

# ASP.NET MVC in Action

Jeffrey Palermo  
Ben Scheirman  
Jimmy Bogard

MEAP

 MANNING



Unedited Draft



**MEAP Edition  
Manning Early Access Program**

Copyright 2007 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

# *Contents*

Preface

Chapter 1: Getting started with the ASP.NET MVC Framework

Chapter 2: The Model in depth

Chapter 3: The Controller in depth

Chapter 4: Views in depth

Chapter 5: Routing

Chapter 6: Customizing and extending the ASP.NET MVC Framework

Chapter 7: Scaling the architecture to more complex sites

Chapter 8: Leveraging existing ASP.NET features

Chapter 9: AJAX in ASP.NET MVC

Chapter 10: Hosting and deployment

Chapter 11: Exploring MonoRail and Ruby on Rails

Chapter 12: Best practices

Chapter 13: Recipes

# 9

## AJAX in ASP.NET MVC

This chapter covers:

- Our view on Ajax
- Difficulties with Web Forms
- Javascript Libraries
- Simple HTML replacement
- JSON & XML responses

Ajax, or Asynchronous Javascript And XML, is a term coined by Jesse James Garret describing a clever new technique to make web applications more dynamic. Ajax introduced a new era of web applications. It is a technique that uses the browser's Javascript capability to send a request to the server asynchronously. This enabled applications to become richer and more user-friendly by updating small sections of the page without requiring a brutal full page refresh. In today's web, the vast majority of major websites have leveraged this technique to their advantage. Users are demanding this type of rich, seamless interaction with websites. You aren't going to let them down, are you?

There is no doubt about it; Ajax is here to stay. With ASP.NET Web Forms, developers have historically had certain frictions when applying Ajax to their sites. Many popular code samples and Ajax libraries seemed to fit well for the PHP and Ruby on Rails examples, but they did not translate as well to the ASP.NET platform. Largely this was due to two things: the Page-centric request lifecycle and the lack of control over HTML DOM identifiers. A WebForms friendly framework called ASP.NET AJAX was released by Microsoft in early 2007 to moderate success. Many developers found it to be overly complicated and cumbersome. ASP.NET AJAX and its associated control toolkit depend deeply on the post back behavior of Web Forms. For ASP.NET MVC applications there are more fitting frameworks available.

In this chapter you will examine how the Ajax technique is applied to ASP.NET MVC in a more seamless way than with WebForms. You will see how to leverage an increasingly popular, lightweight javascript library called jQuery. You will learn a few different types of formatting in common use with Ajax, along with the strengths and weaknesses of each. While a quick introduction on Ajax will be given, you will be best served with at least an introductory knowledge of the subject.

### 9.1 Diving In To Ajax with an Example

A quick example is the best way to describe how Ajax works. We will create a simple HTML page that has a button on it. When the button is clicked, an Ajax request will be sent to the server. The response will be a simple message, which we will display to the user. No browser refresh will occur. Take a look at our HTML page in Listing 9.1

#### Listing 9.1 – A simple HTML page

```
<html>
<head>
```

```

    <title>Ajax Example 1</title>
    <script type="text/javascript" src="ajax-example1.js"></script>
  </head>

  <body>
    <h1>Click the button to see the message...</h1>
    <input type="button" value="Whack! " onclick="get_message();" /> #1

    <div id="result"></div> #2
  </body>
</html>

```

- #1 When this button is clicked, we will issue an ajax request**
- #2 The resulting message from the server will be displayed here**

This is a very basic HTML page with a button on it. When the user clicks the button, the server should get the message without refreshing the page, and display it to the user. Listing 9.2 shows the contents of the referenced Javascript

### Listing 9.2 – Our simple Javascript file

```

function get_message()
{
  var xhr = getXmlHttpRequest(); #1

  xhr.open("GET", "get_message.html", true) #2

  xhr.onreadystatechange = function() { #3
    if(xhr.readyState != 4) return; #4

    document.getElementById('result').innerHTML = xhr.responseText; #5
  };

  xhr.send(null); #6
}

function getXmlHttpRequest()
{
  var xhr;
  if(typeof XMLHttpRequest != 'undefined'){ #7

    try {
      xhr = new XMLHttpRequest("Msxml2.XMLHTTP");
    } catch(e) {
      xhr = new XMLHttpRequest("Microsoft.XMLHTTP");
    }

  } else if(XMLHttpRequest) {
    xhr = new XMLHttpRequest(); #8
  } else {
    alert("Sorry, your browser doesn't support Ajax");
  }

  return xhr;
}

```

- #1 get our xml http request object**
- #2 prepare the request**
- #3 setup the callback function**
- #4 readyState 4 means we're done**
- #5 populate the page with the result**
- #6 fire the ajax request**
- #7 check for IE implementation(s)**
- #8 this works for Firefox, Safari, Chrome, Opera, etc**

The resulting page looks like Figure 9.1.

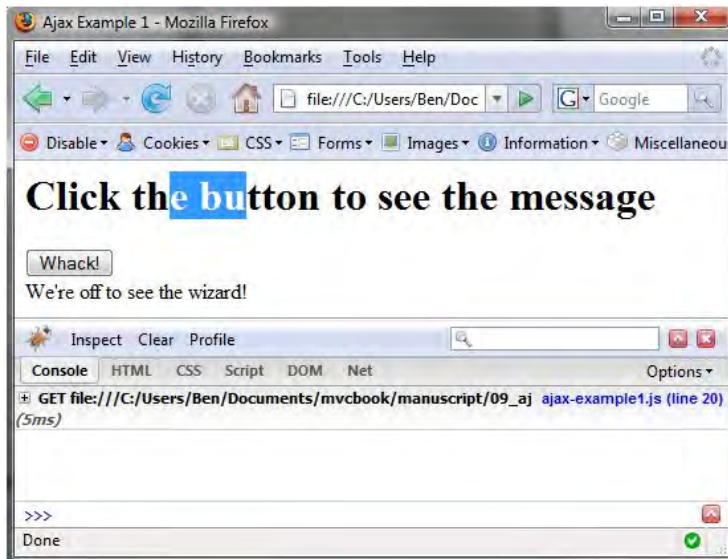


Figure 9.1 - The highlighted text remains, indicating the request was submitted asynchronously. Firebug (shown at the bottom of the browser window) also allows us to inspect Ajax calls for a better debugging experience. Get Firebug at <https://addons.mozilla.org/en-US/firefox/addon/1843>

### Unobtrusive Javascript

You might notice throughout this chapter that I prefer unobtrusive Javascript. This means that the functionality of the page degrades gracefully in the absence of Javascript. I also adhere to common Javascript standards, such as `document.getElementById('myDiv')` rather than the non-standard `document.myDiv` or others.

Have you ever seen code that looks like this?

```
<a href="javascript:window.open('...')">info</a>
```

The href attribute is supposed to point to a document, not contain Javascript code. Other times we see this:

```
<a href="javascript:void(0)" onclick="window.open(...)">info</a>
```

We still have that funky Javascript string where it doesn't belong, and this time we're using the onclick handler of the a tag. This is marginally better, but if you followed unobtrusive scripting, you'd end up with something like this:

```
<a href='info.html' class='popup'>info</a>
```

With Javascript enabled, we can loop over all links with a class of 'popup' and attach an onclick event handler that calls `window.open()` with the link's href property. If Javascript is disabled, the link functions as normal, and the user can still see the info.html page. We get the benefit of graceful degradation in the absence of Javascript and separation of behavior from presentation.

In some cases the examples show what is most easily displayed in book format, however in practice, it is worthwhile to follow the unobtrusive javascript principles.

For more information on unobtrusive Javascript, see Jeremy Keith's excellent book DOM Scripting.

If you're thinking that this is a lot of code in the previous example for a simple Ajax request, then you're not alone. The simple act of creating the XMLHttpRequest object isn't consistent across browsers. We'll see how to clean that up later on. First, let's see how this example would be applied in ASP.NET Web Forms.

## 9.2 Ajax with ASP.NET Web Forms

If we take the example in listing 9.1 and 9.2 and apply it to Web Forms, we may run across some friction. First is the issue of the actual web request. Above we specified the URL to be `get_message.html`, but in reality this is

probably going to be a dynamic page. Let's assume that we used `get_message.aspx` and the message actually comes from a database. ASP.NET pages go through the page lifecycle events and then render the template (.ASPX) that we have defined. These templates represent a full HTML document; however we only wanted to render the message. We could instead utilize a custom `IHttpHandler` to intercept a different file extension and not use the page template. This would look something like listing 9.3.

### Listing 9.3 - A Custom Ajax HttpHandler

```
public class AjaxHandler : IHttpHandler
{
    public bool IsReusable
    {
        get { return true; }
    }

    public void ProcessRequest(HttpContext context)
    {
        if (context.Request.QueryString["operation"] == "get_message")
        {
            context.Response.Write("yuck");
            context.Response.ContentType = "text/plain";
        }

        context.Response.End();
    }
}
```

Quickly we can see that using `Response.Write()` from our code is a cumbersome way to render content for an Ajax request. We'd like to use the templating power of ASPX, without using full HTML documents.

Another road-bump that we might come across is in the callback function. When the request comes back from the server, we get the element with the id of `result` and update its contents with the response text. If our target element is a server control - such as a `TextBox`, `Panel`, or `Label` - ASP.NET will generate the id for us. Thus we are forced to generate this id using some method of `<%= theControl.ClientID %>`, which will give us the correct ID. This means we either need to pass in the id to the Javascript function, or generate the entire function definition inside our ASPX page so that we can execute the above snippet.

#### Ajax Return Values

The 'X' in Ajax stands for 'XML', but that doesn't mean we have to return XML for our Ajax calls. There are a few different options for return values. Some are better for over-the-wire performance, some are easy to create on the server side, and some are easy to consume with Javascript. You should choose the one that fits your needs best.

Simple return values can be passed, such the example in this chapter, or partial HTML snippets can be returned to be added to the DOM. Often you need to work with structured data. XML documents can be returned, and while they are easy to create on the server they are not a very common choice due to their bloated nature. A better solution for representing data is to use JSON (JavaScript Object Notation).

JSON strings are Javascript. They just need to be passed to the `eval()` method to be evaluated as Javascript. JSON happens to be a much more concise representation than XML. For more information on the JSON format, see <http://json.org>. Lastly, when you want to take advantage of templates, you can return HTML fragments and update the HTML directly with the result. This option tends to be the easiest, however over-the-wire performance of presentation HTML is the worst of all the options.

Always choose the most appropriate method of response given your scenario.

With ASP.NET MVC we can do better. We have complete control over our HTML, and as such have responsibility for naming our elements in a way that will not collide with other elements on the page. We also have a better method of having templates for our results, so that we may return an HTML fragment for an Ajax call and not rely on `Response.Write()`.

## 9.3 Ajax in ASP.NET MVC

In ASP.NET MVC our Ajax scenario is *much* cleaner. We have control over the rendered HTML, so we can choose our own element IDs and not rely on ASP.NET server controls to generate them for us. We can also choose to render views that can be plain text, XML, JSON, HTML fragments, or even Javascript that can be run on the client. Let's take a more complicated scenario and look at how it looks in ASP.NET MVC.

Most of the examples in this chapter will utilize an excellent Javascript library called jQuery. jQuery is becoming increasingly popular for its simplicity and elegant syntax. It has been so popular in fact, that Microsoft has decided to ship jQuery along with all ASP.NET MVC projects. Visual Studio 2010 will have jQuery included by default for all web applications. The Microsoft Ajax client library that comes with ASP.NET AJAX is also used for a few of the Ajax helpers, most notably `<% Ajax.BeginForm() %>`. We will see how this functions later in this chapter.

jQuery is a javascript library that make Javascript development more concise, more consistent across browsers, and more enjoyable. jQuery has a powerful selector system, where you use CSS rules to pinpoint and select elements from the DOM and manipulate them. The entire library is contained in a single minified Javascript file (`jquery.js`) and is freely available for download at <http://www.jquery.com>. Take a look at the sidebar below to get a quick primer on how to use jQuery. There are many other excellent Javascript libraries that you can use with the ASP.NET MVC Framework as well. Prototype & script.aculo.us, dojo, mootools, YUI, etc, all have strengths and weaknesses; however jQuery will be included in all MVC projects by default. At the time of writing the current version of jQuery is 1.31.

### A jQuery Primer

To use jQuery, you must reference the `jquery.js` Javascript file in the `<head>` element of your page.

The `$( )` function accepts a string and is used to:

- Select elements by ID or CSS selector (i.e. `$('#myDiv')` => `<div id="myDiv" />`)
- Select elements within some context (i.e. `$('#input:button', someContainer)`)
- Create HTML dynamically (i.e. `$('#<span>updating...</span>')`)
- Extend an existing element with jQuery functionality (i.e. `$(textbox)`)
- Execute a function once the entire DOM is ready (i.e. `$(do_stuff)` => executes `do_stuff()` when the DOM has been loaded (without waiting for images to load).

To have some code executed when the DOM is ready, rather than putting the script at the bottom of the page, you can put it in the `<head>` by doing:

```
$(document).ready(function() { /* your code here */ });
```

This is the same as

```
$.ready(function() { /* your code here */ });
```

It can be shortened even further like so:

```
$(function { /* your code */ });
```

(There's usually a shorter way of doing *anything* in jQuery). The nice thing about `$(document).ready` is that it will fire as soon as the DOM is loaded, but it doesn't wait for images to finish loading. This results in a faster startup time than if you relied on `window.onload`.

The `$.ajax([options])` function can be used to send Ajax requests to the server. `$.get()` and `$.post()` are also useful simplifications of the `$.ajax()` function.

To serialize a form's values into `name1=val&name2=val2` format, use `$(form).serialize()`.

I have just scratched the surface here. For a real introduction on jQuery, visit the jQuery website. I highly recommend the book **jQuery in Action** also from Manning for more serious studies.

*For more detailed information, see the documentation online at <http://docs.jquery.com>.*

The first example in this chapter used a button click to fire off the request. There were no parameters sent to the server, so the same message would always be returned. This is hardly a useful way to build Ajax applications. Another more realistic approach (and one that is quite popular) is to take a form and hook into the `onsubmit` event. The form values are sent via Ajax instead and the standard form submission is cancelled. Jeremy Keith calls this technique “Hijax.”

### **9.3.1 Hijaxing Code Camp Server**

Our first example will cover a small feature in Code Camp Server. We will implement the hijax technique. Let’s take a look at the user story for this feature:

*As a potential speaker, I would like to add sessions to the conference (with a name and description) so that the organizer can review them and approve the ones that fit. I would like the interaction to be seamless so that I can add multiple sessions very quickly.*

Figure 9.2 is the form (in Code Camp Server) where you can add sessions to a conference. It consists of two textboxes, a dropdown list, and a submit button. When the form is submitted, a track is created and added to the conference, and then the page is rendered again with a styled list of current tracks.

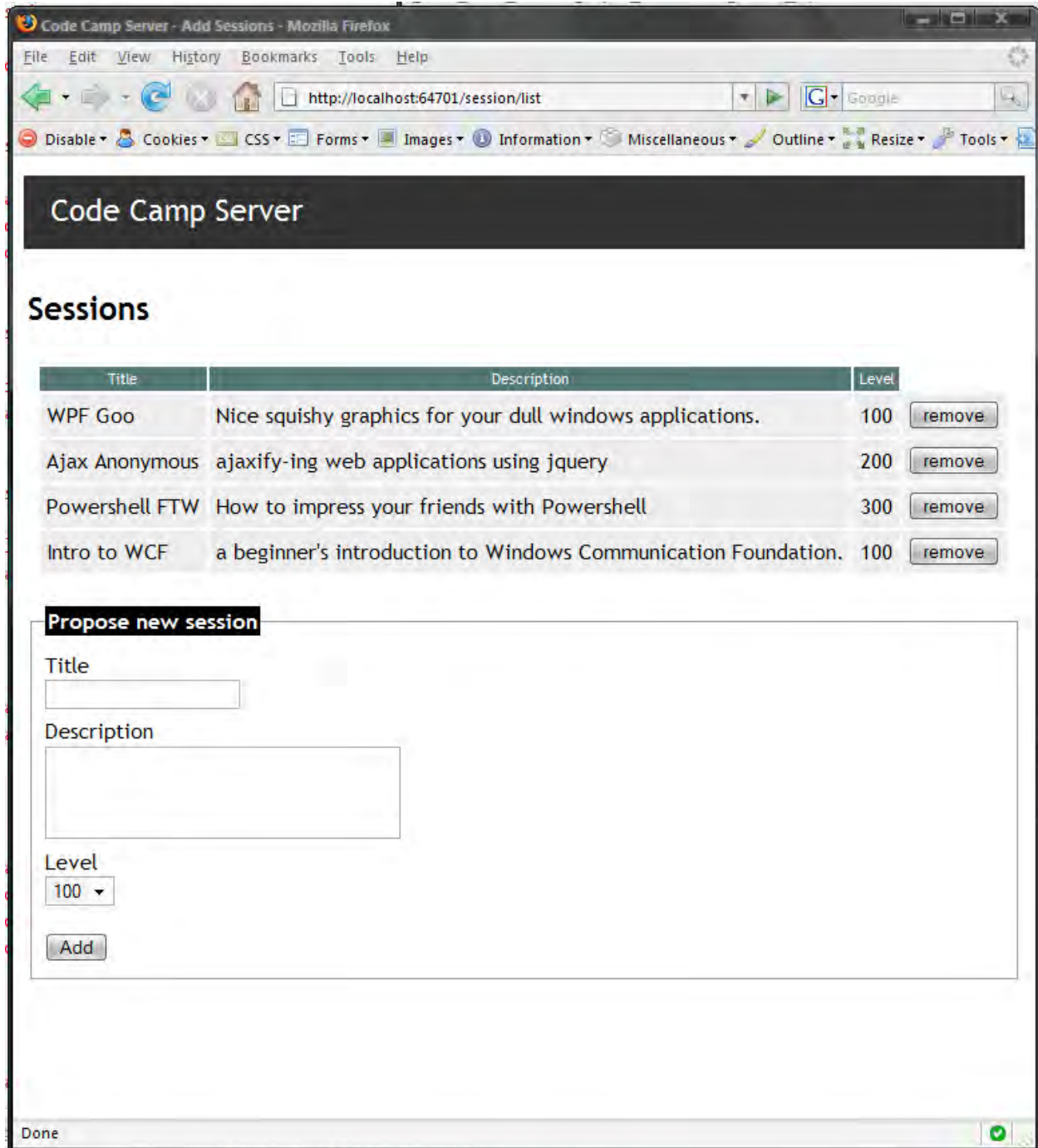


Figure 9.2 – These form values are serialized and sent to the server via Ajax. The result is a seamless method of adding sessions without a page refresh. Of course, when you disable Javascript it still works the old way.

When you submit the form, the session is added and the user is redirected back to /session/index to view the updated table. The HTML behind this form looks like:

```
<% using(Html.BeginForm("add", "session",
    FormMethod.Post, new{id="new_session"})) { %>
<fieldset>
  <legend>Propose new session</legend>
  <label for="title">Title</label>
  <input type="text" name="title" />

  <label for="description">Description</label>
```

```

<textarea name="description" rows="3" cols="30"></textarea>

<label for="level">Level</label>
<select name="level">
  <option selected="selected" value="100">100</option>
  <option value="200">200</option>
  <option value="300">300</option>
  <option value="400">400</option>
</select>

  <input type="submit" value="Add" />
</fieldset>
</form>

```

It is important to ensure that your application works without Ajax, because your users might decide to run with Javascript turned off. Our example works, so we can now focus on spot-welding Ajax onto this form without touching the html. We can apply a simple jQuery script that will hijack this form post and provide the seamless Ajax experience instead (when the user has enabled Javascript).

Let's see how that is implemented, shall we? When the user clicks on the submit button the browser physically posts to the server. We need to cancel this action so the browser doesn't go anywhere. If we add an `onsubmit` Javascript handler to the form and return `false`, then we can capture the form post and circumvent the *actual* post operation. We can then gather the form values and submit the form post instead with Ajax.

#### Listing 9.4 - The jQuery script that sets up the form *hijacking*

```

$(document).ready(function() {
  $("form#new_session").submit(function() {
    hijack(this, update_sessions, "html");
    return false;
  });
}); #1

function hijack(form, callback, format)
{
  $("#indicator").show();
  $.ajax({
    url: form.action,
    type: form.method,
    dataType: format,
    data: $(form).serialize(),
    completed: $("#indicator").hide(),
    success: callback
  });
} #2

function update_sessions(result)
{
  $("#session-list").html(result);
  $("#message").hide().html("session added")
  .fadeIn(2000)
  .fadeOut(2000);
} #3

```

**#1** This block executes when the DOM has been loaded. This wires up our form's `onsubmit` handler.

**#2** Sends the form data via Ajax. The `callback` is a function that will be called when the response is received.

**#3** This is the callback function we specified. It replaces the content of the `session-list` (our table of sessions above) and shows a fading message indicating that something happened.

#### WARNING:

When using `return false` in your event handlers to prevent default behavior in the browser, be sure to add proper error handling. If an error occurs before our `return false` statement, then it won't be passed down to the caller and the browser will continue with the form post behavior. At the very least, surround this behavior in a `try {} catch {}` block and alert any errors that occur. Detecting and tracking down Javascript errors after the browser has left the page is difficult and annoying.

This script can reside in a separate file referenced by the page or in a script tag of the `<head>` element. It is sometimes common to see `<script>` tags in the middle of the `<body>`, but this is not valid XHTML.

Notice how the Ajax call is made. The `$.ajax()` method accepts a number of options for customizing the call. Isn't this a lot cleaner than our manual approach? More simplified Ajax calls might opt to use `$.post()` or `$.get()`. Read up on the jQuery documentation to see the various options you have available to you.

Now, the form submits via Ajax when Javascript is enabled, which is what we were aiming for. Nobody loses functionality in the absence of Javascript, but rather the experience is *enhanced* with Javascript. The best part about this hijax technique is that it is purely *additive*; you simply apply the extra Javascript to an existing functioning form to enhance it with asynchronous behavior.

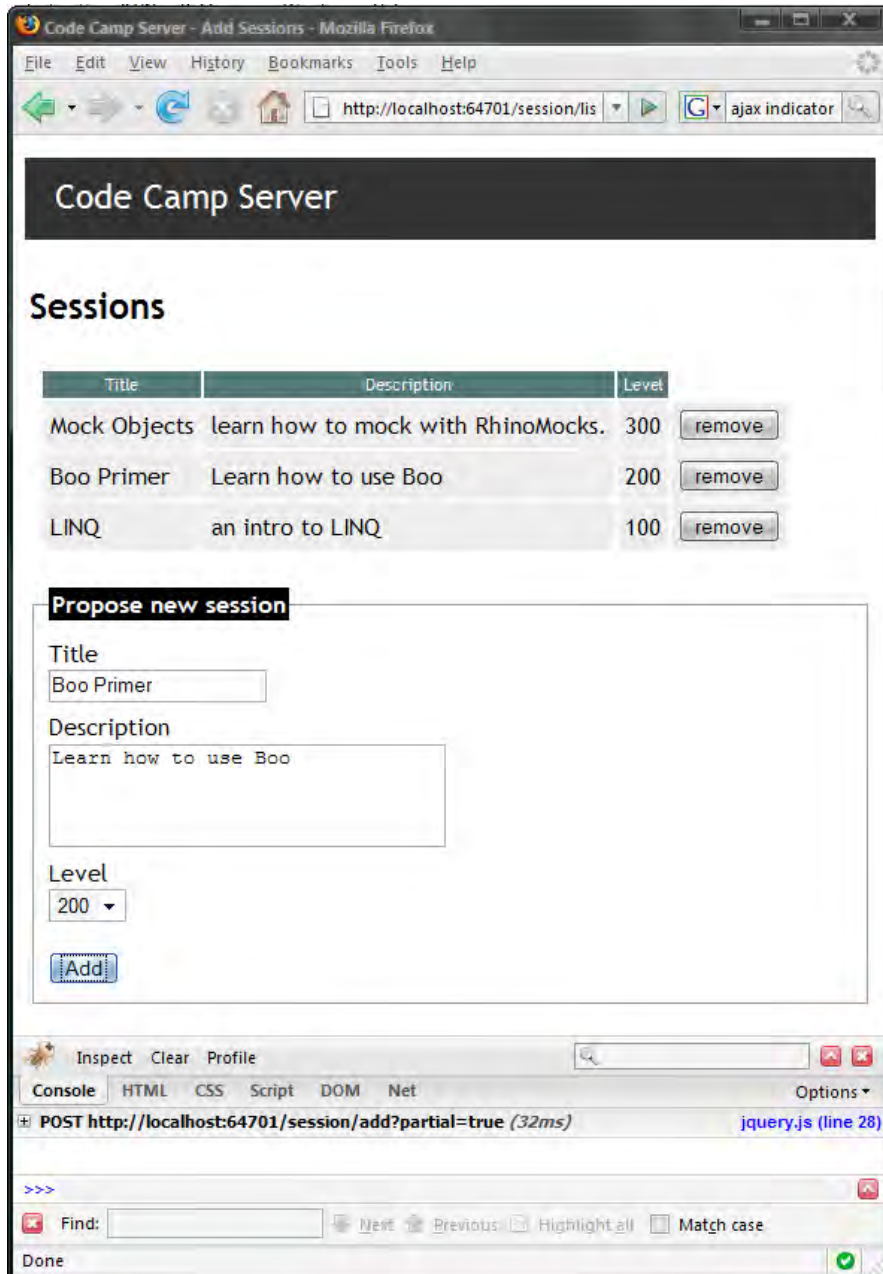


Figure 9.3– When an Ajax call is initiated Firebug shows it in the Console. You can use this tool to inspect the actual request and response of an Ajax call. Firebug is invaluable when doing Ajax development.

Listing 9.5 shows the `SessionController` actions in detail. Notice how we are reusing the same actions for both full layout and partial html requests. This is implemented with a user control called `_list.ascx`. This user control is embedded in the full layout, and rendered independently for partial requests.

#### Listing 9.5 – The actions for `SessionController`.

```
public ActionResult Index()
{
    var sessions = _sessionRepository.FindAll();

    if(Request.IsAjaxRequest())
        return View("_sessionList", sessions);           #1

    return View(sessions);
}

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Add(string title, string description, string level)
{
    var session = new Session
    {
        Id = Guid.NewGuid(),
        Title = title,
        Description = description,
        Level = level
    };

    _sessionRepository.SaveSession(session);

    if(Request.IsAjaxRequest())
        return Index();                                 #2

    return RedirectToAction("index");
}
```

**#1 If this is an Ajax request, we need to render the partial view, not the full one**

**#2 Ajax requests will need the partial view returned directly. Full requests can simply be redirected**

The `Index` action checks to see whether the request is an Ajax request. If so, it will render the user control that represents the HTML fragment being displayed. If it is a regular request, the full HTML document (with the template) will be rendered.

The `Add` action is decorated with a `AcceptHttpVerbs` attribute to protect it from GET requests. If this is an Ajax request – which is defined by an extra HTTP header in the request - then the response needs to be the updated session list HTML. In the standard case without Ajax, the browser should redirect to the `List` action.

The Ajax technique that we've applied here is both easy to implement (with the help of jQuery) and easy to understand. This is probably the most common method of applying Ajax. Don't believe me? This is essentially what the beloved `UpdatePanel` does in ASP.NET AJAX. We hear advertisements for other commercial Ajax components for "no-touch Ajax" or "zero-code Ajax" all the time. This is basically the technique they are using. Your authors firmly believe that "no-code" solutions are great for some scenarios, but they break down and become difficult to work with in more complex situations. It is often better to leverage a simple framework that lets you explicitly control the Ajax integration to give you the flexibility to adapt your application to increasingly complex functionality requirements. Here we have applied a simple script than can be re-used on other pages to enhance a page with Ajax.

This example returned a snippet of HTML to the client. Sometimes we don't want HTML as our return value. HTML is the heaviest of the choices because it contains all of the formatting along with the data. Our example returned the entire rendered table. If over-the-wire performance is a concern (for example, if you intend to have users on slow connections or you have a lot of data to transfer) then you might opt for a lighter-weight representation of the data. If updated display information is needed, then Javascript can dynamically build DOM elements to represent the data.

There are two more common choices in data formats for Javascript calls: XML and JSON. JSON stands for Javascript Object Notation, and is much lighter weight than XML. Plain text is also sometimes useful if you don't need any structure to your data.

### 9.3.2 Ajax with JSON

Our next example will be a speaker listing. We will see the names of the speakers in a list. If the user clicks on a speaker's name, he will be directed to a speaker detail page.

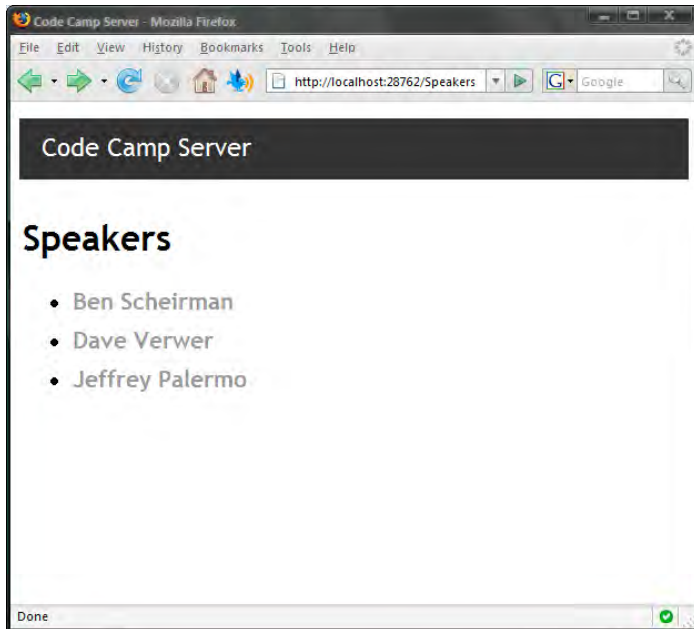


Figure 9.4 – Listing the speakers. When you click on the name, the user is directed to a speaker detail page.

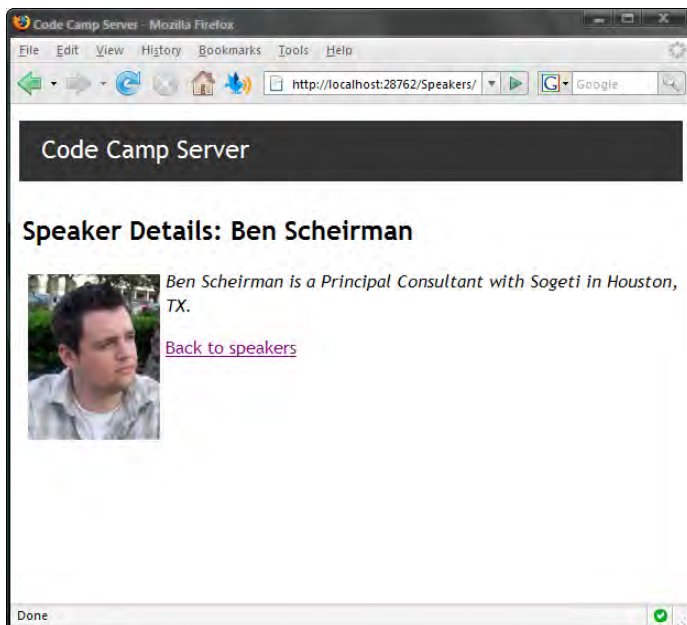


Figure 9.5 – The speaker details are shown on a separate page.

Let's provide a richer user experience by applying Ajax to the speaker listing page. We'd like to enhance the speaker listing to show the speaker details next to the name when the user hovers over the name with the mouse.

To accomplish this, we will leverage JSON as our transfer format. Why JSON? Well first off, our previous example used HTML, which we can all agree is verbose over-the-wire. If this is a concern, then we should be transmitting data only, leaving presentation to the client. One choice might be to represent the data using XML. Let's take a look at a sample XML document:

#### Listing 9.7 – a simple XML document representing a speaker contains a lot of extraneous noise

```
<speaker>
  <id>313bd98d-525c-4566-bfa1-7a4f8b01ef7b</id>
  <firstName>Ben</firstName>
  <lastName>Scheirman</lastName>
  <bio>Ben Scheirman is a Principal Consultant with Sogeti in Houston, TX.</bio>
  <picUrl>/content/ben.png</picUrl>
</speaker>
```

There is a lot of noise text in there (such as all of the closing tags). The same example, represented in JSON looks like this:

#### Listing 9.8 – a simple JSON string representing a speaker is much more lightweight

```
{
  "id": "313bd98d-525c-4566-bfa1-7a4f8b01ef7b",
  "firstName": "Ben",
  "lastName": "Scheirman",
  "bio": " Ben Scheirman is a Principal Consultant with Sogeti in Houston, TX.",
  "picUrl": "/content/ben.png"
}
```

The JSON format is pretty easy to understand, once you understand the basic rules. At the core, an object is represented like Figure 9.4.

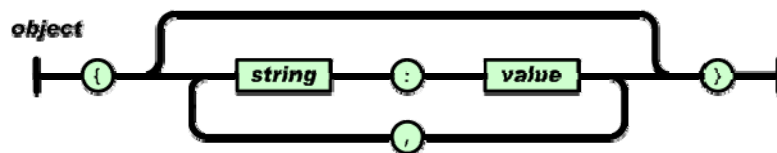


Figure 9.4 - The JSON object diagram shows us a simple way of understanding the format. Taken from <http://json.org>

Isn't the JSON representation a lot more concise? Sure it might be a tad harder to read, but this is primarily for machines to consume, not humans. JSON documents will require fewer bytes to transmit (21% fewer in the above example), leading to less strain on the server and faster download times for your users. But this isn't the only reason that JSON is a better choice. JSON *is* Javascript. Your result can be evaluated and treated as a first-class Javascript object. (This evaluation is much faster than parsing XML as well). Take your pick: get a real Javascript object, or deal with XML parsing & manipulation.

A number of .NET JSON libraries can make your life easier. I've used JSON.NET by Newtonsoft, which is free to use and works well. You can download it at <http://www.codeplex.com/json>. The ASP.NET MVC Framework also includes a mechanism for serializing objects into JSON, which we'll see in a minute.

Now that we have settled on the JSON format for our Ajax feature, how do we get the controller to render it? Let's see how we can accommodate different view formats in our controllers.

### 9.3.3 Adding Alternate View Formats to the Controller

Currently we have a controller action that finds the speaker from our repository and renders a "detail" view, passing the speaker in as ViewData. We would like to take advantage of this action, but alter the view that gets rendered. We still want to get a speaker based on the name, but in our Ajax call we'd like the server to return a JSON string instead of an HTML document.

### Listing 9.8 – The controller action before any modifications

```
public ActionResult Details(string id)
{
    var speaker = _repository.FindSpeakerByName(id.Humanize());
    return View(speaker);
}
```

The `id` passed in is a “URLized” representation of the name. Thus, “Ben Scheirman” becomes “ben-scheirman” in order to have cleaner URLs. This method is not safe for names that are already hyphenated (those would have to be escaped) or contain any other special characters, but for our simple example it works. We perform the reverse operation, called `Humanize()`, to get the actual name to search for.

#### NOTE:

Rather than searching for a single record by name, we could instead have a pseudo-key stored in the database that we can use as a unique, human-readable, URL-friendly identifier. This is sometimes called a *slug*. This would avoid the problem of hyphenated names or names with invalid URL characters. If we employed this technique, our URL would look like `/speakers/13/ben-scheirman`. The 13 would be a unique identifier, and the remaining segment of the URL would simply exist for the benefit of readability. Refer to Chapter 5 for more information on creating custom routes like this.

In our Ajax case, we don’t want an entire view to be returned from the action. This would result in a large HTML document being returned in an Ajax call. For an Ajax call, we want to return the JSON data directly. We’ll leverage the same technique we did in Listing 9.5 and notify the action about the type of request. We can also use this opportunity to allow for multiple formats to be rendered.

The modified, shown in Listing 9.9 action accepts an optional format as an argument. Valid values would be *html* (the default), *partial* (for HTML fragments), *xml*, and *json*. Our view can choose to respond to any one - or all - of those formats.

### Listing 9.9 – A modified controller action that accepts an optional format

```
public ActionResult Details(string id, string format)
{
    var speaker = _repository.FindSpeakerByName(id.Humanize());

    if(format == "json")
        return Json(speaker);

    return View(speaker);
}
```

The `Json()` method returns a `JsonResult` from the action and contains the object formatted as JSON.

#### TIP:

You can send anonymous objects to the `Json()` method and have your object serialized to JSON format correctly. This is useful when you want to return some JSON data that does not directly map to a class in your project. For example, this is valid:

```
return Json( new { Name="Joe", Occupation="Plumber" } );
```

To test out our different rendering formats, we will open up the same speaker detail page from before, but this time we’ll add `?format=json` to the end of the URL. The MVC framework will match up query string and form parameters to action arguments if the names are the same. We could easily add more formats, such as XML. In the event that `format` is omitted (like in our original URL) then this value will be `null`.

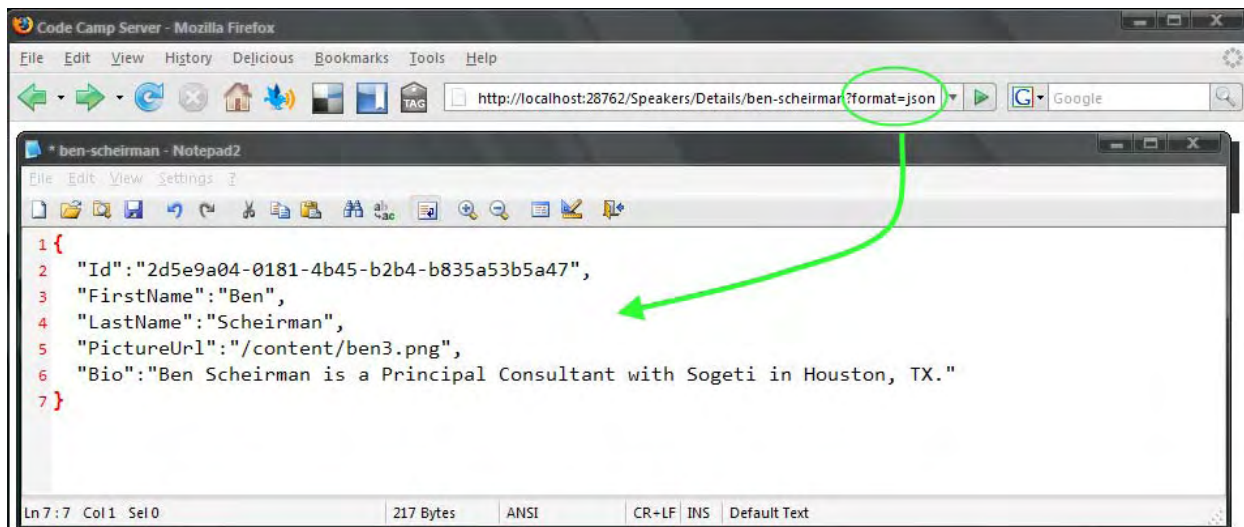


Figure 9.5 – Seeing our JSON result from the browser.

Now that we have our JSON enabled Ajax action ready for use, let's see how we can modify the speaker listing page to consume this.

### 9.3.4 Consuming a JSON Action from the View

The first task is to hook into the *mouseover* and *mouseout* events of each list item. When the user hovers over a list item, an Ajax call will be made to get the speaker details (as JSON) and construct a small detail box along-side the link.

#### Listing 9.11 – Hooking up hover behavior on each of the links

```
$(document).ready(function() {
  $("ul.speakers a")
    .mouseover(function() {
      show_details(this);
    }).mouseout(function() {
      hide_details(this)
    });
});
```

It may not be apparent at first glance, but the `$("ul.speakers a")` function above is actually a CSS selector that returns multiple elements. We attach a handler to each element's *mouseover* and *mouseout* events.

Next we have to actually do something when the user hovers over the link. I added a hidden `<div>` tag to each list item that serve as the container for the speaker's detailed information. The `show_details()` function should show this box along with an Ajax loading indicator. When the data comes back from the server, we will build some elements to display the information.

#### Listing 9.12 – When the user is hovering over the link

```
function show_details(link)
{
  var box = $(link).next(); #1
  box.show();

  box.html('');
  $('<img/>').attr('src' '/content/load.gif').attr('alt', 'loading...').appendTo(box); #2

  var url = link.href.replace(/\?format=html/, "?format=json");
  $.get(url, function(result) { #3
    loadSpeakerDetails(box, eval(result));
  }, "json");
}
```

- #1 The box is sitting next to the link
- #2 Clear the existing box contents & replace with a loading graphic
- #3 Execute the Ajax request to get the details

This function has a lot going on, so let us break it down for each step. The link itself is passed in to the function, and we know that the “box” (the container where we’ll put the user details) is the very next node in the DOM, so we use the jQuery `next()` function to retrieve it. We then clear the contents of it and display it. The next line creates an image tag pointing to `load.gif` and appends it inside the box element. To retrieve the JSON object for the speaker details we have to use the same URL as the link, but we need to replace the format to specify “json,” so we use a regular expression to do the replacement for us. Finally we issue an Ajax GET request for the URL. The callback for this Ajax operation is the next function, `loadSpeakerDetails`.

#### Listing 9.13 – Creating the HTML to display the speaker details

```
function loadSpeakerDetails(box, speaker)
{
    box.html('');
    $('
```

- #1 Clear out the loading graphic
- #2 Building an image to display the speaker picture
- #3 Building a span tag to hold the speaker bio

In this function we are simply creating a few HTML elements to display the user details and we are adding them to the box element. The last thing to do is hide the box when the user leaves the link region.

#### Listing 9.14 – When the user leaves the link.

```
function hide_details(link)
{
    $(link).next().hide();
}
```

Using jQuery in these examples has allowed me to be productive and expressive, while not worrying about cross browser Javascript quirks and incompatibilities. The resulting code is more durable and more concise. A good Javascript library such as jQuery is a must in any web developer’s tool belt.

All of the pieces are now tied together, and we can now see the results of our work. In Figure 9.6 you can see the Ajax call at the bottom (in the Firebug window) and the page gives us the information we need without any page redirects or refreshes. How refreshing!



Figure 9.6 – Our finished Ajax-enabled page.

### 9.3.5 Ajax Helpers

The ASP.NET MVC Framework ships with a couple of Ajax helpers that you can use to quickly create Ajax behaviors on your site. Just as the HTML helpers are accessed with `<%= Html.HelperName() %>` the Ajax helpers are accessed via `<%= Ajax.HelperName() %>`. In order to utilize these helpers in your application you must reference `MicrosoftAjax.js` and `MicrosoftMvcAjax.js`, which are included in the project template in the `/scripts` folder. It is safe to reference these in combination with jQuery.

#### AJAX.ACTIONLINK

The first Ajax helper that we will examine is `Ajax.ActionLink`. This helper provides the ability to invoke an action asynchronously and update an element on the page. The usage is simple:

```
<%= Ajax.ActionLink("Click here", "GetMessage", new AjaxOptions {
    UpdateTargetId = "message_container",
    InsertionMode = InsertionMode.Replace
}) %>
```

This will render a link with displayed text “Click here.” When the user clicks on the link the `GetMessage` action will be invoked via Ajax. The response from this action (probably some HTML fragment) will be placed in an element with id “message\_container.” The available parameters you can pass to the `AjaxOptions` class to customize the behavior of the link are:

HttpMethod	can be “GET” or “POST”. The default is “GET”
UpdateTargetId	The element that will receive the content
InsertionMode	Can be <code>InsertBefore</code> , <code>InsertAfter</code> , or <code>Replace</code>

OnBegin	Javascript function to be called before invoking the action
OnComplete	Javascript function to be called after the response comes back
OnFailure	Javascript function to be called in the event of an error
OnSuccess	Javascript function to be called if no errors occur
Confirm	Confirmation message to provide an OK/Cancel dialog before proceeding
Url	Url to use if the anchor tag has a different destination than the Ajax request
LoadingElementId	An element that displays Ajax progress. The element should be marked as <code>visibility:hidden</code> initially.

### WARNING:

It is tempting to put a simple Javascript expression in the `OnBegin` handler or its counterparts, however this causes a syntax error in the generated `onclick` handler for the anchor tag. Make sure you simply reference the Javascript function by name (without parentheses) like this: `OnBegin = "ajaxStart"`

The Ajax link is just one of the helpers that you can use to invoke an action asynchronously. It is useful in scenarios where there isn't much more than notifying the server of some action, or retrieving a simple value. For more complicated scenarios, where there is data to be sent to the server, an Ajax form is more appropriate.

### AJAX-BEGINFORM

The Ajax form is achieved through an Ajax helper called `Ajax.BeginForm`. It behaves similar to the *hijax* technique from earlier in this chapter. Usage is similar to the Ajax action link:

```
<% using(Ajax.BeginForm("AddComment", new AjaxOptions{
    HttpMethod = "POST",
    UpdateTargetId = "comments",
    InsertionMode = InsertionMode.InsertAfter})) { %>

    <!-- form elements here -->

<% } %>
```

The same `AjaxOptions` class applies to this helper, and is used in the same way. In this example the form is appending comments to an element on the page.

### WHAT'S WITH THE USING() BLOCK?

The using block above might look a bit strange to you. It is purely optional, however it does give you the benefit of automatically entering your closing form tag through the magic of the `IDisposable` interface. You are free to do it the other way, like:

```
<%= Ajax.BeginForm() %>

</form>
```

However, it looks a bit unbalanced. The choice is yours to make.

The Ajax helpers can quickly give you Ajax behaviors; however they do have a couple of drawbacks that are difficult to ignore. First, you can see that even simple examples require many lines of code – code that is mixed in with your HTML markup. For more advanced scenarios you can easily eat up 10 lines or more, which detracts from readability. Secondly the actual Javascript is hidden from you, so you cannot reliably trap errors that occur as a result of your Javascript handlers. Server errors will be trapped by the `OnError` handler, however if your `OnBegin` code throws an error then your Ajax behavior cannot be completed. Because of these deficiencies many will choose to simply write the Javascript by hand and get more control over the Ajax interaction. The jQuery

samples in this chapter should have given you all you need to create the same effect with pure jQuery. That said, the Ajax helpers do allow you to get quick Ajax functionality for minimal effort.

## **9.4 Summary**

Ajax is an important technique to today's web applications. Using it effectively means that the majority of your users will get a better experience, but also does not 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 good libraries such as jQuery, Prototype & script.aculo.us, Mochikit, and others, you can be much more productive.

In this chapter you have learned how to apply Ajax in different ways: using partial HTML replacement and JSON. You have learned how to hijack a form submission and provide a more seamless Ajax experience for those users that support Ajax, while continuing functionality for those who don't. Throughout this chapter you have seen how to apply jQuery, a productive Javascript library.

Next up, we will take a look at hosting and deployment options for ASP.NET MVC.