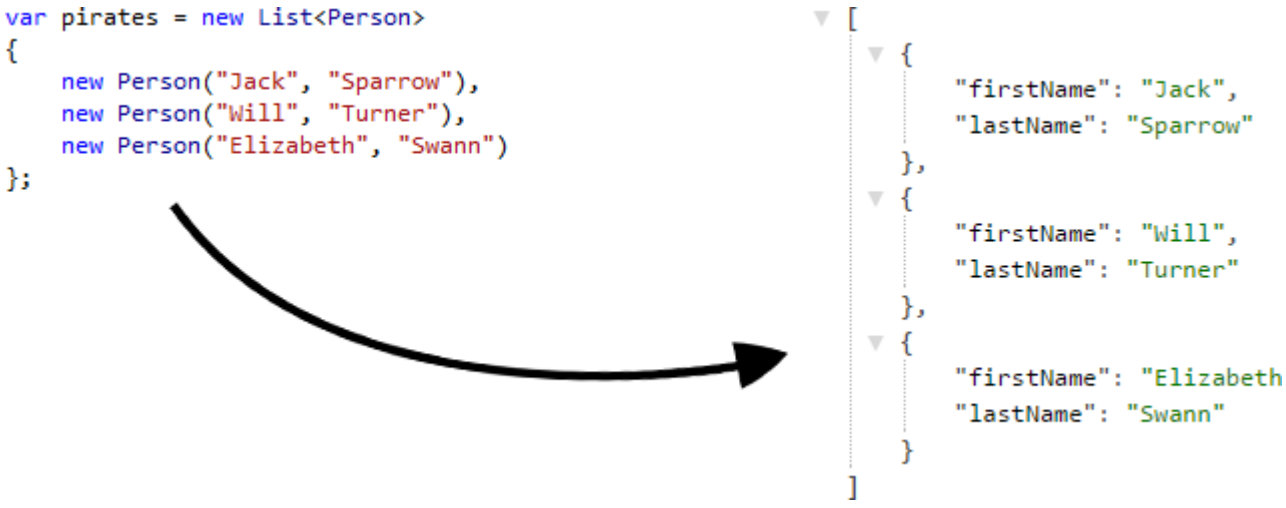
Marius Schulz

Passing .NET Server-Side Data to JavaScript

February 5, 2014

There are a lot of different ways to pass C# data from an ASP.NET back end to a JavaScript front end. In this post, I want to highlight a variety of methods and point out their pros and cons. In particular, I'll focus on how to embed data within the HTML document that's being loaded.



Method #1: Fetching Data by Making an AJAX Request

I would argue that this is the classic: After an HTML page has finished loading, an AJAX request is made to fetch some data from an endpoint provided by the back end. Borrowing from *Pirates of the Caribbean*, this could look something along the lines of the following:

```
var pirates = [];

// Assuming you have referenced jQuery
$(function() {
    $.getJSON("/black-pearl/crew", function(crewResponse) {
        pirates = crewResponse.pirates;
    });
});
```

- **Pros:** Making an AJAX request is a well-known, easy-to-implement solution to pass data from ASP.NET to JavaScript. With technologies such as ASP.NET Web API, pretty much all of the plumbing work like content negotiation and serialization is done for you.
- **Cons:** An additional HTTP request takes time to complete, which means the requested data isn't available immediately after page load. You'll also have to adapt your code to work with data coming in asynchronously.

Method #2: Loading Data Through an External JavaScript File

Instead of directly fetching the desired data from an endpoint through an AJAX request, you can also put the data in an external JavaScript file and reference it in a `<script>` tag. You can even embed Razor code withing that script file while still getting first-class tooling from Visual Studio, as shown in my blog post Generating External JavaScript Files Using Partial Razor Views. You'd then simply reference the script file like this:

```
<script src="/black-pearl.js"></script>
```

Using an external JavaScript file is very similar to making an AJAX request and pretty much has the same pros and cons. However, working with a classical AJAX request is probably a little nicer because you can very easily register a callback to be executed once the response arrives; doing the same with external JavaScript files might be more cumbersome. For the sake of completeness, this method is included in this overview, though.

Method #3: Opening a Persistent Connection with SignalR

You can also choose to open a persistent connection to your server with SignalR. This is a great method for any type of application which benefits from real-time data, like chat programs or client/server games.

Under the hood, SignalR tries to establish the connection using Web Sockets if both the server and the client support them. If not, it gracefully falls back to mechanisms like server events, forever frames, long polling, etc. and thereby ensures wide browser (and server) support. I encourage you to check out Damian Edwards' and David Fowler's talk Building Real-time Web Apps with ASP.NET SignalR for an introduction to SignalR.

Method #4: Attaching Data to HTML Elements

If you have primitive data that's closely related to an HTML element, it might be best to attach that data to the element using HTML 5 data- attributes:

```
<ul>
    @foreach (var pirate in pirates)
    {
        <li id="@pirate.FirstName" data-rank="@pirate.Rank">@pirate.FullName</li
    }
</ul>
```

Let's assume the following output:

```
<ul>
    <li id="jack" data-rank="captain">Jack Sparrow</li>
    <!-- The rest of the crew is omitted for brevity. Sorry. -->
</ul>
```

Finding out Jack Sparrow's rank is now as simple as that, again using jQuery:

```
var jacksRank = $("#jack").data("rank"); // "captain"
```

While this method of rendering inline-data is great for simple data, it doesn't work well for anything beyond primitive types. This is where #5 will come in handy in just a moment.

Method #5: Assigning Data Directly to a JavaScript Variable

Coming back to primitive values once more, there's also the possibility to assign data to a JavaScript variable, just like this:

```
var blackPearlHomePort = "@Url.Content("~/tortuga")";
```

For simple values like the above one, this is my favorite approach since it neither requires loading an external resource (thus resulting in an additional HTTP request) nor pollutes any HTML elements.

Method #6: Serializing a .NET Object into a JavaScript Literal

Finally, let me show you the approach I like to use when dealing with complex objects. It uses a custom Json.NET serializer to turn .NET objects into their JavaScript literal representation.

Let's assume we have defined the following ASP.NET MVC action ...

```
public ActionResult Index()
{
    var pirates = new List<Person>
    {
        new Person("Jack", "Sparrow"),
        new Person("Will", "Turner"),
        new Person("Elizabeth", "Swann")
    };

    return View(new BlackPearlViewModel(pirates));
}
```

... and view model:

```
public class BlackPearlViewModel
{
    public IEnumerable<Person> Pirates { get; private set; }

    public BlackPearlViewModel(IEnumerable<Person> pirates)
    {
        Pirates = pirates;
    }
}
```

Let's also create a `JavaScriptConvert` class that will serialize arbitrary .NET objects:

```
public static class JavaScriptConvert
{
    public static IHtmlString SerializeObject(object value)
    {
        using (var stringWriter = new StringWriter())
        using (var jsonWriter = new JsonTextWriter(stringWriter))
        {
            var serializer = new JsonSerializer
            {
                // Let's use camelCasing as is common practice in JavaScript
                ContractResolver = new CamelCasePropertyNamesContractResolver()
            };

            // We don't want quotes around object names
            jsonWriter.QuoteName = false;
            serializer.Serialize(jsonWriter, value);

            return new HtmlString(stringWriter.ToString());
        }
    }
}
```

`JavaScriptConvert` almost works like Json.NET's native `JsonConvert` serializer, except that it removes quotes around object names and enforces camelCased property names. The cool thing is that the `CamelCasePropertyNamesContractResolver` is being smart about abbreviations like "ID", which will not be turned into "iD", but into the all-lower "id".

In the corresponding Razor view, we can now use our custom serializer as follows:

```
<script>
    var pirates = @JavaScriptConvert.SerializeObject(Model.Pirates);
</script>
```

For the pirates defined in our above action method, the output will be this:

```
<script>
    var pirates = [{firstName:"Jack",lastName:"Sparrow"},{firstName:"Will",lastN
</script>
```

Note that we can configure the serializer to prettify the output by setting its `Formatting` property to `Formatting.Indented`, which will insert appropriate line breaks and spaces to make the output more readable. Also note that the output is *not* valid JSON because the property names aren't wrapped in quotes.

Conclusion

You have a variety of options how you can move data from your .NET back end to a JavaScript client. Depending on your requirements, you can choose to either load the data by making an additional HTTP request (#1 and #2), connect to your server using SignalR (#3), or embed the data within the document that's being requested (#4, #5, and #6). The big advantage of #4, #5, and #6 is that the JavaScript data is inlined within the HTML document, so you won't have to wait for it to load. As long as you don't put megabytes of data in there, you should be fine.

Hope this helps,

Marius

P.S: Also make sure to **check out my favorite tech-related books**!

Related Articles:

- Generating External JavaScript Files Using Partial Razor Views
- Conditionally Serializing Fields and Properties with Json.NET
- Bootstrapping AngularJS Applications with Server-Side Data from ASP.NET MVC & Razor
- Asynchronously Bootstrapping AngularJS Applications with Server-Side Data

Thanks to Stewart and Julien who pointed out in their comments to include SignalR in this list.