Microsoft

INTRODUCING MICROSOFT® ASP.NET AJAX

Dino Esposito

SQL
Solid Quality Learning

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2007924643

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 2 1 0 9 8 7

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Acquisitions Editor: Ben Ryan
Developmental Editor: Lynn Finnel
Editorial Production: Abshier House
Copy Editor: Roger LeBlanc
Technical Reviewer: Kenn Scribner
Indexer: Sharon Hilgenberg

Body Part No. X13-68385

Contents at a Glance

Table of Contents

Introduction

AJAX stands for "Asynchronous JavaScript and XML," and it's a sort of blanket term coined in 2005 to indicate rich, highly interactive, and responsive Web applications that do a lot of work on the client and place out-of-band calls to the server. An *out-of-band* call is a server request that results in a page update rather than a page replacement. The net effect is that an AJAX Web application tends to look like a classic desktop Microsoft Windows application and has advanced features such as drag-and-drop and asynchronous tasks, a strongly responsive and nonflickering user interface, and far less user frustration.

ASP.NET AJAX Extensions is a significant extension to the ASP.NET platform that makes AJAX-style functionalities possible and effective. ASP.NET AJAX Extensions is designed to be part of ASP.NET and, therefore, seamlessly integrate with the existing platform and application model.

Architecturally speaking, the ASP.NET AJAX framework is made of two distinct elements: a client script library and a set of server extensions. The client script library is entirely written in JavaScript and, therefore, works with any modern browser. Server extensions are fully integrated with ASP.NET server-based services and controls. As a result, developers can write rich Web pages using nearly the same approach they know from developing classic ASP.NET server-based pages.

Most ASP.NET AJAX developers are former ASP.NET developers and, as such, are familiar with the server-side development model based on controls. The server-centric programming model is the next big step in the evolution of the ASP.NET programming model. ASP.NET AJAX server controls are great, especially if you don't feel confident enough to create AJAX client scripts manually.

This book provides an overview of the ASP.NET AJAX framework with numerous examples to familiarize you with a variety of techniques and tools.

AJAX is a real breakthrough for ASP.NET developers and professionals. It makes cross-browser programming a reality and enables desktop-like functionalities over the Web.

Who This Book Is For

The book is recommended for virtually any ASP.NET developer and professional. As mentioned, ASP.NET AJAX is the next big thing in the ASP.NET evolution and follows a key industry trend—the AJAX model. In addition, ASP.NET AJAX goes beyond the classic AJAX model, pushing a framework that spans the client and server to provide an end-to-end solution for Web applications. As far as the Microsoft Web platform is concerned, ASP.NET AJAX Extensions weds rich functions with wide reach—an old dream of Web professionals that comes true. At last.

If you're a Web professional developing for Microsoft-based Web technologies, AJAX is your next big opportunity to seize. This book is your starting point. And even a bit more.

How This Book Is Organized

The book is divided into three parts: an overview of the platform and its building blocks, techniques to effectively enhance existing sites, and client-centric development. In the first part, you'll learn the basics of the AJAX model and the extensions made to the JavaScript language to back it. The second part is dedicated to the elements in the framework that you use to add new capabilities to existing server controls and to transform existing classic ASP.NET pages into full-fledged AJAX pages. Finally, the third part covers tools and techniques that express the real power of AJAX applications—out-of-band calls to server code.

System Requirements

You'll need the following hardware and software to build and run the code samples for this book:

- Microsoft Windows Vista, Microsoft Windows XP with Service Pack 2, Microsoft Windows Server 2003 with Service Pack 1, or Microsoft Windows 2000 with Service Pack 4
- Microsoft Visual Studio 2005 Standard Edition or Microsoft Visual Studio 2005 Professional Edition
- Microsoft SQL Server 2005 Express (included with Visual Studio 2005) or Microsoft SQL Server 2005
- 600-MHz Pentium or compatible processor (1-GHz Pentium recommended)
- 192 MB of RAM (256 MB or more recommended)
- Video monitor (800 x 600 or higher resolution) with at least 256 colors (1024 x 768 High Color 16-bit recommended)
- Microsoft mouse or compatible pointing device

Configuring SQL Server 2005 Express Edition

Some chapters of this book require that you have access to SQL Server 2005 Express Edition (or SQL Server 2005) to create and use the Northwind Traders database. If you are using SQL Server 2005 Express Edition, follow these steps to grant access to the user account that you will be using to perform the exercises in this book:

1. Log on to Windows on your computer by using an account with administrator privileges.

2. On the Windows Start menu, click All Programs, click Accessories, and then click Command Prompt to open a command prompt window.
3. In the command prompt window, type the following case-sensitive command:

   ```
   sqlcmd -S YourServer\SQLExpress -E
   ```

   Replace *YourServer* with the name of your computer.

   You can find the name of your computer by running the *hostname* command in the command prompt window before running the *sqlcmd* command.
4. At the 1> prompt, type the following command, including the square brackets, and then press Enter:

   ```
   sp_grantlogin [YourServer\UserName]
   ```

   Replace *YourServer* with the name of your computer, and replace *UserName* with the name of the user account you will be using.
5. At the 2> prompt, type the following command and then press Enter:

   ```
   go
   ```

   If you see an error message, make sure that you have typed the `sp_grantlogin` command correctly, including the square brackets.
6. At the 1> prompt, type the following command, including the square brackets, and then press Enter:

   ```
   sp_addsrvrolemember [YourServer\UserName], dbcreator
   ```
7. At the 2> prompt, type the following command and then press Enter:

   ```
   go
   ```

   If you see an error message, make sure that you have typed the `sp_addsrvrolemember` command correctly, including the square brackets.
8. At the 1> prompt, type the following command and then press Enter:

   ```
   exit
   ```
9. Close the command prompt window.
10. Log out of the administrator account.

The Northwind Traders database no longer ships with SQL Server 2005 (either version), so you'll need to download that separately. You can download the necessary installation scripts for the Northwind database from *http://www.microsoft.com/downloads/details.aspx?FamilyId=06616212-0356-46A0-8DA2-EEBC53A68034&displaylang=en*. Installation instructions are included on the download page.

Code Samples

The downloadable code includes projects for most chapters that cover the code snippets and examples referenced in the chapter. All the code samples discussed in this book can be downloaded from the book's companion content page at the following address:

http://www.microsoft.com/mspress/companion/9780735624139

Support for This Book

Every effort has been made to ensure the accuracy of this book and the companion content. As corrections or changes are collected, they will be added to a Microsoft Knowledge Base article.

Microsoft Press provides support for books and companion content at the following Web site:

http://www.microsoft.com/learning/support/books/

Questions and Comments

If you have comments, questions, or ideas regarding the book or the companion content, or questions that are not answered by visiting the site just mentioned, please send them to Microsoft Press via e-mail to

mspinput@microsoft.com

Or via postal mail to

Microsoft Press
Attn: *Introducing Microsoft ASP.NET AJAX* Editor
One Microsoft Way
Redmond, WA 98052-6399

Please note that Microsoft software product support is not offered through the above addresses.

Part I
ASP.NET AJAX Building Blocks

In this part:

Chapter 1

The AJAX Revolution

In this chapter:

Gone are the days when a Web application could be architected and implemented as a collection of related and linked pages. The advent of the so-called *AJAX* model is radically modifying the user's perception of a Web application, and it is subsequently forcing developers to apply newer and richer models to the planning and implementation of modern Web applications. But what is the AJAX model, anyway?

AJAX is a relatively new acronym that stands for *Asynchronous JavaScript and XML*. It is a sort of blanket term used to describe highly interactive and responsive Web applications. What's the point here? Weren't Web applications created about a decade ago specifically to be "interactive," "responsive," and deployed over a unique tool called the browser? So what's new today?

The incredible success of the Internet has whetted people's appetite for Web-related technology beyond imagination. Over the years, the users' demand for ever more powerful and Web-exposed applications and services led architects and developers to incorporate more and more features into the server platform and client browser. As a result, the traditional pattern of Web applications is becoming less adequate every day. A radical change in the design and programming model cannot be further delayed.

At the current state of the art, the industry needs more than just an improved and more powerful platform devised along the traditional guidelines and principles of Web applications—a true paradigm shift is required. AJAX is the incarnation of a new paradigm for the next generation of Web applications that is probably destined to last for at least the next decade.

From a more developer-oriented perspective, *AJAX* collectively refers to a set of development components, tools, and techniques for creating highly interactive Web applications that give users a better experience. According to the AJAX paradigm, Web applications work by exchanging data rather than pages with the Web server. From a user perspective, this means that faster roundtrips occur and, more importantly, page loading and refresh is significantly reduced. As a result, a Web application tends to look like a classic desktop Microsoft Windows

application and has advanced features such as drag-and-drop and asynchronous tasks, a strongly responsive and nonflickering user interface, and other such features that minimize user frustration, provide timely feedback about what's going on, and deliver great mashed-up content. (Hold on! This doesn't mean AJAX Web applications *are the same as* desktop applications; they simply allow for a few more desktop-like features.)

AJAX is the philosophy that has inspired a new generation of components and frameworks, each designed to target a particular platform, provide a given set of capabilities, and possibly integrate seamlessly with existing frameworks. Microsoft ASP.NET AJAX Extensions is the AJAX addition to the ASP.NET 2.0 platform. In the next major release of the .NET Framework platform ASP.NET AJAX Extensions will officially fuse to ASP.NET and the rest of the Microsoft Web platform and application model. The next release of Microsoft Visual Studio (code-named "Orcas") will also integrate ad hoc design-time support for AJAX-specific features.

In this chapter, I'll dig deeper into the motivation for and driving force behind AJAX and then review the basic system requirements common to all AJAX frameworks.

The Paradigm Shift

We are all witnessing and contributing to an interesting and unique phenomenon—the Web is undergoing an epochal change right before our eyes as a result of our actions. As drastic as it might sound, the Web revolutionized the concept of an application. Only eight years ago, the majority of developers considered an application far too serious a thing to reduce it to an unordered mix of script and markup code. In the late 1990s, the cost of an application was sweat, blood, tears, and endless debugging sessions. According to the common and semi-serious perception there was neither honor nor fame for the "real" programmer in writing Web applications.

Note In the late 1990s, though, a number of Web sites were designed and built. Some of them grew incredibly in the following years to become pillars of today's world economy and even changed the way we do ordinary things. Want some examples? Google, Amazon, eBay. Nonetheless, a decade ago the guys building these and other applications were sort of avant-garde developers, perhaps even just smart and game amateurs.

Since then, the Web has evolved significantly. And although 10 years of Web evolution has resulted in the building of a thick layer of abstraction on the server side, it hasn't changed the basic infrastructure—HTTP protocol and pages.

The original infrastructure—one that was simple, ubiquitous, and effective—was the chief factor for the rapid success of the Web model of applications. The next generation of Web applications will still be based on the HTTP protocol and pages. However, the contents of pages and the capabilities of the server-side machinery will change to provide a significantly richer user experience—as rich as that of classic desktop Windows applications.

Note As we'll see in greater detail in Chapter 8, "Building AJAX Applications with ASP.NET," AJAX applications have a number of plusses but also a few drawbacks. Overall, choosing an AJAX application rather than a classic Web application is simply a matter of weighing the trade-offs. An AJAX application certainly gives users continuous feedback and never appears held up by some remote operation. On the other hand, AJAX applications are not entirely like desktop applications, and their capabilities in terms of graphics, multimedia, and hardware control are not as powerful as in a regular (smart) client. In the end, AJAX applications are just one very special breed of a Web application; as such, they might require some code refactoring to deliver the expected performance and results.

Classic Web Applications

Today Web applications work by submitting user-filled forms to the Web server and displaying the markup returned by the Web server. The browser-to-server communication employs the classic HTTP protocol. As is widely known, the HTTP protocol is stateless, which means that each request is not related to the next and no state is automatically maintained. (The state objects we all know and use in, say, ASP.NET are nothing more than an abstraction provided by the server programming environment.)

Communication between the browser and the Web server occurs through "forms." From a user's perspective, the transition occurs through "pages." Each user action that originates a new request for the server results in a brand new page (or a revamped version of the current page) being downloaded and displayed.

Let's briefly explore this model a bit further to pinpoint its drawbacks and bring to the surface the reasons why a new model is needed.

Send Input via Forms

Based on the URL typed in the address bar, the browser displays a page to the user. The page is ultimately made of HTML markup and contains one or more HTML forms. The user enters some data, and then instructs the browser to submit the form to an action URL.

Using the local Domain Name System (DNS) resolver in the operating system, the browser resolves the specified URL to an IP address and opens a socket. An HTTP packet travels over the wire to the given destination. The packet includes the form and all its fields. The request is captured by the Web server and typically forwarded to an internal module for further processing. At the end of the process, an HTTP response packet is prepared and the return value for the browser is inserted in the body.

Get Output through Pages

When a request is made for, say, an *.aspx* resource, the Web server passes it on to ASP.NET for processing and receives the resulting HTML markup in return. The generated markup

comprises all the tags of a classic HTML page, including <*html*>, <*body*>, and <*form*>. The page source is embedded in the HTTP response and tagged with a Multipurpose Internet Mail Extensions (MIME) type to instruct the browser how to handle it. The browser looks at the MIME type and decides what to do.

If the response contains an HTML page, the browser replaces the current contents entirely with the new chunk of markup. While the request is being processed on the server, the "old" page is frozen but still displayed to the client user. As soon as the "new" page is downloaded, the browser clears the display and renders the page.

Capabilities and Drawbacks

This model was just fine in the beginning of the Web age when pages contained little more than formatted text, hyperlinks, and some images. The success of the Web has prompted users to ask for increasingly more powerful features, and it has led developers and designers to create more sophisticated services and graphics. As an example, consider advertising. Today, most pages—and often even very simple pages, such as blog pages—include ad rotators that download quite a bit of stuff on the client.

As a result, pages are heavy and cumbersome—even though we still insist on calling them "rich" pages. Regardless of whether they're rich or just cumbersome, these are the Web pages of today's applications. Nobody really believes that we're going to return to the scanty and spartan HTML pages of a decade ago.

Given the current architecture of Web applications, each user action requires a complete redraw of the page. Subsequently, richer and heavier pages render slowly and as a result produce a good deal of flickering. Projected to the whole set of pages in a large, portal-like application, this mechanism is just perfect for unleashing the frustrations of the poor end user.

Although a developer can build a page on the server using one of many flexible architectures (ASP.NET being one such example), from the client-side perspective Web pages were originally designed to be mostly static and unmodifiable. In the late 1990s, the introduction of Dynamic HTML first, and the advent of a World Wide Web Consortium (W3C) standard for the page document object model later, changed this basic fact. Today, the browser exposes the whole content of a displayed page through a read/write object model. In this way, the page can be modified to incorporate changes made entirely on the client-side to react to user inputs. (As we'll see, this is a key factor for AJAX and ASP.NET AJAX solutions.)

Dynamic HTML is a quantum leap, but alone it is not enough to further the evolution of the Web.

AJAX-Based Web Applications

To minimize the impact of page redraws, primitive forms of scripted remote procedure calls (RPC) appeared around 1997. Microsoft, in particular, pioneered this field with a technology called Remote Scripting (RS).

RS employed a Java applet to pull in data from a remote Active Server Pages (ASP)-based URL. The URL exposed a contracted programming interface through a target ASP page and serialized data back and forth through plain strings. On the client, a little JavaScript framework received data and invoked a user-defined callback to update the user interface via Dynamic HTML or similar techniques. RS worked on both Internet Explorer 4.0 and Netscape Navigator 4.0 and older versions.

Later on, Microsoft replaced the Java applet with a Component Object Model (COM) object named *XMLHttpRequest* and released most of the constraints on the programming interface exposed by the remote URL—for example, no more fixed ASP pages. At the same time, community efforts produced a range of similar frameworks aimed at taking RS to the next level and building a broader reach for solutions. The Java applet disappeared and was replaced by the *XMLHttpRequest* object.

What Is AJAX, Anyway?

The term *AJAX* was coined in 2005. It originated in the Java community and was used in reference to a range of related technologies for implementing forms of remote scripting. Today, any form of remote scripting is generally tagged with the AJAX prefix. Modern AJAX-based solutions for the Windows platform are based on the *XMLHttpRequest* object. Google Maps and Gmail are the two most popular Web applications designed according to AJAX patterns and techniques. For AJAX, these were certainly the *killer applications* that established its usefulness and showed its potential.

Two combined elements make an AJAX application live and thrive. On one hand, you need to serve users fresh data retrieved on the server. On the other hand, you need to integrate new data in the existing page without a full page refresh.

Browsers generally place a new request when an HTML form is submitted either via client-side script or through a user action such as a button click. When the response is ready, the browser replaces the old page with the new one. Figure 1-1 illustrates graphically this traditional approach.

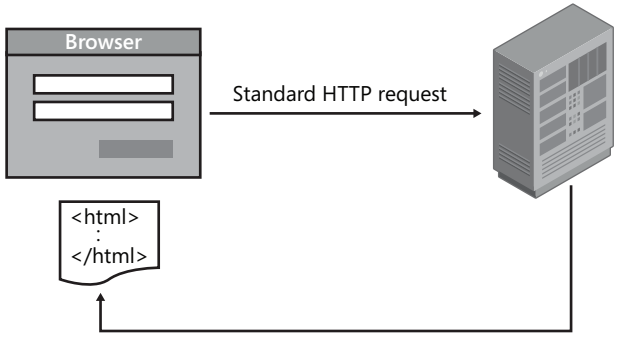

Figure 1-1 Browsers submit an HTML form and receive a new page to display.

The chief factor that enables remote scripting is the ability to issue out-of-band HTTP requests. In this context, an out-of-band call indicates an HTTP request placed using a component that is different from the browser's built-in module that handles the HTML form submission (that is, outside the traditional mechanism you see in Figure 1-1). The out-of-band call is triggered via script by an HTML page event and is served by a proxy component. In the most recent AJAX solutions, the proxy component is based on the *XMLHttpRequest* object; the proxy component was a Java applet in the very first implementation of RS.

Update Pages via Script

The proxy component (for example, the *XMLHttpRequest* object) sends a regular HTTP request and waits, either synchronously or asynchronously, for it to be fully served. When the response data is ready, the proxy invokes a user-defined JavaScript callback to refresh any portion of the page that needs updating. Figure 1-2 provides a graphical overview of the model.

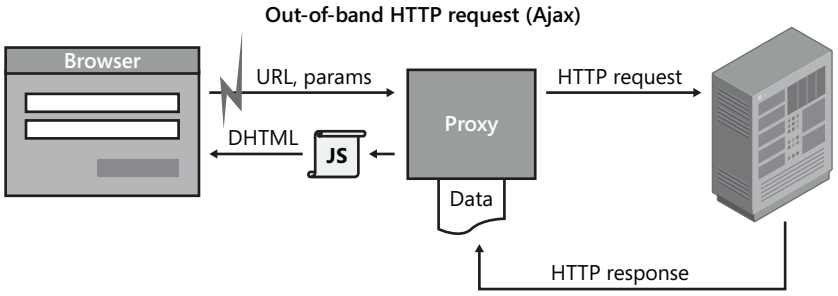

Figure 1-2 Out-of-band calls are sent through a proxy component, and a JavaScript callback is used to update any portion of the page affected by returned data.

All browsers know how to replace an old page with a new page; until a few years ago, though, not all of them provided an object model to represent the current contents of the page. (Today, I can hardly mention a single modern, commercially available browser that doesn't expose a read/write page DOM.) For browsers that supply an updatable object model for HTML pages, the JavaScript callback function can refresh specific portions of the old page, thus making them look updated, without a full reload.

The Document Object Model

The page Document Object Model (DOM) is the specification that defines a platform- and language-neutral interface for accessing and updating the contents, structure, and style of HTML and XML documents. A recognized standard ratified by the W3C committee, the DOM is now supported by virtually all browsers. The DOM provides a standard set of objects for representing the constituent elements of HTML and XML documents. All together, these objects form a standard interface for accessing and manipulating child elements of HTML pages and, more in general, XML documents.

Note that although the first working frameworks for remote scripting date back to a decade ago, the limited support browsers have had for dynamic changes in displayed documents slowed down the adoption of such technologies in the industry. Until now.

The Role of Rich Browsers

As shown in Figure 1-2, the AJAX model has two key requirements as far as browsers are concerned: a proxy component and an updatable page DOM. For quite a long time, only high-end browsers (also known as rich, up-level browsers) provided support for both features. In the past few years, only companies that could exercise strict control over the capabilities of the client browsers were able to choose the AJAX model for their sites. For too long, a *rich* browser also has meant a browser with too limited *reach*. For too long, using such a browser was definitely a bad choice for most businesses because the limited reach excluded significant portions of the customer base.

Rich vs. Reach

Perhaps due to a rare and likely unrepeatable astral conjunction, today more than 90 percent of browsers available on the market happen to have built-in support for the advanced capabilities that the AJAX model requires. Internet Explorer since version 5.0, Firefox, Netscape from version 6 and onward, Safari 1.2, Opera starting with version 8.0, and a variety of mobile devices are all browsers that fully support the AJAX programming model.

For the very first time, a *rich browser* is not synonymous with a *limited reach browser*. Finally, you don't have to choose a particular browser to enjoy advanced, programming-rich features. Designing highly interactive Web applications that implement remote scripting techniques is no longer an impossible dream to chase but a concrete opportunity to seize—whatever browsers you and your clients use.

Each platform and each vendor might have a particular framework and tool set to offer, but this doesn't change the basic fact that living the AJAX lifestyle is now possible with 90 percent of the browsers available today. It's a real breakthrough, and it is now possible to build and distribute applications that were not possible before.

Required Capabilities

Exactly what are the capabilities required of a browser to run AJAX functionalities? As mentioned, a browser needs to provide two key capabilities: a proxy mechanism to make client code able to place out-of-band HTTP calls, and an updatable DOM.

There's a W3C ratified standard for the updatable DOM. A W3C standard for the proxy component is currently being developed. It takes the form of the *XMLHttpRequest* object and is devised as an interface exposed by the browser to allow script code to perform HTTP client functionality, such as submitting form data or loading data from a remote Web site. The latest working draft is available at *http://www.w3.org/TR/XMLHttpRequest*.

In addition, browsers must support JavaScript and preferably cascading style sheets (CSS).

In the end, the AJAX lifestyle is possible and affordable for virtually every developer and nearly 90 percent of the Web audience, regardless of the platform. The tools required to make AJAX work are becoming as ubiquitous as HTML/XML parsers, HTTP listeners, and JavaScript

processors. To paraphrase the catch phrase of a popular advertising campaign, I'd say that "AJAX is now." And as far as the Windows and ASP.NET platforms are concerned, AJAX takes the form of Microsoft ASP.NET AJAX Extensions.

The AJAX Core Engine

AJAX is not a particular technology or product. It refers to a number of client features, and related development techniques, that make Web applications look like desktop applications. AJAX doesn't require any plug-in modules either and is not browser specific. Virtually any browser released in the past five years can serve as a great host for AJAX-based applications. AJAX development techniques revolve around one common software element—the *XMLHttpRequest* object. The availability of this object in the object model of most browsers is the key to the current ubiquity and success of AJAX applications. In addition to *XMLHttpRequest*, a second factor contributes to the wide success of AJAX—the availability of a rich document object model in virtually any browser.

Originally introduced with Internet Explorer 5.0, the *XMLHttpRequest* object is an internal object that the browser publishes to its scripting engine. In this way, the script code found in any client page—typically, JavaScript code—can invoke the object and take advantage of its functionality.

The *XMLHttpRequest* object allows script code to send HTTP requests and handle their response. Functionally speaking, and despite the XML in the name, the *XMLHttpRequest* object is nothing more than a tiny object designed to place HTTP calls via script in a non-browser-led way. When users click the submit button of a form, or perform any action that ends up invoking the *submit* method on the DOM's *form* object, the browser kicks in and takes full control of the subsequent HTTP request. From the user's perspective, the request is a black box whose only visible outcome is the new page being displayed. The client script code has no control over the placement and outcome of the request.

The *XMLHttpRequest* Object

Created by Microsoft and adopted soon thereafter by Mozilla, the *XMLHttpRequest* object is today fully supported by the majority of Web browsers. As you'll see in a moment, the implementation can significantly differ from one browser to the next, even though the top-level interface is nearly identical. For this reason, a W3C committee is at work with the goal of precisely documenting a minimum set of interoperable features based on existing implementations.

Note The *XMLHttpRequest* object originally shipped as a separate component with Internet Explorer 5.0 back in the spring of 1999. It is a native component of all Microsoft operating systems that have shipped since. In particular, you'll certainly find it installed on all machines that run Windows 2000, Windows XP, and newer operating systems.

The Internet Explorer Object

When the *XMLHttpRequest* object was first released, the Component Object Model (COM) was ruling the world at Microsoft. The extensibility model of products and applications was based on COM and implemented through COM components. In the late 1990s, the right and natural choice was to implement this new component as a reusable automation COM object, named *Microsoft.XmlHttp*.

Various versions of the same component (even with slightly different names) were released over the years, but all of them preserved the original component model—COM. Internet Explorer 6.0, for example, ships the *XMLHttpRequest* object in the form of a COM object. Where's the problem?

COM objects are external components that require explicit permission to run inside of a Web browser. In particular, to run the *XMLHttpRequest* object and subsequently enable any AJAX functionality built on top of it, at a minimum a client machine needs to accept ActiveX components marked safe for scripting. (See Figure 1-3.)

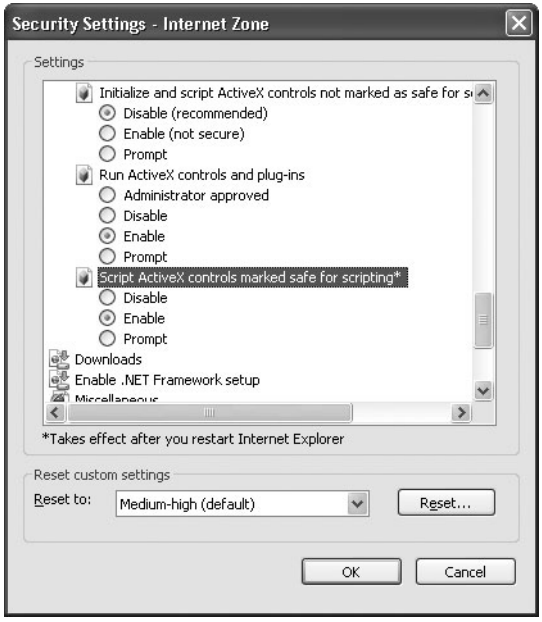

Figure 1-3 The property window used to change the security settings in Internet Explorer

The *XMLHttpRequest* object is certainly a safe component, but to enable it users need to lessen their security settings and accept any other component "declared" safe for scripting that is around the Web sites they visit.

Important The internal implementation of *XMLHttpRequest* is disjointed from the implementation of any AJAX-like frameworks, such as Microsoft ASP.NET AJAX. Under the hood, any framework ends up calling the object as exposed by, or available in, the browser.

The Mozilla Counterpart

Mozilla adopted *XMLHttpRequest* immediately after its first release with Internet Explorer 5.0. However, in Mozilla-equipped browsers the *XMLHttpRequest* object is part of the browser's object model and doesn't rely on external components. Put another way, a Mozilla browser such as Firefox publishes its own *XMLHttpRequest* object into the scripting engine and never uses the COM component, even when the COM component is installed on the client machine and is part of the operating system. Figure 1-4 shows the different models in Internet Explorer (up to version 6.0) and Mozilla browsers.

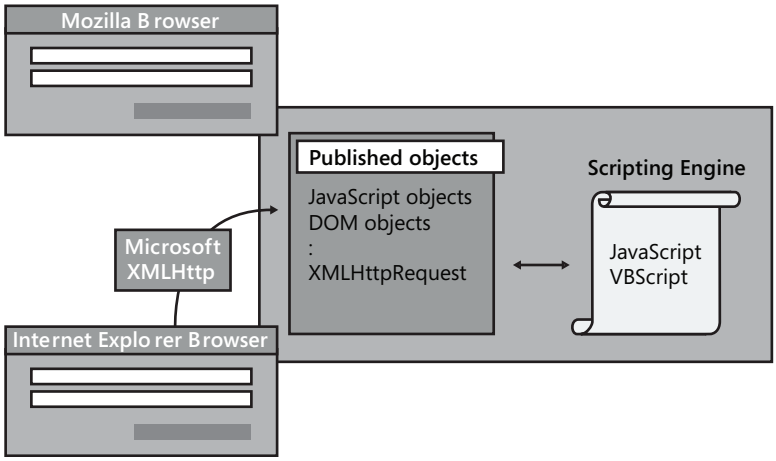

Figure 1-4 *XMLHttpRequest* is a scriptable component exposed by the browser in Mozilla and an external COM component in Internet Explorer (up to version 6.0)

As a result, in Mozilla browsers *XMLHttpRequest* looks like a native JavaScript object and can be instantiated through the classic *new* operator:

```
// The object name requires XML in capital letters
var proxy = new XMLHttpRequest();
```

When the browser is Internet Explorer, the *XMLHttpRequest* object is instantiated using the *ActiveXObject* wrapper, as shown here:

```
var proxy = new ActiveXObject("Microsoft.XmlHttp");
```

Generally, AJAX-style frameworks check the current browser and then decide about the route to take.

Needless to say, as implemented in Mozilla browsers the *XMLHttpRequest* functionality is somewhat safer, at least in the sense it doesn't require users to change their security settings for the browser.

XMLHttpRequest in Internet Explorer 7

Implemented as a COM component for historical reasons on Internet Explorer browsers, the *XMLHttpRequest* object has finally become a browser object with Internet Explorer 7.0. All potential security concerns are removed at the root, and AJAX frameworks can be updated to use the same syntax for creating the *XMLHttpRequest* object regardless of the browser:

```
var proxy = new XMLHttpRequest();
```

In addition, this change in Internet Explorer 7.0 completely decouples AJAX-like functionality in ASP.NET from an ActiveX-enabled environment.

An HTTP Object Model

I spent quite a few words on the *XMLHttpRequest* object and its expected behavior, but I still owe you a practical demonstration of the object's capabilities. In this section, I'll cover the members of the component, the actions it can perform, and details of the syntax.

As mentioned, the *XML* in the name of the component means little and in no way limits the capabilities of the component. In spite of the XML prefix, you can use the object as a true automation engine for executing and controlling HTTP requests, from client code generated by ASP.NET pages or the Windows shell, or Visual Basic 6.0 or C++ unmanaged applications. Using the *XMLHttpRequest* COM object from within .NET applications is nonsensical, as you can find similar functionality in the folds of the *System.Net* namespace in the .NET Framework.

Important If you're going to use Microsoft ASP.NET AJAX Extensions or any other AJAX-like framework for building your applications, you'll hardly hear about the *XMLHttpRequest* object, much less use it directly in your own code. ASP.NET AJAX Extensions completely encapsulates this object and shields page authors and application designers from it. You don't need to know about *XMLHttpRequest* to write great AJAX applications, no matter how complex and sophisticated they are. However, knowing the fundamentals of *XMLHttpRequest* can lead you to a better and more thorough understanding of the platform and to more effective diagnoses of problems.

Behavior and Capabilities

The *XMLHttpRequest* object is designed to perform one key operation: sending an HTTP request. The request can be sent either synchronously or asynchronously. The following listing shows the programming interface of the object as it results from the W3C working draft at the time of this writing:

```
interface XMLHttpRequest {
  function onreadystatechange;
  readonly unsigned short readyState;
  void open(string method, string url);
  void open(string method, string url, bool async);
  void open(string method, string url, bool async, string user);
  void open(string method, string url, bool async,
            string user, string pswd);
  void setRequestHeader(string header, string value);
  void send(string data);
  void send(Document data);
  void abort();
  string getAllResponseHeaders();
  string getResponseHeader(string header);
  string responseText;
  Document responseXML;
  unsigned short status;
  string statusText;
};
```

Using the component is a two-step operation. First, you open a channel to the URL and specify the method (GET, POST, or other) to use and whether you want the request to execute asynchronously. Next, you set any required header and send the request. If the request is a POST, you pass to the *send* method the body of the request.

The *send* method returns immediately in the case of an asynchronous operation. You write an *onreadystatechange* function to check the status of the current operation and, using that function, figure out when it is done.

Sending a Request

Most AJAX frameworks obtain an instance of the *XMLHttpRequest* object for the current browser using code that looks like the following:

```
var xmlRequest, e;
try {
  xmlRequest = new XMLHttpRequest();
}
catch(e) {
    try {
      xmlRequest = new ActiveXObject("Microsoft.XMLHTTP");
    }
    catch(e) {
    }
}
```

The code first tries to instantiate the internal *XMLHttpRequest* object and opts for the ActiveX object in the case of failure. As you can see, the creation of the object requires an exception to be caught when the browser is Internet Explorer 6.0 or any older versions. Such a code will work unchanged (and won't require any exception) in Internet Explorer 7.0.

Note Checking the browser's user agent and foregoing the exception is fine as well. However, ASP.NET AJAX Extensions uses the preceding code because it makes the overall library independent from details of user agent strings and browser details. In this way, you do "object detection" instead of "browser detection." The final result, though, is the same. The exception is fired only if the browser is Internet Explorer older than version 7.0 or any other browser that doesn't support AJAX functionalities. If you're building your own AJAX framework, you need to check the user agent only against Internet Explorer.

The *open* method prepares the channel for the request; no physical socket is created yet, though. To execute a POST statement, you need to add the proper content-type header. The Boolean argument indicates whether the operation is asynchronous:

```
xmlRequest.open("POST", url, false);
xmlRequest.setRequestHeader("Content-Type",
                            "application/x-www-form-urlencoded");
xmlRequest.send(postData);
```

The *send* method opens the socket and sends the packet. In the preceding code snippet, the method returns only when the response has been fully received.

An asynchronous request requires slightly different code:

```
xmlRequest.open("POST", url, true);
xmlRequest.onreadystatechange = CallbackComplete;
xmlRequest.setRequestHeader("Content-Type",
                            "application/x-www-form-urlencoded");
xmlRequest.send(postData);
```

The *CallbackComplete* element is a placeholder for a JavaScript function that retrieves and processes the response generated by the request.

Note that the function assigned to the *onreadystatechange* member will be invoked whenever *readyState* changes value. Possible values for the state are the integers ranging from 0 through 4, which mean "Uninitialized," "Open method called successfully," "Send method called successfully," "Receiving data," and "Response received," respectively. The *CallbackComplete* framework-specific function will generally check that state and proceed.

Receiving a Response

The response of the request is available in two formats: as raw text and as an XML document. The *responseText* property is empty if the state is 0 through 2—that is, no data has been received yet. When the state transitions to 3 (receiving data), the property contains the data received so far, interpreted using the character encoding specified in the response. If no character encoding was specified, it employs UTF-8.

The *responseXml* property is not available until the full response has been downloaded and successfully parsed to an XML document. If the body of the response is not XML or if the

parsing fails for any reason, the property returns null. It is important to note that the construction of the XML document takes place on the client once the raw HTTP response has been fully received.

Roll Your Own (Little) AJAX Framework

As mentioned, you don't need to use the straight *XMLHttpRequest* object in your AJAX-based application, regardless of the framework (for example, ASP.NET AJAX) you end up using. For completeness, though, let's briefly review the steps required to use the object in a sample ASP.NET 2.0 page. The same code can be used with ASP.NET 1.x as well. The following code represents the minimal engine you need for building homemade AJAX solutions.

Note Although a homemade AJAX framework might not be recommended, it's not impossible to write. The fact that more than 100 AJAX frameworks have been counted just demonstrates that writing AJAX homemade solutions is not a mission-impossible task. Personally, I would consider it only as a way to enrich existing applications with quick and dirty AJAX functionality limited to placing remote, non-browser-led calls.

Executing an Out-of-Band Call from an ASP.NET Page

Web pages that shoot out-of-band calls need to have one or more trigger events that, when properly handled with a piece of JavaScript code, place the request via the *XMLHttpRequest* object. Trigger events can only be HTML events captured by the browser's DOM implementation.

The JavaScript code should initiate and control the remote URL invocation, as shown in the following code:

```
<script type="text/JavaScript">
function SendRequest(url, params)
{
   // Add some parameters to the query string
   var pageUrl = url + "?outofband=true&param=" + params;

   // Initialize the XmlHttpRequest object
   var xmlRequest, e;
   try {
        xmlRequest = new XMLHttpRequest();
   }
   catch(e) {
      try {
          xmlRequest = new ActiveXObject("Microsoft.XMLHTTP");
      }
      catch(e) { }
   }

   // Prepare for a POST synchronous request
   xmlRequest.open("POST", pageUrl, false);
   xmlRequest.setRequestHeader("Content-Type",
```

```
                                "application/x-www-form-urlencoded");
   xmlRequest.send(null);
   return xmlRequest;
}
</script>
```

The sample function accepts two strings: the URL to call and the parameter list. Note that the format of the query string is totally arbitrary and can be adapted at will in custom implementations. The URL specified by the programmer is extended to include a couple of parameters. The first parameter—named *outofband* in the example—is a Boolean value and indicates whether or not the request is going to be a custom callback request. By knowing this, the target page can process the request appropriately. The second parameter—named *param* in the example—contains the input parameters for the server-side code.

The host ASP.NET page looks like the following code snippet:

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Testing Out-of-band</title>
</head>
  <body>
    <form id="Form1" runat="server">
      <h1>Demonstrate Out-of-band Calls</h1>
      <h2><%=Request.Url%></h2>
      <hr />

      <asp:DropDownList runat="server" ID="EmployeeList" />
      <input id="Button1" type="button" value="Go Get Data"
             onclick="MoreInfo()" />
      <hr />
      <span id="Msg" />
    </form>
  </body>
</html>
```

The code-behind class is shown in the following listing:

```
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (IsOutOfBand())
            return;
        if (!IsPostBack)
            PopulateList();
    }

    private bool IsOutOfBand()
    {
        bool isCallback = false;
        isCallback = String.Equals(Page.Request.QueryString["callback"],
                                   "true",
                                   StringComparison.OrdinalIgnoreCase);
```

```
        if (isCallback)
        {
            string param = Request.QueryString["param"].ToString();
            Response.Write(ExecutePageMethod(param));
            Response.Flush();
            Response.End();
            return true;
        }
        return false;
    }

    private void PopulateList()
    {
        SqlDataAdapter adapter = new SqlDataAdapter(
            "SELECT employeeid, lastname FROM employees",
            "SERVER=(local);DATABASE=northwind;UID=...;");
        DataTable table = new DataTable();
        adapter.Fill(table);

        EmployeeList.DataTextField = "lastname";
        EmployeeList.DataValueField = "employeeid";
        EmployeeList.DataSource = table;
        EmployeeList.DataBind();
    }

    string ExecutePageMethod(string eventArgument)
    {
        return "You clicked: " + eventArgument;
    }
}
```

A couple of issues deserve more attention and explanation. The first one is the need to find out whether the request is an out-of-band call or a regular postback. Next, we need to look at the generation of the response. The *IsOutOfBand* method checks the *outofband* field in the posted form. If the *outofband* field is found, the request is served and terminated without going through the final part of the classic ASP.NET request life cycle—events for changed values, postback, pre-rendering, view-state serialization, rendering, and so forth. An out-of-band request is therefore short-circuited to return more quickly, carrying the least data possible.

What does the page do when an out-of-band call is placed? How does it determine the response? Most of the actual AJAX-based frameworks vary on this point, so let's say it is arbitrary. In general, you need to define a public programming interface that is invoked when an out-of-band call is made. In the sample code, I have a method with a contracted name and signature—*ExecutePageMethod*—whose output becomes the plain response for the request. In the sample code, the method returns and accepts a string, meaning that any input and output parameters must be serializable to a string.

```
string param = Request.QueryString["param"].ToString();
Response.Write(ExecutePageMethod(param));
Response.Flush();
Response.End();
```

As in the code snippet, the response for the out-of-band request is the output of the method. No other data is ever returned; and no other data except for the parameters is ever sent. In this particular implementation, there will be no view state sent and returned.

Important Although you'll probably never get to write any such code, be aware that thus far I've just provided a minimal but effective description of the underlying mechanism common to most frameworks that supply AJAX-like functionality. Each framework encapsulates a good number of details and adds new services and capabilities. At its core, though, this is how AJAX libraries work.

Displaying Results

One more step is missing—what happens on the client once the response for the out-of-band call is received? The following snippet shows a piece of client code that, when attached to a button, fires the out-of-band call and refreshes the user interface:

```
function MoreInfo()
{
   var empID = document.getElementById("EmployeeList").value;
   var xml = SendRequest("default.aspx", empID);

   // Update the UI
   var label = document.getElementById("Msg");
   label.innerHTML = xml.responseText;
}
```

According to the code, whenever the user clicks the button a request is sent at the following URL. Note that *1* in the sample URL indicates the ID of the requested employee. (See Figure 1-5.)

```
default.aspx?outofband=true&param=1
```

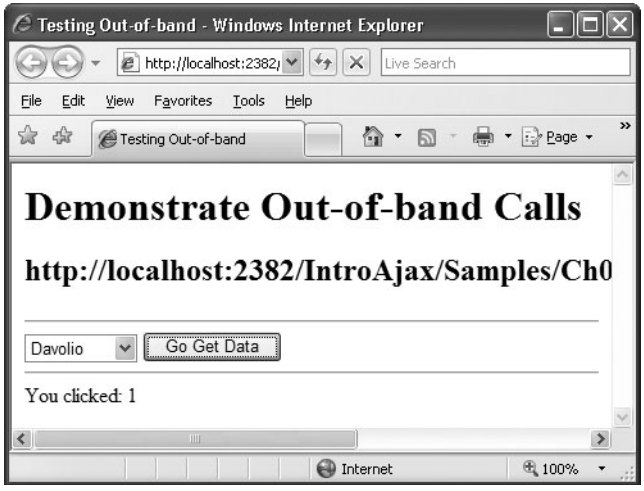

Figure 1-5 A manually coded out-of-band request in ASP.NET 1.x and ASP.NET 2.0

Displaying results correctly on most browsers can be tricky. Internet Explorer, in fact, supports a number of nonstandard shortcuts in the DOM that just don't work with other browsers. The most common snag is retrieving references to HTML elements using the *document.getElementById* method instead of the direct name of the element. For example, the following code works on Internet Explorer but not on Firefox and other Mozilla-equipped browsers:

```
// Msg is the ID of a <span> tag.
// This code works only with Internet Explorer
Msg.innerHTML = xml.requestText;
```

In summary, cross-browser JavaScript code is required to update the currently displayed page on the client. At the same time, a number of assumptions must be made on the server to come up with a working and effective environment. For this reason, frameworks are the only reasonable way of implementing AJAX functionalities. Different frameworks, though, might provide a different programming interface on top of an engine that uses the same common set of parts.

The Switch to the Document Object Model

Microsoft has pioneered updatable Web pages since the late 1990s. With Internet Explorer 4.0 (released back in 1997), Microsoft introduced Dynamic HTML (DHTML), which is a powerful combination of HTML, style sheets, and scripts that allows programmatic changes to any displayed page. Several companies since then have worked out their own DHTML object model—often referred to as the Browser Object Model (BOM). The W3C committee worked hard to bring vendors to agree on an interoperable and language-neutral solution for exposing Web pages through an updatable programming interface. The result is the Document Object Model (DOM), as opposed to a browser-specific BOM.

The DOM is a platform-independent and language-neutral representation of the contents of a Web page that scripts can access and use to modify the content, structure, and style of the document.

For AJAX, it's all about exchanging data with a remote server. But once the data is downloaded out-of-band on the client, what can you do with that? The DOM provides an outlet for the data to flow into the current page structure and update it.

Representation of a Document

The DOM is a standard API exposed by the browser in which a displayed page has a tree-based structure. Each node in the logical tree corresponds to an object. On the other hand, the name "Document Object Model" hints at an object model in the common interpretation of the object-oriented design terminology. A Web page—the document—is modeled through objects. The model includes the structure and behavior of a document. Each node in the logical tree is not static data; rather, it is a live object with a known behavior and its own identity.

DOM Implementation

The W3C DOM consists of three levels that indicate, for the browser, three different levels of adherence to the standard. For more information, take a look at *http://www.w3.org/DOM*.

The DOM is made of nodes, and each node is an object. For a Web page, each node maps to an object that represents an HTML tag. The object, therefore, has properties and methods that can be applied to an HTML tag. There are three fundamental operations you can accomplish on a node: find the node (including related nodes such as children, parent, or sibling nodes), create a node, and manipulate a node.

Identifying a particular node is easy as long as the page author knows the ID of the corresponding element. In this case, you use the following standard piece of code:

```
var node = document.getElementById(id);
```

In particular, if there are multiple elements sharing the same ID value, the method returns the first object in the collection. This method is supported in the DOM Level 1 and upper levels. Another interesting method to find elements is the following:

```
var coll = document.getElementsByTagName(tagname);
```

The method retrieves a collection of all objects based on the same HTML tag. For example, the method retrieves a collection of all <*div*> or all <*input*> tags in the page.

Related DOM objects are grouped in node lists. Each node has a name, type, parent, and collection of children. A node also holds a reference to its siblings and attributes. The following code snippet shows how to retrieve the parent of a node and its previous sibling:

```
var oParent = oNode.parentNode
var oPrevious = oNode.previousSibling
```

How can you modify the contents of a node? The easiest and most common approach entails that you use the *innerHTML* property:

```
var node = document.getElementById("button1");
node.innerHTML = "Hey click me";
```

The *innerHTML* property is supported by virtually all browsers, and it sets or retrieves the HTML between the start and end tags of the given object. Some browsers such as Internet Explorer also support the *innerText* property. This property is designed to set or retrieve the text inside of a given DOM object. Unfortunately, this property is not supported by all browsers. It exists in Internet Explorer and Safari but, for example, it is not supported by Firefox. Firefox, on the other hand, supports a property with a similar behavior but a different name—*textContent*.

Note The advent of the Microsoft AJAX Client Library (discussed in Chapter 2, "The Microsoft Client Library for AJAX") shields developers from having to know much about the little differences between the DOM implementation of the various browsers. For example, you should know about *innerText* and *textContent* if you're embedding your own JavaScript in the page; however, you don't have to if you rely on the AJAX Client Library to refresh portions of the displayed page.

Note Finally, note that you should not check the user agent string to figure out whether the current browser supports a given feature. You should check the desired object instead. For example, to know whether the browser supports *innerText*, you're better off running the following code:

```
var supportsInnerText = false;
var supportsInnerText = false;
if (temp != undefined)
supportsInnerText = true;
...
```

In this way, you directly check the availability of the property without having to maintain a list of browsers.

Nodes are created using the *createElement* method exposed only to the document object. Alternatively, you can add new elements to the document hierarchy by modifying the *innerHTML* property value, or by using methods explicit to particular elements, such as the *insertRow* and *insertCell* methods for the *table* element. Here's an example:

```
// Create an <IMG> element
var oImg = document.createElement("<img>");
...
// Create a new option for the SELECT element
var oOption = new Option(text, id);
control.options.add(oOption);
```

With this information, I have only scratched the surface of the DOM implementation in the various browsers. Nonetheless, the DOM is a key part of the AJAX jigsaw puzzle and deserves a lot of attention and skilled use. For a primer, you can take a look at *http://msdn.microsoft.com/workshop/author/dom/domoverview.asp*.

Be Pragmatic: DHTML vs. DOM

In the beginning, only the browser's support for the DHTML object model provided JavaScript developers with the ability to update the page contents dynamically. The success of DHTML led to the definition of a standard document object model—the W3C's DOM. Quite obviously, the DOM evolved from DHTML and became much more generalized than DHTML. As mentioned, the DOM provides a tree-based model for the whole document, not just for an individual HTML tag.

Most browsers, though, support a mix of DOM and DHTML. Which one should you use? In particular, to update some contents, should you obtain a reference to the textual child node of the node that matches the intended HTML tag (the DOM way) or just grab a reference to a node and use *innerHTML* (the DHTML way)? Likewise, to add a new element, should you create a new element or just stuff in a chunk of updated HTML via *innerHTML*? Admittedly, one of the most interesting debates in the community is whether to use DHTML to manipulate pages or opt for the cleaner approach propounded by the DOM application programming interface (API).

The key fact is that the DOM API is significantly slower than using *innerHTML*. If you go through the DOM to generate some user interface dynamically, you have to create every element, append each into the proper container, and then set properties. The alternative only entails that you define the HTML you want and render it into the page using *innerHTML*. The browser, then, does the rest by rendering your markup into direct graphics.

Overall, DHTML and DOM manipulation are both useful depending on the context. There are many Web sites that discuss performance tests and DHTML is always the winner. Anyway, DOM is still perfectly fast as long as you use it the right way—that is, create HTML fragments and append them to the proper container only as the final step.

Be Pragmatic: Use Events

Let's make it clear: without events, there would be no point in adding JavaScript to Web pages. To be effective, therefore, scripts have to react to some user action as well as to actions generated by the browser, such as when loading the page. Events and event handlers are old companions to Web pages, as they appeared the first time with Netscape 2.

For quite some time, largely incompatible event models lived and thrived in different browsers—mainly in Internet Explorer and Netscape. A few years ago, the W3C standardized the event model with a paper that you can read at *http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113/events.html*.

With Internet Explorer and Netscape having their own original event model, and making themselves compatible to the W3C standard, you understand that writing model-agnostic event handlers is going to be a hard task. There are a lot of events, but not all of them are supported by all browsers. The following categories of events can be considered standard: user interface events (blur, focus, scroll), device events (click, keydown), and form events (submit, select). The second big point concerns how you set event handlers. The most reliable way is still the following:

```
<a href="page.aspx" onclick="doClick()" />
```

An excellent paper that discusses the theme of events in JavaScript can be found here: *http://www.quirksmode.org/js/introevents.html*.

Tip If you're looking for a great Web site to learn about the various aspects of JavaScript, DHTML, DOM, CSS and client-side programming in general, the right place to go is *http://www.quirksmode.org*.

Existing AJAX Frameworks for ASP.NET

Today, quite a few APIs exist to implement AJAX functionality in ASP.NET, and one of these APIs is already integrated into ASP.NET 2.0. Other APIs come from third-party vendors or take form from open-source projects. I'll briefly look at some of these APIs. Note, though, that as long as ASP.NET is your development environment, the most reasonable choice you can make is Microsoft ASP.NET AJAX Extensions. However, ASP.NET AJAX Extensions can coexist pretty well with a large number of the existing alternative AJAX frameworks. ASP.NET AJAX is not a mutually exclusive choice.

Since early 2005, some aggressive independent software vendors (for example, Telerik, Infragistics, and ComponentArt) have integrated AJAX functionality into their existing suite of controls for rapid and rich Web development. In the beginning, each vendor developed its own internal and proprietary AJAX engine and integrated it with the product. The advent of ASP.NET AJAX Extensions will likely prompt vendors to offer native ASP.NET AJAX controls or, at a minimum, provide controls that work seamlessly with ASP.NET AJAX.

Let's review some of the options you have today for developing AJAX-enabled ASP.NET Web applications. As you can see, the list is not exhaustive and features libraries from both independent software vendor (ISV) companies and open-source projects started by outstanding members of the ASP.NET community.

ASP.NET Script Callbacks

ASP.NET 2.0 contains a native API, named ASP.NET Script Callback, to implement out-of-band calls to the same URL of the current page. This API makes the out-of-band request look like a special-case page request. It transmits the view state along with original input fields. A few additional input fields are inserted in the body of the request to carry extra information. Once on the server, the request passes through the regular pipeline of HTTP modules and raises the expected sequence of server-side events up to the pre-rendering stage.

Just before the pre-rendering stage, the page method is executed, the return value is serialized to a string, and then the string is returned to the client. No rendering phase ever occurs, and the view state is not updated and serialized back.

ASP.NET Script Callback provides its own JavaScript API to wrap any needed calls to *XMLHttpRequest*. As a developer, you are not required to know about this API in detail. As a developer, you should instead focus on the programming interface of the *GetCallbackEventReference* method of the *Page.ClientScript* object. This method simply returns the JavaScript code to attach to a client-side event handler to place an out-of-band call. The JavaScript code

also references another piece of JavaScript used to update the page with the results generated on the server. But what happens on the server when the secondary request is made? Which page method is executed?

ASP.NET Script Callback defines an interface—the *ICallbackEventHandler* interface—that any server object that is the target of an out-of-band call can implement. The target of the out-of-band call can be either the page or any of its child controls. The execution of an out-of-band call is divided into two steps: preparation and results generation. The *RaiseCallbackEvent* method of the *ICallbackEventHandler* interface is invoked first to prepare the remote code execution. The *GetCallbackResult* method is invoked later in the request life cycle when it is time for the ASP.NET runtime to generate the response for the browser.

All in all, the programming interface of ASP.NET Script Callback is a bit clumsy. Although the programming interface shields developers from a lot of internal details, it still requires the programmer to have good JavaScript skills and is articulated in a bunch of boilerplate server code. You need server code to bind HTML elements to client-side event handlers, and you need ad hoc server code to publish a programming interface that is callable from the client. Each request carries with it a copy of the original view state and rebuilds the last known good state on the server. In other words, the original value of all input fields in the currently displayed page (regardless of any changes entered before the out-of-band call is made) are sent to the server along with any parameters for the server method. Any out-of-band calls are processed as a regular postback request up to the pre-rendering stage, meaning that all standard server events are fired: *Init*, *Load*, *LoadComplete*, and so on. Before the pre-rendering stage, the callback is prepared and executed shortly after. The requests ends immediately after the server method executes. The view state is not updated to reflect the state of the page after the out-of-band call and subsequently, it is not sent back to the client.

The advantage of using ASP.NET Script Callback is that it is a native part of ASP.NET and can be easily encapsulated in server controls. For example, the *TreeView* control in ASP.NET 2.0 uses script callbacks to expand its nodes.

ASP.NET Script Callback is not free of significant issues, however. In particular, the server method is constrained to a fixed signature and can only take and return a string. Sure, you can place any contents in the string, but the serialization and deserialization of custom objects to the string is something you must take care of entirely on your own. In addition, a page based on ASP.NET Script Callback can have only one endpoint for remote calls. This means that if a client page needs to place two distinct calls to the same remote page, you have to implement a switch in the implementation of the *ICallbackEventHandler* interface to interpret which method was intended to be executed.

The AJAX.NET Professional Library

To effectively implement out-of-band calls in application-wide scenarios, a kind of framework is required that hides all the nitty-gritty details of HTTP communication and exposes additional and higher-level controls and services.

AJAX.NET Professional (AjaxPro) is a pretty popular open-source library that adds a good layer of abstraction over the *XMLHttpRequest* machinery. Written by Michael Schwarz, the library creates proxy classes that are used by client-side JavaScript to invoke methods on the server page. The AjaxPro framework provides full data type support and works on all common Web browsers, including mobile devices. Nicely enough, the library can be used with both ASP.NET 1.1 and ASP.NET 2.0.

The key tool behind the AjaxPro library is an HTTP handler that hooks up any HTTP requests generated by the client-side part of the library:

```
<httpHandlers>
   <add verb="POST,GET" path="ajaxpro/*.ashx"
        type="AjaxPro.AjaxHandlerFactory, AjaxPro.2" />
</httpHandlers>
```

Once the *web.config* file has been correctly set up, you write JavaScript functions to trigger and control the out-of-band call. Each call targets a JavaScript object that represents the publicly callable method on the server ASP.NET page. A client-callable method is just a public method decorated with a specific attribute, as shown here:

```
[AjaxPro.AjaxMethod]
public DateTime GetCurrentTimeOnServer()
{
  return DateTime.Now;
}
```

The class with public methods, as well as any custom types used for I/O, has to be registered with the framework to have the corresponding JavaScript proxy created:

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

If you do this, the handler guarantees that any managed .NET object that is returned by a server method will be serialized to a dynamically created JavaScript object to be seamlessly used on the client. You can return any managed type, your own classes, or enum types as you would do in plain .NET code. No view state is available during the AJAX request, meaning that you can't do much with page controls. In light of this, it is recommended that you create callable methods as static methods preferably, though not necessarily, on a separate class.

AjaxPro has some key advantages over the ASP.NET Script Callback API. It uses an attribute to mark server methods that can be called from the client. This means that you have the greatest flexibility when it comes to defining the server public interface callable from the client. In particular, you don't have to change the flow of the code or add new ad hoc methods just to comply with the requested programming interface.

In addition, you can register server types for use on the client, which provides for a strong-typed data transfer. The AjaxPro infrastructure serializes .NET types to JavaScript objects and vice versa. The AJAX.NET hooks up and replaces the standard request processing mechanism of ASP.NET—the page handler. As a result, you won't receive classic ASP.NET server events such as *Init*, *Load*, and postback. At the same time, you won't have the view state automatically transmitted with each out-of-band request. An AjaxPro request, though, is still processed by the ASP.NET HTTP runtime, meaning that the request is still subject to the modules registered with the HTTP pipeline, including session state management, roles, and authentication.

For more information about the AjaxPro library, you can take a look at *http://www.ajaxpro.info*. There you will also find a link to the CodePlex Web site to get the source code of the library.

The Anthem.NET Framework

Anthem.NET is a free, cross-browser AJAX toolkit for both ASP.NET 1.1 and 2.0, written by Jason Diamond. The library is made of a number of server controls that use *XMLHttpRequest* to post back. It sets itself apart from AjaxPro because it fully integrates with the classic life cycle of each ASP.NET request. The view state is sent across the wire, and server-side page and control events such as *Init*, *Load*, and *PreRender* are regularly fired. As a result, you write a page using the same programming model of ASP.NET, you are not required to write any JavaScript yourself, and you still leverage the beauty of the AJAX model. The only difference with a traditional ASP.NET application is that you use a different set of server controls, most of which are just subclassed versions of the original ASP.NET controls.

Extremely lean and easy to use, Anthem.NET implements AJAX functionalities through the "partial rendering" model applied at the control level. The partial rendering model is the same model that ASP.NET AJAX pushes hard. (See Chapter 4, "Partial Page Rendering.") For more information, check out *http://www.anthemdotnet.com*.

The ComfortASP.NET Framework

Conceptually similar to Anthem.NET, but significantly different in its implementation, is Daniel Zeiss' ComfortASP.NET framework. ComfortASP.NET uses a manager server control to inject script code in the client page. Invisible to the page author, the script code hooks up client postbacks and replaces them with calls to *XMLHttpRequest*.

Once back on the server, the manager component takes control of the operations and determines the delta between the current page and the page resulting from the processing of the current request. The markup that describes the changes in the displayed page is sent back and used to dynamically modify the page contents on the client via the previously emitted script. The server life cycle of the page is executed as usual, and events such as *Init* and *Load* are fired when expected.

The ComfortASP.NET framework refers to this technique as "selective update;" but in the end it is just another term to indicate what ASP.NET AJAX calls "partial rendering." (We'll cover partial rendering in Chapter 4.)

Using ComfortASP.NET couldn't be easier and faster. It only requires you to tweak the *web.config* file to add an HTTP handler for ASP.NET requests and add a manager control to each page you intend to expand with AJAX capabilities. The manager control features a few interesting properties such as compression, automatic form disabling during postback, and request timeout handling.

Taken alone, the manager control works on the page as a whole. The framework also includes a *PanelUpdater* control for you to selectively update specific portions (panels) of the page. You can learn more about the ComfortASP.NET Framework at *http://www.comfortasp.de/*.

The Telerik r.a.d.controls for ASP.NET Framework

Telerik r.a.d.controls for ASP.NET is a suite of versatile user-interface (UI) components, which offer complete interoperability with Microsoft ASP.NET AJAX Extensions. The product allows developers to build a sophisticated and largely customizable user interface based on ASP.NET AJAX. This means that r.a.d.controls are safe for use inside of any ASP.NET AJAX page and interact smoothly with any built-in ASP.NET AJAX controls.

Telerik is currently working on a special update of the r.a.d.controls suite, which will leverage the complete capabilities of the ASP.NET AJAX Framework. The new version of the product should be available by the time you read this book. Among the novelties, you can certainly expect a client-side object model that is consistent with the Microsoft AJAX Client Library conventions and controls that fully participate in the client life cycle of the request. (See Chapter 2.) In addition, the r.a.d.controls suite has rich type information similar to the .NET type descriptors, easy component discoverability and enumeration, and optimized resource management and disposal on partial page updates.

Apart from that, Telerik offers its own AJAX framework, called r.a.d.ajax. The purpose of the product is to eliminate the complexities of building JavaScript-intensive AJAX applications so that developers can take advantage of this new technology with no additional learning curve to climb. Complexities are eliminated by encapsulating the AJAX engine and all surrounding logic, including scripts, into classic ASP.NET server components, which can be configured visually with convenient builders in Visual Studio 2005. As a result, developers can simply write regular postback-based applications and turn them into AJAX-enabled ones without writing any JavaScript or server-side code.

The Telerik engine completely preserves the life cycle of the ASP.NET page, which is imperative for the proper operation of your application. The view state, event validation, and client-side scripts are also preserved as if a normal postback takes place. All form values are automatically sent to the server for processing. Telerik's framework is based on a patent-pending technology that manages AJAX postbacks internally.

For more information, visit *http://www.telerik.com*.

The ComponentArt Web.UI Framework

ComponentArt features Web.UI for ASP.NET AJAX—the first suite of controls designed specifically for ASP.NET AJAX. The library has a variety of advanced user interface controls for use in sophisticated Web applications—for example, grids, splitters, tree views, and drop-down lists. The *Callback* control, on the other hand, provides base AJAX capabilities.

The Web.UI library goes beyond mere compatibility or basic interoperability with ASP.NET AJAX. Rather, it offers deep integration into the new Microsoft AJAX framework. You find a bunch of server controls enriched with a client-side object model that fully leverages the Microsoft AJAX Client Library type system. (We'll cover the Microsoft AJAX Client Library in Chapter 2.) In particular, all controls inherit from the *Sys.UI.Control* client-side base class and expose extensive client-side methods and attributes to be invoked and set via script. Controls participate in the client-side life cycle, notify events, and communicate with native DOM elements.

All Web.UI controls have the ability to command AJAX postbacks on their own or through the ASP.NET AJAX's *UpdatePanel* control. (See Chapter 4.) For example, the *TreeView* and *Grid* controls implement their own built-in lightweight callback mechanisms for things such as load on demand or paging.

Similar to the ASP.NET AJAX *UpdatePanel*, the *CallBack* component can optionally wrap controls to update, and it can either bypass the standard page life cycle and execute server-side logic more quickly or maintain the latest state of all ASP.NET controls contained in the page through the view state. The client-side model of the *CallBack* component can be used from the client to execute server-side code. The *CallBack* control, though, is not used by any other Web.UI control internally.

For more information, visit *http://www.componentart.com*.

Infragistics's NetAdvantage for ASP.NET AJAX

NetAdvantage for ASP.NET—the Infragistics's flagship product for ASP.NET development—offers a full range of components ranging from a tree and menu, to a hierarchical grid, and even a charting engine. Infragistics employs a technique known as "Embedded AJAX" to build the AJAX functionality directly into their controls. By embedding the AJAX into the control itself, performance levels are achieved that would not otherwise be possible (when utilizing a separate AJAX engine or wrapper). In addition to the built-in AJAX features, Infragistics also supplies the "WARP Panel," which can be used to give any control(s) AJAX capabilities, much in the same manner as Microsoft's UpdatePanel.

Because the AJAX capabilities are built into Infragistics' WebControls, a developer need only know how to set a property to start using AJAX. Infragistics refers to this concept as

"No-Touch AJAX." Should you want to get your hands dirty, Infragistics provides a full client-side object model with API's and even an event model that can be programmed entirely through JavaScript. NetAdvantage for ASP.NET offers interoperability with Microsoft's ASP.NET AJAX Extensions—enabling you to use these two powerful toolsets side-by-side. Though the current level of integration with ASP.NET AJAX is not as deep as we've seen with ComponentArt's Web.UI, Infragistics manages to provide much of the same functionality through their own framework.

For more information, visit *http://www.infragistics.com/ajax*.

Categorizing AJAX Frameworks

As you can witness yourself, each AJAX-oriented framework falls into one of the following three main categories:

- RPC-style frameworks
- A suite of rich controls
- AJAX frameworks

RPC-style frameworks are ASP.NET libraries that simply provide the capability of calling back server code from the client via JavaScript. ASP.NET Script Callbacks and AjaxPro certainly have this capability.

Commercial products from popular vendors such as Telerik, ComponentArt, and Infragistics offer a suite of controls with AJAX capabilities. Currently, they don't provide the same level of integration with the ASP.NET AJAX platform; however, in the short term they will be aligned at the same level and differentiate their product offerings by extending differing levels of features and capabilities, some solidly Microsoft ASP.NET AJAX compliant (Component Art) and others to a lesser degree.

Finally, there will be pure AJAX frameworks—that is, a code library that enables pages and applications to do AJAX. Of course, ASP.NET AJAX Extensions is the most rich and powerful option, and it's certainly the standard to follow for the largest share of developers. However, a number of good frameworks (often, open-source frameworks) exist—such as Anthem.NET and ComfortASP.NET—that simply help you build AJAX pages quickly and effectively. They have anticipated most of the features you find today in ASP.NET AJAX Extensions.

ASP.NET AJAX in Person

Architecturally speaking, the ASP.NET AJAX framework is made of two distinct elements: a client script library and a set of server controls that add AJAX capabilities to ASP.NET 2.0. The client script library is written entirely in JavaScript and therefore works with any modern browser. ASP.NET AJAX offers an end-to-end programming model that spans the client and

server environment. It's also seamless to use for most developers because it simply extends the popular and known application model of classic ASP.NET.

Setting Up ASP.NET AJAX Extensions

Before we delve into the ASP.NET AJAX architecture, let's briefly review some common issues related to installing, configuring, and running ASP.NET AJAX applications.

Installing ASP.NET AJAX Extensions

The setup phase of ASP.NET AJAX Extensions installs debug and release copies of the AJAX Script Library and any needed binaries. If you have any version of Visual Studio 2005 installed, the package also configures the Integrated Development Environment (IDE) to show a ready-made AJAX project template. (See Figure 1-6.)

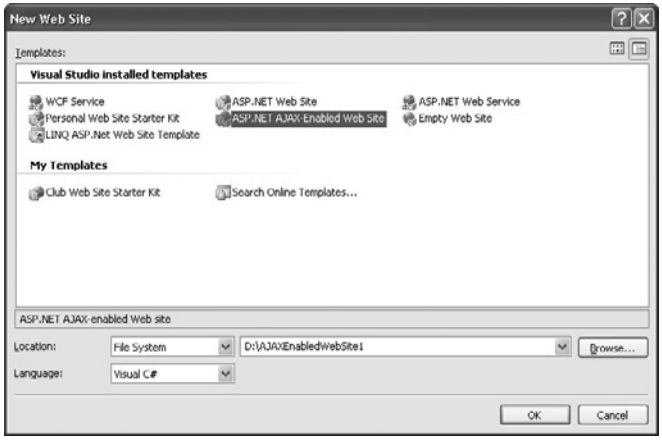

Figure 1-6 The new AJAX project template that shows up when you create a new Web site.

The ASP.NET AJAX Extensions Microsoft Windows Installer (MSI) package installs a handful of files on your computer under the following folder:

```
%DRIVE%:\Program Files\Microsoft ASP.NET\ASP.NET 2.0 AJAX Extensions\v1.0.61025
```

In addition, it places an assembly named *System.Web.Extensions.dll* in the global assembly cache (GAC). The ASP.NET AJAX assembly incorporates a bunch of JavaScript files (*.js* files) that form the client script library.

Note The official installer of ASP.NET AJAX copies the binaries in the GAC. This is still the recommended way to go. However, simply copying the *System.Web.Extensions* assembly in the Bin folder does suffice to deploy an ASP.NET AJAX Web site.

Deploying ASP.NET AJAX Applications

The simplest way to create an ASP.NET AJAX application is by choosing the Visual Studio 2005 project template. (See Figure 1-6.) Visual Studio adds to the project a *web.config* file that contains all settings required to run an ASP.NET AJAX application. In particular, the configuration file links the ASP.NET AJAX assembly to the project.

If your ASP.NET AJAX application consumes Web services, you should ensure that these Web services—local to the application—have correctly installed and can find all of their required resources.

Important Not all configuration entries created in the default *web.config* file are required in all cases. You might want to remove those that you don't need. In particular, you might want to remove HTTP handlers and HTTP modules that serve calls to Web services and page methods, respectively, if your application doesn't place remote out-of-band calls directly from JavaScript. However, when you edit the web.config file pay a lot of attention and limit yourself to commenting out parts rather than deleting them. You might inadvertently remove an important setting that breaks the whole application.

Core Components

The ASP.NET AJAX framework is made of a client and a server part. Applications use a client-side JavaScript library mostly to manage the page user interface, to call server-based components, and order partial page refreshes. Server components generate the response for the client and emit predefined client script that integrates and sometimes extends the client library. The server-side part of ASP.NET AJAX includes Web services, ad hoc controls, and the JavaScript Object Notation (JSON) infrastructure. (I discuss the JSON data interchange technology a bit later in this chapter.)

The Microsoft Client Library for AJAX

The AJAX client library is made of a set of JavaScript (**.js*) files that are linked from client pages in case of need. These **.js* files are downloaded on each client that consumes ASP.NET AJAX pages. These files are transparent to ASP.NET developers, as they are embedded in the ASP.NET AJAX assembly.

The client library provides object-oriented and cross-browser extensions to the JavaScript language such as classes, namespaces, inheritance, and data types. In addition, it defines a largely shrink-wrapped version of the .NET base class library that includes string builders, regular expressions, timers, and tracing. The key part of the ASP.NET AJAX client library is the networking layer that manages the complexity of making asynchronous calls over *XMLHttpRequest*. This layer allows the client page to communicate with Web services and Web pages through out-of-band calls.

Server-Based Components

ASP.NET AJAX is an extension to ASP.NET, and ASP.NET is a server-side development platform. Hence, ASP.NET AJAX sports a number of server-based components, including Web services and controls, that offer a double benefit. On one end, you can program these components from the client and update the current page without a full refresh; on the other hand, though, the programming model remains unaltered for the most part. In this way, at least limited to core functionalities, the ASP.NET AJAX learning curve might be pleasantly short.

Built-in Web services expose a handful of ASP.NET features to client pages, including user profiles, membership, and roles. Server controls look like classic ASP.NET server controls, except that they emit additional script code. The script code enriches the user's experience with the control by optionally taking advantage of the facilities provided by the AJAX client library. Some key AJAX server controls you will work with are *UpdatePanel*, *UpdateProgress*, and *Timer*. I'll cover them in Chapter 4.

The JSON Infrastructure

The growing use of out-of-band calls in Web applications poses a new issue—moving more and more complex data around. It's not a mere serialization issue for which the .NET Framework and other platform-specific frameworks have a ready-made solution. The serialization involved with out-of-band calls is not just cross-platform; it also involves distinct tiers and radically different tools and languages. With out-of-band calls, you move data from a client to a server and back. But the client is a browser (if not a mobile device), and JavaScript is the native format of data. The server is a Web server hosted on a variety of hardware/software platforms and running a specific Web application framework.

JSON is the emerging technology for passing structured data across the Web. It is a data interchange format and is fully described at *http://www.json.org*. Relatively easy to read for humans and to parse and generate for machines, JSON describes data using two universal data structures—collections and array—that are supported in one way or another by most modern programming languages and class libraries.

JSON is a text format that is completely language independent, although it relies heavily on a number of conventions inherited from the C family of languages.

The JSON client infrastructure can serialize a JavaScript object to an interchange format and send it over the wire to a server-side receiver. The platform-specific receiver will parse the data stream to build a platform-specific object. Likewise, the JSON server infrastructure can take any platform-specific object and serialize to an interchange format. Back on the client, the data stream is promptly transformed in a JavaScript object. As far as the .NET Framework and ASP.NET are concerned, a bit of reflection is used to examine the internal structure of classes and create proper JavaScript wrappers.

Virtually all AJAX-based frameworks implement a JSON infrastructure. ASP.NET AJAX is no exception.

Note For a while, XML has been touted as the lingua franca of the Web because it is ideal and made-to-measure for developers and architects to package and exchange data in a totally cross-platform way. Today, you find out that JSON (a non-XML technology) is sold for the same task. Is there any difference? Both JSON and XML do the same work. XML is more complex, quirky in some respects, and general-purpose, and it is preferable to describe data to be styled using an XSLT style sheet for UI purposes. For raw data, JSON is a more light-weight format that is easier to read and parse for both humans and computers.

Conclusion

Most attentive developers have been developing around interactive Web technologies since the late 1990s. Various technologies (for example, Microsoft Remote Scripting and open-source and commercial variations) have been developed without forming a critical mass of acceptance and use. Or perhaps the mass was big enough, but everyone was waiting for the spark of a killer application. Another factor that slowed down the adoption of more advanced client techniques was the lack of cross-browser support for them.

Today, the situation is radically different from what it was only three or four years ago. Now about 90 percent of the available browsers support all the minimal requirements for implementing interactive Web applications, known as AJAX applications. In addition, the W3C is standardizing the *XMLHttpRequest* object, which is the necessary communication workhorse behind all existing platforms for AJAX. The next generation of Web applications will be based on a different mechanism: it is no longer, or not just, forms posted in a change of pages, but individual requests for data and dynamic updates to displayed pages.

As a server technology aimed at the creation of Web pages, ASP.NET takes advantage of the opportunity for providing this much desired functionality. Script callbacks were the first Microsoft attempt to offer an API for building AJAX-style pages. Modeled after the classic postback event, callbacks are sometimes unnecessarily heavy and inflexible.

An add-on to ASP.NET 2.0, Microsoft ASP.NET AJAX Extensions, shows the way ahead for AJAX applications as far as the ASP.NET platform is concerned. It integrates the AJAX lifestyle into the existing application model of ASP.NET, resulting in a familiar programming model with greatly improved and richer functionality.

Chapter 5

The AJAX Control Toolkit

In this chapter:

Although ASP.NET AJAX is a framework designed to bring more programming power to the Web client, it happens to be mostly used by server developers—for example, ASP.NET developers or, at least, Web developers with strong server-side skills. Unfortunately, for the time being, there's no way to add rich capabilities and functionalities to the Web client other than by crafting good and tricky JavaScript code.

ASP.NET AJAX takes up the challenge and provides two ways for developers to build rich Web applications using a server-centric development approach. As we discussed in Chapter 4, "Partial Page Rendering," developers can refresh specific regions of the page using partial rendering instead of normal ASP.NET postbacks. To create such regions, you just use a particular set of server controls—the most important of which is the *UpdatePanel* control.

In addition to partial rendering, developers can use *control extenders* to add a predefined client-side behavior to new and existing ASP.NET controls. A client-side behavior is a block of JavaScript code that adds a new capability to the markup generated by a given ASP.NET control. An extender is basically a server control that emits proper script code—the client behavior—to enhance how a given ASP.NET control behaves on the client. An extender is not simply a custom control derived from an existing control. Rather, it represents a general behavior—such as auto-completion, focus management, generation of popups, and draggability—that can be declaratively applied to various target control types. For example, a special behavior can be applied to any focused control—be it a *TextBox*, *Button*, or *CheckBox* control.

ASP.NET AJAX Extensions 1.0 simply delivers the base class for extender controls. No concrete extender controls are provided with the binaries. The online documentation provides some good tutorials on how to build extenders. You can find one at *http://ajax.asp.net/docs/tutorials/ExtenderControlTutorial1.aspx*. A fair number of sample extenders

and additional rich client controls are provided through a separate download—the AJAX Control Toolkit (ACT).

In this chapter, I'll first review the syntax and semantics of control extenders and then take you on a tour of the major components in the ACT.

Extender Controls

ASP.NET pages are made of server controls. ASP.NET comes with a fairly rich collection of built-in controls. In addition, plenty of custom controls are available for developers from third-party vendors, from community projects, and even from contributions by volunteers. If you need a text box with a set of features that the ASP.NET control can't provide (for example, a numeric text box), you typically write one yourself or buy a new specialized control that extends the original control and adds the desired behavior. Object orientation, of course, encourages this approach.

However, it's rare that you need to write a completely new control yourself. More often, your control will derive from an existing ASP.NET control base class. Blindly using inheritance for building specialized versions of controls might not be a wise choice, though. Even in relatively small projects, in fact, it can lead straight to a proliferation of controls. For example, you can end up with a regular text box, plus a numeric text box, a filtered text box, a text box that changes its style when focused, a text box that displays a prompt when left empty, and so on. On the other hand, merging all these behaviors into a single super *TextBox* control might not be wise either. In this case, the resulting code will be literally full of branches, logical conditions, and properties to check. For just a simple extra feature, you would load a huge control. There has to be a different approach. Enter extender controls.

What Is an Extender, Anyway?

First and foremost, an extender control is a server control itself. It represents a logical behavior that can be attached to one or more control types to extend their base capabilities.

Formalizing the Concept of a "Behavior"

Imagine you want only the text boxes in a given input form to change their style when focused. If you create a new control, say *FocusedTextBox*, you're fine. What if, instead, you want the same behavior from buttons, check boxes, and drop-down lists? You should create a bunch of new controls—all of which will extend the target controls with the same logical behavior. Extender controls are just a formal way to define such a behavior.

Note Virtually all behaviors require the injection of some script code in the client page. For this reason, extenders are naturally associated with ASP.NET AJAX. From a technology standpoint, on the other hand, ASP.NET AJAX and extenders are independent concepts. You could

develop extenders for ASP.NET 1.1 and ASP.NET 2.0 that work without ASP.NET AJAX Extensions. However, ASP.NET AJAX Extensions provides some interesting facilities for writers of extender controls—specifically, base classes and, more importantly, the Microsoft AJAX library for developing JavaScript functionalities more comfortably.

A typical extender control is made of a set of properties and one or more JavaScript files that, all together, define the expected behavior of the target control in the browser. The ASP.NET developer adds extenders declaratively to a server page and configures properties to obtain the desired behavior.

Next, when the extender renders out, it emits proper script code in the client page. This script code typically registers handlers for client-side events and modifies the Document Object Model (DOM) of the markup elements it is associated with. As a result, the original control looks and behaves in a slightly different manner while its programming interface remains intact.

To some extent, the concept of a "behavior" is similar to a theme. The theme is used to change the control's look and feel. Where the behavior and theme differ, however, is that the behavior might change some visual aspects of the control, but it is not limited to graphical attributes. It can alter the structure of the control by accessing the client DOM, add event handlers, and even expose a true object model with properties and methods.

Examining a Sample Extender

To better understand the goals and characteristics of AJAX extenders, let's briefly consider the behavior encapsulated by one of the extenders contained in the ACT—the *TextBoxWatermark* extender.

A text box watermark is a string of text that is displayed in an empty text box as a guide to the user. This help text is stripped off when the text box is submitted and is automatically removed as the user starts typing in the field. Likewise, it is automatically re-inserted when the user wipes out any text in the text box.

The watermark behavior hooks up three HTML events: *onfocus*, *onblur*, and *onkeypress*. In its initialization stage, it also sets a new style and default text for the target text box if the body of the field is empty. When the text box gets the input focus, the event handler promptly removes the watermark text and restores the original style. As the user types, the handler for *onkeypress* ensures that the current text box is watermarked. Finally, when the input field loses the focus—the *onblur* event—the handler sets the watermark back if the content of the field is the empty string.

To associate this behavior with an ASP.NET *TextBox*, you use the extender. Alternatively, if you feel comfortable with ASP.NET control development and JavaScript, you can use a client-side code fragment to achieve the same results.

Note The concept of AJAX extenders closely resembles Dynamic HTML (DHTML) behavior. Introduced with Internet Explorer 5.0, DHTML behaviors were nothing more than a script file (or a compiled COM object) that hooked up HTML events and modified the DOM of a given HTML tag to implement a given behavior. DHTML behaviors were used to extend the capabilities of individual HTML tags. ASP.NET AJAX behaviors are used to extend the capabilities of the markup block generated by individual ASP.NET controls.

Target Properties

Extender controls are characterized by a set of properties that determine the resulting behavior. The values of these properties are passed on to the client and incorporated in the client script.

Obviously, a made-to-measure framework is required both on the server and the client to make the implementation of behaviors effective and, more importantly, affordable. This framework is exactly the benefit that ASP.NET AJAX Extensions provides. We'll examine the internals of extenders in a moment while going through some sample code. Meanwhile, let's take a quick look at how you actually use extenders in ASP.NET pages.

In a page, you have one extender instance for each control you want to enhance. The extender is decorated with a set of properties, as shown here:

```
<act:TextBoxWatermarkExtender ID="Watermark1" runat="server"
     TargetControlID="TextBox1"
     WatermarkText=" ... "
     WatermarkCssClass=" ... " />
</act:TextBoxWatermarkExtender>
<act:TextBoxWatermarkExtender ID="Watermark2" runat="server"
     TargetControlID="TextBox2"
     WatermarkText=" ... "
     WatermarkCssClass=" ... " />
</act:TextBoxWatermarkExtender>
```

The *TargetControlID* property is common to all extenders and indicates the control in the page that is the target of the extender. Other properties specific to the extender tailor its individual behavior. The *WatermarkText* and *WatermarkCssClass* properties are implemented only by the Watermark Extender, for example, and serve to assign the text and style the watermarked text should exhibit.

Armed with this background information, let's take the plunge into the programming interface of extender controls.

The *ExtenderControl* Class

As mentioned, ASP.NET AJAX Extensions doesn't include any concrete implementation of an extender. However, it defines the base class from which all custom extenders, as well as all extenders in the ACT, derive. This class is named *ExtenderControl*.

Generalities of Extender Controls

The *ExtenderControl* class derives from *Control* and implements the *IExtenderControl* interface. The class defines one specific property—*TargetControlID*. The property is a string and represents the ID of the server control being extended. The *Visible* property, common to all server controls is overridden and made virtually read-only. More precisely, you can't override a read/write property to remove the set modifier, but you can just make it throw an exception if invoked. Here's the pseudocode of the property:

```
public override bool Visible
{
      get { return base.Visible; }
      set { throw new NotImplementedException(); }
}
```

An extender requires a script manager control in the page, just as any ASP.NET AJAX server controls do. Note that extenders are mostly used declaratively and are never modified programmatically. For this reason, an extender doesn't need (and doesn't use) view state.

The *IExtenderControl* Interface

The *IExtenderControl* interface defines the contract of an extender control. It comprises two methods: *GetScriptDescriptors* and *GetScriptReferences*. Here's the definition of the interface:

```
public interface IExtenderControl
{
   IEnumerable<ScriptDescriptor> GetScriptDescriptors(
        Control targetControl);
   IEnumerable<ScriptReference> GetScriptReferences();
}
```

Both methods return a collection of specific objects—script descriptors and script references, respectively.

A script descriptor is represented by an instance of the *ScriptDescriptor* class, whereas the *ScriptReference* class represents a linked script file. What is a script descriptor, anyway? It describes the JavaScript class that provides the expected client behavior. A script descriptor indicates the client type to create, the properties to set, and the client events for which handlers are required.

Actually, *ScriptDescriptor* is just a base class that you use only as a reference. The real class you work with is *ScriptBehaviorDescriptor*. Here's some code that demonstrates the typical implementation of a *GetScriptDescriptors* method in a sample extender control:

```
protected IEnumerable<ScriptDescriptor> GetScriptDescriptors(
    Control targetControl)
{
    ScriptBehaviorDescriptor descriptor;
    descriptor = new ScriptBehaviorDescriptor(className, id);
```

```
    descriptor.AddProperty(propertyName1, value1);
   ...
    return new ScriptDescriptor[] { descriptor };
}
```

The *ScriptBehaviorDescriptor* class doesn't feature public properties, but it does expose a cargo collection property that is filled with property descriptions—typically, name and value.

As discussed in Chapter 3, "The Pulsing Heart of ASP.NET AJAX," a *ScriptReference* object describes a piece of JavaScript code. In particular, in this context it represents the client script included with the behavior. All referenced scripts define client types and any other auxiliary JavaScript code that is required. We'll return to this method in a moment.

The *ExtenderControl* base class implements the *IExtenderControl* interface by falling back into internal members that are declared as *protected* and *abstract* (or as must-override in Microsoft Visual Basic .NET):

```
IEnumerable<ScriptDescriptor> IExtenderControl.GetScriptDescriptors(
      Control targetControl)
{
   return this.GetScriptDescriptors(targetControl);
}
IEnumerable<ScriptReference> IExtenderControl.GetScriptReferences()
{
   return this.GetScriptReferences();
}
```

The internal members must be overridden in any derived classes.

Creating a Sample Extender

Let's apply the previously exposed concepts to a practical scenario and create a sample extender control. The new extender adds a highlighting behavior that changes the appearance of the control when this gets focused. This extender, named *FocusExtender*, is not specific to just one control but can be applied to virtually any ASP.NET control.

The *FocusExtender* Control

The focus extender is implemented as a control type that derives from *ExtenderControl* and overrides the two abstract members on the base class:

```
[TargetControlType(typeof(Control))]
public class FocusExtender : ExtenderControl
{
    protected override IEnumerable<ScriptReference> GetScriptReferences()
    {
       ...
    }
```

```
    protected override IEnumerable<ScriptDescriptor> GetScriptDescriptors(
         Control targetControl)
    {
       ...
    }
    ...
}
```

The *TargetControlType* attribute defines the base class of controls that can be extended. In this case, all Web controls can be extended. As you can see, if multiple control types must be extended, it is required that they all derive from the same base class.

The skeleton of the extender is pretty much done. You only have to add some properties to make the extender configurable and flesh out the body of abstract methods.

The focus extender is designed to change the appearance of the target control when this gets the focus. At a minimum, you need two sets of visual properties: one for the highlighted state and one for the normal state. However, these attributes will be processed and applied on the client. On the client browser, though, there's a better way to set the appearance of elements than by using individual properties—cascading styles. To assign these styles, the extender has two properties—*HighlightCssClass* and *NormalCssClass*:

```
public class FocusExtender : ExtenderControl
{
    private string _highlightCssClass;
    private string _normalCssClass;

    public string HighlightCssClass
    {
        get { return _highlightCssClass; }
        set { _highlightCssClass = value; }
    }

    public string NormalCssClass
    {
        get { return _normalCssClass; }
        set { _normalCssClass = value; }
    }
    ...
}
```

As mentioned, an extender class doesn't need a view state and implements storage for properties through private fields. This requires that an extender be either used declaratively (and not modified programmatically during postbacks) or fully configured in the page initialization phase regardless of postbacks.

Defining the Client Behavior

The *GetScriptDescriptors* method instantiates a descriptor class and it uses that class to bind the *IntroAjax.FocusBehavior* JavaScript class to the HTML subtree rooted in the client ID of the

specified control. The descriptor registers bindings between properties on the JavaScript class and properties on the extender control. The *AddProperty* method is called upon to link, say, the *highlightCssClass* on the JavaScript's *FocusBehavior* class and the *HighlightCssClass* on the extender.

```
protected override IEnumerable<ScriptDescriptor> GetScriptDescriptors(
       Control targetControl)
{
    ScriptBehaviorDescriptor descriptor;
    descriptor = new ScriptBehaviorDescriptor("IntroAjax.FocusBehavior",
          targetControl.ClientID);
    descriptor.AddProperty("highlightCssClass", this.HighlightCssClass);
    descriptor.AddProperty("normalCssClass", this.NormalCssClass);

    return new ScriptDescriptor[] { descriptor };
}
```

The binding between client and server properties serves to have the client property set with the value specified on the server. An extender should be seen as a pair of classes—one server control class, and a JavaScript class that exposes the extender's object model on the client.

You save the JavaScript class in one or more *.js* files and then register all of them with the ASP.NET AJAX infrastructure through the *ScriptReference* class. Here's a typical implementation of the *GetScriptReferences* method:

```
protected override IEnumerable<ScriptReference> GetScriptReferences()
{
    ScriptReference reference = new ScriptReference();
    reference.Path = ResolveClientUrl("FocusBehavior.js");

    return new ScriptReference[] { reference };
}
```

The *ResolveClientUrl* method is defined on the *Control* class, and it resolves a relative URL in the context of the application. Used as in the preceding code snippet, the method looks for a *FocusBehavior.js* file in the same folder as the host page. (However, that's not the place where reasonably most developers would put it.)

The Extender's Client Object Model

The *FocusBehavior.js* file defines a behavior class with properties that reflect the properties of the server-side extender control—*highlightCssClass* and *normalCssClass*. In addition, the behavior registers a couple of handlers for *blur* and *focus* client events. When the *focus* event fires, the behavior sets the highlighted cascading style sheet (CSS) style. It sets the normal CSS style when the *blur* event occurs. Here's the full source code:

```
Type.registerNamespace('IntroAjax');

// Constructor
IntroAjax.FocusBehavior = function(element)
```

```
{
    IntroAjax.FocusBehavior.initializeBase(this, [element]);
    this._highlightCssClass = null;
    this._normalCssClass = null;
}

// Create the prototype for the behavior
IntroAjax.FocusBehavior.prototype =
{
    initialize : IntroAjax$FocusBehavior$initialize,
    dispose : IntroAjax$FocusBehavior$dispose,
    _onFocus : IntroAjax$FocusBehavior$_onFocus,
    _onBlur : IntroAjax$FocusBehavior$_onBlur,
    get_highlightCssClass : IntroAjax$FocusBehavior$get_highlightCssClass,
    set_highlightCssClass : IntroAjax$FocusBehavior$set_highlightCssClass,
    get_normalCssClass : IntroAjax$FocusBehavior$get_normalCssClass,
    set_normalCssClass : IntroAjax$FocusBehavior$set_normalCssClass
}

// Internal methods
function IntroAjax$FocusBehavior$initialize()
{
    IntroAjax.FocusBehavior.callBaseMethod(this, 'initialize');

    this._onfocusHandler = Function.createDelegate(this, this._onFocus);
    this._onblurHandler = Function.createDelegate(this, this._onBlur);

    $addHandlers(this.get_element(),
                 { 'focus' : this._onFocus,
                   'blur' : this._onBlur },
                 this);

    this.get_element().className = this._normalCssClass;
}

function IntroAjax$FocusBehavior$dispose()
{
    $clearHandlers(this.get_element());
    IntroAjax.FocusBehavior.callBaseMethod(this, 'dispose');
}

function IntroAjax$FocusBehavior$_onFocus(e)
{
    if (this.get_element() && !this.get_element().disabled)
        this.get_element().className = this._highlightCssClass;
}

function IntroAjax$FocusBehavior$_onBlur(e)
{
    if (this.get_element() && !this.get_element().disabled)
        this.get_element().className = this._normalCssClass;
}

function IntroAjax$FocusBehavior$get_highlightCssClass()
```

```
{
    return this._highlightCssClass;
}

function IntroAjax$FocusBehavior$set_highlightCssClass(value)
{
    if (this._highlightCssClass !== value)
    {
        this._highlightCssClass = value;
        this.raisePropertyChanged('highlightCssClass');
    }
}

function IntroAjax$FocusBehavior$get_normalCssClass()
{
    return this._normalCssClass;
}

function IntroAjax$FocusBehavior$set_normalCssClass(value)
{
    if (this._normalCssClass !== value)
    {
        this._normalCssClass = value;
        this.raisePropertyChanged('normalCssClass');
    }
}

// Register the class
IntroAjax.FocusBehavior.registerClass('IntroAjax.FocusBehavior',
       Sys.UI.Behavior);

Sys.Application.notifyScriptLoaded();
```

The *IntroAjax.FocusBehavior* class derives from *Sys.UI.Behavior*—a Microsoft AJAX library provided class that sets the baseline for behaviors. Note the final call to the *notifyScriptLoaded* method on the *Sys.Application* class. It serves to notify the ASP.NET AJAX client infrastructure that a required script has been loaded.

Note When writing a JavaScript class, you can optionally define a JSON serializer to be used when an instance of the class is serialized—for example, if the instance of the class is used as an argument to a remote method call. (We'll cover remote scripting in Chapter 7, "Remote Method Calls with ASP.NET AJAX.")

```
IntroAjax.FocusBehavior.descriptor = {
    properties: [   {name: 'highlightCssClass', type: String},
                    {name: 'normalCssClass', type: String} ]
```

To support JSON serialization, you add a *descriptor* member defined as in the preceding code snippet. The *properties* member refers to an array of pairs made by property name and type.

Using the Focus Extender Control

To use the sample extender in an ASP.NET page, you first register the control using the *@Register* directive:

```
<%@ Register Namespace="IntroAjax.Controls" TagPrefix="x" %>
```

The host page must also have a script manager just as any other ASP.NET AJAX page. Imagine a sample page with three text boxes and a button that you want to change style when focused. Here is what the code looks like:

```
<asp:ScriptManager ID="ScriptManager1" runat="server" />
<table>
   <tr>
      <td><asp:Label runat="server" ID="Label1">Name</asp:Label></td>
      <td><asp:TextBox ID="TextBox1" runat="server" /></td>
    </tr>
    <tr>
       <td><asp:Label runat="server" ID="Label2">Phone</asp:Label></td>
       <td><asp:TextBox ID="TextBox2" runat="server" /></td>
    </tr>
    <tr>
       <td><asp:Label runat="server" ID="Label3">E-mail</asp:Label></td>
       <td><asp:TextBox ID="TextBox3" runat="server" /></td>
    </tr>
</table>
<asp:Button runat="server" ID="Button1" Text="Submit Form" />
```

You extend these controls through the focus extender using the following ASP.NET markup:

```
<x:FocusExtender ID="FocusExtender1" runat="server"
          NormalCssClass="Normal"
          HighlightCssClass="HighLight"
          TargetControlID="TextBox1" />
<x:FocusExtender ID="FocusExtender2" runat="server"
          NormalCssClass="Normal"
          HighlightCssClass="HighLight"
          TargetControlID="TextBox2" />
<x:FocusExtender ID="FocusExtender3" runat="server"
          NormalCssClass="Normal"
          HighlightCssClass="HighLight"
          TargetControlID="TextBox3" />
<x:FocusExtender ID="FocusExtender4" runat="server"
          NormalCssClass="NormalButton"
          HighlightCssClass="HighLightButton"
          TargetControlID="Button1" />
```

As you can see, each instance of the extender is bound to a particular target control and sets properties. In particular, all text boxes share the same normal and highlighted CSS styles. Figure 5-1 shows the sample page in action.

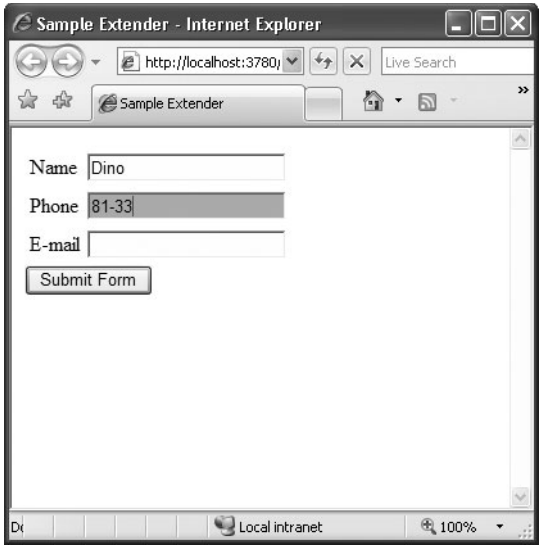

Figure 5-1 The focus extender in action

In the code snippet, *Normal* and *HighLight* are just CSS classes. Here's a possible definition:

```
<style type="text/css">
.Normal {
    background-color:#FFFFEE;
}
.HighLight {
    background-color:Orange;
}
.NormalButton {
    font-weight:normal;
    width:100px;
}
.HighLightButton {
    font-weight:bold;
    width:100px;
}
</style>
```

CSS classes are the preferred way to define visual properties for behaviors and client controls because they let you group multiple visual attributes in a single object and, subsequently, a single class property of type *string*.

Introducing the AJAX Control Toolkit

As mentioned, currently the vast majority of control extenders are compiled into an assembly known as the AJAX Control Toolkit. This assembly is a separate download and must be registered with any applications in which you plan to use it. The ACT project is an open-source project that results from a joint effort of Microsoft and the ASP.NET community. You can learn

more about the project by visiting the home page of the project on the CodePlex Web site. The exact URL is *http://www.codeplex.com/AtlasControlToolkit*.

If interested, you can become a contributor to the project and have your work highlighted. The main purpose of the ACT is to provide a collection of controls and extenders that fully benefit from the ASP.NET AJAX programming model and provide free and effective Web components to developers. The ACT is expected to remain and evolve separately from ASP.NET AJAX. However, chances are that, in the future, part of the contents of the ACT assembly will be merged with the ASP.NET core binaries. Time will tell.

As of today, you have two main options for using ACT controls. You can link the full assembly to your application, or you can incorporate some of the controls as source code in your project. Before you do this, though, make sure you read the license page at *http://www.codeplex.com/AtlasControlToolkit/Project/License.aspx*.

Get Ready for the Toolkit

To download the latest bits of the ACT, visit the project's home page on CodePlex: *http://www.codeplex.com/AtlasControlToolkit*. Once it is downloaded, the Toolkit looks like a ZIP file that you click to unpack and install any contained files.

The ACT Project

At the end of the setup, you have a Microsoft Visual Studio 2005 ASP.NET project on your hard disk. If you run the project, you should see something like the screen shown in Figure 5-2.

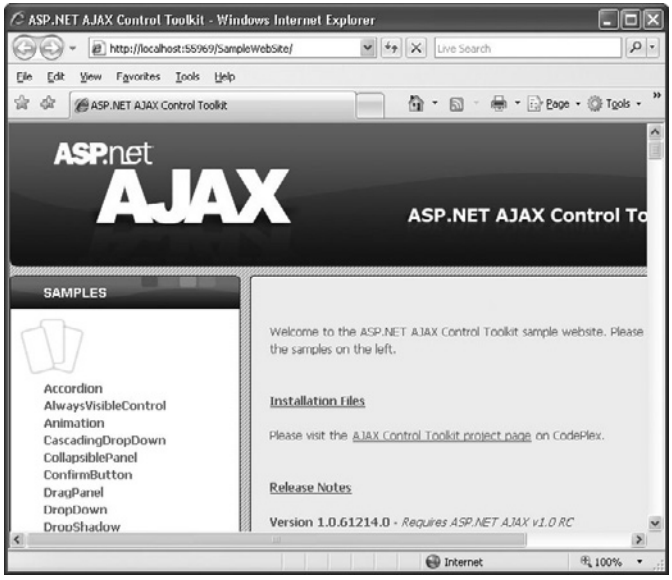

Figure 5-2 The sample Web site built to illustrate the facilities in the ACT

If you take a tour of the project files, you'll see that the ACT is a collection of ASP.NET AJAX controls and extenders plus many examples that illustrate their functionality and tests. In addition, the ACT contains a development kit to simplify the creation and re-use of your own custom controls and extenders. The development kit includes Visual Studio 2005 templates for Visual Basic and Visual C# to let developers write their own controls and extenders. In addition, it features a library of helper classes to speed up the creation of controls and extenders.

Adding ACT Components to the Toolbox

So the ACT contains a number of controls and extenders that you might want to compose and combine in your ASP.NET pages. The Visual Studio 2005 toolbox appears to be the perfect place to list all these components so that developers can pick up exactly the one they need and paste it to the current page. To add ACT components to the toolbox, you first create a new tab in the toolbox. You right-click on the toolbox surface and select Add Tab from the context menu. Once the new tab is added, you can give it any name you want—for example, AJAX Control Toolkit.

Next, you populate the tab with all components in the ACT assembly. To do this, you right-click the toolbox area below the newly added tab and select Choose Items. The standard dialog box that lists .NET assemblies and COM components will show up. Browse to the location where you installed the ACT, and select the *AjaxControlToolkit* assembly from the *Bin* folder of the SampleWebsite folder. That's all there is to it. When you're done, your toolbox will magically look like the one shown in Figure 5-3.

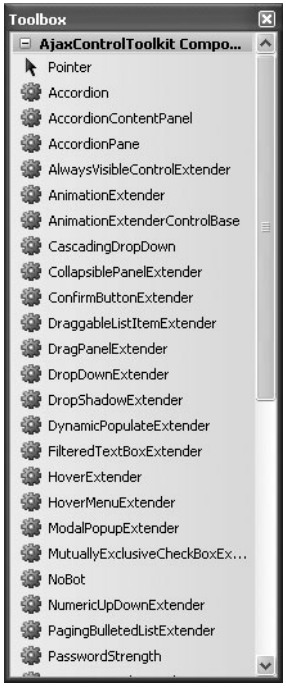

Figure 5-3 Creating a new tab in the Visual Studio 2005 toolbox to host the ACT components

Note In the ACT, you find a few server controls (such as *TabContainer*) and many more extenders (such as *RoundedCorners*, *DropShadow*, and *TextBoxWatermark*). What's the difference between controls and extenders? A *control* provides a closed set of functionalities and a well-known, fixed behavior that you can rule only through properties and events. An *extender* is a (mostly) client-side behavior that can be attached to a variety of controls. For example, you can attach the *DropShadow* extender to a *TextBox* as well as to a *Panel* and, in both cases, the target control renders out with a shadow. In spite of such logical differences, both server controls and extenders are implemented in the same way—both are, in the end, ASP.NET server controls.

Registering ACT Components with the Page

Although the ACT is a native part of the ASP.NET AJAX framework and a constituent part of the next ASP.NET platform, as of today it is simply an external library. For this reason, you need to explicitly register the ACT with each and every page where you happen to use any of its controls or extenders. You use the *@Register* directive, as shown here:

```
<%@ Register Assembly="AjaxControlToolkit"
             Namespace="AjaxControlToolkit"
             TagPrefix="act" %>
```

If you don't like the idea of repeating the same code for all the pages of a given application, you can take the same shortcut that the ASP.NET team took for avoiding the same burden with ASP.NET AJAX controls.

In fact, the same ASP.NET AJAX core library discussed here is also an external library to the ASP.NET platform and would need explicit registration with each page. In ASP.NET 2.0, you can use the following configuration script to enable a *@Register* directive for all pages of the application:

```
<pages>
   <controls>
      <add tagPrefix="asp"
           namespace="System.Web.UI"
           assembly="System.Web.Extensions, ..." />
   </controls>
</pages>
```

For example, all controls in the *System.Web.UI* namespace are associated with the *asp* tag prefix and don't need an explicit *@Register* directive in the pages where they're used. You can add a similar block to the *web.config* file for ACT controls as well:

```
<pages>
   <controls>
      <add namespace="AjaxControlToolkit"
           assembly="AjaxControlToolkit, ..."
           tagPrefix="act" />
      ...
   </controls>
</pages>
```

As you may already know, the string you choose as the tag prefix is arbitrary. For the sake of clarity, though, it is not recommended that you choose the *asp* prefix as well. The *asp* prefix should be reserved for system controls.

What's in the AJAX Control Toolkit

The ACT is a library designed to extend the capabilities of the ASP.NET AJAX framework. Its control set might change in the future and could be further extended or cut down. You can check the progress of the project from the aforementioned CodePlex site. At the time of this writing, the ACT contains a fair number of extenders and controls, but they certainly don't address every possible development scenario.

Extenders in the ACT

Table 5-1 lists the components currently available in the ACT. Note that the full name of the extender class contains an "Extender" suffix that I omitted in the table for brevity. So, for example, there will be no *CollapsiblePanel* component in the ACT assembly, but you will find a *CollapsiblePanelExtender* control instead.

Table 5-1 Extenders in the ACT

Component	Description
AlwaysVisibleControl	Pins a control to a corner of the page, and keeps it floating over the page background as the user scrolls or resizes the page. You use this extender to make sure that, say, a given panel shows up at the top-left corner of the page regardless of the scroll position or the size of the browser window.
Animation	Provides a specialized framework for adding animation effects to controls hosted in ASP.NET pages. You associate client-side events of the target control with one or more of the predefined animation effects.
AutoComplete	Associated with a text box, provides a list of suggestions for the text to type in the field.
Calendar	Attached to a text box, the extender provides client-side date-picking functionality with customizable date format and pop-up control.
CascadingDropDown	Associated with a *DropDownList* control. This extender automatically populates the list with data retrieved from a Web service method. The nice thing about this extender is that you can create a hierarchy of drop-down lists and have the extender automatically populate child drop-down lists based on the current selections in any of the previous lists in the heirarchy, if any.
CollapsiblePanel	Adds collapsible sections to a Web page. This extender can be used only with panel controls—that is, with the ASP.NET *Panel* control or any class derived from it. You let the extender know which panel in the page acts as the header and which panel provides the contents that collapse and expand.

Table 5-1 Extenders in the ACT (Continued)

Component	Description
ConfirmButton	Associated with a button control. This extender adds a confirmation JavaScript dialog box to the click event of the button. The extender is supported on any class that implements the *IButtonControl* interface, including *Button*, *LinkButton*, and *ImageButton*.
DragPanel	Associated with panel controls. This extender adds drag-and-drop capabilities so that you can move the panel around the page. You can specify the contents to move as well as the handle that, if pressed, triggers the dragging operation.
DropDown	The extender provides a mouse-over link to open a drop-down panel.
DropShadow	Adds drop shadows to any control available on the page. With this extender, you can specify the opacity and width of the shadow.
DynamicPopulate	Updates the contents of a control with the result of a Web service or page method call.
FilteredTextBox	Lets users enter text in a *TextBox* control that matches a given set of valid characters.
HoverMenu	Displays the contents of an associated panel control when the mouse hovers next to a given control. You can associate this extender with any ASP.NET control. The extender works as a kind of specialized and extremely flexible ToolTip.
MaskedEdit	Lets users enter text in a *TextBox* control according to a given input layout.
ModalPopup	Associated with a control that can fire a client-side *onclick* event (typically, buttons and hyperlinks), this extender implements a classic modal dialog box without using HTML dialog boxes. Basically, it displays the contents of a given panel and prevents the user from interacting with the rest of the page.
MutuallyExclusiveCheckBox	Associated with *CheckBox* controls, this extender lets you define logical groups of check boxes so that users can check only one in each group.
NoBot	Applies some *anti-bot* techniques to input forms. Bots, or robot applications, are software applications that run automated tasks over the Internet. For example, bots are used to fill input forms and submit ad hoc values.
NumericUpDown	Associated with text box controls, this extender allows you to click automatically displayed buttons to enter the next/previous value in the field. It works with numbers, custom lists, and Web service methods.
PagingBulletedList	Associated with *BulletedList* controls, this extender groups all items bound to the list and organizes them in client-side sorted pages.
PasswordStrength	Associated with text box controls used to type a password, this extender provides visual feedback on the strength of the password being typed.

Table 5-1 Extenders in the ACT (Continued)

Component	Description
PopupControl	Transforms the contents of a given panel into a pop-up window without using HTML dialog boxes. You can associate this extender with any control that can fire any of the following client-side events: *onfocus*, *onclick*, and *onkeydown*.
ResizableControl	Attaches to any page element, and allows the user to resize the element using a handle placed at the lower-right corner of the control.
RoundedCorners	Adds a background panel to any ASP.NET control so that the control appears with rounded corners. The overall height of the original control changes slightly.
Slider	Extends a *TextBox* control with a slider user interface.
TextBoxWatermark	Associated with *TextBox* controls. This extender adds sample or prompt text, called a "watermark," that illustrates the type of text the user is expected to enter in the field. For example, the watermark might say, "Type your name here." The watermark text disappears as soon as the user starts typing and reappears as the text box becomes empty.
ToggleButton	Associated with *CheckBox* controls. This extender enables you to use custom images to render the check buttons. You can use different images to indicate the selected and cleared states.
UpdatePanelAnimation	Plays animations during key steps of a partial update. You can use the extender to animate the page both while the panel is being updating and when the update has completed.
ValidatorCallout	Works on top of ASP.NET validators, and improves their user interface. In particular, the extender displays a yellow callout with the error message.

As you can probably guess, some extenders listed in the table require rich browser capabilities, whereas others are just a smart piece of JavaScript code attached to a block of markup elements. Note that all these features work in a cross-browser way. I'll return to each of the aforementioned extenders with code samples and more details in a moment.

Controls in the ACT

Along with all the extenders listed in Table 5-1, the ACT also supplies a few traditional server controls with rich capabilities: the *Accordion*, *Rating*, *ReorderList* and *TabContainer* controls.

The *Accordion* control allows you to provide multiple collapsible panes and display only one at a time. When the user clicks a new pane, the currently displayed pane is collapsed to leave room for the new one.

The *Rating* control provides an intuitive user interface to let users select the number of stars that represents their rating of a given subject. The control is the wrapped-up version of the user interface that several Web sites provide to let users rate published items.

A data-bound control, *ReorderList*, allows its child elements to be reordered on the client using drag-and-drop functionality. To move an item in the list, the user drags the item's handle up to its new position. At the end of the operation, the control posts back so that the new status of the data source can be recorded.

Finally, the *TabContainer* control is a purely client-side container of tabbed forms.

Let's first get to know more about these controls and then move on to discuss extenders.

The *Accordion* Control

Collapsible panels are a frequent feature in modern and cutting-edge Web sites. They allow you to display a short highlight—the header—and keep more text hidden and available on demand. The *CollapsiblePanel* extender (discussed later) allows you to hide and display any block of markup. But what if you need to build a sort of hierarchy of panels?

The *Accordion* control allows you to group multiple collapsible panels in a single control, and it manages the collapsed/expanded state of each panel so that only one can be expanded at a time.

Generalities of the *Accordion* Control

The *Accordion* control contains a collection of child *AccordionPane* controls, each of which features a template property to define header and content. Each pane can include any HTML, ASP.NET, or ASP.NET AJAX markup you want.

Properties of the Control

Table 5-2 lists the key properties of the *Accordion* control.

Table 5-2 Properties of the *Accordion* Control

Property	Description
AutoSize	The value assigned to this property indicates how the control will determine its actual size.
ContentCssClass	Gets and sets the CSS class used to style the content of child panes.
FadeTransitions	Indicates whether to use a fade-out transition effect while hiding the current pane. This property is set to *false* by default.
FramesPerSecond	Gets and sets the number of frames per second used in the transition animation for the newly selected pane. This property is set to *30* frames by default.
HeaderCssClass	Gets and sets the CSS class used to style the header of child panes.
Panes	Returns the collection of child panes.
SelectedIndex	Gets and sets the currently expanded pane.
TransitionDuration	Gets and sets the number of milliseconds to animate the transition. This property is set to *250* by default.

As you can see, the control has no visual properties except for the ASP.NET base properties defined on the parent *WebControl* class (such as *BackColor*, *ForeColor*, and so on). In particular, there's no style property for child panes. To style the header and content of child panes, you use CSS classes. Note that if the control worked through regular postbacks, you would probably have had *Style* objects instead of CSS class properties, as in many other classic ASP.NET controls.

Animating the Control

The *Accordion* control supports animation in two different contexts. First, it might optionally fade out the content of the current pane when a new one is selected. This effect is controlled by the *FadeTransitions* Boolean property. The effect is not very visible on small-sized content panes. Note also that the fade animation is tightly coupled with the transition animation that slowly rolls down the content of the new pane.

This second form of animation is controlled by the *TransitionDuration* and *FramesPerSecond* properties. If this latter effect is disabled, you won't see any fade-out effect regardless of the setting of the *FadeTransitions* property.

To skip transition animation completely and obtain a quick swap of panes, you set both *FramesPerSecond* and *TransitionDuration* to *0*. If you do so, also set *FadeTransitions* to *false* to save the control unnecessary tasks that will produce no observable effects.

Sizing the Control

The size of the *Accordion* control is clearly determined by any container element as well as the content of the various panes. This means that the overall size of the accordion might vary with the selected pane. In this case, other elements in the page might be pushed up or down. The *AutoSize* property helps to keep the overall size of accordion under control.

The *AutoSize* property takes its values from the *AutoSize* enumeration, whose values are *None* (default), *Limit*, and *Fill*. When the *AutoSize* property is set to *None*, the accordion modifies its size freely following the size of the selected pane.

When you set *AutoSize* to *Limit*, the accordion never grows larger than the value specified by its *Width* and *Height* properties. The *Accordion* control inherits these properties from its base class. The *Height* and *Width* properties are both ignored if *None* is set. If the content to display is larger than the designated size of the accordion, scrollbars are employed to let users see all of it.

With the *Fill* value set, the accordion stays exactly within the bounding box delimited by the *Width* and *Height* properties. In this case, the content will be expanded or made scrollable if it isn't the right size. Expanding the contents just means making the pane larger.

Using the *Accordion* Control

The accordion consists of a number of child panes, each of which is an instance of the *AccordionPane* class. You add panes using markup and retrieve the collection of panes programmatically using the *Panes* collection property.

The *Accordion* Control in Action

The following code demonstrates the usage of the *Accordion* control:

```
<div style="width:300px;">
   <act:Accordion ID="Accordion1" runat="server" Height="400px"
        SelectedIndex="0"
        ContentCssClass="accordionContent"
        HeaderCssClass="accordionHeader"
        FadeTransitions="true"
        AutoSize="Fill">
      <Panes>
         <act:AccordionPane ID="AccordionPane1" runat="server">
            <Header>One</Header>
            <Content>This is the first pane</Content>
         </act:AccordionPane>
         <act:AccordionPane ID="AccordionPane2" runat="server">
            <Header>Two</Header>
            <Content>
               <div style="height:400px">
               This is the second pane</div>
            </Content>
         </act:AccordionPane>
         <act:AccordionPane ID="AccordionPane3" runat="server">
            <Header>Three</Header>
            <Content>This is the third pane</Content>
         </act:AccordionPane>
         <act:AccordionPane ID="AccordionPane4" runat="server">
            <Header>Four</Header>
            <Content>This is the fourth pane</Content>
         </act:AccordionPane>
      </Panes>
   </act:Accordion>
</div>
```

You add an *<act:AccordionPane>* element for each collapsible panel you want to display. Each pane consists of a *<header>* and *<content>* template.

The sample accordion is embedded in a fixed-width * * tag. The outermost container determines the width and height of the accordion. The *AutoSize* property set to *Fill* forces the accordion to cover and fill the whole bounding box. Figure 5-4 shows the results.

Note that the content of the second pane is set to a height of 400 pixels; the accordion itself, though, can't be taller than 400 pixels and can't grow uncontrolled because of the *AutoSize*

setting. As a result, the second pane displays a scrollbar to let you see the content that exceeds that limit.

Figure 5-4 The *Accordion* control in action

Accordion Panes

An accordion pane is a simple Web control named *AccordionPane*. It has a *Header* template property that you use to define the highlighting of the panel—that is, the portion of the panel that is visible also when it is collapsed. You can style the header using the settings in the CSS class defined by the *HeaderCssClass* property.

Likewise, the content of the pane is defined by the *Content* template property and styled using the settings in the CSS class referenced by the *ContentCssClass* property.

The *Rating* Control

The satisfaction of users is one of the key metrics used to determine the success of a Web site. And how do you recognize the satisfaction level of a user? Many Web sites kindly ask users to provide their feedback through ad hoc panels scattered through the pages. The most common user-interface pattern for user feedback is a ratings system. The Web site shows a fixed number of stars and lets users click those stars to rate a given feature (usually, more stars clicked indicates a higher level of satisfaction).

Most ASP.NET developers can easily arrange a rating mechanism that works through classic postbacks. The ACT *Rating* control, on the other hand, provides a standard rating mechanism that works over callbacks.

Generalities of the *Rating* Control

The output of the control consists of a repeated <*span*> tag that is decorated through a set of CSS classes. Each <*span*> tag represents a "star" in the rating system and is styled according to the status it represents. Figure 5-5 gives you an idea of the *Rating* control.

Figure 5-5 The *Rating* control in action

Properties of the Control

Table 5-3 lists the key properties of the *Rating* control.

Table 5-3 Properties of the *Rating* Control

Property	Description
AutoPostBack	Indicates whether the control will post back whenever the user rates a given associated content.
CurrentRating	Gets and sets the current value rendered by the control. The default value is *3*.
EmptyStarCssClass	Gets and sets the CSS class to render an unselected star.
FilledStarCssClass	Gets and sets the CSS class to render a selected star.
MaxRating	Gets and sets the maximum value that can be rated through the control. The default value is *5*.
RatingAlign	Indicates the alignment of the stars. The default is *horizontal*.
RatingDirection	Indicates the orientation of the stars. The default is left to right if the *RatingAlign* value is horizontal and top to bottom if *RatingAlign* is vertical.
ReadOnly	Indicates whether the control accepts user input.

Table 5-3 Properties of the *Rating* Control (Continued)

Property	Description
StarCssClass	Gets and sets the CSS class to style the whole control.
Tag	A string to pass to the server-side code that handles the user's click.
WaitingStarCssClass	Gets and sets the CSS class to render selected stars during a server post-back following a user update.

The markup generated by the *Rating* control looks like the following code:

```
<div id="Rating1" style="float: left;">
    <span id="Rating1_Star_1" class="ratingStar filledRatingStar"
         style="float:left;"> </span>
    <span id="Rating1_Star_2" class="ratingStar filledRatingStar"
         style="float:left;"> </span>
    ...
</div>
```

As you can see, each "star" is characterized by a unique ID and is represented with a <*span*> tag set to the empty string. In Figure 5-5, though, each star is clearly an image. How is that possible?

Styling the Control

The *Rating* control supports four different CSS classes. *StarCssClass* defines the style for the entire control. *WaitingStarCssClass* defines the style to be used when the control is posting back. *EmptyStarCssClass* and *FilledStarCssClass* define the style for the <*span*> tag. You can use either an image to fill the tag or a contrasting background color. Let's consider the following CSS classes:

```
.filledRatingStar
{
    background-color: #2E4d7B;
}
.emptyRatingStar
{
    background-image: url(images/NotSelected.png);
}
```

These classes produce an effect like the second rating object shown in Figure 5-5. The selected part is rendered as a gauge bar; the unselected part is rendered using empty stars. By editing the CSS classes, you can choose the "star" images to meet the expectations of the users of your application.

Using the *Rating* Control

The *Rating* control is relatively simple to use. It only requires you to set a few properties—the CSS classes—and it has no child elements or templates.

The *Rating* Control in Action

The following code demonstrates the usage of the *Rating* control:

```
<h2>Rate this item:</h2>
<div>
    <act:Rating ID="Rating1" runat="server"
         CurrentRating="3"
         MaxRating="10"
         StarCssClass="ratingStar"
         WaitingStarCssClass="savedRatingStar"
         FilledStarCssClass="filledRatingStar"
         EmptyStarCssClass="emptyRatingStar"
         OnChanged="Rating_Changed" />
</div>
```

The *MaxRating* property determines the number of stars to render. Of these, the first stars (or the last ones, depending on the direction) up to the value specified by *CurrentRating* are styled using the class name specified by *FilledStarCssClass*. The remaining stars are styled using the attributes set by *EmptyStarCssClass*.

Specifying valid CSS class names is key. If you omit, say, the empty-star style, the corresponding <*span*> tag will be rendered as is, without any graphical adjustments. Because the <*span*> tag is set to the empty string, no visible output will be generated.

The Eventing Model

The *Rating* control injects into the client browser a piece of script code that does two main things. First, it captures mouse movements over the bounding box of the control. Second, it handles the user's clicking.

As the user moves the mouse over the unselected stars, the script code automatically toggles the class name of the underlying <*span*> tag to give you an idea of the interface if you select the given number of stars.

To change the current value, you just click on the star that represents the new value. For example, in an rating system with five stars, you click on the fourth star to set a rating of 4. When this happens, the *Rating* control makes an out-of-band call to the server and raises the *Changed* event:

```
protected void Rating_Changed(object sender, RatingEventArgs e)
{
    // Perform any significant server-side action
    // such as storing the new value to a database
}
```

The *RatingEventArgs* class contains three main properties: *Tag*, *Value*, and *CallbackResult*. *Tag* is a cargo property used to carry any custom string from the client to the server. *Value* indicates the currently selected value in the control. *CallbackResult*, on the other hand, is a string

property you can set on the server with any information you want to bring back to the client. For example, you can use the *Changed* event to store the rating value to a database and use the *CallbackResult* string to return an error message if the operation fails.

The *ReorderList* Control

Data-bound lists of data are commonly displayed to Web users and, most of the time, are read-only, immutable lists. There might be situations, though, in which the end user might want to reorder a displayed list of items. A good example is a page on which the user can select multiple cities in the world to be informed about the current weather. The cities appear in a given order on the page, but the user might want to change the order at some point.

As a developer, you can add a *Move Up* button to the page and have the user click three times to bring the last city to the top of a list with four cities. Although this approach is functionally effective, it is certainly not very user friendly. A more natural approach would be to enable the user to simply select the item of a given city and drop it to the desired new location.

This is exactly what the *ReorderList* control allows you to do. Let's learn more about this new ASP.NET AJAX control.

Generalities of the *ReorderList* Control

The contents of the *ReorderList* control are expressed through a series of templates that you use to indicate the structure of each data-bound item, the user interface for the drag handle, and the user interface for the insertion point where the dragged item is being moved. More properties and events, though, interact to form the programming interface of the control.

Properties of the Control

Table 5-4 lists the key properties of the *ReorderList* control. In particular, the list describes the properties that can be set through attributes in an ASP.NET Web page. The control, in fact, derives from *DataBoundControl* and implements the *IRepeatInfoUser* interface—the typical interface for controls that support a variety of alignments and layouts, such as *CheckBoxList*. The base class and the interface include a bunch of additional properties that relate to data binding and control layout.

Table 5-4 Properties of the *ReorderList* Control

Property	Description
AllowReorder	Indicates whether the control supports drag-and-drop reordering. This property is automatically set to *true* if a reorder template is specified.
DataKeyField	Indicates the name of the data-bound source that operates as the primary key field.
DataSourceID	Indicates the ID of the data source control used to populate the control.
DragHandleAlignment	Indicates the position of the drag handle with respect to the item to drag. It can have any of the following values: *Top*, *Left*, *Bottom*, or *Right*.

Table 5-4 Properties of the *ReorderList* Control (Continued)

Property	Description
DragHandleTemplate	The template for the drag handle that the user clicks to drag and reorder items.
EditItemTemplate	The template used to show that a row is in edit mode.
EmptyListTemplate	The template used to show that the list has no data. This item is not data bindable.
InsertItemTemplate	The template used to add new items to the list.
ItemInsertLocation	When the *InsertItemTemplate* property is used to add a new item to the displayed list, this property indicates where the item has to be inserted. Feasible values are *Beginning* and *End*.
ItemTemplate	The template for any items in the list.
PostBackOnReorder	Indicates whether the control has to post back at the end of a reorder operation.
ReorderTemplate	The template used to show the drop location during a reorder operation. This template is not data bindable.
SortOrderField	The field, if any, that represents the sort order of the items.

The key properties are *ItemTemplate*, *DragHandleTemplate*, and *ReorderTemplate*. *ItemTemplate* allows you to populate the list control with any information and layout you need. *DragHandle-Template* defines the graphical elements that users need to identify when they want to reorder the list. Finally, the *ReorderTemplate* defines the template of the visual feedback that is shown to the user while the operation is occurring.

Events of the Control

The *ReorderList* control is primarily a data-bound control with a rich user interface largely made up of templates. This means that the control can include a number of child controls that might trigger postbacks and cause the contents of the control to change.

It is not surprising, then, to find a long list of events associated with this control. These events are detailed in Table 5-5.

Table 5-5 Events of the *ReorderList* Control

Event	Description
CancelCommand	Occurs when a button with the *CommandName* of "Cancel" is clicked from within the control.
DeleteCommand	Occurs when a button with the *CommandName* of "Delete" is clicked from within the control.
EditCommand	Occurs when a button with the *CommandName* of "Edit" is clicked from within the control.
InsertCommand	Occurs when a button with the *CommandName* of "Insert" is clicked from within the control.
ItemCommand	Occurs when a button is clicked from within the item template of a row.

Table 5-5 Events of the *ReorderList* Control (Continued)

Event	Description
ItemCreated	Occurs when an item row is created.
ItemDataBound	Occurs when an item row is bound to its data.
ItemReorder	Occurs when an item row is moved to a new location at the end of a reorder operation.
UpdateCommand	Occurs when a button with the *CommandName* of "Update" is clicked from within the control.

Most command events are related to the capabilities of the bound data source. For example, the *InsertCommand* event occurs if the control features a proper template with controls to capture data and invoke an *Insert* command on the bound data source control. Here's a quick example:

```
<act:ReorderList ...>
    ...
    <InsertItemTemplate>
        <div>
            <asp:TextBox ID="TextBox1" runat="server"
                Text='<%# Bind("Title") %>'></asp:TextBox>
            <asp:LinkButton ID="LinkButton1" runat="server"
                CommandName="Insert">Add</asp:LinkButton>
        </div>
    </InsertItemTemplate>
</act:ReorderList>
```

By clicking on the link button, you cause the *ReorderList* control to invoke the *Insert* command on the bound data source, if there is any. The new item, which will have a *Title* property, is added at the top or bottom of the data source based on the value of the *ItemInsertLocation* property.

The *ItemReorder* event is fired on the server before the new control is rendered back to the client with the new order of items. The event carries a *ReorderListItemReorderEventArgs* object with the following structure:

```
public class ReorderListItemReorderEventArgs : EventArgs
{
      public ReorderListItem Item { get; set; }
      public int NewIndex { get; set; }
      public int OldIndex { get; set; }
}
```

The *Item* property indicates the item being moved, whereas *NewIndex* and *OldIndex* specify the new and old positions (a 0-based index), respectively.

Using the *ReorderList* Control

Let's consider a sample page that makes use of the *ReorderList* control. As you'll see in a moment, the *ReorderList* control must be bound to a data source and embedded in an *UpdatePanel* control.

Configuring the *ReorderList* Control

The following code snippet demonstrates a sample reorder list control bound to an *ObjectDataSource* control:

```
<act:ReorderList runat="server" ID="list"
    DataSourceID="ObjectDataSource1">
    <ItemTemplate>
        <asp:Label ID="Label1" runat="server"
            Text='<%# Eval("lastname") %>' />
    </ItemTemplate>
    <ReorderTemplate>
        <asp:Panel ID="Panel2" runat="server" CssClass="reorderCue" />
    </ReorderTemplate>
</act:ReorderList>
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    TypeName="IntroAtlas.EmployeeManager"
    SelectMethod="LoadAll">
</asp:ObjectDataSource>
```

The sample control has no *DragHandleTemplate* set, which means that its user interface has no visible element to start dragging. In this case, the output looks like a bulleted list, and to start reordering you simply click and drag the text beside the bullet point, as shown in Figure 5-6.

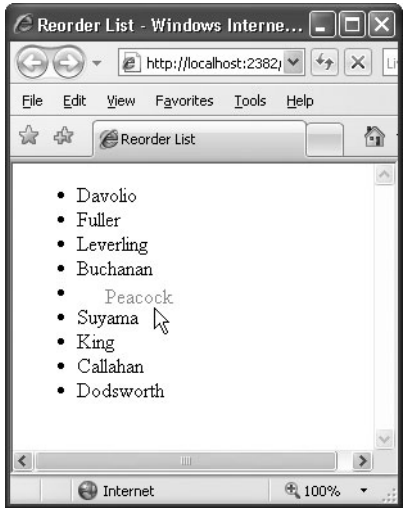

Figure 5-6 Reordering the items in a *ReorderList* control

Generally, the item being moved is rendered through a template that you specify via the *ReorderTemplate* property. In this case, the reorder template consists of the sole text of the item plus a cascading style. If you omit the template but still enable reordering through the *AllowReorder* Boolean property, the text of the item, rendered with a gray color, is used to give feedback to users about the ongoing operation.

Building a reorder list that allows items to be moved around is a breeze. However, a couple of essential issues still need to be properly addressed.

Reordering and Postback Events

The *ReorderList* control can be configurated to post back at the end of each drag-and-drop operation. To avoid a full page refresh, you need to wrap the *ReorderList* control in an *UpdatePanel* control. As explained in Chapter 4, in this way you guarantee that only the user interface of the list control is refreshed rather than the entire page being updated with a complete postback. The following few lines of markup can accomplish this for us:

```
<asp:UpdatePanel runat="server" ID="UpdatePanel1">
    <ContentTemplate>
        ...
        <!-- ReorderList goes here -->
        ...
    </ContentTemplate>
</asp:UpdatePanel>
```

Why does the control do a postback of its own? The *ReorderList* control attempts to persist the changes the user made so that if any postback occurs from within the page—say, a postback caused by the user clicking on another button in the page—the contents of the *ReorderList*, as modified via a drag-and-drop operation, are maintained. Note, in fact, that any changes made on the client should be persisted on the server; otherwise, they will be lost in the first subsequent page postback.

Persisting Reordered Items

If you run a page that contains a *ReorderList* control, you notice that invariably after a postback, any moved element is restored to its original location. Why is that so? At the end of the drop operation, the *ReorderList* control posts back, fires the *ItemReorder* event on the server, and rebuilds the list. The list is data bound, so unless you change something in the binding process, the list will be bound back to the same record set in the same old order.

Subsequently, a page that hosts a *ReorderList* control should wire up the *ItemReorder* event and make sure that the data source bound to the control properly reflects the changes generated on the client. The *ItemReorder* event has the following signature:

```
void OnItemReorder(object sender, ReorderListItemReorderEventArgs e)
```

In this event handler, you should do the real work of the reorder—that is, move the item from the old position to the new position in the data source used to populate the list. How you do this depends on the binding mechanism. If you opted for a data source control (for example, *ObjectDataSource*), you can try sorting on a given field, if any, that reflects the new order. If you set the data source assigning an *IEnumerable* object to the *DataSource* property, you can retrieve and modify this object to reflect the new order. For a *DataTable*, for example, this means swapping two rows.

Note The *ReorderList* can automatically perform server-side reorders if its *SortOrderField* property is set and if the data source can be sorted on that field. In addition, the type of the sort field must be *integer*.

The *TabContainer* Control

Multiple views are a common feature in most pages. They group information in tabs and let users click to display only a portion of the information available. In ASP.NET 2.0, the *MultiView* control provides an effective shortcut to this feature. But it requires a postback to update the page when the user selects a new tab. In the ACT, the *TabContainer* control provides an AJAX version of the multiview control.

Generalities of the *TabContainer* Control

The *TabContainer* control is made of a collection of tabs, each of which is represented by an instance of the *TabPanel* class. You can add and remove panels programmatically, as well as define them declaratively.

Properties of the Control

Table 5-6 lists the properties supported by the control.

Table 5-6 Properties of the *TabContainer* Control

Property	Description
ActiveTab	Returns a reference to the currently selected tab.
ActiveTabIndex	Gets and sets the 0-based index of the selected tab.
CssClass	Gets and sets the CSS class to use to style the control.
Height	Gets and sets the height of the tabs. The value of the property is expressed as a *Unit* value. This value doesn't include headers.
OnClientActiveTabChanged	Gets and sets the JavaScript code to be executed on the client when the user changes the selection.
Scrollbars	Gets and sets the desired support for scrollbars. This property is set to *Auto* by default. Feasible values come from the ASP.NET 2.0 *Scrollbars* type.
Tabs	Returns the collection of *TabPanel* objects that defines the user interface of the control.
Width	Gets and sets the width of the tabs. The value of the property is expressed as a *Unit* value.

In addition, the control fires the *ActiveTabChanged* event when the selected tab changes. The event is a mere notification, and the required delegate is *EventHandler*. No additional information is passed along with the event.

Properties of Tab Panels

The *TabPanel* class represents an individual tab in the container. Each tab defines its header either as plain text or as a template. Likewise, a tab features a template to let developers specify its content. The most recent tab should remain selected after a postback. Table 5-7 lists the properties of the *TabPanel* class.

Table 5-7 Properties of the *TabPanel* Class

Property	Description
ContentTemplate	Sets the contents of the tab.
Enabled	Indicates whether the tab should be displayed. The value of the property can be changed on the client.
HeaderTemplate	Gets and sets the template to use to define the header of the tab.
HeaderText	Gets and sets the text to display in the tab's header.
OnClientClick	JavaScript code to attach to the client-side *click* event of the tab.
OnClientPopulated	JavaScript code to run on the client when the tab has been fully populated.
OnClientPopulating	JavaScript code to run on the client when the tab is going to be populated.
Scrollbars	Gets and sets the desired support for scrollbars. This property is set to *Auto* by default. Feasible values come from the ASP.NET 2.0 *Scrollbars* type.

The JavaScript code you can attach to some client events can be either the name of function embedded in the host page or a string of JavaScript executable code.

Using the *TabContainer* Control

Let's consider a sample page that makes use of the *TabContainer* control. As you'll see in a moment, the markup required for a tab container is straightforward.

Configuring the *TabContainer* Control

The *TabContainer* tag maps its child tags to the *Tabs* collection of *TabPanel* objects. You add one <*TabPanel*> tag for each desired tab and configure it at will. Here's an example:

```
<act:TabContainer runat="server" ID="TabContainer1">
    <act:TabPanel runat="server" ID="TabPanel1" HeaderText="Your Tab">
        <ContentTemplate>
            <h3>Some text here</h3>
        </ContentTemplate>
    </act:TabPanel>
    ...
</act:TabContainer>
```

All tabs are given the same size, and you can control the size designation through the *Width* and *Height* properties of the container. The height you set refers to the body of tags and doesn't include the header.

Changing the Selected Tab

You can add some script code to run when the user selects a new tab. You can wrap up all the code in a page-level JavaScript function and bind the name of the function to the *OnClientActiveTabChanged* property of the tab container. The following code writes the name of the currently selected tab to a page element (originally, an ASP.NET *Label* control) named *CurrentTab*:

```
<script type="text/javascript">
    function ActiveTabChanged(sender, e)
    {
        var tab = $get('<%=CurrentTab.ClientID%>');
        tab.innerHTML = sender.get_activeTab().get_headerText();
    }
</script>
```

Note the usage of code blocks in JavaScript. In this way, the client ID of the label is merged in the script regardless of whether the page is a regular page or a content page (with a hierarchy of parent controls and naming containers). Figure 5-7 shows the control in action.

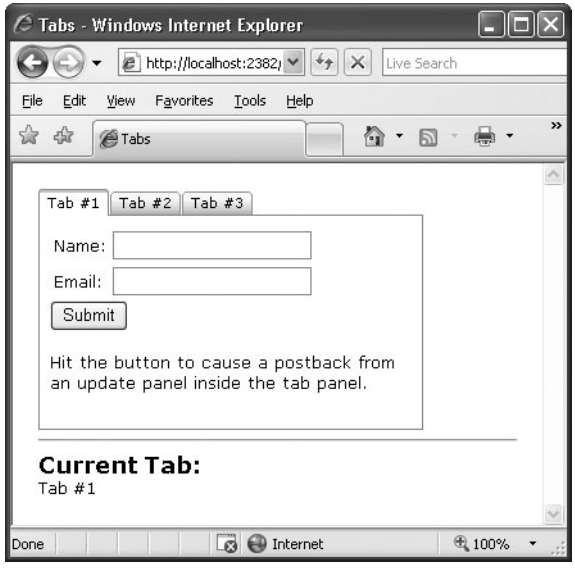

Figure 5-7 The *TabContainer* control in action

The Client-Side Object Model

As a full-fledged ASP.NET AJAX control, the *TabContainer* control exposes a client-side object model. In particular, there's a set of properties that represents the programming interface of the container and another set of properties for each tab panel.

The container features read/write properties—such as *activeTabIndex*, *activeTab*, *tabs*, and *scrollBars*—plus the *activeTabChanged* event.

The tab panel exposes read/write properties such as *enabled*, *headerText*, and *scrollBars* along with a read-only *tabIndex* property and a few events—*click*, *populating*, and *populated*.

AJAX Control Toolkit Extenders

In addition to finding full-fledged server controls such as *ReorderList* and *Accordion* in the ACT, you find a bunch of other server controls designed to extend existing controls on the page and provide them with new and additional behaviors. Existing extenders can be categorized into a few groups: panel, input, popup, user interface, animation, and button. Let's dig deeper into these groupings.

Panel Extenders

ASP.NET pages are full of blocks of markup that, ideally, users would love to move around, collapse if too large, and expand on demand. The perfect panel control in ASP.NET is, therefore, both draggable and expandable. Purposely, ASP.NET AJAX defines a few server-side behaviors that allow you to easily create collapsible sections and drag panels around the page.

The *CollapsiblePanel* Extender

The extender builds up a collapsible section in your pages by combining two panels—one acting as the content panel, and one being the expand/collapse controller. In its simplest form, the *CollapsiblePanel* extender looks like the following code sample:

```
<act:CollapsiblePanelExtender ID="CollPanel" runat="server"
        TargetControlID="ContentPanel"
        ExpandControlID="HeaderPanel"
        CollapseControlID="HeaderPanel" />
```

As usual, the *TargetControlID* property sets the target panel to expand or collapse. *ExpandControlID* and *CollapseControlID* indicate the panel to use to expand and collapse the content panel. Note the extreme flexibility of the component design—it might not make sense in all cases, but you can use different panels to control the expansion and collapsing of the content panel. In most cases, though, you'll be using the same header panel with an image button that changes according to the state of the content panel. The following code snippet shows a more complete usage for the extender:

```
<act:CollapsiblePanelExtender ID="cpe" runat="server"
         TargetControlID="CollapsibleCustomersPanelContent"
         ExpandControlID="CollapsibleCustomersPanel"
         CollapseControlID="CollapsibleCustomersPanel"
         Collapsed="true"
         ExpandDirection="Vertical"
         ImageControlID="ToggleImage"
         ExpandedImage="~/images/collapse.jpg"
         ExpandedText="Collapse"
         CollapsedImage="~/images/expand.jpg"
         CollapsedText="Expand" />
```

The *ImageControlID* indicates the *Image* control, if any, that if clicked causes the panel to expand or collapse. The *ExpandedImage* and *CollapsedImage* properties set the URL of the images to use to expand and collapse. Likewise, *CollapsedText* and *ExpandedText* set the ToolTip text for the image. *Collapsed* sets the state of the panel, whereas *ExpandDirection* indicates whether the panel expands horizontally or vertically. Figure 5-8 provides a view of the control in action.

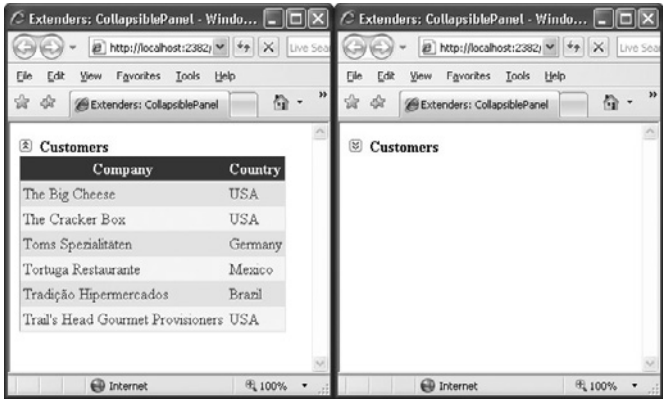

Figure 5-8 The *CollapsiblePanel* extender in action

The following code demonstrates a typical pair of *Panel* controls used with the extender:

```
<asp:Panel ID="CollapsibleCustomersPanel" runat="server">
    <asp:Image ID="ToggleImage" runat="server"
               ImageUrl="~/images/collapse.jpg" />
    <b>Customers</b>
</asp:Panel>
<asp:Panel ID="CollapsibleCustomersPanelContent" runat="server"
     Height="0" CssClass="collapsePanel">
    ...
</asp:Panel>
```

Unlike draggable panels, the header and content panels are distinct and are typically placed one after the next in the page layout. The extender panel is also *postback aware*, meaning that, on a client postback, it automatically records and restores its collapsed/expanded client state.

Note To avoid the initial flickering when a collapsible panel is displayed, make sure you properly style the panel that is going to be collapsed and expanded. This panel needs to have *Height=0* and the CSS *overflow* style set to *hidden*.

The *DragPanel* Extender

The *DragPanel* extender is one of the simplest extenders in the ACT. It has only two properties—one to indicate the panel to drag, and one to indicate the panel to use as the drag handle:

```
<act:DragPanelExtender ID="DragPanelExtender1" runat="server"
         TargetControlID="CustomerPanel"
         DragHandleID="CustomersDragHandle" />
```

As the name suggests, the *TargetControlID* property refers to the ID of the panel control in the page that is going to be moved. The *DragHandleID*, on the other hand, indicates the ID of the panel control that is used as the handle of the drag. In other words, to drag the target panel users drag and drop the handle panel. Although functionally distinct, the two panels are, in effect, logically correlated and rendered through nested tags:

```
<asp:Panel ID="CustomersPanel" runat="server" >
    <asp:Panel ID="CustomersDragHandle" runat="server">
        <div style="background-color:yellow">Customers</div>
    </asp:Panel>
    <asp:Panel runat="server">
        <asp:gridview runat="server" DataSourceID="ObjectDataSource1">
            <Columns>
                ...
            </Columns>
        </asp:gridview>
        <asp:ObjectDataSource ID="ObjectDataSource2" runat="server"
            TypeName="IntroAjax.CustomerManager"
            SelectMethod="LoadAll">
         </asp:ObjectDataSource>
    </asp:Panel>
</asp:Panel>
```

The target panel usually contains as a child the drag handle panel. In this way, you obtain the effect of moving the whole panel as if it were a Microsoft Windows window. (See Figure 5-9.)

Figure 5-9 The *DragPanel* extender in action

The *DropDown* Extender

The *DropDown* extender can be attached to virtually any ASP.NET control. Once attached to a control, the extender provides a mouse-over link to open a drop-down panel. The contents of the panel are entirely up to you—typically, arranged as a menu. The drop-down is activated by clicking the extended control with any mouse buttons.

```
<asp:Label ID="Label1" runat="server" Text="Move the mouse here" />
<asp:Panel ID="Panel1" runat="server" CssClass="ContextMenuPanel"
     Style="display:none;visibility:hidden;">
    <asp:LinkButton runat="server" ID="Option1" Text="I'm the first"
         CssClass="ContextMenuItem" OnClick="LinkButton1_Click" />
    <asp:LinkButton runat="server" ID="Option2" Text="I'm the second"
         CssClass="ContextMenuItem" OnClick="LinkButton1_Click" />
    <asp:LinkButton runat="server" ID="Option3" Text="I'm the third"
         CssClass="ContextMenuItem" OnClick="LinkButton1_Click" />
</asp:Panel>

<act:DropDownExtender runat="server" ID="DropDownExtender1"
     TargetControlID="Label1" DropDownControlID="Panel1" />
```

In the sample just shown, the drop-down user interface is a *Panel* that contains a list of link buttons. Link buttons are styled to look like menu items. Link and push buttons, and indeed embedded controls in general, operate normally. (See Figure 5-10.)

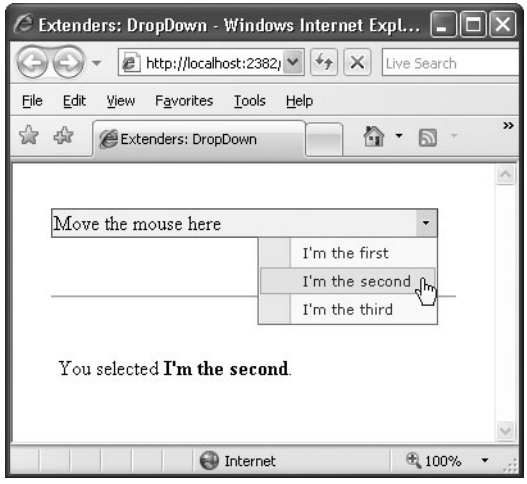

Figure 5-10 The *DropDown* extender in action

Button Extenders

Buttons are by far one of the most common elements in ASP.NET pages. However, as pages become functionally richer, additional features are required for buttons to stay in sync with users' expectations. ASP.NET AJAX provides a few extenders that apply to submit buttons and to the pseudo-buttons that form a *CheckBox* element.

The *ConfirmButton* Extender

Many times, a safe approach to responding to a user clicking a button is to ask the user for a confirmation for the operation she's going to start. A common solution for implementing this behavior entails that the ASP.NET page attach some script code to the button to pop up a JavaScript message box with a confirmation message. The *ConfirmButton* extender greatly simplifies this common task by making it declarative:

```
<asp:Button runat="server" ID="Button1" Text="Click me" />

<act:ConfirmButtonExtender ID="ConfirmButtonExtender1" runat="server"
        TargetControlID="Button1"
        ConfirmText="Are you sure you want to click this?\nReally sure?" />
```

The *ConfirmText* property specifies the text of the message box being displayed as the user clicks the button. Note that HTML entities can be used in the text, but by design no HTML formatting, such as <*b*> or <*i*>, can be used. You can, however, use entitized special characters. For example, you can use *&#10*; to break the line and continue the text on the next line. The reason for this lies in the JavaScript code for the *ConfirmButton*—it's using JavaScript's *window.confirm*. Providing HTML formatting makes no sense because the confirmation dialog box is basically a non-HTML Windows *MessageBox* call.

Internally, the *ConfirmButton* extender sets a handler for the *onsubmit* event of the form and swallows the event if the user doesn't confirm the operation. Only controls that implement the *IButtonControl* interface can be used with the extender, including *LinkButton* and *ImageButton* controls. (See Figure 5-11.)

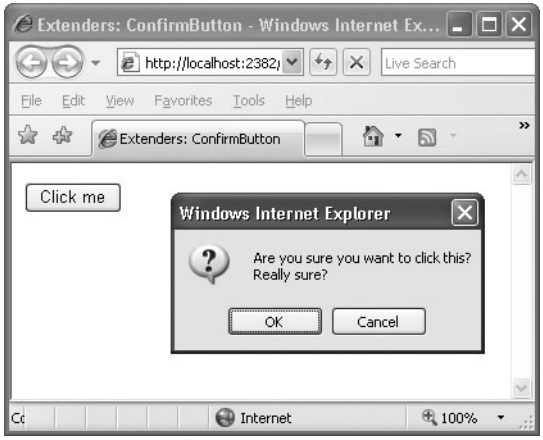

Figure 5-11 The *ConfirmButton* extender in action

The *MutuallyExclusiveCheckBox* Extender

The *MutuallyExclusiveCheckBox* extender can be attached to any ASP.NET *CheckBox* control to make it part of a group of logically related options. The extender implements a behavior that

looks a lot like a list of radio buttons—multiple options are available but only one can be chosen. So what's the point of having a mutually exclusive set of check boxes rather than a radio button list?

A list of radio buttons can be initially unselected, but once one option has been selected there's no way for the user to return to the initial state of having all options unselected. Returning to the original, completely unselected state is possible with the *MutuallyExclusiveCheckBox* extender.

The idea is that you group a number of check boxes under the same key. The extender then ensures that only one check box with the specified key can be selected at a time:

```
<h2>What Kind of Experience Do You Have with ASP.NET?</h2>
<asp:CheckBox runat="server" ID="chkBeginner" Text="Beginner" />
<asp:CheckBox runat="server" ID="chkIntermediate" Text="Intermediate" />
<asp:CheckBox runat="server" ID="chkExpert" Text="Expert" />

<act:MutuallyExclusiveCheckBoxExtender runat="server" ID="Mutual1"
    TargetControlID="chkBeginner"
    Key="AspNetExpertise" />
<act:MutuallyExclusiveCheckBoxExtender runat="server" ID="Mutual2"
    TargetControlID="chkIntermediate"
    Key="AspNetExpertise" />
<act:MutuallyExclusiveCheckBoxExtender runat="server" ID="Mutual3"
    TargetControlID="chkExpert"
    Key="AspNetExpertise" />
```

It can be argued that the same functionality could have been applied to radio buttons instead of check boxes. Using check boxes was the choice of developers, and it also provides a more consistent and expected user interface. However, you can re-implement the behavior to use JavaScript to allow the deselection of a radio button item.

The *ToggleButton* Extender

Check boxes are graphical HTML elements visually represented by a pair of little bitmaps (selected and unselected) plus companion text. Each browser can use its own pair of bitmaps, thus resulting in the check boxes having a slightly different look and feel. Most browsers, though, tend to represent check-box buttons as square embossed buttons.

The *ToggleButton* extender provides a way to simulate a check-box element that uses custom bitmaps. The extender is applied to a *CheckBox* control, and it replaces the control with a completely new markup block that uses custom images and provides the same behavior as a standard check box:

```
<act:ToggleButtonExtender ID="ToggleButtonExtender1" runat="server"
        TargetControlID="CheckBox1"
        ImageWidth="19"
        ImageHeight="19"
        UncheckedImageUrl="DontLike.gif"
        CheckedImageUrl="Like.gif" />
```

ImageWidth and *ImageHeight* properties indicate the desired size of the images. Note that these attributes are required. *UncheckedImageUrl* and *CheckedImageUrl* specify the images to use when the check box is selected or not selected.

Pop-up Extenders

Virtually every Web developer has a sort of love/hate relationship with pop-up windows. As a matter of fact, pop-up windows often greatly simplify a number of tasks—especially modal dialog boxes. One of the nasty things about HTML pop-up windows is that they are browser windows and require a page to navigate. The pop-up extenders that ASP.NET AJAX Extensions has to offer, on the other hand, do not require a new browser instance. Instead, they are limited to popping up the content of any panel you indicate, with or without modality.

The *HoverMenu* Extender

The *HoverMenu* extender is similar to the *PopupControl* extender and can be associated with any ASP.NET control. Both extenders display a pop-up panel to display additional content, but they do it for different events. The *HoverMenu*, in particular, pops up its panel when the user moves the mouse cursor over the target control. The panel can be displayed at a position specified by the developer. It can be at the left, right, top, or bottom of the target control. In addition, the control can be given an optional CSS style so that it looks like it is in a highlighted state. (See Figure 5-12.)

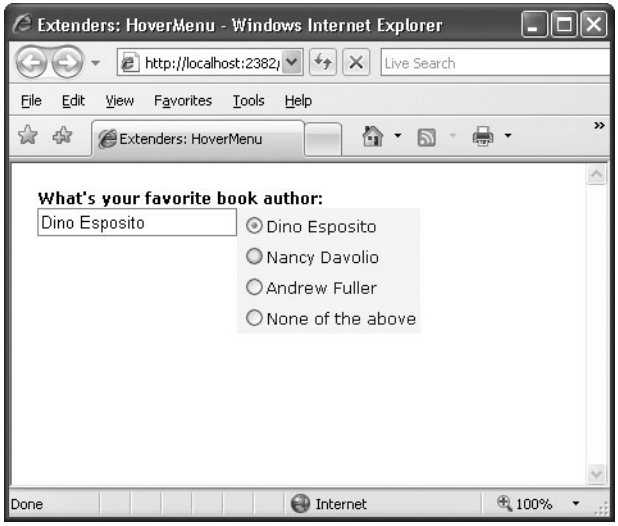

Figure 5-12 The *HoverMenu* extender in action

The *HoverMenu* extender is good for implementing an auto-display context menu for virtually every ASP.NET control instance and for providing tips to fill in some input fields. In Figure 5-12, for example, when the user hovers the cursor over the text box, a list of suggestions appears to simplify the work.

```
<asp:TextBox ID="TextBox1" runat="server" />

<asp:Panel ID="Panel1" runat="server" CssClass="popupMenu">
    <asp:RadioButtonList ID="RadioButtonList1" runat="server"
        AutoPostBack="true"
        OnSelectedIndexChanged="RadioButtonList1_SelectedIndexChanged">
        <asp:ListItem Text="Dino Esposito"></asp:ListItem>
        <asp:ListItem Text="Nancy Davolio"></asp:ListItem>
        <asp:ListItem Text="Andrew Fuller"></asp:ListItem>
        <asp:ListItem Value="" Text="None of the above"></asp:ListItem>
    </asp:RadioButtonList>
</asp:Panel>

<act:HoverMenuExtender ID="HoverMenu1" runat="server"
     TargetControlID="TextBox1"
     HoverCssClass="hoverPopupMenu"
     PopupControlID="Panel1"
     PopupPosition="Right" />
```

The *Panel1* control defines a list of radio buttons, each containing a suggestion for filling the text box. The *HoverMenu* extender targets the text box control and defines *Panel1* as its dynamic pop-up panel. The *PopupPosition* property indicates the position of the panel with respect to the target control. Likewise, other properties not shown in the previous example code, such as *OffsetX* and *OffsetY*, define the desired offset of the panel. The *PopDelay* sets the time (in milliseconds) to pass between the mouse movement and the display of the panel. The *HoverCssClass* can optionally be used to give the text box a different style when the hover menu is on. It is interesting to look at the CSS class associated with the panel:

```
.popupMenu
{
    position:absolute;
    visibility:hidden;
    background-color:#F5F7F8;
}
.hoverPopupMenu
{
    background-color:yellow;
}
```

It is key that the *visibility* attribute of the panel is set to *hidden* just as with *CollapsiblePanel* control; otherwise, the panel will display upon page loading and hidden immediately afterwards.

Just as for the *PopupControl* extender, to take full advantage of the *HoverMenu* extender you need to place extended controls inside of an *UpdatePanel* control. In this way, whenever the user clicks a radio button, the panel posts back asynchronously and fires the *SelectedIndexChanged* event on the server.

```
void RadioButtonList1_SelectedIndexChanged(object sender, EventArgs e)
{
    TextBox1.Text = RadioButtonList1.SelectedValue;
}
```

The server-side event handler will then just update the text in the text box., as shown in Figure 5-12.

The *ModalPopup* Extender

The *ModalPopup* extender displays in a modal way any content associated with the control identified by the *PopupControlID* property. The *TargetControlID* property in this case refers to a clickable control:

```
<act:ModalPopupExtender ID="ModalPopupExtender1" runat="server"
        TargetControlID="LinkButton1"
        PopupControlID="PopupContent"
        OkControlID="Button1"
        CancelControlID="Button2">
```

Notice that the *ModalPopup* extender is fired by the *onclick* event on the target control. It turns out, therefore, that the target control can only be a control that supports clicking. The pop-up control doesn't have to be a *Panel* control; generally, it can be any control. However, it will normally be a control that contains a bunch of other controls—typically, a *Panel*.

In the pop-up panel you can optionally identify an *OK* control and a *Cancel* control. You set the ID of such controls (commonly, buttons) through the *OkControlID* and *CancelControlID* properties. The pop-up behavior is clearly a client-side action, so some JavaScript code might be required in response to the user's clicking the *OK* or *Cancel* control. You use the *OnOkScript* property to specify the JavaScript function to run in case the user clicks the OK button; you use *OnCancelScript* otherwise.

The following markup shows the content of a sample modal panel. Note that the *Panel* control should set its CSS *display* attribute to *none* to make any contents invisible at first.

```
<asp:LinkButton ID="LinkButton1" runat="server" text="Click me" />
<asp:Panel runat="server" ID="PopupContent" BackColor="Yellow">
        <div style="margin:10px">
        Take note of this message and tell us if you strongly agree.
        <br /><br />
        <asp:Button ID="Button1" runat="server" Text="Yes" width="40px"  />
        <asp:Button ID="Button2" runat="server" Text="No"  width="40px" />
        </div>
</asp:Panel>
```

Figure 5-13 shows the modal dialog box in action.

A couple of graphical properties—*DropShadow* and *BackgroundCssClass*—complete the extender. A Boolean property, *DropShadow* indicates whether a drop shadow—as shown in Figure 5-13—should be rendered. *BackgroundCssClass*, on the other hand, determines the style that is temporarily applied to the underlying page:

Figure 5-13 The *ModalPopup* extender in action

```
modalBackground {
    background-color:Gray;
    filter:alpha(opacity=70);
    opacity:0.7;
}
```

The preceding style grays out the page and makes it partially opaque for a nicer effect.

The *PopupControl* Extender

The *PopupControl* extender can be attached to any HTML element that fires the *onclick*, *onfocus*, or *onkeydown* events. The ultimate goal of the extender is to display a pop-up window that shows additional content, such as a calendar on a text box in which the user is expected to enter a date. The contents of the pop-up panel are expressed through a *Panel* control, and they can contain ASP.NET server controls as well as static text and HTML elements:

```
<asp:textbox runat="server" ID="InvoiceDateTextBox" />
<asp:panel runat="server" ID="Panel1">
    ...
</asp:panel>

<act:PopupControlExtender ID="PopupExtender1" runat="server"
        TargetControlID="InvoiceDateTextBox"
        PopupControlID="Panel1"
        Position="Bottom" />
```

The *TargetControlID* property points to the control that triggers the popup, whereas *PopupControlID* indicates the panel to display. The *Position* property sets the position of the panel—either at the top, left, right, or bottom of the parent control. (See Figure 5-14.)

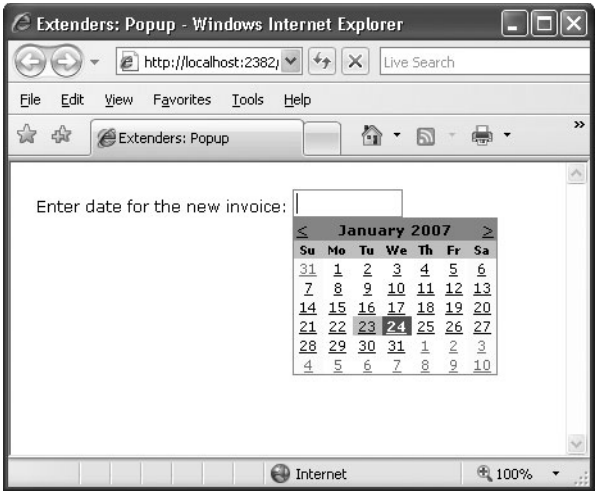

Figure 5-14 The *PopupControl* extender in action

Additional properties are *OffsetX* and *OffsetY*, which indicate the number of pixels to offset the popup from its position, as well as *CommitProperty* and *CommitScript*, which can be used to assign values to the target control.

The pop-up window will probably contain some interactive controls and post back. For this reason, you might want to insert it within an *UpdatePanel* control so that it can perform server-side tasks without refreshing the whole page. Typically, the popup will be dismissed after a postback—for example, the popup shown in Figure 5-14 is configured to be dismissed after the user has selected a date. The calendar in this case fires the *SelectionChanged* event on the server:

```
protected void Calendar1_SelectionChanged(object sender, EventArgs e)
{
    PopupExtender1.Commit(
          Calendar1.SelectedDate.ToShortDateString());
}
```

The *Commit* method sets the default property of the associated control to the specified value. If you want to control which (nondefault) property is set on the target when the popup is dismissed, use the *CommitProperty* property. Likewise, you use the *CommitScript* property to indicate the Javascript function to execute on the client after setting the result of the popup.

Warning Note that an extender can't be placed in a different *UpdatePanel* than the control it extends. If the extended control is incorporated in an *UpdatePanel*, the extender should also be placed in the updatable panel. If you miss this, you get a runtime exception.

User-Interface Extenders

In the family of ASP.NET AJAX extenders, the biggest group is user-interface extenders—that is, special components that help in the implementation of rich and user-friendly features.

The *AlwaysVisibleControl* Extender

The *AlwaysVisibleControl* extender allows you to pin a given control, or panel of controls, to one of the page corners so that it appears to float over the background body as the page is scrolled or resized. You can use the extender with virtually any ASP.NET control.

```
<span style="background-color:yellow;" runat="server" id="Msg">
    Need a bit of dummy text? Look at
    <b>http://www.loremipsum.net</b></span>

<act:AlwaysVisibleControlExtender ID="av1" runat="server"
     TargetControlID="Msg"
     HorizontalSide="Left"
     VerticalSide="Top" />
```

You set the target position for the bound control using the *HorizontalSide* and *VerticalSide* properties to define the corner of the page where the content should be docked. The *HorizontalSide* property accepts *Left* and *Right* as values, whereas *Top* and *Bottom* are feasible values for the *VerticalSide* property. You can also control the offset from each border using the *VerticalOffset* and *HorizontalOffset* properties. Finally, *ScrollEffectDuration* indicates how many seconds the scrolling effect will last when the target control is repositioned. (See Figure 5-15.)

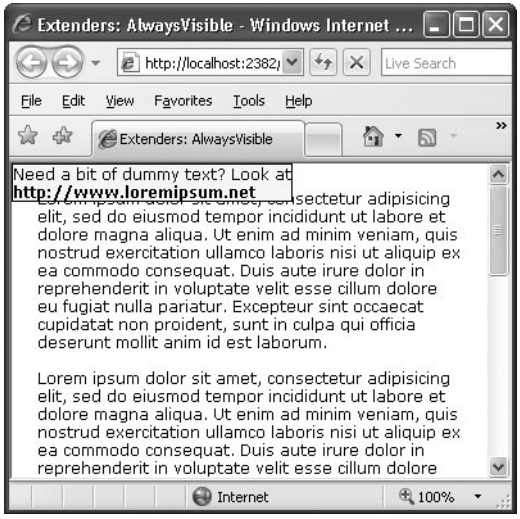

Figure 5-15 The *AlwaysVisible* extender in action

Note that you can't add the extender to a plain HTML element. If you have an HTML block to keep always visible (for example, the <*span*> tag in the previous example), add the *runat=server* attribute and give it a unique ID.

The *CascadingDropDown* Extender

The *CascadingDropDown* extender can be attached to a *DropDownList* control to automatically populate it based on the current selection of one or more parent *DropDownList* controls.

The *CascadingDropDown* extender is designed to fit in a relatively common scenario in which the contents of one drop-down list depend on the selection of another list. With this arrangement, you don't need to transfer to the client the entire data set from which a child list can select a subset of items to display in accordance with the selection on its parent. For example, suppose you want the user to select a country and a city in that country. To minimize data transfer and provide a friendlier user interface, you might want to keep the city list empty until a selection is made on the country list. When a country is selected, you get back to the server to download the list of cities available for that country. The *CascadingDropDown* extender simplifies this scenario by injecting some glue code into the client page and also making some assumptions on the structure of your page code.

All the logic about the contents of the set of *DropDownList* controls is expected to reside on a Web service. The Web service, in turn, can use any suitable method for storing and looking up any relevant data. The Web service, though, is somewhat forced to use a contracted schema. In particular, it needs to have a method with the following signature:

```
[WebMethod]
public CascadingDropDownNameValue[] GetDropDownContents(
    string knownCategoryValues, string category)
{
    ...
}
```

The name of the method can vary, of course. The *CascadingDropDownNameValue* type is an internal collection type that is designed to contain the name/value items to show in the drop-down list. Each drop-down list bound to the extender belongs to a category:

```
<act:CascadingDropDown ID="CascadingDropDown1" runat="server"
     TargetControlID="DropDownList1"
     Category="Country"
     PromptText="Please select a country"
     ServiceMethod="GetDropDownContentsPageMethod" />
<act:CascadingDropDown ID="CascadingDropDown2" runat="server"
     TargetControlID="DropDownList2"
     Category="City"
     PromptText="Please select a city"
     LoadingText="Please, wait ..."
     ServicePath="CityFinderService.asmx"
```

```
    ServiceMethod="GetDropDownContents"
    ParentControlID="DropDownList1" />
```

The content of the *Category* property is any name that helps the Web service method to understand what kind of data should be retrieved and the meaning of the input arguments. The *PromptText* property sets any text that you want to display in the drop-down list when no selection is currently made and the control is typically disabled. The *LoadingText* property indicates any text that has to be displayed while the drop-down list is being populated. The *ServiceMethod* property indicates the method to call to fill in the list control. If no *ServicePath* is specified, the method is assumed to be a page method. Finally, *ParentControlID* creates a hierarchy and designates a list to be the child of another list.

Each time the selection changes in a parent *DropDownList* control, the extender makes a call to the Web service and retrieves the list of values for the next *DropDownList* in the hierarchy. If no selection is currently made, the extender automatically disables the control. (See Figure 5-16.)

Figure 5-16 The *CascadingDropDown* extender in action on two drop-down lists

The *DropShadow* Extender

The *DropShadow* extender is designed to add a drop shadow to panel controls to make them look more professional. You can also set the opacity and width of the shadow:

```
<asp:Panel runat="server" ID="Panel1">
    <div style="padding:8px">
        <asp:TextBox ID="TextBox1" runat="server" />
    </div>
</asp:Panel>

<act:DropShadowExtender ID="DropShadowExtender1" runat="server"
     TargetControlID="Panel1" Opacity=".65" Width="5" Rounded="true" />
```

The *TargetControlID* property sets the control that will be rendered with a drop shadow. This control should generally be a *Panel*; however, as long as you don't set rounded corners, it can also be any other ASP.NET control, such as a *TextBox*. You control the opacity of the shadow through the *Opacity* property. Values for the property range from *0.0* to *1.0*, where *0* (or *0.0*) means total transparency. Hence, the closer the value is to *1* (or *1.0*) the darker the shadow will be.

The *Rounded* Boolean property indicates whether the surrounding panel and the shadow should have rounded corners. The default is *false*. Figure 5-17 shows the extender in action.

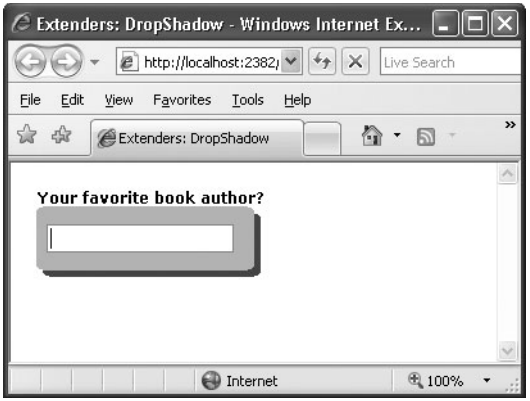

Figure 5-17 The *DropShadow* extender in action

The *DynamicPopulate* Extender

The *DynamicPopulate* extender is a sort of binder component that replaces the markup of a given control with the markup returned by a Web service method call. The extender can be seen as a shrink-wrapped and simplified version of the *UpdatePanel* control that we discussed in Chapter 4. It captures a client event and fires a remote call. The returned string is inserted in the page DOM as the child of the target element. Here's an example:

```
<input type="button" id="Button1" runat="server" value="Refresh ..." />
<hr />
<b>Last updated: </b>
<asp:Panel runat="server" ID="Msg" Style="padding:2px;height:2em;" />

<act:DynamicPopulateExtender ID="DynamicPopulateExtender1" runat="server"
     TargetControlID="Msg"
     ClearContentsDuringUpdate="true"
     PopulateTriggerControlID="Button1"
     ServiceMethod="GetTimeOnServer"
     UpdatingCssClass="updating" />
```

When the user clicks on the specified item—in this case, the button named *Button1*—the extender starts working. It invokes the method *GetTimeOnServer* and replaces the subtree rooted in the *Msg* control with its output. The method *GetTimeOnServer* is a Web service method. You specify the URL to the service using the *ServicePath* property. If this property is not set, the method is assumed to be a page method defined either in the code file of the page or inline through a server *<script>* tag:

```
[WebMethod]
public string GetTimeOnServer(string contextKey)
{
    // Use contextKey to receive data from the client
```

```
    // Get the output—it can be HTML markup
    return DateTime.UtcNow.ToString();
}
```

ClearContentsDuringUpdate is a Boolean property that clears the contents of the target control during the update. If you want to display a special style or a bitmap during the operation, you set a CSS style through the *UpdatingCssClass* property. When creating such a CSS class, bear in mind that you should ensure the target control has a minimum height. (It must be explicitly set on panels.) You use the *background-image* CSS attribute to set the image to display.

Note that you can use any HTML element to trigger the dynamic population of the target control. It doesn't have to be a button control; and it doesn't have to be a submit button such as *LinkButton* or *Button*. In this case, in fact, the page will post back and you'll lose the benefit of the ASP.NET AJAX platform.

Note You can also use a piece of JavaScript code to dynamically populate a given DOM element. In this case, you set the *CustomScript* property to the name of a JavaScript global function. Whether you use a custom script or a server method, you can use the *ContextKey* property of the extender to pass an arbitrary string to the code.

The *PagingBulletedList* Extender

Imagine a page that has to present a long list of items to users—say, a list of customers. A common solution entails using a pageable grid control. ASP.NET grids do a postback for each new page, but by wrapping the grid in an *UpdatePanel* control you can brilliantly fix the issue. What if you figure that a grid is far too heavy a control and that you need to list items using a bullet-list control?

The ASP.NET *BulletedList* control lists the contents of a data source using a variety of bullet-point user interfaces. It doesn't provide paging, though. The *PagingBulletedList* extender is a surprisingly simple and effective extender to page through the contents of a *BulletedList* control. Let's consider the following code that populates a bulleted list with no paging:

```
<asp:BulletedList ID="BulletedList1" runat="server" DisplayMode="Text"
     DataSourceID="ObjectDataSource1" DataTextField="CompanyName" />

<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
         TypeName="IntroAjax.CustomerManager"
         SelectMethod="LoadAll">
</asp:ObjectDataSource>
```

Such a page will generate more than 80 bullet-point items—one for each customer in the Northwind database. By extending the *BulletedList* control with the extender, you get output like that shown in Figure 5-18.

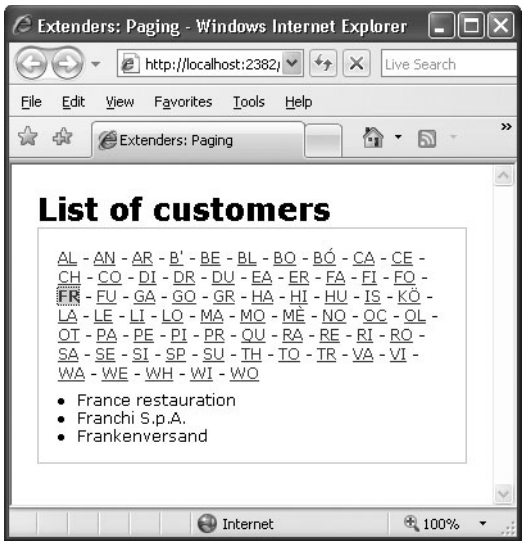

Figure 5-18 The *PagingBulletedList* extender in action

But then add the code for the extender:

```
<act:PagingBulletedListExtender ID="PagingBulletedList1" runat="server"
     TargetControlID="BulletedList1"
     ClientSort="true"
     IndexSize="2"
     Separator=" - "
     SelectIndexCssClass="selectIndex" />
```

The extender organizes all bound items in pages and displays links to each of them. Pages can contain a fixed number of items or all the items that match an initial string. The *IndexSize* property indicates how many letters in the displayed text should be used to create a page. If you set it to *1*, you will have a pure alphabetical menu. If you set it to 2, you obtain a more granular view, as each page contains only the items whose name matches the two-letter initial. As an alternative to using *IndexSize*, you can use *MaxItemsPerPage*. In this case, each page (except perhaps the last one) will have exactly the specified number of items.

The *Separator* property indicates the character used to separate menu items. *SelectIndexCssClass* and *UnselectIndexCssClass* set the CSS classes for selected and unselected menu items. Finally, if *ClientSort* is set to *true*, items are alphabetically sorted on the client before display.

The *ResizableControl* Extender

Most users would welcome pages where they can dynamically resize certain HTML elements, such as panels of text or images. The *ResizableControl* extender attaches to an element of a Web page and provides a graphical handle for users to resize that element. Placed at the

lower-right corner of the element, the handle lets the user resize the element as if it were a window. Let's consider the following markup:

```
<asp:Panel ID="Panel1" runat="server" Style="overflow:hidden"
           Width="130px" Height="65px">
    <asp:Image ID="Image1" runat="server" ImageUrl="~/images/ajax.gif"
           Style="width:100%; height:100%;" />
</asp:Panel>

<asp:Panel ID="Panel2" runat="server" Style="overflow:auto"
           Width="130px" height="100px">
    This text resizes itself to be as large as possible
    within its container.
</asp:Panel>
```

As you can see, the first panel contains an image; the second includes plain text. Some CSS attributes are necessary for the extender to work properly. In particular, you might want to set the overflow attribute to *hidden* for images and to *auto* for text. The *overflow* attribute controls the appearance of scrollbars when the contents of the element exceeds the reserved space. In the example, I stretch or shrink the image and scroll the text.

It is important to give panels an initial correct size. In particular, for images you should give the surrounding panel the same size of the image. Panels surrounding a block of text should be given an explicit size too.

```
<act:ResizableControlExtender ID="Resizable1" runat="server"
     TargetControlID="Panel1"
     ResizableCssClass="resizingStyle"
     HandleCssClass="handleStyle" />
<act:ResizableControlExtender ID="Resizable2" runat="server"
     TargetControlID="Panel2"
     ResizableCssClass="resizingStyle"
     HandleCssClass="handleStyle" />
```

The *ResizableControl* extender features two mandatory properties: *TargetControlID* and *HandleCssClass*. The former indicates the control to resize; the latter sets the name of the CSS class to apply to the resize handle. In addition, *ResizableCssClass* is the CSS class to apply to the element when resizing. You might want to use this class to change or thicken the color of the border to emphasize the operation. Here's a typical handle CSS class:

```
. handleStyle
{
    width:16px;
    height:16px;
    background-image:url(~/images/HandleGrip.gif);
    overflow:hidden;
    cursor:se-resize;
}
```

Figure 5-19 shows the extender in action.

Figure 5-19 The *ResizableControl* extender in action

The *ResizableControl* extender also features two client events (*onresizing* and *onresize*) that can trigger some JavaScript code to do more complex things, such as increasing the font size to fit a larger area. The extender allows you to define a minimum and maximum size for the elements being resized. Any changes resulting from control resizing are automatically persisted across postbacks.

The *RoundedCorners* Extender

The *RoundedCorners* extender is a subset of the *DropShadow* extender, as it is limited to rounding the corners of child panels:

```
<asp:Panel runat="server" ID="Panel1" BackColor="LightBlue" Width="170px">
    <div style="margin-left:2px">
        <asp:TextBox ID="TextBox1" runat="server"
                  BackColor="LightBlue"
                  BorderWidth="0px" />
    </div>
</asp:Panel>

<act:RoundedCornersExtender ID="RoundedCornerExtender1" runat="server"
     TargetControlID="Panel1" Radius="6" Color="LightBlue" />
```

The extender accepts arguments to set the target control and define the desired radius of the rounded corner and the color of the surrounding border. The rounded corner, in fact, is obtained by rendering an additional border all around the target control. By removing all borders around the text box and using the same background color for the panel and the text box, you can obtain the nice effect shown in Figure 5-20—a rounded text box control.

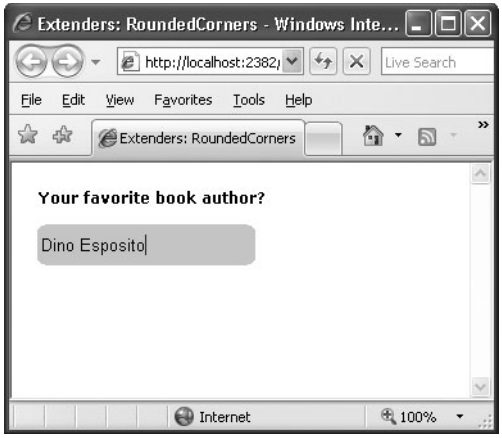

Figure 5-20 The *RoundedCorners* extender in action

Input Extenders

Web pages still rely on the facilities built into the HTML markup language to let users enter data into forms and input fields. But HTML input elements are sometimes too limited and simple for today's applications and users. For this reason, the plain old <*input type="text*"> element, and its *TextBox* ASP.NET counterpart, need some extra features. The ACT supplies a couple of extenders that transform the classic input field into a more interactive and user-friendly text box.

The *AutoComplete* Extender

Typing data into a text box might be boring at times, and even more so when you type the same chunk of text over and over again. For this reason, Web browsers began supporting auto-completion features a while ago. A Web browser's auto-completion feature consists of the code's ability to track any URL that the user ever types in. In this way, the browser can quickly prompt you with a suggestion when you're typing a possibly long and hard-to-remember URL.

Auto-completion is also a feature that some browsers, such as Internet Explorer 5 and newer versions, support for any custom text box you use in HTML forms. The browser saves on the local machine any data ever typed into an input field with a given name, and it makes that information available to all pages in the site that feature an HTML element with the same name. The feature is integrated with the browser and is transparent to users and developers. The list of suggestions builds up incrementally and is entirely beyond the control of page authors.

The *AutoComplete* extender extends the auto-completion to any ASP.NET *TextBox* control; more importantly, it gives page authors the means to define programmatically the list of suggestions (as retrieved via a Web Service).

```
<asp:TextBox ID="CustomerName" runat="server" />

<act:AutoCompleteExtender runat="server" ID="AutoCompleteExtender1"
     TargetControlID="CustomerName"
     Enabled="true"
     MinimumPrefixLength="1"
     ServicePath="Suggestions.asmx"
     ServiceMethod="GetSuggestions" />
```

In the code snippet, the text box is auto-completed using the data returned by the *GetSuggestions* method of the specified Web service. The *ServicePath* and *ServiceMethod* extender properties identify the service to provide auto-completion data; the *MinimumPrefixLength* property sets how many characters the user has to type to trigger the auto-completion feature. In the preceding code, auto-completion begins with the first typed character.

Note The Web service has to be a local Web service that is installed on the same server machine and application as the page that is using it. Note that this consideration holds true for all Web services used by ASP.NET AJAX extenders.

The Web service method you use to provide suggestions must have a known signature:

```
[WebMethod]
public string[] GetSuggestions(string prefixText, int count)
{
    ...
}
```

The *prefixText* argument indicates the text the user has typed so far; the *count* argument sets the desired number of suggestions. The result is displayed in a drop-down panel underneath the text box. The user can use either the mouse or the keyboard to select one of the suggested items. The following code shows a Web service method that returns a subset of customer names that match the provided prefix. (See Figure 5-21.)

```
public class SuggestionService : System.Web.Services.WebService
{
    [WebMethod]
    public string[] GetSuggestions(string prefixText, int count)
    {
        int i=0;
        DataView data = GetData();
        data = FilterData(data, prefixText);
        string [] suggestions = new string[data.Count];

        foreach (DataRowView row in data)
            suggestions[i++] = row["companyname"].ToString();

        return suggestions;
    }

    private DataView GetData()
```

```
    {
        DataView view = (DataView)HttpContext.Current.Cache["Suggestions"];
        if (view == null)
        {
            SqlDataAdapter adapter = new SqlDataAdapter(
                "SELECT * FROM customers", "...");
            DataTable table = new DataTable();
            adapter.Fill(table);
            view = table.DefaultView;

            // Store the entire data set to the ASP.NET Cache for
            // further reuse
            HttpContext.Current.Cache["Suggestions"] = view;
        }

        return view;
    }

    private DataView FilterData(DataView view, string prefix)
    {
        // Filter out undesired strings
        view.RowFilter = String.Format("companyname LIKE '{0}%'", prefix);
        return view;
    }
}
```

Figure 5-21 The *AutoComplete* extender in action

The *Calendar* Extender

Plenty of input forms require users to specify a date. As an ASP.NET developer, you know how frustrating it can be for a user to cope with separators, formats, cultures, and all sorts of things that affect the representation of a date. The most natural way of choosing a date in an input form is through a calendar; and ASP.NET 2.0 does indeed provide such a control. The original

ASP.NET 2.0 *Calendar* control requires full-page postbacks, although you can find a number of good workarounds for it—from wrapping it up in an updatable panel to using the hover-menu extender we considered earlier.

The *Calendar* extender is the ultimate and, I believe, definitive solution. Attached to a text box, it provides client-side date-picking functionality with customizable date format and pop-up control. You can interact with the calendar by clicking on a day, navigating to a month, or selecting a particular link to set the current date. (See Figure 5-22.)

Figure 5-22 The *Calendar* extender in action

```
<asp:TextBox runat="server" ID="TextBox1" />

<act:CalendarExtender runat="server" ID="CalendarExtender1"
    TargetControlID="TextBox1"
    CssClass="MyCalendar"
    Format="dd/MM/yyyy" />
```

By clicking on the title of the calendar, you can change the view from days-per-month to months-per-year, and even years-per-decade. The *Format* property allows you to select the final format of the date, which will be inserted in the bound text box. Normally, the calendar pops up when the companion text box gets the input focus. However, you can associate the display of the calendar with the clicking of an element—for example, an image. The *PopupButtonID* property gets and sets the ID of a control to show the calendar popup when clicked.

The *FilteredTextBox* Extender

The *FilteredTextBox* extender filters out some characters from the buffer of a given text box control. It differs from a validation control in that it just prevents users from entering invalid characters, whereas a validation control operates at a later time when the user tabs out of the input field.

```
<strong>How old are you?</strong>
<asp:TextBox ID="TextBox1" runat="server" />

<act:FilteredTextBoxExtender ID="Filtered1" runat="server"
     TargetControlID="TextBox1"
     FilterType="Numbers" />
```

The preceding text box is expected to accept a number indicating the age of the user. Clearly, it can accept only numbers. The extender ensures that only numbers can be typed in the input field. It does that by adding a piece of JavaScript code to the text box to filter out undesired characters.

The extender supports a few properties, including *FilterType*, which determine the filter applied to the input. The property can take any of the following values: *Numbers*, *UppercaseLetters*, *LowercaseLetters*, and *Custom*. The effect of the filter is obvious for the first three cases. When *Custom* is specified, though, you also set the *ValidChars* property to a comma-separated string where each item denotes a valid character. For example, the following code allows users to enter only A and B characters regardless of the case:

```
<act:FilteredTextBoxExtender ID="Filtered1" runat="server"
     TargetControlID="TextBox1"
     FilterType="Custom"
     ValidChars="A,a,B,b" />
```

You can't combine multiple filters on the same text box. If you want to filter all but letters, you can't add two *FilteredTextBox* extenders to the same control. Instead, you resort to a custom filter where you specify in the *ValidChars* property all acceptable characters.

Important Client-side filtering, as well as validation, is subject to the action of some JavaScript code. This means that by deactivating JavaScript on the client browser, any filtering or validation is subsequently disabled. In any case, you should not blindly trust what's typed by a user in a text box and, instead, apply proper filtering and validation on the server before you use that data for critical operations.

The *MaskedEdit* Extender

Added to a *TextBox* control, the *MaskedEdit* extender forces users to enter input according to the specified mask. In addition, data is validated on the client according to the data type chosen. To achieve the validation, a new validator control is introduced—*MaskedEditValidator*—that verifies the input. The masked edit validator is, in turn, associated with an instance of the masked edit extender:

```
<asp:TextBox runat="server" ID="TextBox1" />

<act:MaskedEditValidator ID="MaskedEditValidator1" runat="server"
    ControlExtender="MaskedEditExtender1"
    ControlToValidate="TextBox1"
    IsValidEmpty="False"
```

```
    EmptyValueMessage="Date is required"
    InvalidValueMessage="Date is invalid"
    TooltipMessage="Input a Date" />
<act:MaskedEditExtender runat="server" ID="MaskedEditExtender1"
    TargetControlID="TextBox1"
    Mask="99/99/9999"
    MessageValidatorTip="true"
    OnFocusCssClass="MaskedEditFocus"
    OnInvalidCssClass="MaskedEditError"
    MaskType="Date" />
```

Figure 5-23 shows the extender at work with date and monetary values.

Figure 5-23 The *MaskedEdit* extender in action

The extender supports three different masks for most common and specialized types: date, number, and time. The *MaskType* property sets the type of the final data; the *Mask* property defines the required input mask.

The *NoBot* Extender

Web applications such as blogs, forums, and portals are subject to having their input forms automatically filled by robot applications (also known as *bots*). For example, a bot can register as a user of the mail service and use the portal to send spam. The techniques employed to ensure that humans are filling an input form instead of a piece of smart software go under the name of CAPTCHA, an acronym for *Completely Automated Public Turing test to tell Computers*

and Humans Apart. In practice, CAPTCHA is a challenge-response test that requires the provision of additional information that must be figured out on the fly. Figuring this out is nothing special for humans, but virtually impossible for automated programs. The most popular example of CAPTCHA technique is a distorted image that represents a number. In a variety of blog systems, users encounter an additional input field to specify a number depicted within a figure before their posts will be accepted.

The *NoBot* extender is a control that attempts to apply anti-bot techniques to input forms. The extender might not be as powerful as a distorted image with a number inside, but it has the advantage of being completely invisible and hard to detect. All in all, the *NoBot* extender's usefulness is probably limited to sites with low traffic where the main goal is stopping spam, and where it's not a big deal if it doesn't achieve 100-percent effectiveness.

The *NumericUpDown* Extender

Added to a *TextBox* control, the *NumericUpDown* extender adds a couple of arrow buttons next to the control to let users increment and decrement the displayed value. Note that increment and decrement apply to numeric input as well as any user-defined enumeration.

```
<strong>How old are you?</strong>
<asp:TextBox ID="TextBox1" runat="server" Width="100px" />

<act:NumericUpDownExtender ID="UpDown1" runat="server"
     Width="100"
     TargetControlID="TextBox1" />
```

You should set the *Width* property on both the text box and the extender to control the size of the input field. Note that the *Width* property on the extender indicates the total width of the control including the arrow buttons. (See Figure 5-24.)

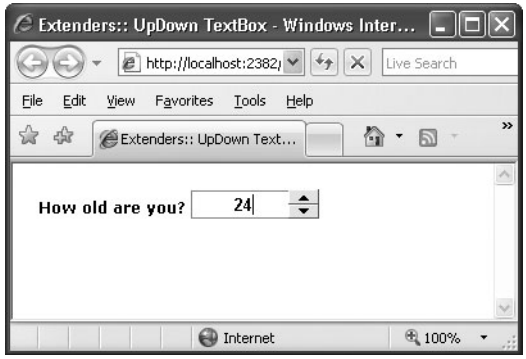

Figure 5-24 The *NumericUpDown* extender in action

The default increment is a numeric +1 or -1. However, the *RefValues* property lets you define a sequence of values to cycle through. The property accepts a string where each value is separated by a semicolon (;) symbol.

```
<act:NumericUpDownExtender ID="UpDown1" runat="server"
    Width="100"
    RefValues="Sun;Mon;Tue;Wed;Thu;Fri;Sat"
    TargetControlID="TextBox1"  />
```

The extender can also use a Web service to calculate the next or the previous value. In this case, you use *ServiceUpPath* and *ServiceUpMethod* to locate the service and method for the *up* operation. Likewise, you use *ServiceDownPath* and *ServiceDownMethod* for the *down* operation. The service methods can receive an arbitrary value from the client through the *Tag* property.

Note The sole use of the up-down extender is usually not enough to guarantee a pleasant experience. The up-down extender, in fact, doesn't force the text box to accept "only" the values generated by the up-down process. So you could type words in a numeric text box with an associated up-down extender. To avoid that, you might want to combine the up-down extender with a filter extender and use the *TextBox*'s *MaxLength* property to set the maximum number of characters.

The *PasswordStrength* Extender

Even the most secure system can't do much to protect your server if authorized users employ weak and easy-to-guess passwords. A number of best practices have been developed lately that characterize a strong password. The *PasswordStrength* extender attaches to a *TextBox* control and measures the strength of the current text if it's being used as a password.

From a syntax point of view, the extender can be attached to any *TextBox* control; from a semantic perspective instead, it makes sense only if applied to a *TextBox* control that is used for the entry of a password. The scenario in which the *PasswordStrength* extender proves useful is not a login page where the user enters his password to access a given functionality. Rather, it is helpful in forms where users registers their credentials to access a system feature.

```
<h2>Choose your password</h2>
<asp:TextBox ID="TextBox1" runat="server" />
<act:PasswordStrength runat="server" TargetControlID="TextBox1" />
```

The extender validates the current text against a set of requirements set by the page author. The result of the validation process is output through either a text message or a bar indicator to let users know about the level of complexity of the chosen password. With all default settings, the extender works as shown in Figure 5-25.

The feedback is displayed as plain text and dynamically as the user types in the buffer. The properties of the extender can be divided into two groups: appearance and behavior.

Appearance properties include *DisplayPosition* to set the position of the feedback text (above, below, left side, or right side), and *StrengthIndicatorType* to choose the type of visual feedback. It can be *Text* or *BarIndicator*. When the indicator type is *Text*, the *PrefixText* property sets

some text to use in the composition of the feedback message. The *TextCssClass* defines the CSS class to style the feedback message.

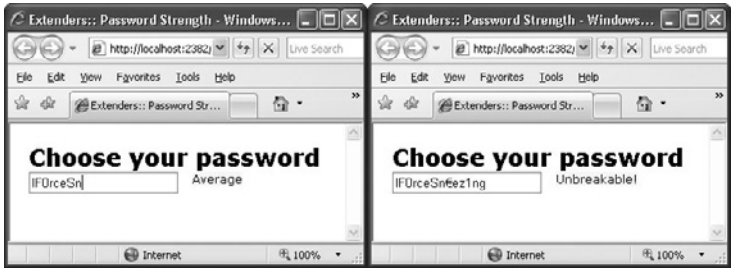

Figure 5-25 The *PasswordStrength* extender in action

When the indicator type is *BarIndicator*, the feedback appears as a gauge bar in a framed area. *BarBorderCssClass* and *BarIndicatorCssClass* properties let you style the bar.

The following properties allow you to set the password requirements to check: *RequiresUpperAndLowerCaseCharacters*, *PreferredPasswordLength*, *MinimumNumericCharacters*, and *MinimumSymbolCharacters*. By default, the password length is set to *10* and no other check is enabled on the contents.

The extender ranks the password text based on the requirements set. The score is rendered as a gauge bar or through a text message, as you saw in Figure 5-25. Text messages can be customized by assigning a list of semicolon-separated descriptions to the *TextStrengthDescriptions* property. You can specify a minimum of *2* and a maximum of *10* strings ordered from the weakest to the strongest.

The feedback you get from the extender is split in two parts: score and status message. The score is used to update the indicator or display a strength description. The status message is a help message displayed on a companion control identified by the *HelpStatusLabelID* property:

```
<asp:TextBox ID="TextBox1" runat="server" /> <br />
<asp:Label runat="server" ID="Label1" />

<act:PasswordStrength runat="server"
     TargetControlID="TextBox1"
     DisplayPosition="RightSide"
     PreferredPasswordLength="12"
     HelpStatusLabelID="Label1" />
</act:PasswordStrength>
```

The *Label1* control receives a message that suggests what to do to meet password requirements. You can also keep this message hidden all the time and display it on demand by clicking a button. In this case, you set two new properties: *HelpHandlePosition* and *HelpHandleCssClass*. The former sets the display position for the help icon; the latter styles the icon.

```
<act:PasswordStrength runat="server">
     <act:PasswordStrengthExtenderProperties TargetControlID="TextBox1"
          DisplayPosition="BelowRight"
          PreferredPasswordLength="12"
          HelpHandlePosition="RightSide"
          HelpHandleCssClass="helpHandle" />
</act:PasswordStrength>
```

To be precise, you set the icon to display using the CSS style:

```
.helpHandle
{
    width:16px;
    height:14px;
    background-image:url(images/Question.png);
    overflow:hidden;
    cursor:help;
}
```

Figure 5-26 shows the final page.

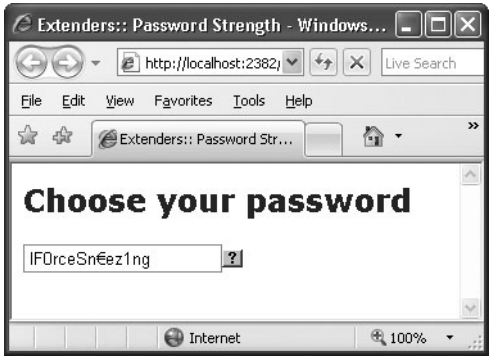

Figure 5-26 The *PasswordStrength* extender in action with a help button

The *Slider* Extender

The *Slider* extender allows you to morph a classic *TextBox* into a graphical slider so that users can pick up a numeric value from a finite range. You can set the orientation of the slider (horizontal or vertical) and also make it accept a discrete interval of values—that is, only a specified number of values within a given range. By default, the slider accepts values in the 0 through 100 range.

```
<h2>Your age</h2>
<asp:TextBox ID="TextBox1" runat="server" />
<hr />
<asp:Label runat="server" ID="Label1" />
```

```
<act:SliderExtender runat="server" ID="SliderExtender1"
     TargetControlID="TextBox1"
     BoundControlID="Label1" />
```

A value chosen using the slider is automatically persisted via a full or partial postback. You can reference the value using the *TextBox* programming interface. (See Figure 5-27.)

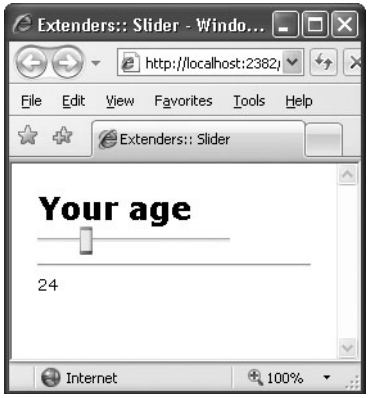

Figure 5-27 The *Slider* extender in action

The *TextBoxWatermark* Extender

The *TextBoxWatermark* extender allows you to define default text to display when the text box is empty. The extender takes care of showing and hiding the text as required.

```
<h2>Watermark</h2>
<br/>
<asp:TextBox ID="TextBox1" runat="server" />
 
<asp:TextBox ID="TextBox2" runat="server" />

<act:TextBoxWatermarkExtender runat="server" ID="TextBoxWatermark1"
     TargetControlID="TextBox1"
     WatermarkText="Type First Name Here"
     WatermarkCssClass="watermarked" />
<act:TextBoxWatermarkExtender runat="server" ID="TextBoxWatermark2"
     TargetControlID="TextBox2"
     WatermarkText="Type Last Name Here"
     WatermarkCssClass="watermarked" />
```

The extender uses the *TargetControlID* property to designate the target text box. The *WatermarkText* property defines the text to display, whereas *WatermarkCssClass* indicates the CSS style to apply to the watermark text. (See Figure 5-28.)

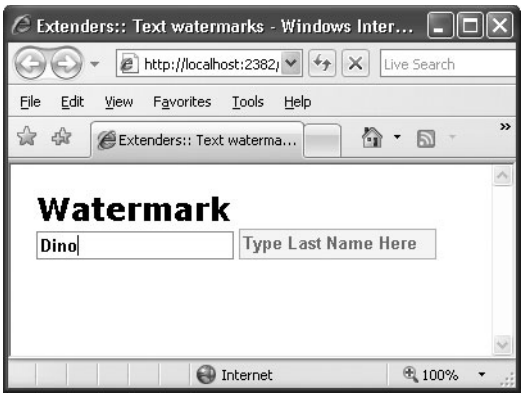

Figure 5-28 The *TextBoxWatermark* extender in action

The *ValidatorCallout* Extender

The *ValidatorCallout* extender enhances the graphical capabilities of existing ASP.NET validators. It displays the error message of a validator using the balloon-style ToolTips of Windows XP. (See Figure 5-29.)

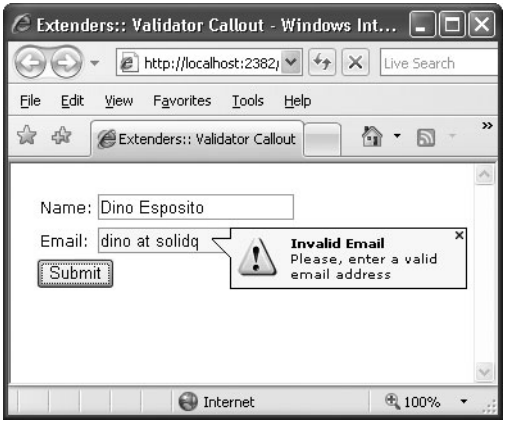

Figure 5-29 The *ValidatorCallout* extender in action

Here's how to use the extender:

```
<table>
<tr>
   <td>Name:</td>
   <td><asp:TextBox runat="server" ID="txtName" /></td>
</tr>
<tr>
   <td>Email:</td>
   <td><asp:TextBox runat="server" ID="txtEmail" /></td>
</tr>
</table>
```

```
<asp:Button runat="server" ID="Button1" Text="Submit" />
<asp:RequiredFieldValidator runat="server" ID="requiredName"
     ControlToValidate="txtName"
     Display="None"
     ErrorMessage="Required Field Missing<br />A name is required." />
<asp:RequiredFieldValidator runat="server" ID="requiredEmail"
     ControlToValidate="txtEmail"
     Display="None"
     ErrorMessage="Required Field Missing<br />Email address required." />
<asp:RegularExpressionValidator runat="server" ID="regularEmail"
     ControlToValidate="txtEmail"
     Display="None"
     ValidationExpression="[a-zA-Z_0-9.-]+\@[a-zA-Z_0-9.-]+\.\w+"
     ErrorMessage="<b>Invalid Email</b><br />Please, enter a valid email
                   address" />
<act:ValidatorCalloutExtender runat="server" ID="CalloutExtender1"
     TargetControlID="requiredName" HighlightCssClass="highlight" />
<act:ValidatorCalloutExtender runat="server" ID="CalloutExtender2"
     TargetControlID="requiredEmail" HighlightCssClass="highlight" />
<act:ValidatorCalloutExtender runat="server" ID="CalloutExtender3"
     TargetControlID="regularEmail" HighlightCssClass="highlight" />
```

Animation Extenders

The ACT comes with a full-fledged framework for building animations over the Web, leveraging the capabilities of rich browsers. The idea is to have an extensible set of animation blocks that can be composed together and run either in sequence or in parallel. The animations to be played are declaratively specified using XML.

The *Animation* Extender

The *Animation* extender allows you to use the built-in animation framework in a mostly declarative and codeless fashion. The extender plays specified animations whenever a client event occurs on the target control. Supported events include *load*, *click*, *mouseover*, and *mouseout*. Here's an example:

```
<act:AnimationExtender ID="AnimationExtender1" runat="server"
     TargetControlID="Panel1">
   <Animations>
      <OnLoad>
         <OpacityAction Opacity=".2" />
      </OnLoad>
      <OnHoverOver>
         <FadeIn Duration=".25" Fps="20"
                 MinimumOpacity=".2" MaximumOpacity=".8" />
      </OnHoverOver>
      <OnHoverOut>
         <FadeOut Duration=".25" Fps="20"
                  MinimumOpacity=".2" MaximumOpacity=".8" />
      </OnHoverOut>
   </Animations>
</act:AnimationExtender>
```

The *Animations* tag fully describes the animation. It is made up of a list of actions bound to specific events. In the above preceding example, the events are *load*, *hover*, and *mouseout*. The tag *OnLoad* describes the animation performed when the extended control is loaded. Tags *OnHoverOver* and *OnHoverOut* describe what happens when the *hover* effect is to take place, which is typically delayed from the actual *mouseover* and *mouseout* events. To describe an animation, you use a combination of existing animation blocks: fade-in, fade-out, opacity, pulse, move, resize, color interpolation, and scaling. You can also specify actions using custom script. The sample code just shown applies a slight opacity filter to the extended control upon loading and then fades its area in when the mouse hovers over it and out of it. For more information on built-in animation blocks, take a look at *http://ajax.asp.net/ajaxtoolkit/Walkthrough/AnimationReference.aspx*.

The *UpdatePanelAnimation* Extender

The *UpdatePanelAnimation* extender applies animation to a very specific situation where custom events need to be handled—before and after an updatable region is refreshed. Using the extender is as simple as defining an updatable panel in the page and adding the following code for the extender:

```
<act:UpdatePanelAnimationExtender runat="server" ID="UpdatePanelAnimation1"
    TargetControlID="UpdatePanel1">
    <Animations>
        <OnUpdating>
            <Sequence>
                <EnableAction AnimationTarget="Button1" Enabled="false" />
                <FadeOut AnimationTarget="Panel1" minimumOpacity=".3" />
            </Sequence>
        </OnUpdating>
        <OnUpdated>
            <Sequence>
                <FadeIn AnimationTarget="Panel1" minimumOpacity=".3" />
                <EnableAction AnimationTarget="Button1" Enabled="true" />
            </Sequence>
        </OnUpdated>
    </Animations>
</act:UpdatePanelAnimationExtender>
```

As you can see, it is nothing more than an animation applied to panel-specific events such as *OnUpdating* and *OnUpdated*. Before the update begins, the area that contains the sensitive contents to update is faded out and a control, *Button1*, is disabled. You use the *FadeOut* animation block for this purpose. In the preceding example, *Panel1* is merely the HTML block that contains the *UpdatePanel* control.

```
<div ID="Panel1">
    <asp:UpdatePanel ID="UpdatePanel1" runat="server" ...>
        ...
    </asp:UpdatePanel>
</div>
```

The *EnableAction* animation block declaratively disables the specified control during the update. In Chapter 4, you learned how to accomplish the same thing programmatically—the extender now provides a declarative approach.

Once the region has been updated, you run a *FadeIn* animation to bring the panel content back to its full colors and re-enable previously disabled controls using the *EnableAction*. The *Resize* animation can also be used to implement a sort of collapse/expand effect during the update: the panel closes, gets updated, and then unfolds to show its new contents. Imagination is your only limit. (See Figure 5-30.)

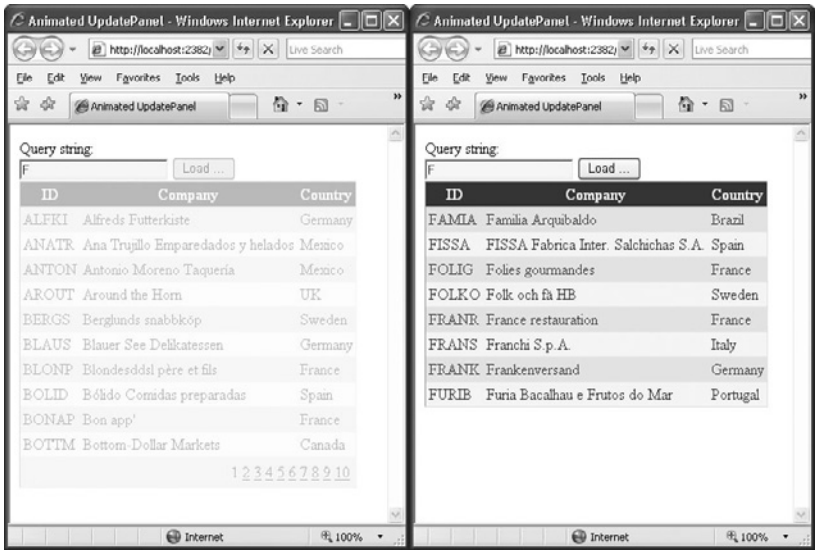

Figure 5-30 The *UpdatePanelAnimation* extender in action

Conclusion

No matter how many controls you have in your arsenal, you'll likely be always lacking just the one that is crucial for your current work. That's why the extensibility model of ASP.NET has been so successful over the years, and that's why so many component vendors crowd the market with excellent product offerings.

Anyway, always deriving new controls from existing ones might not necessarily be a wise strategy. A new control is required for a significant piece of server and client code that can be used to back up a good chunk of user interface. If you only need to filter the values in a text box, a custom text box control is hardly the best option. But until the arrival of ASP.NET AJAX and the AJAX Control Toolkit, there was no other way out.

With control extenders, you define the concept of a "behavior" and work with it as a distinct entity set apart from classic server controls. Extenders are server controls, but they work on top of bound controls and improve their overall capabilities by adding a new behavior.

An extender doesn't necessarily have to be an AJAX control. However, in the implementation of the AJAX Control Toolkit all extenders require a script manager, inject a good quantity of script code into the client pages, and take advantage of the client JavaScript library.

To take advantage of extenders and controls in the ACT, you don't have to import the binaries into your application. The ACT is an open-source project and, according to the license agreement you accept when you download the assembly, it can be incorporated piecemeal in your applications. This means, for example, that you can import only a few components through their source code and perhaps even adapt the code to your specific needs.

With this overview of controls and extenders, and armed with a strong knowledge of partial rendering, we're now ready to tackle an alternative programming model in ASP.NET AJAX—making remote procedure calls over the Web.

INTRODUCING MICROSOFT® ASP.NET AJAX

Get book updates on the Web!

Your first look at next-generation Web development tools—straight from a leading expert.

Get a jump on using the new framework for developing AJAX-enabled ASP.NET applications—with insights from a noted authority on ASP.NET. This reference covers the February 2007 release of ASP.NET AJAX Extensions 1.0. It delivers practical instruction and extensive code samples to help you create state-of-the-art applications with the latest Web development tools.

Discover how to:

- Understand the "AJAX lifestyle" fundamentals for creating interactive, responsive applications
- Use the Microsoft AJAX Library to support object-oriented JavaScript programming
- Access rich, sophisticated controls from the ASP.NET AJAX Control Toolkit
- Apply extenders to add new client-side behavior to existing ASP.NET controls
- Implement partial page rendering without client-side programming
- Use ASP.NET built-in authentication and profile services from JavaScript
- Create server-side services that can be invoked from JavaScript

Companion Web site includes:

- Microsoft Visual C# and JavaScript code samples
- Book updates covering the latest ASP.NET AJAX releases

*For details, including **system requirements**, see the Introduction.*

About the Author:

Dino Esposito is a well-known ASP.NET and workflow expert at Solid Quality Learning, a global provider of advanced education and mentoring for Microsoft technologies. He speaks at industry events, including DevConnections and Microsoft Tech·Ed, contributes to *MSDN® Magazine* and other publications, and is the author of several books, including *Programming Microsoft ASP.NET 2.0 Core Reference.*

Resource Roadmap

Developer Step by Step

- Hands-on tutorial covering fundamental techniques and features
- Practice files on CD
- Prepares and informs new-to-topic programmers

Developer Reference

- Expert coverage of core topics
- Extensive, pragmatic coding examples
- Builds professional-level proficiency with a Microsoft technology

Advanced Topics

- Deep coverage of advanced techniques and capabilities
- Extensive, adaptable coding examples
- Promotes full mastery of a Microsoft technology

ISBN-13: 978-0-7356-2413-9
ISBN-10: 0-7356-2413-5

Part No. X13-68395

U.S.A. $34.99

[*Recommended*]

Web Development/ Microsoft ASP.NET

Microsoft® Visual Studio®

Microsoft®