7

Ajax.NET Professional Library

Every once in a while, a technology is extremely simplified with the introduction of new wrapper libraries. These libraries use existing technologies but make the development process easier to use by wrapping the sometimes difficult concepts into easier-to-use, more simplified concepts. So, the term *wrapper library* comes from having a library of code wrapped around existing technology. You can tell when a great wrapper library is released because of its instant popularity.

This chapter covers one such wrapper library known as the Ajax library for .NET. In this chapter and the next, we will show off the simplicity of talking back and forth between client browsers and your application server without page postbacks. We'll also dig a little under the hood of the library to show you how and why the library works.

This chapter shows you how to get started using the Ajax.NET Pro library. To get started, you'll set up a simple example and get it working. The following topics will be covered:

- ❑ Acquiring Ajax.NET Pro
- ❑ Adding a reference to the Ajax.NET Pro assembly
- ❑ Setting up the `Web.Config` to handle Ajax requests
- ❑ Registering the `page` class
- ❑ Writing methods in code-behind to be accessible on the client
- ❑ Examining the request
- ❑ Executing the Ajax method and getting a server response
- ❑ Digging into callbacks and context
- ❑ Trapping errors

When you have finished these examples, you will have completed your first implementation of the Ajax.NET Pro library. You will have successfully set up an ASP.NET page that uses the library to refresh parts of your page with data from the web server.

Acquiring Ajax.NET Pro Version 6.4.16.1

In Chapters 7 and 8, we're using and talking about Ajax.NET Pro version 6.4.16.1. As with all software, this library is evolving and continually being added to and upgraded. We've made the version 6.4.16.1 library available to you for downloading on our web site, `http://BeginningAjax.com`. You can download the version in one of two ways:

- ❑ *Compiled Library*, ready to use
- ❑ *Library Source Code*, must be compiled first

I would recommend that first you download the Compiled Library. This is a simple zip file that contains a single file named `Ajax.NET`. This is the already compiled library that is ready for you to start using as a reference in the next section. If you would like to have access to the source code, you can download the Library Source Code, which has all the source code files needed for you to do the compiling yourself; then the code can be embedded into your application.

Preparing Your Application

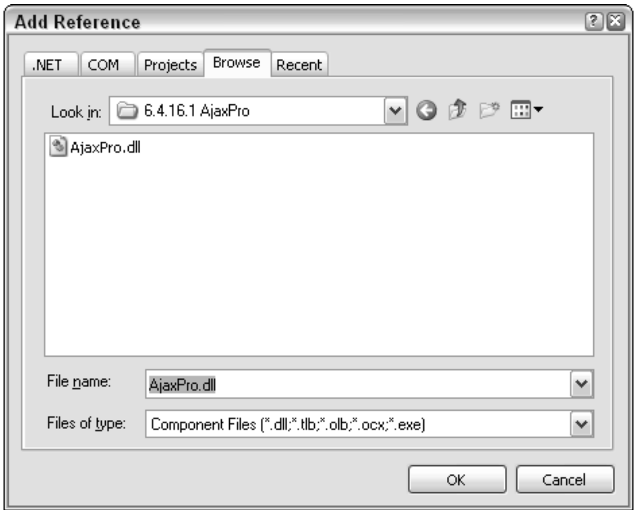
In order to prepare your application to use Ajax.NET Pro, follow these two steps:

1. Add a reference to the Ajax.NET Pro library.
2. Wire up the Ajax.NET Pro library in the `Web.Config` file so that your application can process the special requests created by the Ajax.NET Pro library.

Try It Out — Preparing Your Application to Use Ajax.NET Pro

1. To use the Ajax.NET Pro library, your first step is to set a reference to the library. This allows you to use library functionality inside your application. Create a new web site in Visual Studio. Visual Studio 2005 automatically creates a `Bin` folder for you. Right-click on this folder and select `Add Reference`. Figure 7-1 shows the `Add Reference` dialog box. Select the Browse tab, and navigate to the `AjaxPro.dll` file that you downloaded (or compiled from the Library Source Code). Once this is selected, click the OK button, and you will have successfully referenced the Ajax.NET Pro library from your application.
2. The Ajax.NET Pro library uses a page handler to process requests that come into the server from your client application. This page handler needs to be wired up to your application, and this is done by an inserting it into your `Web.Config` file. This code should be inserted in the `<system.web>` section of `Web.Config`:

```
<httpHandlers>
  <add verb="POST, GET" path="AjaxPro/*.ashx"
type="AjaxPro.AjaxHandlerFactory,AjaxPro" />
</httpHandlers>
```



Look in: 6.4.16.1 AjaxPro

AjaxPro.dll

File name: AjaxPro.dll

Files of type: Component Files (*.dll;*.tlb;*.olb;*.ocx;*.exe)

Figure 7-1

If you don't fully understand what an `HTTP` handler is, you're not alone. This code basically tells ASP.NET to take ownership of all requests that come into your web site with the path of `/AjaxPro/` and have a file extension of `.ashx`, and then process that request with the Ajax.NET Pro library. Later you'll see JavaScript that is loaded dynamically from these `*.ashx` paths. When you see URLs like this, remember that they're being processed by the Ajax.NET Pro library.

You'll examine what is happening with those requests later in this chapter.

Using the Ajax.NET Pro Library

Now that your application is set up to use the Ajax.NET Professional library, you are ready to benefit from the ease of use the library offers. In Chapter 2, you saw how a JavaScript method could be used to change an image. In the first example here, you'll perform that same functionality, but instead of changing the image client side from left to right and back again, you'll ask the server for an image to display.

There are three steps required to use the Ajax.NET Pro library in your application:

1. First, you write the code that is going to be used in your image switching routine.
2. Second, you wire up that code to be used by the Ajax.NET Pro library.
3. Third, you execute that code from JavaScript.

So, your goal in this example is to switch an image by using JavaScript as you did in Chapter 2. However, the major difference will be that you ask the server for an image name, and the response from the server will become the `src` attribute of your image.

The server-side code that is responsible for switching the image looks like this:

ChangeImage Method for Code-Behind Page

```
public string ChangeImage(string input, string left, string right)
{
    //Get the image filename without the file extension
    string filename = System.IO.Path.GetFileNameWithoutExtension(input);
    //Check if the strings match, ignoring case
    if (string.CompareOrdinal(filename, left) == 0)
    {
        //if the strings match then send back the 'right' string
        return input.Replace(filename, right);
    }
    //strings did not match, send back 'left' string
    return input.Replace(filename, left);
}
```

The `ChangeImage` method accepts three parameters; an `input` string, which is the path of the current image that is loaded; a `left` string, which defines what the Left image `src` should be; and a `right` string, which defines the Right image `src`. Calling this method in code would look something like this:

```
MyImage.ImageUrl = ChangeImage(MyImage.ImageUrl, "ArrowLeft", "ArrowRight");
```

This is straightforward code that switches the image.

Try It Out Placing the Image-Switching Code in Your Page

1. Create a page in the root of your web site called `ImageSwitcher.aspx`.
2. Right-click on this file, and select Use as Default, so that when you run your application, this is the page that will be shown in your browser. By default, Visual Studio creates an `ImageSwitch.aspx.cs` file for you.
3. Add `using AjaxPro;` with all the other using statements at the top of your page.
4. Open this file, and insert the `ChangeImage()` method just below your `Page_Load` method. Compile your project.

At this point, your project should compile with zero errors. If you do have compile errors, it's most likely because you haven't referenced the `AjaxPro` assembly correctly, as shown in Figure 7-1. If you run your project, you will not see any page output because you haven't done any UI work just yet. That will come later.

You have completed Step 1 in using the Ajax.NET Pro library in your application. However, the real magic in this example is in Step 2 — making that code accessible using JavaScript so that you can access this functionality in the client browser. This is very simple to do using the Ajax.NET Pro library, and that is just where you're going with this example. One of the nicest features of the Ajax.NET Pro library is that you can easily adapt your existing code without rewriting it. Yes, you read that correctly — you do not have to rewrite any of your code to make it available in your JavaScript. All you have to do is register your code with the Ajax.NET Pro library. That sounds kind of strange — register your code — doesn't it? The first two of the three steps in using the Ajax.NET library are the easiest to implement. And if your code is already written, this next step should take you only about 2 minutes. To make your code accessible using JavaScript, you first register your `page` class with the Ajax.NET Pro library. This class has method(s) on it that you want to expose to JavaScript. Then, you register the method you want to expose. This is all explained in the next two sections.

Registering Your Page for Ajax.NET Pro

Registering your `page` class with the Ajax.NET Pro library is what activates the library. This is the command that generates a JavaScript object that you can use on the client browser. Registering your `page` class is very simple and is done with just one line of code. This single line of code needs to be executed somewhere in the page's lifecycle and is generally inserted into the `Page_Load` method.

```
protected void Page_Load(object sender, EventArgs e)
{   AjaxPro.Utility.RegisterTypeForAjax(typeof(Chapter7_ImageSwitcher));
}
```

By default, your `page` class is the same as the page name that you assigned to the `.aspx` file, preceded by any folder structure that the file is in. In this case, the file `ImageSwitcher.aspx` is in a `/Chapter7/` folder, so the name is automatically created as `Chapter7_ImageSwitcher`. This can get out of sync if you've used the `rename` function in Visual Studio. You can confirm your `page` class name in two places if your application is compiling.

❑ At the top of the `.aspx` page in the page directive, you'll see an `Inherits` attribute. This is your `page` class name.

```
<%@ Page Language="C#" Inherits="Chapter7_ImageSwitcher" %>
```

❑ The second place you can check your `page` class name is in the `.cs` file. The `.cs` file actually defines a partial class that is shared with the same class your `.aspx` page is inherited from. The class signature is the class name.

We bring this up only because if you rename an `.aspx` page, Visual Studio will rename the actual files, but it will not rename the `page` class.

```
public partial class Chapter7_ImageSwitcher: System.Web.UI.Page
```

Remember that C# is a case-sensitive, so `imageswitcher` is different from `ImageSwitcher` is different from `imageSwitcher`. If you've incorrectly cased the name, your application shouldn't compile.

Registering Your Methods for Ajax.NET Pro

Now that you've registered your page, the next step is to register your page methods. You can't have one without the other. You have to register a class that has Ajax.NET Pro method(s) on it. You've already added the `ChangeImage()` method to your `ImageSwitch.aspx.cs` file. Remember, I said you can call this code in JavaScript without rewriting any of it. Here is the magic of the library. Simply mark your method with an `AjaxPro.AjaxMethod()` attribute. If you've never used attributes before you're in for a great surprise. This is a simple way of decorating your existing code. Just add the following line preceding your `ChangeImage()` method:

```
[Ajax.AjaxMethod()]
```

So, your entire server-side `ImageSwitch.aspx.cs` code file should look like this:

Server Side — Chapter7_ImageSwitcher.aspx.cs

```
protected void Page_Load(object sender, EventArgs e)
{
    AjaxPro.Utility.RegisterTypeForAjax(typeof(Chapter7_ImageSwitcher));
}

[AjaxPro.AjaxMethod()]
public string ChangeImage(string input, string left, string right)
{
    //Get the image filename without the file extension
    string filename = System.IO.Path.GetFileNameWithoutExtension(input);
    //Check if the strings match, ignoring case
    if (string.CompareOrdinal(filename, left) == 0)
    {
        //strings match == send back 'right' string
        return input.Replace(filename, right);
    }
    //strings did not match, send back 'left' string
    return input.Replace(filename, left);
}
```

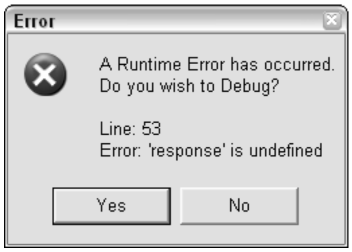
Violà! Step 2 of using the Ajax.NET Pro library in your application is done. You've now registered your `page` class, and that class has an `AjaxMethod()` in it. You can now access this method in JavaScript with a very standard syntax. Your JavaScript is going to be as simple as the following:

```
Chapter7_ImageSwitcher.ChangeImage(img.src, "ArrowLeft", "ArrowRight");
```

This line of code returns a `response` object that has a value of the URL that you want to set as the source of your image, pointing to either the left or the right image. Now you're ready to start writing the UI and the JavaScript, which is the last step in the three-step process to use the Ajax.NET Pro library.

Examining the Request Object

In the coming pages, you'll execute the preceding JavaScript line and work with the `response` object you'll get back from the JavaScript call. This object is very simple to work with and has only five properties: `value`, `error`, `request`, `extend`, and `context`. These are defined in the table that follows. All of these properties have a default value of null, so if they are never populated, you will get a client-side error if you try to use them. This is usually the culprit for the famous ever-so-unhelpful `undefined` error, as seen in Figure 7-2. It's good practice to check for null values just about everywhere. You'll see more of this as you move on, specifically under the section about trapping errors later in the chapter.


Error

A Runtime Error has occurred.
Do you wish to Debug?

Line: 53
Error: 'response' is undefined

Figure 7-2

Property	Default Value	Description
`response.value`	null	The value is populated with what is returned from the server.
`response.error`	null	The error value is either null or the error string that was returned from the server. Normally you want this to be null, although, as you'll see, it can be a nice way to provide information to yourself about the client.
`response.request`	null	This is a copy of the original `request` object that was used to issue the request back to the server. This contains two very helpful properties: `method` and `args`. `method` is the original method name that was called, and `args` is an object describing all of the values passed into the method.
`response.extend`	null	The `extend` property is a JavaScript prototype that is added to all Ajax.NET Pro objects. It is used internally by the library to bind events and is not normally used.
`response.context`	null	The `context` is optional and can be used in certain situations to pass data along from one point to another. You'll see this in use later in the chapter, where it'll be easier to understand.

Executing Your Ajax on the Client

Building on the concept you learned from Chapter 2, you'll start this example with very similar HTML, as you see in the code that follows. The HTML preloads both the left and the right image. This is simply to make the switch faster once you get a response from the server. If you didn't preload these images in a hidden `<div>` tag, the end user would have to wait while the new image was downloaded from the server. You also have a `ChangeMe()` JavaScript function that does the work of actually changing the image. So, how does this `ChangeMe()` function get fired?

Client Side — Chapter7/ImageSwitcher.aspx

```
<script type="text/Javascript" language="Javascript">
function ChangeMe(target, leftName, rightName) {
    var response = Chapter7_ImageSwitcher.ChangeImage(target.src, leftName,
rightName);
    target.src = response.value;
}
</script>

<div style="DISPLAY:none;VISIBILITY:hidden">
    <!-- preload images -->
    <img src="images/ArrowLeft.gif" border="0" runat="server" ID="ImgPreload1">
    <img src="images/ArrowRight.gif" border="0" runat="server" ID="ImgPreload2">
</div>
<img onclick="ChangeMe(this,'ArrowLeft','ArrowRight')" src="images/ArrowLeft.gif"
border="0">
```

What Is a Language Proxy?

The term *proxy* is used when one language represents an object that has all the same properties (and sometimes method calls) as an object in another language. So, a JavaScript proxy object will mimic all the properties of the .NET object. This means you can easily use the same syntax in JavaScript that you would use in .NET. It is the job of the proxy object to talk back and forth between JavaScript and .NET. Proxy classes make your programming life easier because you don't have to worry about communication between the two languages or systems.

Think of a proxy class as a language interpreter, allowing you to communicate easily in your native language, while the proxy does all the interpreting for you to the other language.

Notice that the `<img>` tag has a JavaScript `onclick` attribute that points to your `ChangeMe()` function. You pass in three values: first, the `<img>` tag represented by the keyword `this,` followed by the left and right image names, `ArrowLeft` and `ArrowRight`.

The first line of the function calls into the Ajax.NET Pro library with the `Class.Method` naming convention. Remember, you registered your `page` class, which was named `Chapter7_ImageSwitcher`, and then you attributed your server-side `ChangeImage()` method with the `AjaxPro.AjaxMethod()` attribute. The Ajax.NET Pro library now makes a JavaScript object for you that is a proxy object used to communicate with your ASP.NET application.

This proxy object makes it possible for you to simply call the `Chapter7_ImageSwitcher.ChangeImage()` method, which looks like it's executing your server-side code right inside your JavaScript. What's actually happening is that the proxy object uses the same signature and naming conventions that your server-side code uses, making it look transparent when you call your server code from JavaScript.

When the end user clicks the image, the `ChangeMe()` function is called. The first thing you do is build a response variable that will hold the return value of your `Chapter7_ImageSwitcher.ChangeImage()` method. Notice that you also pass in the appropriate parameters to your server-side method to the proxy. Finally, you get back a value from the server in the `response.value` property, and that becomes the new source value of the image, which is the new URL of the image to be used.

This is the same client-side effect that you saw in Chapter 2, changing the image from left to right when the user clicked the image. However, this time the value came from the server. Although this example is pretty basic, it is very important because what you've really built so far is the ability to use Ajax.NET Pro between the client browser and your ASP.NET server to return a single string from the server and then update the client page.

Imagine the possibilities here. You could return the HTML of an entire datagrid and update the `innerHTML` of a `div` tag. With a little Dynamic HTML (DHTML) you can change the style of a `div` or `span` tag and create some very powerful features using just what you've learned so far.

Digging into response.value

In the first example, you received a string in the `response.value` property. This was the string name of the image `LeftArrow` or `RightArrow` that you used to set the image source to change the image from right to left. Strings can be very helpful and are a very common datatype to be passed back to the client from the server. Imagine using HTML in your return strings. What if you rendered an entire control, such as a datagrid, to its HTML and then returned that HTML as your string value. With this logic, it's pretty easy to magically load a datagrid client side. The code that follows is used to render a control to HTML. The next example then builds a datagrid, and you can easily adapt these examples to any control to get its rendered HTML. The reason that this works is that all controls have a `RenderControl()` method.

RenderControlToHtml() — Getting the HTML String from a Control

```
public string RenderControlToHtml(Control ControlToRender)
{
    System.Text.StringBuilder sb = new System.Text.StringBuilder();
    System.IO.StringWriter stWriter = new System.IO.StringWriter(sb);
    System.Web.UI.HtmlTextWriter htmlWriter = new
System.Web.UI.HtmlTextWriter(stWriter);
    ControlToRender.RenderControl(htmlWriter);
    return sb.ToString ();
}
```

Rendering a Datagrid

```
[AjaxPro.AjaxMethod()]
public string CreateNewDataGrid()
{
    DataGrid myDataGrid = new DataGrid();
    myDataGrid.ShowHeader = false;
    myDataGrid.DataSource = BuildMultiplicationTable();
    myDataGrid.DataBind();
    return RenderControlToHtml(myDataGrid);
}

public DataTable BuildMultiplicationTable()
{
    //Build a Data Table with 11 cells
    DataTable myTable = new DataTable();
    for (int i = 1; i < 11; i++)
        myTable.Columns.Add(new DataColumn(i.ToString()));

    //Populate 10 rows with a 10X10 multiplication chart
    for (int i = 1; i < 11; i++)
    {
        DataRow row = myTable.NewRow();
        for (int j = 1; j < 11; j++)
        {
            row[j-1] = i*j;
        }
        myTable.Rows.Add(row);
    }

    return myTable;
}
```

Using strings will get you moving quickly on your project. After all, just about anything can be converted to a string, but as you'll see in the next section, it's also nice to be able to return custom objects. Ajax.NET Pro has support out of the box for the following .NET types. Any type on this list can be returned from your server side function to your client side call with no additional programming.

- Strings
- Integers
- Double
- Boolean
- DateTime
- DataSet
- DataTable

All the types in this list are pretty easy to understand with exception of the last two items. What would you do with a `DataSet` in JavaScript on the client? What about a `DataTable`? Well, it turns out that you would do pretty much the same thing as you would on the server. You do not have `Databind()` operations as you would on the server, but you do have a `DataTable.Tables` array, each table has a `Rows` array, and finally, each row has properties that are inline with the original table column names.

Try It Out Building an HTML Table from a DataTable Object

The following code gets a `DataTable`, server side, and draws an HTML table on the client.

Client Code — Building an HTML Table from a DataSet

```
<script type="text/javascript" language="javascript">
function BuildHtmlTable() {
    var response = Chapter7_BuildHtmlTable.BuildMultiplicationTable();
    if(response.value != null && response.value.Rows.length>0) {
        var datatable = response.value;
        var table = new Array();
        table[table.length] = '<table border=1>';
        for(var r=0; r<datatable.Rows.length; r++) {
            var row = datatable.Rows[r];
            table[table.length] = '<tr>';
            for(var c=0; c<datatable.Columns.length; c++) {
                table[table.length] = '<td valign=top>' + row[c+1] + '</td>';
            }
            table[table.length] = '</tr>';
        }
        table[table.length] = '</table>';
        document.getElementById("DynamicTable").innerHTML  = table.join('');
    }
}
function ClearHtmlTable() {
    document.getElementById("DynamicTable").innerHTML = '';
}
//-->
</script>
<form id="form1" runat="server">
```

```
<p>Chapter 7 :: Build Html Table.<br />
    <a href="#" onclick="ClearHtmlTable()">Clear Html Table</a><br />
    <a href="#" onclick="BuildHtmlTable()">Build Html Table</a>
</p>
<div id="DynamicTable"></div>
</form>
```

Server Code — Returning a DataTable

```
protected void Page_Load(object sender, EventArgs e)
{
    AjaxPro.Utility.RegisterTypeForAjax(typeof(Chapter7_BuildHtmlTable));
}

[AjaxPro.AjaxMethod()]
public DataTable BuildMultiplicationTable()
{
    //Build a Data Table with 11 cells
    DataTable myTable = new DataTable();
    for (int i = 1; i < 11; i++)
        myTable.Columns.Add(new DataColumn(i.ToString()));

    //Populate 10 rows with a 10X10 multiplication chart
    for (int i = 1; i < 11; i++)
    {
        DataRow row = myTable.NewRow();
        for (int j = 1; j < 11; j++)
        {
            row[j - 1] = i * j;
        }
        myTable.Rows.Add(row);
    }
    return myTable;
}
```

Notice that the last line in the client code block is an empty `<div>` tag named `DynamicTable`. When the hyperlink "Build Html Table" is clicked, the JavaScript function `BuildHtmlTable()` is called. In the server-side code, the `page` class name is `Chapter7_BuildHtmlTable`, and the server side method that returns a `DataTable` is `BuildMultiplicationTable()`. So, inside the JavaScript function `BuildHtmlTable()`, you can call `Chapter7_BuildHtmlTable.BuildMultiplicationTable()`. Then in the `response.value`, you'll get back an object that is very similar to a server-side `DataTable`, with `Rows` and `Columns` properties. As with the `DataTable` object, Ajax.NET Pro also has a `DataSet` object that can be used just as easily.

Returning Custom Objects

So, what about custom objects? Suppose that you have a person class, and the person has name, street, city, state, zip, and phone number properties. Can you return this person from your function and have it magically converted for you? Luckily, the answer is yes — but there is a catch. The custom class that you're returning needs to be registered, just like the page class needed to be registered (you'll see why this needs to happen in Chapter 8), and the custom class needs to be marked with a `Serializable()` attribute. The code block in this section shows the partial code for the custom person class, just showing the name and street properties. To use this class as a return type, it would need to be registered, and the proxy class will automatically be processed by the Ajax.NET Pro library.

```
Ajax.Utility.RegisterTypeForAjax(typeof(Person));
```

The only down side to this is that the automatic conversion is one-way. It allows for you to serialize your .NET objects to JavaScript Object Notation (JSON as described in Chapter 5) for use in JavaScript, but it does not allow you to use the `Person` class as an input value from JavaScript back to .NET. It is possible to send this `Person` class back to .NET (maybe as a parameter), but it's not automatic. You'll look at how to get a custom class back to .NET form JavaScript in Chapter 8.

A Simple Person Class Marked with [Serializable()] Attribute

```
[Serializable()]
public class Person
{
    public Person()
    { }

    private string _Name;
    public string Name
    {
        get { return _Name; }
        set { _Name = value; }
    }

    private string _Street;
    public string Street
    {
        get { return _Street; }
        set { _Street = value; }
    }
}
```

More Advanced Callbacks and Context

The preceding method of using the Ajax.NET Pro library is great, but it's worth going to the next step here. Did you notice that the way the response is requested was inline in your code? What if it took 10 seconds to execute your method and get a response back to the client? That means your browser would be locked and waiting for the response to come back, and nothing else could happen while you were waiting. This, as you've encountered previously in this book, is called *synchronous* execution. There is a very simple way to make this code execute *asynchronously*, which is to say that the request will be fired, but you're not going to wait for the response. Instead you're going to tell the Ajax.NET Pro library what to do with the response once it is returned. This is accomplished with a callback routine. A *callback* is simply another function that can be called where the response can be passed in.

Take a look at a minor difference in this JavaScript `ChangeMe()` function, and compare it to the one earlier in the chapter in the "Executing Your Ajax on the Client" section.

```
function ChangeMe(target, onName, offName) {
    Chapter7_ImageSwitcherCallback.ChangeImage
             (target.src, onName, offName, ChangeMe_Callback,target);
}
```

Notice the number of parameters that you're passing into the `ImageSwitcher.ChangeImage()` method. There are now five parameters, where before there were only three. The server-side method accepts only three, so how can this work? Remember that this is actually a JavaScript proxy object that is created by the Ajax.NET Pro library for you to use. Every proxy object that is created is created with a couple of overloads. An *overload* is a variation of a method using the same method name but with a unique set of parameters known as a *signature*. The signature is defined as the order and types of the parameters the method accepts. The proxy object `Chapter7_ImageSwitcher.ChangeMe()` gets created with the following signatures.

- ❑ `Chapter7_ImageSwitcher.ChangeImage(string, string, string)`
- ❑ `Chapter7_ImageSwitcher.ChangeImage(string, string, string, callback)`
- ❑ `Chapter7_ImageSwitcher.ChangeImage(string, string, string, callback, context)`

Notice the last two parameters, `callback` and `context`. `callback` is the name of the method that you want the response to be sent to. In this example, you would have a problem if all you could work with in the callback method was the response from the server. You'd have a problem because you need to set the value of the image tag, and you wouldn't know what that image tag was. So, Ajax.NET Pro has a last parameter called `context`. Whatever object you pass in as the `context` parameter will be returned in the response object as its `context` property. Remember, the `response` object has five properties, `value`, `error`, `request`, `extend`, and `context`. Now you see where the `context` is helpful. The `context` basically gets a free ride from your execution point (where you ask for the server method to be called) into your callback method.

It is common to name the callback function the same as the original function name, with the `_Callback` appended to it. Take a look at this callback function.

```
function ChangeMe_Callback(response) {
    if(response.error != null)
        alert(response.error.name + ' :: ' + response.error.description);
    else
        response.context.src = response.value;
}
```

Notice that the callback method takes only one parameter. It's common to call this parameter `response`, or `res`. The `response` object's context parameter is populated with the image tag because you sent that tag in as the context when you executed this line.

```
Chapter7_ImageSwitcherCallback.ChangeImage
         (target.src, onName, offName, ChangeMe_Callback,target);
```

In the JavaScript, you can reference the `response.context` as if it's the image. You set its source equal to the value that was returned from the server. To get the value returned from the server, you look in the `response.value` property. The entire HTML + JavaScript now should look like this.

Client Side — Chapter7/ImageSwitcherCallback.aspx

```
<script type="text/Javascript" language="Javascript">
<!--
function ChangeMe(target, leftName, rightName) {
    alert(response.value);
    Chapter7_ImageSwitcherCallback.ChangeImage(target.src, leftName,
rightName,ChangeMe_Callback,target);
}
function ChangeMe_Callback(response) {
    if(response.error != null)
        alert(response.error.name + ' :: ' + response.error.description);
    else
        response.context.src = response.value;
}
//-->
</script>

<div style="DISPLAY:none;VISIBILITY:hidden">
  <!-- preload images -->
  <img src="images/ArrowLeft.gif" border="0" runat="server" ID="ImgPreload1">
  <img src="images/ArrowRight.gif" border="0" runat="server" ID="ImgPreload2">
</div>
<img onclick="ChangeMe(this,'ArrowLeft','ArrowRight')" src="images/ArrowLeft.gif"
border="0">
```

In the preceding code, you're now executing the image changer in an asynchronous manner. This means that you don't have to wait for the server to return a response. The code can continue to execute and do other functions. When the server does return a response, the Ajax.NET Pro library will receive that response and execute the callback method, sending in the response as its object. This is a much more fluid style of programming, and you'll enjoy the benefits of not having your code bottlenecked because of a long-running process on the server.

Ajax.NET Pro Request Events — Keeping Your Users Updated

The `AjaxPro.Request` object has several placeholder events that you can set up JavaScript functions against. These event placeholders are `onLoading`, `onError`, `onTimeout`, and `onStateChanged`. I call them placeholders because they're null unless you assign a function to them. In this section, we're going to look at the `onLoading` event and how it can be used to let your users know that something is happening in the background. For example, the user might have just clicked a button that loads a large set of data and takes 5 seconds to execute. During this 5 seconds (which seems like forever) if nothing updates to let the user know something is happening, they might get antsy and press the button again, and again, and again — which we know just further delays the problem.

The following example uses the `onLoading` event from the class object that was registered using the `Utility.RegisterTypeForAjax` call (probably in your page load event). This `onLoading` event is called by the Ajax.NET Pro library before and after all calls are made to `AjaxMethod`s on your registered class. The code-behind page for this example has a simple method on it called `WaitXSeconds`. Its job is to simply wait for a given number of seconds and then return true. In a real application, this might be your database call or some other routine that might take a while. While this code is working, you can let your users know the code is busy with a little DHTML. When the text button is clicked, a red "Loading . . ." box will show in the top-left corner of the window. When the method is done, the "Loading . . ." box is hidden.

Chapter7_Onloading.aspx.cs Example

```
protected void Page_Load(object sender, EventArgs e)
{
    Utility.RegisterTypeForAjax(typeof(Chapter7_OnLoading));
}
[AjaxPro.AjaxMethod()]
public string WaitXSeconds(int SecondsToWait)
{
    System.Threading.Thread.Sleep(SecondsToWait*1000);
    return string.Format("{0} seconds have passed",SecondsToWait.ToString());;
}
```

The `onLoading` event in this example will get assigned to a function that you create. The `onLoading` function takes a single parameter, which is `true` or `false`, and lets you know if the `onLoading` function is being called at the beginning of the request (`true`) or at the end of the request (`false`). You use this parameter to set the visible property of an absolute positioned `div` tag named `"loading"`. When it's `true,` you set the visibility to `visible`, and when it's `false` (the end), you set its value to `hidden`.

Chapter7_Onloading.aspx Example

```
<form id="form1" runat="server">
<div id="loading"
style="visibility:hidden;position:absolute;top:0px;left:0px;background-
color:Red;color:White;">Loading...</div>
<div>
    <A href="#" onclick="javascript:ShowLoading(4);void(0);">Click Here</A>
    to run a 4 second process, and let the user know with a "Loading..."
    tag in the top left corner of the window.
</div>
</form>
<script language="javascript" type="text/javascript">
    Chapter7_OnLoading.onLoading = function(currentVis) {
        var loadingDiv = document.getElementById("loading");
        if(loadingDiv != null) {
            loadingDiv.style.visibility = currentVis ? "visible" : "hidden";
        }
    }
    function ShowLoading(seconds) {
        Chapter7_OnLoading.WaitXSeconds(seconds,WaitXSeconds_Callback);
    }
    function WaitXSeconds_Callback(response) {
        alert(response.value);
    }
</script>
```

A nice feature of the way the `onLoading` event works is that it's tied to the class object. So, if your registered class (in this case the `Page` class) has many `AjaxMethod`s on it, the `onLoading` event will be fired at the beginning and the end of every method that is called. And remember, you had to program this only once!

Errors, Errors, Errors. They Happen, You Trap 'em.

So far you've covered the `response`'s `value`, `request`, and `context` properties. Last, and maybe most important, is the `error` property. When Ajax.NET Pro processes your request, if all goes as planned and no error occurs, this property will have a null value. If any unhandled error is thrown during the Ajax.NET

Checking for Null Values

If the `error` property is null, there is no error, at least not one that was sent back with the response object. If the `value` property is null, the server didn't return any values, and if the `context` is null, no context was originally set up. But no matter what they mean, usually you account for these nulls with a simple `if()` statement in your code, as demonstrated in code earlier in the chapter using this line:

```
    if(response.error != null)
```

This line lets you know if an error was returned. In the sample, you coded that if there is an error, the error is shown to the user, and the value of the image source tag is not set. But you can choose to handle the error however you want.

Pro request, the string value of that error is returned. The `response.error` object has three properties — name, description, and number.

Property	Description
`response.error.name`	The name of the error, usually the namespace of the error that was thrown.
`response.error.description`	This is the equivalent of the `Exception.Message` in the .NET Framework.
`response.error.number`	If the exception thrown has an error number, it is populated here.

Notice that the last code block in the preceding section contains code to check the `response.error` property to see if it is null. If it is found not to be null, then you know you have an error, and you display the error message to the user. This will probably not be what you want to do in your real-world application. You would probably check the error number, and then handle the error appropriately, according to what caused the error.

Using the Ajax.NET Pro Library — Looking under the Hood

So, in summary, preparing your application to use the Ajax.NET Pro library requires the following steps:

1. First, you have to set up your ASP.NET application to be Ajax.NET Pro–enabled. This is done with the reference being set to the `AjaxPro.dll` assembly.
2. Second, you must modify your `Web.Config` file to register the `AjaxPro.AjaxHandlerFactory` to process all the `AjaxPro/*.ashx` requests.

3. Now that your application is Ajax.NET Pro–enabled, you're ready to build some Ajax functionality into your pages. On the page level, register your `page` class with the `AjaxPro.Utility.RegisterTypeForAjax()` method.
4. Then decorate your methods with the `AjaxPro.AjaxMethod()` attribute.
5. Finally, the Ajax.NET Pro library will create a JavaScript proxy object for you that follows the `ClassName.MethodName()` naming convention. Just add a little client UI and JavaScript to activate your server response, and your application is running on Ajax fuel.

You also now know how to trap errors, check for those errors, and check for null values.

However, although you now know the steps to make this all happen, you may have some questions about what's really happening under the hood to make this functionality work. The next few sections answer some of the common questions you may have about how all this functionality is really working.

When Is the Proxy JavaScript Created?

Remember preparing your page to be enabled with the Ajax.NET Pro library? This involved two steps. The first was registering your `page` class, with a line like the following:

```
AjaxPro.Utility.RegisterTypeForAjax(typeof(Chapter7_ImageSwitcher));
```

The second was attributing your public method with the `AjaxPro.AjaxMethod()` attribute:

```
[AjaPro.AjaxMethod()]
```

If you set a breakpoint on the `RegisterTypeForAjax` call and walk through the code, you'll find that this part of the library is doing something very simple. It's creating JavaScript tags that will be inserted at the top of your page with source values that point back to the `AjaxPro/*.ashx` page handler. When you set up your `Web.Config` file, you registered the page handler so that these types of calls are processed by the Ajax.NET Pro library. So, the entire job of the `RegisterTypeForAjax` method is to write `<script>` tags to the page. If you view source on the ASPX page from the browser, look towards the top of the page scripts that match this format.

```
<script type="text/javascript" src="/ajaxpro/prototype.ashx"></script>
```

```
<script type="text/javascript" src="/ajaxpro/core.ashx"></script>
```

```
<script type="text/javascript" src="/ajaxpro/converter.ashx"></script>
```

```
<script type="text/javascript" src="/ajaxpro/Chapter7_BuildHtmlTable,App_Web_it-
_kzny.ashx"></script>
```

The first three tags point to `/AjaxPro/common.ashx`, `core.ashx`, and `converter.ashx` files. These are JavaScript files that contain some general housekeeping helpers for Ajax.NET Pro (you'll look at this in the next chapter).

The fourth script source has a little more interesting name. The `RegisterTypeForAjax` created this script source, and the format is:

```
ClassName,AssemblyName.ashx
```

But this file doesn't exist — how does it get processed? Remember the `/AjaxPro/*.ashx` mapping to the Ajax page handler in the `Web.Config`? Because of that, any requested file path in the `/AjaxPro/` directory that ends with `.ashx` will be processed by the Ajax.NET Pro library. This URL is parsed, and from the named assembly that is found and the named class, the library is able to create the JavaScript proxy objects.

How does it know what proxy objects to create? It knows by using something in the .NET Framework called reflection. *Reflection* allows you to use code to inspect other code. So, the Ajax.NET Pro library creates a page class instance, in the example, a `Chapter7_ImageSwitcher.aspx` page, and then inspects that page class for any methods that are marked with the `AjaxPro.AjaxMethod()` attribute. Any methods that are found will automatically have JavaScript proxy objects created for them. These proxy objects are generated on demand and are written as the source of the `AjaxPro/Chapter7_ImageSwitcher, App_Web_it-kzny.ashx` request.

Why the funky assembly name `App Web it-kzny`? This is a framework-generated assembly where the page class `Chapter7_ImageSwitcher` lives. The .NET framework uses these unique names to keep separate versions of the page when it shadow compiles them, so it's helpful to the framework to have a set of random characters added to the assembly name. These unique assembly names are generated by the framework, and the Ajax.NET Pro library is smart enough to figure this out for you. So luckily, you don't ever have to know or care where the page class lives. The `RegisterTypeForAjax` call does this for you.

What Does the JavaScript Do?

When the browser requests a URL, the server (in this case your ASP.NET application server) returns to it a bunch of HTML. You can easily see this HTML with a view source in your browser. As soon as this HTML is received, it has to be parsed. You don't ever see the raw HTML (unless you do a view source) because the browser actually renders that HTML into a web page. When the `<script>` tag that has a source attribute is loaded, the browser will make another request back to the server to get that data. It's important to realize that these are loaded as separate requests. The same thing is done for images; that is they are loaded in separate synchronous requests one right after the other.

What Happens on the Server after the Proxy JavaScript Has Been Fired?

When the JavaScript source is called on the server, based on the URL, you know that these script files are going to be processed by the Ajax.NET Pro library. The JavaScript that is created, creates the proxy objects that enable you to call the `PageClassName.MethodName()` JavaScript function calls. These are called proxy objects because they don't actually do any of the work; they simply proxy the call through an Ajax protocol to the server.

How Is the Method in the Code-Behind Actually Executed and How Is the Page Actually Created?

You'll see how this is done in detail in the next chapter. Essentially, this is where the reflection is done, the page class is created in code, and then the method is executed with the parameters that were sent in from the JavaScript call.

What Is Really Being Sent Back to the Client

In an earlier example, you read about a string being sent back to the client. Remember `response.value`? `response.value` can hold any value or object, or even stay null. When a simple type is sent back (such as `String` or `Integer`), the actual value is what is stored. When something like a `DataSet` is returned, because there is no such thing as a `DataSet` object in JavaScript, the `DataSet` is converted into a JavaScript Object Notation (JSON) object (JSON is discussed in Chapter 5) that holds much of the same data that a `DataSet` holds. But understand, you're not dealing with a full ADO.NET `DataSet`, you're dealing with a JSON object that has simply been populated with data that matches the data in the `DataSet` you were expecting. This was shown earlier in the chapter when you dynamically created an HTML table from an Ajax.NET Pro call that returned a `DataTable`. Once you know that JSON objects can be held in the `response.value` property, the question to answer is this — "Can I return my own custom types?" Absolutely, you can. In fact if your custom type is a simple type, you can simply mark you class with the `[Serializable()]` attribute, and the Ajax.NET Pro library will do the rest. You'll look more into this functionality when you look at extending the Ajax.NET Framework in Chapter 8.

Summary

In this chapter:

- ❑ You were given an introduction to the Ajax.NET Pro library.
- ❑ You were able to download the code from `www.BeginningAjax.com` and reference the Ajax.NET Pro library.
- ❑ And once you had the project set up, you were able to create a client/server conversation, and updated your web page with the server content Ajax style.

In Chapter 8, you'll look under the hood of the Ajax.NET library and see more about how the magic is happening. You'll walk through the code that comes with the library and show what's happening when and why. Remember, the idea of a wrapper library is to make life easier, which hopefully the simplicity of the code in this chapter has shown. Chapter 8 is a little more advanced. If you're interested in extending the library with custom objects, then Chapter 8 will be a good read for you.

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.