Lesson : JASON

Introduction to JSON

Communicating data between the server and browser is a significant task for modern web applications. Initial attempts to do so included unstructured text documents and heavily structured XML documents, both of which required significant effort to convert to a usable format. JavaScript Object Notation, or JSON, is an efficient, structured format for data based on a subset of the JavaScript language. JSON (pronounced "Jason") is intended to be easily readable by humans and computers. Debugging communication that uses JSON is easy because humans can read JSON. Communication is efficient because computers can transmit and parse JSON quickly. As a result, JSON has rapidly become the dominant format of data transfer between web browsers and servers.

JSON structure and values

JSON has six basic data types:

1. **String** - Unicode characters enclosed within double quotes (`"`). A few special characters must be escaped with a backslash (`\`). Ex: backslashes (`\\`), double quotes (`\"`), newlines (`\n`), and tabs (`\t`).
2. **Number** - Either an integer or decimal number. Ex: `42`, `3.141`, `-1.1e-5`.
3. **Object** - Unordered list of zero or more name/value pairs separated by commas and enclosed within braces (`{}`). A name in a JSON object must be a string in double quotes. A value can be any legal JSON value. Each name and value is separated by a colon. Ex: `{ "Name": "Joe", "Age": 35 }`
4. **Array** - Ordered list of zero or more JSON values separated by commas and enclosed within brackets (`[]`). Ex: `[]` and `[13,"blue"]`.
5. **Boolean** - Either `true` or `false`.
6. **null** - Represents "nothing".

A JSON value can be any of the above data types.

The JSON structure is defined recursively so that objects can contain arrays and arrays can contain objects to any arbitrary depth.

A common error when generating JSON programmatically is to include a trailing comma after the list of name/value pairs in a JSON object or after the list of JSON values in a JSON array. Ex: `[0,`

`1, 2,]`.

An example JSON data structure.

```
{
   "name": "Seema Fernandez",
   "vehicles": [
      {
         "make": "Audi",
         "model": "X123",
         "color": "Red"
      },
      {
         "make": "Toyota",
         "model": "RAV4",
         "color": "blue"
      }
   ],
   "married": true,
   "previous_customer": false,
   "known_associates": [],
   "notes": null
}
```

The JSON structure above is an object with six name/value pairs:

1. `name` has the string value Seema Fernandez.
2. `vehicles` has an array value of two objects. Each object in the vehicles array has three name/value pairs: make, model, and color.
   1. The array's first object's `make` is the string `Audi`, `model` is the string X-123, and `color` is the string `red`.
   2. The array's second object's `make` is `Toyota`, `model` is `RAV4`, and `color` is `blue`.
3. `married` is `false`.
4. `previous_customer` is `true`.
5. `known_associates` is an empty array.
6. `notes` is `null`.

Working with JSON

JavaScript provides a built-in JSON object that provides two methods for working with JSON:

1. The JSON.parse() method creates a JavaScript object from a string containing JSON. Ex: `JSON.parse('[1,"two",null]')` converts the string `'[1,"two",null]'` into the JavaScript array `[1,"two",null]`. Typically, `JSON.parse()` is used with data received from a server.

2. The JSON.stringify() method creates a string from a JavaScript object. Typically, `JSON.stringify()` is used with data sent to a server. `JSON.stringify()` creates a string representation of any passed object by either calling the object's `toJSON()` method if defined or recursively serializing all enumerable, non-function properties. Ex: `JSON.stringify(new Date('2020-08-06'))` converts the JavaScript Date object to the string `2020-08-06T00:00:00.000Z` by calling the Date object's `toJSON()` method.

Good practice is to use single quotes around JavaScript strings containing JSON notation so that the double quotes for strings and JSON object names do not need to be escaped. Ex:
Use `'{"name":"Bob"}'` instead of `"{\"name\":\"Bob\"}"`.

Extending and customizing JSON output

The `JSON.parse()` method's second parameter is an optional parameter for a reviver function. A reviver function is used to modify parsed values before being returned, and is helpful when a JSON string represents a data type not available in JSON. Ex: A reviver function can convert a string representing a date, `"2010-12-30"`, to a JavaScript Date object.

The `JSON.stringify()` method has two optional parameters: a replacer and a spacer. The replacer enables customization of the generated string. If replacer is a function, `JSON.stringify()` will use the value returned by the function as the string representation. Ex: A replacer can convert a JavaScript type not directly supported in JSON to a string representation of that data type. If replacer is an array, `JSON.stringify()` will filter the returned value by converting only the properties listed in the replacer array. Ex: `JSON.stringify({a:1,b:2,c:3},["a","b"])` returns the string `'{"a":1,"b":2}'`.

The spacer controls the indentation spacing of output JSON string, which indicates the depth of values in the object. When the spacer parameter is specified and not an empty string, the output will also include newlines. Ex: `JSON.stringify({a:1,b:2}, null, " ")` returns the string below because the spacer parameter is a string with two spaces.

```
'{
  "a": 1,
  "b": 2
}'
```