Javascript Outside of the Browser

JSON

The JSON format is not strictly a NodeJS topic, but you need to be familiar with it to move forward with NodeJS.

JSON stands for JavaScript Object Notation. The original specification was popularized by Douglass Crockford in 2001, though the convention of sending serialized Javascript objects dates back earlier than that. It's a data exchange format similar to XML, but with notable differences that make it not a 1-to-1 comparison. Because it fundamentally uses Javascript as a data structure, it is natively parseable in Javascript, and due to its simplicity, most popular programming languages today support serialization/deserialization methods natively.

At its core, JSON is just a JSON Object literal or Array with slightly stricter conventions. Here's a sample JSON object:

```
{
  "title": "User Data",
  "version": 1,
  "active": false,
  "people": [
    {
      "name": "Bob Barker",
      "age": 95
    }, {
      "name": "Grigori Rasputin",
      "age": 150
    }
  ]
}
```

All of these properties are custom. There's no mandatory properties in JSON.

JSON data structures can support most of the same basic data types Javascript can, including Numbers, Strings, Booleans, Nulls, Arrays, and Objects.

Any other data type is not supported. There are no Functions in JSON, as JSON is simply a data exchange format, not a dynamic script. There is also no support for Undefined values. Data can explicitly be defined to not exist with `null`, but `undefined` suggests a value is implicitly not defined.

There are other restrictions between JSON and a traditional Javascript object. For example, JSON strings **must** be defined with double quotes: single quotes or back-ticks are not allowed. Also, all JSON properties need to have quotes around them.

Now that we have a JSON file, we'll move on to parsing it for use inside a NodeJS script. First I'll paste the JSON object directly in the script to show that it is in fact valid Javascript syntax.

```
var json = {
  "title": "User Data",
  "version": 1,
  "active": false,
  "people": [
    {
      "name": "Bob Barker",
      "age": 95
    }, {
      "name": "Grigori Rasputin",
      "age": 150
    }
  ]
}

console.log('Title: ' + json.title)
console.log('People:\n  -')
for (var i = 0; i < json.people.length; i++) {
  console.log('  Name: ' + json.people[i].name)
  console.log('  Age: ' + json.people[i].age)
  console.log('  -')
}
```

My static analysis tool doesn't like the double quotes but it's not an error.

You can see that it's parsed like any object. You can access properties and array elements the same way you would any Javascript object.

You typically won't include raw JSON data in your script files though. Instead, you'll get data from a network request, or in our case, from a separate local file. One way to load JSON files in NodeJS is to use the `require` statement. It works similarly to loading a global module like `path` or `fs`: just pass in a local path to the JSON file, and the function returns the JSON Object:

```
var json = require('./data.json')

console.log('Title: ' + json.title)
console.log('People:\n  -')
for (var i = 0; i < json.people.length; i++) {
  console.log('   Name: ' + json.people[i].name)
  console.log('   Age: ' + json.people[i].age)
  console.log('   -')
}
```

You use a dot and forward slash to specify a relative path regardless of whether you're on a Windows OS or not.

Besides its simplicity, the power in the JSON format is that it can be serialized, or converted into a universal format that is easily transmitted or stored, and deserialized, converted back into a format the target system can work with. For example, JSON data may come from a network request, and then needs to be parsed in the browser to update the page in some way.

As is, this Javascript Object cannot be transferred or stored. It needs to be converted to a simpler format. This can be done with Javascript's built in `JSON.stringify` function.

`JSON.stringify` converts a Javascript or JSON object into a string of characters. This data can then safely be sent to a file or transmitted in a network request.

The argument passed in does not need to be a JSON object. Any object or array that does not have circular references (properties that reference parent properties) can be serialized, though any `undefined` or `function` properties will be discarded.

Here's an example of a Javascript object being serialized:

```
var sampleObject = {
  value: 'Apple',
  doSomething: function () {}
}

console.log(JSON.stringify(sampleObject))
```

Notice that the resulting output does not include the function. `stringify` automatically removes it.

Here is an example of a circular reference to show that the stringify function fails:

```
var sampleObject = {
  value: 'Apple',
  doSomething: function () {}
}

sampleObject.circularReference = sampleObject

console.log(JSON.stringify(sampleObject))
```

The `stringify` function also includes optional parameters to tailor the output. The `replacer` function can be used to replace particular properties when generating output. It's called for each property being parsed, and is expected to return the result. For example, you could make all string properties return in all caps like this:

```
var sampleObject = {
  value: 'Apple',
  doSomething: function () {}
}

// sampleObject.circularReference = sampleObject

var serializedJson = JSON.stringify(
  sampleObject,
  function replacer (key, value) {
    if (typeof value === 'string') {
      return value.toUpperCase()
    }
    return value
  }
)
console.log(serializedJson)
```

The `typeof` operator returns the string representation of the specified data type. For strings, that's the string 'string'.

It's unlikely you'll be using the replacer function much, but the next optional argument does show up often: the `space` argument. By default, `JSON.stringify` will squish everything on one line, which is optimal for storage and data transfer. However, it's harder for humans to read. The `space` argument allows indenting each line of the resulting JSON string with spaces or custom characters. I most often use the numeric argument to specify that I want the output to include 2 spaces after each property. I typically leave the replacer argument null since I don't use it.

```
var sampleObject = {
  value: 'Apple',
  doSomething: function () {}
}

// sampleObject.circularReference = sampleObject

var serializedJson = JSON.stringify(
  sampleObject,
  null, // replacer function not required
  2 // add two spaces to the output
)
console.log(serializedJson)
```

Another uncommon capability of `JSON.stringify` is the option to define how to generate the JSON output inside the object itself. If an object defines a `toJSON` function, `JSON.stringify` will convert the output of that into serialized JSON, as opposed to the object itself. Here's an example of how to add an additional property to the output:

```
var sampleObject = {
  value: 'Apple',
  doSomething: function () {},
  toJSON: function () {
    return {
      value: this.value,
      additional: 'property'
    }
  }
}

// sampleObject.circularReference = sampleObject

var serializedJson = JSON.stringify(sampleObject)
console.log(serializedJson)
```

So we went over how to load JSON files in NodeJS and convert JSON or Javascript Objects into a serialized string. To convert this string back into a Javascript Object, use the `JSON.parse` function.

```
var sampleObject = {
  value: 'Apple',
  doSomething: function () {}
}

// sampleObject.circularReference = sampleObject

var serializedJson = JSON.stringify(sampleObject)
console.log(serializedJson)

var deserializedJson = JSON.parse(serializedJson)
console.log(deserializedJson.value)
```

As long as the string argument passed in is a valid JSON Object or Array, a Javascript Object or Array will be returned. Otherwise an error is thrown.

Here's a standalone example:

```
console.log(JSON.parse('{"foo": "bar"}'))
```

`JSON.parse` also has an optional argument called the `reviver`. It's similar to the `replacer` in the `JSON.stringify` function: it is called for each property of parsed data and allows the developer to transform the result. For example, all the characters in strings could be made lowercase like this:

```
var sampleObject = {
  value: 'Apple',
  doSomething: function () {}
}

// sampleObject.circularReference = sampleObject

var serializedJson = JSON.stringify(sampleObject)
console.log(serializedJson)

var deserializedJson = JSON.parse(
  serializedJson,
  function reviver (key, value) {
    if (typeof value === 'string') {
      return value.toLowerCase()
    }
    return value
  }
)
console.log(deserializedJson.value)
```

Like the `replacer` function, it's rare to see this in the wild.

So, we went over what JSON is, and how it can be serialized and deserialized in NodeJS and the browser. Let me restate that JSON is not exclusively for NodeJS, or even Javascript. It's an extremely common format that you'll see in many other languages.

Knowing how the JSON data format works will allow us to create NodeJS packages and utilize the Node Package Manager, which we'll get into next.