

Levi Botelho's Coding Blog

[Open Source](#) [About Me](#)

Posting JavaScript objects with Ajax and ASP.NET MVC

Nov 24, 2013

3 minute read

As websites become more and more interactive the need frequently arises to send data back and forth between web clients and servers using Ajax. ASP.NET combined with jQuery makes this process simple to implement for web developers.

For this example, we're going to POST the following JavaScript object to our server, and return a Boolean value indicating if the data was correctly received.

```
var tommy = {  
    name: "Tommy",  
    birthday: new Date(1921, 0, 1),  
    hobbies: ["Pinball", "Holiday camp"]  
};
```

The server side

The first thing we need to do on the server side is define a data structure corresponding to the JavaScript object we wish to send.

```
public class Person  
{  
    public string Name { get; set; }  
    public DateTime Birthday { get; set; }  
    public string[] Hobbies { get; set; }  
}
```

Notice how the casing of the variable names in the JavaScript and C# objects doesn't have to match. We can declare our JavaScript object properties in camel case and our C# properties in Pascal case and the model binder will work just fine.

Now let's write our method. All we need to do is create a standard ASP.NET MVC controller method which takes a single parameter of the Person type, like so.

```
public bool ProcessData(Person person)
{
    return person != null;
}
```

As mentioned, I'm returning a Boolean value indicating whether the person object was successfully received.

That's all we need to do on the server side. Let's write the client-side code.

The client side

The first thing we'll do is use jQuery to write the Ajax call.

```
$.ajax({
    url: "@Url.Action("ProcessData", "Home")",
    type: "POST",
    contentType: "application/json",
    data: JSON.stringify({ person: tommy }),
    success: function(response) {
        response ? alert("It worked!") : alert("It didn't work.");
    }
});
```

Here we're invoking the ubiquitous "ajax" method, passing it an object containing all the information it needs to get the job done in the form of field values. Let's take a brief look at each one.

url

The url field contains the dynamically-calculated address of our target method.

type

The type field indicates the type of HTTP request we wish to send. In this case it is a POST request.

contentType

Here we are specifying that the data is of the JSON format. This is important as it provides the server with the knowledge it needs to be able to deserialize our data into a POCO object.

data

The data field contains the actual data that we want to send in the POST request. There are two things of note here. The first is that we have assigned our "tommy" object to a field named "person" in a new anonymous object. Why is this? Well, the JSON object that we send to the server needs to correspond to the signature of the method that will receive the request. As our method looks like this:

```
public bool ProcessData(Person person)
```

We need to send it an object that looks like this:

```
{ person: [our person object] }
```

The second thing to note is that we invoke the `JSON.stringify` method in order to serialise the whole thing into a JSON string. This JSON string is what the server will receive and attempt to deserialise when we make the Ajax call from a web browser. We need to do this because “person” is a complex object and not a primitive JavaScript value, such as a string. Were we to send an object to the server consisting entirely of primitive values, this stringify call would not be necessary.

success

This function executes if the request succeeds. While in this case I’ve used an anonymous function, I could also have externalised this method and set “success” to a function pointer instead.






This is all the server side code we need. Let’s fire up our website and run the code to see what happens.

Making the call

If we insert a breakpoint in our `ProcessData` method and execute our JavaScript, this the result we get in Visual Studio.

0 references

```
public bool ProcessData(Person person)
{
    return person != null;
}
```

 person.Name	value = "Tommy"
 person.Birthday	{1921-01-01 12:00:00 AM}
 person.Hobbies	{string[2]}
 person.Hobbies[0]	value = "Pinball"
 person.Hobbies[1]	value = "Holiday camp"

As you can see, the model binder has done a wonderful job. Not only has it instantiated our object and mapped the simple Name property, but it has also translated the JavaScript Date value into a valid C# DateTime object, and mapped out our string array perfectly. We can now go ahead and do what we like with the data.

Share this post!



