Among JSF 2's many new features, two of the most compelling are arguably composite components and Ajax support. But their strength is most apparent when the two are combined, making it easy to implement Ajax-enabled custom components with a minimum of fuss.

In this article, I'll show you how to implement an autocomplete component that uses Ajax to manage its list of completion items. In doing so, you'll see how you can integrate Ajax into your own composite components.

The code for this series is based on JSF 2 running in an enterprise container, such as GlassFish or Resin. The last section in this article is a step-by-step tutorial on installing and running the article's code with GlassFish.

A JSF autocomplete custom component

Made famous by Google's search field, autocomplete fields (also known as suggest boxes), are a staple of many Web applications. They are also a typical use case for Ajax. Autocomplete fields come with most Ajax frameworks, such as Scriptaculous and JQuery, as Figure 1 — a look at AjaxDaddy's collection of autocomplete components (see Related topics) — attests:
This article will explore one way to implement an Ajax-enabled autocomplete field with JSF. You'll see how to implement the autocomplete field shown in Figure 2, which shows a short list of fictional countries (culled from Wikipedia's "List of fictional countries" article; see Related topics):
Figure 2. The autocomplete field

Figure 3 and Figure 4 show the autocomplete field in action. In Figure 3, when Al is typed into the field, the country list is reduced to names that start with those two letters:

Figure 3. Completion items that start with Al

Similarly, Figure 4 shows the results when Bar is typed into the field. The list shows only country names that begin with Bar:
Using the autocomplete component

Composite components: The basics

If you aren't familiar with using or implementing JSF 2 composite components, you'll find an introduction in "JSF 2 fu, Part 2: Templating and composite components."

The Locations autocomplete field is a JSF composite component, and it is used in a facelet, as shown in Listing 1:

Listing 1. The facelet

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:h="http://java.sun.com/jsf/html"
     xmlns:util="http://java.sun.com/jsf/composite/util">
<h:head>
    <title>#{msgs.autoCompleteWindowTitle}</title>
</h:head>

<h:body>
    <div style="padding: 20px;">
        <h:form>
            <h:panelGrid columns="2">
                #{msgs.locationsPrompt}
                <util:autoComplete value="#{user.country}" completionItems="#{autoComplete.countries}" />
            </h:panelGrid>
        </h:form>
    </div>
</h:body>
</html>
```

The facelet in Listing 1 uses the `autoComplete` composite component by declaring an appropriate namespace — `util` — and using the component's associated tag, `<util:autoComplete>`.

Notice the two attributes for the `<util:autoComplete>` tag in Listing 1:

- `value` is the country property of a managed bean named `user`.
- `completionItems` is the initial set of completion items for the field.

The `User` class is a simple managed bean, obviously contrived for just this occasion. Its code is shown in Listing 2:
**Listing 2. The User class**

```java
package com.corejsf;
import java.io.Serializable;
import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
@Named()
@SessionScoped
public class User implements Serializable {
    private String country;
    public String getCountry() { return country; }
    public void setCountry(String country) { this.country = country; }
}
```

Notice the `@Named` annotation, which, along with `@SessionScoped`, instantiates a managed bean named `user` and places it in `session` scope the first time JSF encounters `${user.country}` in a facelet. This application's only reference to `${user.country}` takes place in Listing 1, where I specify the country property of the user managed bean as the value for the `<util:autoComplete>` component.

**Listing 3** shows the `AutoComplete` class, which defines the `countries` property that I specified as the autocomplete component's list of completion items:

**Listing 3. The completion items**

```java
package com.corejsf;
import java.io.Serializable;
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Named;
@Named
@ApplicationScoped
public class AutoComplete implements Serializable {
    public String[] getLocations() {
        return new String[] {
            "Al Hari", "Alpine Emirates", "Altruria",
            "Allied States of America", "Babakiueria", "Babalstan",
            "Babar's Kingdom", "Backhairistan", "Bacteria",
            "Bahar", "Bahavia", "Bahkan", "Bakaslavia",
            "Balamkadar", "Baki", "Balinderry", "Balochistan",
            "Baltish", "Baltonia", "Bataniland, Republic of",
            "Bayview", "Banania, Republica de", "Bandrika",
            "Bangalia", "Bangstoff", "Bapetikosweti", "Baracq",
            "Baraza", "Barataria", "Barclay Islands",
            "Barringtonia", "Bay View", "Basenji"
        };
    }
}
```

That's all there is to using the autocomplete component. Now you'll see how it works.
How the autocomplete component works

The autocomplete component is a JSF 2 composite component, so, like most composite components, it is implemented in an XHTML file. The component consists of a text input and a listbox, and some JavaScript. Initially, the listbox's style is display: none, which makes the listbox invisible.

The autocomplete component responds to three events:

- keyup events in the text input
- blur (losing focus) events in the text input
- change (selection) events in the listbox

When the user types in the text input, the autocomplete component calls a JavaScript function for every keyup event. That function coalesces keystroke events to make no more than one Ajax call every 350ms. So, in response to keyup events in the text input, the autocomplete component makes an Ajax call, at most every 350ms, to the server. (All of that is to prevent fast typists from flooding the server with Ajax calls. In practice, coalescing events may be overrated in this case, but it affords an opportunity to illustrate coalescing events in JavaScript, which in general is a useful tool.)

When the user selects an item from the listbox, the autocomplete component makes another Ajax call to the server.

Both the text input and the listbox have listeners attached to them that do most of the meaningful work on the server during Ajax calls. In response to keyup events, the text input's listener updates the listbox's completion items. In response to listbox selection events, the listbox's listener copies the listbox's selected item into the text input and hides the listbox.

Now that you have a good idea of how the autocomplete component works, you're ready to take a look at its implementation.

Implementing the autocomplete component

The autocomplete component implementation consists of these artifacts:

- A composite component
- A handful of JavaScript functions
- A value-change listener that updates completion items

I'll start with the composite component in Listing 4:

Listing 4. The autoComplete component

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:composite="http://java.sun.com/jsf/composite">

    <!-- INTERFACE -->
    <composite:interface>
```

JSF 2 fu: Ajax components
Three things are going on in Listing 4’s implementation section. First, the component makes Ajax calls in response to `keyup` events in the text input, and it hides the listbox when the text input loses focus by virtue of JavaScript functions assigned to `keyup` and `blur` events in the text input.

Second, the component makes Ajax calls in response to `change` events in the listbox with JSF 2’s `<f:ajax>` tag. When the user makes a selection from the listbox, JSF makes an Ajax call to the server and updates the text input’s value when the Ajax call returns.

Third, both the text input and the listbox have value-change listener methods attached to them, so when JSF makes Ajax calls in response to the user typing in the text input, JSF invokes the text input’s value-change listener on the server. When the user selects an item from the listbox, JSF makes an Ajax call to the server and invokes the listbox’s value-change listener.

Listing 5 shows the JavaScript used by the autocomplete component:

### Listing 5. The JavaScript

```javascript
if (!com)
    var com = {};

if (!com.corejsf) {
    var focusLostTimeout
    com.corejsf = {
        errorHandler : function(data) {
```
The JavaScript in **Listing 5** consists of three functions that I placed inside a namespace named `com.corejsf`. I implemented the namespace (which is technically a JavaScript literal object) to prevent someone from accidentally (or not) clobbering any of my three functions.

If those functions were not tucked away inside `com.corejsf`, someone could implement their own `updateCompletionItems` function, thereby replacing my implementation with theirs. It's feasible that some JavaScript library might implement a function named `updateCompletionItems`, but it's a pretty good bet nobody's going to come up with `com.corejsf.updateCompletionItems`. (In retrospect, dropping the `com`, and going with `corejsf.updateCompletionItems` probably would've sufficed, but sometimes it's easy to get carried away.)

So, what do the functions do? The `updateCompletionItems()` function makes an Ajax request to the server — by calling JSF's `jsf.ajax.request()` function — asking only that JSF render the listbox component when the Ajax call returns. The `updateCompletionItems()` function also passes two extra parameters to `jsf.ajax.request()`: the x and y coordinates of the upper left-hand corner of the listbox. The `jsf.ajax.request()` function turns those function parameters into request parameters that it sends with the Ajax call.
JSF calls the `inputLostFocus()` function when the text input loses focus. That function simply hides the listbox, using Prototype's `Element` object.

Both `updateCompletionItems()` and `inputLostFocus()` store their functionality in a function. Then they schedule their functions to execute in 350ms and 200ms, respectively. In other words, each function has a job to do, but it delays that job for either 350ms or 200ms. The text input delays after a `keyup` event, so that the `updateCompletionItems()` method sends an Ajax request once per 350ms, at most. The idea is that if the user is an (extremely!) fast typist, you don't want to flood the server with Ajax calls.

The `inputLostFocus` function, called when the text input loses focus, delays its work for 200ms. That delay is needed because the value will be copied out of the listbox when the Ajax call returns, and the listbox must be visible for that to work.

Finally, notice the `getListBoxId()` function. That helper function obtains the client identifier of the listbox from the client identifier of the text input. The function is able to do that because it's in cahoots with the `autoComplete` component in Listing 4. The `autoComplete` component assigns `input` and `listbox` as the component identifiers for the text input and listbox respectively, so the `getListBoxId()` function merely chops off `input` and appends `listbox` to get from the text input's client identifier to the listbox's.

**Listing 6** shows the implementation of the listener that pulls everything together:

**Listing 6. The listener**

```java
package com.corejsf;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import javax.enterprise.context.SessionScoped;
import javax.faces.component.UIInput;
import javax.faces.component.UISelectItems;
import javax.faces.component.UISelectOne;
import javax.faces.context.FacesContext;
import javax.faces.event.ValueChangeEvent;
import javax.inject.Named;

@Named
@SessionScoped
public class AutocompleteListener implements Serializable {
    private static String COMPLETION_ITEMS_ATTR = "corejsf.completionItems";

    public void valueChanged(ValueChangeEvent e) {
        UIInput input = (UIInput)e.getSource();
        UISelectOne listbox = (UISelectOne)input.findComponent("listbox");

        if (listbox != null) {
            UISelectItems items = (UISelectItems)listbox.getChildren().get(0);
            Map<String, Object> attrs = listbox.getAttributes();
            List<String> newItems = getNewItems((String)input.getValue(),
                getCompletionItems(listbox, items, attrs));

            items.setValue(newItems.toArray());
            setListBoxStyle(newItems.size(), attrs);
        }
    }
}
```
JSF invokes the listener's `valueChanged()` method during Ajax calls in response to `keyup` events in the text input. That method creates a new set of completion items and then sets the listbox's items to this new set. The method also sets style attributes for the listbox that determine whether the listbox is displayed when the Ajax call returns.

The `setListBoxStyle()` method in Listing 6 uses the `x` and `y` request parameter values that I specified when I made an Ajax call in Listing 5.
JSF invokes the listener's only other public method, `completionItemSelected()`, during Ajax calls in response to selection events in the listbox. That method copies the listbox's value into the text input and hides the listbox.

Notice that the `valueChanged()` method also stores the original completion items in an attribute of the listbox. Because each `autoComplete` component maintains its own list of completion items, multiple `autoComplete` components can peacefully coexist in the same page without stomping on one another's completion items.

### Running the examples with GlassFish and Eclipse

The code in this series of articles is best suited to a JEE 6 container, such as GlassFish or Resin. You can get things to work with a servlet container, such as Tomcat, but note the "get things to work" part. Since my goal is to focus on your getting the full potential out of JSF 2 and JEE 6, and not on configuration issues, I will stick to GlassFish v3.

For the rest of this article, I'll show you how to run this article's sample code using GlassFish v3 and Eclipse. The instructions here will also suffice for the code from the rest of this series of articles. (I'm using Eclipse 3.4.1, so the closer you can match that when running the examples, the better.)

Figure 5 shows the directory structure that you'll find in the code for this article. (See Download to get the code now.) There's an `autoComplete` directory containing the application and an empty workspace directory for Eclipse.

**Figure 5. Source code in this article's download**

Now that you have the code, you're almost ready to get it running. First, you need the GlassFish Eclipse plug-in.

Follow the installation instructions for the plug-in, and you're ready to go.

To install the code for this article, create a Dynamic Web project in Eclipse. You can do that from the File > New menu: if you don't see Dynamic Web project there, select Other, and in the ensuing dialog open the Web folder and select Dynamic Web Project, as shown in Figure 7:
The next step is to configure the project. Make the following selections on the first screen of the New Dynamic Web Project wizard, as shown in Figure 8:

1. Under **Project contents**, leave the **Use default** box unchecked. In the **Directory** field, enter (or browse to) the sample code's autoComplete directory.
2. For **Target Runtime**, select GlassFish v3 Java EE 6.
3. For **Dynamic Web Module version**, enter 2.5.
4. For **Configuration**, select Default Configuration for GlassFish v3 Java EE 6.
5. Under **EAR Membership**, leave the **Add project to an EAR** box unchecked, and enter `autoCompleteEAR` in the **EAR Project Name:** field.

**Figure 7. Creating a Dynamic Web project**
Figure 8. Configuring the application, step 1

Click **Next**, then enter the values shown in Figure 9:

1. For **Context Root**: enter `autoComplete`.
2. For **Content Directory**: enter `web`.
3. For **Java Source Directory**: enter `src/java`. Leave the Generate deployment descriptor box unchecked.
Now you should have an autoComplete project, visible in Eclipse’s Project Explorer view, as shown in Figure 10:

Figure 10. The autoComplete project

Now select the project, right click on it, and select Run on Server, as shown in Figure 11:
Figure 11. Run on server in Eclipse

Select GlassFish v3 Java EE 6 from the list of servers in the Run On Server dialog, shown in Figure 12:
Click **Finish**. Eclipse should start GlassFish and, subsequently, the autoComplete application, as shown in **Figure 13**: 

**Figure 12. Selecting GlassFish**
Conclusion

JSF 2 makes it easy to create powerful Ajax-enabled custom components. You don’t have to implement a Java-based component or renderer, or declare that component or renderer in XML, or integrate third-party JavaScript to make Ajax calls. With JSF 2, all you need to do is create a composite component, with markup almost identical to any JSF 2 facelet view, and perhaps add a little JavaScript or Java code, and voilà — you have a cool custom component that will make data input a breeze for your application's users.

In the next installment of JSF fu, I'll discuss more aspects of implementing Ajaxified JSF custom components, such as integrating the `<f:ajax>` tag so your custom components can participate in Ajax initiated by others.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample code for this article</td>
<td>j-jsf2fu-0410-src.zip</td>
<td>39KB</td>
</tr>
</tbody>
</table>
Related topic

- **List of fictional countries**: This article's example uses Wikipedia's list of fictional countries.

© Copyright IBM Corporation 2010
Trademarks
(www.ibm.com/developerworks.ibm/trademarks/)