IN ACTION

Jeffrey Palermo Jimmy Bogard Eric Hexter Matthew Hinze Jeremy Skinner

FOREWORD BY Phil Haack







ASP.NET MVC 4 in Action

Jeffrey Paleermo, Jimmy Bogard Eric Hexter, Matthew Hinze and Jeremy Skinner

Chapter 7

brief contents

PART 1	HIGH-SPEED FUNDAMENTALS
	1 Introduction to ASP.NET MVC 3
	2 Hello MVC world 12
	3 View fundamentals 38
	4 • Action-packed controllers 59
PART 2	WORKING WITH ASP.NET MVC79
	5 View models 81
	6 Validation 92
	7 Ajax in ASP.NET MVC 104
	8 Security 135
	9 Controlling URLs with routing 153
	10 Model binders and value providers 185
	11 Mapping with AutoMapper 197
	12 Lightweight controllers 207
	13 • Organization with areas 220
	14 Third-party components 232
	15 Data access with NHibernate 244

vi BRIEF CONTENTS

PART 3	MASTERING	ASP.NET MVC	265
	16	Extending the controller 267	
	17	Advanced view techniques 276	
	18	Dependency injection and extensibility 294	
	19	Portable areas 311	
	20	Full system testing 321	
	21	Hosting ASP.NET MVC applications 339	
	22	Deployment techniques 365	
	23	Upgrading to ASP.NET MVC 4 374	
	24	ASP.NET Web API 385	

Ajax in ASP.NET MVC

This chapter covers

- Unobtrusive Ajax using jQuery
- ASP.NET MVC's Ajax helpers
- JSON responses and client-side templates
- jQuery UI's Autocomplete plugin

Most of the examples that we've looked at so far have focused on using the serverside components in ASP.NET MVC to render views and send them to the browser. But with the increased performance of modern web browsers, we can often move much of our rendering logic to the client. This can result in applications that are far more interactive and user friendly.

Although there are many client-side technologies available on today's web (including Adobe Flash and Microsoft Silverlight), the most popular is undoubtedly JavaScript due to its ubiquitous support across all modern web browsers. Today, many web applications rely heavily on JavaScript to produce rich user experiences that can almost mimic the instant responses of a desktop application (popular examples include Gmail, Facebook, and Twitter) and Ajax is one technique that can be used to achieve this.

Ajax is a term initially coined by Jesse James Garrett to describe the technique of using JavaScript to make an asynchronous request with a web server and dynamically

update a section of the page with the result, all without having to do a full-page refresh. You make these calls from the client, and the server running ASP.NET MVC can generate the content that the client-side code can then use to manipulate the page.

In this chapter, we'll examine how Ajax can be used with ASP.NET MVC to add client-side interactivity to a page. We'll explore using the popular jQuery library to create Ajax requests as well as using ASP.NET MVC's built-in Ajax helpers. Finally, we'll look at how Ajax can be combined with client-side templates to generate markup on the fly in order to simplify the repetitive process of constructing HTML elements through JavaScript.

The "X" in Ajax

The term "Ajax" was initially an acronym that stood for Asynchronous JavaScript and XML, where data was returned asynchronously from the server in XML format. However, modern web applications rarely use XML due to its verbosity and instead opt for sending data in JSON format, which we'll explore later in this chapter.

7.1 Ajax with jQuery

Working with JavaScript in web applications is becoming increasingly important because of the increased focus on having a rich-client experience. Unfortunately, working with raw JavaScript can be a demanding process. Different browsers have different features and limitations that can make writing cross-browser JavaScript a fairly involved process (for example, Internet Explorer uses a different mechanism for attaching events to elements than other browsers). In addition to this, navigating and manipulating the HTML DOM¹ can be fairly verbose and complex. This is where JavaScript libraries come in.

There are many popular JavaScript libraries today (including jQuery, Prototype, MooTools, and Dojo) all of which aim to make working with JavaScript easier and help normalize cross-browser JavaScript functionality. For the examples in this section, we'll be using the open source jQuery library (http://jquery.com).

jQuery was initially released by John Resig in 2006, and it has become one of the most popular JavaScript libraries due to its simple yet powerful mechanisms for interacting with the HTML DOM. In fact, jQuery has become so popular that Microsoft has contributed several features to its codebase and provides official support for it as well as shipping it as part of ASP.NET MVC's default project template.

In this section, we'll first look at the basics of using jQuery and at how it can be used to make asynchronous calls to the server that can be processed by ASP.NET MVC. We'll then look at how *progressive enhancement* can be used to ensure clients without scripting enabled can still use our site. Finally, we'll see how jQuery can be used to submit form data back to the server in an asynchronous fashion.

DOM stands for "Document Object Model." It's a hierarchy of objects that represents all of the elements in a page.

7.1.1 jQuery primer

When working with jQuery, you mainly work with the jQuery function (primarily using the \$ alias) that can perform a variety of different operations depending on its context. For example, to use jQuery to find all of the <div /> elements on a page and add a CSS class to each one, you could use the following line of code:

```
$('div').addClass('foo');
```

When you pass a string to the \$ function, jQuery will treat it as a CSS selector and attempt to find any elements in the page that match this selector. In this case, it will find all the <div /> elements in the page. Likewise, calling \$('#foo') would find the element whose ID is foo, whereas a call to \$('table.grid td') would find all of the elements nested within tables that have a class of grid.

The result of calling this function is another instance of the jQuery object that wraps the underlying DOM elements that matched the selector. Because it returns another jQuery instance, you can continue to chain calls to jQuery methods that in turn allow you to perform complex operations on DOM elements in a very succinct manner. The preceding example calls the addClass method, which adds the specified CSS class to each element contained in the wrapped set (in this example, all of the <div /> elements in the page).

You can also attach events to elements in a similar fashion. If you wanted to show a message box when a button was clicked, one approach could be to place the JavaScript inline in an onclick event:

```
<button id="myButton" onclick="alert('I was clicked!')">
   Click me!
</button>
```

The downside of this approach is that it mixes code with markup. This can impact the maintainability of your application and make the logic difficult to follow. Using jQuery, you can attach an event handler to the button's click event externally.

```
<button id="myButton">Click me!</button>
<script type="text/javascript">
    $('button#myButton').click(function() {
       alert('I was clicked!');
    });
</script>
```

This example introduces a script element within the page to contain the JavaScript code and tell jQuery to find any <button /> elements with an id of myButton and run a function when the button is clicked. In this case, the browser will simply display a message indicating that the button was clicked.

This approach is known as *unobtrusive JavaScript*. By keeping the site's markup separate from its behavior (code), maintainability is improved and it's easier to follow the flow of the code.

In the same way that you can attach events to elements, you can also attach a ready event to the entire page. This event will be fired once the page's DOM hierarchy has

been loaded, which is the earliest possible point when it's safe to interact with HTML elements. As such, it's better that all event bindings and other jQuery code are contained within in the ready handler:

```
$(document).ready(function() {
   $('button#myButton').click(function() {
     alert('Button was clicked!');
   });
});
```

The end result here will be exactly the same as in the previous example, but it is safer because you ensure that the DOM has been loaded before the event handler is attached to the button.

Although working with jQuery is a subject for an entire book, knowing these core concepts should enable you to understand the following examples. For a more indepth look at jQuery, you may wish to read jQuery in Action, Second Edition by Bear Bibeault and Yehuda Katz, also from Manning publications.

7.1.2 Using jQuery to make Ajax requests

To demonstrate how to use jQuery to make Ajax requests, we'll begin by creating a new ASP.NET MVC project using the default Internet Application template and adding a simple controller. This controller will have two actions that will both render views—one called Index and the other called PrivacyPolicy.

The Index action will contain a hyperlink that, when clicked, will make a request back to the server to get the privacy policy and then load its contents into our index page. The desired result is shown in figure 7.1.



Figure 7.1 The privacy policy will be loaded when the link is clicked.

The code for this controller is shown in the following listing.

Listing 7.1 A simple controller

```
public class CustomAjaxController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
    public ActionResult PrivacyPolicy()
    {
        return PartialView();
    }
}

    Renders a
    partial view
}
```

Note that we return a partial view from the PrivacyPolicy action ① so that the site's layout isn't applied to the view. This ensures that the surrounding chrome (such as the menu) that's inside the layout page is not included in the markup returned from our action.

The PrivacyPolicy partial view contains some very basic markup:

```
<h2>Our Commitment to Privacy</h2>
...privacy policy goes here...
```

The contents of the index view are as follows.

Listing 7.2 The index view including script references

```
@section head {

<script type="text/javascript"
    src="@Url.Content("~/scripts/AjaxDemo.js")">
</script>

@Html.ActionLink("Show the privacy policy",
    "PrivacyPolicy", null, new { id = "privacyLink" })

<div id="privacy"></div>

Tag the head section

    Reference demo code

Container for results
```

We begin by defining the head section to render ①. Newly created MVC projects automatically include the latest version of jQuery using a NuGet package, which makes it very easy to update jQuery when a new release is available. At the time of writing, jQuery 1.7.2 is the latest version, and the appropriate scripts reside within the Scripts subdirectory. We wrap the path in a call to Url. Content rather than using an absolute path to ensure that the path will be correctly resolved at runtime, irrespective of whether the site is running in the root of a website or a subdirectory.

Secondly, we have another script reference 2 that points to a custom JavaScript file called AjaxDemo.js which we haven't yet created. This file will hold our custom jQuery code.

Next, we declare a standard ASP.NET MVC action link ③. The arguments in order are the text for the hyperlink, the action that we want to link to (in this case, our PrivacyPolicy action), any additional route parameters (in this case there aren't any, so we can pass null), and finally an anonymous type specifying additional HTML attributes (in this case we simply give the link an ID).

Finally, we have a div with an id of privacy 4, which is where our privacy policy will be inserted after the Ajax request has fired.

Now we can create the AjaxDemo.js file in our Scripts directory. In this file, we can add some jQuery code to intercept the click of the privacyLink, as follows.

Listing 7.3 Custom jQuery code in the AjaxDemo.js file

```
$(document).ready(function () {
    $('#privacyLink').click(function (event) {
        event.preventDefault();

    var url = $(this).attr('href');
    $('#privacy').load(url);
    });
});
```

We begin by creating a document-ready handler 1 that will be invoked once the DOM has loaded. Inside this handler, we tell jQuery to look for a link with the id of privacyLink and attach a function to its click event 2.

The click handler accepts a reference to the event as a parameter. We call the preventDefault method on this object to prevent the default behavior of the link from occurring (that is, going to the page specified in the link's href attribute). Instead, we extract the value of the href attribute 3 and store it in a variable called url.

The final line of the event handler issues the actual Ajax request 4. This line tells jQuery to find an element on the page with the id of privacy (which refers to the <div /> element we created in listing 7.2) and then load into this element the contents of the URL we extracted from the link. This load method internally creates an Ajax request, calls the URL asynchronously, and inserts the response into the DOM.

When you run the application and click on the link, you should see the privacy policy inserted into the page. If you use the Firefox web browser and also have the Firebug extension installed (from http://getfirebug.com), you can easily see the Ajax request being made, as illustrated in figure 7.1.

This is an example of unobtrusive JavaScript—all of the JavaScript code is kept out of the page in a separate file.

7.1.3 Progressive enhancement

The previous example also illustrates another technique called *progressive enhancement*. Progressive enhancement means that we begin with basic functionality (in this case, a simple hyperlink) and then layer additional behavior on top (our Ajax functionality). This way, if the user doesn't have JavaScript enabled in their browser, the link will

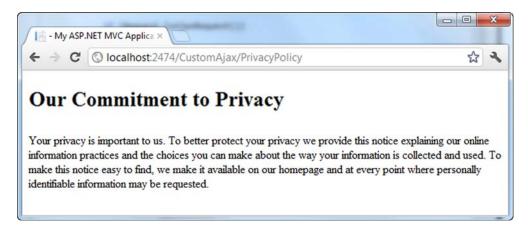


Figure 7.2 The browser goes directly to the Privacy Policy page if JavaScript is disabled.

gracefully degrade to its original behavior and instead send the user to the privacy policy page without using Ajax, as shown in figure 7.2.

Unfortunately, this page doesn't look very nice. We are currently rendering this page as a partial view in order to strip away the additional page chrome (added by our application's layout) so that it can be easily inserted into the DOM by our Ajax request. However, in the case where JavaScript is disabled, it would be nice to continue to include the page layout and associated styling. Thankfully, it is easy to modify our PrivacyPolicy action to handle this scenario.

Listing 7.4 Using IsAjaxRequest to modify action behavior

```
public ActionResult PrivacyPolicy()
{
    if(Request.IsAjaxRequest())
    {
        return PartialView();
    }
    return View();
}
Check if invoked through Ajax
```

The PrivacyPolicy action now checks to see whether the action has been requested via Ajax or not by calling the IsAjaxRequest extension method on the controller's Request property 1. If this returns true, then the action has been called by an Ajax request, in which case the view should be rendered as a partial; if the page has not been called by an Ajax request, it returns a normal view.

Now, when you click the link with JavaScript disabled, the page is rendered with the correct layout, as shown in figure 7.3.



Figure 7.3 Rendering the privacy policy with a layout for non-Ajax requests

7.1.4 Using Ajax to submit form data

In section 7.1.2, you saw how you could leverage jQuery to retrieve data from the server when a link is clicked, but we can also go a stage further by sending data to the server by submitting a form asynchronously. To illustrate this, we'll expand our previous example by showing a list of comments on the page that a user can add to. The end result of this page is shown in figure 7.4.

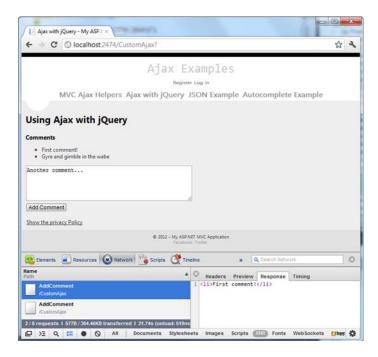


Figure 7.4 The form is posted via Ajax and the result is appended to the list.

To begin, we'll add a collection of comments to our controller in a static field. When the index action is requested, this list of comments will be passed to the view. We'll also add another action (called AddComment) that will allow the user to add a comment to this list. The extended controller is shown here.

Listing 7.5 Introducing the AddComment action

```
public class CustomAjaxController : Controller
    private static List<string> comments
                                                                     Holds list of
        = new List<string>();
                                                                      comments
    public ActionResult Index()
                                                                Sends comments
                                                                to view
        return View( comments);
                                                                    Accepts comment
    [HttpPost]
                                                                    as parameter
    public ActionResult AddComment(string comment)
        comments.Add(comment);
                                                                 Stores new
                                                                 comment
        if (Request.IsAjaxRequest())
          ViewBag.Comment = comment;
                                                                  Sends comment
          return PartialView();
                                                                  to view
        return RedirectToAction("Index");
                                                                Redirects to
    }
                                                                index action
}
```

We begin by creating a list of strings in our controller that will hold some comments ①. These comments are passed to the index view as its model ②. We also add a new action called AddComment that accepts a comment as a parameter ③ and that is decorated with the HttpPost attribute to ensure that this action can only be invoked as the result of a form post.

This action adds the comment to the list of comments 4 and then passes it to a partial view in the ViewBag 5 if the action has been called by an Ajax request. If the user has JavaScript disabled, the action redirects back to the Index action, causing a full-page refresh 6.

NOTE This example is not thread-safe because it stores data inside a static collection. In a real application, this technique should be avoided—a better approach would be to store this data inside a database. However, this example does not use a database for the sake of simplicity.

The partial view returned by the AddComment action simply renders the comment inside a list item:

```
@ViewBag.Comment
```

Next, we can modify our index view to show the current list of comments and add a form to allow the user to submit a new comment. Here's the updated view.

Listing 7.6 Index view with a form for adding comments

```
@model IEnumerable<string>
                                                                     Specify strong
                                                                     type for view
@section head {
    <script type="text/javascript"</pre>
            src="@Url.Content("~/scripts/AjaxDemo.js")">
    </script>
}
<h4>Comments</h4>
ul id="comments">
@foreach (var comment in Model) {
                                                                   Generate list
    @comment
                                                                   of comments
<form method="post" id="commentForm"</pre>
                                                                     Define form to
      action="@Url.Action("AddComment")">
                                                                      add comment
    @Html.TextArea("Comment", new { rows = 5, cols = 50 })
    <input type="submit" value="Add Comment" />
</form>
```

Our modified version of the index view begins by specifying that it is strongly typed to an IEnumerable<string>, which corresponds to the list of comments that is passed to the view from the controller. Following this, it still references our jQuery and Ajax-Demo script files.

We also now include an unordered list of comments ②, which is constructed by looping over the list of comments and writing them out as list items.

Finally, we include a form 3 that posts to our AddComment action and contains a text area where the user can add a comment.

At this point, if you run the page and submit the form, the comment will be added to the list, but it will force a full-page refresh to show the updated comments. The final step is to modify the jQuery code in the AjaxDemo.js file to submit the form via Ajax, as shown here.

Listing 7.7 Submitting the form via Ajax

```
Attach event
$(document).ready(function () {
                                                                          handler
     $('#commentForm').submit(function (event) {
        event.preventDefault();
        var data = $(this).serialize();
                                                                    Serialize form
        var url = $(this).attr('action');
                                                                    to string
        $.post(url, data, function (response) {
                                                                           Send data
             $('#comments').append(response);
                                                                           to server
        });
                                           Append result to
    });
                                              comment list
});
```

Like the example with the link, we begin by declaring a function that will be invoked when the DOM is loaded. Inside this, we tell jQuery to find the form that has an ID of commentForm and attach an event handler to it for when the form is submitted ①, and again we call event.preventDefault to ensure that the form is not submitted. Instead, we serialize the form's contents into a string by calling jQuery's serialize method on the form element ②. This string simply contains a URL-encoded key-value pair representing the fields inside the form. In this case, if we entered the text hello world into the comment box, the serialized form data would contain the value "Comment=hello+world".

Now that we have the contents of the form as a string, it can be posted via Ajax. First, we look at the form action to see where we should submit the data, and we store it in a variable called url 3. Next, we can use jQuery's post method to send this data back to the server. The post function takes several arguments: the URL to where the data should be posted, the data that should be sent, and a callback function that will be invoked once the server has sent back a response.

In this case, the server will be sending back our AddComment partial view, which contains the comment wrapped in a list item. We append it to the end of the comments list by using ¡Query's append method 4.

Now when you visit the page and add a comment, you can see the Ajax request being sent in Firebug and the result being added to the list, as illustrated in figure 7.4.

JavaScript and the "this" keyword

Due to JavaScript's use of functions as objects, it isn't always obvious what the this keyword points to, because it is context sensitive.

In listing 7.7, because this is referenced from within an event handler, it points to the element on which the event was raised (in this case, the form).

7.2 ASP.NET MVC Ajax helpers

So far in this chapter, we've looked at how you can write client-side JavaScript code to send and retrieve data from the server. However, there is another approach that you can use to perform Ajax calls when using ASP.NET MVC, and this is by using Ajax helpers. To begin, we'll look at the Ajax helper methods available in ASP.NET MVC and at how they relate to jQuery and other JavaScript libraries. Following this, we'll look at how these helpers can be used to achieve the same results we've seen so far by manually writing jQuery code.

These helpers are available as extension methods on the AjaxHelper class and can be used to generate markup that will automatically use Ajax to send and retrieve data. They are listed in table 7.1.

Although the last two methods aren't actually Ajax-related, they can be useful when working with JavaScript in an MVC application.

Table 7.1 Ajax helper methods

Helper method	Description
Ajax.ActionLink	Creates a hyperlink to a controller action that fires an Ajax request when clicked
Ajax.RouteLink	Similar to Ajax.ActionLink, but generates a link to a particular route instead of a named controller action
Ajax.BeginForm	Creates a form element that submits its data to a particular controller action using Ajax
Ajax.BeginRouteForm	Similar to Ajax.BeginForm, but creates a form that submits its data to a particular route instead of a named controller action
Ajax.GlobalizationScript	Creates an HTML script element that references a script that contains culture information
Ajax.JavaScriptStringEncode	Encodes a string to make sure that it can safely be used inside JavaScript

Under the covers, these Ajax helpers make use of a JavaScript library to perform the actual Ajax request. This markup is not directly tied to any particular library, but rather makes use of an adapter layer that knows how to use a JavaScript library to issue the Ajax request. Out of the box, ASP.NET MVC has adapters for both jQuery and Microsoft Ajax. Which one is used depends on how the application is configured.

When you create a new ASP.NET MVC project, the following lines are included in the web.config file:

With this setting enabled, the markup generated by the Ajax helpers uses unobtrusive JavaScript in a similar manner to the jQuery examples in section 7.1. However, when this setting is disabled, the helpers will instead generate markup that uses the Microsoft Ajax library. It's best to leave this set to true, but we'll explore what happens if you set it to false in section 7.2.4.

NOTE As an alternative to setting UnobtrusiveJavaScriptEnabled to true in the web.config, you can also set the static property HtmlHelper.UnobtrusiveJavaScriptEnabled in the Application Start method of your Global.asax.

Depending on whether UnobtrusiveJavaScriptEnabled is set to true or false, ASP.NET MVC's Ajax helpers will generate markup that is compatible with a particular adapter layer. This adapter layer knows how to take this markup and invoke the appropriate JavaScript library to perform the actual work. This relationship between the Ajax helpers and the underlying JavaScript libraries is shown in figure 7.5.

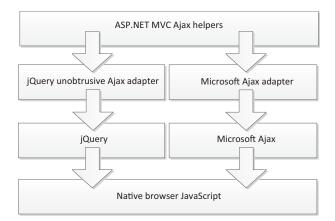


Figure 7.5 The relationship between ASP.NET MVC Ajax helpers and JavaScript libraries

7.2.1 Ajax.ActionLink

We'll begin by taking the same example of the Privacy Policy page from section 7.1.2 and look at how we can use the ActionLink Ajax helper to achieve the same result. Our controller does not need to be modified, but our Index view will need the changes shown in the following listing.

Listing 7.8 Using Ajax. ActionLink @section head { Render in head section <script type="text/javascript"</pre> src="@Url.Content(Reference jQuery.unobtrusive "~/scripts/jquery.unobtrusive-ajax.js")"> script </script> **Define hyperlink** @Ajax.ActionLink(text Action to "Show the privacy Policy", link to "PrivacyPolicy", new AjaxOptions { **Additional** InsertionMode = InsertionMode.Replace, options UpdateTargetId = "privacy" }) <div id="privacy"></div>

As with our previous examples, we begin by rendering part of the head section ①. But this time we also add a reference to the jQuery.unobtrusive-ajax.js file ②, which also comes as part of the default ASP.NET MVC project template. This is the adapter that knows how to use jQuery to perform Ajax calls based on the elements that we render.

Next we include a call to Ajax.ActionLink. There are several overloads for this method, but the one we're using has three arguments. The first is the text that should become a hyperlink 3. The second is the name of the action that should be invoked asynchronously 4—in this case, our PrivacyPolicy action. The final argument is an AjaxOptions object that can be used to customize the Ajax request 5.

The UpdateTargetId property on this object indicates that an HTML element with an id of privacy should be updated to contain the result of invoking our PrivacyPolicy action, and the InsertionMode property indicates that whatever is currently contained inside this element should be replaced.

When you run the application, the result is exactly the same as the previous example—the privacy policy is inserted into the page underneath the action link. But the rendered markup looks slightly different:

```
<a data-ajax="true" data-ajax-mode="replace"
  data-ajax-update="#privacy"
  href="/AjaxHelpers/PrivacyPolicy">Show the privacy Policy</a>
```

In our previous example, we used jQuery to locate the link on the page with a particular ID and then attached an event handler to it. The links generated by Ajax. ActionLink take a slightly different approach.

These links are annotated with several additional attributes. It is the presence of these attributes that indicates that this link should be submitted via Ajax. So instead of explicitly creating an Ajax request in a custom JavaScript file, the link contains all of the metadata that the jquery-unobtrusive.ajax script needs to know in order to construct the appropriate Ajax request.

The data-ajax attribute is used to indicate that the hyperlink should perform its work asynchronously, while the data-ajax-mode and data-ajax-update attributes correspond to the AjaxOptions object specified in listing 7.8.

When the page loads, the script inside the jquery-unobtrusive.ajax script will find all links with the data-ajax attribute and attach a click event, much in the same way we did manually back in listing 7.7. Likewise, if the browser does not have JavaScript enabled, the link will continue to function as a regular hyperlink and fall back to its non-Ajax behavior.

HTML5 Data Attributes

The data-* attributes such as data-ajax and data-ajax-update are known as HTML5 Data Attributes. They provide a way to annotate an HTML element with additional metadata. Although they're being used here to provide information about the Ajax request, you can write your own attributes to provide any metadata that you need to access on the client.

Although these custom attributes are considered to be part of the HTML5 specification, they will also work without any problems on older browsers that don't support HTML5 (including Internet Explorer 6).

7.2.2 Ajax.BeginForm

You can also use ASP.NET's Ajax.BeginForm helper to submit a form asynchronously much in the same way. Let's modify the form definition that we previously created for adding comments to use this helper.

Listing 7.9 Ajax form declaration

Like the Html.BeginForm method you saw in chapter 2, the Ajax.BeginForm method is wrapped with a using statement to delineate the scope of the form ①. The call to BeginForm causes the start of the form to render, and the using statement's closing bracket renders the end of the <form /> tag.

The overload for BeginForm being used takes two parameters—the first is the name of the controller action we want to post to (in this case, AddComment) and the second is an AjaxOptions object. Like the Ajax.ActionLink method, these options are used to specify how the result of the Ajax request should be processed. In this case, after the request has been completed, the result should be inserted into the end of the comments list.

Like the form defined in listing 7.6, this form contains a text area and a submit button 2.

When this example is run, it functions in exactly the same way, although the form is also decorated with the additional data-ajax attributes, like the ActionLink. Here's the resulting markup.

Listing 7.10 Resulting markup of Ajax.BeginForm

```
<form action="/AjaxHelpers/AddComment"
    data-ajax="true" data-ajax-method="POST"
    data-ajax-mode="after" data-ajax-update="#comments"
    id="form0" method="post">
    <textarea cols="50" id="Comment" name="Comment" rows="5">
    </textarea>
    <br />
        <input type="submit" value="Add Comment" />
</form>
```

Again, this form also uses progressive enhancement. By having the jquery.unobtrusive-ajax script included in the page, this form will be submitted via Ajax, but if JavaScript is disabled in the user's browser, the form will perform a regular post.

7.2.3 Ajax options

In the previous section, you saw how both the ActionLink and BeginForm Ajax helpers can take an AjaxOptions object that can be used to indicate how the result of an

Ajax request can be processed. The AjaxOptions class has several options available as properties; they're listed in table 7.2.

Table 7.2 Properties of the AjaxOptions class

Option	Description
HttpMethod	Specifies the HTTP method, which can be GET or POST. If not specified, this defaults to POST for forms and GET for links.
UpdateTargetId	Specifies the element into which the resulting markup should be inserted.
InsertionMode	Sets the insertion mode, which can be InsertBefore (insert the content before the target element's existing children), InsertAfter (insert the content after the element's existing children), or Replace (replaces the element's inner content completely).
OnBegin	Specifies a JavaScript function to be called before invoking the action.
OnComplete	Specifies a JavaScript function to be called after the response comes back.
OnFailure	Specifies a JavaScript function to be called in the event of an error.
OnSuccess	Specifies a JavaScript function to be called if no errors occur.
Confirm	Sets the confirmation message to be displayed in an OK/Cancel dialog box before proceeding.
Url	Specifies the URL to use if the anchor tag has a different destination than the Ajax request.
LoadingElementId	Specifies an element that displays Ajax progress. The element should be marked as display: none initially.
LoadingElementDuration	Specifies how long the animation to show/hide the Loading- ElementId should last if the LoadingElementId has been specified.

With the exception of LoadingElementDuration, all of these options were previously available in ASP.NET MVC 2. But the way in which they are injected into the page's markup now is very different. As you've already seen, these options are generated as data-* attributes in the HTML elements, whereas in MVC 2 they were inserted into the page in a far more obtrusive manner.

7.2.4 Differences from earlier versions of ASP.NET MVC

Although the Ajax helpers have been part of ASP.NET MVC since the first version, jQuery is now the default. In previous versions of the framework, these helpers always used the Microsoft Ajax library and did not generate the JavaScript in an unobtrusive way. You can revert to this previous behavior by setting UnobtrusiveJavaScriptEnabled to false in the AppSettings section of the web.config:

Now, if we were to call Ajax. ActionLink the same way we did in listing 7.8, the following markup would be generated instead:

```
<a href="/AjaxHelpers/PrivacyPolicy"
  onclick="Sys.Mvc.AsyncHyperlink.handleClick(
    this, new Sys.UI.DomEvent(event), {
      insertionMode: Sys.Mvc.InsertionMode.replace,
      updateTargetId: &#39;privacy&#39;
  });">Show the privacy Policy</a>
```

Instead of using the data-ajax attributes, all of the metadata is placed inside an onclick event. It also requires you to reference the MicrosoftAjax.js and MicrosoftMvcAjax.js scripts in order for this to work correctly. This is not as intuitive as before, and it also breaks the unobtrusive JavaScript principle by including a method call directly inside the element's onclick attribute.

If you're upgrading a site from early versions of ASP.NET MVC, you may need to preserve this behavior in order to maintain backwards compatibility, but in all other situations it's best to leave UnobtrusiveJavaScriptEnabled set to true because it results in cleaner markup and is the approach that Microsoft will be investing in going forward.

7.3 Ajax with JSON and client templates

The previous examples in this chapter have all returned HTML markup fragments from the controller action in response to an Ajax request. Our link example returned a markup snippet containing a privacy policy, and the form submission returned a comment wrapped in an /> element.

Although there is nothing wrong with this approach, you aren't limited to simply returning HTML from actions called via Ajax. You could return any of a variety of formats including plain text, XML, and JSON.

This next section will show how JSON can be used alongside Ajax to provide enhanced client-side functionality. The following examples take place in the context of an application that displays information about speakers at a fictitious conference.

7.3.1 Ajax with JSON

JSON (pronounced "Jason") stands for JavaScript Object Notation and provides a very succinct way to represent data. It is widely used in Ajax-heavy applications because JSON strings require very little parsing in JavaScript—you can simply pass a JSON string to JavaScript's eval function, and it will deserialize it to an object graph.

If you're already familiar with JavaScript object literals, the structure of a JSON string will look immediately familiar. Listing 7.11 shows an XML representation of a speaker at our fictitious conference, while listing 7.12 shows the same data represented in JSON.

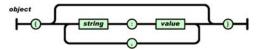
Listing 7.11 An XML representation of a speaker

```
<Speaker>
 <Id>5</Id>
 <FirstName>Jeremy</FirstName>
 <LastName>Skinner</LastName>
 <PictureUrl>/content/jeremy.jpg</PictureUrl>
 <Bio>Jeremy Skinner is a C#/ASP.NET software developer in the UK.</Bio>
</Speaker>
```

Listing 7.12 JSON representation of a speaker

```
"Id":5,
"FirstName": "Jeremy",
"LastName": "Skinner",
"PictureUrl": "/content/jeremy.jpg",
"Bio": "Jeremy Skinner is a C#/ASP.NET software developer in the UK."
```

The JSON format is easy to understand, once you grasp the basic rules. At the core, an object is represented as in figure 7.6.



You can also see that the ISON representation is much less verbose than XML due to the lack of angle brackets, which can drastically reduce download sizes, especially for large documents.

Figure 7.6 The JSON object diagram shows a simple way of understanding the format. (Used with permission from http://json.org.)

To show JSON in action, we'll add a SpeakersController to the application. The Index action will display a list of speakers at the fictitious conference and allow the user to click on them. When a speaker is clicked on, we'll fire an Ajax request to the Details action, which will return the speaker's details in JSON format. The end result will simply display the speaker's name in a dialog box as shown in figure 7.7.

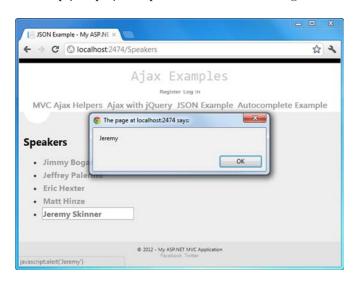


Figure 7.7 Displaying the speaker's first name as the result of an Ajax request.

Here's the basic implementation.

Listing 7.13 The SpeakersController

```
public class SpeakersController : Controller
    private SpeakerRepository _repository
                                                        Instantiate
      = new SpeakerRepository();
                                                        repository
    public ActionResult Index()
                                                                  Retrieve list
                                                                  of speakers
        var speakers = repository.FindAll();
        return View(speakers);
                                                              Pass speakers
                                                              to view
    public ActionResult Details(int id)
        var speaker = _repository.FindSpeaker(id);
        return Json(speaker,
                                                                  Serialize speaker
          JsonRequestBehavior.AllowGet);
                                                                   to ISON
}
```

The controller contains a reference to a SpeakerRepository object, which can be used to retrieve the Speaker objects that represent the speakers at the conference.

NOTE If you're following along with the sample code for this chapter, you'll see that this repository is implemented entirely in memory, although a real application would most likely store this data in a database.

The controller's Index action uses the SpeakerRepository to retrieve a list of all the speakers 1 and pass them to the view 2.

The Details action accepts the ID of a particular speaker and retrieves the corresponding speaker object from the repository. It then serializes this object into JSON format by calling the controller's Json method, which returns a JsonResult ③. JsonResult is an ActionResult implementation that when executed simply serializes an object to JSON and then writes it to the result stream.

ASP.NET MVC, JSON, and GET requests

You'll notice in listing 7.13 that we have to pass an enum value of <code>JsonRequest-Behavior.AllowGet</code> to the controller's JSON method. By default, ASP.NET MVC's <code>JsonResult</code> will only work in response to an HTTP <code>POST</code>. If we want to return JSON in response to a <code>GET</code> request, we have to explicitly opt in to this behavior.

This behavior is in place to prevent JSON hijacking, which is a form of cross-site scripting.

If a site were to return sensitive data in JSON format in response to a GET request, then a malicious site could potentially trick an unwitting user into revealing this data by embedding a script reference to the susceptible site in the page.

(continued)

If an authenticated user were to visit this malicious site, then the data would be downloaded and the malicious site could get access to it. We'll explore JSON hijacking in the next chapter.

In our particular example, we aren't returning sensitive data, so it is perfectly safe to enable JSON responses to GET requests.

Next, we'll implement the Index view.

Listing 7.14 The speaker list page

```
@model IEnumerable<AjaxExamples.Models.Speaker>
                                                                          Strongly
<link rel="Stylesheet" type="text/css"</pre>
                                                                          typed
        href="@Url.Content("~/content/speakers.css")" />
<script type="text/javascript"</pre>
                                                                     Custom script
  src="@Url.Content("~/scripts/Speakers.js")"></script>
                                                                     reference
                                                 CSS reference
<h2>Speakers</h2>
@foreach (var speaker in Model) {
  <
                                                                      Generate list
      @Html.ActionLink(speaker.FullName, "Details",
                                                                      of speakers
        new { id = speaker.Id })
  }
<imq id="indicator"</pre>
                                                                 Display progress
  src="@Url.Content("~/content/load.gif")"
                                                                 spinner
  alt="loading..." style="display:none" />
<div class="selected-speaker"
                                                                Results
                                                                container
     style="display:none"></div>
```

We begin by ensuring that our view is strongly typed to an IEnumerable<Speaker> ①, which corresponds to the list of speakers being passed to the view from the controller. Next, we include a reference to a CSS stylesheet ②, followed by a reference to a script file that will contain our client-side code ③.

We then loop over all of the speakers, creating an unordered list containing their names within a hyperlink 4.

Following this, we add an image to the page that will be displayed while the Ajax request is in progress (3) (also known as a *spinner*).

Finally, we have a <div /> element that will be used to display the speaker's details after they've been fetched from the server 6. We won't be using this just yet, but we'll make use of it in section 7.3.2.

Now that we have our view implemented, we can implement our client-side code within the Speakers, is file.

Listing 7.15 Client-side behavior for the speakers page

```
$(document).ready(function () {
    $("ul.speakers a").click(function (e) {
        e.preventDefault();
                                                                      Show progress
                                                                      indicator
        $("#indicator").show();
        var url = $(this).attr('href');
                                                                    Retrieve URL
        $.getJSON(url, null, function (speaker) {
                                                                          Invoke Ajax
             $("#indicator").hide();
                                                                          request
             alert(speaker.FirstName);
                                                              Display
        });
                                                              result
    });
});
```

As usual when working with jQuery, we begin by waiting for the DOM to load and then attach a function to the click event of the links within our speaker list. The first thing this does is show our loading indicator 1.

Following this, we extract the URL from the hyperlink that the user clicked, and store it in a variable called url 2. This variable is then used to make an Ajax request back to the server 3. This time we use jQuery's \$.getJSON function, passing in the URL to call, any additional data that we want to send (in this case we don't have any data, so we pass null), and a callback function that will be invoked once the request is complete. This function will automatically deserialize the JSON string returned from the server and convert it into a JavaScript object. This object is then passed to the callback function.

The callback function accepts as a parameter the object that was describlized from the server's JSON response (in this case, our Speaker object). Inside the callback, we hide the loading indicator and then display the speaker's FirstName property in a message box 4.

Displaying a modal dialog box with the speaker's first name isn't the most useful behavior. Instead, it would be much nicer to inject some markup into the page that shows the speaker's details along with their photo. This is where client-side templates come in.

7.3.2 Client-side templates

Much like we create server-side templates in the form of Razor's .cshtml files, we can also create templates on the client.

Client-side templates allow us to generate markup on the fly in the browser without having to go back to the server or having to manually construct elements using JavaScript. There are several client-side templating libraries available, but we'll be using jQuery-tmpl, a templating library for jQuery that was written by Microsoft and then contributed to the jQuery project as open source.

We'll modify the speaker list page so that when a speaker's name is clicked, their bio and photo will be displayed, as shown in figure 7.8.

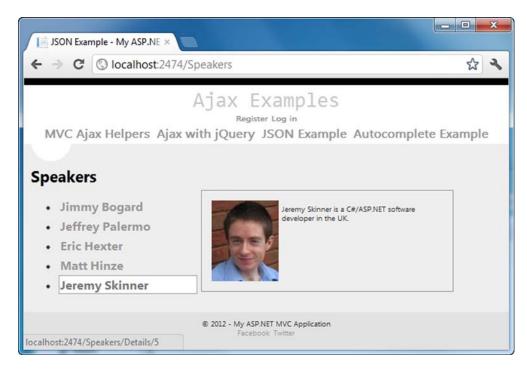


Figure 7.8 Displaying the rendered template next to the speaker list

To reference jQuery-tmpl, we can either download it from the project page at https://github.com/jquery/jquery-tmpl and place it in our application's Scripts directory, or we can reference it directly from Microsoft's CDN at http://ajax.microsoft.com/ajax/jquery.templates/beta1/jquery.tmpl.js. Once referenced, we can also add a template to our view.

Listing 7.16 Using client-side templates <script type="text/javascript"</pre> Reference jQuery src="@Url.Content("~/scripts/jquery.tmpl.js")"> templates </script> <script id="speakerTemplate" type="text/x-jquery-tmpl"> Define <img src="\${PictureUrl}"</pre> template alt="Speaker image" class="speaker-pic" /> section Photo template \${Bio} **Bio line** template <br style="clear:both;" /> </script>

We begin by including a reference to the jQuery-tmpl script from our scripts folder 10 and then declare a template 2. Templates are defined inside script elements within the

page with a type of text/x-jquery-tmpl. Keeping the template's markup within a script element ensures that the template elements are not rendered directly into the page.

Our template includes the speaker's photo 3 as well as the speaker's bio line 4. We can refer to the JSON object's properties by wrapping them within \${} code nuggets, which will be replaced by the actual value when the template is rendered.

Next, we need to modify our JavaScript in Speakers.js to render the template. Here's the updated code.

Listing 7.17 Modifying our script to render the template

```
$(document).ready(function () {
    $("ul.speakers a").click(function (e) {
        e.preventDefault();
                                                                Hide speaker
                                                                details
        $(".selected-speaker").hide().html('');
        $("#indicator").show();
        var url = $(this).attr('href');
        $.getJSON(url, null, function (speaker) {
            $("#indicator").hide();
                                                              Render template
                                                              with data
            $("#speakerTemplate")
                .tmpl(speaker)
                 .appendTo('.selected-speaker');
            $('.selected-speaker').show();
        });
    });
});
```

This code is mostly the same as the code in listing 7.15 but with a couple of differences. First, if we're already showing a speaker's details, then we hide them before making a new request to the server ①. Second, instead of simply displaying a message box within the Ajax request's callback, we now render the template. This is done by first telling jQuery to find the template element and then invoking the tmpl method to render the template ②. This method accepts an object that should be passed to the template, which in this case is a reference to our speaker. The rendered template is then appended to the <div /> element in our page with a CSS class of selected-speaker.

The end result is that when the speaker's name is clicked, the template is rendered next to the list, as shown in figure 7.8. Note that extra styling has been added to make the page look presentable. This extra styling can be found in the chapter's associated sample code.

7.3.3 Finishing touches

Our speaker page is largely complete, but it does have one flaw. If JavaScript is disabled in the browser, then when we click on the speaker's name the corresponding JSON will be downloaded as a text file rather than rendered as a template.

To get around this, we can use a similar technique to listing 7.4 and render a view if the action has not been requested via Ajax.

Listing 7.18 Adding graceful degradation to the Details action

Instead of relying on an if statement within our code, we could use an action method selector to differentiate between Ajax and non-Ajax requests. We first saw how action method selectors could be used in chapter 2, and we can create an AcceptAjaxAttribute by simply inheriting from the ActionMethodSelector attribute as shown here.

Listing 7.19 Implementing the AcceptAjaxAttribute

The AcceptAjaxAttribute simply returns true from the IsValidForRequest method if the current action is being requested via Ajax.

We can now use this attribute from within our SpeakersController by defining two separate actions—one for handling Ajax requests, the other for normal requests.

Listing 7.20 Using the AcceptAjaxAttribute

```
[AcceptAjax]
public ActionResult Details(int id)
{
    var speaker = _repository.FindSpeaker(id);
    return Json(speaker, JsonRequestBehavior.AllowGet);
}
[ActionName("Details")]
public ActionResult Details_NonAjax(int id)
{
    var speaker = _repository.FindSpeaker(id);
    return View(speaker);
}
Accessible only for
Ajax requests

Aliased action
using ActionName
```

The first overload of the Details action is annotated with our AcceptAjaxAttribute ①, which ensures that it is only invoked for Ajax requests. This version of the action returns the JSON-serialized speaker details.

The other overload does not have the AcceptAjaxAttribute, which means that it will be invoked for non-Ajax requests. This action simply passes the Speaker instance to a view. Note that because C# cannot define two methods with the same name and same signature, the second version of the action is named Details_NonAjax, but it can still be accessed at the URL /Speakers/Details because it is annotated with an ActionName attribute 2.

NOTE The AcceptAjaxAttribute can also be found as part of the ASP.NET MVC Futures DLL that can be downloaded from http://aspnet.codeplex.com.

In this particular example, there isn't really much benefit from using the AcceptAjax-Attribute, but in a situation where the Ajax and non-Ajax versions of an action perform significantly different work, splitting them up can help with readability.

We also need to define a view for the non-Ajax version of the action. This view simply displays the speaker's details, much like in the client-side template.

Listing 7.21 Non-Ajax speaker details

When we now click the speaker's name with JavaScript disabled, we'll be taken to a separate page, as shown in figure 7.9.



Figure 7.9 Speaker details displayed without Ajax

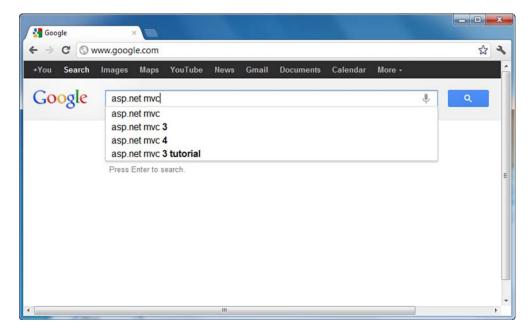


Figure 7.10 Google Suggest filters options as you type

7.4 Creating an autocomplete text box

So far in this chapter, you've seen how you can leverage Ajax and JSON to make requests back to the server to retrieve data. In addition to manually issuing these Ajax requests, you can also make use of client-side control libraries and jQuery plugins that abstract away much of the boilerplate code for dealing with Ajax requests.

jQuery UI (http://jqueryui.com) is one such set of plugins. It's built on top of the jQuery core to provide several client-side user interface widgets, including an accordion, an autocomplete text box, themeable buttons, a datepicker, a modal dialog, a progress bar, slider, and tabs. In this example, we'll look at how we can make use of the Autocomplete plugin to present the user with a searchable list of cities, something like Google's suggest functionality, shown in figure 7.10.

7.4.1 Building the CitiesController

To begin, we'll create a CitiesController that will render a page containing our text box, as follows.

Listing 7.22 The CitiesController

```
public class CitiesController : Controller
{
    private readonly CityRepository _repository;
    public CitiesController()
```

```
{
    __repository = new CityRepository();
}

public ActionResult Index()
    {
      return View();
}
```

The CitiesController instantiates a CityRepository in its constructor ①. This repository exposes a single method, Find, which takes a search term and finds all the cities whose name starts with the specified search term. The internal implementation of the CityRepository isn't important for this example, but if you're following along with the sample code for this chapter, you'll see that it loads the cities data from a CSV file.

The City object itself is defined in the following listing.

Listing 7.23 City class definition

```
public class City
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string State { get; set; }

    public string DisplayName
    {
        get { return Name + ", " + State; }
    }
}
```

The City object is a very simple POCO (Plain Old CLR Object)—it simply defines three read/write properties (a numeric ID, the name of the city, and the state in which it's located) and a read-only property that constructs a user-friendly display name.

The view rendered by the Index action is shown in the following listing.

Listing 7.24 The autocomplete page

```
<script
  src="@Url.Content("~/Scripts/jquery-1.7.1.js")"
  type="text/javascript"></script>
                                                                    Reference jQuery
                                                                    scripts
<script
  src="@Url.Content("~/Scripts/jquery-ui-1.8.16.js")"
  type="text/javascript"></script>
link
                                                                  Reference jQuery
  href="@Url.Content(
                                                                  UI styles
    "~/content/themes/base/jquery-ui.css")"
  rel="Stylesheet" type="text/css" />
                                                                      Document-
<script type="text/javascript">
                                                                      ready handler
    $(function () {
```

```
var autocompleteUrl = '@Url.Action("Find")';
                                                                       Build
                                                                       search URL
        $("input#city").autocomplete({
                                                               Add
            source: autocompleteUrl,
                                                                autocomplete
            minLength: 2,
                                                                behavior
            select: function (event, ui) {
                alert("Selected " + ui.item.label);
        });
    });
</script>
<h2>Cities</h2>
   Start typing a city to see
    the autocomplete behavior in action.
>
                                                                    Container
    <label for="city">City</label>
                                                                    for results
    <input type="text" id="city" />
```

As with our previous examples, we need jQuery. 1 If you have not customized your layout, these script references are included.

Next we add a reference to the jQuery UI stylesheet 2, which also ships with the default project template. Again, if you have not customized your layout, you will already have this.

Following this, we include a script block that runs when the page loads ③. We begin by defining a variable called autoCompleteUrl, which contains the URL of the Find action of the CitiesController (which we haven't created yet) ④. This is the URL that will be requested each time the user types a character in the box in order to perform the search. We then find any text boxes on the page with the ID of city and invoke the Autocomplete plugin on this element ⑤. We tell it where it should look for data (in this case, our autoCompleteUrl), the minimum number of characters that have to be entered before searching (in this case, 2), and a callback function that should be invoked when the user has selected a search result. For simplicity, we'll just pop up an alert with the name of the selected city. Finally, we define the text box that will allow the user to perform the search ⑥.

Running the page at this point will display a text box. However, as we haven't yet implemented the Find action, it currently produces an error, as shown in figure 7.11.

When a search term is entered in the box, the Autocomplete plugin makes an Ajax request back to the server. In this case, it is to our Find action and it passes the search term as a query string parameter called term. The Autocomplete plugin expects this URL to return an array of JSON objects with the following properties: an id, a label (which will be displayed in the search results), and a value (which will be inserted into the text box when clicked).

At the moment, this is causing a 404 error because we haven't yet implemented the Find action. We can now go ahead and do this.

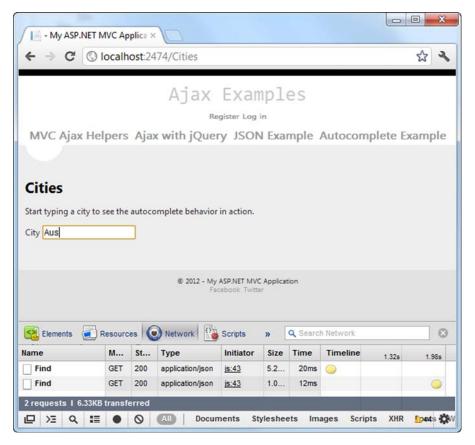


Figure 7.11 The autocomplete text box makes an Ajax request when the user types a search term.

Listing 7.25 Implementation of the Find action public ActionResult Find(string term) Search for city City[] cities = repository.FindCities(term); var projection = from city in cities select new **Create projection** of results id = city.Id, label = city.DisplayName, value = city.DisplayName return Json(projection.ToList(), Serialize JsonRequestBehavior.AllowGet); result to JSON

Here we begin by finding all of the cities whose names start with the specified search term ①. We then use an in-memory LINQ query to project the resulting City objects

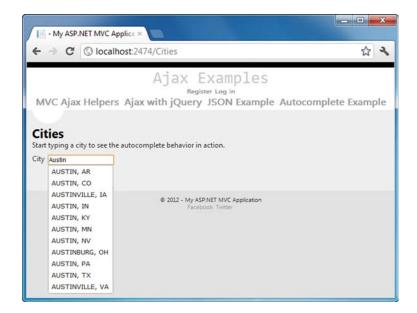


Figure 7.12
Displaying the search results

into a collection of anonymous types that match the JSON structure that the Autocomplete plugin expects (an id property, a label property, and a value property) 2. Finally, we serialize these results to JSON by calling the Json method 3. As with our example in listing 7.13, we have to explicitly allow JSON in response to a GET request by using the AllowGet behavior.

Finally, when you rerun the page and enter a search term, you'll see the results come back from the server as shown in figure 7.12.

You can also see the JSON being returned from the server by inspecting the Ajax requests using Firebug, as shown in figure 7.13.

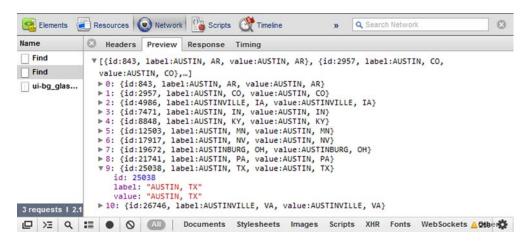


Figure 7.13 The JSON returned from the server in response to the search

The resulting page now allows us to search for a city by entering the start of a city's name; the server will perform a search and produce the appropriate JSON. The Autocomplete plugin will handle the result and automatically generate the drop-down without us needing to write any code to parse the results. Finally, if we select an item in the drop-down, the value property of the underlying JSON is inserted into the text field.

7.5 **Summary**

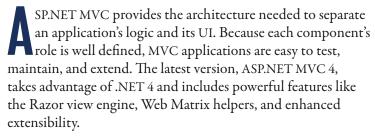
Ajax is an important technique to use with today's web applications. Using it effectively means that the majority of your users will see a quicker interaction with the web server, but it doesn't prevent users with JavaScript disabled from accessing the site. This is sometimes referred to as progressive enhancement. Unfortunately, with raw JavaScript, the technique is cumbersome and error-prone. With JavaScript libraries such as jQuery, you can be much more productive.

In this chapter, you've seen how to apply Ajax in different ways: using partial HTML replacement and JSON. You've learned how to intercept a form submission and provide a more seamless Ajax experience for those users who support Ajax, while continuing to provide functionality for those who don't. You've also seen how client-side templates can be used to delegate the rendering of mark-up to the client, rather than performing all rendering on the server.

We also briefly mentioned how ASP.NET MVC has some built-in security features, such as how you can't return JSON data in response to a GET request by default. In the next chapter, we'll explore this in more detail, along with other security issues.

ASP.NET MVC 4 IN ACTION

Palermo • Bogard • Hexter • Hinze • Skinner



ASP.NET MVC 4 in Action is a hands-on guide that shows you how to apply ASP.NET MVC effectively. After a high-speed ramp up, this thoroughly revised new edition explores each key topic with a self-contained example so you can jump right to the parts you need. Based on thousands of hours of real-world experience, the authors show you valuable high-end techniques you won't find anywhere else. Written for developers, the book arms you with the next-level skills and practical guidance to create compelling web applications.

What's Inside

- Complete coverage of ASP.NET MVC 4
- The new Web API
- Full-system testing

You need some knowledge of ASP.NET and C#, but no prior ASP.NET MVC experience is assumed.

Jeffrey Palermo, Jimmy Bogard, Eric Hexter, Matthew Hinze, and Jeremy Skinner are all ASP.NET MVPs, ASP Insiders, and early adopters of ASP.NET MVC.

To download their free eBook in PDF, ePub and Kindle formats, owners of this book should visit manning.com/ASP.NETMVC4inAction



- "Guides you through the inner workings of ASP.NET MVC."
- —From the Foreword by Phil Haack, GitHub
- "A brilliant book for a great framework."
- —Jonas Bandi, TechTalk
- "A complete guide, with established open source tools."
- —Apostolos Mavroudakis, UBS AG
 - "A great addition to a great series of books."
 - —Paul Stack, Toptable.com
 - "Practical web application construction for the pragmatic practitioner."
 - —Arun Noronha Guardian Protection Services



