An Introduction to JavaScript Object Notation (JSON) in JavaScript and .NET

Atif Aziz, Scott Mitchell

February 2007

Applies to:
JSON
Ajax

Summary: This article discusses JavaScript Object Notation (or JSON), an open and text-based data exchange format, that provides a standardized data exchange format better suited for Ajax-style web applications. (22 printed pages)

Contents

Introduction
Understanding Literal Notation in JavaScript
Comparing JSON to XML
Creating and Parsing JSON Messages with JavaScript
Working with JSON in the .NET Framework
Conclusion
References

Download the source code for this article.

Introduction

When designing an application that will communicate with a remote computer, a data format and exchange protocol must be selected. There are a variety of open, standardized options, and the ideal choice depends on the applications requirements and pre-existing functionality. For example, SOAP-based web services format the data in an XML payload wrapped within a SOAP envelope.

While XML works well for many application scenarios, it has some drawbacks that make it less than ideal for others. One such space where XML is often less than ideal is with Ajax-style web applications. Ajax is a technique used for building interactive web applications that provide a snappier user experience through the use of out-of-band, lightweight calls to the web server in lieu of full-page postbacks. These asynchronous calls are initiated on the client using JavaScript and involve formatting data, sending it to a web server, and parsing and working with the returned data. While most browsers can construct, send, and parse XML, JavaScript Object Notation (or JSON) provides a standardized data exchange format that is better-suited for Ajax-style web applications.

JSON is an open, text-based data exchange format (see RFC 4627). Like XML, it is human-readable, platform independent, and enjoys a wide availability of implementations. Data formatted according to the JSON standard is lightweight and can be parsed by JavaScript implementations with incredible ease, making it an ideal data exchange format for Ajax web applications. Since it is primarily a data format, JSON is not limited to just Ajax web applications, and can be used in virtually any scenario where applications need to exchange or store structured information as text.

This article examines the JSON standard, its relationship to JavaScript, and how it compares to XML. Jayrock, an open-source JSON implementation for .NET, is discussed and examples of creating and parsing JSON messages are provided in JavaScript and C#.

Understanding Literal Notation in JavaScript

Literals are used in programming languages to *literally* express fixed values, such as the constant integer value of 4, or the string "Hello, World." Literals can be used in most languages wherever an expression is allowed, such as part of a condition in a control statement, an input parameter when calling a function, in variable assignment, and so forth. For example, the following C# and Visual Basic code initializes the variable **x** with the constant integer value of **42**.

```
int x = 42;  // C#Dim x As Integer = 42  ' Visual Basic
```

Different programming languages allow for literals of different types. Most programming languages support, at minimum, literals for scalar types like integers, floating-point numbers, strings, and Boolean. What's interesting about JavaScript is that in addition to scalar types, it also supports literals for structured types like arrays and objects. This feature allows for a terse syntax for on-demand creation and initialization of arrays and objects.

Array literals in JavaScript are composed of zero or more expressions, with each expression representing an element of the array. The array elements are enclosed in square brackets (**[]**) and delimited by commas. The following example defines an array *literally* with seven string elements holding the names of the seven continents:

```
var continents = ["Europe", "Asia", "Australia", "Antarctica", "North America", "South America", "Africa"];alert(continents[0] + " is one
```

Compare this now to how you would create and initialize an array in JavaScript without the literal notation:

```
var continents = new Array();continents[0] = "Europe";continents[1] = "Asia";continents[2] = "Australia";continents[3] = "Antarctica";cont
```

An object literal defines the members of an object and their values. The list of object members and values is enclosed in curly braces (**{}**) and each member is delimited by a comma. Within each member, the name and value are delimited by a colon (**:**). The following example creates an object and initializes it with three members named **Address**, **City**, and **PostalCode** with respective values "**123 Anywhere St.**", "**Springfield**", and "**99999**."

```
var mailingAddress = {      "Address"    :   "123 Anywhere St.",      "City"       :   "Springfield",      "PostalCode" :   99999};alert("
```

The examples presented thus far illustrate using string and numeric literals within array and object literals. You can also express an entire graph by using the notation recursively such that array elements and object member values can themselves, in turn, use object and array literals. For example, the following snippet illustrates an object that has an array as a member (**PhoneNumbers**), where the array is composed of a list of objects.

```
var contact = {     "Name": "John Doe",     "PermissionToCall": true,     "PhoneNumbers": [       {           "Location": "Home",
```

> **Note** A more thorough discussion of literal support for JavaScript can be found in the Core JavaScript 1.5 Guide under the Literals section.

From JavaScript Literals to JSON

JSON is a data exchange format that was created from a subset of the literal object notation in JavaScript. While the syntax accepted by JavaScript for literal values is very flexible, it is important to note that JSON has much stricter rules. According to the JSON standard, for example, the name of an object member *must* be a valid JSON string. A string in JSON *must* be enclosed in quotation marks. JavaScript, on the other hand, allows object member names to be delimited by quotation marks or apostrophes or to omit quoting altogether so long as the member name doesn't conflict with a reserved JavaScript keyword. Likewise, an array element or an object member value in JSON is limited to a very limited set. In JavaScript, however, array elements and object member values can refer to pretty much any valid JavaScript expression, including function calls and definitions!

The charm of JSON is in its simplicity. A message formatted according to the JSON standard is composed of a single top-level object or array. The array elements and object values can be objects, arrays, strings, numbers, Boolean values (true and false), or null. That, in a nutshell, is the JSON standard! It's really that simple. See www.json.org or RFC 4627 for a more formal description of the standard.

One of the sore points of JSON is the lack of a date/time literal. Many people are surprised and disappointed to learn this when they first encounter JSON. The simple explanation (consoling or not) for the absence of a date/time literal is that JavaScript never had one either: The support for date and time values in JavaScript is entirely provided through the Date object. Most applications using JSON as a data format, therefore, generally tend to use either a string or a number to express date and time values. If a string is used, you can generally expect it to be in the ISO 8601 format. If a number is used, instead, then the value is usually taken to mean the number of milliseconds in Universal Coordinated Time (UTC) since epoch, where epoch is defined as midnight January 1, 1970 (UTC). Again, this is a mere convention and not part of the JSON standard. If you are exchanging data with another application, you will need to check its documentation to see how it encodes date and time values within a JSON literal. For example, Microsoft's ASP.NET AJAX uses neither of the described conventions. Rather, it encodes .NET DateTime values as a JSON string, where the content of the string is **\/Date(ticks)\/** and where *ticks* represents milliseconds since epoch (UTC). So November 29, 1989, 4:55:30 AM, in UTC is encoded as "**\/Date(628318530718)\/**". For some rationale behind this rather contrived choice of encoding, see "Inside ASP.NET AJAX's JSON date and time string."

Comparing JSON to XML

Both JSON and XML can be used to represent native, in-memory objects in a text-based, human-readable, data exchange format. Furthermore, the two data exchange formats are isomorphic—given text in one format, an equivalent one is conceivable in the other. For example, when calling one of Yahoo!'s publicly accessible web services, you can indicate via a querystring parameter whether the response should be formatted as XML or JSON. Therefore, when deciding upon a data exchange format, it's not a simple matter of choosing one over the other as a silver bullet, but rather what format has the *characteristics* that make it the best choice for a particular application. For example, XML has its roots in marking-up document text and tends to shine very well in that space (as is evident with XHTML). JSON, on the other hand, has its roots in programming language types and structures and therefore provides a more natural and readily available mapping to exchange structured data. Beyond these two starting points, the following table will help you to understand and compare the key characteristics of XML and JSON.

Key Characteristic Differences between XML and JSON

Characteristic	XML	JSON
Data types	Does not provide any notion of data types. One must rely on XML Schema for adding type information.	Provides scalar data types and the ability to express structured data through arrays and objects.
Support for arrays	Arrays have to be expressed by conventions, for example through the use of an outer placeholder element that models the arrays contents as inner elements. Typically, the outer element uses the plural form of the name used for inner elements.	Native array support.
Support for objects	Objects have to be expressed by conventions, often through a mixed use of attributes and elements.	Native object support.
Null support	Requires use of **xsi:nil** on elements in an XML instance document plus an import of the corresponding namespace.	Natively recognizes the **null** value.
Comments	Native support and usually available through APIs.	Not supported.
Namespaces	Supports namespaces, which eliminates the risk of name collisions when combining documents. Namespaces also allow existing XML-based standards to be safely extended.	No concept of namespaces. Naming collisions are usually avoided by nesting objects or using a prefix in an object member name (the former is preferred in practice).
Formatting decisions	Complex. Requires a greater effort to decide how to map application types to XML elements and attributes. Can create heated debates whether an element-centric or attribute-centric approach is better.	Simple. Provides a much more direct mapping for application data. The only exception may be the absence of date/time literal.
Size	Documents tend to be lengthy in size, especially when an element-centric approach to formatting is used.	Syntax is very terse and yields formatted text where most of the space is consumed (rightly so) by the represented data.
Parsing in JavaScript	Requires an XML DOM implementation and additional application code to map text back into JavaScript objects.	No additional application code required to parse text; can use JavaScript's **eval** function.
Learning curve	Generally tends to require use of several technologies in concert: XPath, XML Schema, XSLT, XML Namespaces, the DOM, and so on.	Very simple technology stack that is already familiar to developers with a background in JavaScript or other dynamic programming languages.

JSON is a relatively new data exchange format and does not have the years of adoption or vendor support that XML enjoys today (although JSON is catching up quickly). The following table highlights the current state of affairs in the XML and JSON spaces.

Support Differences between XML and JSON

Support	XML	JSON
Tools	Enjoys a mature set of tools widely available from many industry vendors.	Rich tool support—such as editors and formatters—is scarce.
Microsoft .NET Framework	Very good and mature support since version 1.0 of the .NET Framework. XML support is available as part of the Base Class Library (BCL). For unmanaged environments, there is MSXML.	None so far, except an initial implementation as part of ASP.NET AJAX.
Platform and language	Parsers and formatters are widely available on many platforms and languages (commercial and open source implementations).	Parsers and formatters are available already on many platforms and in many languages. Consult json.org for a good set of references. Most implementations for now tend to be open source projects.
Integrated language	Industry vendors are currently experimenting with support *literally* within languages. See Microsoft's LINQ project for more information.	Is natively supported in JavaScript/ECMAScript only.

Note Neither table is meant to be a comprehensive list of comparison points. There are further angles on which both data formats can be compared, but we felt that these key points should be sufficient to build an initial impression.

Creating and Parsing JSON Messages with JavaScript

When using JSON as the data exchange format, two common tasks are turning a native and in-memory representation into its JSON text representation and vice versa. Unfortunately, at the time of writing, JavaScript does not provide built-in functions to create JSON text from a given object or array. These methods are expected to be included in the fourth edition of the ECMAScript standard in 2007. Until these JSON formatting functions are formally added to JavaScript and widely available across popular implementations, use the reference implementation script available for download at http://www.json.org/json.js.

In its latest iteration at the time of this writing, the json.js script at www.json.org adds **toJSONString()** functions to array, string, Boolean, object, and other JavaScript types. The **toJSONString()** functions for scalar types (like Number and Boolean) are quite simple since they only need to return a string representation of the instance value. The **toJSONString()** function for the **Boolean** type, for example, returns the string "true" if the value is true, and "false" otherwise. The **toJSONString()** functions for Array and Object types are more interesting. For Array instances, the **toJSONString()** function for each contained element is called in sequence, with the results being concatenated with commas to delimit each result. The final output enclosed in square brackets. Likewise, for Object instances, each member is enumerated and its **toJSONString()** function invoked. The member name and the JSON representation of its value are concatenated with a colon in the middle; each member name and value pair is delimited with a comma and the entire output is enclosed in curly brackets.

The net result of the **toJSONString()** functions is that any type can be converted into its JSON format with a single function call. The following JavaScript creates an Array object and adds seven String elements deliberately using the verbose and non-literal method for illustrative purposes. It then goes on to displays the arrays JSON representation:

```
// josn.js must be included prior to this pointvar continents = new Array();continents.push("Europe");continents.push("Asia");continents.p
```

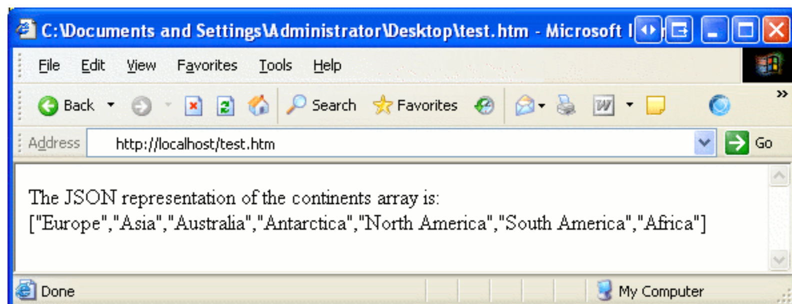

Figure 1. The toJSONString() function emits the array formatted according to the JSON standard.

Parsing JSON text is even simpler. Since JSON is merely a subset of JavaScript literals, it can be parsed into an in-memory representation using the eval(expr) function, treating the source JSON text as JavaScript source code. The **eval** function accepts as input a string of valid JavaScript code and evaluates the expression. Consequently, the following single line of code is all that is needed to turn JSON text into a native representation:

```
var value = eval( "(" + jsonText + ")" );
```

> **Note** The extra parentheses are used make **eval** unconditionally treat the source input like an expression. This is especially important for objects. If you try to call eval with a string containing JSON text that defines an object, such as the string "**{}**" (meaning an empty object), then it simply returns undefined as the parsed result. The parentheses force the JavaScript parser to see the top-level curly braces as the literal notation for an Object instance rather than, say, curly braces defining a statement block. Incidentally, the same problem does not occur if the top-level item is an array, as in **eval("[1,2,3]")**. For sake of uniformity, however, JSON text should always be surrounded with parentheses prior to calling **eval** so that there is no ambiguity about how to interpret the source.

When evaluating literal notation, an instance corresponding to the literal syntax is returned and assigned to **value**. Consider the following example, which uses the **eval** function to parse the literal notation for an array and assigning the resulting array to the variable **continents**.

```
var arrayAsJSONText = '["Europe", "Asia", "Australia", "Antarctica", "North America", "South America", "Africa"]';var continents = eval( a
```

Of course, in practice the evaluated JSON text will come from some external source rather than being hard-coded as in the above case.

The **eval** function blindly evaluates whatever expression it is passed. An untrustworthy source could therefore include potentially dangerous JavaScript along with or mixed into the literal notation that makes up the JSON data. In scenarios where the source cannot be trusted, it is highly recommended that you parse the JSON text using the **parseJSON()** function (found in json.js):

```
// Requires json.jsvar continents = arrayAsJSONText.parseJSON();
```

The **parseJSON()** function also uses **eval**, but only if the string contained in **arrayAsJSONText** conforms to the JSON text standard. It does this using a clever regular expression test.

Working with JSON in the .NET Framework

JSON text can easily be created and parsed from JavaScript code, which is part of its allure. However, when JSON is used in an ASP.NET web application, only the browser enjoys JavaScript support since the server-side code is most likely written in Visual Basic or C#.

Most Ajax libraries designed for ASP.NET provide support for programmatically creating and parsing JSON text. Therefore, to work with JSON in a .NET application, consider using one of these libraries. There are plenty of open-source and third-party options, and Microsoft also has their own Ajax library named ASP.NET AJAX.

In this article we will look at examples that use Jayrock, an open-source implementation of JSON for the Microsoft .NET Framework created by coauthor Atif Aziz. We chose to use Jayrock instead of ASP.NET AJAX for three reasons:

- Jayrock is open-source, making it possible to extend or customize as needed.
- Jayrock can be used in ASP.NET 1.x, 2.0, and Mono applications, whereas ASP.NET AJAX is for ASP.NET version 2.0 only.
- Jayrock's scope is limited to JSON and JSON-RPC, and the former is the main focus of this article. While ASP.NET AJAX includes some support for creating and parsing JSON text, its primary purpose is to offer a rich platform for building end-to-end Ajax-style web applications in ASP.NET. The extra bells and whistles can be distracting when your main focus is JSON.

Working with JSON in .NET using Jayrock is similar to working with XML through the XmlWriter, XmlReader, and XmlSerializer classes in the .NET Framework. The classes **JsonWriter**, **JsonReader**, **JsonTextWriter**, and **JsonTextReader** found in Jayrock mimic the semantics of the .NET Framework classes **XmlWriter**, **XmlReader**, **XmlTextWriter**, and **XmlTextReader**. These classes are useful for interfacing with JSON at a low- and stream-oriented level. Using these classes, JSON text can be created or parsed piecemeal through a series of method calls. For example, using the **JsonWriter** class method **WriteNumber(number)** writes out the appropriate string representation of *number* according to the JSON standard. The **JsonConvert** class offers **Export** and **Import** methods for converting between .NET types and JSON. These methods provide a similar functionality as found in the **XmlSerializer** class methods **Serialize** and **Deserialize**, respectively.

Creating JSON Text

The following code illustrates using the **JsonTextWriter** class to create the JSON text for a string array of continents. This JSON text is sent to a **TextWriter** instance passed into the

constructor, which happens to be the output stream from the console in this example (in ASP.NET you can use **Response.Output** instead):

```
using (JsonTextWriter writer = JsonTextWriter(Console.Out)){     writer.WriteStartArray();     writer.WriteString("Europe");     writer.Write
```

In addition to the **WriteStartArray**, **WriteString**, and **WriteEndArray** methods, the **JsonWriter** class provides methods for writing other JSON value types, such as **WriteNumber**, **WriteBoolean**, **WriteNull**, and so on. The **WriteStartObject**, **WriteEndObject**, and **WriteMember** methods create the JSON text for an object. The following example illustrates creating the JSON text for the contact object examined in the "Understanding Literal Notation in JavaScript" section:

```
private static void WriteContact(){     using (JsonWriter w = new JsonTextWriter(Console.Out))     {         w.WriteStartObject();
```

Export and **ExportToString** methods in the **JsonConvert** class can be used to serialize a specified .NET type into JSON text. For example, rather than manually building the JSON text for the array of the seven continents using the **JsonTextWriter** class, the following call to **JsonConvert.ExportToString** produces the same results:

```
string[] continents = {       "Europe", "Asia", "Australia", "Antarctica", "North America",       "South America", "Africa"};string jsonTex
```

Parsing JSON Text

The **JsonTextReader** class provides a variety of methods to parse the tokens of JSON text with the core one being **Read**. Each time the **Read** method is invoked, the parser consumes the next token, which could be a string value, a number value, an object member name, the start of an array, and so forth. Where applicable, the parsed text of the current token can be accessed via the **Text** property. For example, if the reader is sitting on Boolean data, then the **Text** property will return "true" or "false" depending on the actual parse value.

The following sample code uses the **JsonTextReader** class to parse through the JSON text representation of a string array containing the names of the seven continents. Each continent that begins with the letter "A" is sent to the console:

```
string jsonText = @"[""Europe"", ""Asia"", ""Australia"", ""Antarctica"", ""North America"", ""South America"", ""Africa""]";using (JsonTe
```

> **Note** The **JsonTextReader** class in Jayrock is a fairly liberal JSON text parser. It actually permits a lot more syntax than is considered valid JSON text according to rules laid out in RFC 4627. For example, the **JsonTextReader** class allows single-line and multi-line comments to appear within JSON text as you'd expect in JavaScript. Single-line comments start with slash-slash (**//**) and multi-line comments being with slash-star (**/***) and end in star-slash (***/**). Single-line comments can even begin with the hash/pound sign (**#**), which is common among Unix-style configuration files. In all instances, the comments are completely skipped by the parser and never exposed through the API. Also as in JavaScript, **JsonTextReader** permits a JSON string to be delimited by an apostrophe ('). The parser can even tolerate an extra comma after the last member of an object or element of an array.
>
> Even with all these additions, **JsonTextReader** is a conforming parser! **JsonTextWriter**, on the other hand, produces only strict standard-conforming JSON text. This follows what is often coined as the robustness principal, which states, "Be conservative in what you do; be liberal in what you accept from others."

To convert JSON text directly into a .NET object, use the **JsonConvert** class import method, specifying the output type and JSON text. The following example shows conversion of a JSON array of strings into a .NET string array:

```
string jsonText = @"[""Europe"", ""Asia"", ""Australia"", ""Antarctica"", ""North America"", ""South America"", ""Africa""]";string[] cont
```

Here is a more interesting example of conversion that takes an RSS XML feed, deserializes it into a .NET type using **XmlSerializer**, and then converts the object into JSON text using **JsonConvert** (effectively converting RSS in XML to JSON text):

```
XmlSerializer serializer = new XmlSerializer(typeof(RichSiteSummary));RichSiteSummary news;// Get the MSDN RSS feed and deserialize it...u
```

> **Note** The definition of **RichSiteSummary** and its related types can be found in the samples accompanying this article.

Using JSON in ASP.NET

Having looked at ways to work with JSON in JavaScript and from within the .NET Framework using Jayrock, it's time to turn to a practical example of where and how all this knowledge can be applied. Consider the client script callback feature in ASP.NET 2.0, which simplifies the process of making out-of-band calls from the web browser to the ASP.NET page (or to a particular control on the page). During a typical callback scenario, the client-side script in the browser packages and sends data back to the web server for some processing by a server-side method. After receiving the response data from the server, the client then uses it to update the browser display.

> **Note** More information can be found in the MSDN Magazine article Script Callbacks in ASP.NET 2.0.

The challenge in a client callback scenario is that the client and server can only ship a string back and forth. Therefore, the information to be exchanged must be converted from a native, in-memory representation to a string before being sent and then parsed from a string back to its native, in-memory representation when received. The client script callback feature in ASP.NET 2.0 does not require a particular string format for the exchanged data, nor does it provide any built-in functionality for converting between the native in-memory and string representations; it is up to the developer to implement the conversion logic based on some data exchange format of his or her choice.

The following example illustrates how to use JSON as the data exchange format in a client script callback scenario. In particular, the example consists of an ASP.NET page that uses data from the Northwind database to provide a listing of the categories in a drop-down list; products in the selected category are displayed in a bulleted list (see Figure 3). Whenever the drop-down list is changed on the client side, a callback is made passing in an array whose single element is the selected **CategoryID**.

> **Note** We are passing in an array that contains the selected **CategoryID** as its sole element (rather than just the **CategoryID**) because the JSON standard requires that any JSON text must have an object or an array as its root. Of course, the client is not required to pass JSON text to the server—we could have had this example pass just the selected **CategoryID** as a string. However, we wanted to demonstrate sending JSON text in both the request and response messages of the callback.

The following code in the **Page_Load** event handler configures the **Categories DropDownList** Web control so that when it is changed, the **GetProductsForCategory** function is called and passed the selected drop-down lists value. This function initiates the client script callback if the passed-in drop-down list value is greater than zero:

```
// Add client-side onchange event to drop-down listCategories.Attributes["onchange"] = "Categories_onchange(this);";// Generate the callba
```

The GetCallBackEventReference method in the ClientScriptManager class, which is used to generate the JavaScript code that invokes the callback, has the following signature:

```
public string GetCallbackEventReference (     Control control,     string argument,     string clientCallback,     string context,)
```

The **argument parameter** specifies what data is sent from the client to the web server during the callback and the **clientCallback** parameter specifies the name of the client-side function to invoke upon completion of the callback (**showProducts**). The **GetCallBackEventReference** method call generates the following JavaScript code and adds it to the rendered markup:

```
WebForm_DoCallback('__Page','[' + categoryID + ']',showProducts,null,null,false)
```

'[' + categoryID + ']' is the value that is passed to the server during the callback (an array with a single element, **categoryID**) and **showProducts** is the JavaScript function that is executed when the callback returns.

On the server side, the method that is executed in response to the callback uses **JsonConvert** class from Jayrock to parse the incoming JSON text and format the outgoing JSON text. In particular, the names of the products that are associated with the selected category are retrieved and returned as a string array.

```
// Deserialize the JSON text into an array of integersint[] args = (int[]) JsonConvert.Import(typeof(int[]), eventArgument);// Read the se
```

> **Note** The **JsonConvert** class is used twice—once to convert the JSON text in **eventArgument** into an array of integers and then to convert the string array **productNames**

into JSON text to return to the client. Alternatively, we could have used the **JsonReader** and **JsonWriter** classes here, but **JsonConvert** does the same job fairly well when the data involved is relatively small and easily mapped to existing types.

When the data is returned from the server-side, the JavaScript function specified from the **GetCallBackEventReference** method is called and passed the return value. This JavaScript method, **showProducts**, starts by referencing the **<div>** element **ProductOutput**. It then parses the JSON response and dynamically adds an unordered list with a list item for each array element. If no products are returned for the selected category, then a corresponding message is displayed instead.

```
function showProducts(arg, context){    // Dump the JSON text response from the server.    document.forms[0].JSONResponse.value = arg;
```

Figure 2 illustrates the sequence of events while Figure 3 shows this example in action; the complete code is included in this articles download.

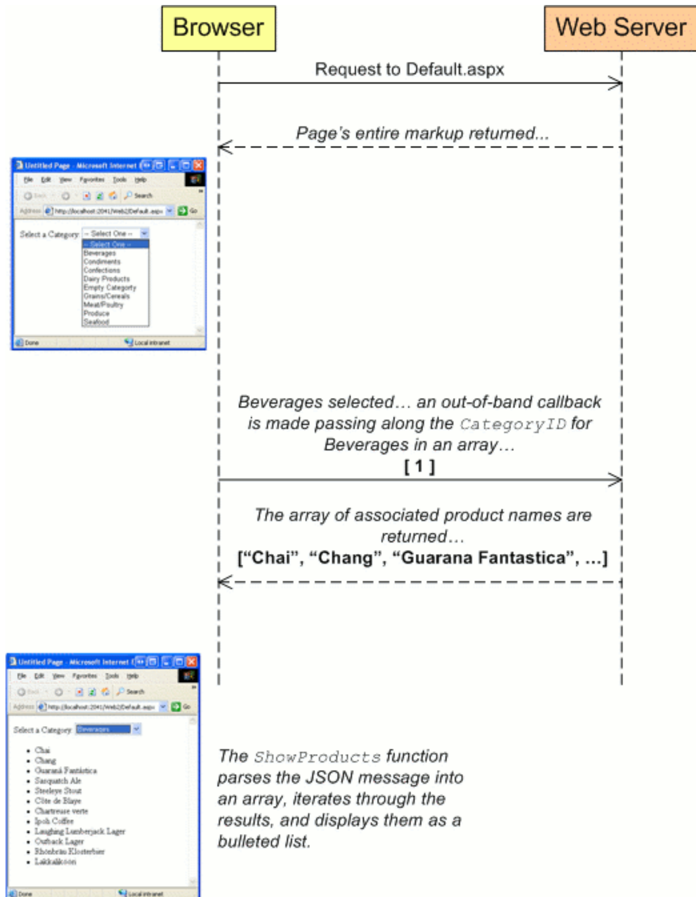

Figure 2: The client sends the selected CategoryID as the single element in an array and the server returns an array of associated product names.

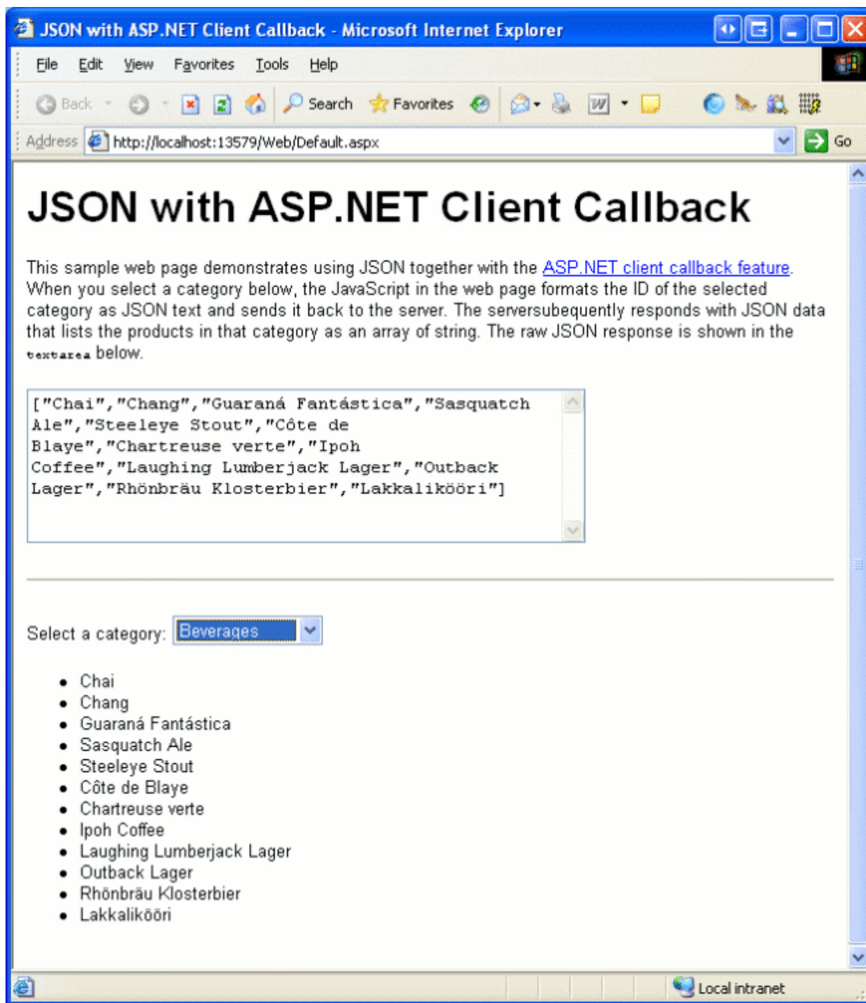

Figure 3: The products are displayed in a bulleted list inside the selected category.

Conclusion

JSON is a lightweight, text-based data exchange format based on a subset of the literal notation from the JavaScript programming language. It provides a succinct encoding for application data structures and is typically used in scenarios where a JavaScript implementation is available to one or both of the applications exchanging data, such as in Ajax-style web applications. The allure of JSON lies in its simplicity to understand, adopt, and implement. JSON has virtually no learning curve for developers already familiar with JavaScript or other programming languages with similar support for a rich literal notation (like Python and Ruby). Parsing JSON text in JavaScript code can be accomplished by simply calling the **eval** function, and creating JSON text is a breeze with the json.js script provided at http://www.json.org/json.js.

There are a blossoming number of libraries for working with JSON across all major platforms and frameworks. In this article we looked at Jayrock, an open-source library for creating and parsing JSON text in .NET applications. Jayrock can be used in ASP.NET 1.x, 2.0, and Mono applications. ASP.NET AJAX offers similar JSON functionality, but for ASP.NET 2.0 applications only.

Happy Programming!

References

- ASP.NET AJAX
- Client-Side Web Service Calls with AJAX Extensions
- Core JavaScript 1.5 Guide
- eval(expr) function
- Jayrock
- JSON.org
- Script Callbacks in ASP.NET
- RFC 4627

Ajax or AJAX?

The term Ajax was initially coined by Jesse James Garrett to describe the style of web applications and set of technologies involved in making highly interactive web applications. Historically, the term Ajax spread around the web as the acronym AJAX, meaning Asynchronous JavaScript And XML. With time, however, people realized that the "X" in AJAX was not very representative of the underlying data format used to communicate with the web server in the background since most implementations were switching to JSON as a simpler and more efficient alternative. So rather than coming up with a replacement acronym like AJAJ that's a bit of tongue-twister, the acronym is generally being retired in favor of Ajax the term rather than AJAX the acronym.

At the time of this writing, expect to see a mixed and wide use of "AJAX" and "Ajax" to mean one and the same thing. In this article, we've stuck with "Ajax the term." Commercial products that provide frameworks enabling Ajax-style applications, however, tend to use the acronym form to distinguish from a similarly named cleaning agent product and to avoid any potential trademark or legal disputes.

ASP.NET AJAX: Inside JSON date and time string

The AJAX JSON serializer in ASP.NET encodes a **DateTime** instance as a JSON string. During its pre-release cycles, ASP.NET AJAX used the format "**@ticks@**", where ticks

represents the number of milliseconds since January 1, 1970 in Universal Coordinated Time (UTC). A date and time in UTC like November 29, 1989, 4:55:30 AM would be written out as "**@62831853071@**." Although simple and straightforward, this format cannot differentiate between a serialized date and time value and a string that looks like a serialized date but is not meant to be deserialized as one. Consequently, the ASP.NET AJAX team made a change for the final release to address this problem by adopting the "**\/Date(ticks)\/**" format.

The new format relies on a small trick to reduce the chance for misinterpretation. In JSON, a forward-slash (**/**) character in a string can be escaped with a backslash (****) even though it is not strictly required. Taking advantage of this, the ASP.NET AJAX team modified JavaScriptSerializer to write a **DateTime** instance as the string "**\/Date(ticks)\/**" instead. The escaping of the two forward-slashes is superficial, but significant to JavaScriptSerializer. By JSON rules,
"
\/Date(ticks)\/
"
is technically equivalent to
"
/Date(ticks)/
"
but the JavaScriptSerializer will deserialize the former as a **DateTime** and the latter as a **String**. The chances for ambiguity are therefore considerably less when compared to the simpler "**@ticks@**" format from the pre-releases.