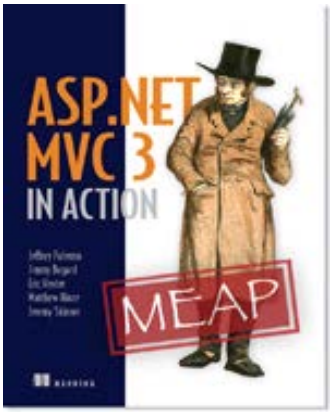
MANNING PUBLICATIONS



ASP.NET MVC 3 in Action
Jeffrey Palermo, Jimmy Bogard, Eric Hexter, Matthew Hinze, and Jeremy Skinner

jQuery has quickly become one of the most popular JavaScript libraries due to its simple yet powerful mechanisms for interacting with the HTML DOM. In this article, based on chapter 12 of ASP.NET MVC 3 in Action, the authors discuss the basics of using jQuery and how it can be used to make asynchronous calls to the server that can be processed by ASP.NET MVC.

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 DOM[1] can be 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 to normalize cross-browser JavaScript functionality. For related examples, we'll be using the open-source jQuery library (http://jquery.com) that was initially released by John Resig in 2006.

jQuery has quickly 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 have contributed several features to its codebase and provide official support for it as well as shipping as part of ASP.NET MVC's default project template.

In this article, we'll first look at the basics of using jQuery and 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 enabled scripting 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.

jQuery Primer

When working with jQuery, you mainly work with the `jQuery` object (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, we 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

[1] DOM stands for "Document Object Model" and is a hierarchy of objects that represents all of the elements in a page.

`$('#foo')` would find the element whose ID is `foo`, while a call to `$('table.grid td')` would find all of the `<td />` 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. In this case, we call the `addClass` method, which adds the specified CSS class to each element contained in the wrapped set (in this case, all of the `<div />` elements in the page).

You can also attach events to elements in a similar fashion. If we wanted to show a message box when a button is 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, we 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 time, we introduce a script element within the page to contain our 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) it aids in maintainability and makes it easier to follow the flow of the code.

In the same way we can attach events to elements, we can also attach a `ready` event to the entire page. This event will be fired once the page's DOM hierarchy has been loaded and is the earliest possible point where it is safe to interact with HTML elements. As such, it is better that all event bindings and other jQuery code are contained within the `ready` handler:

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

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

These core concepts should give you enough to be able to understand the following examples. For a more in-depth look at jQuery, you may wish to read the book *jQuery in Action, Second Edition*.

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 3 project using the default Internet Application template and adding a simple controller. This controller has two actions. Both will 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 1.

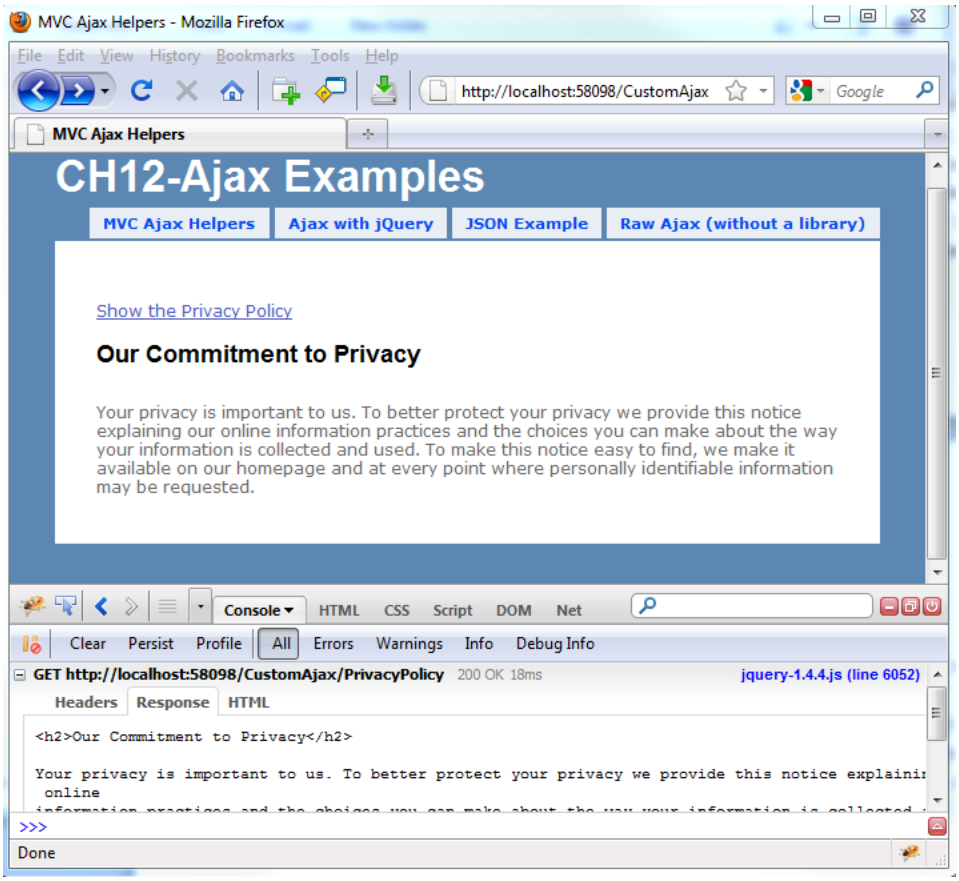


Figure 1 The privacy policy will be loaded when the link is clicked.

The code for this controller is shown in listing 1.

Listing 1 A simple controller

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


    public ActionResult PrivacyPolicy()
    {
        return PartialView();                    #1
    }
}
```

#1 Renders a partial view

Note that we return a partial view from our `PrivacyPolicy` action (#1) rather than a view to ensure that the site's layout isn't applied to the view. This is done to ensure that the surrounding chrome (such as the menu) that is 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 shown in listing 2.

Listing 2 The index view including script references

```
<script type="text/javascript"                                    |#1
```

```
    src="@Url.Content("~/scripts/jquery-1.5.2.js")"></script>          |#1

<script type="text/javascript"                                         |#2
    src="@Url.Content("~/scripts/AjaxDemo.js")"></script>          |#2


@Html.ActionLink("Show the privacy policy",                            |#3
  "PrivacyPolicy", null, new { id = "privacyLink" })                   |#3

<div id="privacy"></div>                                                #4
```

#1 Reference jQuery script
#2 Reference our demo code
#3 Link to action
#4 Container for results

We begin by including a reference to the jQuery script (#1). Newly created MVC 3 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 1.5.2 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 privacyPolicy link, as shown in listing 3.

Listing 3 Custom jQuery code in the AjaxDemo.js file

```
$(document).ready(function () {                                        #1
    $('#privacyLink').click(function (event) {                         #2
        event.preventDefault();                                        #3

        var url = $(this).attr('href');                                #4
        $('#privacy').load(url);                                       #5
    });
});
```

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 (#4) and store it in a variable called `url`.

The final line of the event handler issues the actual Ajax request (#5). 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 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 we run the application and click on the link, we should see the privacy policy inserted in to 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 1.

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

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 of this (our Ajax functionality). This way, if the user does not have JavaScript enabled in their browser, the link will gracefully degrade to its original behavior and instead send the user to the privacy policy page without using Ajax, as shown in figure 2.

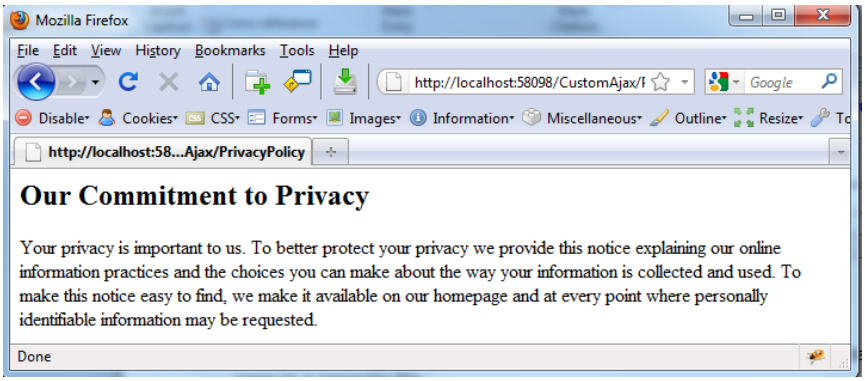


Figure 2 The browser goes directly to the Privacy Policy page if JavaScript is disabled

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, as shown in listing 4.

Listing 4 Using IsAjaxRequest to modify action behavior

```
public ActionResult PrivacyPolicy()
{
    if(Request.IsAjaxRequest())                                    #1
    {
        return PartialView();
    }

    return View();
}
```

#1 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 but, if the page has not been called by an Ajax request, it returns a normal view.

Now, when we click the link with JavaScript disabled, the page is rendered with the correct layout, as shown in figure 3.

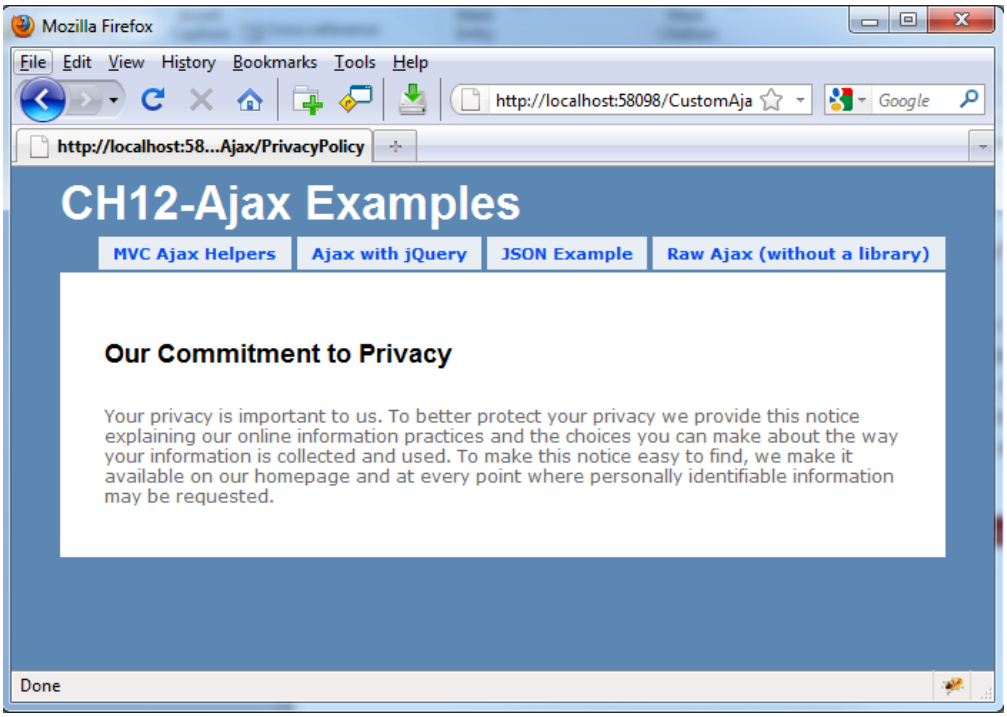


Figure 3 Rendering the privacy policy with a layout for nonAjax requests

Using Ajax to submit form data

We saw how we 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 to which a user can make additions. The end result of this page is shown in figure 4.

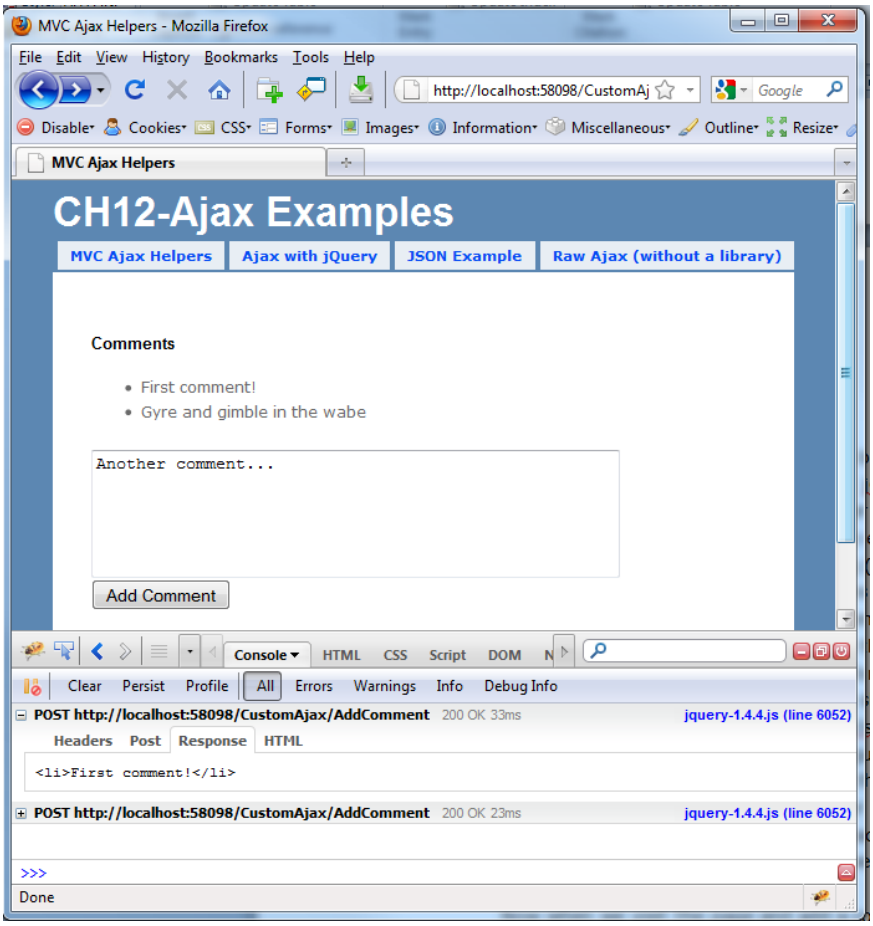


Figure 4 The form is posted via Ajax and the result is appended to the list.

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 in listing 5.

Listing 5 Introducing the AddComment action

```
public class CustomAjaxController : Controller
{
    private static List<string> _comments        |#1
        = new List<string>();                    |#1

    public ActionResult Index()
    {
        return View(_comments);                   #2
    }

    [HttpPost]
    public ActionResult AddComment(string comment)  #3
    {
        _comments.Add(comment);                     #4

        if (Request.IsAjaxRequest())
        {
          ViewBag.Comment = comment;                #5
          return PartialView();
        }
        return RedirectToAction("Index");           #6
    }
```

```
}
```

#1 Holds list of comments
#2 Sends to comments to view
#3 Accepts comment as parameter
#4 Stores new comment
#5 Sends comment to view
#6 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 our index view as its model (#2). We also add a new action called `AddComment` which accepts a comment as a parameter (#3) and is also decorated with the `HttpPost` attribute to ensure that this action can only be invoked as the result of a form post.

Inside this action, it 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, then the action redirects back to the `Index` action, causing a full-page refresh (#6).

NOTE

This example is not thread safe as 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:

```
<li>@ViewBag.Comment</li>
```

Next, we can modify our index view to show the current list of comments as well as add a form to allow the user to submit a new comment. The updated view is shown in listing 6.

Listing 6 Index view with a form for adding comments

```
@model IEnumerable<string>                                  #1

<script type="text/javascript"
  src="@Url.Content("~/scripts/jquery-1.5.2.js")"></script>
<script type="text/javascript"
  src="@Url.Content("~/scripts/AjaxDemo.js")"></script>

<h4>Comments</h4>

<ul id="comments">                                          |#2
@foreach (var comment in Model) {                           |#2
    <li>@comment</li>                                       |#2
}                                                           |#2
</ul>                                                       |#2

<form method="post" id="commentForm"                        |#3
      action="@Url.Action("AddComment")">                   |#3

    @Html.TextArea("Comment", new { rows = 5, cols = 50 })
    <br />
    <input type="submit" value="Add Comment" />
</form>
```

#1 Specifies strong type for view
#2 Generates list of comments
#3 Defines form to add comment

Our modified version of the index view begins by specifying that it is strongly-typed (#1) 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 AjaxDemo 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 we run the page and submit the form, then 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 our AjaxDemo.js file to submit the form via Ajax, as shown in listing 7.

Listing 7 Submitting the form via Ajax

```
$(document).ready(function () {
     $('#commentForm').submit(function (event) {                #1
        event.preventDefault();
        var data = $(this).serialize();                         #2
        var url = $(this).attr('action');

        $.post(url, data, function (response) {                 #3
            $('#comments').append(response);                    #4
        });

    });
});
```

#1 Attaches event handler
#2 Serializes form to string
#3 Sends data to server
#4 Appends 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 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 (#2) by calling jQuery's `serialize` method on the form element. 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 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 first is the URL to where the data should be posted, the second is the data that should be sent, and the third is 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, and we append it to the end of the comments list using jQuery's `append` method (#4).

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

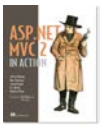
In listing 7, because `this` is referenced from within an event handler, it points to the element on which the event was raised (in this case, the form).

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.

Here are some other Manning titles you might be interested in:

ASP.NET 4 in Practice
Daniele Bochicchio, Stefano Mostarda, and Marco De Sanctis



ASP.NET MVC 2 in Action
Jeffrey Palermo, Ben Scheirman, Jimmy Bogard, Eric Hexter, and Matthew Hinze

jQuery in Action, Second Edition
Bear Bibeault and Yehuda Katz

Last updated: September 5, 2011