11

Atlas Controls

This chapter introduces the concept of the Microsoft Atlas controls. These controls function like other ASP.NET server controls. Once rendered, these controls provide the HTML necessary to communicate back to the server from the client. In this chapter, you look at:

- Referencing and creating client-side controls with Atlas
- Client-side events — events that are raised by user controls and processed in client-side script
- Extending existing ASP.NET controls
- Data binding, which allows for the connection of components and controls to manage the flow of data between the two

Controls

With the Atlas framework, the page developer has the ability to create rich applications that do not need to post back to the server on every action that a user performs. Much of this functionality is provided by a set of controls referred to as the Atlas server controls. These controls output the appropriate markup for the web browser client so that actions on the client do not require the dreaded postback.

You start by looking at some generic controls and then move into several more complicated controls.

Buttons

An HTML button is one of the most basic controls in a web application. It is typically used as the last step in some type of user interaction. For example, a user will fill out a form with contact information. The last step in the process is for the user to submit that information to the web server. Atlas has support for working with the buttons in the `Sys.UI.Button()` class.

Try It Out Creating a Button and Changing Its Properties

Take a look at an example of dynamically creating a button and then changing one of the properties of the button. Creating a button dynamically can be done through the Document Object Model (DOM):

```
var associatedElement = document.getElementById("divTest");
var btnCreate = document.createElement("button");
btnCreate.id = "btnCreated";
btnCreate.value = "Hi";
btnCreate.innerHTML = "Hi";
associatedElement.appendChild(btnCreate);
```

For more information regarding the DOM, turn to Chapter 3.

Atlas allows a control to be manipulated through JavaScript. The preceding code will hold a reference to a button. After the reference to the button is created, the `visible` and `accessKey` properties of the object are set. In the first code snippet, a button is created using the DOM. In the second snippet, a reference is created to the button. The `visible` property of the button is set to `true`, and the access property is set to `"o"`.

Now that you have a button in the browser, you may need to reference the button in Atlas. To reference a button in Atlas, use the following code.

```
var btnAlreadyThere = new Sys.UI.Button($("btnCreated"));
btnAlreadyThere.set_visible(true);
btnAlreadyThere.set_accessKey("o");
```

While looking at the preceding code, take notice of several things:

- The dollar sign ($) is a shortcut to `document.getElementById`. Using the DOM methods means that the method will provide support across multiple browsers that support the DOM.
- There is a class titled `Sys.UI.Button`. This class encapsulates functionality of an HTML button and allows that button to be modified through Atlas.
- The `Sys.UI.Button` class provides a reference to a button. The class does not provide support for creating a new button, merely for referencing the button.
- Setting the properties is easy. The preceding code shows the `visible` and `accessKey` properties being set. These are set by calling `set_visible(boolean)` and `set_accessKey(string)`.

Approximately 40 properties can be set on a button or other user control in Atlas. The most common ones will be the display/user interface (UI) oriented properties, such as `behaviors`, `cssClass`, `enabled`, and other UI properties. It is beyond the scope of this book to go through all the properties of the button control or all the properties of every single control. Suffice it to say, each of the additional UI wrapper controls provided by Microsoft Atlas provides a similar set of properties. You can obtain them by the using the `for()` loop presented at the end of Chapter 10 to interrogate the JavaScript objects.

Sys.UI.Data Controls

The `Sys.UI.Data` namespace provides a couple of key visual/UI-related controls — namely the `listView` and `itemView` classes. We are going to take a look at the `listView` control here.

Try It Out Using listView

A `listView` is an Atlas control that is used to display a tabular set of data. It is very similar to a `GridView/DataGrid` in ASP.NET or an HTML table in classical ASP used to present data. A `listView` is set up within the `<page>` tag of the `xml-script` section of the page. Take a look at the following code using the `listView` taken from the data binding section later in the chapter.

```
<page xmlns:script="http://schemas.microsoft.com/xml-script/2005">
<components>
    <listView id="ProjectResults"
        itemTemplateParentElementId="ProjectTemplate" >
        <layoutTemplate>
            <template layoutElement="ProjectTemplate" />
        </layoutTemplate>
        <itemTemplate>
            <template layoutElement="ProjectItemTemplate">
                <label id="ProjectNameLabel">
                    <bindings>
                        <binding dataPath="ProjectName" property="text" />
                    </bindings>
                </label>
            </template>
        </itemTemplate>
    </listView>
</components>
</page>
```

Notice some of the settings in this `listView`:

- The `listView` has an ID of `"ProjectResults"`.
- The target element is `"ProjectTemplate"`. The target element from this example is a `div` tag that will receive the tabular results.
- The bindings define how data is bound to the element. More information on data binding is found later in this chapter. In this example, the data column `ProjectName` is bound to the label `ProjectNameLabel`.
- The `layoutTemplate` defines the layout that will be used for the records. In this situation, the `ProjectItemTemplate` is the element that will hold the contents of data that is bound to the `listView`.
- The `itemTemplate` defines the layout that will be used for a single record. Within the `itemTemplate`, there is a binding setup that associates the `ProjectItemTemplate` tag with the layout of the individual records as defined by the `ProjectNameLabel` element.

Server Controls

One of the interesting pieces of Atlas is the support for various server-centric controls and integration with the server-centric ASP.NET way of thinking. The idea behind these controls is to keep the server control model and add support for client-side/Ajax scenarios.

Partial Updates and the UpdatePanel

One of the simplest scenarios is the ability to perform updates incrementally. The goal is to minimize the use of postbacks and whole refreshes and to instead use targeted and partial updates. This scenario is enabled by turning on partial updates through the `ScriptManager` control.

```
<atlas:ScriptManager runat="server" id="scriptManager"
EnablePartialRendering="true" />
```

Setting the `EnablePartialRending` attribute to `true` causes a postback to be simulated using the Atlas `Sys.Net.WebRequest` class (which is based on the `XMLHttpRequest` object). On the server, the page is processed as if a "classical" page postback has occurred. This works with the server controls, both in-the-box and third-party controls, which call `doPostBack`. The result is that `Page.IsPostBack` returns a `true`, if that is necessary. In addition, server-side events fire as they are designed to, and event handlers continue to be processed.

Try It Out Using an UpdatePanel

The next step in this process is to determine what to do when the server returns the data. The `Script Manager` needs to determine what parts of the page have changed. The `UpdatePanel` helps the `ScriptManager` in this situation. `UpdatePanels` are used to define portions/regions of a page that can be updated together. The `ScriptManager` will override the rendering of an entire HTML page and display only the content of the `UpdatePanels`. In addition, the `ScriptManager` will handle the updating of page titles, viewstate (and other hidden fields), updated styles, and the like.

Consider this small example. In this example, you will have a drop-down list of employees. When an employee is selected, a `GridView` will be displayed and filled based on the employee that is selected. Here is some ASPX code:

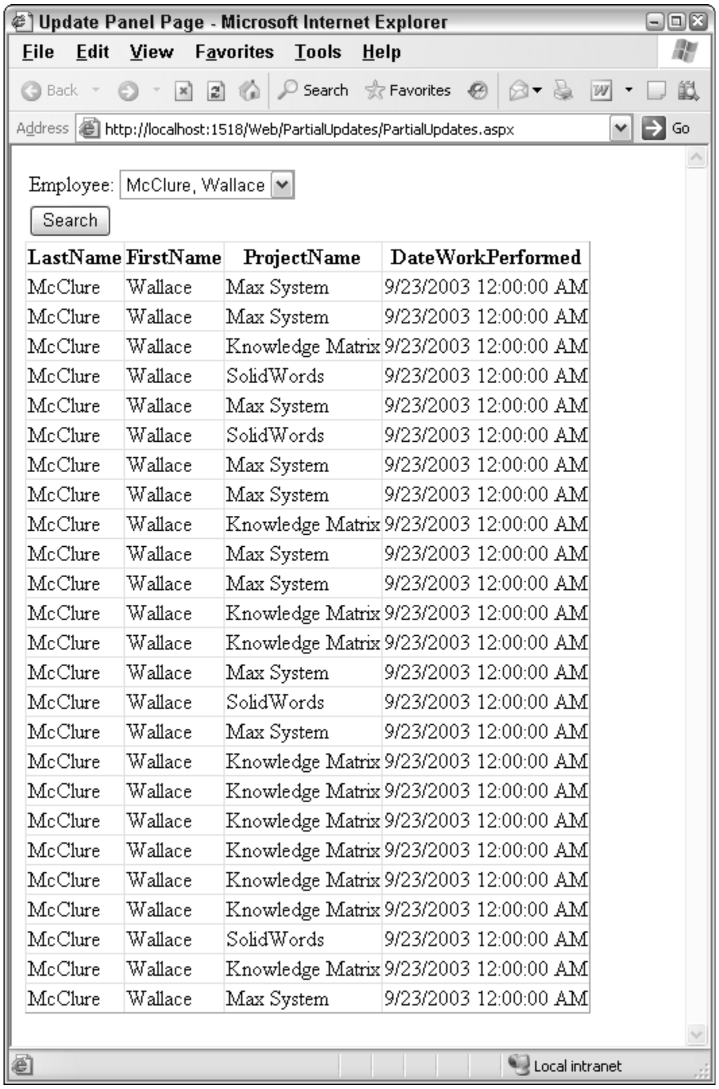
```
<atlas:ScriptManager ID="ScriptManager1" runat="server"
EnablePartialRendering="true" />
        <div>
        </div>
    <div>
<table>
    <tr>
        <td><asp:Label runat="server" ID="lblEmployee">Employee:</asp:Label></td>
        <td><asp:DropDownList runat="server"
ID="ddlEmployee"></asp:DropDownList></td>
    </tr>
    <tr>
        <td colspan="2">
            <asp:Button runat="server" ID="btnSearch" Text="Search"
OnClick="btnSearch_Click" />
        </td>
    </tr>
</table>
    </div>
    <atlas:UpdatePanel runat="server" ID="upSearch">
        <ContentTemplate>
            <asp:GridView ID="gvSearchResults" runat="server"
EnableSortingAndPagingCallbacks="true">
            </asp:GridView>
        </ContentTemplate>
        <Triggers>
            <atlas:ControlEventTrigger ControlID="btnSearch" EventName="Click" />
        </Triggers>
    </atlas:UpdatePanel>
```

How It Works

Conceptually, there are several things you should take notice of in the preceding code:

- ❑ The `UpdatePanel` allows the developer to define a region that will be updated.
- ❑ The `<atlas:ControlEventTrigger>` allows for the `ControlID` to be specified through the named property. This allows for Atlas to specifically act on a control.
- ❑ There is an `EventName` property. This property "listens" for the specific event as specified by the `EventName` property. This property works with the specified `ControlID` property.

Based on the preceding code, you are able to get the screen of results shown in Figure 11-1.


Update Panel Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:1518/Web/PartialUpdates/PartialUpdates.aspx
Go
Local intranet

Employee: McClure, Wallace

Search

LastName	FirstName	ProjectName	DateWorkPerformed
McClure	Wallace	Max System	9/23/2003 12:00:00 AM
McClure	Wallace	Max System	9/23/2003 12:00:00 AM
McClure	Wallace	Knowledge Matrix	9/23/2003 12:00:00 AM
McClure	Wallace	SolidWords	9/23/2003 12:00:00 AM
McClure	Wallace	Max System	9/23/2003 12:00:00 AM
McClure	Wallace	SolidWords	9/23/2003 12:00:00 AM
McClure	Wallace	Max System	9/23/2003 12:00:00 AM
McClure	Wallace	Max System	9/23/2003 12:00:00 AM
McClure	Wallace	Knowledge Matrix	9/23/2003 12:00:00 AM
McClure	Wallace	Max System	9/23/2003 12:00:00 AM
McClure	Wallace	Max System	9/23/2003 12:00:00 AM
McClure	Wallace	Knowledge Matrix	9/23/2003 12:00:00 AM
McClure	Wallace	Knowledge Matrix	9/23/2003 12:00:00 AM
McClure	Wallace	Max System	9/23/2003 12:00:00 AM
McClure	Wallace	SolidWords	9/23/2003 12:00:00 AM
McClure	Wallace	Max System	9/23/2003 12:00:00 AM
McClure	Wallace	Knowledge Matrix	9/23/2003 12:00:00 AM
McClure	Wallace	Knowledge Matrix	9/23/2003 12:00:00 AM
McClure	Wallace	Knowledge Matrix	9/23/2003 12:00:00 AM
McClure	Wallace	Knowledge Matrix	9/23/2003 12:00:00 AM
McClure	Wallace	Knowledge Matrix	9/23/2003 12:00:00 AM
McClure	Wallace	Knowledge Matrix	9/23/2003 12:00:00 AM
McClure	Wallace	SolidWords	9/23/2003 12:00:00 AM
McClure	Wallace	Knowledge Matrix	9/23/2003 12:00:00 AM
McClure	Wallace	Max System	9/23/2003 12:00:00 AM

Figure 11-1

Just to show that there is not an update of the complete page, take a look at the source code of the preceding page after the method returns its values.

```
<!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>
Update Panel Page
</title><style type="text/css">
atlas__delta { font-family:Lucida Console; }
</style></head>
<body>
<form name="form1" method="post" action="PartialUpdates.aspx" id="form1">
<div>
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="....." />
</div>
<script src="../ScriptLibrary/Atlas/Debug/Atlas.js"
type="text/javascript"></script>
<div>
</div>
<div>
<table>
<tr>
<td><span id="lblEmployee">Employee:</span></td>
<td><select name="ddlEmployee" id="ddlEmployee">
<option value=""></option>
<option value="ee8d807d-fab3-4e39-948a-67362c61a470">McClure, Wallace</option>
<option value="4d39f850-2b65-477d-9cf4-c90158e26b5f">Coder, Lou</option>
<option value="3a3221a8-5043-46cb-bd61-d12e77195f61">Smith, Joe</option>
</select></td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="btnSearch" value="Search" id="btnSearch" />
</td>
</tr>
</table>
</div>
<span id="upSearch_Start"></span>
<div>
</div>
        <span id="upSearch_End"></span>
<div>
<input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
value="/wEWBgK9+Yf9BgLO+4f+BgKWzeL+BQL4qeKmDgKdwp/VDgKln/PuCnnVW0Hi59nE7dq0vREhD29F
VgTo" />
</div>
<script type="text/xml-script"><page
xmlns:script="http://schemas.microsoft.com/xml-script/2005"><components />
</page></script><script
type="text/javascript">Sys.WebForms._PageRequest._setupAsyncPostBacks(document.getE
lementById('form1'), 'ScriptManager1');
</script></form>
</body>
</html>
```

The preceding code contains several things it's important to understand:

- This is the same code that would be generated when the page is first loaded. The `ViewState` is at its default setting within the "view source." While the `ViewState` has been "clipped" for brevity in this example, the `ViewState` does not seem to hold all the data-bound content of the `GridView` that you would expect to see from a page that contained the data bound to the `GridView` and resent to the client. The `UpdatePanel` has handled the updating of `ViewState`.
- The `GridView` is not displayed on the page. The `UpdatePanel` is replaced by the `<span>` tag. This will be the location where the `GridView` is placed when the data is returned to the web client. When the data is returned, the data is placed within the `<span>` tag.
- This methodology allows for the integration of Ajax functionality into ASP.NET applications, while preserving the server-centric development methodology that ASP.NET has traditionally provided.

Timed Refreshes

Atlas contains a timer control that provides for timed actions to occur. Situations where this would be useful might be:

- Stock symbol updates for an investor
- Inventory updates within a manufacturing environment
- Retail inventory
- Executive information system or dashboard

Try It Out Performing a Timed Update with the Timer Control

Take a look at some code to perform a timed update:

```
<atlas:TimerControl runat="server" ID="tc" Interval="5000" OnTick="tc_Tick"/>
<atlas:UpdatePanel ID="up" runat="server" Mode=Conditional>
<ContentTemplate>
<asp:Label ID="lblInv" runat="server" AssociatedControlID="lblNum" />:
<asp:Label ID="lblNum" runat="server" />
</ContentTemplate>
<Triggers>
    <atlas:ControlEventTrigger ControlID="tc" EventName="Tick" />
</Triggers>
</atlas:UpdatePanel>
```

The ASP.NET event control is:

```
protected void tc_Tick(object sender, EventArgs e)
{
    lblInv.Text = "Widget";
    lblNum.Text = DateTime.Now.Millisecond.ToString();
    if ( /* Something happened */ true)
    {
        up.Update();
    }
}
```

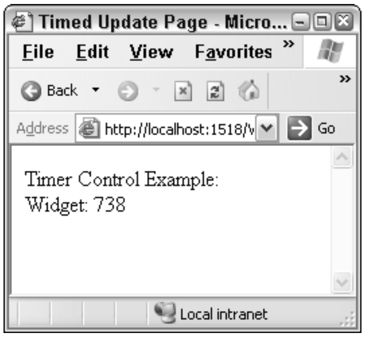
How It Works

In this example, an Atlas server control called the `<atlas:TimerControl>` is set up. There are two properties that are of interest. These are:

- `Interval` — The interval is the number of milliseconds before the `OnTick` event is called. In this example, the interface is set to 5,000 milliseconds. As a result, every 5,000 milliseconds, the timer control will update and run.
- `OnTick` — This event fires when the period of time defined by the `Interval` has elapsed.

Along with the timer, the `<Trigger>` in the example shows that when the timer control counts down from 5 seconds to 0, a call is made to the specified method. In this case, the server-side event fires and updates the label controls without the unnecessary page postback. In this example, the inventory listed on screen is updated every 5 seconds.

Figure 11-2 shows what the output might look like.



Timer Control Example:
Widget: 738

Figure 11-2

If you look at the source for this page, the code of interest is defined as:

```
<script type="text/xml-script"><page
xmlns:script="http://schemas.microsoft.com/xml-script/2005">
  <components>
    <timer interval="5000" enabled="true">
      <tick>
        <postBack target="tc" argument="" />
      </tick>
    </timer>
  </components>
</page></script>
<script
type="text/javascript">Sys.WebForms._PageRequest._setupAsyncPostBacks(document.getE
lementById('form1'), 'ScriptManager1');
</script>
```

This code contains the definition for the component configuration on the client. This code is generated by the server-side `TimerControl`. When the server control is rendered to the client, this is the code that is generated.

Control Extenders

With ASP.NET, there is a set of controls that developers have become fairly familiar with. These controls include things like the textbox, label, drop-down list box, and many others. One of the questions that Atlas brings to the table is how to provide additional functionality to these controls, while maintaining the programming model that developers have come to be familiar with. Into this problem step *control extenders*. With control extenders, client-side functionality is added to an existing server-side control, while maintaining the server-side programming model. There are numerous examples of these extensions of existing controls.

You are going to see the `AutoComplete` extender control in the next section. The `AutoComplete` extender control is designed to extend the capabilities of an ASP.NET `Textbox` control. The `AutoComplete` control will extend the functionality of the `Textbox` control by hooking it to a web service to get information.

In Chapter 12, the Drag and Drop extender will be presented. It will provide Drag and Drop support through Atlas.

AutoComplete

One of the classic examples of Ajax has been a textbox that is similar in concept to the Windows combo box (similar, but not quite the same). This feature has been popularized by the Google Suggest service.

Atlas provides the capability to extend the `<asp:Textbox>` in ASP.NET 2.0. This extension takes input from the textbox, passes the text to the web service, and the web service returns a list of possible items, similar to the Windows Combo Box. Take a look at the following simple example. Here is the ASPX code.

```
<atlas:ScriptManager ID="ScriptManager1" runat="server"
EnablePartialRendering="true" />
<div>
<asp:Textbox runat="server" id="txtBox" />
<atlas:AutoCompleteExtender runat="server" ID="ace">
    <atlas:AutoCompleteProperties TargetControlID="txtBox"
    Enabled="true" ServicePath="AutoCompleteEx.asmx"
    ServiceMethod="TextBoxAutoComplete" />
</atlas:AutoCompleteExtender>
</div>
```

Look at what is being specified.

- ❑ There is an ASP.NET textbox. This is the standard textbox in ASP.NET.
- ❑ There is a new type of server control. This is an extender control, and specifically the `AutoCompleteExtender`. It is a server control that acts on the ASP.NET textbox. The extender control "extends" the functionality in the textbox.
- ❑ There are series of properties specified within the `<atlas:AutoCompleteProperties>` tag:
    - ❑ `TargetControlID` — The `TargetControlID` is the control that will be the target of the extender.
    - ❑ `ServicePath` — The `ServicePath` property is the path to the web service that will be called.
    - ❑ `ServiceMethod` — The `ServiceMethod` property is the name of the function within the web service.

Now that you have seen the ASPX code, look at the web service:

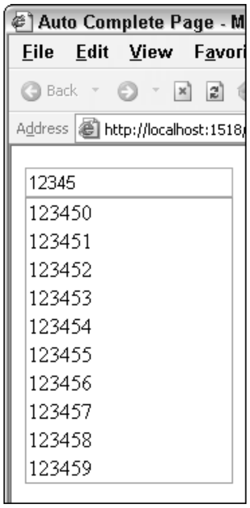
```
[WebMethod]
public String[] TextBoxAutoComplete(string prefixText, int count) // Seems to be a
problem if the names are not prefixText and count
{
    int i = 0;
    int iLength = 10;
    List<String> Values = new List<string>();

    for (i = 0; (i < iLength); i++ )
    {
        Values.Add(Convert.ToString(prefixText + i.ToString()));
    }
    String[] strReturn = new String[Values.Count];
    strReturn = Values.ToArray();
    return (strReturn);
}
```

The issues with the web service are:

- ❑ A set of strings in the form of an array must be returned.
- ❑ The input parameters must be a string and an integer. In addition, at the time of this writing, these parameters *must* be named `prefixText` and `count`. Naming these values something different will result in the code not working correctly.
- ❑ In this example, the code is designed to take an input and add the values 0–9. This code merely takes the input and adds a number. It expects a number to be input, but there is no specific checking in the example.

Now that you have seen the code, look at the output of the `AutoCompleteExtender` in Figure 11-3.



```
12345
123450
123451
123452
123453
123454
123455
123456
123457
123458
123459
```

Figure 11-3

From there, take a look at the source code generated. Here is the HTML output from the View Source functionality in Internet Explorer.

```
<script src="../ScriptLibrary/Atlas/Debug/Atlas.js"
type="text/javascript"></script>
<div>
<input name="txtBox" type="text" id="txtBox" />
</div>
<div>
   <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
value="/wEWAgK44YDODwKF+8K0ARfViGhYgqOYxdy6jHmmcbQs826z" />
</div>
<script type="text/xml-script"><page
xmlns:script="http://schemas.microsoft.com/xml-script/2005">
   <components>
    <control id="txtBox">
      <behaviors>
        <autoComplete serviceURL="AutoCompleteEx.asmx"
serviceMethod="TextBoxAutoComplete" />
      </behaviors>
    </control>
  </components>
</page></script><script
type="text/javascript">Sys.WebForms._PageRequest._setupAsyncPostBacks(document.getE
lementById('form1'), 'ScriptManager1');
</script></form>
```

In reviewing this code, take note of the following:

❑ There are no special parameters on the HTML textbox definition.

❑ There is a definition of components.

❑ The definition of components contains a control ID and a behavior. These definitions associate the textbox with the behavior.

Data Binding

Data binding allows for the interchange of data between components and user interface controls. Atlas allows datasources and data controls to directly interact in the web browser without the need to post back to the server. Atlas provides the mechanism to create datasources. These datasources provide services for performing CRUD (create, read, update, delete) style operations. The associated database operations are select, insert, update, and delete.

Atlas supports two types of data binding — declarative and programmatic. Declarative data binding is what most ASP.NET developers are familiar with, but the next two sections look at these both in more detail.

Declarative Data Binding

When a developer ties together data components and user interface components that is known as *data binding*. With Atlas and ASP.NET, there is a further type of data binding known as declarative data binding. With *declarative data binding*, all of the binding information is declared statically within a section of the web page.

You will notice that there is still some code in the example that follows that is programmatic. Declarative data binding is typically not 100 percent declarative.

Try It Out Declarative Data binding

In this example, you will take look at the pieces of code and the steps taken for getting data in a declarative manner:

```
<form id="form1" runat="server">
<atlas:ScriptManager runat="server" ID="ScriptManager1" >
   <Services>
      <atlas:ServiceReference GenerateProxy=true Path="WebServiceProjects.asmx" />
   </Services>
</atlas:ScriptManager>
<input type="button" id="btnGetData" value="Get Project List" onclick="GetData()"
/>
<script language="javascript">
   function GetData()
   {
      WebServiceProjects.GetProjects(OnServiceComplete);
   }
   function OnServiceComplete(result)
   {
       var projectName = $("ProjectResults");
       projectName.control.set_data(result);
   }
</script>
<div id="ProjectResults">
</div>
<div id="ProjectTemplate">
This is a list of all project in the table tblProject:<br />
<div id="ProjectItemTemplate">
    Project: <strong><span id="ProjectNameLabel"></span></strong>
</div>
</div>
</form>
<script type="text/xml-script">
<page xmlns:script="http://schemas.microsoft.com/xml-script/2005">
<components>
<listView id="ProjectResults"
   itemTemplateParentElementId="ProjectTemplate" >
   <layoutTemplate>
      <template layoutElement="ProjectTemplate" />
   </layoutTemplate>
   <itemTemplate>
      <template layoutElement="ProjectItemTemplate">
      <label id="ProjectNameLabel">
         <bindings>
            <binding dataPath="ProjectName" property="text" />
         </bindings>
      </label>
      </template>
   </itemTemplate>
</listView>
</components>
</page>
</script>
```

How It Works

Now take a look at the details of the example:

1. The page is set up just like any other page that uses Atlas. The page has the `ScriptManager` along with a reference to a web service.
2. There is a `<serviceMethod>` tag that defines a web service to call and to create the JavaScript proxy.
3. There is an onclick event defined in the HTML for the `btnGetData` button. When the web service returns data, the `OnServiceComplete` method is called and processing is completed there. Within the `OnServiceComplete` method, a reference to the `ProjectResults div` is obtained and data is bound to the div tag.
4. A "holder" for the final results is defined within the `<div>` tag with an ID of `ProjectResults`.
5. A `listView` control is defined. This `listView` control is associated with the `ProjectResults <div>` tag within the script definition.
6. The binding section defines where to get the data from. The `ProjectName` field in the web service's dataset is bound to the `projectNameLabel`.
7. The `itemTemplate` defines the items to be contained within a binding on a per row basis. In this example, the `ProjectNameLabel` output span is bound to the `ProjectName` property.

Programmatic Data Binding

Most ASP.NET developers are familiar with declarative data binding. It is also possible to programmatically set up and perform data binding programmatically. *Programmatic data binding* means you are setting up the data binding through imperative program code rather than through declarative tags and structures.

Try It Out Programmatic Data Binding

The following example uses programmatic data binding:

```
<atlas:ScriptManager runat="server" ID="ScriptManager1">
  <Services>
    <atlas:ServiceReference GenerateProxy="true" Path="WebServiceProjects.asmx" />
  </Services>
  <Scripts>
    <atlas:ScriptReference Path="~/ScriptLibrary/CustomTemplates.js" />
  </Scripts>
</atlas:ScriptManager>
<input type="button" id="btnGetData" onclick="GetData()" value="Get Project List"
/>
<script type="text/javascript">
function pageLoad()
{
    var listView = new Sys.UI.Data.ListView($("ProjectResults"));
    listView.set_itemTemplateParentElementId("ProjectTemplate");
    var layoutTemplate = new GenericTemplate($("ProjectTemplate"));
```

```
        listView.set_layoutTemplate(layoutTemplate);
        var itemTemplate = new GenericTemplate($("ProjectItemTemplate"),
    createItemTemplate);
        listView.set_itemTemplate(itemTemplate);
        itemTemplate.initialize();
        layoutTemplate.initialize();
        listView.initialize();
    }
    function createItemTemplate(markupContext, dataContext)
    {
        var
    associatedElement = markupContext.findElement("ProjectNameLabel");
        var projectNameLabel = new Sys.UI.Label(associatedElement);
        projectNameLabel.set_dataContext(dataContext);
        var bindings = projectNameLabel.get_bindings();
        var textBinding = new Sys.Binding();
        textBinding.set_property("text");
        textBinding.set_dataPath('ProjectName');
        textBinding.initialize(projectNameLabel);
        bindings.add(textBinding);
        projectNameLabel.initialize();
    }
    function GetData()
    {
        WebServiceProjects.GetProjects(OnServiceComplete);
    }
    function OnServiceComplete(result)
    {
        var projectName = $("ProjectResults");
        projectName.control.set_data(result);
    }
    </script>
    <div id="ProjectResults">
    </div>
    <div id="ProjectTemplate">
        This is a list of all projects in the table tblProject:<br />
        <div id="ProjectItemTemplate">
            Project: <strong><span id="ProjectNameLabel"></span></strong>
        </div>
    </div>
```

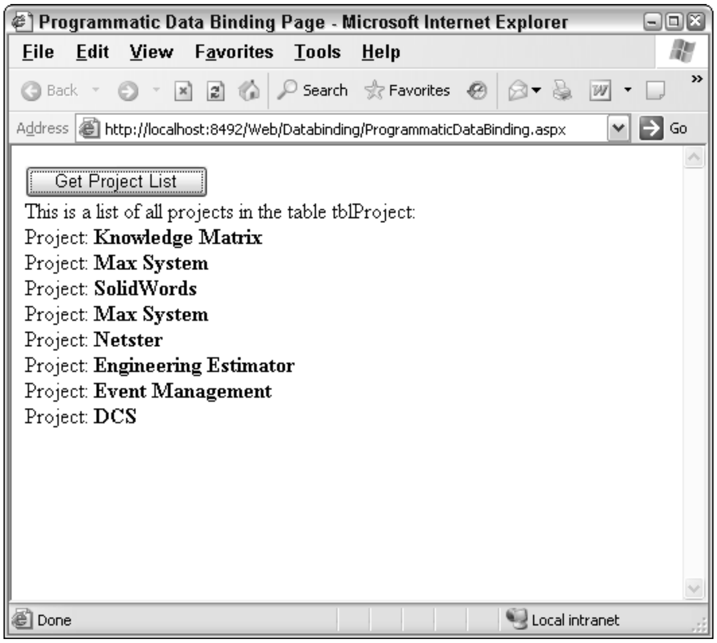
How It Works

Now, take a look at this code in a step-by-step process:

1. The display information is set up exactly like the declarative data-binding example. As a result, the pages work the same.
2. There is a custom template in a JavaScript file that is included by using the `ScriptManager` control. The custom template is defined as the class `GenericTemplate()`. This custom template makes it easier for developers to programmatically data bind.

3. The `pageLoad()` event creates and sets up the `listView`, `layoutTemplate`, and `itemTemplate` controls. The `layoutTemplate` and `itemTemplate` are defined using the `GenericTemplate` class that is defined in the included JavaScript file.
4. An item template is created by the `createItemTemplate` method. Within the `pageLoad()` event, the `createItemTemplate` method is passed as a callback to the `GenericTemplate()` class.
5. The `GetData()` method is called when the `onclick` event of the button occurs.
6. The `OnServiceComplete()` method binds the data to the `listView`.

Figure 11-4 shows the output on screen of a call to both the declarative and programmatic code versions.


Programmatic Data Binding Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8492/Web/Databinding/ProgrammaticDataBinding.aspx
Go
Get Project List
Done
Local intranet

This is a list of all projects in the table tblProject:
Project: **Knowledge Matrix**
Project: **Max System**
Project: **SolidWords**
Project: **Max System**
Project: **Netster**
Project: **Engineering Estimator**
Project: **Event Management**
Project: **DCS**

Figure 11-4

The question that you most likely have is why would a developer choose programmatic data binding versus declarative data binding. The simple answer is ease of use versus control. Programmatic data binding depends on the developer to know and understand many aspects of data binding, creating templates, and the intricacies of Atlas, which at this time are not all known. At the same time, programmatic data binding provides an amount of flexibility. Declarative data binding, on the other hand, will most likely be supported by designers, wizards, and graphical interface in a version of Visual Studio .NET after the 2005 release.

Binding Directions

As previously indicated, data binding allows data to be interchanged between components and user interface controls. This interchange may be cast in several directions — specifically, `In`, `Out`, and `InOut`. These directions are defined within the `Sys.BindingDirection` enumeration. The meanings of these directions are:

- `In` — Defines data going from a datasource into a user interface control
- `Out` — Defines data going from a user interface control in a datasource
- `InOut` — Defines data going back and forth between a user interface control and a datasource

The following code displays the allowed values of the `Sys.BindingDirection` enumeration:

```
function PerformEnumerations()
{
    var strOut = "";
    var strReturn = "<br />";
    for (var strItems in Sys.BindingDirection)
    {
        strOut += strItems + strReturn;
    }
    document.getElementById("Output").innerHTML = strOut;
}
```

Binding Transformations

Bindings provide the ability to attach handlers and methods for performing operations along with the binding. Two of the built-in transforms are `ToString` and `Invert`. The `ToString` transform converts the data into a string. The `Invert` transform is designed for boolean operations. It will output the opposite of the input value. Atlas provides the flexibility to create custom transforms.

```
var custBinding = new Sys.Binding();
..
custBinding.transform.add(CustomTransformHandler);
..
function CustomTransformHandler(sender, eventArgs) { .. }
```

The class that makes this possible is the `Sys.Bindings()` class.

Validation

Validation is a mechanism to verify data input. There are a number of ASP.NET server validators that include client functionality. Atlas provides a set of client-side controls that perform a similar function. The built-in validators are:

- `requiredFieldValidator` — Verifies that data is within the associated control
- `typeValidator` — Verifies the type of data. This may be `String` or `Number`
- `rangeValidator` — Verifies that the data within a lower and upper value

- `regexValidator` — Verifies the data against the supplied regular expression
- `customValidator` — Defines a custom expression handler

Validators are defined together through a collection and are typically fired on a `propertyChanged` event. During this event, the validation is checked, and the validate event is raised. From there, code can subscribe to a validation event. This event is raised after validation. During the validation, the validators are checked. Checking the validators may result in the client-side control `invalid` and `validationMessage` properties being set.

In addition to validators, a special control exists for displaying this information. This control is of type `validationErrorLabel` and is used to display error messages — similar in function to the ASP.NET `ValidationSummary` control. The error messages may be displayed through an asterisk, tooltip, or other mechanism. In addition, validators may be grouped together. A rollup of the validator controls can then occur.

Try It Out Using a requiredFieldValidator

In the following code, you are going to investigate the use of the `requiredFieldValidator`.

```
<form id="form1" runat="server">
    <atlas:ScriptManager runat="server" ID="ScriptManager1" />
    <div class="description">
        The textbox requires data entry. A requiredFieldValidator is attached.
        Enter a text field, then remove to see the effect.  The validator is shown
        via the tooltip.
        <br /><br />
        <input type="text" id="textboxRequired" class="input" />
        &nbsp;
        <span id="valRequired" style="color: red">*</span>
    </div>
    <script type="text/xml-script">
        <page xmlns:script="http://schemas.microsoft.com/xml-script/2005">
            <components>
                <textBox targetElement="textboxRequired">
                    <validators>
                        <requiredFieldValidator errorMessage="You must enter some
text." />
                    </validators>
                </textBox>
                <validationErrorLabel targetElement="valRequired"
associatedControl="textboxRequired" />
            </components>
        </page>
    </script>
```

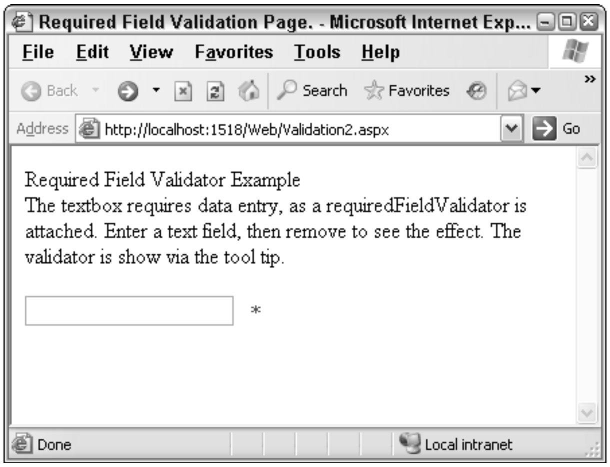
How It Works

Now look at the steps taken to get this code to work.

1. An HTML textbox and span are defined along with the Atlas `ScriptManager`.
2. Within the `xml-script` section, there is a set of defined components. The `textBox` is defined, and a `validator` is assigned to the `textBox` control.

3. The `validationErrorLabel` is set up with a `targetElement` and the `associatedControl`.

Figure 11-5 shows the output of the preceding required field validator.


Required Field Validation Page. - Microsoft Internet Exp...
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:1518/Web/Validation2.aspx Go
Done Local intranet

Required Field Validator Example
The textbox requires data entry, as a requiredFieldValidator is attached. Enter a text field, then remove to see the effect. The validator is show via the tool tip.

*

Figure 11-5

Try It Out Adding Datatype and Range Validation Support

The next step in this example adds datatype and range validation support. By making the changes that follow to the `xml-script` section, the validation is added. In this example, the `<textbox>` tag contains a `<validators>` tag. Within the `<validators>` tag, there are several validators that are set up. The first validator is the `requiredFieldValdiator`. The `requiredFieldValidator` requires that a value be entered for that field. The second validator is the `typeValidator`. The `typeValidator` sets the type that must be entered. In this example, the `Number` type must be entered. The third validator in this example is the `rangeValidator`. In this example, if a number outside of the range from 10 to 20 is entered, a message is presented to the user regarding the number being out of the specified range.

```
<page xmlns:script="http://schemas.microsoft.com/xml-script/2005">
    <components>
        <textBox targetElement="textboxRequired">
            <validators>
                <requiredFieldValidator errorMessage="You must enter some text." />
                <typeValidator type="Number" errorMessage="You must enter a valid
number." />
                <rangeValidator lowerBound="10" upperBound="20" errorMessage="You
must enter a number between 10 and 20." />
            </validators>
        </textBox>
        <validationErrorLabel targetElement="valRequired"
associatedControl="textboxRequired" />
    </components>
</page>
```

Try It Out Regex-Based Validation

This next example involves the source code to perform a `regex`-based validation. This example is similar to the previous example. There is an `xml-script` section that contains a list of components. In this situation, the text that is entered is validated for being in the form of an email address. In this example, a `regexValidator` is placed within the `<validators>` tag. Within the `regexValidator`, a regular expression is passed for processing by the validator.

```
<form id="form1" runat="server">
    <atlas:ScriptManager runat="server" ID="ScriptManager1" />
    <div class="description">
        Regex Validation Example.  Enter a valid email address. The validator is
show
        via the tooltip.
        <br /><br />
        <input type="text" id="textboxRegex" class="input" />
        &nbsp;
        <span id="valRegex" style="color: red">*</span>
    </div>
    <script type="text/xml-script">
        <page xmlns:script="http://schemas.microsoft.com/xml-script/2005">
            <components>
                <textBox targetElement="textboxRegex">
                    <validators>
                        <requiredFieldValidator errorMessage="You must enter some
text." />
                        <regexValidator regex="/(\w[-._\w]*\w@\w[-
._\w]*\w\.\w{2,3})/"  errorMessage="You must a valid email address in the form of
an email address." />
                    </validators>
                </textBox>
                <validationErrorLabel targetElement="valRegex"
associatedControl="textboxRegex" />
            </components>
        </page>
    </script>
</form>
```

Although the code looks the same as the previous validator example, there is one important difference. The `regex` that is used for the validation has "`/`" and "`/`" characters at the beginning and the end of the string representing the `regex` statement.

You may question under what situation a regular expression validation would need to occur. While testing for datatypes, required fields, and ranges can meet many validation requirements, there are situations where looking for pattern is required. In those scenarios, using regular expressions will work well in meeting those requirements.

Try It Out Custom Validation

Next, take look at a custom validator. The code will look very similar to the existing validator code. There is a `customValidator` within the `<components>` section that will call the custom JavaScript routine. Take a look at the code for this:

```
<form id="form1" runat="server">
<atlas:ScriptManager runat="server" ID="ScriptManager1" />
<div class="description">
    Regex Validation Example.  Enter a value.  Entering the term "fail" will cause
the validation to fail.
    <br /><br />
    <input type="text" id="textboxValue" class="input" />
    &nbsp;
    <span id="valLabel" style="color: red">*</span>
</div>
<script language="javascript">
function onValidateValue(sender, eventArgs) {
    var val = eventArgs.get_value();
    var valid = true;
    if (val == "fail")
    {
        valid = false;
    }
    //You could do something like this to send an alert to to the user.
    //alert("The entry is: " + valid);
    eventArgs.set_isValid(valid);
}
</script>
<script type="text/xml-script">
    <page xmlns:script="http://schemas.microsoft.com/xml-script/2005">
        <components>
            <textBox targetElement="textboxValue">
                <validators>
                    <requiredFieldValidator errorMessage="You must enter some
text." />
                    <customValidator validateValue='onValidateValue'
errorMessage="You entered fail." />
                </validators>
            </textBox>
            <validationErrorLabel targetElement="valLabel"
associatedControl="textboxValue" />
        </components>
    </page>
</script>
```

There are a couple of key things to pay attention to in the `customValidator` when writing a custom validator.

- The `onValidateValue` method takes two parameters, just like an event in .NET, such as pressing a button in a web form.
- The `customValidator` takes a JavaScript function as the parameter in the `validateValue` function.

Now that you have seen `requiredFieldValdiator`, `typeValidator`, `rangeValidator`, and `regexValidator`, I am sure that you are asking, "Why would I need to use a `customValidator`?" That is an excellent question. There are situations where data must be validated against custom business rules. There might be a need perform a more complicated validation, for example to validate some data against a database. It's not possible to do this through the other validators. The custom validator allows for more programmatic options when validating.

Try It Out Group Validation

Now that you have seen the code for custom validators, you can tie two or more validators together for a group validation.

```
<atlas:ScriptManager runat="server" ID="ScriptManager1" />
<div id="lblValid" >Valid.  Good data has been entered.<br /><br /></div>
<div id="lblInValid">Invalid.  Bad data has been entered.  Please review your
inputs.<br /><br /></div>
<div class="description">
    This is the group validation page.  This example demonstrates the validation of
two controls together.
    Regex Validation Example.  Enter a value.  Entering the term "fail" will cause
the validation to fail.
    <br /><br />
    <input type="text" id="textboxValue" class="input" />
    &nbsp;
    <span id="valLabel" style="color: red">*</span>
    <br /><br /><br />
    A requiredFieldValidator, typeValidator, and rangeValidator are attached.
    Enter a number between 10 and 20, then remove to see the effect.  The validator
is show
    via the tooltip.
    <br /><br />
    <input type="text" id="textboxRequired" class="input" />
    &nbsp;
    <span id="valRequired" style="color: red">*</span>
</div>
<script language="javascript">
function onValidateValue(sender, eventArgs) {
    var val = eventArgs.get_value();
    var valid = true;
    if (val == "fail")
    {
        valid = false;
    }
    //You could do something like this to send an alert to to the user.
    //alert("The entry is: " + valid);
    eventArgs.set_isValid(valid);
}
</script>
<script type="text/xml-script">
    <page xmlns:script="http://schemas.microsoft.com/xml-script/2005">
        <components>
            <textBox targetElement="textboxValue">
                <validators>
                    <requiredFieldValidator errorMessage="You must enter some
text." />
                    <customValidator validateValue='onValidateValue'
errorMessage="You entered fail." />
                </validators>
            </textBox>
            <validationErrorLabel targetElement="valLabel"
associatedControl="textboxValue" />
```

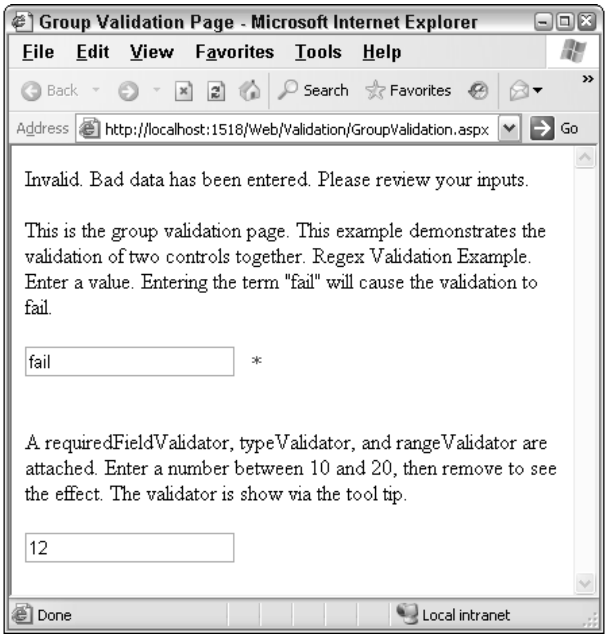
```
            <textBox targetElement="textboxRequired">
                <validators>
                    <requiredFieldValidator errorMessage="You must enter some
text." />
                    <typeValidator type="Number" errorMessage="You must enter a
valid number." />
                    <rangeValidator lowerBound="10" upperBound="20"
errorMessage="You must enter a number between 10 and 20." />
                </validators>
            </textBox>
            <validationErrorLabel targetElement="valRequired"
associatedControl="textboxRequired" />
            <validationGroup id="formGroup" targetElement="formGroup">
                <associatedControls>
                    <reference component="textboxValue" />
                    <reference component="textboxRequired" />
                </associatedControls>
            </validationGroup>
            <label targetElement="lblValid" visibilityMode="Collapse">
                <bindings>
                    <binding dataContext="formGroup" dataPath="isValid"
property="visible" />
                </bindings>
            </label>
            <label targetElement="lblInValid">
                <bindings>
                    <binding dataContext="formGroup" dataPath="isValid"
property="visible" transform="Invert" />
                    <binding dataContext="lblInValid" dataPath="text"
property="text" transform="onValidGroup" />
                </bindings>
            </label>
        </components>
    </page>
</script>
```

How It Works

In this example, take note of several things:

- Two `div` tags have been added. The IDs for the div tags are `lblValid`, which contains the text to display when all validators are satisfied with the input, and `lblInValid`, which contains the text to display when one or more validators are invalid.
- A `validationGroup` has been defined in the `xml-script` section. The validation group defines the controls that will be validated together.
- The `lblValid` and `lblInValid` tags have been added to the `xml-script` section. A set of bindings is defined for each.

Figure 11-6 shows the output of the group validation process.


Group Validation Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:1518/Web/Validation/GroupValidation.aspx Go
Done
Local intranet

Invalid. Bad data has been entered. Please review your inputs.

This is the group validation page. This example demonstrates the validation of two controls together. Regex Validation Example. Enter a value. Entering the term "fail" will cause the validation to fail.

fail *

A requiredFieldValidator, typeValidator, and rangeValidator are attached. Enter a number between 10 and 20, then remove to see the effect. The validator is show via the tool tip.

12

Figure 11-6

Why would you want to group validators together? Obviously, each individual validator will fire. However, there may be situations where the validation should be done together. For example, it would be valuable for checking required fields when a user signs up for a new service. This is what the `validationGroup` control in Atlas is for.

Behaviors

A *behavior* is the name of the set of actions that can be performed based on events in DTHML. These events might be `click`, `hover`, `mouseover`, or other client-side events. These sets of actions that are performed comprise features such as auto-completion and Drag and Drop. In other words, behaviors are used to provide a more sophisticated UI and behavioral features beyond standard DHTML In Atlas, behaviors are defined as a collection on a client-side control. In other words, the individual behaviors are attached to a client-side control.

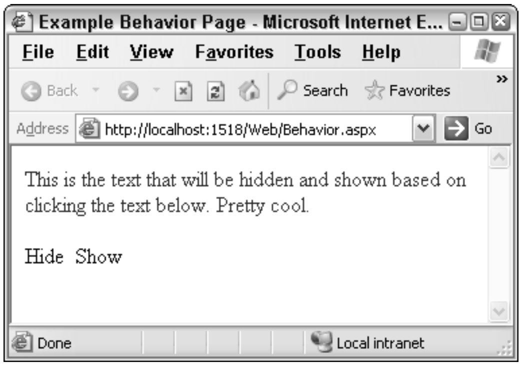
Try It Out Using Behaviors

Take a look at some code that incorporates behaviors. In this example, a `click` behavior is set on the `lblHide` label so that when it is clicked, the visibility of the `displayData` label is set to `false`, and the `displayData` label is hidden from view. When the `lblShow` label is clicked, the visibility of the `display Data` label is set to `true`, and the `displayData` label is displayed on the screen, if it is not already viewable.

```
<atlas:ScriptManager runat="server" ID="ScriptManager1" />
<div>
<div id="displayData">This is the text that will be hidden and shown based on
clicking the text below.  Pretty cool.</div>
```

```
            <br />
            <span id="lblHide" >Hide</span>&nbsp;
            <span id="lblShow" >Show</span>
    </div>
    <script type="text/xml-script">
        <page xmlns:script="http://schemas.microsoft.com/xml-script/2005">
            <components>
                <control targetElement="displayData" cssClass="start" />
                <label targetElement="lblHide">
                    <behaviors>
                        <clickBehavior>
                            <click>
                                <setProperty target="displayData"
    property="visible" value="false" />
                            </click>
                        </clickBehavior>
                    </behaviors>
                </label>
                <label targetElement="lblShow">
                    <behaviors>
                        <clickBehavior>
                            <click>
                                <setProperty target="displayData"
    property="visible" value="true" />
                            </click>
                        </clickBehavior>
                    </behaviors>
                </label>
            </components>
        </page>
    </script>
```

Figure 11-7 shows the output of the preceding code. Clicking Hide will hide the text, assuming that the code is visible. Clicking Show will display the code, assuming that it is hidden. The advantage to using behaviors in this way is that no programming must be done in the preceding code. You don't have to set up any JavaScript onclick events or anything to that effect.


Example Behavior Page - Microsoft Internet E...
File Edit View Favorites Tools Help
Back
Search
Favorites
Address
http://localhost:1518/Web/Behavior.aspx
Go
Done
Local intranet

This is the text that will be hidden and shown based on clicking the text below. Pretty cool.

Hide Show

Figure 11-7

Resources Used

- **Wilco Bauwer** — Wilco Bauwer, a intern on the Microsoft Atlas team provided significant assistance in answering questions regarding many features in Atlas. Wilco's web site and blog are located at `www.wilcob.com`.
- **Nikhil Kothari** — Nikhil provided several helpful articles on his blog regarding how to properly use several features of Atlas. Nikhil's web site and blog are located at `www.nikhilk.net`.
- **Atlas Quickstarts** — The Atlas web site is `http://atlas.asp.net`.
- **Forums on ASP.NET site** — The forums are located at `http://forums.asp.net`.

Summary

In this chapter, you have been introduced some very new and important concepts. These are

- Programming controls through Atlas
- Working with server controls
- Using data binding
- Using behaviors

From these you have seen that there is a lot of functionality in the server controls. Along with that functionality is the ability to extend the server control and add new functionality to the server controls. The integration with the server controls is very important as it brings the server-side methodology of ASP.NET and allows it to provide significant client-side functionality.

Now that you have looked at how to integrate Atlas with server controls, in the next chapter, you are going to look at integrating Atlas with membership, profiles, and other services provided by ASP.NET.

Beginning Ajax with ASP.NET

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2006 by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN-13: 978-0-471-78544-6
ISBN-10: 0-471-78544-X

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

1B/QT/QY/QW/IN

Library of Congress Control Number: 2006016507

About the Authors

Wallace B. "Wally" McClure graduated from the Georgia Institute of Technology in 1990 with a Bachelor of Science degree in electrical engineering. He continued his education there, receiving a master's degree in the same field in 1991. Since that time, he has done consulting and development for such organizations as The United States Department of Education, Coca-Cola, Bechtel National, Magnatron, and Lucent Technologies, among others. Products and services have included work with ASP, ADO, XML, and SQL Server, as well as numerous applications in the Microsoft .NET Framework. Wally has been working with the .NET Framework since the summer of 2000. Wally McClure specializes in building applications that have large numbers of users and large amounts of data. He is a Microsoft MVP and an ASPInsider, and a partner in Scalable Development, Inc. You can read Wally's blog at `http://weblogs.asp.net/wallym`. Wally and coauthor Paul Glavich also co-host the ASP.NET Podcast. You can listen to it at `www.aspnet podcast.com`. In addition, Wally travels around the southeast United States doing user group talks and sessions at various CodeCamps.

When not working or playing with technology, Wally tries to spend time with his wife Ronda and their two children, Kirsten and Bradley. Occasionally, Wally plays golf and on July 30, 2005, broke par on a real golf course for the first time in his life. If he hadn't been there, he would not have believed it.

Scott Cate is the President of myKB.com, Inc., in Scottsdale, Arizona. myKB.com, Inc., is a technology company specializing in commercial ASP.NET applications. His product line includes myKB.com (knowledge base software), kbAlertz.com (Microsoft knowledge base notifications), and EasySearchASP.net (a pluggable search engine for ASP.NET sites). Scott also runs AZGroups.com (Arizona .NET user groups), one of the largest and most active user group communities in the country, and is a member of ASPInsiders.com, a group devoted to giving early feedback to the Microsoft ASP.NET team. In addition, Scott has coauthored the novel *Surveillance*, which can be found at `http://surveillance-the-novel.com`.

Paul Glavich is currently an ASP.NET MVP and works as a senior technical consultant for Readify. He has over 15 years of industry experience ranging from PICK, C, C++, Delphi, and Visual Basic 3/4/5/6 to his current specialty in .NET C++ with C#, COM+, and ASP.NET. Paul has been developing in .NET technologies since .NET was first in beta and was technical architect for one of the world's first Internet banking solutions using .NET technology. Paul can be seen on various .NET related newsgroups, has presented at the Sydney .NET user group (`www.sdnug.org`) and is also a board member of ASPInsiders (`www.aspinsiders.com`). He has also written some technical articles that can be seen on community sites, such as ASPAlliance.com (`www.aspalliance.com`).

On a more personal note, Paul is married with three children and two grandkids, and holds a third degree black belt in budo-jitsu.

Craig Shoemaker can't sit still. As the host of the Polymorphic Podcast (`polymorphicpodcast.com`), Craig teaches on topics as timely as software architecture and as cutting edge as the latest Ajax technologies. Whether he's writing for *CoDe Magazine*, ASPAlliance, or DotNetJunkies or speaking at local user groups, Southern California Code Camp, or VSLive!, Craig loves to share his passion for the art and science for software development. Craig is also a full-time software engineer for Microsoft Certified Partner PDSA, Inc. (`pdsa.com`) in Tustin, California.