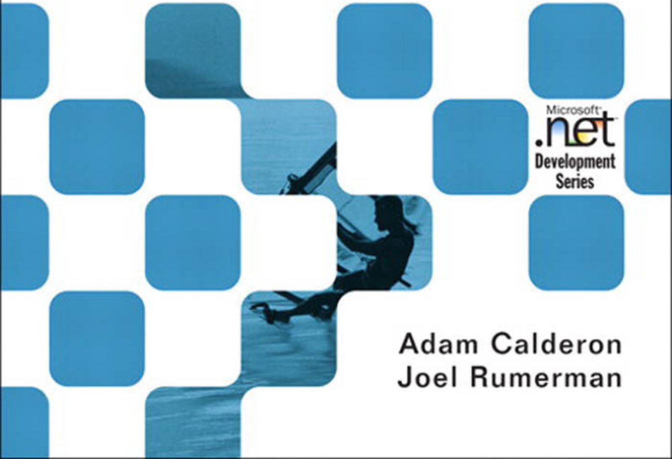
Foreword by Nikhil Kothari,
Software Architect, .NET Developer Platform, Microsoft

Advanced ASP.NET AJAX Server Controls

For .NET Framework 3.5



Adam Calderon
Joel Rumerman

Library of Congress Cataloging-in-Publication Data:

Calderon, Adam, 1964-
Advanced ASP.Net Ajax server controls for .Net 3.5 / Adam Calderon, Joel Rumerman.
p. cm.
ISBN 0-321-51444-0 (pbk. : alk. paper) 1. Internet programming. 2. Active server pages. 3. Microsoft .NET. 4. Ajax (Web site development technology) 5. Web servers. I. Rumerman, Joel, 1980- II. Title.

QA76.625.C34 2008
006.7'882—dc22

2008013462

ISBN-13: 978-0-321-51444-8
ISBN-10: 0-321-51444-0
Text printed in the United States on recycled paper at RR Donnelly in Crawfordsville, Indiana.
First printing: July 2008

Editor-in-Chief
Karen Gettman

Acquisitions Editor
Joan Murray

Development Editors
Sheri Cain
Chris Zahn

Managing Editor
Kristy Hart

Project Editor
Jovana San Nicolas-Shirley

Copy Editor
Keith Cline

Indexer
WordWise Publishing Services

Proofreader
Geneil Breeze

Publishing Coordinator
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
Bronkella Publishing

Foreword

THE ASP.NET PLATFORM POWERS millions of websites around the world today, and is perhaps one of the most productive platforms for web development. During the nearly ten years of its development and use, ASP.NET has formed around itself a strong community and vibrant ecosystem of developers.

The page framework and the associated server controls framework are quintessential to the success of ASP.NET and its developer experience, programming model, and extensibility. Writing this Foreword brings back memories of early ASP.NET days, and reminds me of the continued evolution of the framework as a platform alongside the Web.

In the late 1990s, the Web was very much a nascent application platform. Browsers brought potential for new levels of reach, but offered few and varying degrees of capabilities (remember HTML 3.2?), and concepts such as "stateless programming model" presented an odd paradigm shift. Server controls provided a set of familiar abstractions and created a component-based rapid application development (RAD) programming experience for the Web (à la Visual Basic) and allowed developers to feel at home as they started to look to the Web to build the next generation of data-driven applications.

Flash forward a few years, and in 2006, the AJAX buzz created a renewed interest in the Web as *the* application platform. Today, AJAX is mainstream and, quite literally, everywhere. It enables building interactive experiences that users have come to expect. Still, it brings new but similar challenges: varying browser APIs and an unfamiliar script-based programming model. Once again, ASP.NET (and in particular, server controls) provided a mechanism for creating a productive development model for incorporating AJAX-based techniques, and for encapsulating server and client behaviors into a familiar and consistent component model.

ASP.NET provides an end-to-end AJAX story. Traditional server controls create a simple server-centric AJAX programming model, but they are just a part of the story. This new generation of server controls leverages an AJAX script framework that independently enables a client-centric AJAX programming model. The core framework is complemented by the AJAX Control Toolkit, which offers both a compelling set of out-of-the-box components and an open source project for further developing the ASP.NET AJAX stack. I am excited to see this end-to-end story uncovered and unfold itself over the course of this book.

In this book, Adam and Joel focus on providing a beyond-the-basics drill down of the inner workings and extensibility of the ASP.NET AJAX framework by covering the programming patterns established by the script framework, the architecture, and the techniques to create AJAX-enabled server controls. They also cover advanced but still relevant topics such as localization and error handling. By providing a conceptual guide to understanding and extending the framework, this book is sure to serve any application or component developer who is looking to unlock the true potential of ASP.NET AJAX.

—Nikhil Kothari
Software Architect
.NET Developer Platform, Microsoft

Preface

Introduction

SERVER CONTROLS ARE AN INTEGRAL aspect of every ASP.NET application we build. They encapsulate browser appearance and server functionality in a reusable object. They can be used across multiple pages within a single ASP.NET application and across multiple ASP.NET applications. ASP.NET comes with a lot of prebuilt server controls. We have simple controls such as the label, and we have complex controls such as the `GridView`. We can also create our own server controls to meet a need not met by one of the existing controls by inheriting from the appropriate base class and overriding its methods as needed.

This model of using server controls to encapsulate browser appearance and server functionality has served our needs well since the inception of ASP.NET 1.0, but our server control needs are changing.

A new server control need that has recently surfaced is the ability to incorporate AJAX functionality directly into the server control.

This need arose because our web applications need to be more responsive and visually interactive than the traditional ASP.NET repaint-the-entire-screen model and therefore the traditional server control supplies. This requirement has emerged because users are using websites such as Gmail, Live.com, Yahoo! Mail, and others that don't repaint the screen every time they click a button or need to receive fresh data. Rather, they rely on AJAX to fetch fresh data and then update or add to a portion of the screen based on that data. Because these websites are heavily used and users really enjoy their experience while using these websites, they expect other websites to perform with the same elegance as their favored sites do. When a website doesn't perform with the same elegance, the user often moves on to another website that does. Those popular applications have raised the bar for what is an acceptably user-friendly website.

Because our users are demanding a website experience that essentially uses AJAX and we build our ASP.NET websites using server controls, we need a way of easily creating server controls that not only encapsulate browser appearance and server functionality, but also include AJAX functionality so that the server control itself is AJAX-enabled.

Taking a step back for a moment, unlike other technologies you might have read books about, ASP.NET AJAX server controls don't provide you with anything that you couldn't already do. We've always been able to embed AJAX functionality into server controls… it was just a real pain.

There were a few different methods we could use to include the JavaScript with our server control such as embedding it as a resource, but we eventually ended up having to do the same three tasks. To make our server control have some serious client capabilities, we always had to concatenate strings together to form JavaScript statements and functions, write browser sniffing statements to make sure that the JavaScript was cross-browser compatible, and add attributes or render out HTML that attached the JavaScript functionality to the client versions of our server controls. It wasn't impossible, but it was error-prone, and there was always this mingling of server code and JavaScript that was hard to maintain and even harder to read.

Furthermore, if you had multiple server controls that had client capabilities, it was difficult (but not impossible) to ensure that the client functions that each server control required didn't overwrite each other when rendered on the browser. Tracking down that problem was always a fun hour or so.

The difficulty grew exponentially if we wanted to include a mechanism for asynchronously communicating with the server when the user pressed a button embedded in the server control. Even with a helper communication library, there were always tricks to getting your control to communicate properly with the server.

These hindrances were problematic enough to lead to some bad programming habits and bad code and to scare programmers away from even attempting to include AJAX functionality in their server controls.

These problems are what Microsoft ASP.NET AJAX solves.

In this book, we teach you how to use ASP.NET AJAX to create server controls that encapsulate AJAX functionality. ASP.NET AJAX provides both server and client programming constructs that make adding AJAX capabilities to our server controls easy. Not to sound cliché, but with ASP.NET AJAX reducing the complexity of adding AJAX capabilities to our server controls, we can create server controls whose AJAX capabilities are limited only by our creativity. If we want a listbox that self-updates with fresh data, if we want a type-ahead textbox that dynamically populates from the server, or if we want a button that submits an address for verification, we can easily accomplish these things through ASP.NET AJAX.

The ASP.NET AJAX Components

As we go through the book we'll be talking about the three parts of ASP.NET AJAX: the Microsoft AJAX Library, the ASP.NET 2.0 AJAX Extensions, and the ASP.NET AJAX Control Toolkit. Here's a quick rundown of the different components.

Microsoft AJAX Library

The Microsoft AJAX Library is the JavaScript programming framework of ASP.NET AJAX. It provides all the client programming constructs you'll use to create new client objects and components. It's contained within the MicrosoftAjax.js JavaScript file that's embedded in the System.Web. Extensions DLL.

ASP.NET 2.0 AJAX Extensions

The ASP.NET 2.0 AJAX Extensions are server objects such as the `ScriptManager`, `ScriptControl`, and `ScriptDescriptor`, which provide a connection between the Microsoft AJAX Library and our server ASP.NET development. These server objects provide an important distinction between ASP.NET AJAX and other AJAX frameworks because they provide a server programming model for manipulating client code (and allow us to make AJAX-enabled server controls!). Like the Microsoft AJAX Library, they are included in the System.Web.Extensions DLL.

ASP.NET AJAX Control Toolkit

The ASP.NET AJAX Control Toolkit is a shared source project that is built on top of ASP.NET AJAX. It's an effort shared between Microsoft and the ASP.NET AJAX community with the goal of developing powerful and reusable ASP.NET AJAX extenders and controls.

It's not actually part of ASP.NET AJAX, but because it provides so many great server and extender controls, it's invaluable to the ASP.NET AJAX community. Creating new extender controls through it is a topic we cover fully.

Book Breakdown

The book is divided into four major parts. In the first part, we focus on the basics of the Microsoft AJAX Library and JavaScript, the programming language that powers it. We call this part "Client Code." In the second part, we focus on a creating distributable AJAX-enabled controls, and we call this part "Controls." In the third part, called "Communication," we focus on the different ways your client control can communicate with the server. Finally, in the fourth part, we focus on the AJAX Control Toolkit, a slightly higher-level model of creating AJAX-enabled server controls. This final part is aptly named "AJAX Control Toolkit."

Client Code

Chapter 1, "Programming with JavaScript," focuses on JavaScript, the programming language that powers the Microsoft AJAX Library. We spend a full chapter on JavaScript because so many developers (ourselves included) have glossed over key details when working with the language; and because you're going to be writing so much JavaScript to AJAX-enable your server controls, a solid background is important.

In Chapter 2, "Microsoft AJAX Library Programming," we continue where we left off in Chapter 1 by taking a look at how the Microsoft AJAX Library builds on JavaScript to provide a programming platform a .NET developer will find familiar.

Controls

Starting in Chapter 3, "Components," we begin our path to creating fully encapsulated AJAX-enabled controls by learning how to use and derive from three key client types: components, controls, and behaviors. We talk theory and provide a couple of practical examples.

In Chapter 4, "Sys.Application," we cover maybe the most important portion of the Microsoft AJAX Library as we discuss `Sys.Application` and how it acts like a client runtime with which we can interact.

In Chapter 5, "Adding Client Capabilities to Server Controls," we bring the server into the mix when we cover how to create server components that automatically create corresponding components.

In Chapter 6, "ASP.NET AJAX Localization," we continue adding control capabilities with an in-depth examination of localization in ASP.NET AJAX.

Finally, in Chapter 7, "Control Development in a Partial Postback Environment," we wrap up the "Controls" part with a look at the concerns surrounding how the `UpdatePanel` affects control development.

Communication

With Chapter 8, "ASP.NET AJAX Communication Architecture," we start looking at communication in ASP.NET AJAX using Windows Communication Foundation (WCF) services, page methods, and the client web service proxies.

In Chapter 9, "Application Services," we cover the application services and include a demonstration of how to build your own application service.

AJAX Control Toolkit

Beginning with Chapter 10, "ASP.NET AJAX Control Toolkit Architecture," we start our look at the AJAX Control Toolkit. We cover the base classes that are used by toolkit controls and the support and designer classes that provide additional features.

Finally, we conclude the book with Chapter 11, "Adding Client Capabilities to Server Controls Using the ASP.NET AJAX Control Toolkit," as we attach client capabilities to server controls using the AJAX Control Toolkit. This chapter includes how to build a new extender control and provide design-time features for it.

What Is Not Covered?

You might find it strange to see a note that talks about what we're not covering. We're including it for two reasons.

First, this book covers a pretty narrow topic when compared to ASP.NET AJAX at large. Because of this, we don't have the normal introductory chapter where we walk you through the basics or history of ASP.NET AJAX. Instead, we're making the assumption, good or bad, that you've got some ASP.NET AJAX knowledge under your belt. If you don't, don't worry; getting your ASP.NET AJAX knowledge to the point where you feel comfortable doesn't take long, and this book will pick up right where that basic knowledge leaves off. For this type of information, the Microsoft ASP.NET AJAX website located at http://asp.net/ajax is an excellent source.

Second, we're leaving out a familiar ASP.NET AJAX subject, and we wanted a chance to tell you and defend our decision before we got too far. This is something that we've repeatedly debated between the two of us and asked many colleagues for their opinion and was a decision that we didn't come to easily.

There are no chapters in which we cover how to use the `UpdatePanel` server control.

Okay, you haven't closed the book? Good. Let us explain how and why we came to this decision.

Simply put, the `UpdatePanel` is a server control. It comes with ASP.NET AJAX and provides a quick and dirty way to refresh a portion of a page such that the page goes through its normal lifecycle, but doesn't refresh the entire page when the page processing is done. Using it, we don't have to alter the way we've been programming web pages since ASP.NET 1.0 came out. This is a good thing and was a "quick win" for Microsoft. It allowed ASP.NET AJAX to be adopted quickly by ASP.NET developers and provided a unique advantage against other AJAX frameworks.

However, the `UpdatePanel` is just a server control and it's developed in such a way that it doesn't have a whole lot of comparative properties with the type of ASP.NET AJAX server control development we're covering.

We're not saying it's not an important server control and that it has no place in the AJAX world. Rather, it is an extremely valuable tool whose complexity and correct usage is worthy of a small book; just not this one.

Finally, although we do not cover how to use the `UpdatePanel`, we do cover how to create server controls so that they work correctly in an `UpdatePanel`, or more specifically a partial-postback, environment. We expect that you want your new server controls to work in any ASP.NET environment, and a partial-postback environment is no exception. The partial-postback environment, however, requires us to use some different methods, the new `ScriptManager.RegisterXXX` methods being the most common, and take some care in how we create our server controls. So, we've dedicated Chapter 7 to this topic.

Why Just Server Controls?

Writing a book on just server controls allows us to delve deeply into a narrow topic that is extremely important to web application developers. The ASP.NET AJAX books currently available all generally focus on the technology as a whole. Because they cover a broad range of topics, giving a taste of everything, they have trouble really getting into how certain parts of ASP.NET AJAX work and tend to give shallow coverage of topics that we think are key to creating server controls. It's been our experience that developers tend to move past the content of the more general books fairly quickly because nonbasic situations arise almost immediately when working on a real-life web application.

Target Audience

This book is primarily targeted at the experienced ASP.NET developer who has developed custom web server controls. We expect that you're reading this book to enhance your already proficient ASP.NET development skill set with new ASP.NET AJAX skills. The applications you develop demand elegance and professionalism and easy maintenance and scalability, so you tend to use server controls to your advantage wherever possible.

Besides your experience with ASP.NET, we expect that you're familiar with JavaScript and the basics of ASP.NET AJAX. Therefore, we don't cover how to set up a new ASP.NET AJAX-enabled web application, and although we do cover JavaScript, we start our coverage at a level where we assume some existing knowledge.

Our goal is to provide you with the tools you need to build reusable ASP.NET AJAX server controls or AJAX Control Toolkit extender controls. Our feeling is that reasonably knowledgeable ASP.NET developers will be able to learn the skills necessary to create new ASP.NET AJAX server controls through this book and then add that skill to their ASP.NET development tool bag.

Prerequisites

This book requires ASP.NET 3.5 AJAX and Visual Studio 2008. We heavily cover features included in ASP.NET 3.5 AJAX not included in ASP.NET 2.0 AJAX and C#'s and Visual Studio 2008's new capabilities such as automatic properties and JavaScript IntelliSense.

Source Code

The source code for the book's examples can be found on the book's website: www.informit.com/title/9780321514448.

11

Adding Client Capabilities to Server Controls Using the ASP.NET AJAX Control Toolkit

IN THE PRECEDING CHAPTER, we covered the architecture of the AJAX Control Toolkit, describing at a high level what it has to offer and the attributes, classes, and interfaces that make it all happen. The enhanced functionality you get in the toolkit, from attribute-based programming to rich animations, provides a compelling alternative to coding against the ASP.NET 2.0 AJAX Extensions and the Microsoft AJAX Library directly. In this chapter, we delve into the details of the toolkit a little further as we develop as series of extender controls that demonstrate the rich features the toolkit provides.

Adding Client-Side Behavior Using the ExtenderControlBase

The ASP.NET AJAX Control Toolkit provides many features to assist in the development of extender controls, such as the automatic creation of `$create` statements, the use of attributes to decorate extender control properties that should be included in the `$create` statement creation, built-in designer support, and many more. In this section, we revisit the `Image Rotator` extender we created in Chapter 5, "Adding Client Capabilities to Server Controls," and re-create it using the ASP.NET AJAX Control Toolkit. This approach enables us to compare the alternatives as we build the new extender.

The process of building an extender control using the ASP.NET AJAX Control Toolkit consists of four main steps.

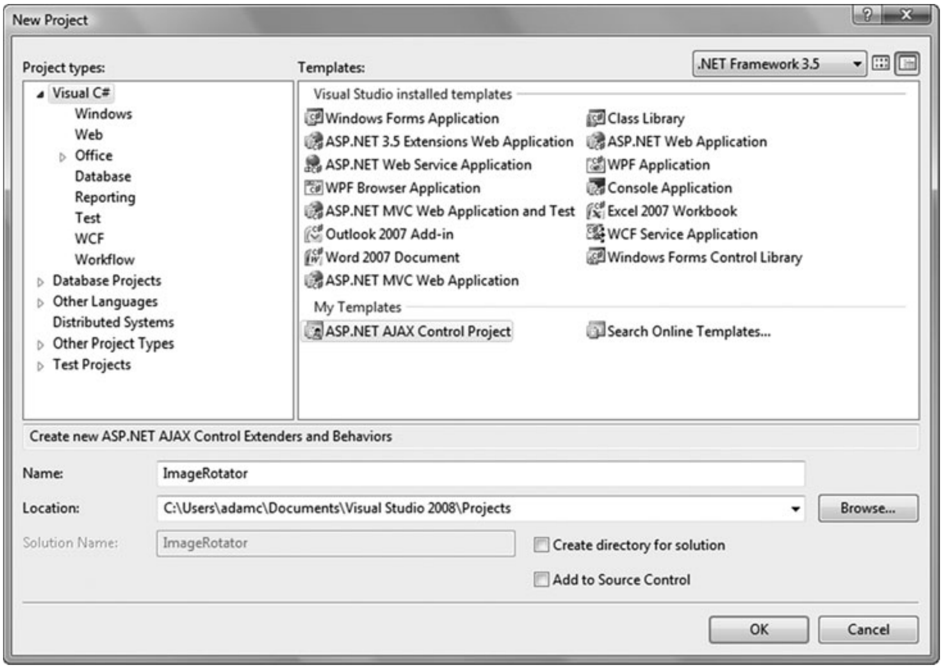
1. Create the template classes.
2. Provide implementation for the inherited extender control class.
3. Provide implementation for the `Sys.UI.BehaviorBase`-based JavaScript class.
4. Attach the extender control to an existing server control.

Visual Studio 2008 Extender Control Library Template

The ASP.NET AJAX Control Toolkit comes with full support for Visual Studio 2008 in the form of a project template that is geared toward creating an extender control library project. The template, shown in Figure 11.1, creates a library project structure (see Figure 11.2) that contains an extender control class, a designer class, and a JavaScript behavior class. In this section, we look at the ImageRotatorExtender.cs, ImageRotatorDesigner.cs, and ImageRotatorBehavior.js files that the template generated for us as we begin to discuss creating a new and improved `ImageRotator` extender.

NOTE Additional Template

The toolkit also comes with a template that generates the same files that can be used when you need to add additional extenders to an existing project, which can be found when you select Add New Item from a project.


New Project

Project types:
- Visual C#
  - Windows
  - Web
  - Office
  - Database
  - Reporting
  - Test
  - WCF
  - Workflow
- Database Projects
- Other Languages
- Distributed Systems
- Other Project Types
- Test Projects

Templates: .NET Framework 3.5

Visual Studio installed templates

Windows Forms Application	Class Library
ASP.NET 3.5 Extensions Web Application	ASP.NET Web Application
ASP.NET Web Service Application	WPF Application
WPF Browser Application	Console Application
ASP.NET MVC Web Application and Test	Excel 2007 Workbook
Outlook 2007 Add-in	WCF Service Application
Word 2007 Document	Windows Forms Control Library
ASP.NET MVC Web Application	

My Templates

ASP.NET AJAX Control Project	Search Online Templates...

Create new ASP.NET AJAX Control Extenders and Behaviors

Name: ImageRotator

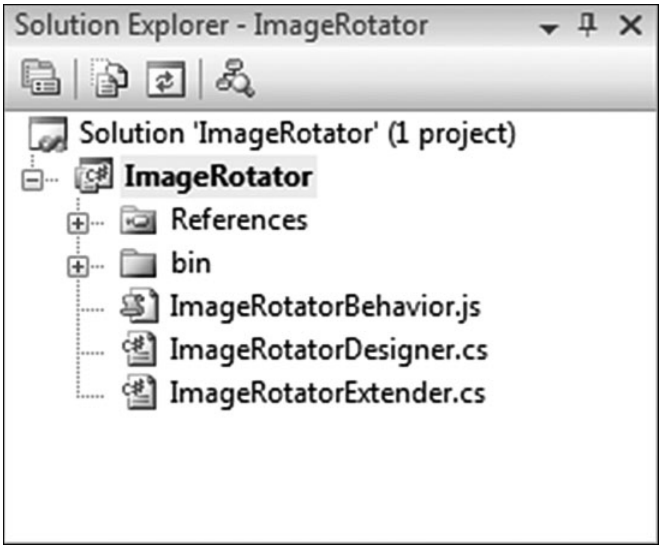
Location: C:\Users\adamc\Documents\Visual Studio 2008\Projects

Solution Name: ImageRotator

Create directory for solution

Add to Source Control

FIGURE 11.1 Extender control project template


Solution Explorer - ImageRotator

```
Solution 'ImageRotator' (1 project)
  ImageRotator
    References
    bin
    ImageRotatorBehavior.js
    ImageRotatorDesigner.cs
    ImageRotatorExtender.cs
```

FIGURE 11.2 Extender control project template structure

The ImageRotatorExtender class shown in Listing 11.1 serves as the basis for our ImageRotator extender control. The class inherits from ExtenderControlBase and provides a template that contains most of the required entries for us, such as the web resource registration of our associated behavior class, class attributes that associate the designer for the extender, the client script to be downloaded, and the target type for the extender. The template also creates a default property, demonstrating the use of the ExtenderControlProperty and DefaultValue attributes and the use of the GetPropertyValue method inside the property setter and getter.

NOTE GetPropertyValue Method Version

The version of the GetPropertyValue method used by the template is an outdated one. When building out the class, we will change the implementation to use the GetPropertyValue<T> version instead.

LISTING 11.1 ImageRotatorExtender Class

```
[assembly: System.Web.UI.WebResource(
"ImageRotator.ImageRotatorBehavior.js","text/javascript")]
namespace ImageRotator
{
  [Designer(typeof(ImageRotatorDesigner))]
  [ClientScriptResource("ImageRotator.ImageRotatorBehavior",
    "ImageRotator.ImageRotatorBehavior.js")]
  [TargetControlType(typeof(Control))]
  public class ImageRotatorExtender : ExtenderControlBase
  {
    [ExtenderControlProperty]
    [DefaultValue("")]
    public string MyProperty
    {
      get
      {
        return GetPropertyValue("MyProperty", "");
      }
      set
      {
        SetPropertyValue("MyProperty", value);
      }
    }
  }
}
```

The `ImageRotatorDesigner` class shown in Listing 11.2 will be the designer class for our `ImageRotator` extender control. The designer class provides default designer functionality for our extender control during design time. We associate the designer with our `ImageRotatorExtender` class by using the `Designer` attribute, which is automatically added when we use the template. The `ExtenderControlBaseDesigner<T>` class that the `ImageRotatorDesigner` class inherits from makes it possible for the properties of our extender control to show up in the Properties window while the design-time focus is on the image control we are extending. This default behavior provides a more efficient way of working with extenders and the controls they are extending.

LISTING 11.2 ImageRotatorDesigner Class

```
namespace ImageRotator
{
  class ImageRotatorDesigner : AjaxControlToolkit.Design.
    ExtenderControlBaseDesigner<ImageRotatorExtender>
  {
  }
}
```

The `ImageRotatorBehavior` class shown in Listing 11.3 will be the client-side behavior class for our `ImageRotator` extender control. The class consists of the same structure we used in Chapter 5, but now inherits from the `AjaxControlToolkit.BehaviorBase` class, which provides added functionality for working with client state and interacting with the asynchronous request events of the `Sys.WebForms.PageRequestManager`.

LISTING 11.3 ImageRotatorBehavior Class

```
/// <reference name="MicrosoftAjaxTimer.debug.js" />
/// <reference name="MicrosoftAjaxWebForms.debug.js" />
/// <reference name="AjaxControlToolkit.ExtenderBase.BaseScripts.js"
    assembly="AjaxControlToolkit" />

Type.registerNamespace('ImageRotator');

ImageRotator.ImageRotatorBehavior = function(element) {
  ImageRotator.ImageRotatorBehavior.initializeBase(this, [element]);
```

Listing 11.3 continued

```
    // TODO : (Step 1) Add your property variables here
    this._myPropertyValue = null;
  }

  ImageRotator.ImageRotatorBehavior.prototype = {
    initialize : function() {
      ImageRotator.ImageRotatorBehavior.callBaseMethod(this,'initialize');
      // TODO: Add your initialization code here
    },

    dispose : function() {
      // TODO: Add your cleanup code here
      ImageRotator.ImageRotatorBehavior.callBaseMethod(this, 'dispose');
    },

    // TODO: (Step 2) Add your property accessors here
    get_MyProperty : function() {
      return this._myPropertyValue;
    },
    set_MyProperty : function(value) {
      this._myPropertyValue = value;
    }
  }

  ImageRotator.ImageRotatorBehavior.registerClass(
    'ImageRotator.ImageRotatorBehavior', AjaxControlToolkit.BehaviorBase);
```

Inheriting from the ExtenderControlBase Class

The ASP.NET AJAX Control Toolkit comes with its own version of the `System.Web.UI.ExtenderControl` class, which provides additional functionality that supports the development pattern the toolkit is designed to work with. The `AjaxControlToolkit.ExtenderControlBase` class provides the inheritor support for serialization of property values, support for working with the toolkit-based attributes, seamless integration with control-based view state, support for working with client state, and the ability to specify an alternate script path for debugging and working with themes. The `ImageRotatorExtender` class in Listing 11.4 shows a much different-looking class than we saw in Chapter 5. The class no longer requires overrides for the `GetScriptDescriptors` and `GetScriptReferences` methods, it has class-level attributes, it has property-level attributes, and the property setters and getters are referencing their values through a method. So, let's go over these changes and see how we develop an extender control building on the structure the template provided for us.

The setting of the assembly-based WebResource attribute in our extender class is a pattern that all the extenders and script controls in the toolkit follow. This pattern helps centralize all the pieces for the component in one location instead of having to add an entry to the assembly when a new control is added to the toolkit. The attributes applied to the class that we cover in this section are the Designer, ClientScriptResource, RequiredScript, and TargetControlType attributes. The Designer attribute is used to specify the class that will provide design-time services to our extender. The ClientScriptResource attribute is used to include the client-side scripts for our extender and consists of the resource type and the full resource name and should refer to an embedded resource. The RequiredScriptResource attribute brings in the timer script file that is associated with the Timer Script class that we will use in our behavior class. Finally, the Target ControlType attribute is used to limit the types of controls our extender can be associated with.

The RotationInterval and ImageList properties of our class have also changed with the use of attributes and the reliance on the GetProperty Value<T> and SetPropertyValue<T> methods to access our property data. The ExtenderControlProperty attribute is used to indicate that the property should be added to the ScriptComponentDescriptor as a property and later included in the $create statement that creates the behavior class on the client. The ClientPropertyName attribute is used to change the name of the property that is used when the property is added to the Script ComponentDescriptor from the default value of the property name to the name provided to the attribute. The DefaultValue attribute, which comes from the System.CompnentModel namespace, is used to indicate to designers and code generators the default value of the property. The Extender ControlBase class provides the GetPropertyValue<T> and GetProperty Value<T> generic methods that get and set the property value directly from the control view state. By using these methods in our property setters and getters, a consumer of our extender can work with it in the designer, declaratively in the HTML editor, or in code and be assured that during a postback the values will be available.

Listing 11.4 ImageRotatorExtender Class

```
[assembly:System.Web.UI.WebResource("ImageRotator.ImageRotatorBehavior.js",
"text/javascript")]
namespace ImageRotator
{
  [ParseChildren(true, "ImageList")]
  [Designer(typeof(ImageRotatorDesigner))]
  [ClientScriptResource("ImageRotator.ImageRotatorBehavior",
     "ImageRotator.ImageRotatorBehavior.js")]
  [RequiredScript(typeof(TimerScript))]
  [TargetControlType(typeof(Image))]
  public class ImageRotatorExtender : ExtenderControlBase
  {
    [ExtenderControlProperty]
    [ClientPropertyName("rotationInterval")]
    [DefaultValue(3), DisplayName("RotationInterval(seconds))")]
    [DesignerSerializationVisibility(
      DesignerSerializationVisibility.Visible)]
    public int RotationInterval
    {
      get { return GetPropertyValue<int>("RotationInterval", 3); }
      set { SetPropertyValue<int>("RotationInterval", value); }
    }

    private ImageUrlList _imageList;
    [ExtenderControlProperty]
    [ClientPropertyName("imageList")]
    [DesignerSerializationVisibility(
      DesignerSerializationVisibility.Content)]
    [PersistenceMode(PersistenceMode.InnerDefaultProperty)]
    public ImageUrlList ImageList
    {
      get
      {
        if (_imageList == null)
        {
            _imageList = GetPropertyValue<ImageUrlList>(
             "ImageList", null);
            if (_imageList == null)
            {
               _imageList = new ImageUrlList();
               SetPropertyValue<ImageUrlList>(
                "ImageList", _imageList);
            }
        }
        return _imageList;
      }
    }
  }
}
```

Creating the AjaxControlToolkit.BehaviorBase Class

The ASP.NET AJAX Control Toolkit comes with its own version of the `Sys.UI.Behavior` class, which provides additional functionality and supports the development pattern the toolkit is designed to work with. The `AjaxControlToolkit.BehaviorBase` class provides inheritor support for working with client state and interacting with the asynchronous request events of the `Sys.WebForms.PageRequestManager`. The support for working with client state is provided by the `get_ClientState` and `set_ClientState` methods that can be used to work with the string-based hidden field associated with your extender. The class also provides two methods tied to the `beginRequest` and `endRequest` events of the `PageRequestManager`, which can be overridden to provide specific functionality in your behavior in situations where an UpdatePanel is being used.

The `ImageRotatorBehavior` class shown in Listing 11.5 inherits from the `BehaviorBase` class and provides the client-side behavior for our extender control. The structure of this class is exactly the same as in Chapter 5, with the `rotationInterval` property used to set the interval at which the images will be swapped out and the `imageList` property containing an array of the images. The one change to the class is in the use of the `Sys.Timer` class, which is part of the ASP.NET AJAX Control Toolkit. This class, which is contained in the Compat/Timer folder, wraps the `window.setInterval` call, providing a cleaner interface for this timer-specific functionality. The `Sys.Timer` class is just one of many that come with the toolkit that provide added functionality to the existing Microsoft AJAX Library. If you look in the Compat and Common folders in the toolkit library project, you will find classes for working with dates, drag and drop, and threading, just to name a few.

Listing 11.5 ImageRotator Behavior Class

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

ImageRotator.ImageRotatorBehavior = function(element) {
  ImageRotator.ImageRotatorBehavior.initializeBase(this, [element]);

  this._imageIndex = 0;
  this._imageList = new Array();
  this._rotationInterval = null;
  this._timer = null;
```

LISTING 11.5 continued

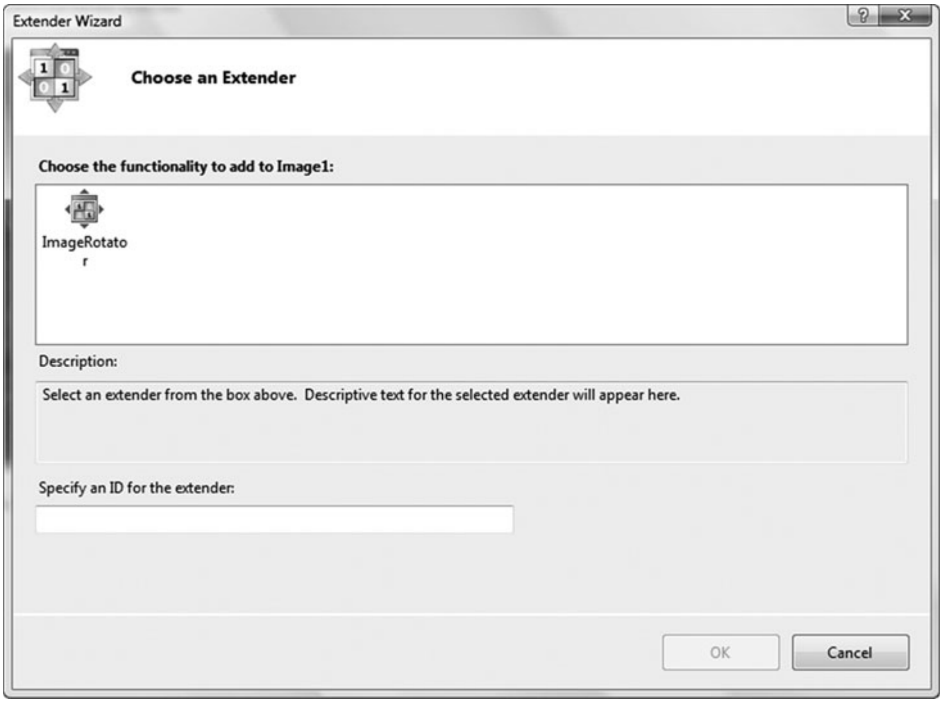
```
}
ImageRotator.ImageRotatorBehavior.prototype = {
  initialize : function() {
    ImageRotator.ImageRotatorBehavior.callBaseMethod(this,
      'initialize');

    var element = this.get_element();

    if(this._imageList)
    {
      this._imageList =
        Sys.Serialization.JavaScriptSerializer.deserialize(
          this._imageList);
      this._imageList[this._imageList.length] = element.src;
    }

    if(this._rotationInterval == null)
      this._rotationInterval = 3;

      if(this._timer == null)
        this._timer = new Sys.Timer();

      this._timer.set_interval(this._rotationInterval * 1000);
      this._timer.add_tick(Function.createDelegate(this,
        this._rotateImage));
      this._timer.set_enabled(true);
  },

  dispose : function() {
    ImageRotator.ImageRotatorBehavior.callBaseMethod(this, 'dispose');
    if (this._timer)
    {
      this._timer.dispose();
      this._timer = null;
    }

    this._imageList = null;

  },
  get_rotationInterval: function(){
    return this._rotationInterval;
  },
  set_rotationInterval: function(value){
    this._rotationInterval = value;
  },
  get_imageList: function(){
    return this._imageList;
  },
  set_imageList: function(value){
    this._imageList = value;
  },
  _rotateImage: function(){
    var element = this.get_element();
    if(element)
    {
      element.src = this._imageList[this._imageIndex++];
      if(this._imageIndex > this._imageList.length - 1)
        this._imageIndex = 0;
    }
  }
}
ImageRotator.ImageRotatorBehavior.registerClass(
  'ImageRotator.ImageRotatorBehavior', AjaxControlToolkit.BehaviorBase);
```

Attaching the Extender to a Control

You can attach the `ImageRotator` extender to an image control by using the new Extender Control Wizard (see Figure 11.3) that comes with Visual Studio 2008 and thus provide the same design-time experience we saw in Chapter 5. The wizard is available from the smart tag of the image control by selecting the Add Extender option, which opens the wizard. The wizard enables the user to select an extender control from a list and associate it with a control. In our case, we would select the `ImageRotator` extender to associate it with the image control. After we do that, we add values to the `RotationInterval` property and `ImageList` property using the Properties window of the image control.

Final Thoughts

If we compare our experience of creating extender controls using the ASP.NET AJAX Control Toolkit to using the classes provided by the ASP.NET 2.0 AJAX Extensions, we can see that our development experience is much simpler. The use of attributes to register our properties to be included in the `$create` statements and to register our associated script files dramatically reduces the complexity of our code compared to implementing logic in the `GetScriptDescriptors` and `GetScriptReferences` methods. This convenience alone makes it worth using the toolkit, but if we tack on the added design-time features, support for working with client state, and the numerous added JavaScript files such as the `Sys.Timer` class, the reasons to switch get greater. The use of the toolkit can be compared to the use of the ActiveX Template Library (ATL) that was used to create ActiveX controls in C++. The template provided a ton of base classes and Visual Studio templates that made creating them a lot easier.


Extender Wizard

Choose an Extender

Choose the functionality to add to Image1:

ImageRotator

Description:

Select an extender from the box above. Descriptive text for the selected extender will appear here.

Specify an ID for the extender:

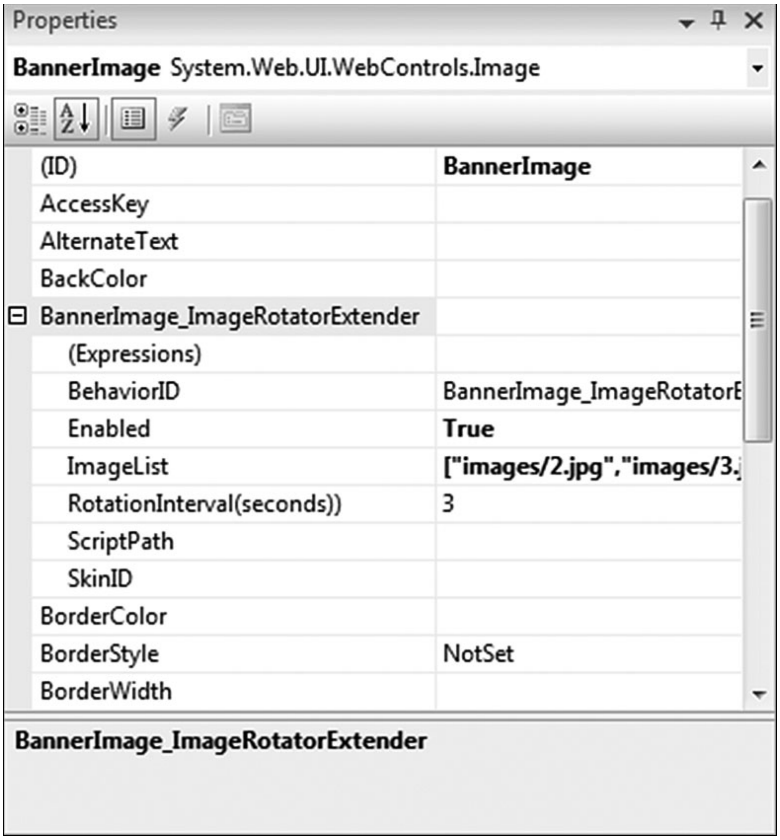
FIGURE 11.3 Extender Control Wizard

Adding Design-Time Support to Your Extender Control

The introduction of the Extender Wizard in Visual Studio 2008 has enhanced the design-time experience with regard to working with extender controls, and this section explains how to add design-time features of your own to give your controls that professional feel that users have become accustomed to.

Default Design-Time Experience

The `ImageRotatorDesigner` class shown in Listing 11.2 provides everything we need to get a basic design-time experience for our extender control. The `ExenderControlBaseDesigner<T>` that it inherits from makes it possible for the properties of our extender control to show up in the Properties window while the design-time focus is on the image control we are extending. Figure 11.4 shows the `RotationInterval` and `ImageList` properties that appear in the Properties window while the image control has focus in the designer. This default feature addresses one issue, which is being able to work with the `ImageRotator` properties in an integrated way, but still does not address the issue of data entry for the properties themselves and how that experience can be enhanced.



Properties

BannerImage System.Web.UI.WebControls.Image

(ID)	BannerImage
AccessKey	
AlternateText	
BackColor	
⊟ BannerImage_ImageRotatorExtender	
(Expressions)	
BehaviorID	BannerImage_ImageRotatorE
Enabled	True
ImageList	["images/2.jpg","images/3.
RotationInterval(seconds))	3
ScriptPath	
SkinID	
BorderColor	
BorderStyle	NotSet
BorderWidth	

BannerImage_ImageRotatorExtender

FIGURE 11.4 Extender properties on the image control

Adding Designers and Editors to Properties

In this section, we look at how to extend the design-time behavior of our `ImageRotator ImageList` property. The `ImageList` property that we worked with in Chapter 5 was rudimentary and prone to errors as a user entered in the values. In this version of the extender, we want to extend the functionality to support design-time editing and HTML source editing.

The road to these modifications requires a few steps as we add the functionality:

1. Add attributes to the class.
2. Add attributes to the property.
3. Add editors to assist in assigning values.
4. Create a type converter to support serialization.

Add Attributes to the Class

Most users expect when adding multiple entries to a control to be able to add them in the body of the HTML element. This is the experience we have when adding web service references or script references to the `Script Manager` and one we want to have in our control.

The `ParseChildren` attribute enables us to add multiple entries inside our `ImageRotator` HTML tag and treat those entries as a single property assignment. By setting the `ChildrenAsProperties` property to `true` and the `DefaultProperty` to `ImageList`, as in Listing 11.6, we are effectively telling the designer that we want to have all the items contained in the body of our `ImageRotator` tag parsed and assigned to the `ImageList` property. The HTML fragment in Listing 11.7 shows what this looks like when the HTML editor is opened and the `ImageRotator` tag has entries.

Listing 11.6 ParseChildren Attribute Assignment

```
[ParseChildren(true, "ImageList")]
...
public class ImageRotatorExtender : ExtenderControlBase
{
    ...
}
```

Listing 11.7 ImageList Assignment in HTML

```
...
<asp:Image ID="BannerImage" runat="server" ImageUrl="~/images/1.jpg" />
<cc2:ImageRotatorExtender ID="BannerImage_ImageRotatorExtender"
  runat="server" Enabled="True" TargetControlID="BannerImage">
  <cc2:ImageUrl Url="~/images/2.jpg" />
  <cc2:ImageUrl Url="~/images/3.jpg" />
  <cc2:ImageUrl Url="~/images/4.jpg" />
</cc2:ImageRotatorExtender>
...
```

NOTE ASP.NET Server Control Designer References

The addition of designer features to your extenders requires some knowledge of how designers work. MSDN has some great information about this at http://msdn2.microsoft.com/en-us/library/aa719973%28VS.71%29.aspx that covers adding design-time support to ASP.NET server controls.

Add Attributes to the Property

To fully implement the ability to add nested image entries to our `Image Rotator` extender, we need to add a couple of attributes, as shown in Listing 11.8, to our `ImageList` property, which provides hooks for the designer to integrate with our property and properly assign the image values.

The `DesignerSerializationVisibility` attribute is added to the property to ensure that the designer will serialize the contents of the property during design time. The setting of `DesignerSerializationVisibility.Content` instructs the designer to generate code for the contents of the tag and not the tag itself.

The `PersistenceMode` attribute is the other piece to this puzzle and is responsible for adding the `<ImageUrl .. />` entries inside our `ImageRotator` tag as we add values to the property in the Properties window. The setting of `PersistenceMode.InnerProperty` specifies that the property is persisted as a nested tag inside the `ImageRotator`, as shown in Listing 11.7.

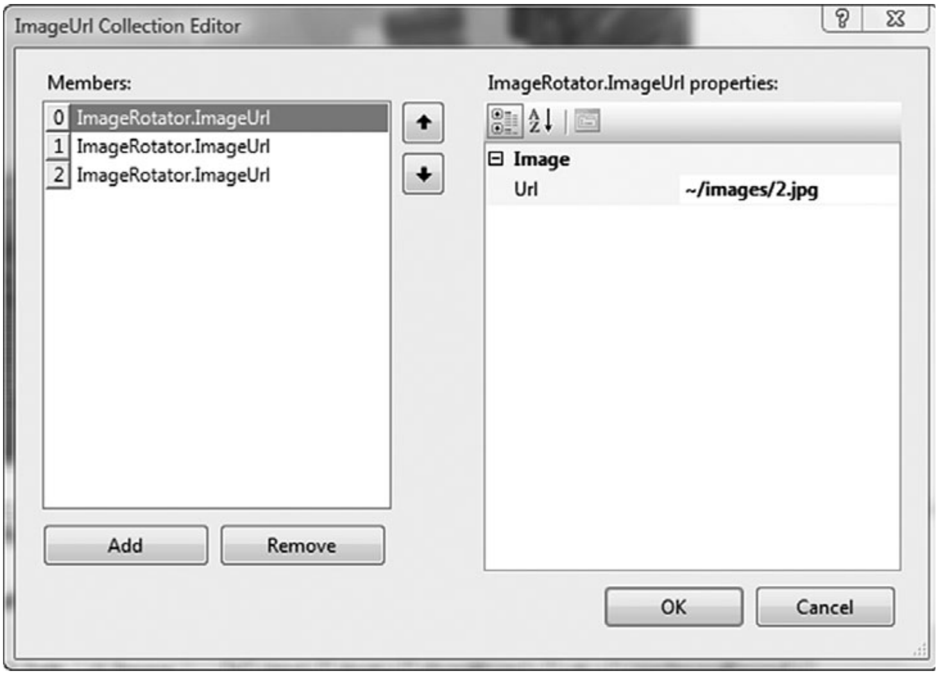
LISTING 11.8 Designer-Specific Attributes for the ImageRotatorExtender Class

```
[ParseChildren(true, "ImageList")]
...
public class ImageRotatorExtender : ExtenderControlBase
{
  ...

  [DesignerSerializationVisibility(
    DesignerSerializationVisibility.Content)]
  [PersistenceMode(PersistenceMode.InnerDefaultProperty)]
  public ImageUrlList ImageList
  {
    ...
  }
}
```

Add Editors to Assist in Assigning Values

The use of editors in your extenders can greatly enhance the user experience during design time and in some cases can lead to more accurate entry of data. Recall from Chapter 5 that we entered images to the `ImageList` property by adding URL entries and separating them using commas. This rudimentary approach would not be expected by a consumer of a professional control. In this version of the `ImageRotator`, we want to enhance the data entry of the images by providing an editor that can be used to add image URL entries and have those entries placed into the body of our `ImageRotator` HTML tag. If we go back to the `ScriptManager` control, this is the experience it provides when adding web service or script references while in the Properties window.

The `ImageList` property in this version of the `ImageRotator` uses two editors to provide a rich design-time experience when adding `ImageUrl` entries. The first editor is a `Collection` editor, shown in Figure 11.5, and is designed to assist in adding, editing, and removing values that are based on a `Collection`. The editor is automatically associated with our `ImageList` property because the type of the property is a `Collection`. The second editor we will use is the `ImageUrlEditor`, shown in Figure 11.6, which the `ImageUrl` entry uses to assist the user in entering a URL. This editor is associated with the `Url` property of the `ImageUrl` class, as shown in Listing 11.9, by adding the `Editor` attribute to the property. We use the `Editor` attribute to configure which editor to use when adding values to the property in the designer. In our case, we are using the `ImageUrlEditor` to provide the user with a clean way to find an image located in a web application and assign the value to the `ImageUrl` property. The use of the associated `UrlProperty` attribute provides a filter that identifies specific file types that can be used to filter against the `ImageUrl` property.


ImageUrl Collection Editor

Members:

0 ImageRotator.ImageUrl
1 ImageRotator.ImageUrl
2 ImageRotator.ImageUrl

ImageRotator.ImageUrl properties:

Image
Url ~/images/2.jpg

FIGURE 11.5 Image URL Collection Editor

LISTING 11.9 ImageUrl Class

```
[Serializable]
public class ImageUrl
{
  [DefaultValue(""),Bindable(true),
    Editor("System.Web.UI.Design.ImageUrlEditor,
      System.Design, Version=2.0.0.0, Culture=neutral,
      PublicKeyToken=b03f5f7f11d50a3a", typeof(UITypeEditor)),
      UrlProperty]
  public string Url { get; set; }
}
```

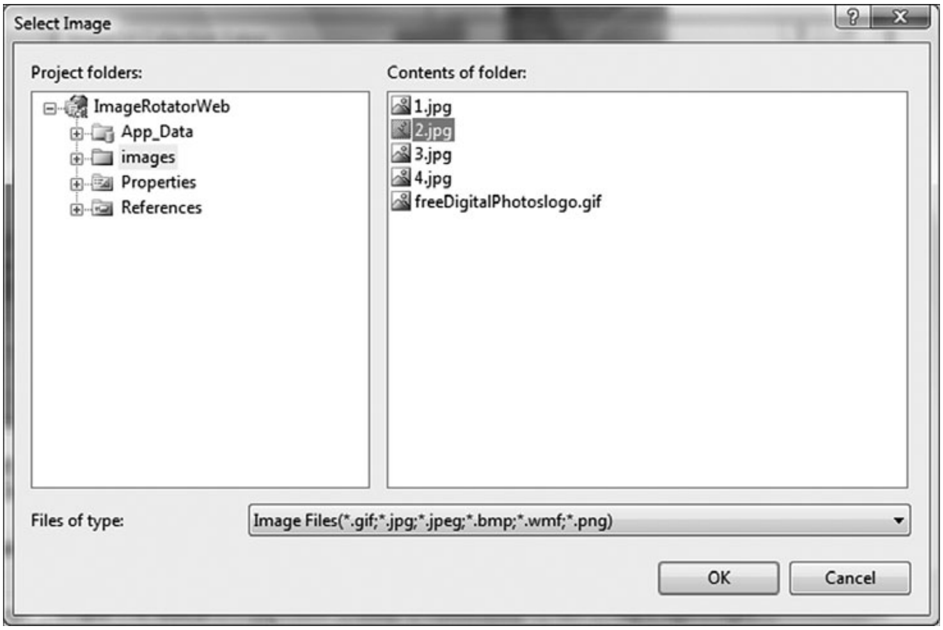


FIGURE 11.6 Image URL Editor

Create a Type Converter to Support Serialization

The use of complex types presents a few challenges in the ASP.NET AJAX Control Toolkit during the script generation process. The problem arises in how the ASP.NET AJAX Control Toolkit serializes your extender control properties as it creates the `$create` statement. By default, the toolkit tries to get the default string representation of the value your property represents. In most cases, this is an easy task because most simple types convert to a string relatively easily. If you are using complex types, however, this can present a problem because the default `ConvertToString()` representation of a complex object is its type name. To resolve this issue, you must create a type converter and associate it with the complex type. When the ASP.NET AJAX Control Toolkit encounters a type during script generation, it looks to see whether the type has a converter. If it does, it uses the converter to convert the data instead of using the default `ConvertToString()` method. In this section, we walk through creating a `System.ComponentModel.TypeConverter` that will be used to convert our `ImageUrlList` type into a JavaScript Object Notation (JSON) string that can be consumed on the client.

The ImageListConverter, shown in Listing 11.10, is designed to convert the ImageList to a JSON array of image URLs that are then passed back to the client. The creation of this type converter now enables us to return a data format that the client can use instead of a string that contains the type name of the ImageList. For the type converter to be used, we need to associate it with the ImageList type. We do this by adding the TypeConverter attribute to the ImageList class, as shown in Listing 11.11, and assigning the type of the ImageList to it. Now when the toolkit performs a Convert ToString on the ImageList, the JSON string representation of the Image List will be returned.

NOTE Use of the DataContractJsonSerializer

In more complex situations, you might use the DataContract JsonSerializer that we discussed in Chapter 8, "ASP.NET AJAX Communication Architecture," which replaces theSystem.Web.UI. JavaScriptSerializer class as the new JSON serializer to convert your data to JSON format.

LISTING 11.10 ImageListConverter Type Converter Class

```
public class ImageListConverter : TypeConverter
{
  public override object ConvertTo(ITypeDescriptorContext context,
    System.Globalization.CultureInfo culture, object value,
    Type destinationType)
  {
    Collection<ImageUrl> imageList = value as Collection<ImageUrl>;
    if (imageList != null && destinationType == typeof(string))
    {
      StringBuilder builder = new StringBuilder();
      builder.Append("[");
      bool first = true;
      foreach (ImageUrl imageUrl in imageList)
      {
        if(first)
        {
          first = false;
        }
        else
```

LISTING 11.10 continued

```
          {
            builder.Append(",");
          }

          builder.Append("\"");
          builder.Append(imageUrl.Url.Replace("~/", ""));
          builder.Append("\"");
        }
        builder.Append("]");
        return builder.ToString();
      }
      return base.ConvertTo(context, culture, value, destinationType);
    }
  }
```

LISTING 11.11 ImageUrlList Collection Class

```
  [Serializable]
  [TypeConverter(typeof(ImageListConverter))]
  public class ImageUrlList : Collection<ImageUrl>
  {

  }
```

Adding Animations to Your Extender Control

The ASP.NET AJAX Control Toolkit comes with a rich animation framework that provides support for creating cool visual effects on your pages. The animation framework consists of a set of JavaScript and .NET classes that enable you to build up animations of all types, including animations that run sequentially or in parallel, animations that fade the opacity of a control in and out, and animations that transition from one color to the next. The framework provides support for building these animations using the JavaScript API directly or using a declarative approach that consists of adding markup in the HTML editor. The following sections examine how to add animation functionality to extender controls.

Animations Using the JavaScript API

The ImageRotator extender we created earlier provided little in the area of effects as the images switched and resulted in very fast transition from one image to the next, which wouldn't catch a viewer's attention. In this section, we create a new version of the ImageRotator, called the AnimatedImageRotator, that fades in the image as it switches from one image to the next and provides this feature in addition to the existing functionality of the ImageRotator. As we cover how to add this new animation functionality, we gloss over the topics we have already covered, focusing only on implementing the animation pieces.

To add this functionality to the AnimatedImageRotator, we need to register the animation scripts with the AnimatedImageRotatorExtender class and add logic to the behavior class to call the animation when the image changes.

Registering the Animation Scripts

To register the script files so that they are downloaded to the browser, we need to add the RequiredScript attribute to the AnimatedImageRotatorExtender class, as shown in Listing 11.12. We use the RequiredScript attribute in this case to ensure that the animation.js, timer.js, and common.js script files associated with the AnimationScripts type are included with the scripts brought down to the browser for our control. This style of adding scripts associated with a type is a common practice in the toolkit and is clean way to include dependent scripts associated with a type.

LISTING 11.12 AnimatedImageRotator Extender Class

```
...
[RequiredScript(typeof(AnimationScripts))]
...
public class AnimatedImageRotatorExtender : ExtenderControlBase
{
  ...
}
```

Calling Animation APIs

The ASP.NET AJAX Control Toolkit contains a JavaScript API that you can use to provide animation support on the client. In the case of our `AnimatedImageRotator` extender, we will use the `FadeAnimation`, which is part of the animation API, to provide a fade-in effect when the images on our image control change. The JavaScript code to implement this functionality will be contained in our behavior class and will integrate with the existing features of the `ImageRotator`.

The `AnimatedImageRotator` behavior class, shown in Listing 11.13, takes the `ImageRotator` behavior and adds a fade animation when the image changes, to fade the image into view. The constructor of the `FadeAnimation` takes the target of the animation, the duration of the animation, the number of steps per second, the effect, the minimum opacity, the maximum opacity, and whether to adjust for layout in Internet Explorer. In our case, the `BannerImage` image control will be the target of our animation, and the duration of our animation will be hard coded to 20% of the time the image is visible. To provide a clean animation, we will set the animation steps to 150, and combine that with a fade-in effect that will cause the image to transition in when the image changes. During this transition, we will start off with an opacity of 0, which will give us a full view of the image background, and then through the 150 steps work our way to a full view of the image with an opacity of 1. Table 11.1 lists some of the `FadeAnimation` properties and provides a little more information about what they do.

After we associate the animation to the element, starting, stopping, and pausing the animation is just a method call away, making it simple to manipulate the animation. In the `AnimatedImageRotator`, the load event of the image is used to trigger the animation to play because it will be fired each time our `Sys.Timer` calls the `_rotateImage` method. To do this, we associated the `_onLoadImage` event handler with the `onLoad` event of the image and called the play method on the animation inside the function. Now each time the load event occurs, the animation plays, transitioning the image into view. One of the side effects of working with an animation in a situation like this is a potential race condition if the duration was set too long. When working with transition-type animations like the `FadAnimation`, pay close attention to how you are using it to ensure the animation will work in all cases.

Listing 11.13 AnimatedImageRotator Behavior Class

```
...

AnimatedImageRotator.AnimatedImageRotatorBehavior = function(element) {
  ...
  this._fadeAnimation = null;
  this._timer = null;
  this._onImageLoadHandler = null;
}
AnimatedImageRotator.AnimatedImageRotatorBehavior.prototype = {
  initialize : function() {
  ...

    if(this._fadeAnimation == null)
    {
      this._fadeAnimation =
        new AjaxControlToolkit.Animation.FadeAnimation(
          element, this._rotationInterval/20, 150,
          AjaxControlToolkit.Animation.FadeEffect.FadeIn,
          0, 1, true);
    }
    if (element)
    {
      this._onImageLoadHandler = Function.createDelegate(this,
        this._onImageLoad);
      $addHandler(element, 'load', this._onImageLoadHandler);
    }
    ...
  },

  dispose : function() {
    ...
    var element = this.get_element();
    if (element) {
      if (this._onImageLoadHandler) {
        $removeHandler(element, 'load',
          this._onImageLoadHandler);
        this._onImageLoadHandler = null;
      }
    }

    ...

    if (this._fadeAnimation)
    {
      this._fadeAnimation.dispose();
      this._fadeAnimation = null;
    }

    ...
```

Listing 11.13 continued

```
    },
    _onImageLoad: function(){
      if(this._fadeAnimation)
        this._fadeAnimation.play();
    },
    ...
  }
  ...
```

Table 11.1 Partial List of Fade Animation Class Properties

Property	Description
`target`	Target of the animation.
`duration`	Length of the animation in seconds. The default is `1`.
`fps`	Number of steps per second. The default is `25`.
`effect`	Determine whether to fade the element in or fade the element out. The possible values are `AjaxControlToolkit.Animation.FadeEffect.FadeIn` and `AjaxControlToolkit.Animation.FadeEffect.FadeOut`. The default value is `FadeOut`.
`minimumOpacity`	Minimum opacity to use when fading in or out. Its value can range from 0 to 1. The default value is `0`.
`maximumOpacity`	Maximum opacity to use when fading in or out. Its value can range from 0 to 1. The default value is `1`.
`forceLayoutInIE`	Whether we should force a layout to be created for Internet Explorer by giving it a width and setting its background color (the latter is required in case the user has ClearType enabled). The default value is `true`.

Animations Using the Declarative Method

The declarative approach to animation in the toolkit provides a nice extensibility path for consumers of your extender. In our previous example, we hard coded all the animation functionality inside our extender, providing little support for developer customization. In some cases, this might be all that is needed. In other cases, however, you might need to provide a more robust solution that provides a JavaScript-free way to customize animations. In this section, we replicate the same functionality we created in the preceding section, but we provide a more extensible approach consumers of our extender can use when they are configuring it in the designer. The extender we create has just one feature: the capability to run a `FadeIn` animation when the `onLoad` event of an associated image control occurs. This new extender will be used in addition to the `ImageRotator` extender we created earlier, which had no animation functionality. This refined approach to adding animation support builds on the principle that many extenders can be placed on a single control to provide combined client-side capabilities. To get started, let's take a look at what the declarative syntax or our control will look like before we go into the implementation details. Just as in the preceding section, as we cover how to add this new animation functionality we gloss over the topics we have already covered, focusing only on implementing the declarative animation pieces.

Overview of Declarative Syntax

To get started, let's look at the HTML source we will be working toward being able to work with in our `ImageAnimation` extender. The source in Listing 11.14 contains an `ImageAnimationExtender` tag that contains in its body an `Animations` tag. As you might guess, the approach here is to add various animations that are driven by events raised by the image control we are extending. In our case, we are working with the `OnLoad` event and adding a `Sequence` animation that will call a child `Fade` animation. A `Sequence` animation is designed to run all its child animations one at a time until all have finished. So, what this source tells us is that our extender will have an animation that will be tied to the `OnLoad` event of the image control and will run the child `Fade` animation whenever the `OnLoad` event occurs.

Listing 11.14 AnimationImageExtender Declarative Syntax

```
<asp:Image ID="BannerImage" runat="server" ImageUrl="~/images/1.jpg" />
<cc3:ImageAnimationExtender ID="Banner_ImageAnimationExtender"
  runat="server" Enabled="True" TargetControlID="BannerImage">
  <Animations>
    <OnLoad>
      <Sequence>
        <FadeIn AnimationTarget="BannerImage" Duration=".3"/>
      </Sequence>
    </OnLoad>
  </Animations>
</cc3:ImageAnimationExtender>
<cc2:ImageRotatorExtender ID="Image1_ImageRotatorExtender"
  runat="server" Enabled="True" TargetControlID="Banner">
  <cc2:ImageUrl Url="~/images/2.jpg" />
  <cc2:ImageUrl Url="~/images/3.jpg" />
  <cc2:ImageUrl Url="~/images/4.jpg" />
</cc2:ImageRotatorExtender>
```

Providing Declarative Support in Your Extender Class

The `AnimationExtenderControlBase` class provides most of the functionality we need to parse the `Animation` tag and all its contents. This class provides internal methods that convert the XML representation of the animation into JSON format, which our behavior will then use to run the animation, and also provides the `Animation` property that we see in Listing 11.15. The following sections cover the steps needed to ensure the extender will work correctly.

1. Add attributes to the class.
2. Create a property for the event.
3. Add attributes to the property.

Add Attributes to the Class

This type of extender has a couple of added class attribute entries of interest to us. The first is the inclusion of the `RequiredScript` attribute for the `AnimationExtender` type. The `AnimationExtender` class provides a lot of the client-side functionality we will be using in our extender control, and by using this type in our `RequiredScripts` attribute, we are guaranteed that the scripts will be present on the client for us to use. The second attribute is the `System.Web.UI.Design.ToolboxItem` attribute, which enables our control to show up in the toolbox of Visual Studio. It might seem strange that we have to add this because all our other extenders didn't. If we look at the attributes on the `AnimationExtenderControlBase` class, however, the support for viewing in the toolbox has been turned off. Therefore, we must reset this value on our control so that it will show up in the toolbox.

Create a Property for the Event

The pattern when creating extenders of this type is to add a property for each event you want to interact with. In our case, we are working with the `OnLoad` event, so we create a property named `OnLoad` (to make it easy to understand what the event is). If we were to choose other events, we would name them based on the DOM event they represent. The property accessor for these events must use the `GetAnimation` and `SetAnimation` methods to ensure proper data conversion into JSON as the data is stored and retrieved out of the extender's view state.

Add Attributes to the Event Property

The event property must have the `Browsable`, `DefaultValue`, `ExtenderControlProperty`, and `DesignerSerializationVisibility` attributes applied to it. The `Browsable` attribute stops the property from showing up in the Properties window and therefore excludes the property from being assigned in the Properties window. This is needed because no editor is associated with this property, and we don't want users to try to add anything into the Properties window that would corrupt the values. The `DesignerSerializationVisibility` attribute with a value of `DesignerSerializationVisibility.Hidden` is used to indicate that the property value should not be persisted by the designer because the `Animation` property will take care of that for us. The `DefaultValue` attribute indicates to the designer that the default value will be null, and the `ExtenderControlProperty` attribute is used to register the property with the `ScriptComponentDescriptor`.

Listing 11.15 ImageAnimationExtender Class

```
[Designer(typeof(ImageAnimationDesigner))]
[ClientScriptResource("ImageAnimation.ImageAnimationBehavior",
  "ImageAnimation.ImageAnimationBehavior.js")]
[RequiredScript(typeof(AnimationExtender))]
[ToolboxItem("System.Web.UI.Design.WebControlToolboxItem, System.Design,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a")]
[TargetControlType(typeof(Image))]
public class ImageAnimationExtender : AnimationExtenderControlBase
{
  private Animation _onLoad;

  [DefaultValue(null)]
  [Browsable(false)]
  [ExtenderControlProperty]
  [DesignerSerializationVisibility(
     DesignerSerializationVisibility.Hidden)]
  public new Animation OnLoad
  {
    get { return GetAnimation(ref _onLoad, "OnLoad"); }
    set { SetAnimation(ref _onLoad, "OnLoad", value); }
  }
}
```

Adding Declarative Support to Your Behavior Class

The `ImageAnimationBehavior` class, shown Listing 11.16, provides all the client-side functionality for our extender with support from the animation script files associated with the `AutomationExtender` class. These associated scripts provide support for converting the JSON representation of the `FadeIn` animation that was captured on the server to an actual animation, support for associating the animation with the high-level `OnLoad` event, and support for playing the animation when the `OnLoad` event occurs.

You need to complete a few steps for each event you plan to work with:

1. Add variables to the class.
2. Create functions.
3. Add handlers.

Add Variables to the Class

Each event that your behavior will work with needs a variable that references the `GenericAnimationBehavior` for the event and a delegate that will be called for the event that will be processed. In the ImageAnimation Behavior class, we use the _onLoad variable to store a reference to the GenericAnimationBehavior class and the _onLoadHandler variable to store a reference to the delegate that will handle the onLoad event. The guidelines established so far in the toolkit use a naming convention that includes the event name in all the variable names.

Create Functions

The behavior needs a series of functions for each event you will work with. The get_OnLoad and set_OnLoad functions in our case take care of working with the JSON-based data for the FadeIn animation and utilize the functionality provided by the GenericAnimationBehavior class to store and retrieve that data. The get_OnLoadBehavior function returns a reference to the GenericAnimationBehavior instance that was created for our FadeIn animation, providing the ability to work with the behavior that directly exposes the play, stop, and quit methods common to all animations.

Add Handlers

Handlers must be added for each event the behavior will process and should correspond to the events exposed on the extender control. In our case, we are working with the onLoad event, so we need to create the _onLoadHandler delegate and associate it with the onLoad event of the image using the $addHandler shortcut. The opposite of this must happen in the dispose of our behavior, when we use the $removeHandler shortcut to ensure proper memory cleanup.

Listing 11.16 ImageAnimationBehavior Class

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

ImageAnimation.ImageAnimationBehavior = function(element) {
  ImageAnimation.ImageAnimationBehavior.initializeBase(this, [element]);
  this._onLoad = null;
  this._onLoadHandler = null;
}
ImageAnimation.ImageAnimationBehavior.prototype = {
  initialize : function() {
    ImageAnimation.ImageAnimationBehavior.callBaseMethod(this,
      initialize');
    var element = this.get_element();
```

LISTING 11.16 continued

```
        if (element)
        {
          this._onLoadHandler = Function.createDelegate(this,
           this.OnLoad);
          $addHandler(element, 'load', this._onLoadHandler);
        }
      },

      dispose : function() {
        ImageAnimation.ImageAnimationBehavior.callBaseMethod(this,
          'dispose');

        var element = this.get_element();
        if (element) {
          if (this._onLoadHandler) {
            $removeHandler(element, 'load', this._onLoadHandler);
            this._onLoadHandler = null;
          }
        }

        this._onLoad = null;

      },
      get_OnLoad : function() {
        return this._onLoad ? this._onLoad.get_json() : null;
      },
      set_OnLoad : function(value) {
        if (!this._onLoad) {
          this._onLoad = new
            AjaxControlToolkit.Animation.GenericAnimationBehavior(
              this.get_element());
          this._onLoad.initialize();
        }
        this._onLoad.set_json(value);
          this.raisePropertyChanged('OnLoad');
      },
      get_OnLoadBehavior : function() {
        return this._onLoad;
      },
      OnLoad : function() {
        if (this._onLoad) {
          this._onLoad.play();
        }
      }
    }
    ImageAnimation.ImageAnimationBehavior.registerClass(
      'ImageAnimation.ImageAnimationBehavior',
      AjaxControlToolkit.BehaviorBase);
```

Final Thoughts

The HTML source for our sample, shown Listing 11.14, contains a `Fade` animation that targets the `BannerImage` control and runs for a duration of .3 seconds. We could have chosen almost any type of animation as long as it occurred when the `OnLoad` event fired on the `BannerImage` image control. This flexibility provides a JavaScript-free way to set up animations of any type when a pattern such as this is used. In fact, this is exactly how the `Animation` extender works; and if it weren't for the way it handles the `OnLoad` event, we would have used it in our example.

SUMMARY

The AJAX Control Toolkit comes with quite a bit of functionality that you can use to create truly interactive extenders that require much less coding than if you were to use the ASP.NET 2.0 AJAX Extensions directly. As you learned in this chapter, the toolkit provides a much richer environment for creating extender controls than using the ASP.NET 2.0 AJAX Extensions alone. In addition, the toolkit includes myriad controls you can either use or build on, making the toolkit a compelling alternative.

Index