scoped bean is only created once and that single instance is referenced by both portlets. A single, separate session bean is created for each portlet. A separate request bean instance for each portlet is created for each request. Confusion occurs if the developer assumes that the session and request beans for portlet A can reference or see the session and request beans for portlet B. Even though we have a single faces-config.xml file, each portlet essentially uses its own copy of the

configuration and other than the application-scoped beans, does not share any reference to each other's beans.

If your portlets don't need to share state for any meaningful reason, then it's probably better to deploy them in separate archives. If, for some other reason, they are deployed in the same archive, take care to manage the JSF configuration.

Inter-Portlet Communication (IPC)

The use of AJAX Push allows portlets to be updated based on server-side events that change the state of the current view. This can be a powerful feature that can also be leveraged to do a form of inter-portlet communication (IPC) in certain configurations.

IPC is only mentioned in the Portlet 1.0 spec (JSR 168) but is formally defined in the Portlet 2.0 specification (JSR 286). It is architected as an Event/Listener model. However, it is possible to use ICEfaces' AJAX Push mechanism to update portlets based on changes to the underlying model.

The way to do this currently with ICEfaces is to:

- Deploy the portlets that need to communicate in the same archive (.war file).
- Use application-scoped beans to manage shared state between the portlets.
- Use ICEfaces' AJAX Push feature to trigger client updates when the shared state changes.

For an example of how to do this, review the sample ICEfaces Chat portlet.

ActionRequest and ActionResponse

The standard way for a portlet to change the state of its underlying model is via an ActionRequest. The portlet developer generates interactive controls like buttons and links using the Portlet API. When the user of the portlet interacts with those controls, the portal container can interpret the incoming request as an ActionRequest and process it accordingly.

Currently, all interactive types of actions are handled by the ICEfaces framework directly and are therefore never handled by the portal container. In other words, ActionRequests are never really issued. While this is transparent to the developer, it does lead to certain restrictions. The API for the ActionResponse includes methods for setting the mode of the portlet as well as the state of the portlet window programmatically. Because ICEfaces bypasses the portal container for these types of requests, the ActionRequest and ActionResponse instances are not available to the portlet developer.

A future release of ICEfaces may deal with this situation but, for now, this is how it impacts the portlet developer.

Portlet Modes and Navigation

The portlet specification defines some standard modes that a portlet can support (VIEW, EDIT, HELP) and allows portal developers and/or portlet developers to potentially add their own. The impact of not being able to change portlet modes programmatically is that it becomes possible to have JSF navigation rules change the current view to a known mode without the portal container knowing about it. For example, you could allow a user to click a button that takes them from the EDIT mode back to the VIEW mode via JSF navigation rules. But because ICEfaces does this without going through the portal

container, the portal container does not know the mode has changed and since the portlet developer cannot access the ActionResponse API, there is no way to let it know.

Some suggestions for dealing with this are:

- Portlet windows typically provide a title bar with icons for switching between supported modes. The icons issue requests to the portal container to switch modes. If possible, use the icons rather than JSF navigation to switch nodes.
- ICEfaces is designed to provide a rich UI experience. It is often possible to design an application or portlet to avoid navigation altogether using other user interface techniques liked tabbed panels to present different views of the data. Consider modifying your interface to take advantage of this.

Window States

Portlet windows can also be in various states (e.g., MINIMIZED). As with portlet modes, we recommend using the title bar icons to control these states rather than trying to adjust them programmatically.

Request Attributes

A unique aspect of the ICEfaces framework is that, due to the use of AJAX techniques, requests can be “long-lived”— somewhere between request and session scope. To maintain the long-lived nature of these types of requests, the ICEfaces framework needs to maintain request-based information during AJAX communications. To maintain this data, ICEfaces makes use of the standard servlet and portlet APIs to get and store things like attributes in its own internal structures. The PortletRequest API provides a couple of different ways to retrieve request attributes:

```
java.lang.Object getAttribute(java.lang.String name)
java.util.Enumeration getAttributeNames()
```

Unfortunately, there is an issue with certain portal containers where the getAttributeNames() method does not return the same set of attributes that can be retrieved with calls to getAttributeName(String name). In order for ICEfaces to ensure that all the required attributes are maintained for all AJAX requests, some request attributes need to be formally specified by the developer so that ICEfaces can copy them.

The ICEfaces framework maintains all the attributes that are specified in the JSR 168 specification (javax.servlet.include.*, javax.portlet.*). For attributes that are specific to the portal container, ICEfaces provides a mechanism for the developer to add them. Custom request attributes can be specified as space separated strings in 'com.icesoft.faces.hiddenPortletAttributes' context parameter. For example, to add Liferay's custom THEME_DISPLAY attribute so that it is properly maintained during AJAX requests, you would have the following context parameter set in your **web.xml** file:

```
<context-param>
  <param-name>com.icesoft.faces.hiddenPortletAttributes</param-name>
  <param-value>THEME_DISPLAY</param-value>
</context-param>
```

As noted, additional attributes can be added to the context parameter as long as they are separated by spaces.

Custom Solutions

If the portal vendor provides a client-side or server-side mechanism for handling these scenarios, then you have the option of using those. However, committing to these APIs probably means giving up the ability to run the portlet as a plain web application or running the portlet on a different portal platform. You can decide whether the trade-off is acceptable.

Running the ICEfaces Sample Portlets

In the binary installation of ICEfaces, pre-built sample applications and their sources can be found at:

```
[install_dir]/ICEfaces-1.7.0-bin/icefaces/samples/
```

Samples that have been specifically developed as portlets can be found at:

```
[install_dir]/ICEfaces-1.7.0-bin/icefaces/samples/portlet/
```

If you downloaded the source code distribution of ICEfaces, then you can build all the samples by running:

```
ant
```

from the following directory:

```
[install_dir]/ICEfaces-1.7.0-src/icefaces/
```

Currently, the default build of the portlet samples is Liferay running on Tomcat 5.5. Targets for additional portal/platform combinations are provided. To see a list of portals, run:

```
ant -p
```

Deploying portlet .war files to a supported portal container varies from vendor to vendor. Please check the documentation for your platform on how to deploy the portlet archive.

Component Showcase

The Component Showcase application is a useful way to see ICEfaces components in action as well as providing useful coding examples. We also provide a build of Component Showcase that can be deployed in a portal container. Each individual component demo has been configured as a separate portlet that can be added and removed from a portal page. The demo can be used to illustrate how to configure ICEfaces portlets as well as demonstrate working examples of the ICEfaces components running inside a portal.

Chat

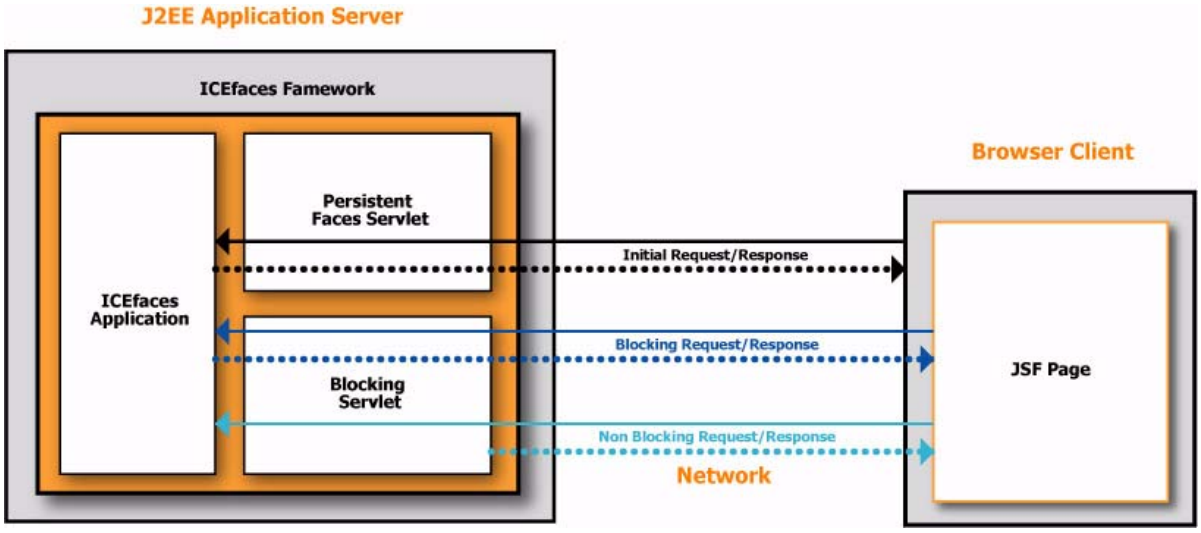
The ICEfaces Chat Portlet example is a very simple application that shows how to use the AJAX Push features of ICEfaces to update multiple portlets on a page. In this case, you can open two separate instances of the chat portlet on a single page and chat between them, watching them both update as messages are sent.

Introduction to the Asynchronous HTTP Server

The Asynchronous HTTP Server (AHS) is an HTTP server capable of handling long-lived asynchronous XMLHttpRequests in a scalable fashion. In the standard Servlet model, each outstanding asynchronous XMLHttpRequest occupies its own thread, which means that thread requirements can grow linearly with the number of clients. The long-livedness of these requests will have significant impact on thread-level scalability of the application. If server-initiated rendering occurs frequently, thread consumption will be somewhat mitigated, but if server-initiated rendering occurs infrequently, large numbers of users will cause excessive thread consumption, and the scalability of the application will be compromised. It is strongly recommended that the Asynchronous HTTP Server be used in all ICEfaces application deployments where asynchronous mode is used and where the used application server is not capable of handling long-lived asynchronous XMLHttpRequests in a scalable fashion.

Before discussing the architecture of an Asynchronous HTTP Server deployment, it is useful to review the communication pattern that ICEfaces uses in Asynchronous mode. Asynchronous updates are primed with a blocking request from the client. When a render occurs in the JSF lifecycle, a response is produced and the blocking request is satisfied. When the response is received at the client, another blocking request is generated. ICEfaces also uses non-blocking requests for user-initiated interactions via the full or partial submit mechanism. Non-blocking requests are passed to the ICEfaces application. In ICEfaces, the Blocking Servlet handles all of this communication. See Figure 14.

Figure 14 Asynchronous Mode Deployment Architecture



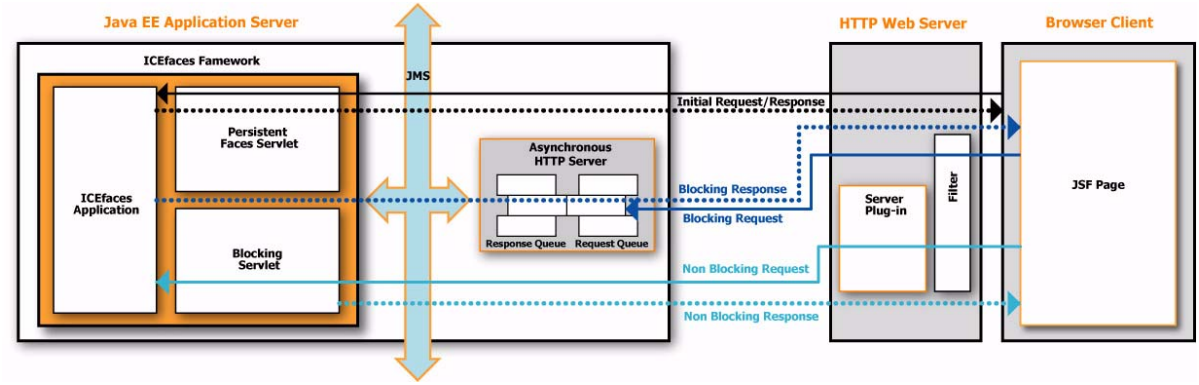
The Asynchronous HTTP Server handles thread-level scalability issues. It is designed to support single as well as clustered application server deployments with one or more ICEfaces applications deployed on it. The Asynchronous HTTP Server can be deployed in two ways: either as a server or as a servlet.

When the Asynchronous HTTP Server is deployed as a server, as a separate application, in the application server, it occupies a single port on the network. Individual ICEfaces application deployments use Java Messaging Service (JMS) to communicate with the Asynchronous HTTP Server. Blocking requests are queued at the Asynchronous HTTP Server on a per client basis, and as responses become available, they are communicated to the Asynchronous HTTP Server, again via JMS. A response is matched to its associated blocking request, the request is unblocked, and the response is returned to

the client. Because it is possible for a response to be generated prior to a request arriving, the Asynchronous HTTP Server also queues up responses waiting for requests that it can satisfy.

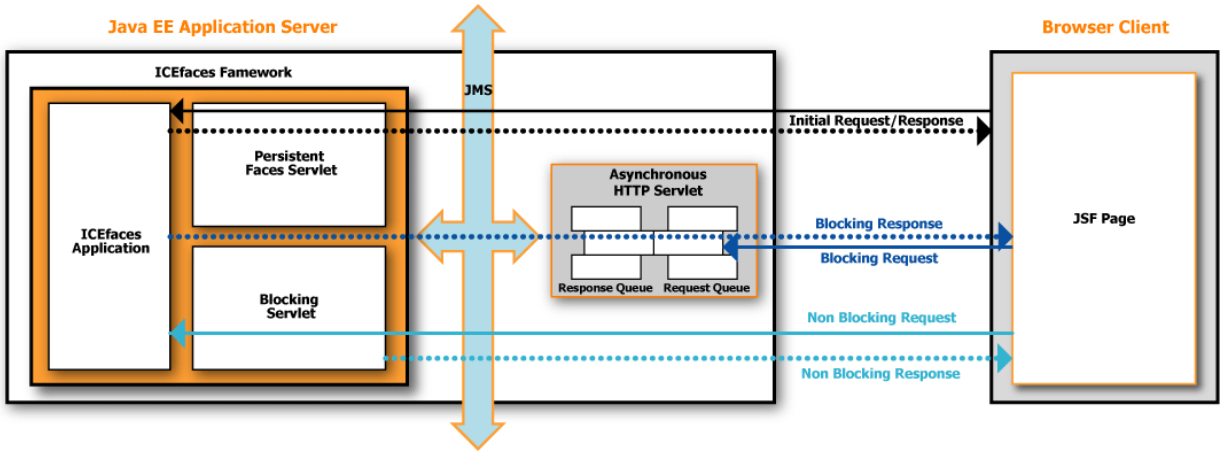
The deployment architecture is fronted with an HTTP web server that filters blocking request and directs them to the Asynchronous HTTP Server for processing. Non-blocking requests are passed directly to the ICEfaces application possibly via an application server-specific plug-in. The web server can also be configured to do user authentication and to use SSL for security. This basic ICEfaces enterprise deployment infrastructure, including the Asynchronous HTTP Server running as a server and the front-end web server, is illustrated in Figure 15.

Figure 15 ICEfaces Enterprise Deployment Architecture



When the Asynchronous HTTP Server is deployed as a servlet in the application server, it does not occupy a port on the network. It basically functions the same as it would have been deployed as a server, but the deployment architecture does not need to be fronted with an HTTP web server in this particular case. This basic ICEfaces enterprise deployment infrastructure, including the Asynchronous HTTP Server running as a servlet, is illustrated in Figure 16.

Figure 16 ICEfaces Enterprise Deployment Infrastructure



Deployment Configurations

This section describes the deployment configuration that must occur to support enterprise deployments of ICEfaces applications with the Asynchronous HTTP Server. The basic steps include:

1. **Configuring the Asynchronous HTTP Server**
2. **Configuring the ICEfaces Application**
3. **Configuring the Web Server**
4. **Configuring the Application Server**

Configuring the Asynchronous HTTP Server

Deploying the Asynchronous HTTP Server

The Asynchronous HTTP Server is an application running in an application server communicating with the ICEfaces applications on the back-end and the clients on the front-end. It can run either as a servlet or as a server. A default deployment .war file, async-http-server.war, for the Asynchronous HTTP Server is available in the `[icefaces-home]/ahs/dist` directory, assuming the .war file is build. This deployment is configured to run as a servlet by default. This deployment is configured to run as a servlet by default. To configure it as a server, set the com.icesoft.faces.async.service context parameter to "server", instead of "servlet", in the **web.xml** as follows:

```
<context-param>
    <param-name>com.icesoft.faces.async.service</param-name>
    <param-value>server</param-value>
</context-param>
```

> **Note:** The old **com.icesoft.faces.async.server** boolean parameter still works as well, but will be deprecated in a future release.

When running as a server, the deployment provides the following defaults:

- port used: 51315
- non-blocking mode (using Java's New I/O)
- persistent connection mode (using persistent HTTP connections)
- compression mode (using HTTP compression)
- execute queue size: 30 (the number of threads used to handle incoming requests)

Should you need to change these defaults, the configuration can be overridden by modifying the **web.xml** file and repackaging the .war.

The relevant attributes are as follows.

```
<servlet>
    <servlet-name>Asynchronous HTTP Servlet</servlet-name>
    ...
    <init-param>
        <param-name>com.icesoft.faces.async.server.port</param-name>
        <param-value>51315</param-value><!-- integer -->
    </init-param>
    <init-param>
        <param-name>com.icesoft.faces.async.server.blocking</param-name>
        <param-value>false</param-value><!-- boolean -->
    </init-param>
    <init-param>
        <param-name>com.icesoft.faces.async.server.persistent</param-name>
        <param-value>true</param-value><!-- boolean -->
    </init-param>
    <init-param>
        <param-name>com.icesoft.faces.async.server.compression</param-name>
        <param-value>true</param-value><!-- boolean -->
    </init-param>
    <init-param>
        <param-name>
            com.icesoft.faces.async.server.executeQueueSize
        </param-name>
        <param-value>30</param-value><!-- integer -->
    </init-param>
</servlet>
```

Note: It is critical that the port number matches the port number in the Apache HTTP Server's configuration file, which is discussed in **Routing Requests**, *p. 65*.

Note: The Asynchronous HTTP Servlet is always required, even if running as a server. This servlet is also responsible for starting up the server inside the application server.

Configuring the JMS Provider

Depending on the application server of choice, the async-http-server.war needs to be configured to use the proper JMS Provider with the correct JMS settings. By default it uses the jboss.properties file which contains the configuration for a non-clustered JBoss deployment. The Asynchronous HTTP Server comes with a number of pre-configured property files described in Table 6.

Table 6 ICEfaces AHS Configuration Property Files

Application Server	Configuration File	Usage
ActiveMQ	activemq.properties	Non-clustered deployment
JBoss	jboss.properties	Non-clustered deployment
	jboss_ha.properties	Clustered deployment using the JBoss HA (High Availability) JMS
WebLogic	weblogic.properties	(Non-)clustered deployment
WebSphere	websphere.properties	Non-clustered deployment
	websphere_ha.properties	Clustered deployment using WebSphere's messaging via the SIB (Service Integration Bus) on a cluster

To specify the JMS Configuration properties, modify the following context parameter in the **web.xml** and repackage the async-http-server.war:

```
<context-param>
    <param-name>com.icesoft.net.messaging.properties</param-name>
    <!-- properties file of choice -->
    <param-value>weblogic.properties</param-value>
</context-param>
```

If the supplied properties files do not suffice, a custom properties file can be created and added to the classpath. A JMS Provider Configuration properties file can contain the following properties:

```
java.naming.factory.initial
java.naming.factory.url.pkgs
java.naming.provider.url

com.icesoft.net.messaging.jms.topicConnectionFactoryNam
com.icesoft.net.messaging.jms.topicNamePrefix
```

Refer to the documentation for your application server for these property values. The following example shows the property values for JBoss:

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
java.naming.provider.url=localhost:1099

com.icesoft.net.messaging.jms.topicConnectionFactoryName=ConnectionFactory
com.icesoft.net.messaging.jms.topicNamePrefix=topic/
```

Configuring the ICEfaces Application

Using the Asynchronous HTTP Server with an ICEfaces application requires the following configuration changes:

1. Indicate that the Asynchronous HTTP Server is used by adding the following code to ICEfaces application's **web.xml** file:

```
<context-param>
    <param-name>com.icesoft.faces.blockingRequestHandler</param-name>
    <param-value>icefaces-ahs</param-value>
</context-param>
```

> **Note:** The old **com.icesoft.faces.async.server** boolean parameter still works as well, but will be deprecated in a future release.

2. If the Asynchronous HTTP Server will be running as a servlet, the following context parameter needs to be added to the ICEfaces application's **web.xml** file as well:

```
<context-param>
     <param-name>com.icesoft.faces.blockingRequestHandlerContext</param-name>
     <param-value>async-http-server</param-value>
</context-param>
```

> **Note:** The old **com.icesoft.faces.asyncServerContext** string parameter still works as well, but will be deprecated in a future release.

3. Just as the Asynchronous HTTP Server (AHS) needs to be configured for your specific application server, an ICEfaces application needs to be configured as well. Refer to **Configuring the JMS Provider**, *p. 63*, for details on configuring the application to use the JMS Provider.
4. Add the following JAR to the web application build (if not already present):
   - icefaces-ahs.jar

Configuring the Web Server

This section elaborates on the ICEsoft recommended configurations for running ICEfaces applications on application servers with a web server as the front-end.

If the Asynchronous HTTP Server is configured to run as a server, specific configuration of the web server is required and explained in this section. The Asynchronous HTTP Server is spawned inside the application server of choice. The web server is therefore responsible for filtering the incoming HTTP requests and forwarding the blocking requests to the Asynchronous HTTP Server and all other requests to the application server. A configuration such as this is mandatory to have full support for all mainstream Internet browsers. Refer to Figure 2, *p. 2*.

If the Asynchronous HTTP Server is configured to run as a servlet, no specific configuration of a web server is required and this section can therefore be skipped. If a web server as the front-end is desired,

it can be configured normally as it would front-end any web application(s) running on a back-end application server.

> **Note:** In the code examples throughout this section, we use boldface text for variables that should be replaced as required.

Apache HTTP Server

Depending on the deployment, either Apache HTTP Server 2.0.x or 2.2.x can be used as the front-end. When the Asynchronous HTTP Server is to be deployed to a single application server or to a single node in a cluster of application servers, both versions of Apache can be used. However, if the Asynchronous HTTP Server is to be deployed to multiple nodes in a cluster of application servers, Apache 2.2.x is to be used.

Routing Requests

To route all blocking requests to the Asynchronous HTTP Server and all other requests to the application server, add the following code to the end of the Apache configuration file.

> **Note:** The VirtualHost container is not mandatory for Apache 2.0.x; when omitted, its directives, such as ServerAdmin, need to be omitted as well.

```
<VirtualHost _default_>
   ServerAdmin webmaster@host.example.com
   DocumentRoot /var/www/html/host.example.com
   ServerName host.example.com
   ErrorLog logs/host.example.com-error_log
   TransferLog logs/host.example.com-access_log

   <IfModule mod_proxy.c>
     ProxyRequests Off

     <Proxy *>
        Order deny,allow
        Allow from all
     </Proxy>

     # The following two directives will route all blocking requests to the
     # Asynchronous HTTP Server (identified by host:port).
     ProxyPass  /application-name/block/receive-updated-views
                http://host:port/application-name/block/receive-updated-views

     ProxyPassReverse /application-name/block/receive-updated-views
                      http://host:port/application-name/block/receive-updated-views

     # The following two directives will route all other requests to the
     # application server (identified by host:port).
     ProxyPass     /application-name
                   http://host:port/application-name
```

```
    ProxyPassReverse /application-name
                     http://host:port/application-name
  </IfModule>
</VirtualHost>
```

> **Note:** The previous ProxyPass and ProxyPassReverse directives must appear on a single line in your configuration file.

If multiple ICEfaces applications are deployed behind the Asynchronous HTTP Server, multiple ProxyPass and ProxyPassReverse directives should be included, two sets for each application. If an Apache HTTP Server plug-in is used, which is discussed in **Apache HTTP Server**, *p. 65*, the last ProxyPass and ProxyPassReverse set can be omitted.

> **Note:** It is critical that the port number, mentioned in the first ProxyPass and ProxyPassReverse set, matches the port number in the Asynchronous HTTP Server's configuration file, which is discussed in **Deploying the Asynchronous HTTP Server**, *p. 61*. Evidently the port number, mentioned in the second ProxyPass and ProxyPassReverse set, should match the port number of the application server.

To have proxy support in the Apache HTTP Server, the mod_proxy and mod_proxy_http modules are required. In the Apache configuration file where all modules are being loaded, add the following if not already added:

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
```

For more information on how to load modules into Apache HTTP Server, refer to Apache Module mod_so which can be found at:

http://httpd.apache.org/docs/2.0/mod/mod_so.html (Apache 2.0.x)

http://httpd.apache.org/docs/2.2/mod/mod_so.html (Apache 2.2.x)

Security Considerations

Authentication

To enforce authentication of the user when accessing an ICEfaces application, add the following code to the Apache configuration file before the added IfModule container:

```
<LocationMatch ^/application-name>
    AuthName "ICEfaces Member-Only Access"
    AuthType Basic
    AuthUserFile /var/www/secrets/.members
    require valid-user
</LocationMatch>
```

The <LocationMatch regex> container determines that every location (Request-URI), which begins with (^) the literal string "/**application-name**" where the **application-name** is the name of the ICEfaces

application, is part of the ICEfaces Member-Only Access realm, and therefore, requires basic authentication for every user.

Secure Sockets Layer (SSL)

1. To enforce the usage of SSL when accessing an ICEfaces application, add the following code to the Apache configuration file before the added LocationMatch container:

   ```
   RewriteEngine On
   RewriteCond %{SERVER_PORT} ^80$
   RewriteCond %{REQUEST_URI} ^/application-name
   RewriteRule ^/(.*) https://%{HTTP_HOST}/$1 [R=301,L]
   ```

   The RewriteCond directives define the following conditions:

   - if the port address of the server (%{SERVER_PORT}) begins with (^) and ends with ($), thus exactly matches the literal string "80"; and
   - if the Request-URI (%{REQUEST_URI}) begins with (^) the literal string "/**application-name**", where the **application-name** is the name of the ICEfaces application.

   If both of the previous conditions are met, the RewriteRule directive defines how the Request-URI is rewritten to use HTTPS as follows:

   - $1 corresponds to the string found in the set of parentheses (that is, everything after the root (/)); and
   - rewrite the Request-URI to https://**%{HTTP_HOST}**/$1, where **%{HTTP_HOST}** is the server's host name.

   After rewriting the Request-URI, force an external redirect (301 Moved Permanently) to the client (R=301).

   Finally, tell the rewrite engine to end rule processing immediately (L), so that no other rules are applied to the last substituted Request-URI.

2. In order to have rewrite engine support, the mod_rewrite module is required. In the Apache configuration file where all modules are being loaded, add the following if not already added:

   ```
   LoadModule rewrite_module modules/mod_rewrite.so
   ```

Configuring the Application Server

When deploying applications using the Asynchronous HTTP Server, a J2EE compliant application server is required with support for JMS. The Asynchronous HTTP Server and the ICEfaces applications communicate over multiple JMS topics. It is necessary to configure the following JMS topics:

- icefaces.contextEventTopic
- icefaces.responseTopic

Application server-specific configurations are provided below.

Tomcat 5.x/6.x and ActiveMQ 5.x

Configuring Tomcat 5.x/6.x and ActiveMQ 5.x

An external message broker is required to use the Asynchronous HTTP Server on Tomcat because it is a servlet container which doesn't include an implementation of JMS. ActiveMQ is such a message broker that can be used next to the various application servers. This section discusses how to set up Tomcat 5.x/6.x using ActiveMQ 5.x for messaging.

There are various ways to set up Tomcat with ActiveMQ, but this section deals with each installed separately in their own folders, namely **`[tomcat-home]`** and **`[activemq-home]`**. If not already done so, ActiveMQ's **`[activemq-home]/activemq-all-5.x.x.jar`** needs to be copied to **`[tomcat-home]/shared/lib`**, in case of Tomcat 5.x, and to **`[tomcat-home]/lib`**, in case of Tomcat 6.x. Additionally, Tomcat 6.x requires Apache's Commons Logging archive to be copied to **`[tomcat-home]/lib`** as well.

The **activemq.properties** file is written to use the ActiveMQ's ability for dynamic topic creation. This ability will let you automatically create the desired topic if it is not yet created. This avoids manual configuration for the AHS' required topics on ActiveMQ.

Finally as this concludes the configuration of both Tomcat and ActiveMQ, the servers can be started. ActiveMQ can be started using **`[activemq-home]/bin/activemq`** on Linux or **`[activemq-home]/bin/activemq.bat`** on Windows; Tomcat can be started as desired.

Web Server Plug-ins

Plug-ins are modules that can be added to a web server installation and can be configured to enable interaction between the web server of choice and the application server of choice. Typically, plug-ins can be used as a load-balancer for the web server by proxying the requests to the back-end application servers, or can be used to proxy requests for dynamic content to the back-end server(s).

Apache HTTP Server 2.0.x

JBoss uses the Apache Tomcat Servlet container, which is configured with the Apache HTTP Server via the Tomcat plug-in known as **mod_jk**.

1. First, copy the **mod_jk.so** module supplied by the Apache Software Foundation into the Apache's **/module** directory. You can obtain this module from the following web site: **http://tomcat.apache.org/connectors-doc/howto/apache.html**
2. Create a new file called **mod_jk.conf** in the Apache HTTP Server's /conf directory and add the following code to it:

```
LoadModule jk_module modules/mod_jk.so

JkWorkersFile conf/workers.properties
JkLogFile /var/log/httpd/mod_jk.log
JkLogLevel info
JkMount /* myworker1
```

3. Create a new file called **workers.properties** in the same directory and add the following to it:

```
worker.list = myworker1
```

```
worker.myworker1.port = 8009
worker.myworker1.host = localhost
worker.myworker1.type = ajp13
worker.myworker1.lbfactor = 1
```

The **ajp13** refers to a worker inside Tomcat that listens to port 8009. This example shows both the Apache HTTP Server and Tomcat running on the same machine (hence, the localhost is the hostname of myworker1), but it is recommended to run the Apache HTTP Server and Tomcat instances on separate machines.

4. The **mod_jk.conf** needs to be loaded. To achieve this, add the following to Apache's configuration file right after the LoadModule directives:

   ```
   Include conf/mod_jk.conf
   ```

5. Finally, the ProxyPass and ProxyPassReverse directives that route all requests other than blocking requests to the application server, need to be removed from the Apache configuration file.

For more information on how to install and configure Apache Tomcat's plug-in, refer to the *Server Configuration Reference for Apache Tomcat 5.0.x* and *Apache Tomcat Configuration Reference for Apache Tomcat 5.5.x* which can be found at:

http://tomcat.apache.org/tomcat-5.0-doc/config/ajp.html

Apache HTTP Server 2.2.x

With the new Apache 2.2.x load-balance and failover capabilities the **mod_jk** became obsolete. However, for non-clustered deployments, it is still more desirable to make use of the AJP protocol. To do this without the **mod_jk** module, the mod_proxy_ajp module is required. In the Apache configuration file where all modules are being loaded, add the following if not already added:

```
LoadModule proxy_ajp_module modules/mod_proxy_ajp.so
```

To utilize the AJP protocol, the ProxyPass and ProxyPassReverse directives, which route all requests other than blocking requests to the application server, need to be rewritten to the following:

```
ProxyPass        /application-name
                 ajp://host:port/application-name
ProxyPassReverse /application-name
                 ajp://host:port/application-name
```

The default port for the AJP protocol is 8009.

Note: The previous ProxyPass and ProxyPass Reverse directives must appear on a single line in your configuration file.

JBoss 4.x

Configuring JMS for JBoss 4.x

When JBoss is started, three different configurations can be specified: default, all, or minimal. The minimal configuration is not sufficient for Asynchronous HTTP Server deployments, as it does not include JMS. The *default* configuration can be used in single node deployments, but the *all* configuration is required for clustered deployments. Table 7 below contains the location of the directory containing the jbossmq-destinations-service.xml file for each valid JBoss configuration.

Table 7 JBoss Configuration Files

Configuration	Directory
default	[jboss-install-dir]/server/default/deploy/jms/
all	[jboss-install-dir]/server/all/deploy-hasingleton/jms/

Note: All information on JBoss 4.x in this section is based on the compressed archive (zip or gz) and not the JBoss installer application.

To configure the JMS topics, add the following code to the selected jbossmq-destinations-service.xml file:

```
<mbean code="org.jboss.mq.server.jmx.Topic"
       name="jboss.mq.destination:service=Topic,name=icefaces.contextEventTopic">
    <depends optional-attribute-name="DestinationManager">
        jboss.mq:service=DestinationManager
    </depends>
    <depends optional-attribute-name="SecurityManager">
        jboss.mq:service=SecurityManager
    </depends>
    <attribute name="SecurityConf">
        <security>
            <role name="guest" read="true" write="true" create="true"/>
        </security>
    </attribute>
</mbean>
<mbean code="org.jboss.mq.server.jmx.Topic"
       name="jboss.mq.destination:service=Topic,name=icefaces.responseTopic">
    <depends optional-attribute-name="DestinationManager">
        jboss.mq:service=DestinationManager
    </depends>
    <depends optional-attribute-name="SecurityManager">
        jboss.mq:service=SecurityManager
    </depends>
    <attribute name="SecurityConf">
        <security>
            <role name="guest" read="true" write="true" create="true"/>
        </security>
    </attribute>
</mbean>
```

Web Server Plug-ins

For information about installing and configuring the Web Server Plug-ins for **Apache HTTP Server 2.0.x** or **2.2.x**, refer to **Web Server Plug-ins** on page 68.

Clustering

This section explains how to set up a clustered deployment of JBoss Application Servers. It includes steps for configuring the cluster, JMS and web server. The following discussion relates to a clean installation of JBoss, and represents a simplified process for configuring JBoss for clustered deployments of ICEfaces applications. Additional deployment-specific issues may exist. Refer to your JBoss documentation for additional information.

This example cluster consists of two servers (nodes) as shown in the table below.

IP Address	Server Name	Description
192.168.1.100	Node1	Server
192.168.1.101	Node2	Server

Configuring the Cluster

The JBoss cluster requires no special configuration.

Configuring JMS for the Cluster

The file **[jboss-install-dir]/server/all/deploy-hasingleton/jms/jbossmq-destinations-service.xml** on each node of the cluster must be modified to include these ICEfaces topics:

- icefaces.contextEventTopic
- icefaces.responseTopic

See **Configuring JMS for JBoss 4.x**, *p. 70* for specifics on adding these two topics.

Web Server Plug-Ins

On each node, the name of the node needs to be specified in the file `[jboss install dir]/server/all/deploy/jbossweb-tomcat5x.sar/server.xml`. In this file, locate the Engine container and add a jvmRoute attribute like the following:

```
<Engine name="jboss.web" defaultHost="localhost" jvmRoute="node1">
    ...
</Engine>
```

Apache HTTP Server 2.0.x

1. The contents of the file **mod_jk.conf**, created in step 2 of the section, **Apache HTTP Server 2.0.x** on page 68, should be changed to the following:

```
LoadModule jk_module modules/mod_jk.so

JkWorkersFile conf/workers.properties
JkLogFile /var/log/httpd/mod_jk.log
JkLogLevel info
```

```
JkMount /* loadbalancer
```

2. The contents of the file **worker.properties**, created in step 3 of the section, **Apache HTTP Server 2.0.x** on page 68 section, should be changed to the following:

```
worker.list = loadbalancer

worker.node1.port = 8009
worker.node1.host = 192.168.1.100
worker.node1.type = ajp13
worker.node1.lbfactor = 1

worker.node2.port = 8009
worker.node2.host = 192.168.1.101
worker.node2.type = ajp13
worker.node2.lbfactor = 1

worker.loadbalancer.type = lb
worker.loadbalancer.balance_workers = node1, node2
worker.loadbalancer.sticky_session = 1
```

> **Note:** With Apache HTTP Server 2.0.x as the front-end, a clustered deployment of the Asynchronous HTTP Server is not supported. However, when the Asynchronous HTTP Server is deployed to one node in the cluster, this version of Apache can be used. All ICEfaces applications can still be deployed to the cluster on all nodes.

Apache HTTP Server 2.2.x

Apache 2.2.x comes with its own support for load balancing and failover. To utilize the load balancing features, the following steps need to be taken.

1. The ProxyPass and ProxyPassReverse directives for all requests, including the blocking requests, need to be rewritten to the following:

   When the Asynchronous HTTP Server is running as a server:

```
# The following directives will route all blocking requests to the
# Asynchronous HTTP Server.
ProxyPass        /application-name/block/receive-updated-views
                 balancer://async-http-server-cluster/
                 application-name/block/receive-updated-views
                 lbmethod=bytraffic nofailover=Off
ProxyPassReverse /application-name/block/receive-updated-views
                 http://192.168.1.100:51315/application-name/block/
                 receive-updated-views
ProxyPassReverse /application-name/block/receive-updated-views
                 http://192.168.1.101:51315/application-name/block/
                 receive-updated-views

# The following directives will route all other requests to the
# application server.
ProxyPass        /application-name
                 balancer://application-server-cluster/application-name
                 stickysession=JSESSIONID lbmethod=bytraffic nofailover=On
ProxyPassReverse /application-name
```

```
                ajp://192.168.1.100:8009/application-name
ProxyPassReverse /application-name
                ajp://192.168.1.101:8009/application-name
```

When the Asynchronous HTTP Server is running as a servlet:

```
# The following directives will route all blocking requests to the
# Asynchronous HTTP Server.
ProxyPass        /async-http-server
                 balancer://async-http-server-cluster/async-http-server
                 lbmethod=bytraffic nofailover=Off
ProxyPassReverse /async-http-server
                 ajp://192.168.1.100:8009/async-http-server
ProxyPassReverse /async-http-server
                 ajp://192.168.1.101:8009/async-http-server

# The following directives will route all other requests to the
# application server.
ProxyPass        /application-name
                 balancer://application-server-cluster/application-name
                 stickysession=JSESSIONID lbmethod=bytraffic nofailover=On
ProxyPassReverse /application-name
                 ajp://192.168.1.100:8009/application-name
ProxyPassReverse /application-name
                 ajp://192.168.1.101:8009/application-name
```

Note: The previous ProxyPass and ProxyPass Reverse directives must appear on a single line in your configuration file.

2. Additionally the balance members for each balancer need to be specified as follows:

   When the Asynchronous HTTP Server is running as a server:

```
<Proxy balancer://async-http-server-cluster>
    BalancerMember http://192.168.1.100:51315
    BalancerMember http://192.168.1.101:51315
</Proxy>

<Proxy balancer://application-server-cluster>
    BalancerMember ajp://192.168.1.100:8009 route=node1
    BalancerMember ajp://192.168.1.101:8009 route=node2
</Proxy>
```

   When the Asynchronous HTTP Server is running as a servlet:

```
<Proxy balancer://async-http-server-cluster>
    BalancerMember ajp://192.168.1.100:8009
    BalancerMember ajp://192.168.1.101:8009
</Proxy>

<Proxy balancer://application-server-cluster>
    BalancerMember ajp://192.168.1.100:8009 route=node1
    BalancerMember ajp://192.168.1.101:8009 route=node2
</Proxy>
```

3. In order to have load balancing support, the mod_proxy_balancer module is required. In the Apache configuration file where all modules are being loaded, add the following if not already added:

```
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
```

WebLogic Server 8.1 Service Pack 4

Configuring JMS for WebLogic Server 8.1

To configure the JMS topics, go through the following steps:

1. Using an Internet browser, go to WebLogic's console (for example http://localhost:7001/console) and login.
2. On the left panel, navigate to **Services** > **JMS** > **Connection Factories**.
3. On the right panel, select **Configure a new JMS Connection Factory**.
4. Enter a **Name** (for example, *ConnectionFactory*) and a **JNDI name** (for example, *ConnectionFactory*), and then click **Create**.
5. Select a server as the target and click **Apply**.
6. On the left panel, navigate to **Services** > **JMS** > **Servers** > **WSStoreForwardInternalJMSServermyserver** > **Destinations** (where myserver is the name of your server).
7. On the right panel, select **Configure a new JMS Topic**.
8. Enter *icefaces.contextEventTopic* for both the **Name** and **JNDI name**, and then click **Create**.
9. Repeat steps 6 to 8 for *icefaces.responseTopic*.

For more information on how to configure JMS on WebLogic, refer to **JMS: Configuring** which can be found at:

http://e-docs.bea.com/wls/docs81/ConsoleHelp/jms_config.html

Web Server Plug-Ins

Plug-ins are modules that can be added to a web server installation and can be configured to enable interaction between the web server of choice and the application server of choice. Typically, plug-ins can be used as a load-balancer for the web server by proxying the requests to the back-end application servers, or can be used to proxy requests for dynamic content to the back-end server(s).

Apache HTTP Server 2.0.x

The following is a simple solution for installing and configuring WebLogic's plug-in for the Apache HTTP Server.

1. First, copy the **mod_wl_20.so** module supplied by BEA into the Apache's /**module** directory. You can obtain the **mod_wl_20.so** module from WebLogic's installation from this directory: **[bea-home-dir]/weblogic81/server/lib/[os]/**

2. Create a new file called **mod_wl_20.conf** in the Apache's /**conf** directory and add the following code to it:

```
LoadModule weblogic_module modules/mod_wl_20.so

<IfModule mod_weblogic.c>
    WebLogicHost host
    WebLogicPort port
</IfModule>

<LocationMatch ^/application-name(?!/block/receive-updated-views)>
    SetHandler weblogic-handler
</LocationMatch>
```

   The LocationMatch container is responsible for forwarding all the requests to the ICEfaces application deployed to WebLogic with the exception of the blocking requests. To explain the regular expression used in the LocationMatch container, the following is a breakdown:

   - ^/**application-name**
     if the location begins with (^) the literal string "/**application-name**", where **application-name** is the name of the ICEfaces application, and
   - (?!/block/receive-updated-views)
     is not followed by (?!) the literal string "/block/receive-updated-views".

   This regular expression ensures that the following possible locations get handled by the plug-in:

   - /**application-name** (initial request)
   - /**application-name**/ (initial request)
   - /**application-name**/xmlhttp/icefaces-d2d.js (part of initial request)
   - /**application-name**/block/receive-send-updates?<query> (synchronous request)
   - /**application-name**/block/send-updates?<query> (UI request)

   But it prevents that the following possible location gets handled by the plug-in:

   - /**application-name**/block/receive-updated-views?<query> (asynchronous request)

   When the Asynchronous HTTP Server is running as a servlet, the mentioned LocationMatch container can be replaced by the following:

```
<LocationMatch ^/async-http-server>
    SetHandler weblogic-handler
</LocationMatch>

<LocationMatch ^/application-name>
    SetHandler weblogic-handler
</LocationMatch>
```

3. The **mod_wl_20.conf** file needs to be loaded. To achieve this, add the following to Apache's configuration file right after the LoadModule directives:

```
Include conf/mod_wl_20.conf
```

4. Finally, when the Asynchronous HTTP Server is running as a server, the ProxyPass and ProxyPassReverse directives, which route all requests other than the blocking requests to the

application servers, need to be removed from the Apache configuration file. The same counts for when running as a servlet, but additionally the ProxyPass and ProxyPassReverse directives, which route the blocking requests to the Asynchronous HTTP Server need to be removed..

For more information on how to install and configure WebLogic's plug-ins, refer to **BEA WebLogic Server - Using Web Server Plug-Ins with WebLogic Server**, which can be found at:

http://e-docs.bea.com/wls/docs81/pdf/plugins.pdf

Apache HTTP Server 2.2.x

BEA plans to have a plug-in for Apache HTTP Server 2.2.x available with the WebLogic Server 8.1 SP 7 and 9.5 releases.

Clustering

This section explains how to set up a clustered deployment of WebLogic Servers. It includes steps for configuring the cluster, JMS and the web server. The following discussion relates to a clean installation of WebLogic Server, and represents a simplified process for configuring WebLogic for clustered deployments of ICEfaces applications. Additional deployment-specific issues may exist. Refer to your WebLogic documentation for additional information.

This example cluster consists of two Managed Servers and an Administration Server, which itself is not part of the cluster, as shown in the table below.

IP Address	Server Name	Description
192.168.1.100	Admin	Administration Server
192.168.1.101	Managed1	Managed Server
192.168.1.102	Managed2	Managed Server

Configuring the Cluster

Ensure that you have WebLogic Server installed on each machine. Refer to your WebLogic documentation for additional information.

The following three procedures describe the steps to create a cluster formed by **Managed1_Server** and **Managed2_Server** and administered by **Admin_Server**.

Configuring the Individual Servers

1. On machine Admin go to `[bea-home-dir]/weblogic81/common/bin` and start the BEA WebLogic Configuration Wizard using:

   ```
   ./config.sh -mode=console
   ```

   Use **config.cmd** on a Windows platform.
2. Select **Create a new WebLogic configuration**.
3. Select **Basic WebLogic Server Domain**.
4. Do not run in express mode; select **No**.

5. Modify the **Name** of the Administration Server to *Admin_Server* and select **Next**.
6. Do not configure the Managed Servers, Clusters and Machines; select **No**.
7. Do not configure JDBC; select **No**.
8. Do not configure JMS; select **No**.
9. Do not configure Advanced Security; select **No**.
10. Modify the user as desired and select **Next**.
11. Select **Production Mode**.
12. Select **JRockit SDK version 1.4.2**.
13. Leave the **Target Location** at its default.
14. Modify the Name of the domain to *ICEfaces_Cluster_Domain* and select **Next**.
15. Repeat steps 1 to 14 for **Managed1_Server** on *Managed1* and **Managed2_Server** on *Managed2*.

Configuring the Cluster

1. On machine Admin go to `[bea-home-dir]/user_projects/domains/ICEfaces_Cluster_Domain` and start the server using:

   ```
   ./startWebLogic.sh
   ```

   Use **startWebLogic.cmd** on a Windows platform.
2. Using an Internet browser, go to `http://192.168.1.100:7001/console` and login as the user specified in step 10 of **Configuring the Individual Servers**.
3. On the left panel, navigate to **Clusters**.
4. On the right panel, select **Configure a new Cluster**.
5. Enter *ICEfaces_Cluster* for **Name** and then click **Create**.
6. On the left panel, navigate to **Machines**.
7. On the right panel, select **Configure a new Machine**.
8. Enter *Managed1* for **Name** and then click **Create**.
9. Repeat steps 6 to 8 for *Managed2*.
10. On the left panel, navigate to **Servers**.
11. On the right panel, select **Configure a new Server**.
12. Enter *Managed1_Server* for **Name**, select *Managed1* for **Machine** and *ICEfaces_Cluster* for **Cluster**; enter *192.168.1.101* in the **Listen Address** field, and then click **Create**.
13. Repeat steps 10 to 12 for *Managed2_Server* running on **Managed2** (192.168.1.102).

Starting the Cluster

1. On machine Managed1 go to `[bea-home-dir]/user_projects/domains/ICEfaces_Cluster_Domain` and start the server using the following command:

   ```
   ./startManagedWebLogic.sh Managed1_Server http://192.168.1.100:7001
   ```

2. Use **startManagedWebLogic.cmd** on a Windows platform.
3. Repeat step 1 for Managed2_Server.
4. Using an Internet browser, go to `http://192.168.1.100:7001/console` and login.
5. On the left panel, navigate to **Servers**.
6. On the right panel, verify that all servers (Admin_Server, Managed1_Server and Managed2_Server) have their state set as *RUNNING*.

Configuring JMS for the Cluster

The following steps describe how to create a ConnectionFactory that is available throughout the cluster and an ICEfaces_JMS_Server deployed on Managed1 servicing the JMS topics, icefaces.contextEventTopic, icefaces.renderTopic, and icefaces.responseTopic, throughout the cluster.

1. Using an Internet browser, go to `http://192.168.1.100:7001/console` and login.
2. On the left panel, navigate to **Services** > **JMS** > **Connection Factories**.
3. On the right panel, select **Configure a new JMS Connection Factory**.
4. Enter *ConnectionFactory* for **Name** and **JNDI Name,** and then click **Create**.
5. Select *All servers in the cluster* for **Targets** and then click **Apply**.
6. On the left panel, navigate to **Services** > **JMS** > **Servers**.
7. On the right panel, select **Configure a new JMS Server**.
8. Enter *ICEfaces_JMS_Server* for **Name** and then click **Create**.
9. Select *Managed1_Server* for **Target** and then click **Apply**.
10. On the left panel, navigate to **Services** > **JMS** > **Servers** > **ICEfaces_JMS_Server** > **Destinations**.
11. On the right panel, select **Configure a new JMS Topic**.
12. Enter *icefaces.contextEventTopic* for **Name** and **JNDI Name**, select *False* for **Enable Store**, and then click **Create**.
13. Repeat steps 10 to 12 for *icefaces.renderTopic* and *icefaces.responseTopic*.

Web Server Plug-Ins

Apache HTTP Server 2.0.x

The following is a simple solution for installing and configuring WebLogic's plug-in for Apache 2.0.x and a cluster of WebLogic Servers. Refer to the previous Apache HTTP Server 2.0.x section for the initial installation and configuration of WebLogic's plug-in for Apache 2.0.x.

1. The IfModule container of the file **mod_wl_20.conf**, created in step 1 of the previous **Apache HTTP Server 2.0.x** section, should be changed to the following: :

```
<IfModule mod_weblogic.c>
    WebLogicCluster 192.168.1.101:7001,192.168.1.102:7001
</IfModule>
```

Apache HTTP Server 2.2.x

BEA is planning to have a plug-in for Apache HTTP Server 2.2.x available with the releases of WebLogic Server 8.1 SP 7 and 9.5.

WebSphere Application Server 6.0.2

This section assumes that you have a working installation of WAS 6.0.2, IBM HTTP Server (IHS), and the Web Server Plug-In for WAS where you can successfully deploy web applications. For complete details on installing and configuring WebSphere Application Server (WAS), see IBM's official documentation for WAS 6.0.x at:

http://www.ibm.com/software/webservers/appserv/was/library/library60.html

There are several different JMS (Java Message Service) options available to run with WAS. The most common solution and the simplest one to configure is the Default messaging provider based on the System Integration Bus (SIB). To configure the JMS topics, follow these steps:

Login to the Admin Console

1. From your web browser, login to the WAS console (for example, http://localhost:9060/ibm/console/). By default there is no security so you should be able to login using any name.

Create the Service Integration Bus

1. In the tree on the left panel, navigate to **Service Integration** > **Buses**.
2. Click **New**.
3. Enter the information for the Bus. You will need to provide a name (e.g,. ICEfacesBus). To make the initial configuration easier, you may also want to turn off security by de-selecting the **Secure** check box. You can enable security later once you have everything running.
4. A **Messages** box at the top of the screen will prompt you to save your changes to the master configuration. Click the **Save** link.
5. In the next panel, click **Save**. When it is done saving, click **OK**.

Add the JMS Topic Connection Factory

1. In the tree on the left panel, navigate to **Resources** > **JMS Providers** > **Default messaging**.
2. In the Default messaging provider **Configuration** panel, under **Connection Factories**, click the **JMS topic connection factory** link.
3. Click **New**.
4. Supply the following information for the connection factory:

   Name=ConnectionFactory

   JNDI name=jms/ConnectionFactory
5. Select the Service Integration Bus you created previously by choosing it from the drop-down list under **Bus** name.
6. A **Messages** box at the top of the screen will prompt you to save your changes to the master configuration. Click the **Save** link.
7. In the next panel, click **Save**. When it is done saving, click **OK**.

Add the JMS Topics

1. In the tree on the left panel, navigate to **Resources** > **JMS Providers** > **Default messaging**.
2. In the Default messaging provider **Configuration** panel, under **Connection Factories**, click the **JMS topic** link.
3. Click **New**.
4. Supply the following information for the topic:

   Name=icefaces.contextEventTopic

   JNDI name=jms/icefaces.contextEventTopic

   Topic name=icefaces.contextEventTopic
5. Select the Service Integration Bus you created previously by choosing it from the drop-down list under **Bus** name.
6. To add one more topic, repeat steps 3 and 4 using the value "icefaces.responseTopic" for the **Name** in step 4.
7. A **Messages** box at the top of the screen will prompt you to save your changes to the master configuration. Click the **Save** link.
8. In the next panel, click **Save**. When it is done saving, click **OK**.

JMS should now be properly configured to support ICEfaces applications running with the Asynchronous Server.

Web Server and Web Server Plug-Ins

IBM provides an option to install and use a custom version of the Apache web server, known as IBM HTTP Server (IHS). It also provides a custom Apache plug-in to support efficient communication between IHS and WAS. Although WAS can run with a standalone Apache configuration as well, the IBM

HTTP Server provides tighter administrative integration with WAS. Official documentation for IHS 6.0.x can be found at:

http://www.ibm.com/software/webservers/httpservers/library/

To get ICEfaces with Asynchronous HTTP Server working with IHS and WAS requires that we include additional configuration information to IHS.

You can modify the IHS configuration in one of two ways. You can use the WAS administration application or you can modify it manually. The default location of the configuration file varies by platform so you'll need to consult the documentation to determine where it is. On Linux, the default installation location of the web server's configuration file is `/opt/IBMIHS/conf/httpd.conf`. Whether you do it manually or use the administration tool, you'll need to modify the configuration file in the following way:

1. Ensure that the rewrite and proxy modules are loaded. If they are commented out, uncomment them.

   ```
   LoadModule proxy_module modules/mod_proxy.so
   LoadModule rewrite_module modules/mod_rewrite.so
   ```

2. Ensure that the WebSphere plug-in module is loaded and that the plug-in configuration file is loaded from the correct location. This information will depend on what platform you are running on as well as the topology of your servers. Typically this will have been done already when you installed the plug-in for IHS.

   ```
   LoadModule was_ap20_module /opt/IBM/WebSphere/Plugins/bin/mod_was_ap20_http.so
   WebSpherePluginConfig <path-to-plugin-config>/plugin-cfg.xml
   ```

3. Add the following lines to engage the rewrite engine and add a rewrite rule that redirects all blocking requests to the Asynchronous HTTP Server. The [P] flag at the end tells Apache to forward the re-written request to the Proxy directive. Replace the host:port entry with the host and port of the Asynchronous HTTP Server (e.g., myhost.com:51315)

   ```
   RewriteEngine on
   RewriteRule ^/(.*)/block/receive-updated-views(.*)
                       http://host:port/block/receive-updated-views$2 [P]
   ```

   > **Note:** The RewriteRule directive must appear on a single line in your configuration file.

4. Enable the proxy.

   > **Note:** The ProxyPass and ProxyPassReverse settings outlined in the Apache HTTP Server section are not required when running with WAS.

   ```
   <IfModule mod_proxy.c>
       ProxyRequests On
           ProxyVia Off

        <Proxy *>
           Order deny,allow
           Allow from all
       </Proxy>
   </IfModule>
   ```

Deploying ICEfaces Applications

Once WebSphere is fully configured, you can deploy your ICEfaces application. We recommend that you review **Deploying the Asynchronous HTTP Server**, *p. 61* and **Configuring the ICEfaces Application**, *p. 64* before deploying your own application. As outlined in those sections, you will need to configure the context-param named *com.icesoft.net.messaging.properties* in the **web.xml** file of both the async-http-server.war and your application war. For running in non-clustered mode in WebSphere, the parameter should be set to *websphere.properties*.

You can deploy the Asynchronous Server (async-http-server.war) and your application separately via the Administration console. However, since we have more than one piece to deploy, it is more convenient to use an .ear file that contains all the necessary pieces of your application (which could also include other things like EJBs).

Clustering

Installing and Configuring the Cluster

You can run ICEfaces applications in a WebSphere Application Server Network Deployment (WAS ND) 6.0.2 cluster. Installing and configuring a WAS ND cluster can be somewhat daunting in that there are a number of different cluster architectures to choose from and a great deal of documentation to wade through. Given that, ICEsoft is providing this high-level summary of what we did to get our test cluster up and running. This summary is in no way intended to replace the official IBM documentation. To keep things simple, we do not include any security configuration. You'll need to adjust the configuration to suit your own security policies.

For our example, we're going to assume our cluster consists of four machines running in the following clustered architecture. You can run with fewer machines and with different configurations to suit your available resources.

IP Address	Server Name	Description
192.168.1.100	admin	Administration Server, IBM HTTP Server (IHS), and WebSphere Plug-Ins
192.168.1.101	managed1	Managed Application Server
192.168.1.102	managed2	Managed Application Server
192.168.1.103	database	DB2 Server

The following instructions assume:

- You have a copy of WebSphere Application Server Network Deployment (WAS ND) version 6.0.2. We recommend getting the latest fix pack for this version as well. This includes the IBM HTTP Server (IHS) and Plug-In installers as well.
- You have a copy of a WAS ND supported database. We used IBM DB2 Universal Database Express Edition.
- You have four WAS ND supported platforms as outlined above. We installed and ran our cluster on RedHat Enterprise 3 Linux.

- You have Root or superuser privileges on the machines and local or remote access to a command line shell.

> **Note:** In the steps that follow, node, cell, and host names must be modified as required to match your environment.

Install the Core WAS Files

1. Install the Core WAS ND files on the admin, managed1, and managed2 nodes. You can use the supplied GUI installer or shell script. Refer to IBM's documentation on running the installer of your choice. For example, on our machines we used the script and a custom response file to do the installation:

   ```
   cd /home/icesoft/WAS-ND-V6/WAS/
   ./install -options /home/icesoft/WAS-ND-V6/WAS/responsefile.nd.ice.txt -silent
   ```

   The default location for the installed files varies by platform. On Linux, the standard location for the installed files is /opt/IBM/WebSphere/AppServer.

   When running on a standalone WAS server, you can take advantage of the built-in database (Cloudscape) that comes with WAS. For a messaging engine to work in a cluster, you must install a separate database. This is solely a WebSphere clustering requirement as the JMS messaging used by ICEfaces does not require the messages to be persisted. Out of the box, WAS supports a number of common database configurations. To avoid problems with databases that are not officially supported, we chose DB2.

Install the Database

1. Using the documentation provided by the vendor, install and start a database instance on the database node.
2. Configure a name/password that has enough privileges to create a schema and tables. Note the **database name**, **user name**, and **password** as you will need them later.

Create the Profiles

1. On the admin server, create a Deployment Manager profile:

   ```
   cd /opt/IBM/WebSphere/AppServer/bin

   ./wasprofile.sh -create -profileName deployMgrProfile -profilePath
   /opt/IBM/WebSphere/AppServer/profiles/deployMgr -templatePath
   /opt/IBM/WebSphere/AppServer/profileTemplates/dmgr -nodeName adminNode -cellName
   mgrCell -hostName admin.com
   ```

2. Start the Deployment Manager:

   ```
   cd /opt/IBM/WebSphere/AppServer/profiles/deployMgr/bin/
   ./startManager.sh
   ```

3. On the managed1 server, create a Managed Server profile:

   ```
   cd /opt/IBM/WebSphere/AppServer/bin
   ```

```
./wasprofile.sh -create -profileName managedServer1 -profilePath
/opt/IBM/WebSphere/AppServer/profiles/managedServer1 -templatePath
/opt/IBM/WebSphere/AppServer/profileTemplates/managed -nodeName managedNode1 -
cellName managed1Cell -hostName managed1.com -dmgrHost admin.com
```

4. Federate and start the node:

```
cd /opt/IBM/WebSphere/AppServer/profiles/managedServer1/bin/
./addNode.sh admin.com
```

5. Create the same type of profile on the managed2 server:

```
cd /opt/IBM/WebSphere/AppServer/bin
./wasprofile.sh -create -profileName managedServer2 -profilePath
/opt/IBM/WebSphere/AppServer/profiles/managedServer2 -templatePath
/opt/IBM/WebSphere/AppServer/profileTemplates/managed -nodeName managedNode2 -
cellName managed2Cell -hostName managed2.com -dmgrHost admin.com
```

6. Federate and start the node:

```
cd /opt/IBM/WebSphere/AppServer/profiles/managedServer2/bin/
./addNode.sh admin.com
```

At this point, the Deployment Manager should be running as well as the node agents on the managed servers so you can centrally administrate the individual servers from the Deployment Manager console. Using the console, we create a cluster and add application server instances to the newly created cluster.

Login to the Admin Console

1. From your web browser, login to the WAS console (for example, http://localhost:9060/ibm/console/). By default there is no security so you should be able to log in using any name.

Create the Cluster and Server Instances

1. Create a new cluster and add managed server instances to the managed nodes:
   a. In the tree on the left panel, navigate to **Servers Clusters**.
   b. Click **New**.
   c. Enter the information for the Cluster. You will need to provide a name (e.g., IceCluster) in the **Cluster name** field. Click **Next**.
   d. On the **Create cluster members** page, enter managed1Member into the **Member name** field and select the managed1Node from the **Select node** list, and then click **Apply**. This creates a new application server instance for this cluster.
   e. Enter managed2Member into the **Member name** field and select the managed2Node from the **Select node** list, and then click **Apply**. This creates another server instance in the other node and includes it in this cluster.
   f. Click **Next**.
   g. Click **Finish**.
2. A **Messages** box at the top of the screen will prompt you to save your changes to the master configuration. Click the **Save** link.
3. In the next panel, click **Save**.

Create a Security Alias

1. In the tree on the left panel, navigate to **Security** > **Global security**.
2. Under **Authentication**, click **JAAS Configuration** to expand the tree.
3. Click on **J2C Authentication** data.
4. Click **New**.
5. Enter an alias for the security configuration, and then supply the **user name** and **password** required to access the database.
6. Click **OK**.
7. Click the **Save** link near the top of the page.
8. Ensure that **Synchronize changes with Nodes** is checked, and then click **Save**.
9. When the changes have finished synchronizing, click **OK**.

Create a DataSource

1. In the tree on the left panel, navigate to **Resources** > **JDBC Providers**.
2. Make sure you are creating the resource in the proper scope (cluster). To set the scope of the resource definition to cluster:
   a. Click **Browse Clusters**.
   b. Click the radio button beside the correct cluster then click **OK**.
3. Click **New**.
4. For the **General Properties**, select the items that apply to your environment. For ours, we chose:

   Step 1: Select the database type -> DB2

   Step 2: Select the provider type -> DB2 Universal JDBC Driver Provider

   Step 3: Select the implementation type -> Connection pool data source
5. Click **Next**.
6. With DB2, the default settings on this page should be sufficient. If you are using another database, you may need to adjust one or more of the entries. The Class path and Native library path values can use environment variables. Ensure that the variables are properly set. They can be found by navigating to Environment -> WebSphere Variables.
7. When you have made all the necessary adjustments, click **Apply**.
8. Under **Additional Properties**, click the **Data sources** link.
9. Click **New**.
10. Adjust the **General Properties**:
    a. Under Component-managed authentication alias, choose the security alias you created in the previous section (e.g., deployMgrNode/db2Alias).
    b. Provide the **database name** that you created in the Install the Database section.
    c. Provide the host name (or IP address) of the server that the database is running on.

> **Note:** Make a note of the **JNDI name** as you will need this information later. With DB2, the default is something like jdbc/DB2 Universal Driver Provider.

11. Provide the port number on which the database is configured to accept remote connections.
12. Click **OK**.
13. A **Messages** box at the top of the screen will prompt you to save your changes to the master configuration. Click the **Save** link.
14. In the next panel, click **Save**. When it is done saving, click **OK**.
15. Test the configuration using the **Test Connection** button.

Now we can create the bus and add the cluster to it. We'll be going through some of the same steps as in the non-clustered configuration.

Create the Service Integration Bus

1. In the tree on the left panel, navigate to **Service Integration** > **Buses**.
2. Click **New**.
3. Enter the information for the Bus. You will need to provide a name (e.g., ClusterBus). To make the initial configuration easier, you may also want to turn off security by de-selecting the **Secure** check box. You can enable security later once you have everything running.
4. Click **OK**.
5. A **Messages** box at the top of the screen will prompt you to save your changes to the master configuration. Click the **Save** link.
6. In the next panel, click **Save**. When it is done saving, click **OK**.

Add the Cluster to the Service Integration Bus (SIB)

1. In the tree on the left panel, navigate to **Service Integration** > **Buses**.
2. Click on the bus that you created.
3. Under the topology heading, click **Bus members**.
4. Click **Add**.
5. Select the **Cluster** radio button and choose the cluster you created from the list.
6. Enter the **JNDI name** of the data source you created earlier. For example, the one we created was:

   ```
   jdbc/DB2 Universal JDBC Driver DataSource
   ```

7. Click **Next** and then click **Finish**.
8. A **Messages** box at the top of the screen will prompt you to save your changes to the master configuration. Click the **Save** link.
9. In the next panel, click **Save**. When it is done saving, click **OK**.

Adding the cluster to the SIB automatically creates a bus-wide messaging engine. We need to ensure that the general properties are set correctly for the messaging engine to use the database.

Adjust the Messaging Engine Properties

1. In the tree on the left panel, navigate to **Service Integration** > **Buses**.
2. Click on the bus you created (e.g., ClusterBus).
3. Under **Topology**, click **Messaging engines**.
4. Click the name of the messaging engine. This name is generated and will look like *clusterName.000-busName*.
5. Under **Additional Properties**, click the **Data store** link.
6. Set the **Authentication** alias to the security alias we created earlier.
7. Adjust the Schema value so that it matches your database configuration.
8. Ensure the **Create tables** check box is enabled.
9. Click **OK**.
10. A **Messages** box at the top of the screen will prompt you to save your changes to the master configuration. Click the **Save** link.
11. In the next panel, click **Save**. When it is done saving, click **OK**.

Add the JMS Topic Connection Factory

1. In the tree on the left panel, navigate to **Resources** > **JMS Providers** > **Default messaging**.
2. In the Default messaging provider **Configuration** panel, under **Connection Factories**, click the **JMS topic connection factory** link.
3. Make sure you are creating the resource in the proper scope (cluster). To set the scope of the resource definition to cluster:
   a. Click **Browse Clusters**.
   b. Click the radio button beside the correct cluster then click **OK**.
4. Click **New**.
5. Supply the following information for the connection factory:

   Name=ConnectionFactory

   JNDI name=jms/ConnectionFactory
6. Select the Service Integration Bus (e.g., ClusterBus) you created previously by choosing it from the drop-down list under **Bus** name.
7. Click **OK**.
8. A **Messages** box at the top of the screen will prompt you to save your changes to the master configuration. Click the **Save** link.
9. In the next panel, click **Save**. When it is done saving, click **OK**.

Add the JMS Topics

1. In the tree on the left panel, navigate to **Resources** > **JMS Providers** > **Default messaging**.
2. In the Default messaging provider **Configuration** panel, under **Destinations**, click the JMS topic link.
3. Click **New**.
4. Supply the following information for the topic:

   Name=icefaces.contextEventTopic

   JNDI name=jms/icefaces.contextEventTopic

   Topic name=icefaces.contextEventTopic
5. Select the Service Integration Bus (e.g., ClusterBus) you created previously by choosing it from the drop-down list under **Bus** name.
6. Click **OK**.
7. To add one more topic, repeat steps 3 – 5 using the value *icefaces.responseTopic* in place of *icefaces.contextEventTopic* for all the values in step 4.
8. A **Messages** box at the top of the screen will prompt you to save your changes to the master configuration. Click the **Save** link.
9. In the next panel, click **Save**. When it is done saving, click **OK**.

JMS should now be properly configured to support ICEfaces applications running with the Asynchronous Server.

Web Server and Web Server Plug-Ins

ICEfaces supports running applications in a WebSphere cluster. However, due to the way the WebSphere plug-in works, it is not currently possible to run multiple Asynchronous Servers for load-balancing and fail-over purposes. Instead, you'll have a single active Asynchronous Server designated to handle all the asynchronous request traffic. So the configuration of the web server and the web server plug-ins is the same as it is for non-clustered WebSphere installations (see **Web Server and Web Server Plug-Ins** on page 80).

Deploying

Once the cluster is fully configured, you can deploy your ICEfaces application. Ensure you review **Deploying the Asynchronous HTTP Server**, *p. 61* and **Configuring the ICEfaces Application**, *p. 64* before deploying your own application. As outlined in those sections, you'll need to configure the context-param named *com.icesoft.net.messaging.properties* in the **web.xml** file of both the async-http-server.war and your application war. For running in clustered mode in WebSphere, the parameter should be set to *websphere_ha.properties*.

Using the Administration console, deploying to a cluster is the same process as deploying to a single, standalone node.

JBoss Seam Integration

JBoss Seam is a middleware technology that integrates JSF with EJB3. It can vastly reduce the amount of XML configuration required to develop applications. For more information on Seam, see **http://www.jboss.com/products/seam**.

ICEfaces v1.6 and v1.7 have been tested with jboss-seam-1.2.1.GA and jboss-seam-2.0.0 GA. ICEfaces v1.6.1 is referenced by the jboss-seam-2.0.0.GA distribution package and any version of ICEfaces v1.6.1 or greater is required for jboss-seam-2.0.0.GA.

Note: Seam 1.2.1.GA may use either JSF 1.1 or 1.2 specifications, whereas Seam 2.0.0 GA requires the use of JSF 1.2 specifications. The two Seam versions also have different dependencies on external libraries (JARs).

Resources

The following additional resources are available if you are developing Seam applications with ICEfaces:

- **Using ICEfaces with JBoss Seam:** ICEfaces Knowledge Base category (**http://support.icesoft.com/jive/category.jspa?categoryID=71**).
  Refer to the Knowledge Base for the latest information on using ICEfaces with Seam.
- **JBoss Seam Integration:** ICEfaces Forum (**http://www.icefaces.org/JForum/forums/show/22.page**).
  This forum contains discussions related to using ICEfaces with JBoss Seam applications.

Getting Started

The simplest way to produce a working ICEfaces/Seam application is to use the icefaces-seam-gen utility which is available as a separate download at **http://downloads. icefaces.org/** under the **Projects** section. The USAGE and README files, which are included in the bundle, describe how to install and run the ICEfaces-specific version of seam-gen from your Seam installation. A tools download for icefaces-seam-gen integration is available for Netbeans-5.5 on the ICEFaces download page.

The seam-gen tool distributed with jboss-seam-2.0.0.GA now includes an icefaces option which will generate all the proper configuration with a build script for your project to be deployed on a jboss-4.2.* AS. A basic seam/ICEfaces project is created with the "new-project" target. See other available targets in the README file of the seam-gen distribution.

A hotel booking example enhanced with ICEfaces is also distributed with jboss-seam-2.0.0.GA in the seam-icefaces examples directory.

For jboss-seam-1.2.1.GA, an EAR deployment of ICEfaces Component Showcase implemented as a Seam application is available as a separate download at **http://downloads.icefaces.org/.** Three targets are included for creating an EAR deployment of seam-comp-showcase:

- **build-myfaces target** which creates an EAR application for JSF 1.1 specs and jboss-seam-1.2.1.GA (recommended server is jboss-4.0.5.GA with EJB3 container)
- **build-Seam1.2WithJsf1.2** which creates an EAR application for JSF 1.2 specifications and jboss-seam-1.2.1.GA (recommended server is jboss-4.2.*.GA but you must copy el-*.jars to embedded Tomcat AS for this server)
- **build-Seam2.0WithJsf2.0** which creates an EAR application for JSF 1.2 specifications and jboss-seam-2.0.0.GA (recommended server is jboss-4.2.*.GA)

With jboss-seam-2.0.0.GA, Seam may also be used without an EJB3 container in the application server. A .war deployment of seam-comp-showcase is available for jboss-seam-2.0.0.GA that may be deployed to various servers. See the README file for a listing of servers and how to build and deploy this application. The best results are obtained when JSF specifications match that of the server. In other words, if an application server has J2EE specifications, jboss-seam-1.2.1.GA with the myfaces (jsf-1.1 specifications) JARs are best to use. ICEfaces 1.6.0 and subsequent versions have all been tested with this configuration. If an application server is denoted J5EE, then jboss-seam-2.0.0.GA, jsf-1.2 specifications will provide the best solution. The .war deployment of seam-comp-showcase requires a minimum version of ICEfaces-1.6.2 and works best on J5EE AS.

Configuring a Seam ICEfaces Application for jboss-seam-1.2.1.GA

To avoid classloading issues between Seam and ICEfaces, you must use the following approach:

1. Use an EAR type of deployment. Examples of how to create an EAR are available in the Seam documentation, or you can use seam-gen and specify 'ear' at the appropriate prompt.
2. Package the ICEfaces JARs in the EAR directory and specify them as modules in the application.xml. The following JARs need to be included:
    - icefaces.jar
    - icefaces-comps.jar
    - icefaces-facelets.jar
    - backport-util-concurrent.jar
    - commons-fileupload.jar
    - jbpm-3.1.4.jar

The EAR contains a reference to the <application>.war (or web tier) package and a reference to the <application>.JAR (or EJB3) package. Ensure that the ICEfaces JARs are not included in the .war file's `WEB-INF/lib` directory, as this will override the JARs being loaded from the EAR's application.xml.

The correct **web.xml** is dependent on which version of the JSF specification is used. This file is included in the web tier.

JSF 1.1 specifications -for jboss-seam-1.2.1.GA (recommended) and earlier distributions of Seam

> **Note:** Modules must be specified in applications.xml and only EAR deployments with ICEfaces is supported.

Although .war deployment is possible, there is no example application showing this. The myfaces*.jars are included in the jboss-seam-1.2.1.GA distribution. To see the configuration and packaging, download the seam-comp-showcase EAR example or icefaces-seam-gen from the ICEfaces download site.

JSF 1.2 specifications - for jboss-seam-1.2.1.GA or jboss-seam-2.0.0.GA

For JSF 1.2 specifications (using Sun JSF 1.2 RI jars), the modules no longer need to be defined as such but the JARs can just be included in the META-INF\lib directory (other than jboss-seam.jar). See the examples or generate an application using icefaces-seam-gen (jboss-seam-1.2.1.GA) or seam-gen(jboss-seam-2.0.0.GA).

Only one version of faces-config.xml is needed. Ensure that no other facelet view handler is used. The following is an example of faces-config.xml for jboss-seam-1.2.1.GA. The same file for jboss-seam-2.0.0.GA does not require the phase-listener.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE faces-config PUBLIC
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
"http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config>
  <application>
   <message-bundle>messages</message-bundle>
      <view-handler>
          com.icesoft.faces.facelets.D2DSeamFaceletViewHandler
      </view-handler>
   </application>
   <!-- Seam transaction management -->
   <lifecycle>
      <phase-listener>
           org.jboss.seam.jsf.TransactionalSeamPhaseListener
      </phase-listener>
   </lifecycle>
</faces-config>
```

Using Server-initiated Rendering

Asynchronous Configuration

To use the RenderManager you must configure the application for Asynchronous execution mode. The **web.xml** should contain the following configuration.

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

> **Note:** You should use Synchronous update mode (synchronousUpdate=true) if your application does NOT use the ICEfaces server-initiated rendering feature.

Prior to ICEfaces v1.6, ICEfaces application developers could use JSF to instantiate the application scope RenderManager and cause this instance to be initialized via a property setter on their state bean for initiating an IntervalRenderer, or some similar mechanism for triggering a render portion of the JSF lifecycle. This still works, but as of v1.6, developers can now configure Seam to load the RenderManager as a named component, and can use injection to pass the RenderManager instance to their beans.

Render Manager as Named Component

To get the RenderManager to be loaded as a named Seam component, add the following line to the components.xml file:

```
<component scope="APPLICATION" auto-create="true" name="renderManager"
        class="com.icesoft.faces.async.render.RenderManager" />
```

This is effectively the same as using JSF to create a managed bean instance. There are a couple of ways to gain access to the RenderManager via injection:

1. Declare the RenderManager with the @In Seam annotation.

   ```
   @In
   private RenderManager renderManager;
   ```

2. Declare the setter property with the @In annotation.

   ```
   @In
   public void setRenderManager(RenderManager x) {
      renderManager = x;
   }
   ```

Using RenderManager

For a working example of TimerBean that uses the RenderManager to set up an interval renderer to do clock ticks, generate a seam-gen project (either jboss-seam-1.2.1.GA and icefaces-seam-gen) or jboss-seam-2.0.0.GA and its distributed seam-gen application.

Refer to the notes that follow for more information on the marked sections of the code example.

```
@Name("timer")
@Scope(ScopeType.PAGE)
public class TimerBeanImpl implements Renderable, TimerBean {   [See NOTE 1 below.]
private DateFormat dateFormatter;
@In
private RenderManager renderManager;
private boolean doneSetup;
private IntervalRenderer ir;
private PersistentFacesState state = PersistentFacesState.getInstance();
private String synchronous;
private int myId;
private static int id;

public PersistentFacesState getState() {
return state;
}
public void renderingException( RenderingException re) {
    if (log.isTraceEnabled()) {
      log.trace("*** View obsoleted: " + myId);
    }
    cleanup();
}

public TimerBeanImpl() {
  dateFormatter = DateFormat.getDateTimeInstance();
  myId = ++id;
}
/**
* This getter is bound to an <ice:outputText> element
*/ public String getCurrentTime() {
     state = PersistentFacesState.getInstance();                [See NOTE 2 below.]
     if (!doneSetup) {                                          [See NOTE 3 below.]
     FacesContext fc = FacesContext.getCurrentInstance();
     synchronous = (String) fc.getExternalContext()
      .getInitParameterMap().get(
      "com.icesoft.faces.synchronousUpdate");
     boolean timed = Boolean.valueOf((String) fc.getExternalContext()
      .getInitParameterMap().get(
      "org.icesoft.examples.serverClock"));
     if (timed) {
  ir = renderManager.getIntervalRenderer("org.icesoft.clock.clockRenderer");
  ir.setInterval(2000);
  ir.add(this);
  ir.requestRender();
      }
    }
      doneSetup = true;
      return dateFormatter.format( new Date( System.currentTimeMillis() ) );
}
public String getRenderMode() {
      return synchronous + " " + myId;
}
public String getCurrentConversation() {
      Manager m = Manager.instance();
        return m.getCurrentConversationId();
}
```

```
public String getLongRunning() {
  Manager m = Manager.instance();
  return Boolean.toString(m.isLongRunningConversation());
}

@Remove
@Destroy
public void remove() {                                    [See NOTE 4 below.]
  if (log.isTraceEnabled()) {
    log.trace("*** View removed: " + myId);
  }
  cleanup();
}
  public void viewCreated() {
  }

  public void viewDisposed() {
    if (log.isTraceEnabled()) {
      log.trace("*** View disposed: " + myId);
    }
    cleanup();
  }

  private void cleanup() {
    if (ir != null) {
      ir.remove(this);
      if (ir.isEmpty()) {
        if (log.isTraceEnabled()) {
          log.trace("*** IntervalRenderer Stopped ");
        }
        ir.requestStop();
      }
    }
  }
}
```

NOTES:

[1] It is important that the scope of the bean in this case matches the intended behavior. Anytime the bean is not found in a Seam context, it will be recreated, causing a new IntervalRenderer to be launched each time, which is not the desired behavior. So, even though this bean doesn't contain any information that cannot be obtained in the EVENT scope, it must be stored in Page (or a really long running Conversation) scope to work as intended. Choosing the appropriate Scope is an important concept in a variety of situations, such as dynamic menu generation. For example, if the backing bean for a dynamic menu was in conversation scope and was created each time a request was handled, this would cause the menu component to have no defined parents during subsequent render passes because the menu component hierarchy returned is not the same one in the rendered view.

[2] The state member variable must be updated inside one of the property methods that is called when the Bean is used in a Render pass. This allows the IntervalRenderer to update its ThreadLocal copy of the state so that it will always be rendering into the correct view. It doesn't matter which getter is used to update the state member variable, since all the getters will be called with the new PersistentFacesState.

[3] It is important to initialize an IntervalRenderer from the application only once.

[4] On Destroy, be sure to clean up the IntervalRenderer if necessary.

In general, as an injected Seam component, the RenderManager reference is only valid during the execution of methods that are intercepted by Seam. Anonymous inner threads do not fall into this category, even if they call methods on the enclosing bean.

Using the File Upload (ice:inputFile) Component

The ICEfaces FileUploadServlet is anomalous in that it does not initiate a JSF lifecycle while it is processing the file upload. While it is processing, the FileUploadServlet is calling the progress method on an object implementing the InputBean interface, to keep the upload progress bar current. In a non-Seam JSF application, these relationships are managed by JSF. However, when running in a Seam application, the InputBean is a Seam component and InputBeanImpl is wrapped by Seam interceptors. This interception of an ad hoc method call from an external bean will fail on return as Seam attempts to clean up contexts that it has not set up. The work-around for this is to turn off Seam interception in the bean declaration, as illustrated in the InputBeanImpl class in the FileUpload example found in the Seam variant of the component showcase demo which is available as a separate download.

Note: The side-effect of turning off this Seam interception is that all Seam features depending on interception (bijection, transactions, annotation support) are disabled for the Bean for which interception is turned off.

For the Seam-1.2.1.GA version of the ICEfaces Component Showcase sample application, we use an inner class in the InputBeanImpl class in the FileUpload example to get around a concurrent access problem when you make the BackingBean generate a render from inside the Bean itself. Concurrent method calls to the same object are not allowed in an EJB container. See the example in the FileUploadServlet. The .war deployment example of jboss-seam-2.0.0.GA looks similar to a regular component-showcase example of fileUpload and takes advantage of the PersistentFacesState in order to update the progressMonitor of the fileUpload component. An IntervalRenderer is used for the example of progressMonitor in this same application.

Spring Framework Integration

The Spring Framework is a full-stack Java/JEE application framework focusing on increased development productivity while improving application testability and quality. For more information on Spring Framework, see **http://www.springframework.org.**

Spring integration with ICEfaces should be considered preliminary only. The Spring Webflow (**http://www.springframework.org/webflow**) has recently released a milestone version 2.0, which is a substantial change to the framework and all integration efforts therein. ICEFaces support for version 1 (currently 1.0.5) of Spring Webflow is therefore preliminary, and not subject for improvement.

Getting Started

The easiest way to produce a working ICEfaces/Seam application is to copy the demonstration application available from **http://downloads.icefaces.org**. This application is based on the original Spring Sellitems demonstration application, and consists of a simple webflow example coupled with ICEfaces components and a demonstration of some pages with enhanced capability using partial submits to access server business logic without full-page transitions. Using the build mechanism in place from the Spring examples allows Ivy to resolve any JAR dependencies. In the demonstration application, a project.properties file contains the directory location of some Springcommon build files. The downloaded demonstration application is intended to go into the following directory:

```
{Spring base directory}/Spring-webflow-1.0.N/projects/spring-webflow-samples
```

where **N** is the version of Spring webflow in use.

Putting the demonstration application in this location allows the build files to work with the existing Spring build resources.

Configuring Spring Applications to Work with ICEfaces

Generally speaking, Spring Webflow applications are about managing state change and managing navigation through the web application. ICEfaces can certainly work in this environment, but ICEfaces brings AJAX technology to web applications simply, without exposure to verbose JavaScript. Using partial submits, you can increase the functionality of a given page and expose more business logic without the need for the same amount of cumbersome full-page navigation states in the application. Any single page can now be enhanced with features such as, autocomplete with values fetched from the Server, or server-based business rules for calculating intermediate costs.

The good news is that requests to the server made by ICEfaces components don't change or affect the state of the current Webflow State as long as the interaction doesn't return a navigation result that is the same as the result defined for the Spring Webflow. This means that you can add a component to the page and go about increasing functionality without worrying about inadvertently changing the application's behavior.

Primarily, the changes to a Spring application to work with ICEfaces consist of:

- Changing the default Spring variable resolver to the following, as per the <variable-resolver>org.springframework.web.jsf.DelegatingVariableResolver</variable-resolver>. This

allows defining beans defined in a webflow to be accessed during a JSF lifecycle. The opposite, defining beans used by Spring in the faces-config file, defeats other aspects of Spring configuration.

- Changing **web.xml** to point to ICEfaces Servlets, and additionally defining a listener that allows access to Session scoped beans. This is required because the entrance into the JSF lifecycle is done through ICEfaces servlets, and not the Spring DispatcherServlet.
- Changing the JSP pages to include the ICEfaces taglib definitions and optionally using ICEfaces components to enhance page functionality.

Changes to web.xml

The following is the **web.xml** file from the downloadable example application:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
     classpath:org/springframework/webflow/samples/sellitem/services-config.xml
     /WEB-INF/webflow-config.xml
    </param-value>
  </context-param>

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

    <!-- Bootstraps the root Spring Web Application Context, responsible for
         deploying managed beans defined in the configuration files above.  These
         beans represent the services used by the JSF application. -->
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
  </listener>



    <!--MyFaces listener that starts up the JSF engine by reading
        faces-config.xml -->
    <listener>
        <listener-class>
            org.apache.myfaces.webapp.StartupServletContextListener
        </listener-class>
    </listener>

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

```
<!-- Listener for initializing contexts in 3rd party servlet environment -->
<listener>
    <listener-class>
        org.springframework.web.context.request.RequestContextListener
    </listener-class>
</listener>

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

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


<context-param>
    <param-name>
        com.icesoft.faces.actionURLSuffix
    </param-name>
    <param-value>.iface</param-value>
</context-param>

<!-- The front controller for the JSF application, responsible for handling all
     application requests -->
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<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>

<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>*.faces</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>

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

    <welcome-file-list>
      <welcome-file>index.html</welcome-file>
    </welcome-file-list>

</web-app>
```

Notes on web.xml

- ICEfaces servlets handle all requests from the browser.
- Synchronous request handling with concurrent DOM views are disabled.

Changes to faces-config.xml

```
<faces-config>
    <application>
    <navigation-handler>
         org.springframework.webflow.executor.jsf.FlowNavigationHandler
    </navigation-handler>
    <variable-resolver>
         org.springframework.web.jsf.DelegatingVariableResolver
    </variable-resolver>

          <!--<variable-resolver>
    org.springframework.webflow.executor.jsf.DelegatingFlowVariableResolver
      </variable-resolver>-->
    </application>

    <lifecycle>
        <phase-listener>
            org.springframework.webflow.executor.jsf.FlowPhaseListener
        </phase-listener>
    </lifecycle>
</faces-config>
```

In this file, be sure to have the appropriate JSF variable resolver specified. See the Spring Framework 2.1 reference Section 15.3, Integrating with JavaServer Faces.

Known Issues

Using server-initiated rendering can cause problems with regard to Webflow states. When a server-initiated render operation starts, the Webflow executor key is retrieved from the UIViewRoot, and the Webflow state is resumed. This is okay, and works properly, but it does cause problems if the Webflow has reached a terminal state. In this case, the Webflow cannot be resumed.

There might be several server-initiated rendering scenarios for your application. Currently, it is difficult to know when a page transition occurs as part of a partial submit; hence, it is difficult to know precisely when to stop the server-initiated rendering if the page transitions to a Webflow terminal state.

Consider an example of an application with an outputText component displaying the time on the bottom of the page every 5 seconds. This type of component would be easy to add to a footer, so it is included on every page of the application. This would cause problems in the Webflow sellitems demonstration application because when the application winds up in the shipping cost summary page, the Webflow has reached a terminal state. If the ticking clock is still updating at this time, this will cause exceptions when it tries to restore the Webflow as part of the server-initiated rendering pass.

The ViewListener and DisposableBean interfaces allow the server-initiated rendering code to know when the user leaves a particular view, but currently these work only if the user leaves the application entirely, or if the user navigates within the application by clicking an anchor tag or by redirection. This mechanism does not work if the user submits a postback and the view is changed by evaluating a navigation rule.

It is best to avoid server-initiated rendering within Spring Webflow applications for now. Once we understand the ramifications of Webflow 2.0, we'll be able to make using server-initiated rendering easier. In the meantime, using ICEfaces components with simple AJAX functionality should offer plenty of design flexibility.

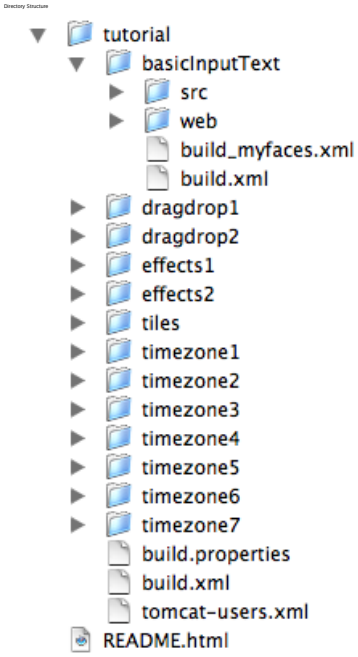
ICEfaces Tutorial: Creating Direct-to-DOM Renderers for Custom Components

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 17. 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 17 Direct-to-DOM Rendering



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

F

G

H

I

J

K

L

M

P

R

S

T

V

W

X