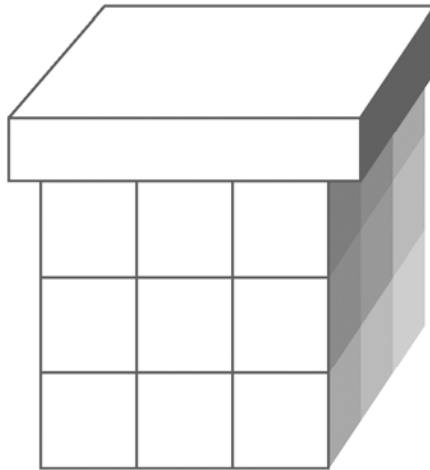


CHAPTER 4

ICEFACES COMPONENTS



OBJECTIVES

After completing “ICEfaces Components,” you will be able to:

- **Demonstrate familiarity with the range of components found in the ICEfaces library, and find and use components that suit a given page-authoring task.**
- **Use layout-management components to organize your JSF views for visual impact and usability.**
- **Choose built-in ICEfaces themes as CSS resources, customize those themes with your own stylesheets, or build entirely new themes yourself.**
- **Use popup windows to present modal and non-modal dialog boxes and message boxes to the user, asserting some control over client-side workflow.**
- **Add ICEfaces graphical data charts to your application.**

The Component Suite Showcase

- In this chapter we move from consideration of whole-framework features – partial submit, server push, etc. – to the wide range of UI components available in the ICEfaces library.
- This being an introduction to ICEfaces, we will not have time to work with each of these components in depth.
- Rather, we will consider groups of components by their makeup and feature set; and we will explore a few of the most important ones in hands-on exercises.
- A great resource for understanding what’s available in ICEfaces (and how to use it) is the Component Suite Showcase.

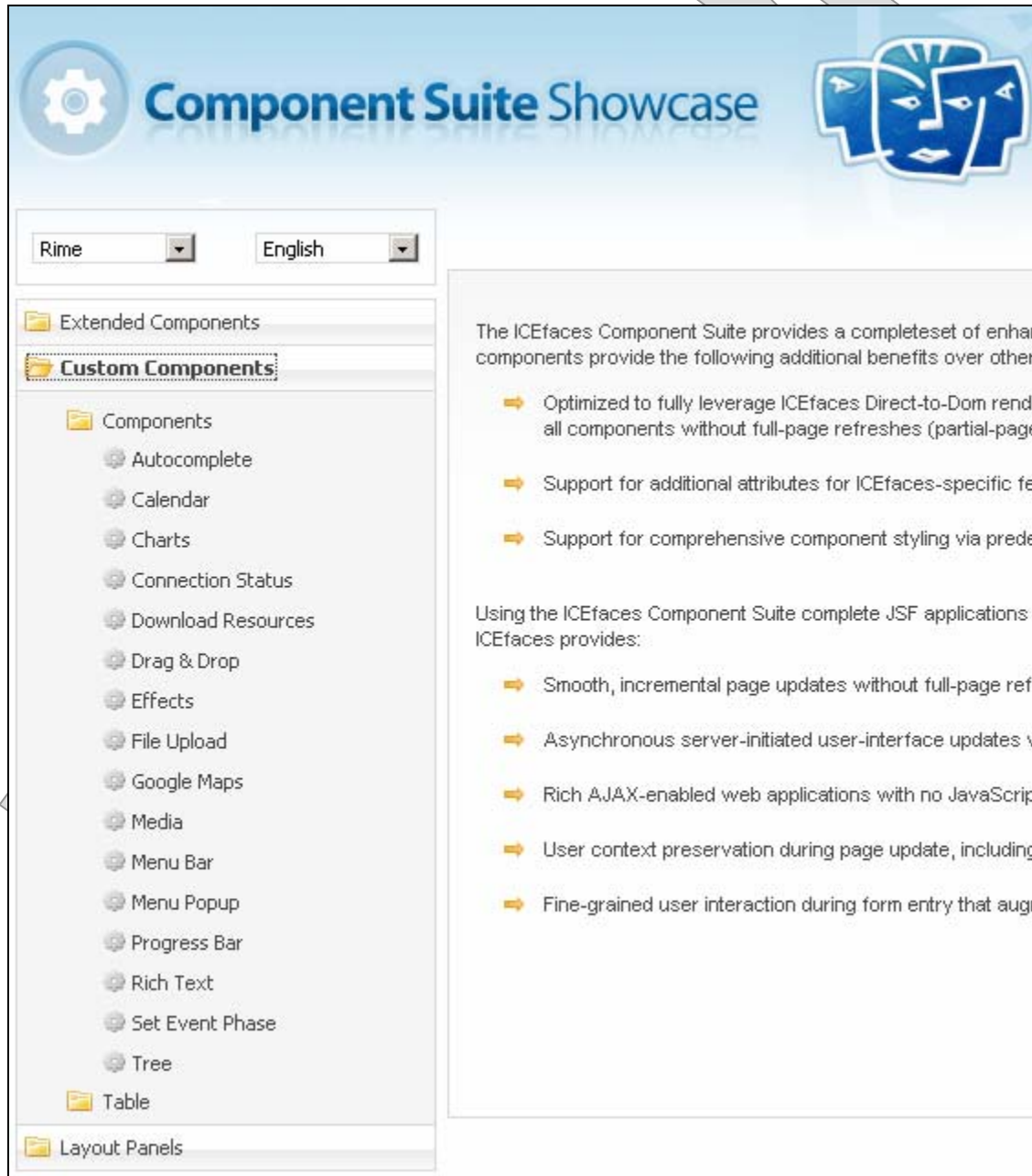
– This is available online at

[http://component-showcase.icefaces.org
/component-showcase/showcase.iface](http://component-showcase.icefaces.org/component-showcase/showcase.iface)

- It is also deployed as a WAR with ICEfaces, in the **samples** directory of the binary download. (This is one of the pieces that we sliced out of our **c:\Capstone\Tools\ICEfaces1.8** bundle.)
- Deploy the WAR to your favorite web server and browse the whole thing locally.
- The “Description” tabs for each component are also found separately assembled under **docs/components**; these have nice “cheat sheets” to remind you of typical usage and get you started with a given component in your own JSF views. (This is available on your lab machines under **c:\Capstone\Tools\ICEfaces1.8**.)

The Component Suite Showcase

- This is a handy reference for this chapter, so bring it up in a browser window now:



Component Suite Showcase

Rime English

- Extended Components
- Custom Components**
 - Components
 - Autocomplete
 - Calendar
 - Charts
 - Connection Status
 - Download Resources
 - Drag & Drop
 - Effects
 - File Upload
 - Google Maps
 - Media
 - Menu Bar
 - Menu Popup
 - Progress Bar
 - Rich Text
 - Set Event Phase
 - Tree
 - Table
- Layout Panels

The ICEfaces Component Suite provides a complete set of enhanced components that provide the following additional benefits over other components:

- ➔ Optimized to fully leverage ICEfaces Direct-to-Dom rendering, all components without full-page refreshes (partial-page updates).
- ➔ Support for additional attributes for ICEfaces-specific features.
- ➔ Support for comprehensive component styling via predefined styles.

Using the ICEfaces Component Suite complete JSF applications, ICEfaces provides:

- ➔ Smooth, incremental page updates without full-page refreshes.
- ➔ Asynchronous server-initiated user-interface updates via Ajax.
- ➔ Rich AJAX-enabled web applications with no JavaScript.
- ➔ User context preservation during page update, including stateful components.
- ➔ Fine-grained user interaction during form entry that augments the user experience.

Better Mousetraps: The Extended Components

- What ICEfaces calls its **extended components** are literally extensions of tools from the JSF RI, with more and improved features:
 - **Buttons**, including a nice image button
 - **Text fields**
 - **Selection components** such as checkboxes, radio buttons, lists, and dropdown boxes
- Mostly these are enhanced so that they can participate in general features such as partial submit, role-based security, and visual effects (which we'll discuss briefly at the end of this chapter).

- They all have additional attributes as follows:

`enabledOnUserRole`
`renderedOnUserRole`
`visible`
`disabled`
`partialSubmit`

- We've used the `<ice:commandButton>` a lot already.
- And we used an `<ice:selectOneListbox>` for our assisted-search facility in the previous chapter's first lab exercise.
 - That was critical, because we took advantage of the **partial submit** capability of the component.

New Tricks: The Custom Components

- Then, there is a much larger array of custom components, which are not extensions of familiar components built into JSF.
- So, they add UI features we didn't have before in any form:
 - **Calendars** (`<ice:inputDateSelect>`)
 - **Tables** (`<ice:dataTable>`) – oddly, this is not an extension of `<h:dataTable>`, but it mimics everything that component does and then adds many of its own capabilities
 - **Trees** (`<ice:tree>`)
 - **Menus** (`<ice:menuBar>` and `<ice:menuItem>`)
 - **Download** and **upload** helpers (`<ice:outputResource>` and `<ice:inputFile>`)
 - **Charts** (`<ice:outputChart>`)

Layout Management

- Now at quite a distance from what the JSF RI might have led us to expect in the way of UI features, ICEfaces also provides a nice set of **layout managers**, whose job it is to lay out other components visually – strikingly similar in nature to AWT **LayoutManagers**.
- These mostly do some heavy work with CSS and DOM event handling to organize window content:
 - **Border** and **split-pane panels** (`<ice:panelBorder>` and `<ice:panelDivider>`), which are right out of the AWT playbook
 - **Collapsible** and **positioned** panels (`<ice:panelCollapsible>` and `<ice:panelPositioned>`)
 - **Stacking** panels and **tab sets** (`<ice:panelStack>` and `<ice:panelTabSet>` / `<ice:panelTab>`)
 - **Popup panels, tooltips, and confirmation** boxes (`<ice:panelPopup>`, `<ice:panelTooltip>`, and `<ice:panelConfirmation>`) break out of 2D constraints by appearing to float content in front of the rest of the view
- All of these work as containers of other content, and it's impressive to see how easy it is to move chunks of a UI from one place to another:
 - From an **HTML table** layout to a **border** layout
 - From a **separate page** to a **popup dialog**, which is an exercise we'll try ourselves later in the chapter

- **The contents of these panels function the same in one place as in another, and have the same access to backing beans, events, etc.**

Evaluation
Only

Component Stylesheets

- Each of the components in the library will render HTML components with **class** attributes to connect to CSS.
- ICEfaces then bundles a set of visual themes that can be included by reference from page definitions – these are:
 - Rime (`rime.css`)
 - XP (`xp.css`)
 - Royale (`royale.css`)
- Each can be drawn into use on a page by referring to a stylesheet at a URL of the form:
`./xmlhttp/css/theme/theme.css`
 - So the **PersistentFacesServlet** hands these out on demand.
- You can do this in one of two ways:
 - A simple `<link>` with `rel` “stylesheet” and `type` “text/css”
 - Using `<ice:outputStyle>`
- The latter is the recommended approach, for a couple reasons:
 - The component can include **browser-specific** CSS content
 - The **theme** can be chosen **dynamically** using a value expression
- And, remember, the ‘C’ in CSS stands for **cascading**.
 - You’re always free to customize your pages by overriding or replacing the ICEfaces styles, too.

<ice:dataTable>

- Now we'll start to dig into a few specific examples of how to use ICEfaces components.
- The data table component does everything that the standard data table does, and then adds several nice features:
 - Plays with partial submit and server push, naturally
 - Fires **row selection events** and gives hover and selection cues
 - Observes a **dynamic column model** – a real shortcoming of the standard table when not both row and column counts are data-driven, or where the user might want to customize presentation
 - Can collaborate with a **paginator** (<ice:dataPaginator>) to present discrete “pages” of data
 - Can do **grouping, expand/collapse, and sorting**

Available Flights

EXAMPLE

- We've seen `<ice:dataTable>` a good bit already:
 - In the **HR** invitation page, where we used a table as a drop target
 - In the **Poll** application, showing poll results – and pushing new values to the table as the results were updated on the server
- We'll take a look back at another, earlier use, in the Flights application – see **Examples/Flights/ICEfaces**.
- **docroot/flights.jspx** presents two tables, one for outbound and one for inbound flights.
 - The outbound flight listing is shown in summary on the following page.
 - The **value/var/value** representation of source data is identical to the approach we'd use with `<h:dataTable>`.
 - But note the `<ice:rowSelector>`, which establishes a **selection model** for the table: not just an event handler for selection gestures, but also a value binding to represent selection state server-side and assure two-way communication about selection.

Available Flights

EXAMPLE

```
<ice:dataTable
  var="flight"
  value="#{reservations.outboundFlights}"
  styleClass="Flights"
>
  <f:facet name="caption" >...</f:facet>
  <ice:column>
    <ice:rowSelector
      id="selected"
      immediate="true"
      value=
        "#{reservations.selected[flight.number]}"
      selectionListener=
        "#{reservations.selectOutboundFlight}"
    />
    <f:facet name="header" >...</f:facet>
    <h:outputText value="#{flight.number}" />
  </ice:column>
  <h:column>
    <f:facet name="header" >...</f:facet>
    <h:outputText value="#{flight.departure}" />
  </h:column>
  <h:column>
    <f:facet name="header" >...</f:facet>
    <h:outputText value="#{flight.arrival}" />
  </h:column>
</ice:dataTable>
```

Available Flights

EXAMPLE

- Also note that in this example we both use a built-in ICEfaces theme and tweak it a bit to fit our own presentation.

- At the top of the page, see how we establish a cascade of stylesheets:

```
<head>
  <title>Airline Reservations</title>
  <link rel="stylesheet" href=
    "../xmlhttp/css/xp/xp.css" type="text/css" />
  <link rel="stylesheet" href="airports.css"
    type="text/css" />
</head>
```

- Notice that we make direct reference to the `xp.css` sheet.
- In other examples we've used `<ice:outputStyle>`, and in order to do so we need the whole HTML tree to be part of the `<f:view>`. Watch for this tactic in upcoming examples.

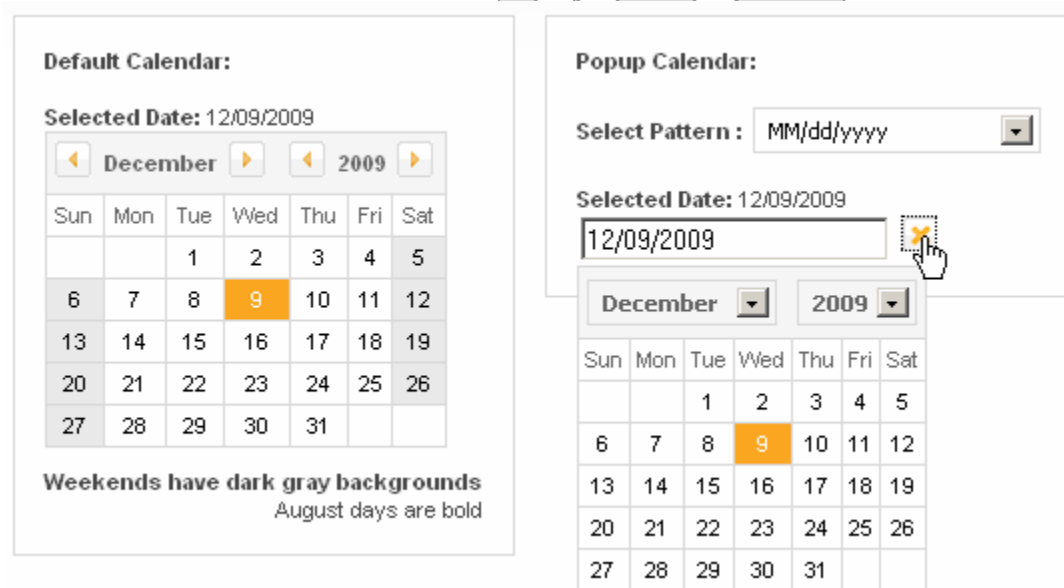
- So the XP theme defines most of our look.
- But we override some of the CSS styles for the purpose of tuning the row-selection and row-hover colors in the tables – see the bottom of `docroot/airports.css`:

```
.iceRowSelSelected      { background-color: #0af; }
.iceRowSelMouseOver    { background-color: #0ff; }
.iceRowSelSelectedMouseOver
                        { background-color: #0ee; }
```

- You can do as much or as little of this as you want, or build a theme entirely yourself.

<ice:selectDateInput>

- The calendar control `<ice:selectDateInput>` can present a date-entry calendar UI, either embedded in the enclosing panel or as a popup triggered by a small button to the side of a text field.
 - From the Component Suite Showcase:



- The text-field entry is customizable to a pattern mask, which will automatically be used as if an `<f:dateConverter>` were attached to the component.
- By specifying a pattern that includes time information, you can also prompt the component to present a time-editing UI.

Popup Calendars

DEMO

- We'll now start on a series of demonstrations, each of which will result in an enhancement to the UI of the LandUse application.
 - This is yet another old friend from the JSF course.
 - The functionality is nearly identical, but as you'll see the summary and detail pages have been reworked considerably to take best advantage of ICEfaces rich-UI features.
 - So our starting point is a little farther along the road than where we were the last time we checked in.
- In this demonstration, we'll convert three date-editing fields from standard text inputs to ICEfaces popup calendars.
 - Do your work in **Demos/Calendar**.
 - The completed demo is in **Examples/LandUse/Step2**.

Popup Calendars

DEMO

1. Build and test the starter application at the following URL:

<http://localhost:8080/LandUse>

USFS Land-Use Proposals

Below is a list of recent and current land-use proposals being considered by the USFS.

Applicant	Parcel	Proposal
Mines-R-Us	Tonto NF	Silver mining
Cranmore Paper	White Mountains NF	Selective logging
Ski USA	Green Mountain NF	Alpine park

2. Select a proposal from the table and click **Edit**.

Applicant: Mines-R-Us **Parcel:** Tonto NF

Application date: 9/17/06

Proposed use: Silver mining

Proposed start date: 2/1/07

Proposed end date: 7/30/08

Contributor	Date	Recommendation	Comment
Sierra Club	9/26/06	REJECT	We believe the small industrial benefit to be found from re-opening this defunct site is far outweighed by the adverse ecological impact.
Ernest	9/27/06	REJECT	I've seen the runoff and other impacts from this mine before -- I

Popup Calendars

DEMO

3. Note the input fields for application date and proposed start and end dates.
4. Open `docroot/detail.jspx`, and find the first of these components in the tree:

```
<td>Application date:</td>
<td><ice:inputText
    id="applicationDate"
    label="application date"
    value=
        "#{DB.selectedProposal.applicationDate.time}"
    required="true"
    >
    <f:convertDateTime pattern="M/d/yy" />
</ice:inputText></td>
```

5. Change this to be a popup calendar field:

```
<td><ice:selectInputDate
    id="applicationDate"
    label="application date"
    value=
        "#{DB.selectedProposal.applicationDate.time}"
    required="true"
    popupDateFormat="M/d/yy"
    renderAsPopup="true"
    /></td>
```


<ice:panelTooltip>

- The ICEfaces tool-tip component, `<ice:panelTooltip>`, will float a small informational window over another component when the mouse hovers there for a short time.
 - Wrap the target component in an `<ice:panelGroup>`.

```
<ice:panelGroup>
  <ice:whatever />
  <h:whatever />
  <p>Whatever whatever ...</p>
</ice:panelGroup>
```

- Define this component as a sibling or cousin of the target component. (You might have expected that it would be a child of the target, but no. Probably the ICEfaces team wanted to allow for multiple components to use a single tooltip.)

```
<ice:panelTooltip id="myTooltip" >
  <f:facet name="body" >
    <h:outputText value="Things to know ..." />
  </f:facet>
</ice:panelTooltip>
```

- The component observes two possible facets, “header” and “body”.
- Refer to it from the target panel group using the attribute **panelTooltip**:

```
<ice:panelGroup panelTooltip="myTooltip" >
  <ice:whatever />
  <h:whatever />
  <p>Whatever whatever ...</p>
</ice:panelGroup>
```

<ice:panelPopup>

- A more ambitious component allows you to show pop-up windows for any amount of time, and that can accept user input: this is `<ice:panelPopup>`.
- This too is defined separately from the component that might trigger it; in fact it can be anywhere in the view and is usually at the top level, wrapped in its own `<ice:form>`.
- But rather than being identified by an attribute of a target component, it is a piece of the view presentation that can be switched on and off by manipulating its **visible** attribute.
- So any component on the page might trigger it, by firing an event whose handler would change the state of a backing-bean property to which **visible** were bound.
 - It could even be activated asynchronously, if that value were to change as part of a view refresh.
- When made visible, the popup window can either be **modal** or non-modal. A modal popup functions like a dialog box, and the rest of the page is disabled until the user closes the popup.
- Popup windows also have header and body facets.
- Any command components will of course trigger event handling and an HTTP roundtrip, so that the rest of the page has the opportunity to update based on whatever actions the user has taken – for instance by entering information into the dialog.

Modal Dialogs

DEMO

- Let's continue working with the LandUse application by adding a modal dialog to the detail page.
 - Currently there is a separate page **addComment.jspx** to which the user navigates in order to add either a “public” or a “professional” comment.
 - We'll draw that functionality into **detail.jspx**.
 - You can either pick up your work from the previous demo or start fresh in **Demos/Dialog**.
 - The completed demo is in **Examples/LandUse/Step3**.
1. If you like, test out the current functionality by choosing a proposal, clicking **Edit**, and then clicking either **Add Public Comment** or **Add Professional Comment**:

Contributor	Date	Recommendation	Comment
Zeldon Shelbow	10/25/06	REJECT	This is going to ruin the view from several prominent hiking trails, and I don't see the benefit.

Add a Public Comment

Modal Dialogs

DEMO

Proposal

Applicant: Cranmore Paper
 Affected parcel: White Mountains NF
 Proposed use: Selective logging

Your Comment

Your name:

Recommendation:

Comment:

- Fill in the form as shown above, and click **Submit**. The comment is added to the appropriate table in the detail view:

Contributor	Date	Recommendation	Comment
Zeldon Shelbow	10/25/06	REJECT	This is going to ruin the view from several prominent hiking trails, and I don't see the benefit.
Alison Geyser	12/9/09	REJECT	The logging trucks clog up traffic on the only through-road in the area.

Modal Dialogs

DEMO

3. Open `detail.jspx` and define a new popup panel – place this at the bottom of the HTML body, in its own form:

```
</ice:form>

<ice:form id="popup" >
  <ice:panelPopup
    id="addComment"
    modal="true"
    draggable="false"
    autoCentre="true"
    style="border: ridge #0c0;"
  >
  </ice:panelPopup>
</ice:form>

</body>
```

- So, this will be a modal dialog that is automatically centered in the browser window, and that can't be repositioned by the user.

4. Define a “body” facet for the panel:

```
<ice:panelPopup
  id="addComment"
  modal="true"
  draggable="false"
  autoCentre="true"
  style="border: ridge #0c0;"
>
  <f:facet name="body" >
  </f:facet>
</ice:panelPopup>
```

Modal Dialogs

DEMO

5. Open `addComment.jspx`, and find the form content under the heading “Your Comment”. Copy the entire `<table>` that is the child of this form to the clipboard.
6. Paste this content in as the body facet of your popup panel:

```
<f:facet name="body" >
  <table>
    <tr>
      <td>Your name:</td>
      <td><ice:inputText
        id="contributor"
        value="..."
        required="true"
      /></td>
      ...
    </tr>
  </table>
</f:facet>
```

7. Now, bind the **visible** state of this popup to the **addingComment** property of the backing bean:

```
<ice:panelPopup
  id="addComment"
  modal="true"
  draggable="false"
  autoCentre="true"
  style="border: ridge #0c0;"
  visible="#{DB.addingComment}"
>
```

- This property is defined, but so far it is always **false**.

Modal Dialogs

DEMO

8. You could build and test at this point, by the way – you'd see nothing new since the **visible** property of the popup can't ever be **true**.
9. Open `src/gov/usda/usfs/landuse/web/DatabaseWrapper.java`, and find the method **openPublicComment**.
10. Change this in two ways: first, set the **addingComment** field to **true**:

```
public String openProfessionalComment ()
{
    commentType = "professional";
    addingComment = true;
    return "comment";
}
```

- The field is already defined, along with an **isAddingComment** method.

11. Now, have the method return an empty string, so that it no longer triggers the JSF page navigation that would show the separate **addComment.jspx** page.

```
return "";
```

12. Make the same changes to **openProfessionalComment**.

Modal Dialogs

DEMO

13. Build and test now, and see that when you click either of the add-a-comment buttons, the modal dialog appears, with a UI nearly identical to what we saw in **addComment.jspx**:

The screenshot shows a modal dialog box titled "Add a Public Comment" overlaid on a web application. The background application displays "Applicant: Cranmore Paper" and "Parcel: White Mountains NF". The modal dialog contains the following fields and controls:

- Your name:** A text input field containing "UNKNOWN".
- Recommendation:** A dropdown menu with three options: "Accept", "Reject", and "Neutral".
- Comment:** A large text area for entering a comment.
- Buttons:** "Submit" and "Cancel" buttons at the bottom of the dialog.

Below the dialog, a button labeled "Add a Public Comment" is visible on the main application page.

14. But there's a problem – you can't clear the dialog!

- If you test this out, and look carefully, you'll see that it's not a matter of failing to fire an event: when you submit a new, valid comment, the table on the main window does indeed grow to show the additional row, even though the UI is disabled.

15. Close the browser.

Modal Dialogs

DEMO

16. Of course the issue is that we've added code to set the **addingComment** flag, but nowhere do we reset it. Find the method **noComment**, which handles the **Cancel** button. **addComment** handles the **Submit** button and also delegates to this method to clean up and navigate. So, change things here:

```
public String noComment ()
{
    possibleComment = new Comment ();
    addingComment = false;

    return "detail";
}
```

17. If you test now, you should see that you can add comments from the dialog, or simply hit **Cancel**, and the dialog does clear away again, leaving the main UI enabled and updated appropriately.

- In the answer code to this demonstration, the **addComment.jspx** and the navigation rules and cases that relate to it have been removed.

<ice:panelTabSet>

- Another useful layout tool is the tabbed pane, which ICEfaces implements using `<ice:panelTabSet>` and `<ice:panelTab>`.
- You can do a lot of slick things with these controls, but the big benefit usually comes in the form of reclaimed screen real estate, and that's easy to achieve:
 - Set up an `<ice:panelTabSet>`.
 - Create one `<ice:panelTab>` for each tab that you want to include in the set.
 - Give the tag control a **label** attribute, which will show in the graphical tab “header” drawn by the tab-set.
 - Pour your content into that tab control.
 - Repeat for as many tabs as you want.
- Then, you can get fancy with tab placement, images and complex content in the tab headers, etc.
- You can also bind things like tab order, visibility, and selection to dynamic expressions.
- There is a server-side API to the tab set's state, allowing you to manipulate these things from action methods.

Organizing with Tabs

DEMO

- The final step in our string of enhancements to the LandUse application will bring four major areas of UI content in the detail page into a consolidated tab set.
 - Continue your work from the previous demonstration, or start fresh in **Demos/Tabs**.
 - The completed demo is in **Examples/LandUse/Step4**.
1. Open **detail.jspx** and find the horizontal rule that separates the top two input fields from the rest of the page:

```
...<td colspan="2" ><ice:message  
    for="affectedParcel"  
    errorClass="errorMessage"  
/></td>  
  
</tr>  
</table>  
  
<hr width="80%" />  
  
<table class="List" >  
  <tr>  
    <td>Application date:</td>  
    <td><ice:selectInputDate ...
```

Organizing with Tabs

DEMO

2. Replace this with the start tag for a tab-set panel, and place the end tag before the two command buttons “Submit” and “Cancel”.

```
</table>
```

```
<ice:panelTabSet id="tabs" height="240px" >
```

```
    <table class="List" >
```

```
    ...
```

```
    </table>
```

```
</ice:panelTabSet>
```

```
<p>
```

```
    <ice:commandButton
    value="Submit" ...
```

- So the tab set is going to encompass nearly all of the existing UI, leaving just a header of two labeled input fields and a footer of two submit buttons.

3. Now find the table that holds the four values application date, proposed use, start and end dates, and bracket it in a tab panel. Give the tab an **id** and a simple, user-friendly **label**:

```
<ice:panelTabSet id="tabs" height="240px" >
```

```
    <ice:panelTab id="proposalTab" label="Proposal" >
```

```
        <table class="List" >
```

```
        ...
```

```
        </table>
```

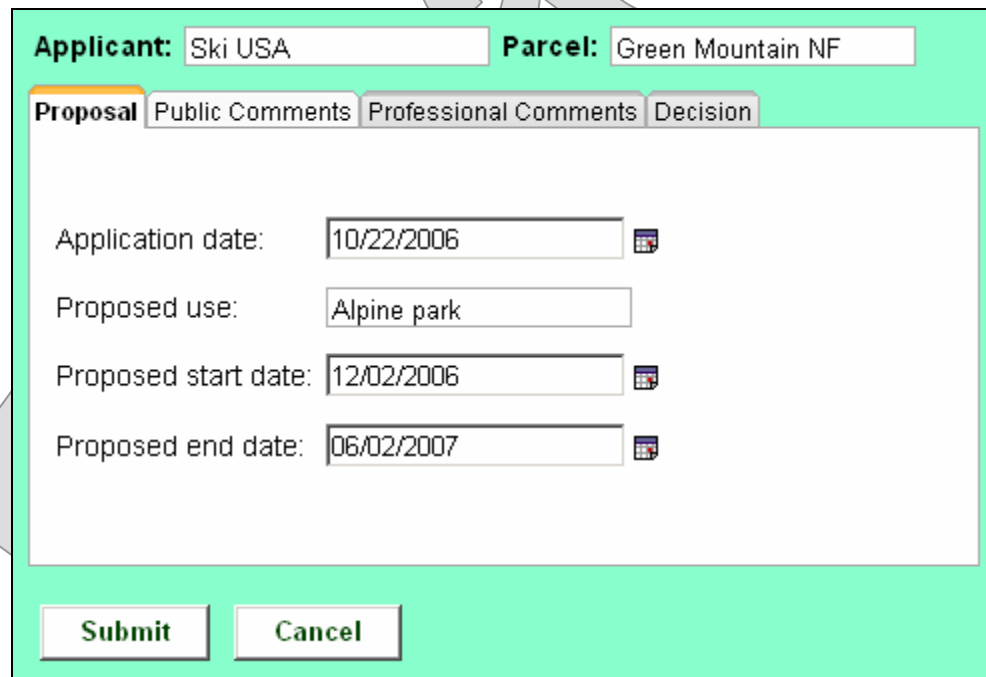
```
    </ice:panelTab>
```

```
</ice:panelTabSet>
```

Organizing with Tabs

DEMO

- Now wrap each of the other three content areas in the same way:
 - The **public-comments** section: you just need the ICEfaces table and the command button that follows
 - The **professional-comments** section – same story here
 - The input fields for making a **decision** on the proposal, which are gathered in a final HTML table
- Build and test the application, and see that the page has been reorganized completely:



The screenshot shows a web application window with a light green border. At the top, there are two input fields: "Applicant:" with the value "Ski USA" and "Parcel:" with the value "Green Mountain NF". Below these is a tabbed interface with four tabs: "Proposal" (selected), "Public Comments", "Professional Comments", and "Decision". The "Proposal" tab is active and contains a form with four rows of input fields, each with a calendar icon to its right:

- Application date: 10/22/2006
- Proposed use: Alpine park
- Proposed start date: 12/02/2006
- Proposed end date: 06/02/2007

At the bottom of the form are two buttons: "Submit" and "Cancel".

- You can try out the various page functions, and see that they are unaffected – the tabs as we've used them just arrange things visually, though it's possible to wire tab-selection events to data updates, events in contained controls to tab navigation, etc.

Visual Effects

- ICEfaces also offers a set of common component attributes that can apply various visual effects.
- These are derived from the **script.aculo.us** JavaScript library, and range from the subtle to the truly silly.
- Depending on your aesthetic sensibility some of them may work for you while others do not.
- Identify a desired effect by binding the appropriate attribute to a property on some backing bean.
 - The attributes are of the form **XXXeffect** where XXX is an HTML event-handling attribute – e.g. **onclickeffect**, **onmouseovereffect**.
 - The value must be an instance of the **Effect** class, and for each event type there is a pre-built subtype of **Effect**, so it's easy to farm out instances of **Highlight**, **Pulsate**, etc.

<ice:outputChart>

- A final component to consider is `<ice:outputChart>`, which calculates and renders a graphical chart of some input data.
- This is quite a departure from the rest of the library, and it can be a nice finishing touch for a page that presents tabular data.
 - Also visually impressive is the `<ice:gMap>` component, and integrating Google Maps is a great feature.
 - But there's not much there we haven't seen before.
- This component is a bit of a bear to manage, with many supported chart types, attributes to customize the appearance of the chart, and lots of constraints on the input data model.
- When it does what you need it to do, it works very nicely.
- We'll explore this last component by way of a lab exercise to end the chapter.

Adding a Bar Chart

LAB 4

Suggested time: 30-45 minutes

In this lab you will enhance the Poll application by adding a bar chart of poll results to the final page.

Detailed instructions are found at the end of the chapter.

Evaluation
Only

SUMMARY

- **As we suggested at the beginning of the chapter, there's a lot more to the ICEfaces library than we have time to cover in great detail.**
- **But the sampling of components that we've used in this chapter has included components that are**
 - Highly useful in and of themselves
 - Typical of the programming model across the board
- **And by now you should have a strong sense of the overall power and productivity of the ICEfaces framework:**
 - A nice toolkit of high-quality, visually appealing UI components
 - Easy-to-use Ajax functionality, including synchronous requests and server-initiated “push” capabilities
 - Rich-UI features such as layout management, drag-and-drop, and eye-catching visual effects
 - Security features including role-based presentation and built-in safeguards against cross-site scripting and injection attacks

Adding a Bar Chart

LAB 4

In this lab you will enhance the Poll application by adding a bar chart of poll results to the final page.

Lab workspace:	Labs/Lab4
Backup of starter code:	Examples/Poll/Step2
Answer folder(s):	Examples/Poll/Step3
Files:	docroot/completed.jspx src/cc/poll/web/Wrapper.java

Instructions:

1. Open **completed.jspx** and add a chart to it, right at the end of the HTML body:

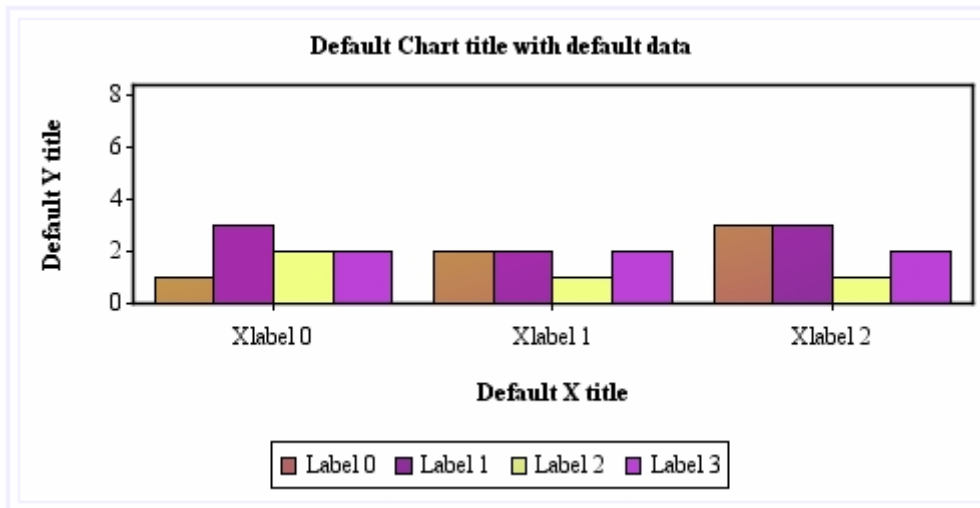
```
<ice:outputChart
  id="barChart"
  type="barclustered"
  data="1, 2, 3: 3, 2, 3: 2, 1, 1: 2, 2, 2"
  width="480"
  height="240"
  style="border: solid grey 1px;"
/>
</body>
```

This is a minimal invocation of the chart component, just to get something on the screen that we can work with. The **data** attribute is a placeholder: the component is willing either to work through a **List<double[]>** or to parse a string like the one above, and we'll start with the string but move to binding this component to our real poll data in a moment.

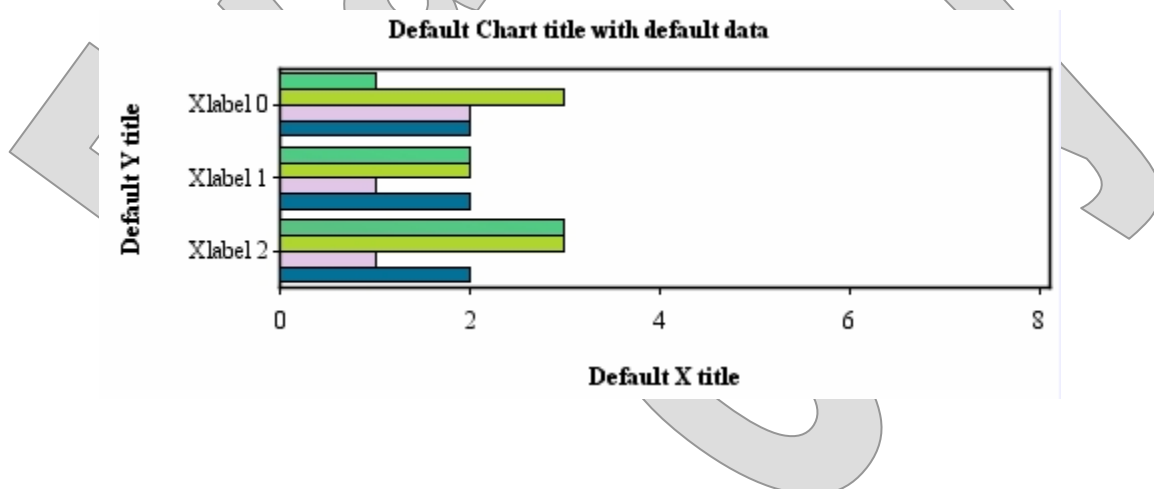
Adding a Bar Chart**LAB 4**

2. Build and test the application. Work through the poll questions and then see your completed poll table followed by this static chart:

<http://localhost:8080/Poll/Ajax>



3. Now we'll work through several quick additions and changes to the attributes on the new component, and build and retest in a few quick cycles to get the chart more in the shape we want it. First, add a **horizontal** attribute with the value **true**. Testing this change shows that the bar chart has rotated; this is more in line with the way we present the data in the table above:

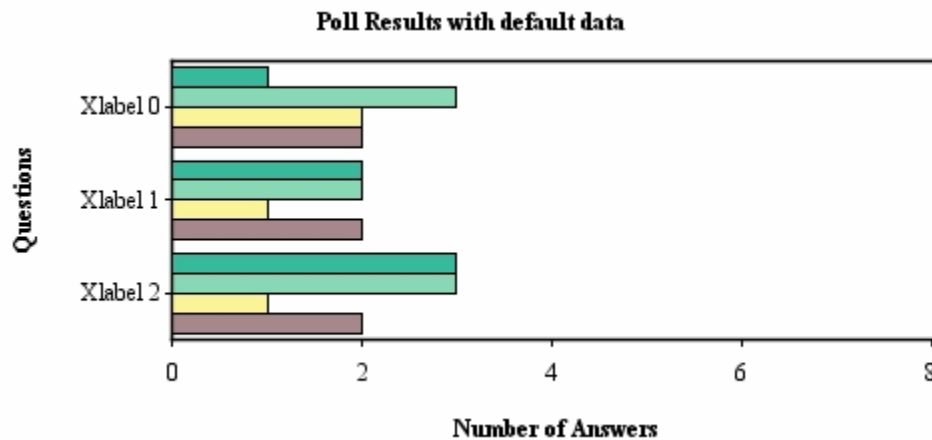


Adding a Bar Chart**LAB 4**

4. Now add the following attributes and values:

```
<ice:outputChart
  id="barChart"
  type="barclustered"
  data="1, 2, 3: 3, 2, 3: 2, 1, 1: 2, 2, 2"
  width="480"
  height="240"
  style="border: solid grey 1px;"
  horizontal="true"
  chartTitle="Poll Results"
  legendPlacement="none"
  yaxisTitle="Questions"
  xaxisTitle="Number of Answers"
/>
```

5. Test again, and see that you have control over the captions and titles:



6. Set colors; you can give this a quick test, but we don't reproduce the results here since they won't look any different in black and white.

```
yaxisTitle="Questions"
xaxisTitle="Number of Answers"
colors="red, green, blue, yellow"
/>
```

7. Now, let's bind the chart to some real data. Change **data** to bind to the **data** property on the **wrapper** bean, getting rid of the hard-coded list of numbers.

Note: The next few steps involve adding backing properties to the **Wrapper** class to support the chart. If you prefer to focus on the page-authoring aspect of this exercise, just copy the answer version of **Wrapper.java** into place, and skip ahead to the step where we add an **xaxisLabels** attribute to the chart component.

Adding a Bar Chart**LAB 4**

8. Open **Wrapper.java** and add a method **getData** that returns a list of arrays of type **double** – that is:

```
public List<double[]> getData ()
{
}
```

This method will have to read the poll data and prime it into a form that the chart control can read. The ideal tactic here would probably include a custom implementation of **List** that could read information on-demand, so that we wouldn't have to spend a lot of time and memory copying information into a temporary data structure as we're about to do. But that's more pure Java programming practice, and for this lab the simple approach will take less time and still illustrate the interface to the component.

9. Start the implementation by initializing a local variable **data** to a new **ArrayList** of double arrays.

You'll need to import **List** and **ArrayList** from **java.util**.

10. Because the polling is carried out by servlets, we can't just dependency-inject the poll results and read them. Instead, get the external JSF context and find two objects as application and session attributes:

```
PollResults results = (PollResults)
    FacesContext.getCurrentInstance ().getExternalContext ()
        .getApplicationMap ().get ("results");

LivePoll poll = (LivePoll)
    FacesContext.getCurrentInstance ().getExternalContext ()
        .getSessionMap ().get ("poll");
```

Import **javax.faces.context.FacesContext**. Import **PollResults**, **LivePoll**, and **QuestionResults** (used in the next step) from **cc.poll**.

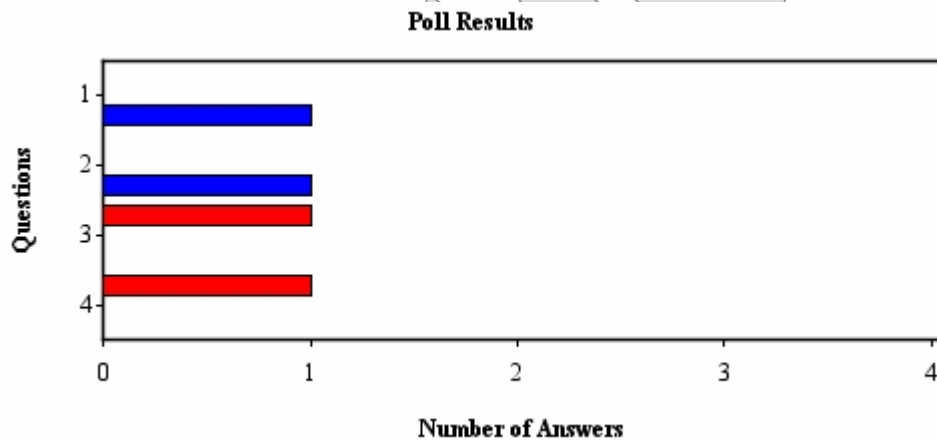
11. Define an outer loop over all the questions in the poll:

```
for (QuestionResults QandA : results.get (poll.getData ().getName ()))
{
}
```

12. Inside that loop, allocate an array of **doubles** called **counts**, whose length will be the **size** of the **QandA answerCount** property, which is a list of integers.
13. Write an inner loop to copy all the answers from the **answerCount** list into **counts**.
14. After the inner loop, add **counts** to the **data** list.
15. After the outer loop, just **return data**.

Adding a Bar Chart**LAB 4**

16. While we're here, create a second method **getLabels**, which returns a list of strings.
17. Implement this to return a list whose length is the same as the list returned by **getData**, and whose values are string representations of integers starting at one, so that the first element is "1", the second is "2", and so on. You'll need to derive the same **poll** reference that you got in the **getData** method.
18. Add an **axisLabels** attribute to bind to the **label** property you just implemented.
19. Build and test the application, and you should see real data in the chart now:



20. You may have noticed that when the table accepts pushed updates, the chart does not change. That's the one remaining piece for us to fix: add an attribute **renderOnSubmit** and set it to **true**.

Adding a Bar Chart**LAB 4**

21. Test, and now you should see your chart updating smoothly each time a new answer comes in on the server side.

