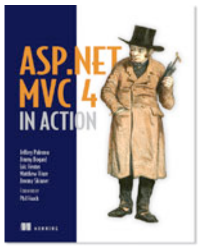


Ajax with JSON and Client Templates

By Jeffrey Palermo, Jimmy Bogard, Eric Hexter, Matthew Hinze, and Jeremy Skinner, authors of *ASP.NET MVC 4 in Action*

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. In this article, based on chapter 7 of ASP.NET MVC 4 in Action*, the authors show you 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.*

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.

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 1 shows an XML representation of a speaker at our fictitious conference, while listing 2 shows the same data represented in JSON.

Listing 1 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 2 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 1.

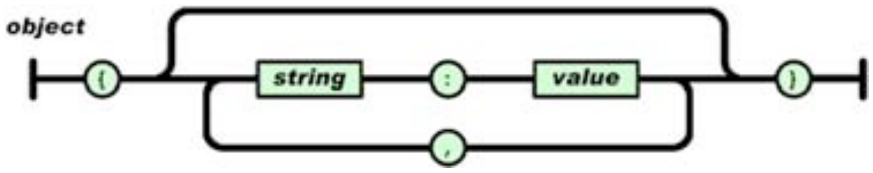


Figure 1 The JSON object diagram shows a simple way of understanding the format. (Used with permission from http://json.org.)

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

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 2.

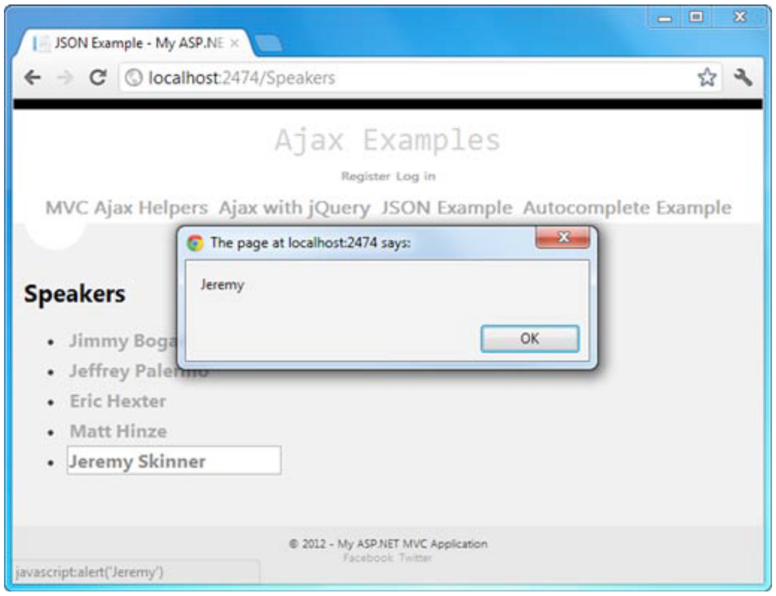


Figure 2 Displaying the speaker's first name as the result of an Ajax request.

Here's the basic implementation.

Listing 3 The `SpeakersController`

```
public class SpeakersController : Controller
{
    private SpeakerRepository _repository   #A
      = new SpeakerRepository();   #A

    public ActionResult Index()
    {
        var speakers = _repository.FindAll();   #1
        return View(speakers);   #2
    }

    public ActionResult Details(int id)
    {
        var speaker = _repository.FindSpeaker(id);
        return Json(speaker,   #3
            JsonRequestBehavior.AllowGet);
        }
    }
```

#A Instantiate repository
#1 Retrieve list of speakers
#2 Pass speakers to view
#3 Serialize speaker to JSON

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.

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 3 that we have to pass an enum value of `JsonRequestBehavior.AllowGet` to the controller's JSON method. By default, ASP.NET MVC's `JsonResult` will only work in response to an HTTP `POST`. If we want to return JSON in response to a `GET` 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.

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.

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 4 The speaker list page

```
@model IEnumerable<AjaxExamples.Models.Speaker>  #1
<link rel="Stylesheet" type="text/css" #2
        href="@Url.Content("~/content/speakers.css")" />

<script type="text/javascript" #3
  src="@Url.Content("~/scripts/Speakers.js")"></script>

<h2>Speakers</h2>

<ul class="speakers">  #4
@foreach (var speaker in Model) {
  <li>
      @Html.ActionLink(speaker.FullName, "Details",
        new { id = speaker.Id })
  </li>
}
</ul>

<img id="indicator" #5
  src="@Url.Content("~/content/load.gif")"
  alt="loading..." style="display:none" />

<div class="selected-speaker" #6
  style="display:none"></div>
```

#1 Strongly typed view
#2 CSS reference
#3 Custom script reference
#4 Generate list of speakers
#5 Display progress spinner
#6 Results container

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 (#5) (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 the next section.

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

Listing 5 Client-side behavior for the speakers page

```
$(document).ready(function () {
    $("ul.speakers a").click(function (e) {
        e.preventDefault();

        $("#indicator").show(); #1

        var url = $(this).attr('href'); #2

        $.getJSON(url, null, function (speaker) { #3
            $("#indicator").hide();
            alert(speaker.FirstName); #4
        });
    });
});
```

#1 Show progress indicator
#2 Retrieve URL
#3 Invoke Ajax
#4 Display result request

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 deserialized 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.

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 3.

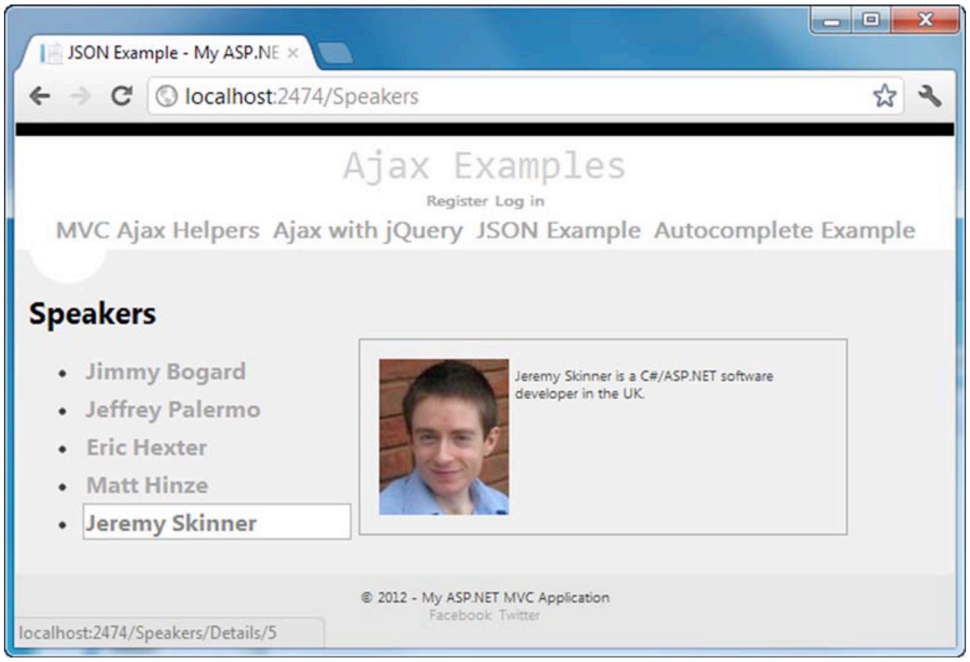

JSON Example - My ASP.NE
localhost:2474/Speakers
Ajax Examples
Register Log in
MVC Ajax Helpers Ajax with jQuery JSON Example Autocomplete Example
Speakers
Jimmy Bogard
Jeffrey Palermo
Eric Hexter
Matt Hinze
Jeremy Skinner
Jeremy Skinner is a C#/ASP.NET software developer in the UK.
© 2012 - My ASP.NET MVC Application
Facebook Twitter
localhost:2474/Speakers/Details/5

Figure 3 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 6 Using client-side templates

```
<script type="text/javascript" #1
   src="@Url.Content("~/scripts/jquery.tmpl.js")">
</script>

<script id="speakerTemplate" type="text/x-jquery-tmpl"> #2
    <img src="${PictureUrl}"
        alt="Speaker image" class="speaker-pic" />  #3

    <p class="speaker-bio">
        ${Bio} #4
    </p>

    <br style="clear:both;" />
</script>
```

#1 Reference jQuery templates
#2 Define template section
#3 Photo template
#4 Bio line template

We begin by including a reference to the jQuery-tmpl script from our scripts folder (#1) 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 Modifying our script to render the template

```
$(document).ready(function () {
    $("ul.speakers a").click(function (e) {
        e.preventDefault();

        $(".selected-speaker").hide().html('');  #1
        $("#indicator").show();

        var url = $(this).attr('href');

        $.getJSON(url, null, function (speaker) {
            $("#indicator").hide();

            $("#speakerTemplate")  #2
                .tmpl(speaker)
                .appendTo('.selected-speaker');
            $('.selected-speaker').show();
        });
    });
});
```

#1 Hide speaker details
#2 Render template with data

This code is mostly the same as the code in listing 5 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 3. Note that extra styling has been added to make the page look presentable.

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 render a view if the action has not been requested via Ajax.

Listing 8 Adding graceful degradation to the `Details` action

```
public ActionResult Details(int id)
{
    var speaker = _repository.FindSpeaker(id);
    if(Request.IsAjaxRequest()) #A
    {
        return Json(speaker,
          JsonRequestBehavior.AllowGet);
    }
    return View(speaker); #B
}
```

#A Return JSON for Ajax requests
#B Render view for non-Ajax requests

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 can create an `AcceptAjaxAttribute` by simply inheriting from the `ActionMethodSelector` attribute, as shown here.

Listing 9 Implementing the `AcceptAjaxAttribute`

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
```

```
public class AcceptAjaxAttribute : ActionMethodSelectorAttribute
{
    public override bool IsValidForRequest(
      ControllerContext controllerContext, MethodInfo methodInfo)
    {
        return controllerContext.HttpContext
         .Request.IsAjaxRequest();
    }
}
```

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 10 Using the `AcceptAjaxAttribute`

```
[AcceptAjax]   #1
public ActionResult Details(int id)
{
    var speaker = _repository.FindSpeaker(id);
    return Json(speaker, JsonRequestBehavior.AllowGet);
}

[ActionName("Details")]   #2
public ActionResult Details_NonAjax(int id)
{

    var speaker = _repository.FindSpeaker(id);
    return View(speaker);
}
```

#1 Accessible only for Ajax requests
#2 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 `AcceptAjaxAttribute`, 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 11 Non-Ajax speaker details

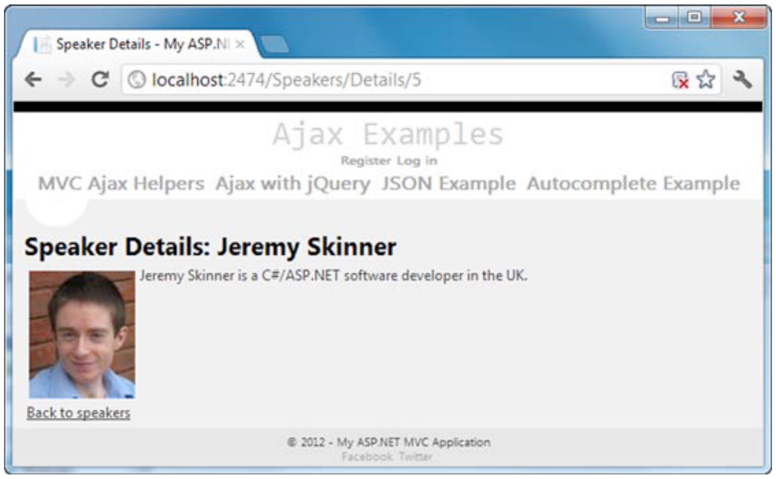
```
@model AjaxExamples.Models.Speaker
<h2>Speaker Details: @Model.FullName</h2>

<p class="speaker">
    <img src="@Model.PictureUrl" #A
      alt="@Model.FullName" />
    <span class="speaker-bio">@Model.Bio</span> #B
</p>

<br style="clear:both" />
@Html.ActionLink("Back to speakers", "index")
```

#A Display speaker photo
#B Display speaker bio line

When we now click the speaker's name with JavaScript disabled, we'll be taken to a separate page, as shown in figure 4.


Speaker Details - My ASP.N
localhost:2474/Speakers/Details/5
Ajax Examples
Register Log in
MVC Ajax Helpers Ajax with jQuery JSON Example Autocomplete Example

Speaker Details: Jeremy Skinner

Jeremy Skinner is a C#/ASP.NET software developer in the UK.

Back to speakers

© 2012 - My ASP.NET MVC Application

Figure 4 Speaker details displayed without Ajax

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.

You've seen how to apply Ajax using JSON. 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.

Last updated: August 2, 2013