Accordion

Adding massive amounts of data to one web page is not a recommended design approach because it can be completely disorienting to the user, and might cause him to go to another site. There are always exceptions, though, which is the case when using an accordion component to display data. Using an accordion component enables a single web page to display much more content without disorienting the user in the process. An accordion has multiple panels that can expand and collapse to reveal only the data that a user is interested in viewing without overwhelming him with everything at one time.

In this chapter, we will learn how to create a custom Ajax-enabled accordion component. An Ajax-enabled accordion can lend itself to many unique situations. For example, you can connect the component to live XML data from a database via a server-side language, which can send and receive XML or any other format that you prefer. The accordion component can be the graphical user interface for a custom web mail application that displays threads in different panels. The server can push new data to the component when mail has been updated, deleted, or added to the database, and the accordion can parse it to update, delete, or add new panels to the thread. This is a perfect example of providing access to massive amounts of content without scaring away the users of the application. It is also a great way to organize the content so that the application is ultimately more usable and purposeful.

Getting Started

In order to get started we must do a few things first. We must define an XML structure for the object to accept, which will be scalable and grow with a large application. Once we have defined this data structure we must then create a process for requesting it. This section will focus on both of these assignments in order to get us started toward creating the object.

The XML Architecture

Before we begin, we need to architect an XML structure that will be used to represent an accordion with all its properties. Aside from the XML declaration, which needs to be added to the top of the file, the first element that we will create will be named `accordion` to represent the actual object or component. If we were to visualize an accordion, we would know that it consists of multiple panels, so we will use `panel` as the first child node name. To identify which panel is expanded by default when the accordion is rendered, we will add an `expanded` attribute to the `panel` element and populate it with a Boolean of `true` for expanded. Each panel should also include a `title` and have `content` that displays when the panel is expanded; therefore, we will create these elements as child nodes of the panel. If multiple panels are necessary to present content, we can easily duplicate the panel and its enclosed children elements so that there are numerous panels, one after the other. There is no limit to the amount of panels that can be added, but the accordion component will render slower as more data is added. Ultimately, however, a difference is not noticeable until your XML file gets very large. Take a look at the sample code in Listing 10.1 to get an idea of how to construct an accordion XML file that will be parsed by our custom component.

Listing 10.1 **The XML Sample for the Accordion** (`accordion.xml`)

```
<?xml version="1.0" encoding="iso-8859-1"?>
<accordion>
    <panel expanded="true">
        <title></title>
        <content></content>
    </panel>
    <panel>
        <title></title>
        <content></content>
    </panel>
</accordion>
```

After the structure has been created, we can add data between the XML node elements. This data will be used to display in the corresponding parts of the accordion component. Accepting HTML in any node element will make this component much more flexible and can be very easily achieved by simply adding CDATA tags between the `content` elements. Here is an example of how easy this is to accomplish:

```
<content><![CDATA[<b>html text goes here</b>]]></content>
```

Adding CDATA tags allows us to use any HTML that we would like to display in any given panel. We could display everything from complex tables, images, and even other components. After you have completed creating all of the components in this book, you can combine them to make additional ways of interacting with data. After we have populated the XML file, we are ready to request it and use its content to render the component.

Requesting the XML

It is now time to set up the request for the XML. We will request the XML that we created in the last section and push it to the parsing method in the component. To make the request, we will first create an HTML file to hold all the code that will create and facilitate communication between the component and Ajax. Keep in mind that aside from building this sample, you will probably not use this component solely as you might have an existing file that you want to incorporate the component into. With the correct files and a few tweaks to the placement of the component, you can easily add one to any page. In the header of the new sample HTML file, add references to the accordion CSS and all the necessary JavaScript files, as in Listing 10.2. Keep in mind that you will have to run the files on a server in order for the XHR to work.

Listing 10.2 **The HTML Container for the Project (`accordion.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">
<head>
<title>Accordion</title>
<link href="css/accordion.css" rel="stylesheet" type="text/css" />
<script type="text/javascript" src="../javascript/Utilities.js"></script>
<script type="text/javascript" src="../javascript/utils/AjaxUpdater.js"></script>
<script type="text/javascript" src="../javascript/utils/HTTP.js"></script>
<script type="text/javascript" src="../javascript/utils/Ajax.js"></script>
<script type="text/javascript" src="../javascript/components/Panel.js"></script>
<script type="text/javascript"
    ➥src="../javascript/components/Accordion.js"></script>
```

We are including a number of JavaScript files—one of which is the `Utilities` object that we created in Chapter 14, "Singleton Pattern"—because it will be used to create the accordion's HTML elements that get rendered on the screen. The other JavaScript files, `Panel` and `Accordion`, are the objects that we will be focusing on creating throughout the rest of this chapter. In order to get started, you can create these files in the corresponding JavaScript directory.

After we have the files included, we need to create an `initialize` method (see Listing 10.3) in the header and add an `Update` call with the `AjaxUpdater` to request the accordion XML file. This object will make the request to the Ajax object based on the HTTP method and the query parameters that you pass. The Ajax object will then make an XHR to the XML file that we are passing and will finally respond to the callback method that you specify. In this case, it is the `display` method for the accordion, which will parse the XML and render the accordion and its panels. The first parameter is the HTTP method for the request. The second is the requested file, plus any query string that you need to append for posting data, which we will be doing more of in Part V, "Server-Side Interaction," when we begin to interact with server-side languages and databases. The last parameter is the method that you would like to be used as a callback method for the request.

Listing 10.3 **The XHR Request Code** (`accordion.html`)

```
<script type="text/javascript">
function initialize()
{
    AjaxUpdater.Update("GET", "services/accordion.xml", Accordion.display);
}
</script>
</head>
```

As you can see in Listing 10.3, we need to make sure that all the code is available or fully instantiated. We must simply wait until the page loads before we call the `initialize` method that makes the request. The following shows an example of the `body onload` method:

```
<body onload="javascript:initialize();">
```

I have also added a `loading div` element (see Listing 10.4) to handle the ready state status of the request. This is a good way to present the user with a message regarding the state.

Listing 10.4 **A `div` Element to Display Loading Status** (`accordion.html`)

```
<div id="loading"></div>
</body>
</html>
```

When we have the HTML file ready to go, we can start creating the objects that make up the accordion component. Let's start with the `Accordion` object itself.

Creating the Accordion Object

The first object that needs to be created for the accordion component is the `Accordion` object. The `Accordion` object will handle parsing the XML as well as creating and controlling a variable number of panel objects. Accordions consist of multiple panels that stack on top of each other and expand and collapse to reveal hidden content. We will create a `panel` object that uses the prototype structure we discussed in Chapter 5, "Object-Oriented JavaScript." This will allow us to create multiple `panel` objects. Before we move onto the details of creating the panels, however, we will finish creating the `Accordion` object. Creating the `Accordion` object and initializing the properties is trivial, but there is an important sequence of events that needs to happen—otherwise, the object will not initialize properly. First, we must instantiate the object so that we can use its other methods. In order to trigger the `initialize` method, we must declare the method before we call it. The code snippet in Listing 10.5 shows an example of how we can accomplish this.

Listing 10.5 **Accordion Instantiation and Initialization (`Accordion.js`)**

```
Accordion = {};

Accordion.initialize = function()
{
    panels = new Array();
    expandedPanel = 0;
}
Accordion.initialize();
```

Calling the `initialize` method before declaring it will cause an error because the method does not exist in memory at this point and is not accessible. Notice that we have two properties in the `initialize` method: A new `panels` array is created and the `expandedPanel` number is set to `0`. The `panels` array is simply an array of `panel` objects that the `Accordion` object will contain after the panels have been created. These panels will be added to the array in the `display` method and then be accessible to the other methods in the `Accordion` object. The `expandedPanel` number is used to determine which panel is expanded by default when the accordion is rendered. This is the property that will be set when we get the results of the XML file's `expanded` attribute.

To render the accordion, we will create a `display` method. This is the method we are using as the callback function for the Ajax request in the HTML file we created at the beginning of the chapter. The first thing we need to do in the `display` method is to check the ready state of the Ajax object. If the ready state returns `"OK"`, we will continue with the method; if we do not receive `"OK"` as the value, we can add a number of branches to handle the different scenarios. For the example, I simply created a `try-catch` to display a generic message for failed requests.

When we receive a successful message, we need to create an accordion `div` element to act as the parent container for all the panels. When we have our `accordion div` element created, we need to iterate through the panels from the response by targeting the `panel` node element by name in the response XML. After we have an array of panel data from the response, we can use the `length` property of the `panel` array to iterate through the array. While iterating through the `panel` array, we need to get the `title` and `content` data for each panel. We will find the `panel` element with an `expanded` attribute that is set to `true` and use its iteration number to set the `expandedPanel` variable. The `expandedPanel` number will be useful for matching purposes because it will represent the unique ID of each panel object. When we have all the data from the XML targeted to local variables, we can push a new `panel` object into the `panels` array we instantiated in the accordion's `initialize` method. When creating the new `panel` we will pass it the iteration number as a unique ID, along with the title and content strings.

Now that we have the `panel` objects created, we can append the panel HTML elements to the accordion. We will accomplish this by using the `appendChild` method in the `Utilities` object and passing the `accordion div` element and each panel `display` method. The panel `display` method will pass all the HTML elements that are created inside the `panel` object and append them to the accordion. When we have completed iterating through the panels array and have appended them to the accordion, we will be able to append the accordion to the document body. Appending the accordion to the document body will render the accordion in the web page. Take a look at Listing 10.6 to see the `display` method in its entirety.

Listing 10.6 **The Accordion's `display` Method (`Accordion.js`)**

```
Accordion.display = function()
{
    try
    {
        if(Ajax.checkReadyState('loading') == "OK")
        {
            var accordion = Utilities.createElement("div", {id:'accordion'});
            var p = Ajax.getResponse().getElementsByTagName('panel');
            for(var i=0; i<p.length; i++)
            {
                var title = Ajax.getResponse().getElementsByTagName('title')[i].
                    ➥firstChild.data;
                var content =
                    ➥Ajax.getResponse().getElementsByTagName('content')[i].
                    ➥firstChild.data;
                if(p[i].getAttribute('expanded')) { expandedPanel = i; }
                panels.push( new Panel(i, title, content) );

                Utilities.appendChild(accordion, panels[i].display());
            }

            Utilities.appendChild(document.body, accordion);
            Accordion.toggle(expandedPanel);
        }
    }
    catch(err)
    {
        document.write(err);
    }
}
```

As you can see, there is a `toggle` method that I did not mention, which is called at the end of the `display` method. This is the reason we created the `panel` array and the `expandPanel` number variables. When the `toggle` method is called, it iterates through the `panel` array and checks to see whether there is a panel `ID` that matches the ID parameter. When it finds a match, it expands that panel by `ID`; when it does not match, it collapses that panel. The expand/collapse panel methods are in the `panel` object, which we will create in the next section. Listing 10.7 shows the entire code for the accordion's `toggle` method.

Listing 10.7 **The Accordion's `toggle` Method (`Accordion.js`)**

```
Accordion.toggle = function(id)
{
    for(var i=0; i<panels.length; i++)
    {
        if(panels[i].id == id)
        {
            panels[i].expand();
        }
        else
        {
            panels[i].collapse();
        }
    }
}
```

As mentioned, the `toggle` method takes an element ID as a parameter and iterates through the `panels` array. When it discovers a matching ID, it expands that particular panel; otherwise, it collapses it.

Now that we have the `Accordion` object created, we can now focus on creating the panels. Another way to handle the accordion panels' `toggle` method is to allow multiple panels to be open at one time. To do this, you need to create a method that does not collapse other panels that are open. You also need to check whether the current panel that is being clicked is already expanded (if so, it should be closed). This keeps the expand/collapse nature of the panel intact.

Panel Functionality and Data Display

The `panel` object uses the prototype structure to keep it reusable, which essentially allows us to create multiple panel objects. An accordion panel needs a unique ID for reference purposes and can include a title, which is displayed in a panel header, and content that is exposed when a user expands the panel. The `id`, `title`, and `content` values will become properties of the `panel` object and will be visually represented in the accordion. These properties will be accessible during runtime and contained within the panels that created them. In order to create these properties, we will use the values we passed to the new `panel` objects while iterating through them in the accordion `display` method. Listing 10.8 shows how these values were passed to the `panel` object's constructor function and are scoped to the panel.

Listing 10.8 **The `Panel` Object Properties and Constructor** (`Panel.js`)

```
function Panel(id, title, content)
{
    this.id = id;
    this.title = title;
    this.content = content;
}
```

The constructor function is used to instantiate the panel and set the property values of the object. After the panel is instantiated, it can be used to call other methods within itself. In the `Accordion` object, the first method we called was the `display` method. This method creates the `div` elements that are used to display the data that is passed to the object. To create the elements that are necessary to render a panel, we will need to use some of the utility methods that we created in the `Utilities` object in Chapter 10. In order to create the display, we will need to create the following elements: `panel`, `header`, `title`, and `content`. The `panel` element is simply a container for the other elements, whereas the `header` element contains the `title` element and has an `onclick` event that will expand and collapse individual panels. The final two elements are `title` and `content`. They both have an `innerHTML` property that is set to the relative properties that were set in the constructor. After we have created all the necessary elements, we need to append them to the `panel` element. When we complete the `display` method, we return the `panel` element and append it to the document body in the `Accordion` object. Listing 10.9 shows the entire code for creating the elements, appending them, and returning the panel.

Listing 10.9 **The Panel's Display Method** (`Panel.js`)

```
Panel.prototype.display = function()
{
    var panel = Utilities.createElement("div");

    var header = Utilities.createElement("div", {
        className: 'header',
        onclick: this.toggle(this.id)
        });

    var title = Utilities.createElement("div", {
        className: 'title',
        innerHTML: this.title
        });

    var content = Utilities.createElement("div", {
        id: 'content_'+ this.id,
        className: 'content',
```

Listing 10.9 **Continued**

```
        innerHTML: this.content
        });

    Utilities.appendChild(panel, Utilities.appendChild(header, title), content);
    return panel;
}
```

As you probably remember when we created the `Accordion` object, the panels display method is called from the accordion's display method as a parameter of the `appendChild` call, along with the parent accordion element. This is how the panels are added to the accordion and then the accordion was added to the document body.

The `toggle` method in Listing 10.10 is used in the `header div` as an `onclick` event. This method is interesting because it returns another method. The method that is returned is triggered during an `onclick` event and ultimately calls the accordion's `toggle` method. The header is also passing the panel ID when the code is executed to be used as a parameter in the accordion's `toggle` method. This is the ID that is used to decipher which panel should be expanded and which panels should be collapsed.

Listing 10.10 **Toggling the Panel State** (`Panel.js`)

```
Panel.prototype.toggle = function(id)
{
    return function()
    {
        Accordion.toggle(id);
    }
}
```

The `collapse` and `expand` methods in Listing 10.11 simply hide and reveal the `content div`s in the panels. They both use the `Utilities getElement` method, which gets the `content` element by name in the document. The `collapse` method sets the `display` style to `none` to hide it, whereas the `expand` method sets the `display` style to an empty string to reveal the `content`'s data.

Listing 10.11 **The Panel's Collapse and Expand Methods** (`Panel.js`)

```
Panel.prototype.collapse = function()
{
    Utilities.getElement('content_'+ this.id).style.display = 'none';
}

Panel.prototype.expand = function()
{
    Utilities.getElement('content_'+ this.id).style.display = '';
}
```

Creating the CSS

The CSS file for the accordion contains the styles for each of the elements within the accordion. As Listing 10.12 shows, the `accordion` element sets the `font-family`, `font-size`, `width`, and `border` attributes. These styles are inherited by each of the panels because they are encapsulated within the `accordion` element.

Listing 10.12 **The Accordion's Styles** (`accordion.css`)

```
#accordion
{
    font-family: Arial, Helvetica, sans-serif;
    font-size: 12px;
    width: 600px;
    border: #ccc 1px solid;
}
```

The `header` holds the title for each panel, and has a style shown in Listing 10.13 that contains a `border-bottom` attribute, which matches the border that surrounds the panels to make it look more incorporated into the accordion. It also has a `background-color`, a `width` to set the size of the clickable area, and a `pointer cursor` to show users that they can click the headers to toggle their state.

Listing 10.13 **The Accordion Header Style** (`accordion.css`)

```
.header
{
    border-bottom: #ccc 1px solid;
    background-color: #eaeaea;
    width: 600px;
    cursor: pointer;
}
```

The `title` class in Listing 10.14 simply bolds the font with the `font-weight`, changes the `color` of the font, and adds a little `padding` to keep the title away from the edges of the header that contains it.

Listing 10.14 **The Accordion Title Style** (`accordion.css`)

```
.title
{
    font-weight: bold;
    color: #333;
    padding: 5px;
}
```

The class for the panel `content` simply sets the `padding` to keep the content away from the edges of the panels, as seen in Lisitng 10.15.

Listing 10.15 **The Accordion Content Style** (`accordion.css`)

```
.content
{
    padding: 10px;
}
```

All these styles are easily editable and can be modified to completely change the look of the accordion. It is now up to you to make it your own, or brand it for any project you would like to incorporate it with.

The completed accordion component will look very similar to Figure 10.1, aside from any content differences or additional panels you may add to the XML. This chapter's sample includes an example of how you can display an email thread in the accordion, which we will incorporate into an internal web mail application for the sample that we create in Part V, when we learn how to combine a database with Ajax.

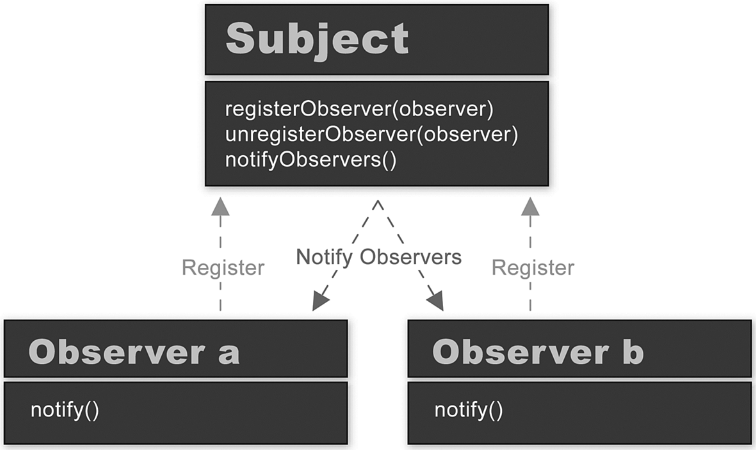

Figure 10.1 The completed accordion component is just one example of the many purposes they can serve in a web application.