INTRODUCING JSON

John Sadd
Fellow and OpenEdge Evangelist
Document Version 1.0
December 2010

Extending Your OpenEdge Application with an RIA User Interface

Introducing JSON

BUSINESS MAKING PROGRESS

John Sadd

Progress OpenEdge

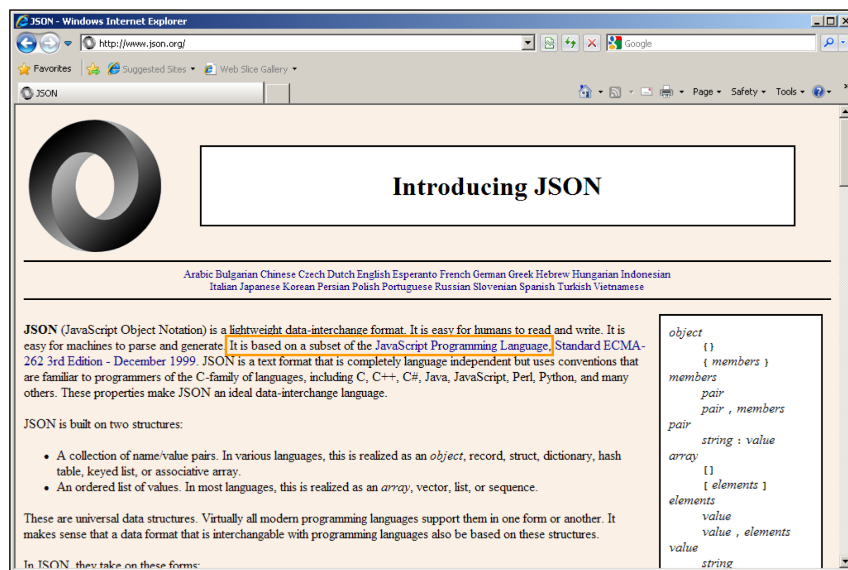
PROGRESS SOFTWARE

PROGRESS SOFTWARE

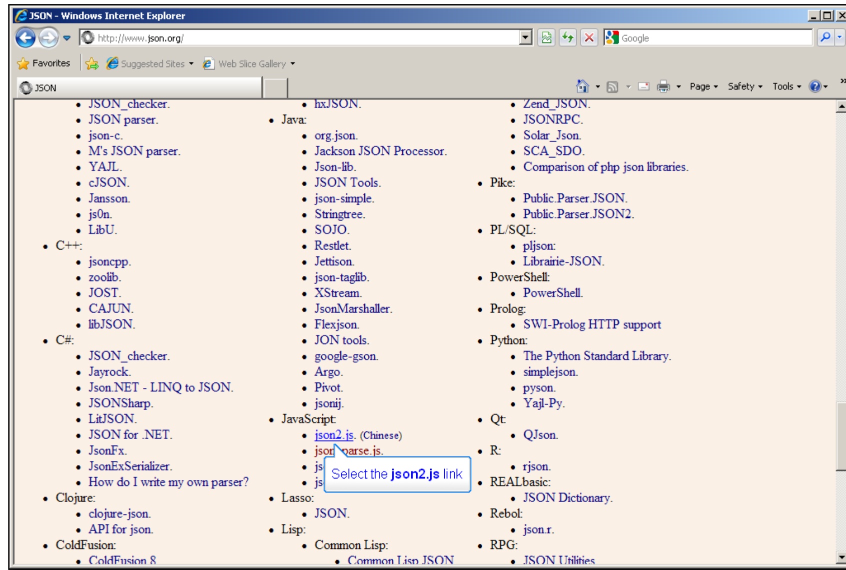
DISCLAIMER

Certain portions of this document contain information about Progress Software Corporation's plans for future product development and overall business strategies. Such information is proprietary and confidential to Progress Software Corporation and may be used by you solely in accordance with the terms and conditions specified in the PSDN Online (http://www.psdn.com) Terms of Use (http://psdn.progress.com/terms/index.ssp). Progress Software Corporation reserves the right, in its sole discretion, to modify or abandon without notice any of the plans described herein pertaining to future development and/or business development strategies. Any reference to third party software and/or features is intended for illustration purposes only. Progress Software Corporation does not endorse or sponsor such third parties or software.

This paper accompanies another in a series of presentations on extending your OpenEdge application by creating a rich browser-based user interface. Another two-part session introduces AJAX, which stands for Asynchronous JavaScript and XML. Along the way I pointed out that XML is not the only data representation you can use with AJAX, and in this session I introduce you to the other principal format for passing complex data across the wire as a string, called JSON. JSON stands for JavaScript Object Notation, and you can learn about it at the website **www.json.org**:



The significant statement here is that JSON is a subset of the JavaScript Programming language, which determines how its syntax is defined, but in fact it can be used with just about any programming language. The website provides you with a complete definition of its syntax, which is quite straightforward. The phrase Object Notation means that it's a notation, that is, a simple string representation, of what JavaScript understands as an object, a representation of a complex set of data as a hierarchy of names and values. In the next screenshot you can see here some of the many links to various code libraries that support the creation and parsing of JSON in many different programming languages.



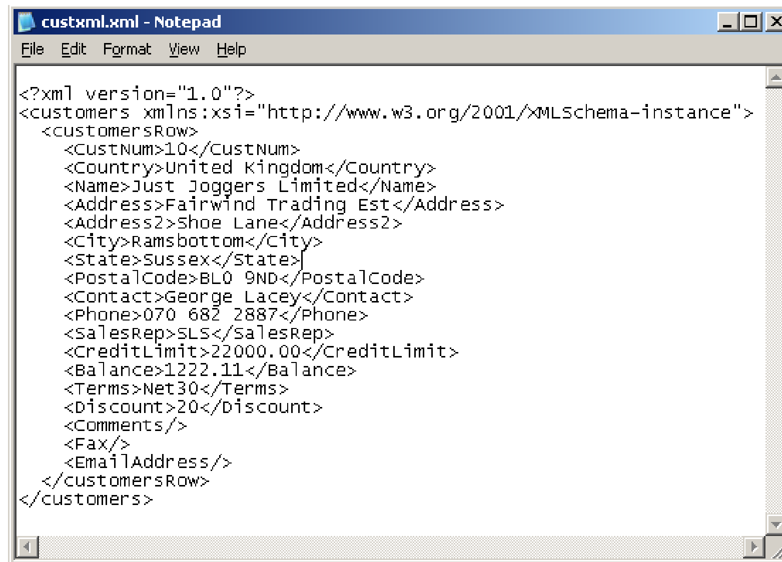
Because JSON is, as it were, native to JavaScript, and because it's instructive to show an AJAX example that uses JSON, you can take a quick look at how JSON is represented and accessed in JavaScript.

This next page shows you an example of JSON syntax.

An object is defined by curly braces. An array of objects is defined by square brackets. A data member name is always in quotation marks, followed by a colon. Its value is also in quotation marks, unless it's a number or a boolean value, or the value null; and multiple members of an object are separated by commas. As in XML, because the notation is simply a character string, every data value is represented as a string, including numbers, and the values true and false for booleans. There's no

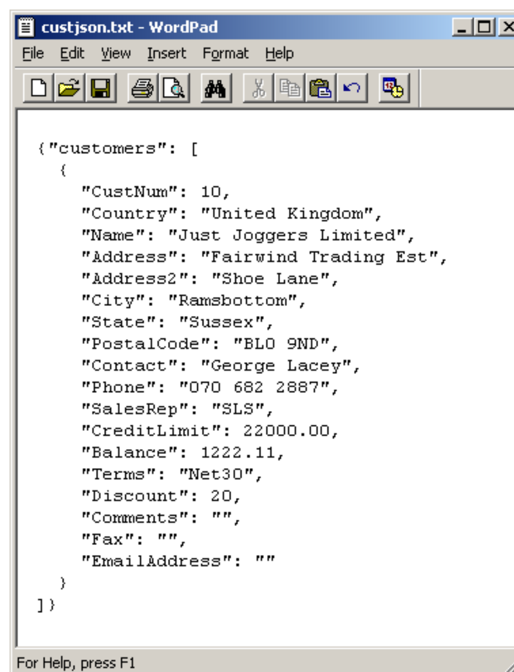
official date representation, so languages need to provide a way to convert a string to a date. And that's about all there is to it.

There are two major differences from XML. In JSON, all names and character values are quoted, but there's no need for a closing tag repeating the name of every member, as there is for elements in XML, so typically JSON is a good deal more compact than XML, which can be a factor in which representation you choose when you're sending data. The following representations of a row in the Customer table in XML and in JSON illustrate the potential different in the length of the data representation in those two formats. Here is the XML:



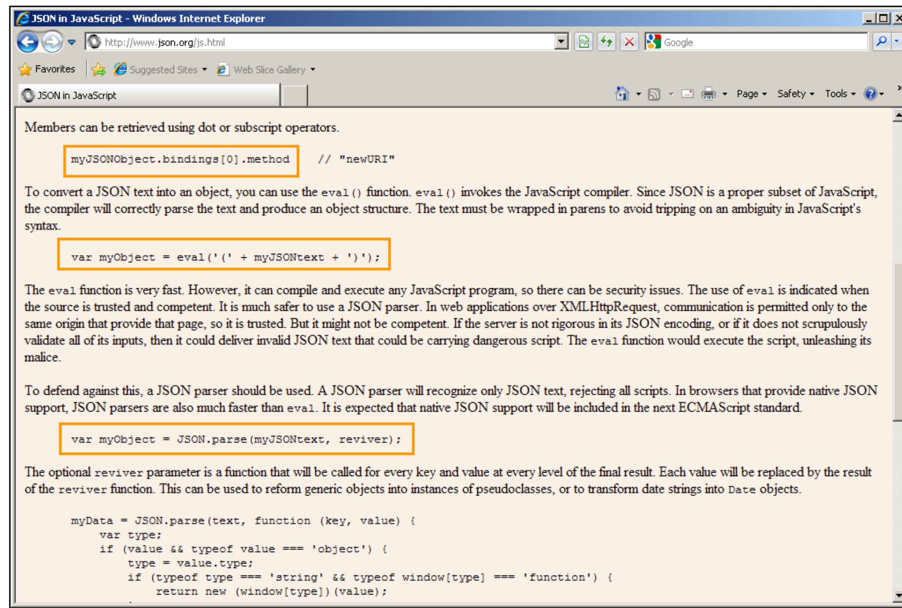

```
<?xml version="1.0"?>
<customers xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <customersRow>
    <CustNum>10</CustNum>
    <Country>United Kingdom</Country>
    <Name>Just Joggers Limited</Name>
    <Address>Fairwind Trading Est</Address>
    <Address2>Shoe Lane</Address2>
    <City>Ramsbottom</City>
    <State>Sussex</State>
    <PostalCode>BL0 9ND</PostalCode>
    <Contact>George Lacey</Contact>
    <Phone>070 682 2887</Phone>
    <SalesRep>SLS</SalesRep>
    <CreditLimit>22000.00</CreditLimit>
    <Balance>1222.11</Balance>
    <Terms>Net30</Terms>
    <Discount>20</Discount>
    <Comments/>
    <Fax/>
    <EmailAddress/>
  </customersRow>
</customers>
```


And here is the JSON:




```
{"customers": [
  {
    "CustNum": 10,
    "Country": "United Kingdom",
    "Name": "Just Joggers Limited",
    "Address": "Fairwind Trading Est",
    "Address2": "Shoe Lane",
    "City": "Ramsbottom",
    "State": "Sussex",
    "PostalCode": "BL0 9ND",
    "Contact": "George Lacey",
    "Phone": "070 682 2887",
    "SalesRep": "SLS",
    "CreditLimit": 22000.00,
    "Balance": 1222.11,
    "Terms": "Net30",
    "Discount": 20,
    "Comments": "",
    "Fax": "",
    "EmailAddress": ""
  }
]}
```

In the same web page, you can see how invididual values in a JSON data object are accessed from JavaScript, using a dot between levels of a data hierarchy, and bracketed subscripts to identify an element of an array. This is instead of using the DOM to extract values from XML.


JSON in JavaScript - Windows Internet Explorer

http://www.json.org/js.html

Members can be retrieved using dot or subscript operators.

```
myJSONObject.bindings[0].method    // "newURI"
```

To convert a JSON text into an object, you can use the `eval()` function. `eval()` invokes the JavaScript compiler. Since JSON is a proper subset of JavaScript, the compiler will correctly parse the text and produce an object structure. The text must be wrapped in parens to avoid tripping on an ambiguity in JavaScript's syntax.

```
var myObject = eval('(' + myJSONtext + ')');
```

The `eval` function is very fast. However, it can compile and execute any JavaScript program, so there can be security issues. The use of `eval` is indicated when the source is trusted and competent. It is much safer to use a JSON parser. In web applications over XMLHttpRequest, communication is permitted only to the same origin that provide that page, so it is trusted. But it might not be competent. If the server is not rigorous in its JSON encoding, or if it does not scrupulously validate all of its inputs, then it could deliver invalid JSON text that could be carrying dangerous script. The `eval` function would execute the script, unleashing its malice.

To defend against this, a JSON parser should be used. A JSON parser will recognize only JSON text, rejecting all scripts. In browsers that provide native JSON support, JSON parsers are also much faster than `eval`. It is expected that native JSON support will be included in the next ECMAScript standard.

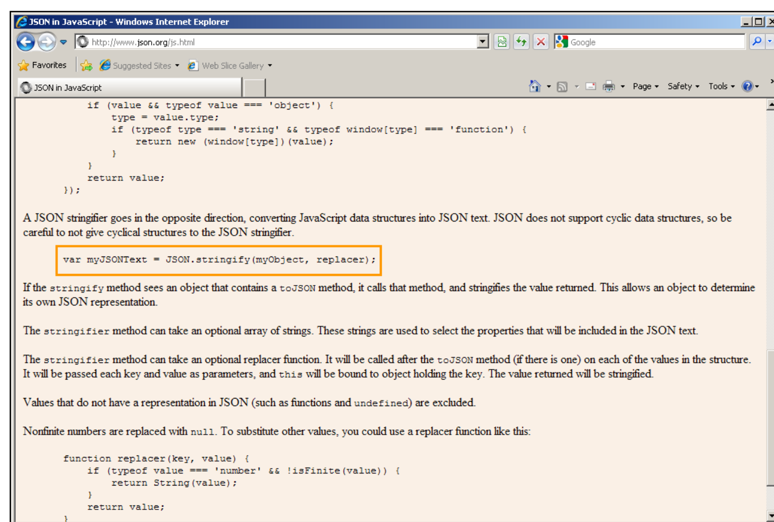
```
var myObject = JSON.parse(myJSONtext, reviver);
```

The optional `reviver` parameter is a function that will be called for every key and value at every level of the final result. Each value will be replaced by the result of the `reviver` function. This can be used to reform generic objects into instances of pseudoclasses, or to transform date strings into `Date` objects.

```
myData = JSON.parse(text, function (key, value) {
    var type;
    if (value && typeof value === 'object') {
        type = value.type;
        if (typeof type === 'string' && typeof window[type] === 'function') {
            return new (window[type])(value);
```

The next point the documentation makes here is that because JSON is a subset of JavaScript, you can use the standard JavaScript eval method to parse it. But that can be dangerous, as the supposed JSON data could contain badly formed constructs or even malicious code that your application would then execute. So JavaScript provides an intrinsic JSON object with two important methods. The first is a **parse** method, which turns a JSON string into a proper JavaScript object. The optional **reviver** function can be used, for instance, to turn date strings into proper date values.

The second basic method is **stringify**, which does what the name implies, turning a JavaScript object into a character string which can then be passed across the wire:


JSON in JavaScript - Windows Internet Explorer

http://www.json.org/js.html

```
        if (value && typeof value === 'object') {
            type = value.type;
            if (typeof type === 'string' && typeof window[type] === 'function') {
                return new (window[type])(value);
            }
        }
        return value;
    });
```

A JSON stringifier goes in the opposite direction, converting JavaScript data structures into JSON text. JSON does not support cyclic data structures, so be careful to not give cyclical structures to the JSON stringifier.

```
var myJSONText = JSON.stringify(myObject, replacer);
```

If the `stringify` method sees an object that contains a `toJSON` method, it calls that method, and stringifies the value returned. This allows an object to determine its own JSON representation.

The `stringifier` method can take an optional array of strings. These strings are used to select the properties that will be included in the JSON text.

The `stringifier` method can take an optional replacer function. It will be called after the `toJSON` method (if there is one) on each of the values in the structure. It will be passed each key and value as parameters, and `this` will be bound to object holding the key. The value returned will be stringified.

Values that do not have a representation in JSON (such as functions and `undefined`) are excluded.

Nonfinite numbers are replaced with `null`. To substitute other values, you could use a replacer function like this:

```
function replacer(key, value) {
    if (typeof value === 'number' && !isFinite(value)) {
        return String(value);
    }
    return value;
}
```

Now take a look at how the AJAX example used in earlier videos can be adapted to use JSON instead. Here is the ABL procedure used in the AJAX presentation:

```
/* getcustomers_param.p
   Server-side application to run to test Ajax / WebSpeed programs
     AjaxClientWebParam.html and
     AjaxCustWebParam.html
*/

{src/web2/wrap-cgi.i}

output-content-type ("text/xml").

define temp-table ttCust like Sports2000.Customer.

temp-table ttcust:serialize-name = "customers".

find Sports2000.Customer where Customer.CustNum =
    INTEGER(get-value("piCustNum")).
create ttcust.
buffer-copy Customer to ttcust.

temp-table ttCust:write-xml("stream", "webstream", true).
```

This paper requires a JSON version of that simple .p that returns a row from the Customer table. Thanks to the support for both XML and JSON in OpenEdge, this is a very simple matter. You first need to define JSON as the output content type. The **serialize-name** gives a name to the top-level member of the JSON, just as it does to the top-level node in XML. Then instead of using the ABL **write-xml** method, which converts any temp-table or ProDataSet to an XML stream, you use the equivalent **write-json** method, which converts the same proprietary OpenEdge data format to standard JSON. That's all you have to do to send JSON back over the webstream instead of XML:

```
/* getcustomersJSON_param.p
   Server-side application to run to test Ajax / WebSpeed programs
     AjaxCustWebParamJSON.html
*/

{src/web2/wrap-cgi.i}

output-content-type ("text/json").

define temp-table ttCust like Sports2000.Customer.

temp-table ttcust:serialize-name = "customers".

find Sports2000.Customer where Customer.CustNum =
    INTEGER(get-value("piCustNum")).
create ttcust.
buffer-copy Customer to ttcust.

temp-table ttCust:write-json("stream", "webstream", true).
```

Saved under a new name, this procedure variant becomes what you run on the OpenEdge server using WebSpeed.

Next you need to make the right changes to the HTML example, shown in the paper ***IntroducingAJAX***, with its embedded JavaScript, which is what runs in the browser, using AJAX to make a request to WebSpeed.

The first key statement in the XML version of the JavaScript is the following one. It defines a string variable called **custXML** by referencing a property of the AJAX **XMLHttpRequest** object called **responseXML**:

```
/* This is the XML document that comes back
 through the webstream: */
var custXML = xmlhttp.responseXML.documentElement;
```

The browser is not going to be getting XML back this time, so in place of custXML, the file needs a new definition of an object variable called **custJSON**:

```
var custJSON = JSON.parse(xmlhttp.responseText);
```

It uses the intrinsic JSON object and its **parse** method, as shown in the JSON documentation, to retrieve the JSON string using the property **responseText**, and turn it into a proper JavaScript object. Just to show exactly what is coming back from WebSpeed and the **getcustomersJSON_param.p** procedure, I put in an alert statement to display the raw text that comes back before I parse it into a JavaScript object:

```
alert("responseText: " + xmlhttp.responseText);
```

The XML version uses the DOM to parse XML, as shown here:

```
var custRec =
        '<table>' +
                '<tr>' +
                        '<th align="left">Customer Number:  </th>' +
                        '<td>' +
                            document.getElementById( "custnum" ).value +
                        '</td>' +
```

The JSON version is going to use JavaScript references to extract individual values from the custJSON object:

```
var custRec =
        '<table>' +
                '<tr>' +
                        '<th align="left">Customer Number:  </th>' +
                        '<td>' +
                            custJSON.customers[0].CustNum +
                        '</td>' +
```

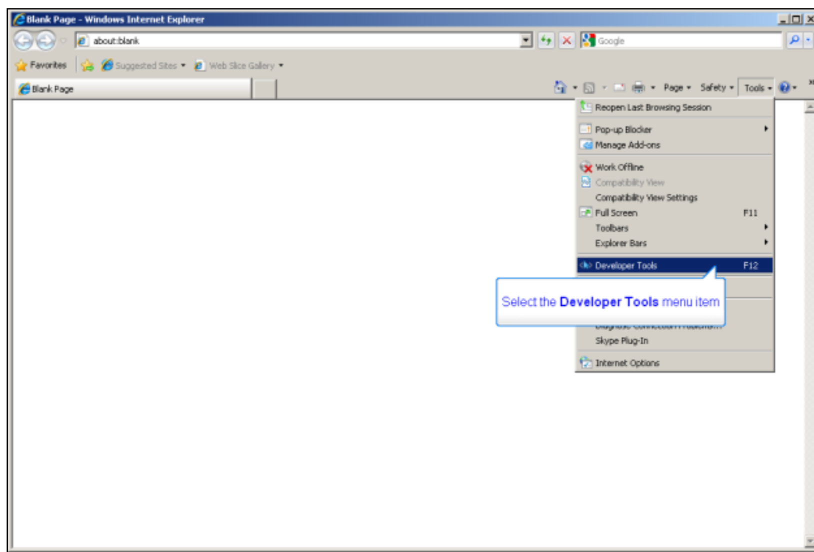
So in the custJSON object, the code references the first and only instance of the **customers** array – remember that **customers** is what was defined in the ABL procedure as the **serialize-name**, the name of the top-level data member – and from there its **CustNum** member value. And then it needs to do the same for the other two fields in the display: the Name field from the customer temp-table, and the Address field as well.

The last change needed is to make sure that the AJAX request is going to go to the right ABL procedure on the backend. That's the one where JSON was added to the procedure name:

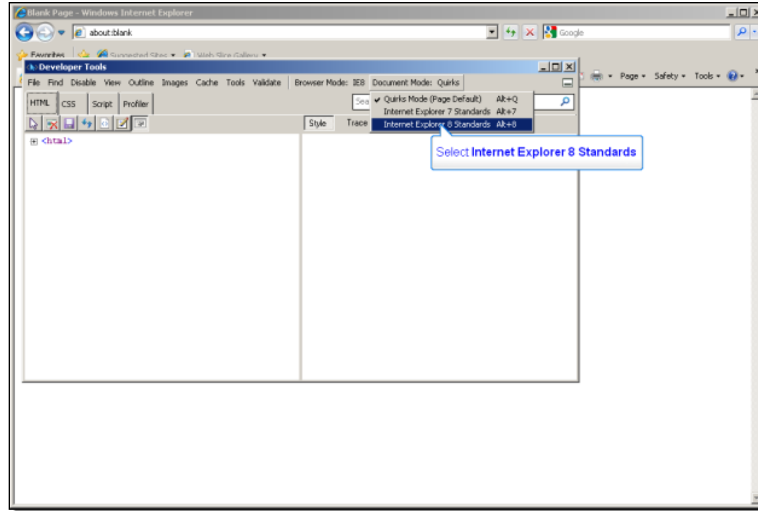
```
xmlhttp.open("GET",
  "cgi-bin/cgiip.exe/WService=wsbroker1/getcustomersJSON_param.p?piCustNum="
   + custval, true);
xmlhttp.send();
```

After saving the HTML under a new name, you can see whether the changes made all work together. In Internet Explorer, it's necessary to explain a complication to the test that's important for anyone using IE. Microsoft did not support the intrinsic JSON object that the code sample uses until Internet Explorer 8. And even then, for the sake of backward compatibility with applications that may have defined it on their own, it's not enabled by default. So to use the JSON object that the HTML file references, you need to enable the **IE8 Standards Document Mode** in the browser.

One way to do that is in the browser **Developer Tools** under the browser's **Tools** menu.

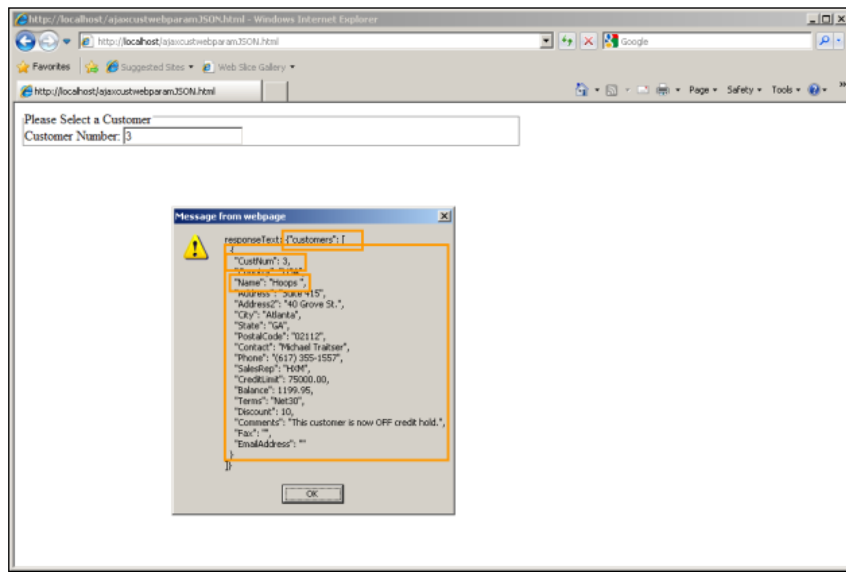


Selecting **Developer Tools** reveals the **Document Mode** dropdown list. By default the browser is set to **Quirks** mode. This means that the browser will interpret HTML, CSS, and JavaScript so as to maintain compatibility with older applications that may have used non-standard constructs to get around missing or inconsistent features in the languages or the browser. Instead you need to select **Internet Explorer 8 Standards**, in order for the JSON object and its methods to be recognized.



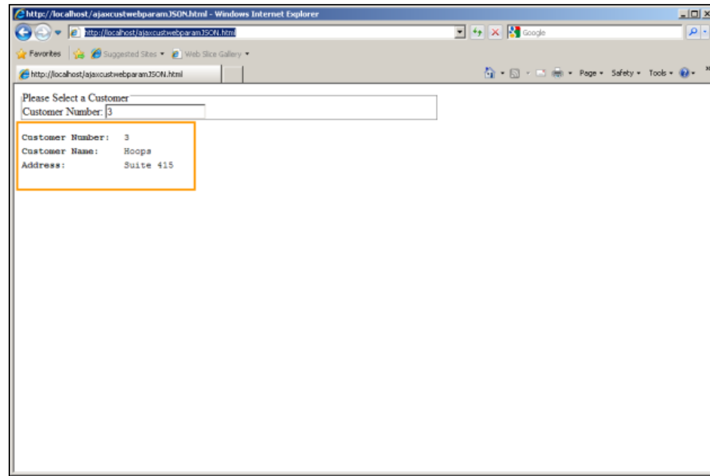
Now you can see if the example works the way it should. Loading the JSON version of the sample HTML file, the same little data entry form comes up that was used in the AJAX examples. If you enter a customer number in the fill-in field and tab out, that executes the **loadXMLdoc** function in the embedded JavaScript, which sets up its own internal function to wait for a ready state of 4, and then opens and sends the **XMLhttpRequest** to the server to run **getcustomersJSON_param.p**.

As you'll remember, I added an alert to the JavaScript so that you can first look at the JSON text that comes back:



Customers is the top-level member name, based on the **`serialize-name`** defined in the ABL. The brackets define a top-level array of objects, and in this case there's just one object in the array, delimited by the braces. Each member of the object has a quoted name, followed by a colon, and a value, which is also quoted unless it's a number or a boolean; and commas separate each member name-value pair.

Dismissing the alert box, you see the formatted fields from the customer data:



As before, if you now enter a new value, you get back just the data for the new request as output from WebSpeed, not a refresh of the entire HTML page. So the behavior is the same whether you use XML or JSON as the data exchange format with the server.

That's all for this session that introduced you to JavaScript Object Notation.