```
        return "97402".equals(zip) ? "Eugene,Oregon" : "NO DATA,NO DATA";
    }
}
```

Ajax Components

Ajax is cool, but implementing—and especially reimplementing and debugging—low-level Ajax code is not cool. To rid ourselves of that burden entirely, we now turn to JSF custom components, which happen to be an excellent vehicle for encapsulating Ajax code. Once our custom components are encapsulated, we can use them via JSP tags to create compelling user experiences.

Hybrid Components

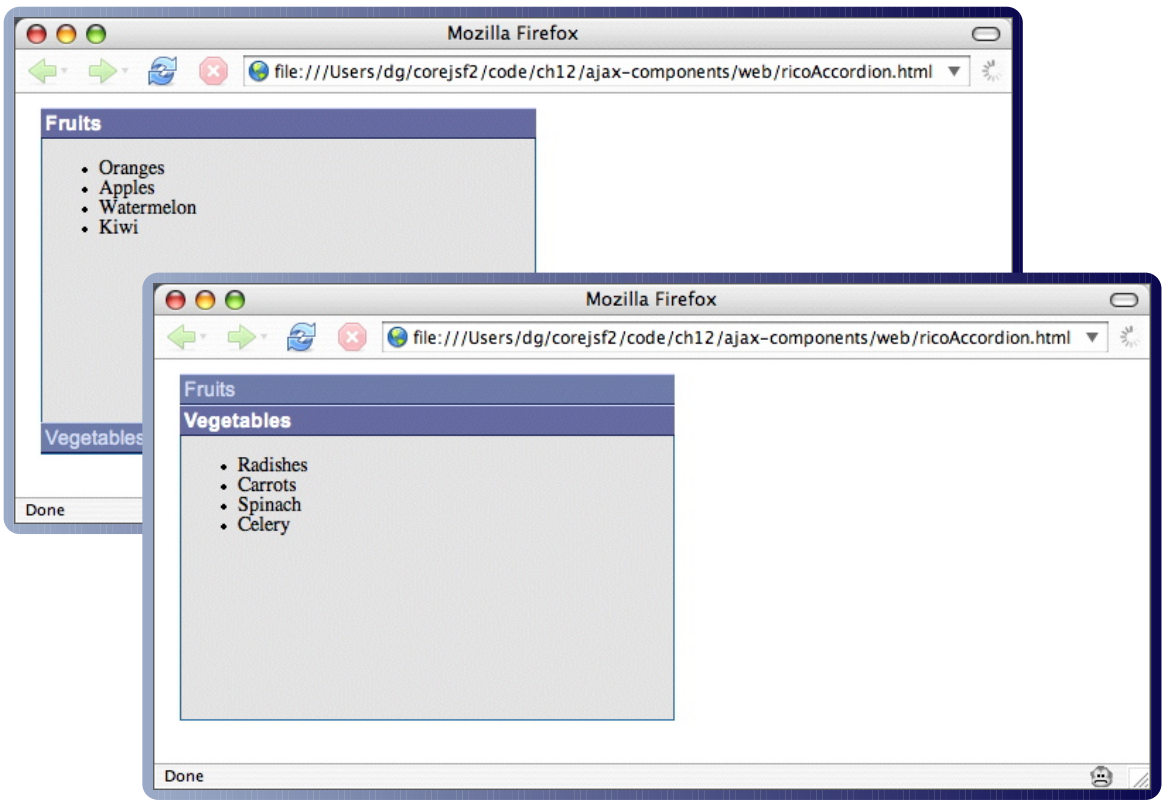
It should be fairly obvious that the road to Ajax bliss can be paved by implementing custom renderers that emit JavaScript code.

Even more interesting, however, are JSF components that wrap existing JavaScript components. After all, why would you want to implement components such as accordions (à la Flash) or drag and drop, from scratch, when you have a wide variety of existing components to choose from, such as Prototype, Scriptaculous, Dojo, and Rico? Wrapping those components with JSF components so that you can use them in your JSF applications is a straightforward task.

The Rico Accordion

Rico is a one of a number of frameworks based on Prototype. Rico provides amenities such as drag and drop and a handful of useful components. One of those components is an accordion, in the Flash tradition, shown in Figure 7.

FIGURE 7
The Rico Accordion



The Rico Accordion component is similar to a tabbed pane with fancy transitions—when you click on the header of an accordion panel, the header animates either up or down to reveal its associated panel. Here's how you implement the accordion, shown in Figure 7, using HTML:

```
<html>
   <head>
      <link href="styles.css" rel="stylesheet" type="text/css"/>
      <script type='text/javascript' src='prototype.js'></script>
      <script type='text/javascript' src='rico-1.1.2.js'></script>
      <script type='text/javascript'>
         function createAccordion() {
            new Rico.Accordion($("theDiv"));
         }
      </script>
   </head>

   <body onload="createAccordion();">
      <div id="theDiv" class="accordion">
         <div class="accordionPanel">
            <div class="accordionPanelHeader">
              Fruits
            </div>

            <div class="accordionPanelContent">
              <ul>
                 <li>Oranges</li>
                 <li>Apples</li>
                 <li>Watermelon</li>
                 <li>Kiwi</li>
              </ul>
```

```
            </div>
         </div>

         <div class="accordionPanel">
            <div class="accordionPanelHeader">
              Vegetables
            </div>

            <div class="accordionPanelContent">
              <ul>
                 <li>Radishes</li>
                 <li>Carrots</li>
                 <li>Spinach</li>
                 <li>Celery</li>
              </ul>
            </div>
         </div>
      </div>
   </body>
</html>
```

When the preceding page loads, Rico creates an instance of `Rico.Accordion`, which adds behaviors to the DIV that it's passed. In this case, Rico endows the DIV with JavaScript event handlers that react to mouse clicks in the header of each panel.

In the next section, we'll see how to wrap the Rico Accordion in a JSF component.

The JSF-Rico Accordion Hybrid

The application shown in Figure 8 is a hybrid component, meaning a JSF component that wraps a JavaScript component—in this case, the Rico Accordion component.

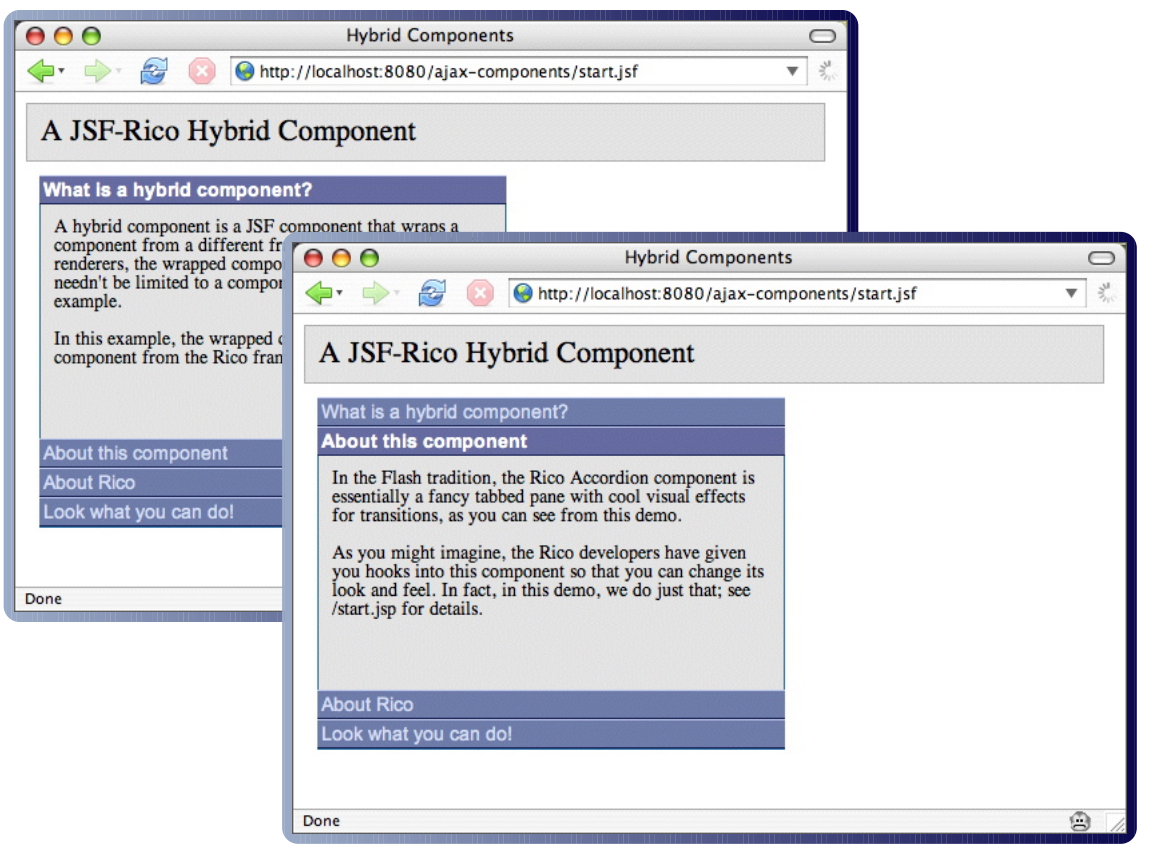
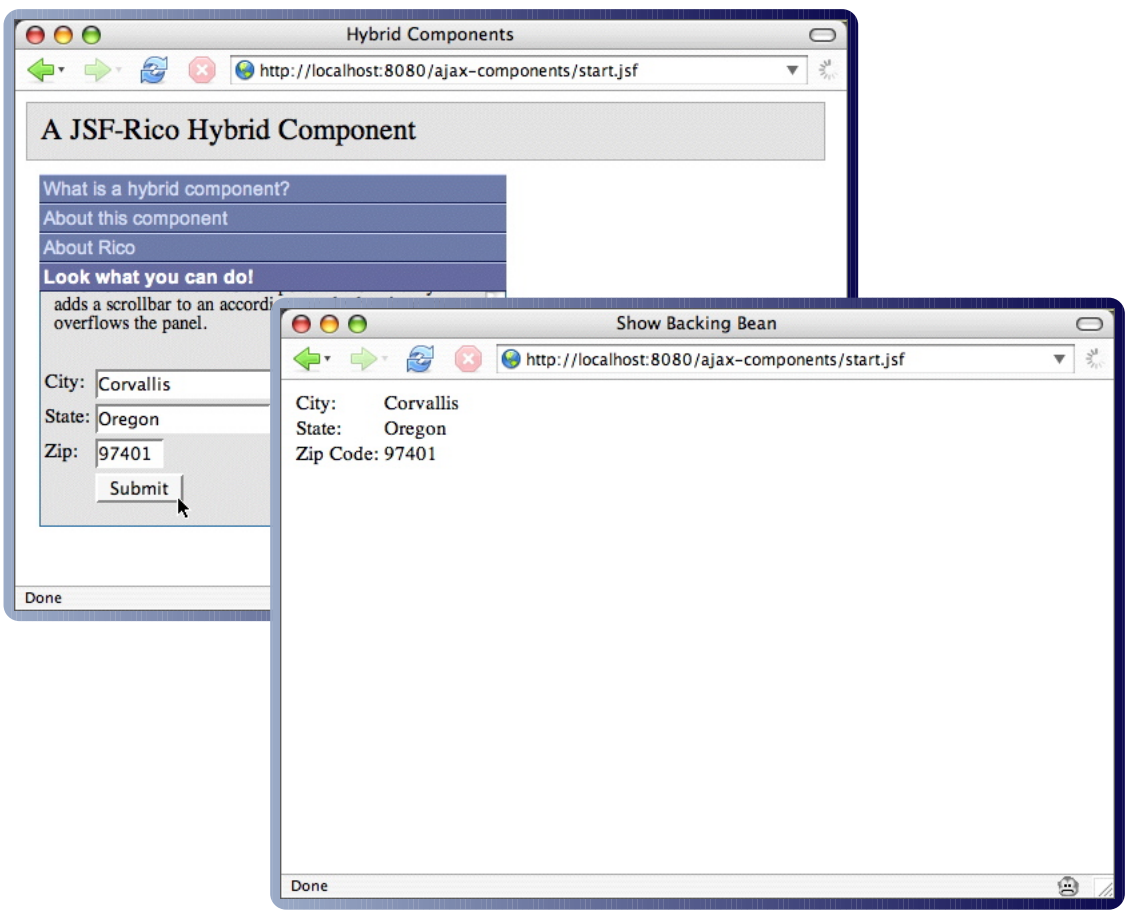


FIGURE 8
A JSF component that wraps a Rico Accordion component

The Rico component automatically adds a scroll bar if the content of a panel overflows the size of the panel, so we get that functionality for free. As Figure 9 illustrates, you can put anything you want in an accordion panel, including forms.

FIGURE 9
You can put forms inside accordion panels



Using the accordion component is simple:

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

   <f:view>
       ...
            <rico:accordion     name="bookAccordion"
                         panelHeight="175"
                          styleClass="accordion"
                          panelClass="accordionPanel"
                         headerClass="accordionPanelHeader"
                        contentClass="accordionPanelContent">
              <rico:accordionPanel heading="#{msgs.whatIsAHybrid}">
                 <jsp:include page="/whatIsAHybrid.jsp"/>
              </rico:accordionPanel>

              <rico:accordionPanel heading="#{msgs.aboutThisComponent}">
                 <jsp:include page="/aboutTheAccordion.jsp"/>
              </rico:accordionPanel>

              <rico:accordionPanel heading="#{msgs.aboutRico}">
                 <jsp:include page="/aboutRico.jsp"/>
              </rico:accordionPanel>
```

```
                <rico:accordionPanel heading="#{msgs.lookWhatYouCanDo}">
                   <jsp:include page="/lookWhatYouCanDo.jsp"/>
                </rico:accordionPanel>
             </rico:accordion>
             ...
   </f:view>
</html>
```

The `rico:accordion` and `rico:accordionPanel` tags represent custom renderers that we pair with `UICommand` components. Those renderers generate the Rico-aware JavaScript that creates the Rico Accordion.

The Rico-aware renderers do two things you may find useful if you decide to implement JSF components with Ajax capabilities of your own: They keep their JavaScript separate from their renderers, and they transmit JSP tag attributes to that JavaScript code.

Keeping JavaScript Out of Renderers

One thing quickly becomes apparent if you start implementing Ajax-enabled custom components: You don't want to generate JavaScript with `PrintWriter.write` statements. It's much easier to maintain JavaScript if it's in a file of its own. Finally, it's convenient to co-locate JavaScript files with the Java classes that use them. Let's see how we can do those things.

The `AccordionRenderer` class generates a `script` element whose `src` attribute's value is `rico-script.jsf`:

```
public class AccordionRenderer extends Renderer {
    public void encodeBegin(FacesContext fc, UIComponent component)
```

```
            throws IOException {
        ResponseWriter writer = fc.getResponseWriter();
        // write the script for loading the Rico JS file
        writer.write("<script type='text/javascript'"
                            + "src='rico-script.jsf'>"
                    + "</script>");
        ...
    }
}
```

That `src` attribute—`rico-script.jsf`—results in a call to the server with the URL `rico-script.jsf`. That URL is handled by a phase listener:

```
public class AjaxPhaseListener implements PhaseListener {
    private static final String RICO_SCRIPT_REQUEST = "rico-script";
    private static final String PROTOTYPE_SCRIPT_FILE = "prototype.js";
    private static final String SCRIPTACULOUS_SCRIPT_FILE = "scriptaculous.js";
    private static final String RICO_SCRIPT_FILE = "rico-1.1.2.js";

    public PhaseId getPhaseId() {    // We need access to the view state
        return PhaseId.RESTORE_VIEW; // in afterPhase()
    }
    public void beforePhase(PhaseEvent phaseEvent) { // not interested
    }
    public void afterPhase(PhaseEvent phaseEvent) { // After the RESTORE VIEW phase
        FacesContext fc = FacesContext.getCurrentInstance();
        if(((HttpServletRequest)fc.getExternalContext()
                                  .getRequest()).getRequestURI()
```

```
                                    .contains(RICO_SCRIPT_REQUEST)) {
                try {
                    readAndWriteFiles(fc, phaseEvent, new String[] {
                            PROTOTYPE_SCRIPT_FILE,
                            SCRIPTACULOUS_SCRIPT_FILE,
                            RICO_SCRIPT_FILE
                    });
                }
                catch(java.io.IOException ex) {
                    ex.printStackTrace();
                }
                phaseEvent.getFacesContext().responseComplete();
            }
        }
        private void readAndWriteFiles(FacesContext fc, PhaseEvent pe, STring[] files) {
            // Read files and write them to the response
        }
}
```

If the request URI contains the string `rico-script`, the phase listener reads three files and writes them to the response: `prototype.js`, `scriptaculous.js`, and `rico-1.1.2.js`.

Realize that we could avoid this roundabout way of reading JavaScript files by simply specifying the files themselves in the script element generated by the `AccordionRenderer`; however, that would require us to hardcode the location of that file. Because we've used a phase listener to load the JavaScript files, we can co-locate those JavaScript files with the phase listener, without having to explicitly specify the JavaScript file locations in the JSP pages.

Transmitting JSP Tag Attributes to JavaScript Code

If you implement Ajax-enabled JSF components, you will most likely need to transfer tag attributes, specified in a JSP page, to JavaScript that's stored in a file of its own, as described in the preceding section of this short cut. Let's see how that's done with the accordion component. First, the accordion tag class provides setters and getters, which are called by JSP, for accessing the tag's attribute values.

After JSP transmits tag attribute values to tag properties, JSF calls the tag's `setProperties` method, which passes those attribute values through to the component:

```
public class AccordionTag extends UIComponentELTag {
    private ValueExpression name = null;
    ...
    public void setName(ValueExpression name) { // Called by JSP
        this.name = name;
    }
    ...
    protected void setProperties(UIComponent component) { // Called by JSF
        ...
        component.setValueExpression("name", name);
        ...
    }
}
```

When the component is rendered, the accordion renderer obtains the tag values from the component and generates a small snippet of JavaScript that passes the component values through to the JavaScript; in this case, we're passing the name of the DIV that Rico will endow with accordion functionality. That DIV was originally specified as the `name` attribute of the `rico:accordion` tag:

```
public class AccordionRenderer extends Renderer {
    ...
    public void encodeEnd(FacesContext fc,
                          UIComponent component)
            throws IOException {
        ResponseWriter writer = fc.getResponseWriter();

        // Finish enclosing DIV started in encodeBegin()
        writer.write("</div>");

        // Write the JS that creates the Rico Accordion component
        Map accordionAttributes = component.getAttributes();
        String div = (String)accordionAttributes.get("name");

        writer.write("<script type='text/javascript'>");
        writer.write("new Rico.Accordion( $('" + div + "'), ");
        writeAccordionAttributes(writer, accordionAttributes);
        writer.write(");");
        writer.write("</script>");
    }
    public boolean getRendersChildren() {
        return false;
    }
    private void writeAccordionAttributes(ResponseWriter writer,
                                          Map attrs) {
        try {
            // Add the rest of the accordion's properties here.
```

```
                // See rico-1.1.2.js, line 179.
                writer.write(" { ");
                writer.write("   panelHeight: " + attrs.get("panelHeight"));
                writer.write(" } ");
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
}
```

Ajax4jsf

Now that we've discussed the particulars of both implementing and encapsulating Ajax with JSF, let's turn our attention to a framework that takes care of a great deal of those details for you: Ajax4jsf.

Ajax4jsf is a java.net project, whose home page—`https://ajax4jsf.dev.java.net/ajax/ajax-jsf`—is shown in Figure 10. Ajax4jsf provides 18 handy JSP tags that you can use to seamlessly integrate Ajax into your JSF applications. You can find a list of all the tags and their corresponding descriptions at the Ajax4jsf home page. In our brief exploration of Ajax4jsf, we will discuss two of those tags: `a4j:support`, which lets you attach Ajax functionality to a component, typically an input component, and `a4j:status`, which renders JSF components at the start and end of each Ajax4jsf Ajax call.

To illustrate both the power and the pitfalls of using Ajax4jsf, let's revisit the form completion and real-time validation examples from earlier in this short cut.