ICEfaces

Getting Started Guide

Version 1.8

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 v1.8

March 2009

About this Guide

The **ICEfaces® Getting Started Guide** will help you quickly get started building ICEfaces applications. By reading through this guide, you will:

- Gain a basic understanding of what ICEfaces is and what it can do for you.
- Install ICEfaces and run the sample applications on your local application server.
- Work through a basic ICEfaces tutorial that transforms a standard JSF application into a rich web application powered by ICEfaces.
- Understand how to integrate ICEfaces with various JEE application servers and development environments.

For more information about ICEfaces, visit the ICEfaces Community web site at:

http://www.icefaces.org

In this guide...

We have organized this guide into these chapters to help you get started using ICEfaces quickly:

- **Chapter 1: Introduction to ICEfaces** — Provides an overview of ICEfaces and its key features and capabilities.
- **Chapter 2: Configuring Your Environment for ICEfaces** — Describes how to set up the ICEfaces environment on your local Tomcat server to run the sample applications and tutorial included with the installation.
- **Chapter 3: Running the ICEfaces Sample Applications** — Highlights the features and capabilities that ICEfaces technology provides through several sample applications.
- **Chapter 4: ICEfaces Tutorial: The TimeZone Application** — Demonstrates how to transform a standard JSF application into a rich interactive ICEfaces application through a series of tutorial exercises. Each exercise in the tutorial illustrates a key capability of ICEfaces.
- **Chapter 5: ICEfaces SessionRenderer Tutorial: Easy Ajax Push** — Demonstrates how to easily add Ajax Push collaborative features to your rich web application
- **Chapter 6: ICEfaces Design Decisions — Using ICEfaces in Your Project** — Provides details for setting up other development environments to use ICEfaces.

Prerequisites

ICEfaces applications are JavaServer Faces (JSF) applications, and as such, the only prerequisite to working with ICEfaces is that you must be familiar with JSF application development. For more information on Java Platform, Enterprise Edition (JEE), which JSF is a sub-component of, refer to **http://java.sun.com/javaee/**.

Additional JSF resources can be found on the **icefaces.org** website.

ICEfaces Documentation

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

- **ICEfaces Release Notes** — Contains information about the new features and bug fixes included in this ICEfaces release. In addition, important time-saving **Known Issues** and detailed information about supported browsers, application servers, portal containers, and IDEs can also be found in this document.
- **ICEfaces Developer's Guide** — Includes materials targeted for ICEfaces application developers and includes an in-depth discussion of the ICEfaces architecture and key concepts, as well as reference material related to markup, APIs, components, and configuration.

ICEfaces Technical Support

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

http://support.icefaces.org/

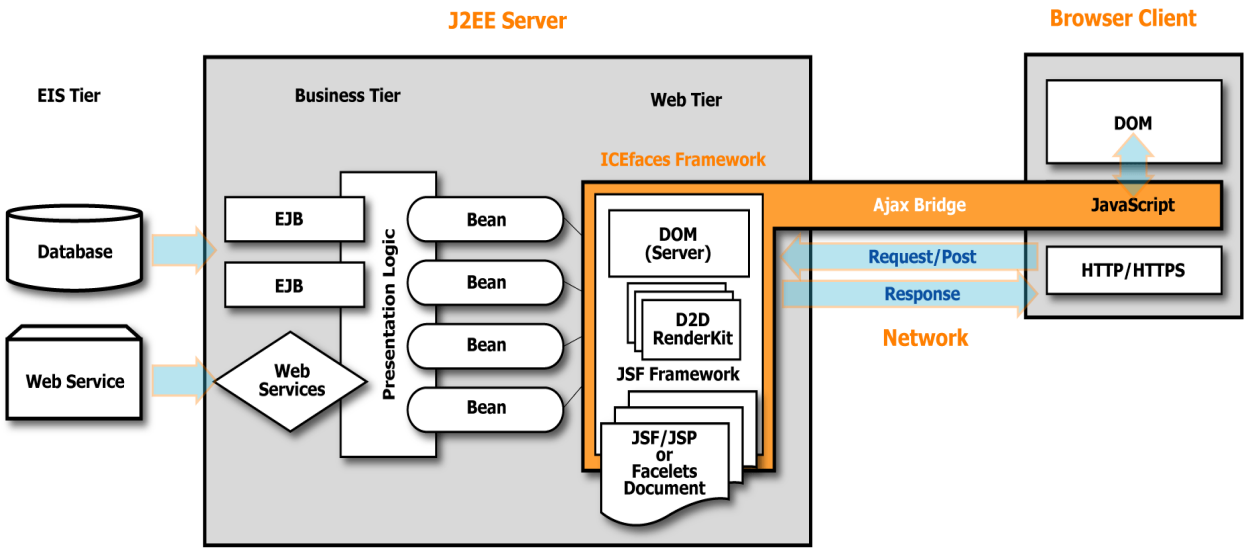
Contents

Chapter 1 Introduction to ICEfaces

ICEfaces® is the industry's leading open-source, standards-compliant Ajax-based solution for rapidly creating enterprise grade, pure-Java rich web applications.

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 limits form processing to the single control that has been altered, but allows the application lifecycle to execute in response to that change. This means that the application developer can introduce intelligent form processing logic that reacts instantaneously to user interactions with the form.
- **Server-initiated asynchronous presentation update (Ajax Push).** Standard JSF applications can only deliver presentation changes in response to a user-initiated event, typically some type of form submit. ICEfaces introduces a server-initiated rendering that enables 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 Configuring Your Environment for ICEfaces

This chapter contains instructions to help you get up and running quickly with ICEfaces technology. We start by outlining the prerequisites for a standard configuration using a Java Platform, Standard Edition (J2SE), Tomcat, and Ant to help build and deploy the ICEfaces sample applications and tutorials.

If you would like to run the sample applications or the tutorial in your chosen development environment, or with a different application server, refer to **Chapter 6, ICEfaces Design Decisions — Using ICEfaces in Your Project,** on page 62.

Prerequisites

ICEfaces applications are JavaServer Faces (JSF) applications, and as such, the only prerequisite to working with ICEfaces is that you must be familiar with JSF application development. For more information on Java Platform, Enterprise Edition (JEE), which JSF is a sub-component of, refer to **http://java.sun.com/javaee/**.

Additional JSF resources can be found on the **icefaces.org** website.

To run the sample ICEfaces applications, you will need to download and install the following:

- Java 2 Platform, Standard Edition
- Ant
- Tomcat
- ICEfaces
- Web browser

The following sections provide detailed instructions for downloading the software to set up an environment where you can run the ICEfaces sample applications and tutorial.

Java 2 Platform, Standard Edition

To run the ICEfaces sample applications with Tomcat, you will need to install a version of the Java Platform, Enterprise Edition (JEE) Platform, version 1.4.2 or higher.

If you already have Java installed on your system, verify your version by typing the following on the command line:

```
java -version
```

To upgrade or install the latest release of the J2SE, visit the Sun web site:

http://java.sun.com/downloads/index.html

Installers and instructions are provided for the various systems that Sun supports. The demo applications can be run on any version of Windows, Linux, and Mac OS X capable of running J2SE version 1.4.2 or higher.

Ant

The ICEfaces tutorial relies on Ant to build and deploy the various stages of the tutorial application. You will need Ant version 1.6.3 or higher for the build files provided in this ICEfaces release.

If you already have a version of Ant installed, you can verify that you have a recommended version by typing the following on a command line:

```
ant -version
```

To upgrade your current version or install a new version of Ant, visit the following location:

http://ant.apache.org/

If you are not familiar with Ant, detailed instructions for downloading and installing Ant for your environment are available in the online manual at:

http://ant.apache.org/manual/index.html

Tomcat

Java web applications require an appropriate JEE runtime environment. ICEfaces applications require support for servlets and JavaServer Pages (JSP). Tomcat is a popular choice for this type of development because the ICEfaces code has been extensively tested on Tomcat.

Tomcat is available from the Apache Jakarta Project at:

http://jakarta.apache.org/tomcat/

Download and install Tomcat 6 according to the instructions provided with the software. Although it is possible to run ICEfaces applications in any standard JEE container, all the instructions provided in this guide refer to Tomcat. In addition, the default ant build for the sample applications targets Tomcat 6.

Once Tomcat is successfully installed, follow the instructions to start the server. This will differ depending on what platform you are using.

Web Browser

Web applications use a web browser as the client. This ICEfaces distribution has been verified with the following browsers:

Vendor	Product	Version
Apple	Safari	1.3, 2.x, 3.x, 4.x
Google	Chrome	1.0
Microsoft	Internet Explorer	6.x+, 7.0, 8.0
Mozilla	Firefox	1.x+, 2.0, 3.0
Opera	Opera	9.x+

ICEfaces

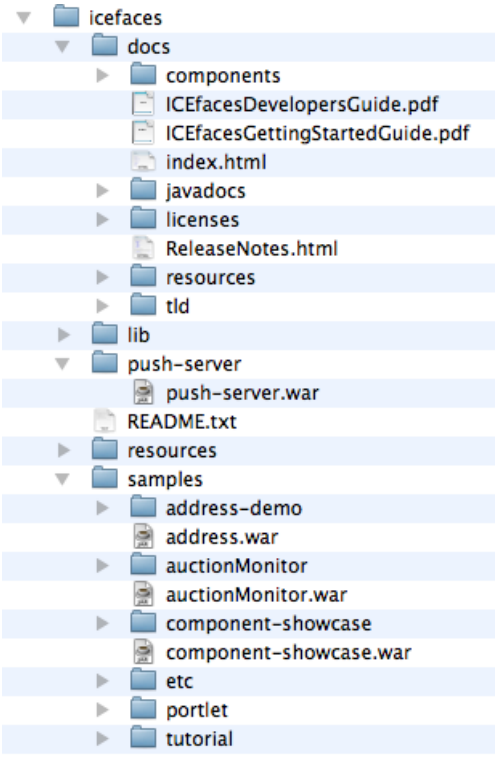
If you are reading this document, you may have already downloaded and installed ICEfaces. If you haven't, you can get the latest version of ICEfaces from:

http://www.icefaces.org

ICEfaces is available in both a binary and source-code distribution. With either distribution, begin by unzipping ICEfaces to your preferred location.

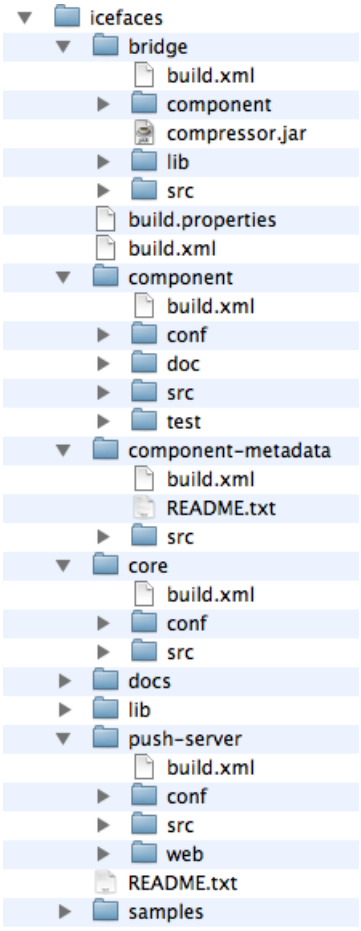
If you downloaded the binary distribution, the resulting directory structure should look similar to the structure shown in Figure 2.

Figure 2 ICEfaces Directory Structure



If you downloaded the source distribution these additional source-code directories will also be present; bridge, component, and core (shown in Figure 3).

Figure 3 ICEfaces Source Distribution Directory Structure



Chapter 3 Running the ICEfaces Sample Applications

ICEfaces is bundled with three sample applications: Auction Monitor, Address Form, and Component Showcase.

If you have downloaded the binary distribution of ICEfaces, these applications are available in prebuilt .war files suitable for deployment to Tomcat 6 in the following location:

```
[install_dir]/icefaces/samples/
```

If you downloaded the source code distribution of ICEfaces, it is necessary to build the samples which can be accomplished by running ant from the `[install_dir]/icefaces/` directory. This will build all of the sample bundles and create distributable .war files suitable for deployment to Tomcat 6 in the distribution directory at:

```
[install_dir]/icefaces/dist/samples/
```

If you would like to deploy the sample applications to an application server other than Tomcat 6, you will need to build the .war file appropriate for the target application server. The ant build script provides additional build targets for many popular JEE application servers. Navigate to the directory for a particular sample application and type **`ant help`** to see a list of available build targets. Then build the sample application for the application server target of your choice.

For example, **`ant glassfishv2`** creates a .war file deployable to a Glassfish v2 application server.

> **Note:** Individual application servers may require a different set of libraries in the application .war file, depending on which libraries they provide themselves; thus, it is not feasible to deploy the same .war file to all application servers. For details on which libraries are required by various application servers, see the **Library Dependencies** section in **Chapter 4, ICEfaces Reference Information** of the **ICEfaces Developer's Guide**.

If you would like to build a sample for an application server for a specific build target that is not provided, you can use custom build options to create a build that will deploy correctly on your application server. The **`ant help`** command provides more information on the available options.

If you are working with Tomcat, the quickest and easiest way to deploy a .war file is to copy the file into the webapps directory of the Tomcat installation. By default, Tomcat periodically checks this directory for updates and, if it finds a new .war file, it automatically deploys the application. Once you've copied the .war file into webapps and Tomcat has deployed the application, you can view and interact

with it by opening a web browser and typing in the appropriate URL for the application that you want to browse.

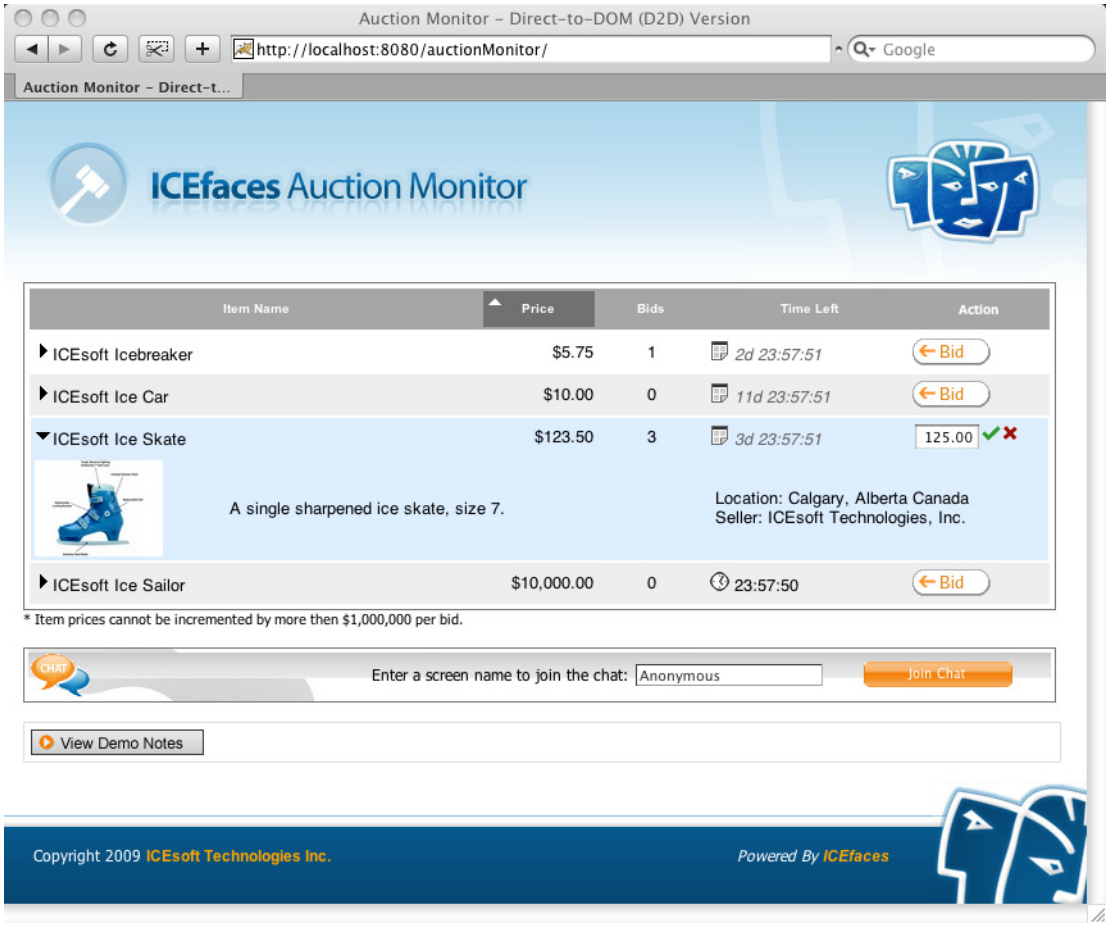
Application	Archive	URL
AuctionMonitor	auctionMonitor.war	`http://localhost:8080/auctionMonitor/`
AddressForm	address.war	`http://localhost:8080/address/`
Component Showcase	component-showcase.war	`http://localhost:8080/component-showcase/`

The sample applications highlight the various features and capabilities that ICEfaces technology provides.

AuctionMonitor

ICEfaces **AuctionMonitor** (auctionMonitor.war) simulates the tracking of live auctions with ticking countdown timers and interactive bidding. It also includes a simple integrated chat and some demo notes on how to interact with the AuctionMonitor.

Figure 4 AuctionMonitor Sample Application



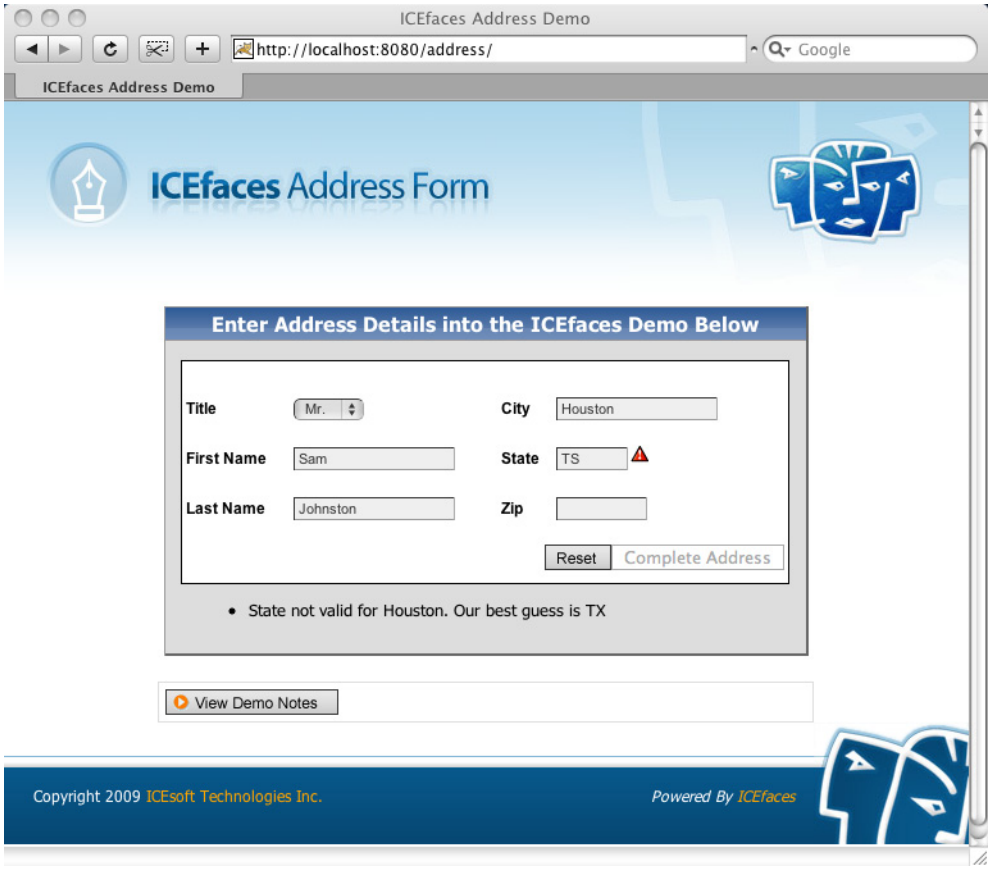
The application uses a number of standard JSF components: dataTable, commandButton, and panelGrid. ICEfaces and Direct-to-DOM rendering provides for asynchronous and dynamic updates of these components without a full page refresh. This is illustrated by the ticking clocks, real-time bid updates, the dynamically rendered buttons, the ability to hide and show table rows on demand, and the integrated chat session. For more information on these ICEfaces features, refer to the **ICEfaces Developer's Guide**.

Open the AuctionMonitor in two browser windows to fully appreciate the interactive features of this application.

AddressForm

ICEfaces **AddressForm** (address.war) shows how a prototypical address form can be validated and updated on the fly using partial submits and without fully refreshing the page.

Figure 5 AddressForm Sample Application

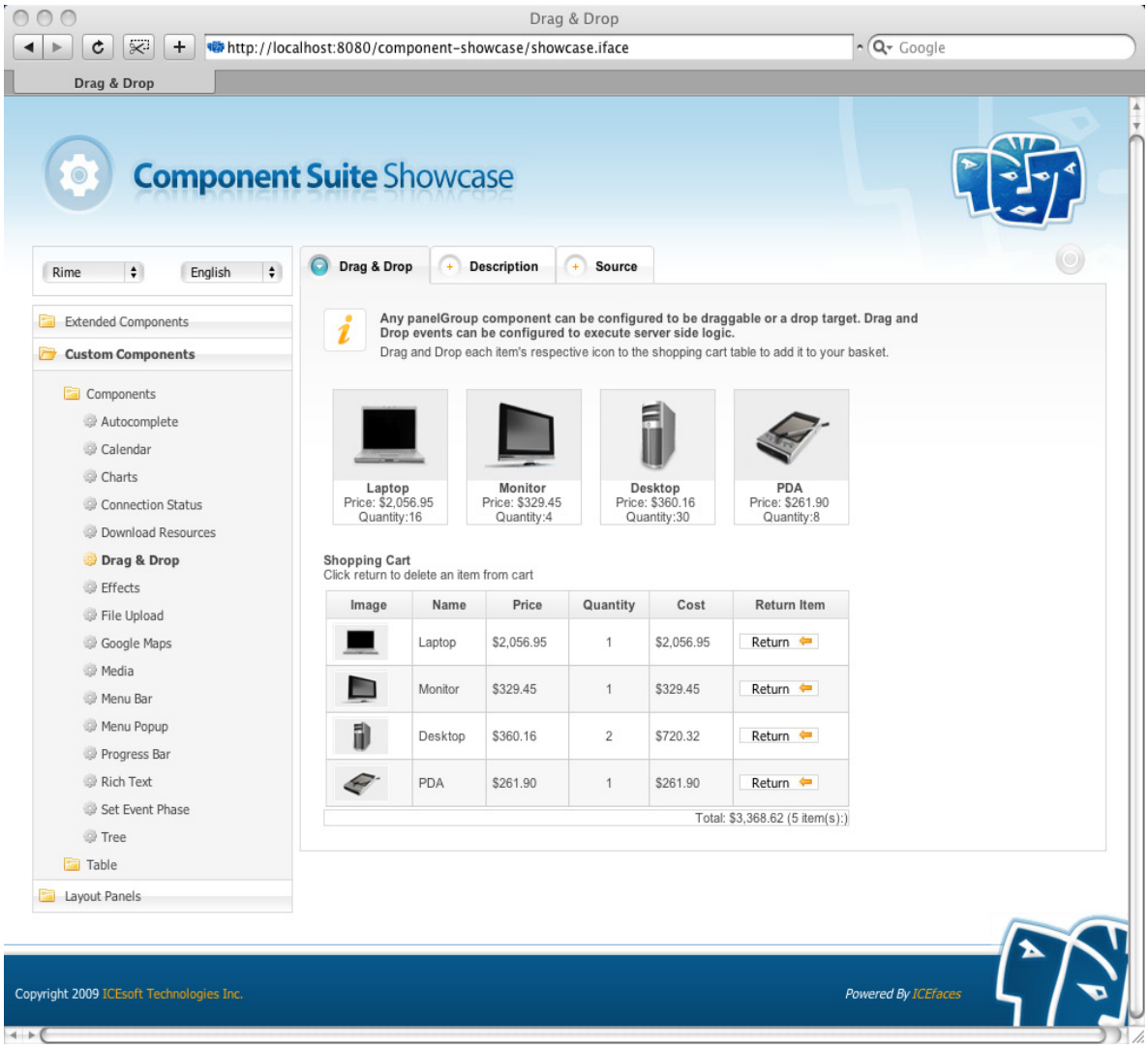


Typically, this type of form would require a user to fill in all the fields and submit it for server-side validation. That, or a raft of JavaScript code to do client-side validation. However, using JavaServer Faces components and ICEfaces, you can do a partial submit with each field entry and only offer the submit button when the form is completely validated. As each field is entered, a partial form submission is sent to the server and only those components that require updates are changed. It's rich, interactive, efficient, and lightweight. Most importantly, it doesn't require any JavaScript programming.

Component Showcase

The **Component Showcase** sample application (component-showcase.war) demonstrates key features and capabilities of the ICEfaces Component Suite. It consists of a collection of small examples intended to aid users in the evaluation of ICEfaces features and to provide developers with simple, straight-forward examples of component usage. Use the Component Showcase to quickly see an overview of the capabilities of the ICEfaces Component Suite and as a reference to each component's properties, documentation, tutorials, and other resources.

Figure 6 Component Showcase



The Component Showcase application is available in three distinct varieties: a JSP, Facelets, and Portlets implementation. Each variety illustrates how the ICEfaces Component Suite can be used within that environment.

- **Facelets**

  Uses dynamic includes and Facelets templating for the navigation and structure of the application. This is the default implementation used in the prebuilt component-showcase.war file included with the binary distribution. Also includes comprehensive resource navigation and source code viewing capabilities.

- **JSP**

  Simple JSP navigation system which has a link for every component example. The main layout construct is a panel stack which is manipulated by the navigation system.

- **Portlets**

  Provides a portlet implementation of the component samples with each sample being a single portlet. Build targets are available for many supported portlet containers.

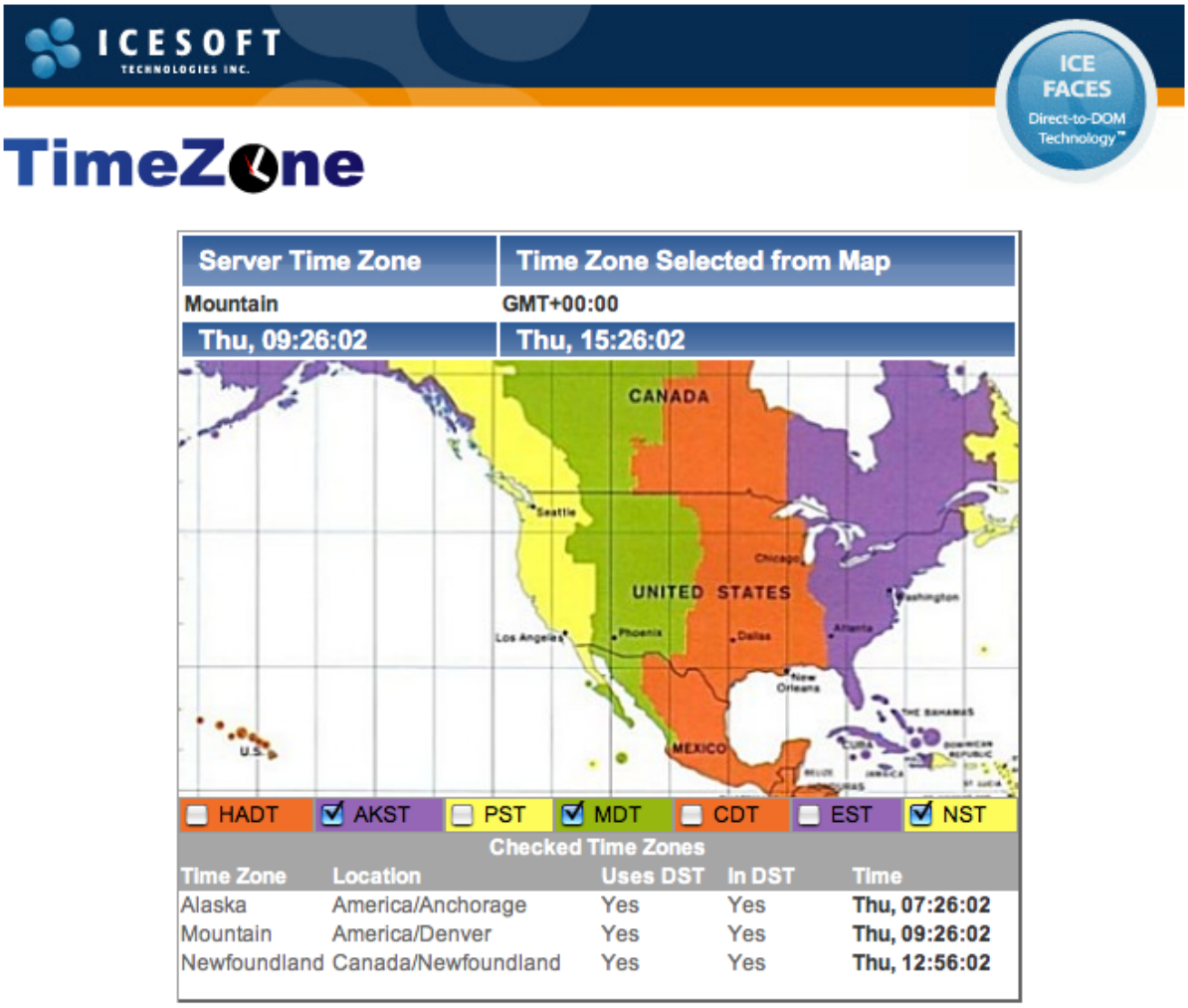
The source-code for each variety of the application is located under `[install_dir]/icefaces/samples/component-showcase2/facelets`, `[install_dir]/icefaces/samples/component-showcase2/jsp`, and `[install_dir]/icefaces/samples/component-showcase2/portlets`.

To build each variety use **`ant clean tomcat6.0`** from within one of the Facelet, JSP, or Portlet subdirectories. Use **`ant help`** to see a list of all available build targets (glassfishv2, jboss4.2, etc.).

Chapter 4 ICEfaces Tutorial: The TimeZone Application

This tutorial guides you through a series of exercises that takes a basic JavaServer Faces application and transforms it, using ICEfaces, into a much more dynamic application with an enriched user experience. The tutorial begins with a simple web application called TimeZone and demonstrates how to build the ICEfaces-enriched application shown in Figure 7.

Figure 7 ICEfaces TimeZone Application



This tutorial consists of seven steps with increasing levels of richness, each designed to demonstrate ICEfaces capabilities:

Step 1 – Basic JSF TimeZone Application shows a basic JSF application built entirely from basic JSF HTML components in JSP.

Step 2 – Integrating ICEfaces uses the same basic JSF application, converted to JSP Document, running with ICEfaces.

Step 3 – Dynamic Updating—Make the Clocks Tick modifies the application to add ticking clocks and to support separate but concurrent instances of the TimeZone application from multiple web browser windows or tabs.

Step 4 – Dynamic Table Rendering adds a dynamic data table, which can be manipulated by checkboxes, to make the TimeZone application more interactive.

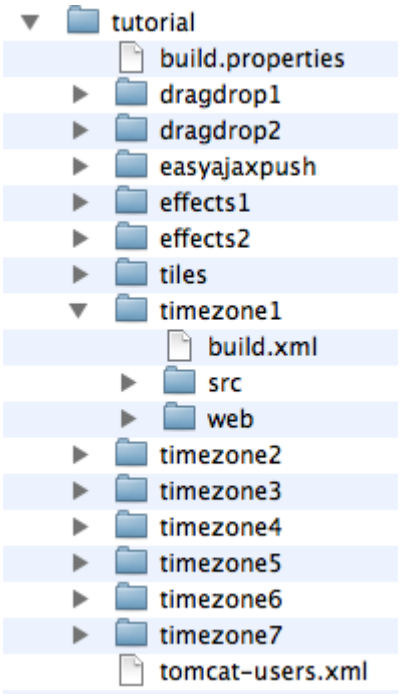
Step 5 – Applying Styles demonstrates how to add CSS formatting and styles to enhance the appearance of the TimeZone application.

Step 6 – Integrating Facelets uses the same JSF application, converted to XHTML, running with ICEfaces and Facelets together.

Step 7 – Capitalize on Facelets uses Facelets functionality to do header inclusion and dynamically generate content from the Java bean.

The seven exercises are organized in the directory structure shown in Figure 8. Prior to starting this tutorial, we recommend that you read **Chapter 2, Configuring Your Environment for ICEfaces,** on page 3 to ensure that your environment is configured properly to deploy and execute ICEfaces applications on your JEE application server.

Figure 8 TimeZone Tutorial Directory Structure



Overview of the TimeZone Application

By following all the steps in this tutorial, you will build an ICEfaces application called TimeZone (shown in Figure 7 on page 13) from source code and deploy it to your local Tomcat server.

The application is not entirely an accurate representation of all time zones represented on the map, but serves as an effective example of how to enrich a web application with ICEfaces. The application was deliberately simplified to demonstrate the features that ICEfaces provides to improve the richness and interactivity of a web application.

The completed ICEfaces TimeZone application has the following features:

- The current time, using the time zone of the application server, is displayed in the top left of the table.
- To the right of the server's current time is a second time display for an alternate time zone, which can be chosen by clicking on any section of the map.
- Below the map are checkboxes that, when selected, add time zone details to the table at the bottom. Deselecting a checkbox removes the details from the table.

Step 1 – Basic JSF TimeZone Application

The first step is to create a regular JavaServer Faces (JSF) version of the TimeZone web application using stock JSF components. All the files for this part of the tutorial are in the **timezone1** directory.

Creating a JSP Page with Standard JSF and HTML Tags

Our first iteration of the TimeZone application (see Figure 9 below) has a panelGrid component at the top to hold the two separate time displays:

- the application's host server time, and
- the time for the zone selected from the map.

A commandButton is used to display a map which contains seven timezone regions. When a region of the map is clicked, the display at the top right updates to show the selected region's time and timezone.

> **Note:** Throughout this tutorial, we use **boldface** text to highlight code we want to bring to your attention. ICEfaces does not require any code to be typed in bold.

Figure 9 TimeZone Application as Stock JSF Application

ICEfaces: TimeZone Sample Application

Server Time Zone	Time Zone Selected from Map
Mountain	Eastern
Thu, 09:35:15	**Thu, 11:35:15**

The code for the timezone.jsp page is as follows:

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
  <html>
    <head><title>ICEfaces: TimeZone Sample Application</title></head>
    <body>
      <h3>ICEfaces: TimeZone Sample Application</h3>
      <h:form>
        <h:panelGrid columns="2">
          <h:outputText style="font-weight:600" value="Server Time Zone"/>
          <h:outputText style="font-weight:600" value="Time Zone Selected from Map"/>
          <h:outputText value="#{timeZoneBean.serverTimeZoneName}"/>
          <h:outputText value="#{timeZoneBean.selectedTimeZoneName}"/>
          <h:outputText style="font-weight:800" value="#{timeZoneBean.serverTime}"/>
          <h:outputText style="font-weight:800"
                        value="#{timeZoneBean.selectedTime}"/>
        </h:panelGrid>
           <h:commandButton id="map" image="images/map.jpg"
                actionListener="#{timeZoneBean.listen}" />
      </h:form>
    </body>
  </html>
</f:view>
```

Most of the components are dynamically bound to backing JavaBeans through JSF expression language bindings as shown below:

```
<h:outputText value="#{timeZoneBean.serverTimeZoneName}"/>
```

Creating the Backing JavaBean (TimeZoneBean.java)

The **com.icesoft.faces.tutorial.TimeZoneBean** class is the backing bean for the timezone.jsp page. The bean stores the current state of the selections and all the time zone information.

The code for the TimeZoneBean.java class is as follows:

```
/*
 * Version: MPL 1.1/GPL 2.0/LGPL 2.1
 *
 * "The contents of this file are subject to the Mozilla Public License
 * Version 1.1 (the "License"); you may not use this file except in
 * compliance with the License. You may obtain a copy of the License at
 * http://www.mozilla.org/MPL/
 *
 * Software distributed under the License is distributed on an "AS IS"
 * basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the
 * License for the specific language governing rights and limitations under
 * the License.
 *
 * The Original Code is ICEfaces 1.5 open source software code, released
 * November 5, 2006. The Initial Developer of the Original Code is ICEsoft
 * Technologies Canada, Corp. Portions created by ICEsoft are Copyright (C)
 * 2004-2006 ICEsoft Technologies Canada, Corp. All Rights Reserved.
 *
 * Contributor(s): _____.
 *
 * Alternatively, the contents of this file may be used under the terms of
 * the GNU Lesser General Public License Version 2.1 or later (the "LGPL"
 * License), in which case the provisions of the LGPL License are
 * applicable instead of those above. If you wish to allow use of your
 * version of this file only under the terms of the LGPL License and not to
 * allow others to use your version of this file under the MPL, indicate
 * your decision by deleting the provisions above and replace them with
 * the notice and other provisions required by the LGPL License. If you do
 * not delete the provisions above, a recipient may use your version of
 * this file under either the MPL or the LGPL License."
 *
 */

package com.icesoft.tutorial;

import javax.faces.context.FacesContext;
import javax.faces.event.ActionEvent;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Map;
import java.util.TimeZone;

/**
 * Bean backing the Time Zone application. Also controls time zone information
 * during the session.
 */

public class TimeZoneBean {
```

```
/**
 * The default {@link TimeZone} for this host server.
 */
private TimeZone serverTimeZone;

/**
 * {@link DateFormat} used to display the server time.
 */
private DateFormat serverFormat;

/**
 * Active {@link TimeZone} displayed at top of UI. Changes when a time zone
 * is selected by pressing one of six commandButtons in UI map.
 */
private TimeZone selectedTimeZone;

/**
 * {@link DateFormat} used to display the selected time.
 */
private DateFormat selectedFormat;

/**
 * List of all possible {@link TimeZoneWrapper} objects, which must mirror
 * the map UI.
 */
private ArrayList allTimeZoneList;


/**
 * Constructor initializes time zones.
 */
public TimeZoneBean() {
    init();
}

/**
 * Initializes this TimeZoneBean's properties.
 */
private void init() {
    serverTimeZone = TimeZone.getDefault();
    serverFormat = buildDateFormatForTimeZone(serverTimeZone);
    // selected time zone set to UTC as default
    selectedTimeZone = TimeZone.getTimeZone("Etc/GMT+0");
    selectedFormat = buildDateFormatForTimeZone(selectedTimeZone);

    // Entries in this list are hardcoded to match entries in
    // the timezone web file, so no parameters can be changed.
    allTimeZoneList = new ArrayList(7);
    allTimeZoneList
            .add(new TimeZoneWrapper("Pacific/Honolulu", "GMTminus10",
                                     hawaiiXCoords, hawaiiYCoords,
                                     hawaiiXCoords.length));
    allTimeZoneList
            .add(new TimeZoneWrapper("America/Anchorage", "GMTminus9",
                                     alaskaXCoords, alaskaYCoords,
                                     alaskaXCoords.length));
    allTimeZoneList
```

```
                .add(new TimeZoneWrapper("America/Los_Angeles", "GMTminus8",
                                         pacificXCoords, pacificYCoords,
                                         pacificXCoords.length));
        allTimeZoneList
                .add(new TimeZoneWrapper("America/Denver", "GMTminus7",
                                         mountainXCoords, mountainYCoords,
                                         mountainXCoords.length));
        allTimeZoneList
                .add(new TimeZoneWrapper("America/Chicago", "GMTminus6",
                                         centralXCoords, centralYCoords,
                                         centralXCoords.length));
        allTimeZoneList
                .add(new TimeZoneWrapper("America/New_York", "GMTminus5",
                                         easternXCoords, easternYCoords,
                                         easternXCoords.length));
        allTimeZoneList
                .add(new TimeZoneWrapper("Canada/Newfoundland", "GMTminus4",
                                         nfldXCoords, nfldYCoords,
                                         nfldXCoords.length));
    }

    /**
     * Gets server time.
     *
     * @return Server time.
     */
    public String getServerTime() {
        return formatCurrentTime(serverFormat);
    }

    /**
     * Gets server time zone display name.
     *
     * @return Server time zone display name.
     */
    public String getServerTimeZoneName() {
        return displayNameTokenizer(serverTimeZone.getDisplayName());
    }

    /**
     * Gets selected time zone time. This is the time zone selected by one of
     * six commandButtons from the map in the UI.
     *
     * @return selectedTimeZone time.
     */
    public String getSelectedTime() {
        return formatCurrentTime(selectedFormat);
    }

    /**
     * Gets selected time zone display name.
     *
     * @return selectedTimeZone display name.
     */
    public String getSelectedTimeZoneName() {
        return displayNameTokenizer(selectedTimeZone.getDisplayName());
    }
```

```
/**
 * Extracts the first word from a TimeZone displayName.
 *
 * @param displayName A TimeZone displayName.
 * @return String The first word from the TimeZone displayName.
 */
public static String displayNameTokenizer(String displayName) {
    if (displayName == null) {
        displayName = "";
    } else {
        int firstSpace = displayName.indexOf(' ');
        if (firstSpace != -1) {
            displayName = displayName.substring(0, firstSpace);
        }
    }
    return displayName;
}

public static DateFormat buildDateFormatForTimeZone(TimeZone timeZone) {
    SimpleDateFormat currentFormat = new SimpleDateFormat("EEE, HH:mm:ss");
    Calendar currentZoneCal = Calendar.getInstance(timeZone);
    currentFormat.setCalendar(currentZoneCal);
    currentFormat.setTimeZone(timeZone);
    return currentFormat;
}

public static String formatCurrentTime(DateFormat dateFormat) {
    Calendar cal = dateFormat.getCalendar();
    cal.setTimeInMillis(System.currentTimeMillis());
    return dateFormat.format(cal.getTime());
}

/**
 * Each TimeZoneWrapper has an id of a component in the UI that corresponds
 * to its time zone.  By this, if an event comes from a component in the web
 * page, then this will return the relevant TimeZoneWrapper.
 *
 * @param componentId Id of component in UI
 * @return TimeZoneWrapper
 */
private TimeZoneWrapper getTimeZoneWrapperByComponentId(
        String componentId) {
    TimeZoneWrapper tzw;
    for (int i = 0; i < allTimeZoneList.size(); i++) {
        tzw = (TimeZoneWrapper) allTimeZoneList.get(i);
        if (tzw.isRelevantComponentId(componentId)) {
            return tzw;
        }
    }
    return null;
}

//
// Implicit interfaces as defined by the callbacks in the web files
//
```

```
/**
 * Listens to client input from the commandButton in the UI map and sets the
 * selected time zone.
 *
 * @param event ActionEvent.
 */
public void listen(ActionEvent event) {

    FacesContext context = FacesContext.getCurrentInstance();
    String clientId = event.getComponent().getClientId(context);
    Map requestParams =
            context.getExternalContext().getRequestParameterMap();
    // get mouse coordinate of user click
    int x = Integer.parseInt((String) requestParams.get(clientId + ".x"));
    int y = Integer.parseInt((String) requestParams.get(clientId + ".y"));

    for (int i = 0; i < allTimeZoneList.size(); i++) {
        if (((TimeZoneWrapper) allTimeZoneList.get(i)).getMapPolygon()
                .contains(x, y)) {
            TimeZoneWrapper tzw = (TimeZoneWrapper) allTimeZoneList.get(i);
            selectedTimeZone = TimeZone.getTimeZone(tzw.getId());
            selectedFormat = buildDateFormatForTimeZone(selectedTimeZone);
        }
    }

}

// Create primary polygon objects for continental country outlines
private static int[] hawaiiXCoords = {0, 29, 54, 58, 58, 61, 61, 0};
private static int[] hawaiiYCoords =
        {186, 194, 208, 215, 223, 243, 254, 254};

private static int[] alaskaXCoords =
        {117, 118, 125, 132, 135, 138, 141, 146, 147, 157, 164, 165, 162,
         156, 144, 120, 75, 72, 60, 45, 1, 0, 0, 14};
private static int[] alaskaYCoords =
        {0, 4, 5, 12, 12, 8, 7, 14, 14, 28, 31, 37, 38, 41, 41, 16, 16, 25,
         35, 38, 55, 55, 1, 0};

private static int[] pacificXCoords =
        {176, 176, 187, 187, 181, 185, 191, 192, 207, 207, 214, 214,
         218, 222, 222, 221, 221, 222, 224, 230, 229, 225,
         222, 219, 220, 218, 214, 214, 219, 107, 219, 232,
         231, 230, 228, 228, 229, 228, 226, 226, 229, 231,
         238, 233, 226, 217, 205, 198, 195, 197, 194, 187,
         188, 189, 190, 186, 169, 152, 145, 158, 164, 164,
         155, 141, 136, 134, 132, 125, 118, 118};
private static int[] pacificYCoords =
        {0, 3, 3, 7, 7, 20, 19, 25, 32, 43, 47, 50, 54, 59, 64, 68, 67,
         71, 71, 80, 86, 90, 92, 89, 90, 93, 95, 97, 106,
         107, 112, 112, 137, 139, 140, 148, 149, 157, 158,
         162, 163, 171, 179, 179, 171, 154, 148, 138, 133,
         130, 130, 118, 114, 103, 88, 77, 61, 54, 41, 41,
         37, 32, 25, 7, 9, 11, 11, 4, 4, 0};

private static int[] mountainXCoords =
        {177, 287, 287, 268, 268, 258, 259, 249, 249, 254, 254,
```

```
            250, 253, 250, 253, 254, 250, 250, 277, 277, 284,
            289, 288, 290, 290, 285, 286, 281, 281, 272, 270,
            265, 258, 256, 256, 264, 263, 268, 269, 275, 276,
            272, 244, 216, 217, 231, 218, 226, 232, 239, 230,
            228, 230, 230, 228, 229, 231, 233, 220, 220, 215,
            215, 221, 220, 223, 225, 231, 231, 225, 222, 222,
            215, 215, 208, 208, 193, 192, 185, 182, 182, 189,
            189};
    private static int[] mountainYCoords =
            {0, 0, 8, 8, 45, 45, 41, 40, 48, 48, 52, 52, 63, 63, 63, 69, 69,
            75, 75, 80, 81, 86, 98, 101, 110, 116, 137, 138,
            160, 160, 164, 161, 161, 165, 181, 190, 194, 198,
            201, 204, 210, 214, 233, 231, 216, 208, 176, 174,
            179, 179, 163, 158, 153, 147, 144, 140, 139, 109,
            110, 106, 104, 98, 93, 89, 91, 93, 88, 84, 77, 70,
            61, 53, 48, 44, 41, 30, 24, 16, 17, 8, 7, 2};

    private static int[] centralXCoords =
            {288, 317, 314, 314, 321, 325, 330, 340, 336, 336, 338, 346,
            348, 349, 350, 351, 347, 347, 357, 356, 358, 357,
            352, 378, 380, 381, 291, 291, 269, 277, 276, 269,
            264, 264, 257, 257, 260, 267, 270, 273, 283, 282,
            287, 286, 292, 291, 289, 290, 284, 277, 277, 252,
            252, 255, 255, 252, 255, 255, 251, 251, 256, 256,
            270, 270, 289};
    private static final int[] centralYCoords =
            {0, 0, 9, 15, 15, 26, 25, 30, 35, 74, 86, 89, 94, 111, 113, 115,
            118, 129, 137, 145, 155, 170, 210, 242, 243, 252,
            252, 241, 217, 210, 203, 198, 193, 189, 179, 164,
            162, 163, 165, 163, 162, 139, 138, 116, 109, 98,
            97, 85, 78, 78, 72, 73, 69, 69, 62, 54, 52, 47, 46,
            41, 43, 47, 47, 9, 9};

    private static int[] easternXCoords =
            {388, 417, 446, 446, 449, 447, 448, 447, 450, 449, 442, 438,
            431, 437, 437, 446, 447, 449, 449, 450, 451, 441,
            429, 430, 433, 433, 435, 424, 419, 415, 415, 463,
            464, 382, 381, 353, 360, 357, 359, 348, 348, 353,
            349, 348, 340, 337, 337, 341, 373};
    private static int[] easternYCoords =
            {0, 0, 5, 10, 16, 19, 21, 28, 36, 41, 40, 38, 44, 52, 57, 57, 51,
            50, 54, 66, 74, 78, 81, 83, 84, 91, 96, 209, 215,
            216, 226, 242, 255, 253, 241, 208, 154, 144, 136,
            127, 118, 114, 110, 92, 84, 78, 35, 29, 9};
    private static int[] nfldXCoords =
            {448, 465, 465, 415, 416, 418, 434, 434, 434, 432, 433, 436, 452,
            452, 450, 453, 450, 447, 447, 444, 440, 440, 434, 440, 443, 450,
            453, 450, 451, 448, 450, 450, 448};
    private static int[] nfldYCoords =
            {0, 0, 242, 242, 226, 217, 209, 95, 85, 83, 80, 82, 75, 56, 54, 51,
            50, 52, 56, 54, 55, 50, 45, 40, 42, 42, 37, 29, 23, 20, 17, 14,
            14};

}
```

TimeZoneBean stores the current state and time zone information, and also handles the actions generated by clicking the commandButtons. The **listen(ActionEvent event)** method in the bean takes the ID of the commandButton clicked, and uses the helper TimeZoneWrapper objects to determine which TimeZone ID should be used to instantiate the **selectedTimeZone** object. For clarity, the commandButton IDs represent the offset from Greenwich Mean Time (GMT) of their respective TimeZone objects. The IDs can be arbitrary, as long as they are all unique in the web application, and match between the web file and the Java event handler.

Binding the Bean to the JSP Page

JSF uses an XML file called faces-config.xml to manage the configuration of beans so that the beans' methods are available to components in the page.

The code for the faces-config.xml file is as follows:

```
<?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 xmlns="http://java.sun.com/JSF/Configuration">

  <managed-bean>
      <managed-bean-name>timeZoneBean</managed-bean-name>
      <managed-bean-class>com.icesoft.tutorial.TimeZoneBean</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>

</faces-config>
```

The entry bean is given a name by which it can be referred to in the JSP page and the name is also associated with a Java class so that it can be instantiated as required. The bean is also given a scope to indicate how the bean can be accessed.

Configuring the Web Application

At the heart of it, the TimeZone application is a standard JEE web application that requires a deployment descriptor. This means we need to create a web.xml file.

The code for the web.xml file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

  <display-name>ICEfaces Tutorial: Timezone Part 1</display-name>

  <description>
     ICEfaces Tutorial: Timezone Part 1
     Create TimeZone as a stock JavaServer Faces application.
  </description>

  <session-config>
    <session-timeout>30</session-timeout>
  </session-config>

  <context-param>
     <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
     <param-value>server</param-value>
  </context-param>

  <!-- Faces Servlet -->
  <servlet>
     <servlet-name>Faces Servlet</servlet-name>
     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
     <load-on-startup> 1 </load-on-startup>
  </servlet>

  <!-- Faces Servlet Mapping -->
  <servlet-mapping>
     <servlet-name>Faces Servlet</servlet-name>
     <url-pattern>*.faces</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
     <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

This is a fairly typical descriptor for a JSF application. The Faces Servlet is declared and configured to load on startup. The .faces extension is mapped to the Faces Servlet.

In TimeZone 1, the deployment descriptor file for running under MyFaces, web_myfaces.xml, is exactly the same as web.xml. It has been included for consistency with the later TimeZone steps, where the deployment descriptor files differ.

Building and Deploying timezone1

For the tutorial applications, we rely on Ant as a build tool and Tomcat 6 as the container to run the application. To build a .war file successfully, complete the following steps:

1. Ensure that Tomcat 6 is running.
2. Use **`ant clean`** to clean up the source directory of the application.
3. To compile the source code and generate a .war file with all the required resources, in the console, type:
   `ant`

   By default, the .war file created is for Tomcat 6. The .war file will be created in the `dist` directory at the root level of the application.
4. To deploy the .war file to Tomcat 6, copy the timezone1.war file into the Tomcat6 `webapps` directory.
5. To interact with the application, point your web browser at the following URL, making adjustments for your environment as required. For example, port 8080 is the default port that Tomcat uses, but if you have changed this in your installation, change the URL accordingly.

   http://localhost:8080/timezone1

If all goes well, you should see the first incarnation of the TimeZone application running in your browser. Click a different time zone on the map to update the time and time zone information in the top right of the table.

> **Note:** If you would like to deploy the .war file to an application server other than Tomcat 6 (JBoss4.2, for example), you need to explicitly refer to a different ant target when building the .war file. Use **`ant help`** to list the available targets for this tutorial.

Step 2 – Integrating ICEfaces

In this step of the tutorial, we integrate ICEfaces technology into our existing JSF application. All the files and resources for this part of the tutorial are contained in the **timezone2** directory.

Turning JSP into JSP Document

JSP pages (unlike the more recent JSP Document specification) are not required to be well-formed XML documents. ICEfaces requires well-formed XML documents, so we need to make some modifications to our JSP page to align it with the JSP Document specification.

1. Change the file extension of the timezone.jsp file from .jsp to .jspx. This is the first step in converting the web page to a JSP Document (XML-compatible JSP). The .jspx extension identifies the file as an XML-compliant JSP page and allows us a bit of flexibility with our servlet mappings.
2. Remove the JSP taglib directives and declare them using xml namespaces in the JSF <f:view> element:

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

   becomes:

   ```
   <f:view xmlns:f="http://java.sun.com/jsf/core"
           xmlns:h="http://java.sun.com/jsf/html">
   ```

If the page consists of well-formed JSF and XHTML tags, this is the only change necessary to convert the page to JSP Document.

Registering ICEfaces Servlets

As an extension to JSF, ICEfaces provides its own version of the FacesServlet (PersistentFacesServlet) as well as an additional Servlet (BlockingServlet) for handling asynchronous updates. We register these Servlets in the deployment descriptor file (web.xml) by adding the following entries:

```
<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>
```

We also need to add a new Servlet mapping of PersistentFacesServlet to .iface.

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

```
</servlet-mapping>
```

We also need to add a couple of mappings for ICEfaces' internal use.

```
<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>
```

When using ICEfaces with JSP pages, we need to change the DEFAULT_SUFFIX entry to specify ".jspx" pages instead of the JSF default ".jsp" pages. This tells JSF to use .jspx as the default suffix so that requests are properly directed to ICEfaces' PersistentFacesServlet.

```
<context-param>
  <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
  <param-value>.jspx</param-value>
</context-param>
```

By default, ICEfaces uses asynchronous update mode that fully supports server-initiated updates (Ajax Push). For this tutorial step, we will not be leveraging asynchronous update mode, so we can optionally configure ICEfaces to use synchronous update mode instead:

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

When using some versions of Tomcat, users have reported seeing the following error occasionally "SEVERE: ICEfaces could not initialize JavaServer Faces. Please check that the JSF .jar files are installed correctly." If you experience this error, specifying the ConfigureListener will resolve this issue with Tomcat, even though it is not generally required for ICEfaces:

```
<listener>
    <listener-class>com.sun.faces.config.ConfigureListener</listener-class>
</listener>
```

Building and Deploying timezone2

For the tutorial applications, we rely on Ant as a build tool and Tomcat 6 as the container to run the application. To build a .war file successfully, complete the following steps:

1. Ensure that Tomcat 6 is running.
2. Use **`ant clean`** to clean up the source directory of the application.
3. To compile the source code and generate a .war file with all the required resources, in the console, type:
   `ant`

   By default, the .war file created is for Tomcat 6. The .war file will be created in the `dist` directory at the root level of the application.

4. To deploy the .war file to Tomcat 6, copy the timezone2.war file into the Tomcat 6 `webapps` directory.
5. To interact with the application, point your web browser at the following URL, making adjustments for your environment as required. For example, port 8080 is the default port that Tomcat uses, but if you have changed this in your installation, change the URL accordingly.

   http://localhost:8080/timezone2

> **Note:** If you would like to deploy the .war file to an application server other than Tomcat 6 (JBoss4.2, for example), you need to explicitly refer to a different ant target when building the .war file. Use **`ant help`** to list the available targets for this tutorial.

This version of TimeZone looks identical to timezone1 and has no functional difference. However, with little effort, we've integrated ICEfaces into our JSF application. The components are now being rendered by the ICEfaces Direct-to-DOM (D2D) RenderKit and we are now ready to enrich this application with some dynamic, asynchronous updates.

Step 3 – Dynamic Updating—Make the Clocks Tick

In this section, we are going to make the clocks tick by pushing updates from the server to the web browser, changing the content dynamically, but without a full page refresh, thanks to ICEfaces and Direct-to-DOM rendering.

We will also make some minor changes to the web.xml and faces-config.xml files, to support concurrent instances of the application, viewed from multiple windows or tabs of the same web browser.

All the files for this part of the tutorial are in the **timezone3** directory.

Enhancing the TimeZoneBean

Now that we have integrated ICEfaces, the work to show the clocks tick is done in the bean. No actual work is done to make the clocks tick because the system time updates automatically for us. Rather, at some interval, the components that display the clock times must be rendered, and those updates must be sent to the web browser. For this, we will use the ICEfaces specific RenderManager facilities to manage a JSF render pass. For timezone3, the following changes are made to the TimeZoneBean.java file.

1. First we add some imports to support the new ICEfaces features:

```
import com.icesoft.faces.async.render.IntervalRenderer;
import com.icesoft.faces.async.render.RenderManager;
import com.icesoft.faces.async.render.Renderable;
import com.icesoft.faces.webapp.xmlhttp.PersistentFacesState;
import com.icesoft.faces.webapp.xmlhttp.RenderingException;
```

```
import com.icesoft.faces.webapp.xmlhttp.FatalRenderingException;
import com.icesoft.faces.context.DisposableBean;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
```

2. Then we make the bean implement **com.icesoft.faces.async.render.Renderable**, so we can use it with the RenderManager facilities:

```
public class TimeZoneBean implements Renderable {
```

3. A rendering interval is added as a bean property:

```
private final int renderInterval = 1000;
```

4. Add helper objects for maintaining the rendering state and managing the threading issues:

```
private PersistentFacesState state;
private IntervalRenderer clock;
```

5. Initialize the rendering state:

```
private void init() {
       ...
       state = PersistentFacesState.getInstance();
}
```

6. Provide a callback method to use the RenderManager to set up the interval rendering:

```
public void setRenderManager(RenderManager renderManager) {
       clock = renderManager.getIntervalRenderer("clock");
       clock.setInterval(renderInterval);
       clock.add(this);
       clock.requestRender();
}
```

7. Allow the RenderManager facilities to access the rendering state:

```
public PersistentFacesState getState() {
       return state;
}
```

8. Provide a callback method to allow notification of rendering problems. An example of an expected invocation would be when the user has closed the web browser, and there is no target to render to:

```
public void renderingException(RenderingException renderingException) {
    if (log.isDebugEnabled()) {
       log.debug("Rendering exception:  " + renderingException);
    }
    if (renderingException instanceof FatalRenderingException) {
       performCleanup();
    }
}

protected boolean performCleanup() {
    try {
       if (clock != null) {
          clock.remove(this);
           if (clock.isEmpty() ) {
             clock.dispose();
          }
```

```
            clock = null;
        }
        return true;
    } catch (Exception failedCleanup) {
        if (log.isErrorEnabled()) {
            log.error("Failed to cleanup a clock bean", failedCleanup);
        }
    }
    return false;
}
```

9. To enable use of the RenderManager requires adding it as a managed application scoped bean, and having the application server tie it to our timeZoneBean's renderManager property. This is accommodated by making the following changes to the faces-config.xml file.

```
<managed-bean>
    <managed-bean-name>renderManager</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>timeZoneBean</managed-bean-name>
    <managed-bean-class>com.icesoft.tutorial.TimeZoneBean</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
    <managed-property>
           <property-name>renderManager</property-name>
           <value>#{renderManager}</value>
    </managed-property>
</managed-bean>
```

10. The RenderManager needs to know about the context and session lifecycles. To provide the appropriate information, the ICEfaces application needs to publish its **Context** events. This is achieved by adding the following code snippet to the web.xml file:

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

11. Implement the DisposableBean method by adding the following code snippet to the TimeZoneBean file:

```
public void dispose() throws Exception {
        if (log.isInfoEnabled()) {
            log.info("Dispose TimeZoneBean for a user - cleaning up");
        }
        performCleanup();
    }
```

Configuring ICEfaces for Concurrent Views

ICEfaces supports the concept of concurrent DOM viewing, which allows multiple windows or tabs of the same browser to view distinct instances of the same application. Without concurrent DOM viewing, pointing two different browser windows at the same application leads to unpredictable behavior since the server-side DOM would be shared between the two views. You can see what happens by opening two browser windows (of the same browser) and direct both windows to the timezone2 demo. Clicking on various time zones will update one view or the other but not both reliably.

Concurrent DOM viewing ensures each view has its own separate DOM and that backing beans are appropriately scoped for their responsibilities. To configure TimeZone to support concurrent DOM viewing, we need to modify both the deployment descriptor (web.xml or web_myfaces.xml) and the JavaServer Faces configuration file (faces-config.xml).

1. Add a context parameter to the deployment descriptor file (web.xml or web_myfaces.xml) so that ICEfaces is properly configured to support concurrent DOM views:

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

2. Now that we are leveraging ICEfaces' server-initiated update feature to push the clock-ticks to the browser, we need to enable asynchronous update mode in the web.xml (or web_myfaces.xml):

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

3. In faces-config.xml, change the scope of the TimeZoneBean from session to request:

```
<managed-bean>
    <managed-bean-name>timeZoneBean</managed-bean-name>
    <managed-bean-class>com.icesoft.tutorial.TimeZoneBean</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
    ...
</managed-bean>
```

This version of TimeZone looks identical to timezone1 and timezone2, but now, you should see the clocks ticking as the page is dynamically updated with each render pass initiated by the bean. With concurrent DOM viewing configured, we can now open timezone3 in two separate browser windows and operate them as if they were two distinct clients with updates and changes being accurately rendered.

Building and Deploying timezone3

For the tutorial applications, we rely on Ant as a build tool and Tomcat 6 as the container to run the application. To build a .war file successfully, complete the following steps:

1. Ensure that Tomcat 6 is running.
2. Use **`ant clean`** to clean up the source directory of the application.

3. To compile the source code and generate a .war file with all the required resources, in the console, type:
   `ant`

   By default, the .war file created is for Tomcat 6. The .war file will be created in the `dist` directory at the root level of the application.

4. To deploy the .war file to Tomcat 6, copy the timezone3.war file into the Tomcat 6 `webapps` directory.

5. To interact with the application, point your web browser at the following URL, making adjustments for your environment as required. For example, port 8080 is the default port that Tomcat uses, but if you have changed this in your installation, change the URL accordingly.

   http://localhost:8080/timezone3

> **Note:** If you would like to deploy the .war file to an application server other than Tomcat 6 (JBoss4.2, for example), you need to explicitly refer to a different ant target when building the .war file. Use **`ant help`** to list the available targets for this tutorial.

This version of TimeZone looks identical to timezone1 and timezone2 but now you should see the clocks ticking as the page is dynamically updated with each render pass initiated by the bean.

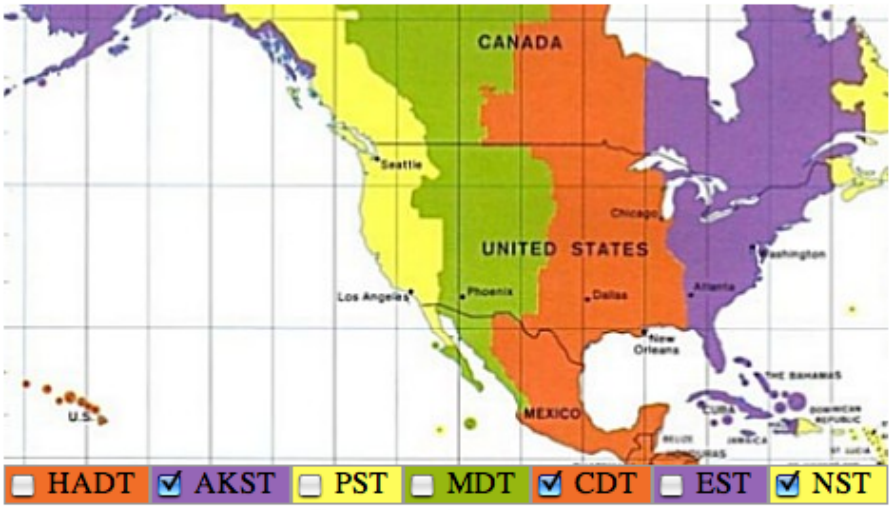
Step 4 – Dynamic Table Rendering

Now that we have the page updating dynamically, let's make it more interactive. We are going to add the ability to select time zones for which we want to see more detailed information. To do this we will add some selectBooleanCheckbox components and a dataTable component. As the checkboxes are selected and de-selected, the rows of the table will show or hide themselves without requiring a full page refresh. This is accomplished using a feature called partial submit, where each form component, such as a selectBooleanCheckbox, can trigger a partial form submission to the server when their state is changed. This is in contrast to regular form submission where only a Submit button would send updates to the server. The finished product will look similar to Figure 10:

Figure 10 TimeZone Application with Dynamic Table Rendering

ICEfaces: TimeZone Sample Application

Server Time Zone	Time Zone Selected from Map
Mountain	Eastern
Thu, 09:34:41	**Thu, 11:34:41**



Checked Time Zones

Time Zone	Location	Uses DST	In DST	Time
Alaska	America/Anchorage	Yes	Yes	Thu, 07:34:41
Central	America/Chicago	Yes	Yes	Thu, 10:34:41
Newfoundland	Canada/Newfoundland	Yes	Yes	Thu, 13:04:41

Modifying timezone.jspx

In the timezone.jspx page, make the following changes:

1. To support partial submission, we need to use an ICEfaces specific component, which requires adding a namespace declaration:

   ```
   xmlns:ice="http://www.icesoft.com/icefaces/component"
   ```

2. The standard JSF form component is replaced with the ICEfaces form component, enabling partial submission:

   ```
   <ice:form partialSubmit="true">
         ...
   </ice:form>
   ```

3. In the panelGrid holding the map, add a row of seven selectBooleanCheckbox components, under the seven commandButton components.

   ```
   <ice:panelGrid columns="7" cellspacing="0" cellpadding="0">
       <ice:panelGroup
               style="background:#f16e28; border:1px solid #999999;">
           <ice:selectBooleanCheckbox id="GMTminus10" required="false"
   ```

```
                                    immediate="true"

valueChangeListener="#{timeZoneBean.timeZoneChanged}"

value="#{timeZoneBean.checkboxStates['GMTminus10']}"
                                    autocomplete="off"/>
        <ice:outputLabel value="HADT" style="margin-right:7px;"/>
    </ice:panelGroup>
    <ice:panelGroup
            style="background: #9566b6; border:1px solid #999999;">
        <ice:selectBooleanCheckbox id="GMTminus9" required="false"
                                    immediate="true"

valueChangeListener="#{timeZoneBean.timeZoneChanged}"

value="#{timeZoneBean.checkboxStates['GMTminus9']}"
                                    autocomplete="off"/>
        <ice:outputLabel value="AKST" style="margin-right:7px;"/>
    </ice:panelGroup>
    <ice:panelGroup
            style="background: #fefc5a; border:1px solid #999999;">
        <ice:selectBooleanCheckbox id="GMTminus8" required="false"
                                    immediate="true"

valueChangeListener="#{timeZoneBean.timeZoneChanged}"

value="#{timeZoneBean.checkboxStates['GMTminus8']}"
                                    autocomplete="off"/>
        <ice:outputLabel value="PST" style="margin-right:7px;"/>
    </ice:panelGroup>
    <ice:panelGroup
            style="background: #96b710; border:1px solid #999999;">
        <ice:selectBooleanCheckbox id="GMTminus7" required="false"
                                    immediate="true"

valueChangeListener="#{timeZoneBean.timeZoneChanged}"

value="#{timeZoneBean.checkboxStates['GMTminus7']}"
                                    autocomplete="off"/>
        <ice:outputLabel value="MDT" style="margin-right:7px;"/>
    </ice:panelGroup>
    <ice:panelGroup
            style="background: #f16e28; border:1px solid #999999;">
        <ice:selectBooleanCheckbox id="GMTminus6" required="false"
                                    immediate="true"

valueChangeListener="#{timeZoneBean.timeZoneChanged}"

value="#{timeZoneBean.checkboxStates['GMTminus6']}"
                                    autocomplete="off"/>
        <ice:outputLabel value="CDT" style="margin-right:7px;"/>
    </ice:panelGroup>
    <ice:panelGroup
            style="background: #9566b6; border:1px solid #999999;">
        <ice:selectBooleanCheckbox id="GMTminus5" required="false"
                                    immediate="true"
```

```
valueChangeListener="#{timeZoneBean.timeZoneChanged}"

value="#{timeZoneBean.checkboxStates['GMTminus5']}"
                                    autocomplete="off"/>
        <ice:outputLabel value="EST" style="margin-right:8px;"/>
    </ice:panelGroup>
    <ice:panelGroup
            style="background: #fefc5a; border:1px solid #999999;">
        <ice:selectBooleanCheckbox id="GMTminus4" required="false"
                                    immediate="true"

valueChangeListener="#{timeZoneBean.timeZoneChanged}"

value="#{timeZoneBean.checkboxStates['GMTminus4']}"
                                    autocomplete="off"/>
        <ice:outputLabel value="NST" style="margin-right:8px;"/>
    </ice:panelGroup>
</ice:panelGrid>
```

4. A dataTable is added below the panelGrid component in the UI. This dataTable will display information on all the selected time zones, getting its data from timeZoneBean's checkedTimeZoneList property, which is a list of TimeZoneWrapper objects. The properties of each object in the list are then displayed through JSF expression language bindings in outputText components in each row of the dataTable.

```
<h:dataTable frame="box" value="#{timeZoneBean.checkedTimeZoneList}"
             var="checkedTimeZone">
       <f:facet name="header">
          <h:outputText value="Checked Time Zones"/></f:facet>
       <h:column>
             <f:facet name="header"><h:outputText value="Time Zone"/></f:facet>
             <h:outputText value="#{checkedTimeZone.displayName}"/>
       </h:column>
       <h:column>
             <f:facet name="header"><h:outputText value="Location"/></f:facet>
             <h:outputText value="#{checkedTimeZone.location}"/>
       </h:column>
       <h:column>
             <f:facet name="header"><h:outputText value="Uses DST"/></f:facet>
             <h:outputText value="#{checkedTimeZone.useDaylightTime}"/>
       </h:column>
       <h:column>
             <f:facet name="header"><h:outputText value="In DST"/></f:facet>
             <h:outputText value="#{checkedTimeZone.inDaylightTime}"/>
       </h:column>
       <h:column>
             <f:facet name="header"><h:outputText value="Time"/></f:facet>
             <h:outputText value=" #{checkedTimeZone.time} "/>
       </h:column>
</h:dataTable>
```

Modifying TimeZoneBean.java

Make the following additions to TimeZoneBean.java.

1. Import these classes to allow our use of JSF ValueChangeEvents.

```
import javax.faces.event.ValueChangeEvent;
import javax.faces.component.UIComponent;
import java.util.Hashtable;
```

2. Declare a list to hold the user's checked time zone selections.

```
private ArrayList checkedTimeZoneList;
```

3. Give the IDs of the selectBooleanCheckbox components, from timezone.jspx, for their respective time zones. These are Cminus5 through Cminus10. This way, when we receive a ValueChangeEvent, we'll know to which time zone it applies.

```
allTimeZoneList.add(new TimeZoneWrapper("Pacific/Honolulu", "GMTminus10",
                                    hawaiiXCoords, hawaiiYCoords,
                                    hawaiiXCoords.length));
allTimeZoneList.add(new TimeZoneWrapper("America/Anchorage", "GMTminus9",
                                    alaskaXCoords, alaskaYCoords,
                                    alaskaXCoords.length));
allTimeZoneList.add(new TimeZoneWrapper("America/Los_Angeles", "GMTminus8",
                                    pacificXCoords, pacificYCoords,
                                    pacificXCoords.length));
allTimeZoneList.add(new TimeZoneWrapper("America/Denver", "GMTminus7",
                                    mountainXCoords, mountainYCoords,
                                    mountainXCoords.length));
allTimeZoneList.add(new TimeZoneWrapper("America/Chicago", "GMTminus6",
                                    centralXCoords, centralYCoords,
                                    centralXCoords.length));
allTimeZoneList.add(new TimeZoneWrapper("America/New_York", "GMTminus5",
                                    easternXCoords, easternYCoords,
                                    easternXCoords.length));
allTimeZoneList.add(new TimeZoneWrapper("Canada/Newfoundland", "GMTminus4",
                                    nfldXCoords, nfldYCoords,
                                    nfldXCoords.length));
```

4. Initialize the list for storing the time zones that the user has checked, and wishes to display in the dataTable.

```
checkedTimeZoneList = new ArrayList();
```

5. Provide a getter accessor method for the checkedTimeZoneList bean property.

```
public ArrayList getCheckedTimeZoneList(){
       return checkedTimeZoneList;
}
```

6. Add a timeZoneChanged(ValueChangeEvent event) method to be called when a selectBooleanCheckbox is checked or unchecked. This uses our TimeZoneWrapper helper objects to map from the selectBooleanCheckbox component ID to the appropriate time zone, and its related information. Simply adding or removing the TimeZoneWrapper to or from checkedTimeZoneList is sufficient to add or remove a row in the web page's dataTable.

```
public void timeZoneChanged(ValueChangeEvent event){
       UIComponent comp = event.getComponent();
       FacesContext context = FacesContext.getCurrentInstance();
       String componentId = comp.getClientId(context);
       TimeZoneWrapper tzw = getTimeZoneWrapperByComponentId( componentId );
       if( tzw != null ) {
              boolean checked = ((Boolean)event.getNewValue()).booleanValue();
              // If checkbox is checked, then add tzw to checkedTimeZoneList
              if( checked ) {
                     if( !checkedTimeZoneList.contains(tzw) ){
                            checkedTimeZoneList.add( tzw );
                     }
                     // Otherwise, if checkbox is unchecked, then remove tzw from
                     // checkedTimeZoneList
                     else {
                            checkedTimeZoneList.remove( tzw );
                     }
              }
              checkboxStates.put(tzw.getMapCommandButtonId(),
                             checked ? "true" : "false");
          }
       }
```

Modifying TimeZoneWrapper.java

Each row in the dataTable is populated by a TimeZoneWrapper bean. Each cell in the dataTable is then populated by properties of the TimeZoneWrapper beans. So, we have to modify TimeZoneWrapper to add the requisite properties and accessor methods.

1. Add imports for the utility classes we use for calculating times with.

```
import java.util.TimeZone;
import java.util.Calendar;
import java.text.DateFormat;
```

2. Declare a helper DateFormat instance for the time bean property.

```
private DateFormat dateFormat;
```

3. Alter the constructor to initialize the new fields; checkboxId and dateFormat.

```
/**
 * @param id      id used to identify the time zone.
 * @param mapId   map button component id in web page
 * @param xCoords array of X-coordinates for the image map object.
 * @param yCoords array of Y-coordinates for the image map object.
 * @param coords  number of coordinates in the image map object.
 */
public TimeZoneWrapper(String id, String mapId, int[] xCoords,
                       int[] yCoords, int coords) {
    this.id = id;
    this.mapCommandButtonId = mapId;
    this.dateFormat =
            TimeZoneBean.buildDateFormatForTimeZone(
                    TimeZone.getTimeZone(id));
    mapPolygon = new Polygon(xCoords, yCoords, coords);
```

```
}
```

4. Add getter accessor methods for the displayName, time, useDaylightTime, inDaylightTime, location properties.

```
public String getDisplayName() {
       TimeZone timeZone = TimeZone.getTimeZone(id);
       return TimeZoneBean.displayNameTokenizer( timeZone.getDisplayName() );
}
...
public String getTime() {
       return TimeZoneBean.formatCurrentTime( dateFormat );
}
...
public String getUseDaylightTime() {
       TimeZone timeZone = TimeZone.getTimeZone(id);
       if( timeZone.useDaylightTime() )
              return "Yes";
       return "No";
}
...
public String getInDaylightTime() {
       TimeZone timeZone = TimeZone.getTimeZone(id);
       Calendar cal = Calendar.getInstance(timeZone);
       if( timeZone.inDaylightTime(cal.getTime()) )
              return "Yes";
       return "No";
}
...
public String getLocation() {
       return id;
}
```

Building and Deploying timezone4

For the tutorial applications, we rely on Ant as a build tool and Tomcat 6 as the container to run the application. To build a .war file successfully, complete the following steps:

1. Ensure that Tomcat 6 is running.
2. Use **`ant clean`** to clean up the source directory of the application.
3. To compile the source code and generate a .war file with all the required resources, in the console, type:
   `ant`

   By default, the .war file created is for Tomcat 6. The .war file will be created in the `dist` directory at the root level of the application.
4. To deploy the .war file to Tomcat 6, copy the timezone4.war file into the Tomcat 6 `webapps` directory.

5. To interact with the application, point your web browser at the following URL, making adjustments for your environment as required. For example, port 8080 is the default port that Tomcat uses, but if you have changed this in your installation, change the URL accordingly.

   http://localhost:8080/timezone4

> **Note:** If you would like to deploy the .war file to an application server other than Tomcat 6 (JBoss4.2, for example), you need to explicitly refer to a different ant target when building the .war file. Use `ant help` to list the available targets for this tutorial.

This version of TimeZone should now have checkboxes. As you click the checkboxes, the rows in the table should show and hide themselves accordingly. The interface is now richer and more dynamic thanks to ICEfaces and Direct-to-DOM rendering.

Step 5 – Applying Styles

This step of the tutorial describes how to apply styles throughout the TimeZone application to make it more visually appealing. This is an important aspect of web application development, and shows how the dual roles of application developer and page developer come together to put the final polish on an application.

Adding a Style Sheet to the Application

A Cascading Style Sheet is added to the web folder of timezone.jspx. This file is accessed by the application through the following line, added to timezone.jspx under the <head> tag:

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

Adding Images to the Application

Any images used by the style sheet should be dropped into the images subfolder of the application web folder.

Implementing Styles

In tutorial examples timezone1 through 4, inline styles were used:

```
<h:outputText style="font-weight:600" value="Server Time Zone"/>
<ice:panelGroup style="background: #fefc5a; border:1px solid #999999;">
```

Tutorial example timezone5 uses only styles from the style sheet, as shown in timezone.jspx, below:

```
<f:view xmlns:f="http://java.sun.com/jsf/core"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:ice="http://www.icesoft.com/icefaces/component">
    <html>
    <head><title>ICEfaces: TimeZone Sample Application</title></head>
    <link rel="stylesheet" type="text/css" href="./timezone_style.css"/>
    <body bgcolor="white">
    <div id="headerDiv">
        <table width="100%" cellpadding="0" cellspacing="0">
            <tr>
                <td valign="top">
                    <table width="100%" cellpadding="0" cellspacing="0">
                        <tr>
                            <td background="images/demo-page-bkgnd.gif"><img
                                    src="images/demo-page-left.gif"/></td>
                        </tr>
                        <tr>
                            <td height="45" valign="bottom"><img
                                    src="images/timezone_logo.gif"/></td>
                        </tr>
                    </table>
                </td>
                <td valign="top" align="right" width="119"><img
                        src="images/demo-page-right.gif"/></td>
            </tr>
        </table>
    </div>

    <div id="timeZonePanel">
        <ice:form partialSubmit="true">
            <ice:panelGrid columns="2" rowClasses="floatingDialogHeader, , "
                           width="100%">
                <ice:outputText value="Server Time Zone"/>
                <ice:outputText value="Time Zone Selected from Map"/>
                <ice:outputText styleClass="formLabel"
                                value="#{timeZoneBean.serverTimeZoneName}"/>
                <ice:outputText styleClass="formLabel"
                                value="#{timeZoneBean.selectedTimeZoneName}"/>
                <ice:outputText value="#{timeZoneBean.serverTime}"/>
                <ice:outputText value="#{timeZoneBean.selectedTime}"/>
            </ice:panelGrid>
            <ice:commandButton id="map" image="images/map.jpg"
                               actionListener="#{timeZoneBean.listen}"/>
            <ice:panelGrid columns="7" width="100%" cellpadding="0"
                           cellspacing="0"
                           columnClasses="orange, purple, yellow, green">
                <ice:panelGroup>
                    <ice:selectBooleanCheckbox id="GMTminus10" required="false"
                                               immediate="true"

valueChangeListener="#{timeZoneBean.timeZoneChanged}"

value="#{timeZoneBean.checkboxStates['GMTminus10']}"
                                               autocomplete="off"/>
                    <ice:outputLabel value="HADT" />
                </ice:panelGroup>
                <ice:panelGroup>
```

```
                    <ice:selectBooleanCheckbox id="GMTminus9" required="false"
                                               immediate="true"

valueChangeListener="#{timeZoneBean.timeZoneChanged}"

value="#{timeZoneBean.checkboxStates['GMTminus9']}"
                                               autocomplete="off"/>
                    <ice:outputLabel value="AKST" />
                </ice:panelGroup>
                <ice:panelGroup>
                    <ice:selectBooleanCheckbox id="GMTminus8" required="false"
                                               immediate="true"

valueChangeListener="#{timeZoneBean.timeZoneChanged}"

value="#{timeZoneBean.checkboxStates['GMTminus8']}"
                                               autocomplete="off"/>
                    <ice:outputLabel value="PST" />
                </ice:panelGroup>
                <ice:panelGroup>
                    <ice:selectBooleanCheckbox id="GMTminus7" required="false"
                                               immediate="true"

valueChangeListener="#{timeZoneBean.timeZoneChanged}"

value="#{timeZoneBean.checkboxStates['GMTminus7']}"
                                               autocomplete="off"/>
                    <ice:outputLabel value="MDT" />
                </ice:panelGroup>
                <ice:panelGroup>
                    <ice:selectBooleanCheckbox id="GMTminus6" required="false"
                                               immediate="true"

valueChangeListener="#{timeZoneBean.timeZoneChanged}"

value="#{timeZoneBean.checkboxStates['GMTminus6']}"
                                               autocomplete="off"/>
                    <ice:outputLabel value="CDT" />
                </ice:panelGroup>
                <ice:panelGroup>
                    <ice:selectBooleanCheckbox id="GMTminus5" required="false"
                                               immediate="true"

valueChangeListener="#{timeZoneBean.timeZoneChanged}"

value="#{timeZoneBean.checkboxStates['GMTminus5']}"
                                               autocomplete="off"/>
                    <ice:outputLabel value="EST" />
                </ice:panelGroup>
                <ice:panelGroup>
                    <ice:selectBooleanCheckbox id="GMTminus4" required="false"
                                               immediate="true"

valueChangeListener="#{timeZoneBean.timeZoneChanged}"

value="#{timeZoneBean.checkboxStates['GMTminus4']}"
                                               autocomplete="off"/>
```

```
                    <ice:outputLabel value="NST" />
                </ice:panelGroup>
            </ice:panelGrid>
            <ice:dataTable id="timezoneDataTable"
                           value="#{timeZoneBean.checkedTimeZoneList}"
                           var="checkedTimeZone" headerClass="tableHeader"
                           width="100%"

columnClasses="checkedDisplayName,checkedLocation,checkedUseDaylightTime,
                checkedInDaylightTime,checkedTime">
                <f:facet name="header">
                    <ice:outputText styleClass="tableHeaderTitle"
                                    value="Checked Time Zones"/>
                </f:facet>
                <ice:column>
                    <f:facet name="header">
                        <ice:outputText value="Time Zone"/>
                    </f:facet>
                    <ice:outputText value="#{checkedTimeZone.displayName}"/>
                </ice:column>
                <ice:column>
                    <f:facet name="header">
                        <ice:outputText value="Location"/>
                    </f:facet>
                    <ice:outputText value="#{checkedTimeZone.location}"/>
                </ice:column>
                <ice:column>
                    <f:facet name="header">
                        <ice:outputText value="Uses DST"/>
                    </f:facet>
                    <ice:outputText value="#{checkedTimeZone.useDaylightTime}"/>
                </ice:column>
                <ice:column>
                    <f:facet name="header">
                        <h:outputText value="In DST"/>
                    </f:facet>
                    <h:outputText value="#{checkedTimeZone.inDaylightTime}"/>
                </ice:column>
                <ice:column>
                    <f:facet name="header">
                        <ice:outputText value="Time"/>
                    </f:facet>
                    <ice:outputText styleClass="formLabel"
                                    value=" #{checkedTimeZone.time} "/>
                </ice:column>
            </ice:dataTable>
        </ice:form>
    </div>
    </body>
    </html>
</f:view>
```

There are two elements applied to the page. The first helps create a page heading:

```
<div id="headerDiv">
```

The second creates a container for the rest of the application:

```
<div id="timeZonePanel">
```

The styleClass attribute is used to apply styles from the style sheet to JSF elements:

```
<ice:outputText styleClass="formLabel" value="#{timeZoneBean.serverTimeZoneName}"/>
```

The JSF panelGrid component has a **rowClasses** attribute that applies styles from the style sheet to the rows of the table it creates. In this example, the floatingDialogHeader class is applied to the first row and the second and third rows are left blank, meaning no style is applied.

```
<ice:panelGrid columns="2" rowClasses="floatingDialogHeader, , " width="100%">
```

The JSF dataTable component has a headerClass attribute that applies styles to all the headers in the table. It also has a columnClasses attribute that applies styles to all of the columns in the table:

```
<ice:dataTable id="timezoneDataTable" var="checkedTimeZone"
               value="#{timeZoneBean.checkedTimeZoneList}"
               headerClass="tableHeader" width="100%"
               columnClasses="checkedDisplayName,
               checkedLocation,checkedUseDaylightTime,
               checkedInDaylightTime,checkedTime">
```

Building and Deploying timezone5

For the tutorial applications, we rely on Ant as a build tool and Tomcat 6 as the container to run the application. To build a .war file successfully, complete the following steps:

1. Ensure that Tomcat 6 is running.
2. Use `ant clean` to clean up the source directory of the application.
3. To compile the source code and generate a .war file with all the required resources, in the console, type:
   `ant`

   By default, the .war file created is for Tomcat 6. The .war file will be created in the `dist` directory at the root level of the application.
4. To deploy the .war file to Tomcat 6, copy the timezone5.war file into the Tomcat 6 `webapps` directory.
5. To interact with the application, point your web browser at the following URL, making adjustments for your environment as required. For example, port 8080 is the default port that Tomcat uses, but if you have changed this in your installation, change the URL accordingly.

   http://localhost:8080/timezone5

> **Note:** If you would like to deploy the .war file to an application server other than Tomcat 6 (JBoss4.2, for example), you need to explicitly refer to a different ant target when building the .war file. Use `ant help` to list the available targets for this tutorial.

This version of TimeZone has the same functionality as timezone4, but should look completely different. Styles have been applied to the application through the stylesheet, which uses images located in the web/images folder. With the finishing touches applied, the application is now ready to be presented to the world.

Step 6 – Integrating Facelets

Each of the steps, until now, have added new functionality to the users of the TimeZone web application, showing more data or being more interactive. This time though, our application should be functionally equivalent to the previous, but instead of being based on JSP, will instead use Facelets (see **https://facelets.dev.java.net/**). The intent of this step is to show the minimal effort required to port an ICEfaces JSF JSP Document application to ICEfaces JSF Facelets.

> **Note:** This step is related to Facelets and will be of interest only if you intend to use Facelets.

Facelets Dependencies

There are three new JAR files, distributed with ICEfaces, which must be included with any ICEfaces Facelets application:

- icefaces-facelets.jar
- el-api.jar (not required for Tomcat 6, Jetty 6.1, JBoss 4.2, or Glassfish v2)
- el-ri.jar

Configuring for Facelets

To configure TimeZone to support Facelets, we need to modify both the deployment descriptor (web.xml or web_myfaces.xml) and the JavaServer Faces configuration file (faces-config.xml).

1. Add a context parameter to the deployment descriptor file (web.xml or web_myfaces.xml), to inform the application server that the default file extension for Facelets is .xhtml, since some application servers assume .jsp or .jspx extensions.

```
<context-param>
      <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
```

```
        <param-value>.xhtml</param-value>
</context-param>
```

2. For development purposes, add a context parameter to the deployment descriptor file, to enable Facelets debug logging.

```
<context-param>
        <param-name>facelets.DEVELOPMENT</param-name>
        <param-value>true</param-value>
</context-param>
```

3. Enable some strict verifications in JSF, by adding these context parameters to the deployment descriptor file.

```
<context-param>
        <param-name>com.sun.faces.validateXml</param-name>
        <param-value>true</param-value>
        <description>
            Set this flag to true, if you want the JavaServer Faces Reference
            Implementation to validate the XML in your faces-config.xml resources
            against the DTD. Default value is false.
        </description>
</context-param>

<context-param>
        <param-name>com.sun.faces.verifyObjects</param-name>
        <param-value>true</param-value>
        <description>
            Set this flag to true, if you want the JavaServer Faces Reference
            Implementation to verify that all of the application objects you have
            configured (components, converters, renderers, and validators) can be
            successfully created. Default value is false.
        </description>
</context-param>
```

4. Set the application view handler in the JavaServer Faces configuration file (faces-config.xml).

```
<application>
        <view-handler>
            com.icesoft.faces.facelets.D2DFaceletViewHandler
        </view-handler>
</application>
```

Change Web Files from JSP Document to Facelets

Rename timezone.jspx to timezone.xhtml. No JSP Document specific features were used in previous steps that are not already supported by Facelets, so no change to the contents of timezone.xhtml is necessary.

Building and Deploying timezone6

For the tutorial applications, we rely on Ant as a build tool and Tomcat 6 as the container to run the application. To build a .war file successfully, complete the following steps:

1. Ensure that Tomcat 6 is running.
2. Use **`ant clean`** to clean up the source directory of the application.
3. To compile the source code and generate a .war file with all the required resources, in the console, type:
   `ant`

   By default, the .war file created is for Tomcat 6. The .war file will be created in the `dist` directory at the root level of the application.
4. To deploy the .war file to Tomcat 6, copy the timezone6.war file into the Tomcat 6 `webapps` directory.
5. To interact with the application, point your web browser at the following URL, making adjustments for your environment as required. For example, port 8080 is the default port that Tomcat uses, but if you have changed this in your installation, change the URL accordingly.

   http://localhost:8080/timezone6

> **Note:** If you would like to deploy the .war file to an application server other than Tomcat 6 (JBoss4.2, for example), you need to explicitly refer to a different ant target when building the .war file. Use **`ant help`** to list the available targets for this tutorial.

Step 7 – Capitalize on Facelets

The main focus of this step is to use Facelets to make our application more dynamic.

> **Note:** This step is related to Facelets and will be of interest only if you intend to use Facelets.

In previous steps of the tutorial, TimeZoneBean.java and timezone.xhtml (or timezone.jspx) were codependent. Components in the UI would generate events from user interactions, requiring hard-coded constants in the bean to interpret the source and relevancy of the events. Changes to component IDs, or addition or removal of components would necessitate lock-step changes in both the bean and the web file. This step of the tutorial puts the bean in charge of generating content for the web file. With a few changes to one section of code in TimeZoneBean.java, any number of arbitrary time zones could be displayed with timezone.xhtml.

Putting the TimeZoneBean in Charge

First we change timezone.xhtml, to use Facelets mechanisms, to be more dynamic.

1. Add namespace declarations for Facelets components, and Facelet's implementation of JSTL components and functions:

```
<f:view xmlns:f="http://java.sun.com/jsf/core"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:ice="http://www.icesoft.com/icefaces/component"
        xmlns:ui="http://java.sun.com/jsf/facelets"
        xmlns:c="http://java.sun.com/jstl/core"
        xmlns:fn="http://java.sun.com/jsp/jstl/functions">
```

2. Pull out the headerDiv section, and put it into header.xhtml to demonstrate parameterized inclusion:

```
<ui:include src="/header.xhtml">
       <ui:param name="sectionName" value="header"/>
</ui:include>
```

3. Replace the seven selectBooleanCheckboxes and outputLabels with JSTL forEach sections, that are populated by bean properties using JSF expression language bindings.

```
<ice:panelGrid columns="#{fn:length(timeZoneBean.allTimeZoneList)}"
               width="100%" cellpadding="0" cellspacing="0"
               columnClasses="orange, purple, yellow, green">
    <c:forEach var="allTimeZone"
               items="#{timeZoneBean.allTimeZoneList}">
        <ice:panelGroup>
            <ice:selectBooleanCheckbox
                    id="#{allTimeZone.mapCommandButtonId}" required="false"
                    immediate="true"
                    value="#{allTimeZone.currentlyShowing}"
                    valueChangeListener="#{timeZoneBean.timeZoneChanged}"
                    autocomplete="off"/>
            <ice:outputLabel value="#{allTimeZone.abbreviation}" />
        </ice:panelGroup>
    </c:forEach>
</ice:panelGrid>
```

Adding New Properties

Each section in the map, and each check box, and each row in the bottom table, are populated by properties from a TimeZoneWrapper object. We will change TimeZoneWrapper.java to add the new properties that timezone.xhtml uses JSF expression language bindings to access:

1. Add a new property to hold the time zone abbreviated name for the time zone being represented by this TimeZoneWrapper:

```
private String abbreviation;
```

2. Add another property for managing both the current state of the check box for this time zone, and the visibility of the corresponding row in the bottom table:

```
private boolean currentlyShowing;
```

3. Initialize the new properties:

```
/**
 * @param id      id used to identify the time zone.
 * @param mapId   map button component id in web page
 * @param abbreviation timezone abbreviated label
 * @param xCoords array of X-coordinates for the image map object.
 * @param yCoords array of Y-coordinates for the image map object.
 * @param coords number of coordinates in the image map object.
 */
public TimeZoneWrapper(String id, String mapId,
           String abbreviation, int[] xCoords, int[] yCoords, int coords) {
    this.id = id;
    this.mapCommandButtonId = mapId;
    this.abbreviation = abbreviation;
    this.currentlyShowing = false;
    this.dateFormat = TimeZoneBean.buildDateFormatForTimeZone(
            TimeZone.getTimeZone(id));
    mapPolygon = new Polygon(xCoords, yCoords, coords);
```

4. Add accessor methods for the new properties:

```
public String getAbbreviation(){
    return abbreviation;
}
...
public boolean getCurrentlyShowing() {
    return currentlyShowing;
}
...
public void setCurrentlyShowing(boolean showing) {
    currentlyShowing = showing;
}
```

Updating TimeZoneBean.java

Finally, update TimeZoneBean.java to initialize and access the newer more dynamic data structures.

1. Create the master list of all time zones in the application, along with their images, and the component IDs:

```
allTimeZoneList.add(new TimeZoneWrapper("Pacific/Honolulu",
        "GMTminus10", "HADT",
        hawaiiXCoords, hawaiiYCoords, hawaiiXCoords.length));
allTimeZoneList.add(new TimeZoneWrapper("America/Anchorage",
        "GMTminus9", "AKST",
        alaskaXCoords, alaskaYCoords, alaskaXCoords.length));
allTimeZoneList.add(new TimeZoneWrapper("America/Los_Angeles",
        "GMTminus8", "PST",
        pacificXCoords, pacificYCoords, pacificXCoords.length));
allTimeZoneList.add(new TimeZoneWrapper("America/Denver",
        "GMTminus7", "MDT",
        mountainXCoords, mountainYCoords, mountainXCoords.length));
allTimeZoneList.add(new TimeZoneWrapper("America/Chicago",
        "GMTminus6", "CDT",
        centralXCoords, centralYCoords, centralXCoords.length));
allTimeZoneList.add(new TimeZoneWrapper("America/New_York",
```

```
        "GMTminus5", "EST",
        easternXCoords, easternYCoords, easternXCoords.length));
allTimeZoneList.add(new TimeZoneWrapper("Canada/Newfoundland",
        "GMTminus4", "NST",
        nfldXCoords, nfldYCoords, nfldXCoords.length));
```

2. Provide a getter accessor method for the master time zone list property:

```
public ArrayList getAllTimeZoneList(){
       return allTimeZoneList;
}
```

3. Modify the checkbox event handler to properly maintain the new bean properties:

```
public void timeZoneChanged(ValueChangeEvent event){
       UIComponent comp = event.getComponent();
       FacesContext context = FacesContext.getCurrentInstance();
       String componentId = comp.getClientId(context);
       TimeZoneWrapper tzw = getTimeZoneWrapperByComponentId( componentId );
       if( tzw != null ) {
              boolean checked = ((Boolean)event.getNewValue()).booleanValue();
              // If checkbox is checked, then add tzw to checkedTimeZoneList
              if( checked ) {
                     tzw.setCurrentlyShowing( true );
                     if( !checkedTimeZoneList.contains(tzw) )
                            checkedTimeZoneList.add( tzw );
              }
              // Otherwise, if checkbox is unchecked, then remove tzw from
              // checkedTimeZoneList
              else {
                     tzw.setCurrentlyShowing( false );
                     checkedTimeZoneList.remove( tzw );
              }
       }
}
```

Building and Deploying timezone7

For the tutorial applications, we rely on Ant as a build tool and Tomcat 6 as the container to run the application. To build a .war file successfully, complete the following steps:

1. Ensure that Tomcat 6 is running.
2. Use **`ant clean`** to clean up the source directory of the application.
3. To compile the source code and generate a .war file with all the required resources, in the console, type:
   `ant`

   By default, the .war file created is for Tomcat 6. The .war file will be created in the `dist` directory at the root level of the application.
4. To deploy the .war file to Tomcat 6, copy the timezone7.war file into the Tomcat 6 `webapps` directory.

5. To interact with the application, point your web browser at the following URL, making adjustments for your environment as required. For example, port 8080 is the default port that Tomcat uses, but if you have changed this in your installation, change the URL accordingly.

   http://localhost:8080/timezone7

> **Note:** If you would like to deploy the .war file to an application server other than Tomcat 6 (JBoss4.2, for example), you need to explicitly refer to a different ant target when building the .war file. Use `ant help` to list the available targets for this tutorial.

For more ICEfaces tutorials, visit the ICEfaces Community web site:

http://www.icefaces.org

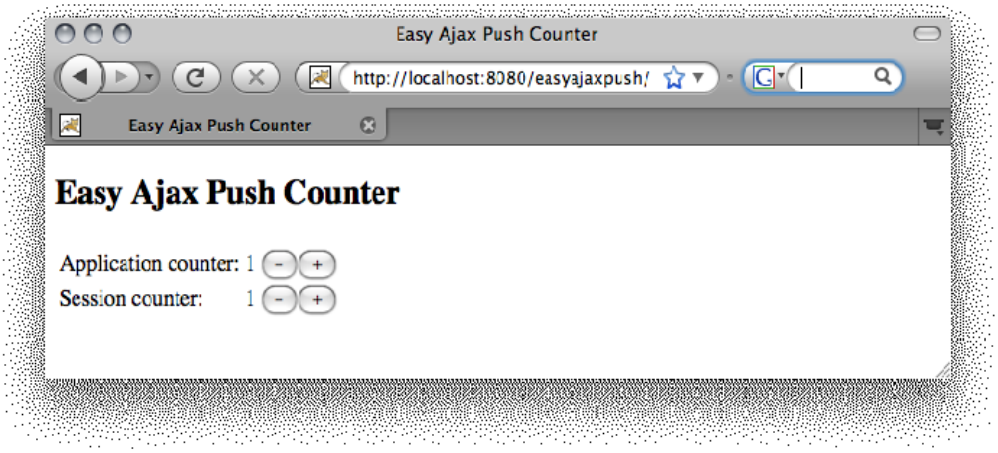
Chapter 5 ICEfaces SessionRenderer Tutorial: Easy Ajax Push

Overview

The essence of Ajax Push is the ability to trigger updates to the client from events or changes that occur on the server. This makes Ajax Push highly suitable for collaborative applications where one user does something that changes the current state of the application and other users are interested in seeing that change. There are many different ways to take advantage of Ajax Push to add collaborative features to your rich web application. From the developer's perspective, Ajax Push is exposed through the SessionRenderer API. This tutorial is designed to provide a simple and clear illustration of how to get started with the SessionRenderer API to add Ajax Push to your application.

The project we are going to build in this tutorial is a page that contains simple, adjustable counters. The counter values are displayed on the page and buttons are provided that can be used to increment or decrement the counter values. The finished project looks like this:

Figure 11 Easy Ajax Push Counter Project



The application consists of two counters. One is an application-scoped counter that can be adjusted and seen by all users. The second counter is session-scoped so changes to the value of this counter are restricted to the individual user. The goal is to use the SessionRenderer API so that any modifications to the application-scoped counter result in Ajax Push updates being sent to all application users.

All the source code for this tutorial is available in the ICEfaces bundle:

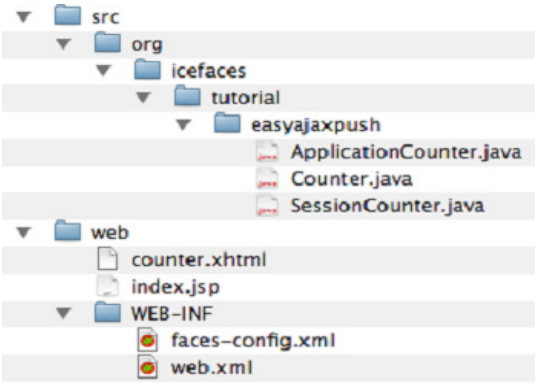
```
[install_dir]/samples/tutorial/easyajaxpush
```

Tools and Environment

You can develop ICEfaces applications using the tools of your choice. This tutorial doesn't make any assumptions about which editor or IDE you are using. The code in this tutorial is designed to run compatibly on JDK 1.4. so there is no use of annotations, concurrency utilities, etc.

We'll be creating the files for this tutorial based on the same directory structure as it exists in the product. As we reference the creation of various files, we'll be referring to how they are distributed in the ICEfaces bundle:

Figure 12 SessionRenderer Tutorial Directory Structure



Creating the Beans

Counter.java

Our first task is to create the backing beans that will be used to hold the state of the counters. First we'll start with a generic counter that we can use as the base class for our more specialized counter implementations.

Create a new class called Counter by typing or copying the following into your editor:

```
package org.icefaces.tutorial.easyajaxpush;

import javax.faces.event.ActionEvent;

public class Counter {

private int count;

public Counter() {
    }
public int getCount() {
        return count;
    }
public void setCount(int count) {
```

```
        this.count = count;
    }

    public void increment(ActionEvent event) {
        count++;
    }

    public void decrement(ActionEvent event) {
        count--;
    }

}
```

ApplicationCounter.java

Since multiple users can potentially access and modify the value of our application-scoped counter, we need to create a version that guards against concurrent access. So we'll create an ApplicationCounter that extends our basic Counter and synchronizes methods that allow modification of our counter value.

Create a new class called ApplicationCounter by typing or copying the following into your editor:

```
package org.icefaces.tutorial.easyajaxpush;

import javax.faces.event.ActionEvent;

public class ApplicationCounter extends Counter {

    public ApplicationCounter() {
    }

    public synchronized void setCount(int count){
        super.setCount(count);
    }

    public synchronized void increment(ActionEvent event) {
        super.increment(event);
    }

    public synchronized void decrement(ActionEvent event) {
        super.decrement(event);
    }

}
```

Define Managed Beans

To make use of our classes as JSF managed beans, we need to declare them in our faces-config.xml file. We plan to have two counters: an application-scoped counter based on the ApplicationCounter class and a session-scoped counter based on our basic Counter class. So we need to define two JSF managed beans.

Create a file called faces-config.xml by typing or copying the following into your editor:

```
<?xml version='1.0' encoding='UTF-8'?>

<faces-config>

    <managed-bean>
        <description>An application-scoped counter.</description>
        <managed-bean-name>applicationCounter</managed-bean-name>
        <managed-bean-class>
            org.icefaces.tutorial.easyajaxpush.ApplicationCounter
        </managed-bean-class>
        <managed-bean-scope>application</managed-bean-scope>
    </managed-bean>

    <managed-bean>
        <description>A session-scoped counter.</description>
        <managed-bean-name>sessionCounter</managed-bean-name>
        <managed-bean-class>
            org.icefaces.tutorial.easyajaxpush.Counter
        </managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>

</faces-config>
```

Creating the Pages

Now we need to create our user interface that includes components that we can bind to using the property accessors and methods of our managed beans. For this tutorial, we'll be using Facelets because it is the recommended templating language for JSF.

Configure Facelets Support

There are two things you need to do in an ICEfaces application that are specifically related to Facelets.

icefaces-facelets.jar

ICEfaces includes its own version of Facelets. You need to ensure that this library is included with your deployment. The build file included with the easyajaxpush project already does this for you.

D2DFaceletViewHandler

ICEfaces needs to be configured to use a specific ViewHandler implementation when running with Facelets. This is done by specifying the D2DFaceletViewHandler in your faces-config.xml file.

Copy or type the bolded section of code into the faces-config.xml file:

```
<?xml version='1.0' encoding='UTF-8'?>
```

```
<faces-config>

    <application>
        <view-handler>
            com.icesoft.faces.facelets.D2DFaceletViewHandler
        </view-handler>
    </application>

   <managed-bean>
        <description>An application-scoped counter.</description>
        <managed-bean-name>applicationCounter</managed-bean-name>
        <managed-bean-class>
            org.icefaces.tutorial.easyajaxpush.ApplicationCounter
        </managed-bean-class>
        <managed-bean-scope>application</managed-bean-scope>
    </managed-bean>

    <managed-bean>
        <description>A session-scoped counter.</description>
        <managed-bean-name>sessionCounter</managed-bean-name>
        <managed-bean-class>
            org.icefaces.tutorial.easyajaxpush.Counter
        </managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>

</faces-config>
```

Create the Page

Next we need to create a page for displaying our counters and the buttons that allow us to increment or decrement the values.

Create a file called counter.xhtml by typing or copying the following into your editor:

```
<f:view xmlns="http://www.w3.org/1999/xhtml"
        xmlns:ui="http://java.sun.com/jsf/facelets"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:ice="http://www.icesoft.com/icefaces/component"
        xmlns:f="http://java.sun.com/jsf/core">

    <html>
    <head>
        <title>Easy Ajax Push Counter</title>
    </head>

    <body>

    <h2>Easy Ajax Push Counter</h2>

    <ice:form>
        <ice:panelGrid columns="3">
            <ice:outputText value="Application counter:"/>
            <ice:outputText value="#{applicationCounter.count}"/>
```

```
            <ice:panelGroup>
                <ice:commandButton value="-"

actionListener="#{applicationCounter.decrement}"/>
                <ice:commandButton value="+"

actionListener="#{applicationCounter.increment}"/>
            </ice:panelGroup>

            <ice:outputText value="Session counter:"/>
            <ice:outputText value="#{sessionCounter.count}"/>
            <ice:panelGroup>
                <ice:commandButton value="-"
                                   actionListener="#{sessionCounter.decrement}"/>
                <ice:commandButton value="+"
                                   actionListener="#{sessionCounter.increment}"/>
            </ice:panelGroup>

        </ice:panelGrid>
    </ice:form>

    </body>

    </html>
</f:view>
```

Create the Descriptor (web.xml)

And of course all good web applications need a web.xml. There is nothing specific related to Ajax Push that is required in the web.xml file.

Create a file called web.xml by typing or copying the following into your editor:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app version="2.5"
         xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

    <display-name>ICEfaces SessionRenderer Tutorial: Easy Ajax Push</display-name>

    <description>
        A simple shared counter application that shows
        how to use the SessionRenderer API to add a
        collaborative element to your ICEfaces applications.
    </description>

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

    <context-param>
        <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
```

```
    <param-value>.xhtml</param-value>
</context-param>

<context-param>
    <param-name>facelets.DEVELOPMENT</param-name>
    <param-value>true</param-value>
</context-param>

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

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

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

<welcome-file-list>
    <welcome-file>counter.iface</welcome-file>
```

```
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>

</web-app>
```

Running without Ajax Push

So we now have an application that is ready to run, but without Ajax Push functionality. We'll add that right after we give our counters a test.

For the tutorial applications, we rely on Ant as a build tool and Tomcat 6 as the default container to run the application. To build a .war file successfully, complete the following steps:

1. Ensure that Tomcat 6 is running.
2. Use **`ant clean`** to clean up the source directory of the application.
3. To compile the source code and generate a .war file with all the required resources, in the console, type:

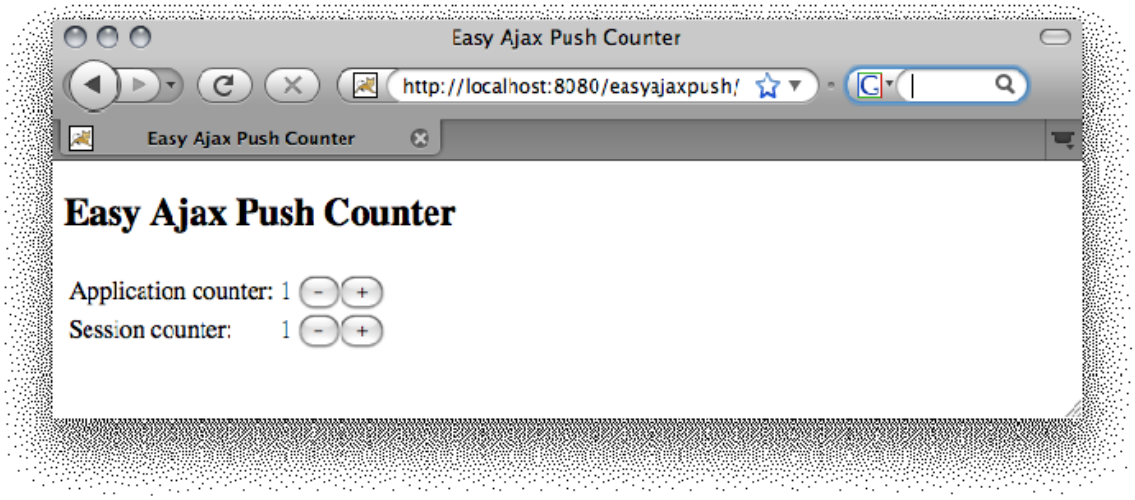
   ```
   ant
   ```

   By default, the .war file created is for Tomcat 6. The .war file will be created in the dist directory at the root level of the application.
4. To deploy the.war file to Tomcat 6, copy the easyajaxpush.war file into the Tomcat6 webapps directory.
5. To interact with the application, point your web browser at the following URL, making adjustments for your environment as required. For example, port 8080 is the default port that Tomcat uses, but if you have changed this in your installation, change the URL accordingly.

   ```
   http://localhost:8080/easyajaxpush/
   ```

If all goes well, you should see the first incarnation of the Easy Ajax Push Counter application running in your browser.

Figure 13 Easy Ajax Push Counter Completed Project



Clicking the increment and decrement buttons for each counter should change the values. To fully investigate how the counters work, open up another browser window. Some interesting things to look for:

- If you open another window using the same browser (i.e., two Firefox windows) you should notice that the session counter is shared between the two tabs/windows. That's because the session cookie is the same. What you have is two views into the same application from the same session. If you open up another window from a different browser (i.e., one from Firefox and one from IE), the session counters are not shared.
- The counters are only updated in the browser that you interact with. That's because we haven't yet added Ajax Push so the other browser windows don't get the latest values until you interact with them as well.

> **Note:** If you would like to deploy the .war file to an application server other than Tomcat6 (JBoss4.2, for example), you need to explicitly refer to a different ant target when building the .war file. Use **ant help** to list the available targets for this tutorial.

Adding Ajax Push with SessionRenderer

There are basically two parts to using the SessionRenderer API.

- adding sessions to the group or groups we wish to render
- calling a render when something of interest changes for that group

The entire SessionRenderer API supports this in a straight-forward manner.

- public static void addCurrentSession(final String groupName)

- public static void removeCurrentSession(final String groupName)
- public static void render(final String groupName)

SessionCounter.java

Since we want everyone to be able to access the application-scoped counter, we're going to keep it simple and just add everyone to a single, global group. The easiest way to do this is to have the group membership logic in the constructor of the session-scoped bean. That way, when our session-scoped bean is created, it's added to a group. Since we only want to do this for our session-scoped counter (and not our application-scoped counter) we'll create another subclass of Counter that does this for us.

Create a new class called SessionCounter by typing or copying the following into your editor:

```
package org.icefaces.tutorial.easyajaxpush;

import com.icesoft.faces.async.render.SessionRenderer;

public class SessionCounter extends Counter {

    public SessionCounter() {
        SessionRenderer.addCurrentSession("all");
    }
}
```

> **Note:** For ICEfaces v1.7.1 and v1.7.2, the SessionRenderer API was considered experimental so the package for it reflects this (org.icefaces.x.core.push.SessionRenderer). As of ICEfaces v1.8, the API is no longer considered experimental and resides with the rest of the Ajax Push APIs under com.icesoft.faces.async.render.

Adjust the Managed Bean Description

Now that we have a new bean implementation for our session-scoped counter, we need to update the JSF descriptor to make use of it.

Change the class of the managed bean to use our new implementation:

```
<managed-bean>
    <description>A session-scoped counter.</description>
    <managed-bean-name>sessionCounter</managed-bean-name>
    <managed-bean-class>
        org.icefaces.tutorial.easyajaxpush.SessionCounter
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
 </managed-bean>
```

SessionRender for Ajax Push

Our final step in adding Ajax Push features to our application counter requires modifications to the ApplicationCounter. What we want is for updates to be pushed out whenever there is a change. To do this, you simply need to call the render method of the SessionRenderer when a notable change occurs.

```
public static void render(final String groupName)
```

Copy or type the bolded section of code into the ApplicationCounter.java file:

```
package org.icefaces.tutorial;

import javax.faces.event.ActionEvent;
import com.icesoft.faces.async.render.SessionRenderer;

public class ApplicationCounter extends Counter {

    public ApplicationCounter() {
    }

    public synchronized void setCount(int count){
        super.setCount(count);
        SessionRenderer.render("all");
    }

    public synchronized void increment(ActionEvent event) {
        super.increment(event);
        SessionRenderer.render("all");
   }

    public synchronized void decrement(ActionEvent event) {
        super.decrement(event);
        SessionRenderer.render("all");
   }

}
```

So whenever we note a change to the application-scoped counter, we issue a request for a render to the group “all” which, in our case, means every active session.

Running with Ajax Push

You can follow the same procedure as noted before to build and deploy the modified version of the application. Now if you point two browsers at the application:

```
http://localhost:8080/easyajaxpush/
```

Clicking the buttons for the application-scoped counter should provide a richer experience. Updates from interaction in one browser that result in a change to the application counter are now pushed out to other active users. In other words, all browsers should notice when the application counter is modified. It’s left as an exercise for the reader to enhance the application so that the session counter is updated when using two tabs/windows from the same browser.

And there you have it. Using the SessionRenderer API is an easy and straightforward way to add collaborative features to your ICEfaces applications.

Chapter 6 ICEfaces Design Decisions — Using ICEfaces in Your Project

Before you begin to use ICEfaces for a new project there are several important considerations and design decisions to be made regarding how ICEfaces will be configured, used, and deployed.

Choosing your Markup: JSPs or Facelets

ICEfaces supports JSF development using both JSPs and Facelets. However, each markup-type has its own strengths and weaknesses that must be considered:

JSPs	Facelets
• Leverage existing JSP knowledge	• Improved performance and scalability
• Excellent IDE vendor support	• Fast templating/decorators for comp., page re-use
	• Precise error reporting
	• Full EL support

Generally speaking, unless your project has compelling reasons to use JSPs, Facelets should be seriously considered for its many advantages over JSP-based development for JSF.

Asynchronous vs. Synchronous Update Mode

By default ICEfaces uses asynchronous update mode. This provides powerful capabilities for fully leveraging ICEfaces' server-initiated update facilities in your application (Ajax Push). Using these Ajax Push capabilities it is simple to create a compelling, rich application interface unlike anything possible with traditional web development technologies.

However, not all applications require the unique capabilities provided by asynchronous update mode, and there are a few special considerations when using it. See **Asynchronous vs. Synchronous Updates**, **Optimizing Asynchronous Communications for Scalability**, and **Using ICEfaces in Clustered Environments** in the **ICEfaces Developer's Guide** for details.

JSF 1.1 vs. JSF 1.2

ICEfaces is runtime compatible with the following JSF runtime libraries:

- Sun JSF 1.1
- MyFaces JSF 1.1
- Sun JSF 1.2

Integration with 3rd Party Frameworks

Many projects use a combination of frameworks to achieve the goals of the project. ICEfaces supports several of the most popular Java frameworks that work with JavaServer Faces (JSF):

- Spring Web Flow
- JBoss Seam

If your project requires integration with either of these frameworks, it is important to review the ICEfaces documentation for detailed information that will help you to avoid potential pitfalls and common mistakes. See **Spring Web Flow Integration** and **JBoss Seam Integration** sections in the **ICEfaces Developer's Guide** for more information.

Integration with 3rd Party JSF Components

ICEfaces supports integration with third party JSF components. By default, using the **icefaces.jar** in your application classpath will provide full support for using ICEfaces components ("**ice:**") together with standard JSF components on the same page ("**h:**"). In addition, it is generally possible to combine other third party components in your ICEfaces pages successfully simply by declaring their namespace and adding them to the page.

However, JSF components that utilize a custom JSF ViewHandler or leverage Ajax and/or JavaScript in their component renderers may conflict with the ICEfaces framework and bridge preventing their use together with ICEfaces components on the same page. In these cases, you can still use ICEfaces in the same application as another component set by segregating the application pages into ICEfaces/non-ICEfaces categories. By using the `just-ice.jar` library in place of the **icefaces.jar** in your classpath, it is possible to configure ICEfaces to limit its processing to **.iface** pages exclusively. Thus, pages with other extensions can be processed by another view handler, avoiding the conflicts that may occur when combining certain third party components on the same page as ICEfaces components.

Special Considerations for Portlet Development

ICEfaces supports developing and deploying JSR 168 compliant portlets.

Standard portal and portlet development have their own set of design considerations that need to be kept in mind by the developer. Adding ICEfaces can significantly enhance the richness of your portlet interface and, through Ajax Push, can provide a means for seamlessly updating multiple portlets on a single page. However, in addition to the general design decisions discussed previously, there are some specific things to consider to ensure that your portlet development experience is successful. More details on these topics can be found in the **Developing Portlets with ICEfaces** section in the **ICEfaces Developer's Guide**.

- While ICEfaces tests against and supports a number of the most popular portal containers, differences between implementations can still present incompatibilities. Check to see that the portal container you are using is supported or has been successfully used by somebody in the ICEfaces community.
- Because ICEfaces is a JSF-based framework, there are some aspects of the Portlet API that are accessed differently than they would be in a plain portlet application. This is important if you are porting legacy portlets or applications to run with ICEfaces.
- While it's possible to run ICEfaces portlets along with portlets that use other technologies or frameworks, the chances of incompatibilities can increase significantly. For best results, it is recommended to confine your portal application to a small, consistent set of technologies. The typical problem faced here is JavaScript conflicts because different frameworks use different libraries or different versions of the same library.
- The benefits provided by ICEfaces for enriching the user interface can be reduced through certain design decisions. For example, while it is possible to navigate between different views in a portlet, it is often a better choice to provide a different user interface abstraction, like tab panels, to provide different views into the same application in a single page.

Application Server Support

While the discussion in this **Getting Started Guide** focuses on the Apache Tomcat 6 application server, ICEfaces has been tested against most popular application servers. Refer to the **ICEfaces Release Notes** for a detailed list of application servers supported in this release.

If you are familiar with the application server on which you intend to deploy, it should be straightforward to deploy the ICEfaces sample application .war files to your environment and verify that ICEfaces is compatible with your specific installation. If an ICEfaces application fails to run in your environment, the problem is most often related to classpath conflicts. Resolve these conflicts to ensure that the JARs packaged in the sample application .war file are being used.

The exact combination of JARs required in your application classpath varies, depending on the application server you intend to deploy your ICEfaces application into, and the specific configuration you choose to use for your application (i.e., Facelets, JSP, icefaces.jar vs. just-ice.jar, etc.). Detailed library dependency information is available in the following sections of the **ICEfaces Developer's Guide**:

- **ICEfaces Library Dependencies** in Chapter 4
- **Appendix A ICEfaces Library/App. Server Dependencies**

Client Browser Support

ICEfaces applications can be deployed to the most popular desktop and mobile web browsers. The following browser families are supported:

Desktop Browsers

- Apple Safari
- Google Chrome
- Internet Explorer
- Mozilla Firefox
- Opera

Mobile Browsers

- Apple Safari (iPhone + iPod Touch)
- Blackberry Bold
- Opera Mobile

Refer to the **ICEfaces Release Notes** for a complete list of browser versions supported in this release.

Using ICEfaces with IDEs

ICEfaces supports integration plugins for many of the most popular Java IDE tools for web-application development, such as Eclipse, NetBeans, Genuitec MyEclipse, and Rational RAD. Refer to the **ICEfaces Release Notes** for a complete list of Java IDEs supported in this release.

> **Note:** ICEfaces tool integration plugins are available as separate downloads from **www.icefaces.org**.

ICEfaces can also be used with IDEs that do not have specific ICEfaces integration plugins available, such as IntelliJ, by manually configuring the ICEfaces libraries in your IDE as required by the application server you are deploying to at development time.

Index

R

S

T