10 JSON: Javascript Object Notations

10.1 Introduction

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

The content of this chapter is sourced from various sources on the Net. The major ones are:
- http://www.json.org
- http://www.w3schools.com
- http://www.jsonexample.com
- http://bit.ly/1Enp3O1 (www.hunlock.com/...)

10.2 Comparison with XML

Much Like XML –

- JSON is plain text
- JSON is "self-describing" (human readable)
- JSON is hierarchical (values within values)
- JSON can be parsed by JavaScript
- JSON data can be transported using AJAX

Much Unlike XML –

- JSON blocks are enclosed within various braces
- JSON data is shorter
- JSON is quicker to read and write by humans
- JSON can be parsed using Javascript built-in `eval()`
- JSON uses arrays
- JSON has no reserved words

10.3 *Usage of JSON*

JSON is used extensively in AJAX calls used by web applications as it is faster and easier than XML. For evaluating an XML data part requires fetching it, using DOM or sax parser to parse its segments, and then extract its values. On the other hand, JSON based data part requires fetching it and passing it to Javascript's `eval()` function.

10.4 *Syntax Rules*

Following rules are applicable for building JSON based data parts:

- The object-attributes should be defined in name-value pairs.
- In a list, the items must be separated by commas.
- Object-attributes should be enclosed within curly braces.
- Sequences (arrays) should be enclosed within square braces.
- JSON values can be any of the following:
  - A number (integer or floating point)
  - A string (in double quotes)
  - A Boolean (true or false)
  - An array (in square braces)
  - An object (in curly brackets)
  - null

10.5 *Structure of JSON Objects*

Following are the details of various data or object types supported in JSON.

1. **Null:** JSON has a special value called `null` which can be set to any type of data including arrays, objects, number and boolean types. It represents emptiness.
2. **Boolean:** It can be either true or false.
3. **Number:** A JSON number is very much like an int in C or Java, except that its octal and hexadecimal formats are not applicable here.

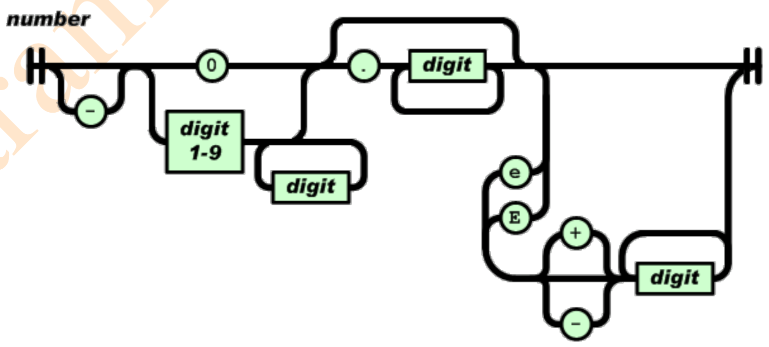

Figure 10.1: Structure of JSON Number

4. **String:** A string is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes. A character is represented as a single character string. A string is very much like a C or Java string.

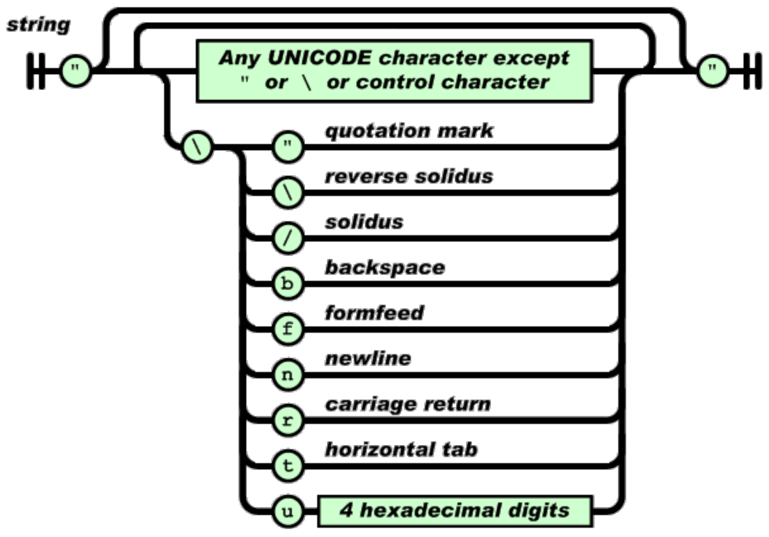

Figure 10.2: Structure of JSON String

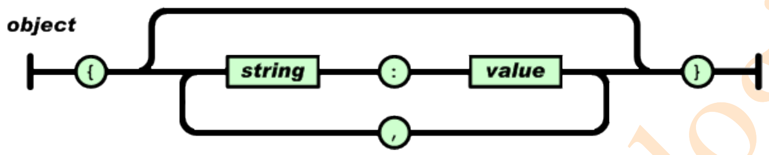

Figure 10.3: Structure of JSON Object

5. **Object:** An object is an unordered set of name-value pairs. An object begins with { (left curly brace) and ends with } (right curly brace). Each attribute of the object is followed by a : (colon) and each name-value pairs are separated by , (comma).
   The object may contain multiple comma-delimited key-value pairs. Object properties can be accessed using the the object name followed by a period and property name or it can be accessed like an array using the property name in quotes (in place of an index).

```
myObject = {"first": "Raj",
            "last": "Kumar",
            "age": 57,
            "sex": "M"}

myObject.last returns ‘Kumar’.
```

Object with nested array contains multiple comma-delimited key-value pairs and one or more nested arrays. The nested array can be accessed with the object name, property or ‘key’ containing the array and an array index.

```
myObject = {"first": "Raj",
            "last": "Kumar",
            "age": 57,
            "sex": "M",
            "interests": ["Reading", "Surfing", "Running"]}

myObject.interests[0] returns ‘Reading’.

myObject["interests"][0] returns ‘Reading’.
```

Object with nested object contains multiple comma-delimited key-value pairs and one or more nested objectx. To access an object within an object, property names or 'keys' can be chained together or property names can be used like an array in place of indexes.

```
myObject = {"first": "Raj",
            "last": "Kumar",
            "age": 57,
            "sex": "M",
            "favorites": {"color": "Red",
                          "sport": "Cricket",
                          "food": "Fried Rice"}}

myObject.favorites.food returns 'Fried Rice'.

myObject["favorites"]["food"] returns 'Fried Rice'.
```

Object with nested arrays and objects is more complex and typical of what might come by accessing an API. It contains key-value pairs, nested arrays and nested objects. Any of its elements can be accessed with a combination of the above techniques.

```
myObject = {"first": "Raj",
            "last": "Kumar",
            "age": 57,
            "sex": "M",
            "interests": ["Reading", "Surfing", "Running"],
            "favorites": {"color": "Red",
                          "sport": "Cricket",
                          "food": "Fried Rice"},
            "skills": [{"topic": "Javascript",
                        "tests": [{"name": 1, "score": 90},
                                  {"name": 2, "score": 96}]},
                       {"topic": "CouchDB",
                        "tests": [{"name": 1, "score": 79},
                                  {"name": 2, "score": 84}]}]}

myObject.skills[0].topic returns 'Javascript'.
myObject["skills"][0]["topic"] returns 'Javascript'.
myObject.skills[1].tests[0].score returns 79.
myObject["skills"][1]["tests"][0]["score"] returns 79.
```

6. **Array:** An array is an ordered collection of values. An array begins with [(left square brace) and ends with] (right square brace). Values are separated by , (comma). The values in an array need not be of same type.

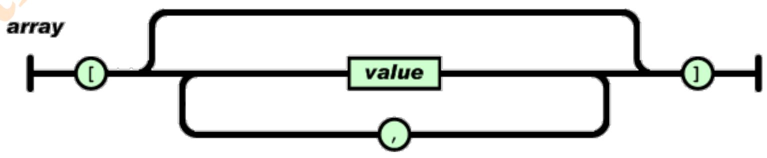

Figure 10.4: Structure of JSON Array

Array elements are generally of a basic type (number, string, boolean, or null). However it can also be a nested array (array of arrays or array of objects). Elements of an array are comma-delimited and contained within square braces.

```
myArray = [ "Raj Kumar", 57, "Male"]
myArray = [ [], {} ]
```

An array of objects contains comma-delimited objects, where every object may contain multiple comma-delimited key-value pairs. Objects within an array can be accessed using the array name and its item-index.

```
myArray = [{"name": "Raj Kumar", "age": 57},
           {"name": "Sana Shekh", "age": 22},
           {"name": "Peter Smith", "age": 19}]
```

`myArray[0].name returns 'Raj Kumar'.`

10.6 JSON Using Javascript

JSON can be manipulated very easily in Javascript. Following sections provide further details.

10.6.1 Creating a JSON Object

Earlier, the JSON object was created using the `new` keyword.

```
var myJSON = new Object();
```

This method has been deprecated now in favor of simply defining an empty object with a pair of curly braces.

```
var myJSON = {};
```

10.6.2 Objects as Data

At its most base level, a Javascript Object is a very flexible and robust data format expressed as name-value pairs. That is, an object holds a name which is an object's property – think of it as a plain old variable name that's attached to the object name. The object holds the value of that name. Here is an example:

```
var myFirstJSON = {"firstName": "Raj",
                   "lastName": "Kumar",
                   "age": 57};

document.writeln(myFirstJSON.firstName);  // Outputs Raj
document.writeln(myFirstJSON.lastName);   // Outputs Kumar
document.writeln(myFirstJSON.age);        // Outputs 57
```

This object has 3 properties or name-value pairs. The name is a string – in our example, firstName, lastName, and age. The value can be any Javascript object (and remember everything in Javascript is an object so the value can be a string, number, array, function, even other Objects). In this example, our values are Raj, Kumar, and 57. Raj and Kumar are strings but age is a number and as you can see this is not a problem at all.

This data format is called JSON – short for Javascript Object Notation. What makes it particularly powerful is that since the value can be any data type, you can store other arrays and other objects, nesting them as deeply as you need. Here is an example of a somewhat complex JSON structure:

```
var employees = {"accounting": [ // an array in employees
                                {"firstName": "Raj", // 1st item
```

```
                                   "lastName" : "Kumar",
                                   "age": 57},
                                  {"firstName": "Sana", // 2nd item
                                   "lastName": "Shekh",
                                   "age": 22}], // end: "accounting"
                    "sales": [ // another array in employees
                              {"firstName": "Peter", // 1st item
                               "lastName": "Smith",
                               "age": 19},
                              {"firstName": "Muskaan", // 2nd item
                               "lastName": "Kanu",
                               "age": 21}] // end: "sales"
                    } // end: employees
```

Here employees is an object. That object has two properties or name-value pairs. The accounting is an array which holds two JSON objects showing the names and age of two employees. Likewise sales is also an array which holds two JSON objects showing the name and age of two employees who work in that department. All of this data exists within the employees object. There are several different ways to access this data.

10.6.3 *Accessing Data*

The most common way to access JSON data is through dot notation. This is simply the object name followed by a period and then followed by the name-property you would like to access.

```
var myObject = {'color': 'blue'};
document.writeln(myObject.color); // outputs blue
```

If your object contains another object in it, then just add another period and name.

```
var myObject = {'color': 'blue',
                'animal': {'dog': 'friendly'}};

document.writeln(myObject.animal.dog); // outputs friendly
```

Using the employee example above, if we wanted to access the first person who worked in sales:

```
document.writeln(employees.sales[0].firstName + \
    ' ' + employees.sales[0].lastName); // outputs Peter Smith
```

We can also access the second person who works in accounting.

```
document.writeln(employees.accounting[1].firstName + \
    ' ' + employees.accounting[1].lastName);
```

To recap, the employees example is an object which holds two arrays each of which holds two additional objects. The only limits to the structure are the amount of storage and memory available to it. Because JSON can store objects within objects within objects and arrays within arrays that can also store objects, there is no virtual limit to what a JSON object can store. Given enough memory and storage requirement, a simple JSON data structure can store, and properly index, all the information ever generated by humanity.

10.6.4 *Simulation of Associative Array*

You can also access JSON data as if it is an Associative Array.

```
var myFirstJSON = { "firstName": "Raj",
                    "lastName": "Kumar",
                    "age": 57};
document.writeln(myFirstJSON["firstName"]); // Outputs Raj
document.writeln(myFirstJSON["lastName"]);  // Outputs Kumar
document.writeln(myFirstJSON["age"]); // Outputs 57
```

Be aware that this is NOT an associative array, however it appears. If you attempt to loop through myFirstObject you will get, in addition to the three properties above, any methods or prototypes assigned to the object, so while you're more than free to use this method of addressing JSON data, just treat it for what it is (Object Properties) and not for what it is not (Associative Array).

10.6.5 *More Examples*

1. Following snippet of JSON describes how a widget can be defined for building a UI.

```
{"widget": {"debug": "on",
            "window": {"title": "Sample Widget",
                       "name": "main_window",
                       "width": 500,
                       "height": 500},
            "image": {"src": "Images/Sun.png",
                      "name": "sun1",
                      "hOffset": 250,
                      "vOffset": 250,
                      "alignment": "center"},
            "text": {"data": "Click Here",
                     "size": 36,
                     "style": "bold",
                     "name": "text1",
                     "hOffset": 250,
                     "vOffset": 100,
                     "alignment": "center"}}}
```

2. Following represents the content of a menu. See the flexibility of leaving off some items (object-properties) from the contained objects. The action and label values only need to be provided if they are not the same as the id.

```
{"menu": {
    "header": "SVG Viewer",
    "items": [
        {"id": "Open"},
        {"id": "OpenNew", "label": "Open New"},
        null,
        {"id": "ZoomIn", "label": "Zoom In"},
        {"id": "ZoomOut", "label": "Zoom Out"},
        {"id": "OriginalView", "label": "Original View"},
        null,
        {"id": "Quality"},
        {"id": "Pause"},
        {"id": "Mute"},
        null,
        {"id": "Find", "label": "Find..."},
        {"id": "FindAgain", "label": "Find Again"},
        {"id": "Copy"},
        {"id": "CopyAgain", "label": "Copy Again"},
        {"id": "CopySVG", "label": "Copy SVG"},
        {"id": "ViewSVG", "label": "View SVG"},
        {"id": "ViewSource", "label": "View Source"},
        {"id": "SaveAs", "label": "Save As"},
        null,
        {"id": "Help"},
        {"id": "About", "label": "About Adobe CVG Viewer..."}]}}
```

3. Following example shows how to use nested structures to simply group data together.

```
{"id": "0001",
 "type": "donut",
 "name": "Cake",
 "image": {"url": "images/0001.jpg",
           "width": 200,
           "height": 200},
 "thumbnail": {"url": "images/thumbnails/0001.jpg",
               "width": 32,
               "height": 32}}
```

10.6.6 *Parsing JSON*

The way to parse a JSON snippet is shown below. It is implemented by using `string.parseJSON()` method of Javascript. This prototype has been released into the Public Domain, 2007-03-20. Original Author was Douglas Crockford. Originating Website was http://www.JSON.org. Originating URL was http://www.JSON.org/JSON.js.

Augment is `String.prototype`. We do this in an immediate anonymous function to avoid defining global variables. Here m is a table of character substitutions. (Please note that the indentation of code shown in this particular example is not compliant to good coding practice. This code snippet has been reformatted to fit within a page for publication.)

```
(function(s) {var m = {’\b’: ’\\b’,
                       ’\t’: ’\\t’,
                       ’\n’: ’\\n’,
                       ’\f’: ’\\f’,
                       ’\r’: ’\\r’,
                       ’"’ : ’\\"’,
                       ’\\’: ’\\\\’};
// Parsing happens in three stages. In the first stage,
// we run the text against a regular expression which looks
// for non-JSON characters. We are especially concerned with
// ’()’ and ’new’ because they can cause invocation, and ’=’
// because it can cause mutation. But just to be safe, we
// will reject all unexpected characters.
s.parseJSON = function(filter) {try {
if (/^("(\\.|[^"\\\n\r])*?"|[,:{}\[\]0-9.\-+Eaeflnr-u \n\r\t])+?$/.test(this)) {
// In the second stage we use the eval function to compile
// the text into a JavaScript structure. The ’{’ operator
// is subject to a syntactic ambiguity in JavaScript: it can
// begin a block or an object literal. We wrap the text in
// parenthesis to eliminate the ambiguity.
  var j = eval('(' + this + ')');

// In the optional third stage, we recursively walk the new
// structure, passing each name-value pair to a filter function
// for possible transformation.
  if (typeof filter === 'function') {function walk(k, v) {
    if (v && typeof v === 'object') {for (var i in v) {
      if (v.hasOwnProperty(i)) {v[i] = walk(i, v[i]);}}}
    return filter(k, v);}
    j = walk('', j);}
    return j;}} catch (e) {
// Fall through if the regexp test fails.
  } throw new SyntaxError("parseJSON");};}) (String.prototype);


// Begin sample code
// Example of what is received from the server.
JSONData = '{"color" : "green"}';
testObject=JSONData.parseJSON();
document.writeln(testObject.color); // Outputs: Green.
```

10.7 *JSON via AJAX*

JSON data snippets are very useful to transfer data between client and server using AJAX calls. Elaboration on how to use it in AJAX requests and responses follows.

There's no standard naming convention for these methods, however assignment method is a good descriptive name because the file comming from the server creates a javascript expression which will assign the JSON object to a variable. When the `responseText` from the server is passed through `eval()`, `someVar` will be loaded with corresponding JSON object and you can access it from there onwards.

```
// Following is received from the server
var JSONFile = "someVar = { 'color' : 'blue' }";
eval(JSONFile); // Execute the value of JSONFile
document.writeln(someVar.color); // Outputs 'blue'
```

Another way of of handling JSON in AJAX is via callback mechanism. Here a predefined method is invoked to which the JSON data is passed as the first argument. A good name for this method is the callback method. This approach is used extensively when dealing with third party JSON files, *i.e.* the JSON data from domains which are not under your control.

```
function processData(incommingJSON) {
  document.writeln(incommingJSON.color); // Outputs 'blue'
}

// Example of what is received from the server...
var JSONFile = "processData({'color' : 'blue'})";
eval(JSONFile);
```

The third option is to parse the JSON data. The details of parsing JSON is explained in section 10.6.6.

When you are communicating with your own servers, AJAX is one of the better ways to transfer data from the server to the browser. Provided the server is under your control, you can be reasonably assured of the safety of this kind of data transfer. Here is an example of a simple AJAX request which communicates with the server, retrieves some data, and passes it along to a processing function. We'll be using the callback method here where the JSON object will execute a predefined function after it's loaded, in this case – processData(JSONData).

```
function processData(JSONData) {
  alert(JSONData.color);
}
var ajaxRequest = new ajaxObject('http://server/get.php');
ajaxRequest.callback = function (responseText) {
  eval(responseText);
}
ajaxRequest.update();

// In this example we assume the server sends back the
// following data file (which the ajax routine places in
// responseText)
processData({ "color" : "green" })
```

In this example our data file, when passed through the eval statement will execute processData(). This happens because our data file is actually Javascript code. The JSON data is passed to the function as its first argument.

The next example will use the parse method and assumes you have the parseJSON() prototype located elsewhere in your code.

```
function processData(JSONData) {
  alert(JSONData.color);
}

var ajaxRequest = new ajaxObject('http://server/getdata.php');
ajaxRequest.callback = function (responseText) {
  JSONData = responseText.parseJSON();
  processData(JSONData);
}
ajaxRequest.update();

// In this example we assume the server sends back the following
// data file (which the ajax routine places in responseText)
//
// {"color": "green"}
```

Now when the the server returns the JSON file, it's parsed on the line which reads JSONData = responseText.parseJSON(); and then JSONData is passed to processData(). The end result is the same as the first example, but if, for some reason, the JSON data contained malicious or invalid data then the second example would be more likely to securely handle any problems.

Because AJAX data is sent as an encoded string, some preparation of JSON data must be made before it can be sent to the server. Fortunately, Douglas Crockford at JSON.org has released a set of very useful routines which will convert any Javascript data type into a JSON string which can be easily sent to the server.

The source for this library can be obtained from http://www.JSON.org/JSON.js. The code is public domain and is as easy to use as clipping the data and pasting it into your toolbox.

The following example defines a JSON object then uses method toJSONString() to convert the object into a string which is ready to be sent to the server.

```
var employees = {"accounting": [{"firstName": "Raj",
                                  "lastName": "Kumar",
                                  "age": 57},
                                 {"firstName": "Peter",
                                  "lastName": "Smith",
                                  "age": 19}],
                  "sales": [{"firstName": "Muskaan",
                             "lastName": "Kanu",
                             "age": 21},
                            {"firstName": "Sukhvinder",
                             "lastName": "Singh",
                             "age": 39}]}
var toServer = employees.toJSONString();
document.writeln(toServer);
```

This outputs the following on a single line (ignore line breaks):

```
{"accounting": [{"firstName": "Raj", "lastName": "Kumar",
"age": 57}, {"firstName": "Peter", "lastName": "Smith",
"age": 19}], "sales": [{"firstName": "Muskaan",
"lastName": "Kanu", "age": 21}, {"firstName": "Sukhvinder",
"lastName": "Singh", "age": 39}]}
```

Retrieving JSON from third party server is a very tricky issue. The code shared in this section comes with a bunch of warnings. Up until this point, JSON and AJAX has been relatively secure since you are communicating with servers under your control, receiving data that is under your control. With third party services all of this changes.

As of Internet Explorer 7 and Firefox 2, use of third party JSON data exposes your web page to malicious attacks and great security risks. The moment you request third party data you have given up control of your web page and the third party can do whatever they want with it from scraping the data and sending back sensitive information to installing listeners to wait for sensitive data to be entered.

For most trusted sources the only thing an external JSON request will get you is a bunch of useful data. However if the page making the third party request contains form elements, requires a secure/encrypted connection, or contains ANY personal or confidential data you absolutely, positively should NOT use third-party JSON calls on that page.

Before you attempt to use third party JSON data take a good, hard look at the page you are creating and now imagine that the worst sort of person is looking at that page with you. If you are uncomfortable with what that other person can do with the information displayed on that page, you should not be using third-party JSON calls.

Quite a few services are starting to offer JSON in addition to RSS/XML as data formats. Yahoo, in particular, is quite progressive at implementing JSON. What's very cool about JSON is that the web page can directly request and process that data, unlike XML which must be requested by the server and then passed along to the browser. Unfortunately there is no RSS/FEED standard for JSON, though most services try to emulate RSS XML in a JSON structure. Please note that jQuery's jFeed can generate JSON data from and RSS feed.

For the following example we'll use a fictitious site's JSON feed, a simple one-to-one conversion of XML to JSON. The feed is even generated by the exact same program which generates the XML version, only the arguments passed on the URL determine which format is sent. The URL for this fictitious site's feed is: http://mysite/feed.php?format=JSON&callback=JSONFeed

All third party JSON requests use the callback method where, once the JSON file has been loaded it will call a specific function, passing the JSON data as the first argument. Most services have a default function name they will call, and some services will allow you to change the name of the function which is called. This site's JSON feed will call `JSONFeed()` by default but you can change this by changing the name in the URL. The site can be designed in such a way that if you changed the URL to read http://mysite/feed.php?format=JSON&callback=processJSON then whenever the feed gets loaded, it will call the function named `processJSON()` and pass the JSON data as the first argument. This is fairly important when you are handling multiple JSON requests on your page.

All third party JSON requests are loaded via the `<script>` tag, just like you would use to load external Javascript files. What's a little different is that we create the script tag manually and attach it to the page manually as well. This is a fairly simple process and the code to do it is quite simple and readable. Here's a small function which will accept a URL and loads it.

```
function loadJSON(url) {
  var headID = document.getElementsByTagName("head")[0];
  var newScript = document.createElement('script');
  newScript.type = 'text/javascript';
  newScript.src = url;
  headID.appendChild(newScript);
}
```

This function gets the (first) `<head>` element of our page, and creates a new script element. It also gives it a type, sets the `src` attribute to the URL which is passed and then appends the new script element to the page `<head>`. Once appended, the script will be loaded and then executed.

Here is a very simple little program to get this site's JSON feed and displays the items.

```
function loadJSON(url) {
  var headID = document.getElementsByTagName("head")[0];
  var newScript = document.createElement('script');
  newScript.type = 'text/javascript';
  newScript.src = url;
  headID.appendChild(newScript);
}

function processJSON(feed){
  document.writeln(feed.title+'<BR>');
  for(i=0; i<feed.channel.items.length; i++) {
    document.write("<a href='"+feed.channel.items[i].link+"'>");
    document.writeln(feed.channel.items[i].title+"</a><BR>");
  }
}
```

Invocation of Javascript method
`loadJSON('http://mysite/feed.php?format=JSON&callback=processJSON');`
creates a function to load third party Javascripts (`loadJSON()`) and a function to process the data it receives (`processJSON()`). The last line calls `loadJSON()` and passes it the URL to use. Once the script has finished loading, it's executed automatically which will call `processJSON()` (because that's the name we specified in the callback), and `processJSON()` will loop through all the items in the file showing the article title as a clickable link.

10.8 JSON Security

JSON is a data interchange format. It is used in the transmission of data between machines. Since it carries only data, it is security-neutral. The security of systems that use JSON is determined by the quality of the design of those systems. JSON itself introduces no vulnerabilities.

The web browser is a peculiar application environment. The security model of the browser was forged through a long series of foreseeable and painful blunders. Most of the holes in the browser have been plugged, but in some cases the plugs become annoyances which must be circumvented, and that circumvention leads to a continuing series of new blunders.

This pain can be avoided by adopting good practices. Often, so-called experts seem to be incapable of distinguishing the good practices from the bad ones, so there is a lot of bad advice available on the web.

Following is a small set of principles which can be seen to be true. If you hold to these principles, you will be much less likely to adopt bad practices while writing applications using JSON data formats.

Never Trust The Browser: The browser cannot and will not protect your secrets, so never send it your secret sauce. Keep your critical processes on the server. If you are doing input validation in the browser, it is for the user's convenience. Everything that the browser sends to the server must be validated.

Keep Data Clean: JSON is a subset of JavaScript, which makes it especially easy to use in web applications. The eval function can take a text from `XMLHttpRequest` and instantly inflate it into a useful data structure. Be aware however that the eval function is extremely unsafe. If there is the slightest chance that the server is not encoding the JSON correctly, then use the `parseJSON()` method instead. The `parseJSON()` method uses a regular expression to ensure that there is nothing dangerous in the text. If your version of Javascript doesn't have it by default, then you can get `parseJSON()` at `http://www.JSON.org/json.js`. On the server side, always use good JSON encoders and decoders.

Script Tags: Script tags are exempt from the *Same Origin Policy*. That means that any script from any site can potentially be loaded into any page. There are some very important consequences of this.

[Following two paragraphs need verfication!]

Any page that includes scripts from other sites is not secure. External scripts can be used to deliver ads or search result options, or logging, or alerts, or buddy lists, or lots of other nice things. Unfortunately, the designs of JavaScript and the DOM did not anticipate such useful services, so they offer absolutely no security around them. Every script on the page has access to everything on the page. When you load a script on a page, you are authorizing that script to have access to all of your confidential information and all of your user's confidential information. You are also authorizing that script to access your server on the user's behalf to do anything it wants. It is not possible to distinguish between requests made by the user and requests made by an invited script. Hopefully, some day soon, browsers will offer some degree of modularity that would limit the danger. Until then, script tags are extremely dangerous.

Another use of script tags is to deliver JSON data from different sites. There is absolutely no protection against the case where the different site sends dangerous scripts instead of benign JSON data. Hopefully, some day soon, browsers will offer safe cross-site data transport. Script tags are too dangerous to use for data transport.

Script tags can also be used to deliver Ajax Libraries. Do not load libraries from other sites unless you have a high level of trust in the vendors.

Don't Send Data To Strangers: If your server sends sensitive data to an evil web page, there is a good chance that the browser will deliver it. In some cases the Same Origin Policy is supposed to block such deliveries, but the fact is that browsers are buggy and sometimes the data will go through. It doesnâĂŹt make sense to get mad at the browser. The secure release of confidential information is the serverâĂŹs responsibility. That responsibility should never be delegated to the browser.

All requests must be authenticated. You must have some secret which is known only by the page that indicates that it should be given the goods. Cookies are not adequate authentication. DonâĂŹt use a cookie as the secret. JSON often works best in a dialog: POST a request including the secret in the JSON payload, and get a JSON response in return along with a new secret.

Since script tags are exempt from the Same Origin Policy, a script tag can be used from any page to make a GET request of your server. The request will even include your cookies. If the response contains confidential information, then your server must refuse the request.

There are people who are selling magic wrappers to put around JSON text to prevent delivery to unauthorized receivers. Avoid that stuff. It will fail when new browser bugs are created and discovered, and in some cases might introduce painful new exploits.

Use SSL: Any time you are transmitting confidential information or requests for confidential information, use SSL. It provides link encryption so that your secrets are not revealed in transit.

★★★★★