The JSON Web Data Model for JavaScript

Computer Science & Information Technology

School of Matematical & Computational Sciences
Massey University (AKLI, DISD & MTUI)

Slide deck main resource:

Tom Marrs (2017), *JSON at Work: Practical Data Integration for the Web*. Chapters 1 and 2: pp. 3 - 55. O'Reilly Media, Inc. ISBN:

978-1-449-35832-7

Learning Objectives

In this topic, you will learn how to do the following:

1. Explain the JSON Basics;
2. Model JSON data;
3. Structure JSON documents with JSON schema;
4. Search contents of JSON documents;
5. Convert JSON documents to other formats; and
6. Use JSON with JavaScript.

JSON as a Standard

JSON is an *international data processing* standard:

- Internet Engineering Task Force (IETF) - RFC 4627 (2006) and RFC 7158, 7159 (2014);
- ECMA (European Computer Manufacturers Association) International — ECMA 404 (2013)

JSON Example

```
{ "BookTitle": "JSON at Work", "author":"Tom Marr" }
```

This example shows an Object that that contains two key-value pairs, where:

- The key "BookTitle" has the value "JSON at Work";
- The key "author" has the value "Tom Marr".

The JSON document in the above example can be **validated** using JSONLint

A **valid JSON document** can be either of the following:

- An Object surrounded by curly braces, { and }; or
- An Array enclosed by brackets, [and]

Why JSON — I

The following factors make the JSON data model popular:

- Explosive growth of RESTful APIs based on JSON;
- Simplicity of JSON's basic data structures; and
- Increasing popularity of JavaScript
- JavaScript rising to become 1st class development language and environment with the following in its ecosystem:
  - *Node.js;*
  - *Model-View-Contnroller (MVC) Frameworks — AngularJS, React, Backbone and Ember.*
- Growing literature on JavaScript Objects and Patterns with JSON as JavaScript's Object Literal notation;

- RESTful APIs using JSON, e.g., LinkedIn, Twitter, Facebook, Amazon Web Services (AWS).
    - *See the RESTful API directory — Programmable Web*

Why JSON — 2

- JSON is simple and replacing XML as Internet's *data interchange format*;
- JSON structure translate to concepts well understood by developers — arrays, objects and name-value pairs;
- JSON is more compact than XML and hence more effient to process and transmit;
- JSON is also now being used in software configuration files;

JSON Data Types

- Core JSON data format — JSON data plus Value Types;
- The core JSON data types are as follows:

    1. Name-Value or key-value pairs — consists of a key (an attribute) and a value;

    2. Object — an unordered collection of name-value pairs;

    3. Array — a collection of ordered pairs.

Characteristics of Name-Value Pairs

- Example name-value pairs:

```
{
    "conference": "IEEE GHTC 2019",
    "speechTitle": "Customisability of Knowledge Incorporated into Global Health Technologies",
    "track": "Knowledge Engineering Track"
}
```

- Name-Value pairs has the following characteristics:
  - *Each name such as "conference":*
    - Is on the left side of the colon (:), and
    - Is a String and must be surrounded by double quotes.
  - *The Value such as "IEE GHTC 2019" is to the right of the colon. In this example, the Value Type is a String, but there are several types of other Value Types*

JSON Object Example — I

An Address

```
{
    "address" : {
                    "line1"             : "55 College Street",
                    "line2"             : "Awapuni"
                    "city"              : "Palmerston North",
                    "Province"          : "Manawatu",
                    "PostalCode"        : "4412",
                    "country"           : "New Zealand"
                }
}
```

A JSON Object with a Nested Array

```
{
    "speaker" : {
                    "firstName" : "Kuda",
                    "lastName"  : "Dube",
                    "topics"    : [ "JSON", "XML", "Validation using Schemas" ]
                }
}
```

JSON Object Example — 2

The following speaker object contains other objects: topics and address:

```
{
    "speaker" : {
                    "firstName" : "Larson",
                    "lastName"  : "Richard",
                    "topics"    : [ "JSON", "REST", "SOA" ],
                    "address"   : {
                                        "line1"           : "555 Any Street",
                                        "city"            : "Denver",
                                        "stateOrProvince" : "CO",
                                        "zipOrPostalCode" : "80202",
                                        "country"         : "USA"
                                    }
                }
}
```

Characteristics of JSON Objects

Objects have the following characteristics:

- Are enclosed within a beginning left curly brace ({) and an ending right curly brace (});
- Consist of comma-separated, unordered, name/value pairs;
- Can be empty, { };
- Can be nested within other Objects or Arrays.

JSON Array Example — I

An Array (containing nested Objects and Arrays) that describes conference presentations, including title, length, and abstract.

```
{
    "presentations": [
        {
            "title": "JSON at Work: Overview and Ecosystem",
            "length": "90 minutes",
            "abstract": [ "JSON is more than just a simple replacement for XML when",
                "you make an AJAX call."
            ],
            "track": "Web APIs"
        },
        {
            "title": "RESTful Security at Work",
            "length": "90 minutes",
            "abstract": [ "You've been working with RESTful Web Services for a few years",
                "now, and you'd like to know if your services are secure."
            ],
            "track": "Web APIs"
        }
    ]
}
```

Characteristics of JSON Arrays

Arrays have the following characteristics:

- Are enclosed within a beginning left brace ([) and an ending right brace (]);
- Consist of comma-separated, ordered values (see the next section);
- Can be empty, [ ];
- Can be nested within other Arrays or Objects; and
- Have indexing that begins at 0 or 1.

JSON Value Types

JSON Value Types represent the Data Types that occur on the righthand side of the colon (:) of a Name/Value Pair.

JSON Value Types include the following:

- object;
- array;
- string;
- number;
- boolean; and
- null.

Previous slides have already covered *Objects* and *Arrays;* now let's focus on the remaining Value Types: *string, number, boolean,* and *null.*

Properties of JSON Strings

The following JSON array contains examples of typical strings:

```
[ "John", "John\t", "\b", "", "\t", "\u004A" ]
```

Strings have the following properties:

- Strings consist of zero or more Unicode characters enclosed in quotation marks ("");
- Strings wrapped in single quotes (') are not valid; and
- Additionally, JSON Strings can contain backslash-escaped characters.

JSON Strings — *Backslash Escape Characters*

Escape Character	Decription
\"	Double quote
\\	Backslash
\/	Forward slash
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Tab
\u	Trailed by four hex digits

JSON Numbers

The following JSON code illustrates permissible numbers:

```
{
    "age": 29, "cost": 299.99,
    "temperature": -10.5, "unitCost": 0.2,
    "speedOfLight": 1.23e11,"speedOfLight2": 1.23e+11,
    "avogadro": 6.023E23, "avogadro2": 6.023E+23,
    "oneHundredth": 10e-3, "oneTenth": 10E-2
}
```

Numbers follow JavaScript's double-precision floating-point format and have the following properties:

- Numbers are always in base 10 (only digits 0–9 are allowed) with no leading zeros.
- Numbers can have a fractional part that starts with a decimal pont (.).

- Numbers can have an exponent of 10, which is represented with the e or E nota tion with a plus or minus sign to indicate positive or negative exponentiation.
- Octal and hexadecimal formats are not supported.
- Unlike JavaScript, numbers can’t have a value of NaN (not a number for invalid numbers) or Infinity.

JSON Boolean Types

The following is an example of Boolean value types in JSON:

```
{
    "isRegistered": true,
    "emailValidated": false
}
```

Booleans have the following properties:

- Booleans can have a value of only true or false.
- The true or false value on the righthand side of the colon(:) is not surrounded by quotes.

The null Special Value in JSON

Although technically not a Value Type, **null** is a special value in JSON. The example blow shows a **null** value for the "line2" key/property.

```
{
    "address": {
        "line1"          : "555 Any Street",
        "line2"          : null,
        "city"           : "Denver",
        "stateOrProvince": "CO",
        "zipOrPostalCode": "80202",
        "country"        : "USA"
    }
}
```

null values have the following characteristics:

- Are not not surrounded by quotes; and
- The true or false value on the righthand side of the colon(:) is not surrounded by quotes.

JSON: Summary of Other Issues

- JSON versions — there will never be another version of the core JSON standard (Douglas Crockford), this applies only to the core JSON data format;
- There are no vommends in a JSON document;
- According to the core JSON specification, **.json** is the standard JSON file type when storing JSON data on filesystems.
- JSON's *Internet Assigned Numbers Authority* (IANA) media (or MIME) type is **application/json**, which can be found at the **IANA Media Types site**.
- Google has published a **JSON Style Guide to support** maintainability and best practices. This guide is extensive,and the most important things (subsections of the Guide) for an API designer and developer are as follows:

    1. Property Names;

2. Date Property Values; and

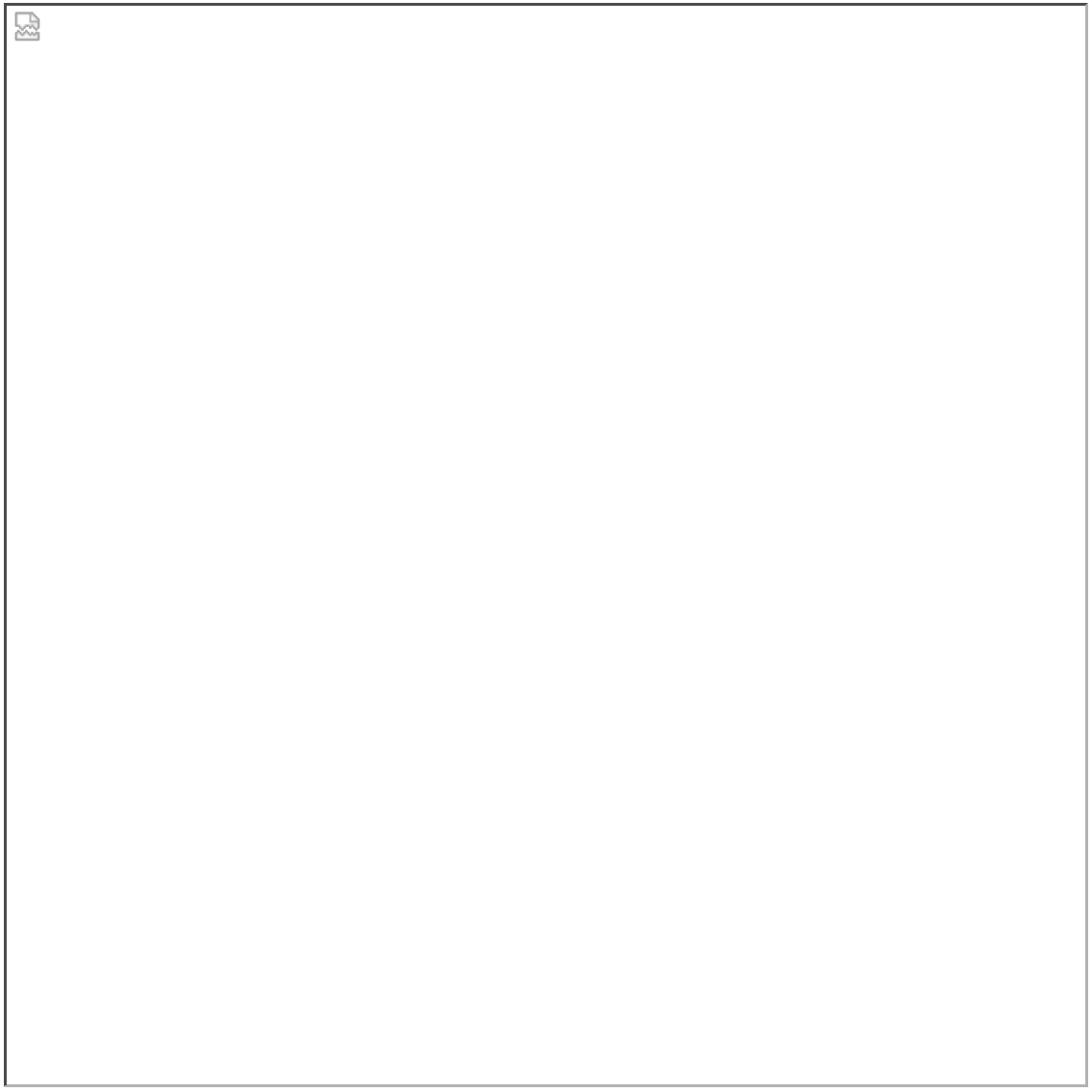
3. Enum Values.

Model JSON Data with JSON Editor Online

- JSON Editor Online (JEO) - https://jsoneditoronline.org/
- A great web-based tool that does the following:
  - *Enables you to model your JSON document as Objects, Arrays, and name/value pairs; and*
  - *Makes it easier to rapidly generate the text for a JSON document in an iterative manner.*
- In addition to the above, JEO provides the following features:
  - *JSON validation;*
  - *JSON pretty-printing;*
  - *Full roundtrip engineering between the model and JSON text;*
  - *Save JSON document to disk; and*
  - *Import JSON document.*

- **Warning**: JSON Editor Online is publicly available, which means that any data you paste into this app is visible to others — *don't use this tool with sensitive information* (personal, proprietary, and so forth).

Example: Speaker data model and speakers.json in JSON Editor Online



The **JSON code** (left) can be generated from the **JSON model** (right) by pressing the arrow point to the left.

Download and examine the file **speaker.json** in **JSON Editor Online**

Create and Deploy a Stub JSON API

Step 1: Set up your JSON development environment

1. Install Node.js and add it to system PATH.
2. Install **json-server** — json-server is a Node.js module, hence why you need to install Node.js first.

   ```
   npm install -g json-server
   ```

   and sun it using the command:

   ```
   json-server -p 5000 ./speakers.json
   ```

   and test your installation by going to http://localhost:5000/speakers
3. Install JSONView (Chrome, Firefox) and Postman.

4. **JSONView pretty-prints** — a JSON extension in Chrome and Firefox. Postman can also run as a standalone GUI application alternative on most major operating systems.

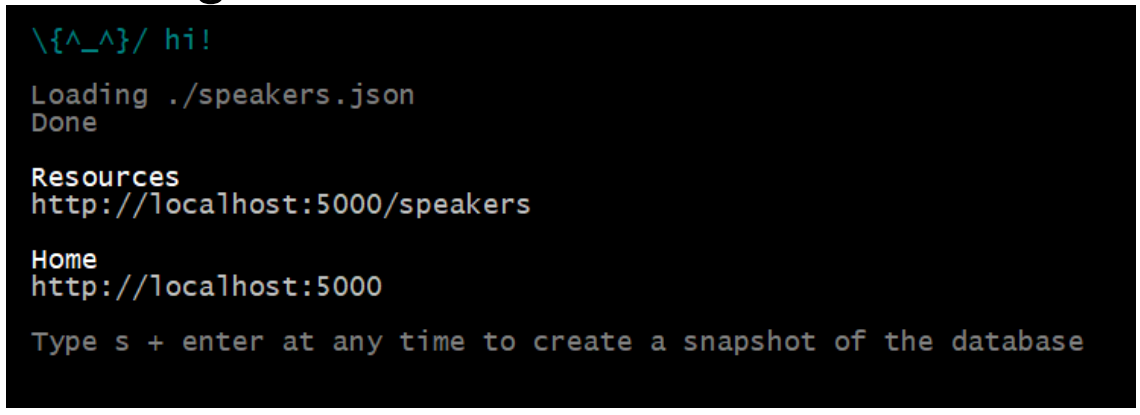
Create and Deploy a Stub JSON API

Step 2: Run the json-server

Open a terminal session and run json-server on port 5000 from your command line:

```
cd PATH/speakers.json
json-server -p 5000 ./speakers.json
```

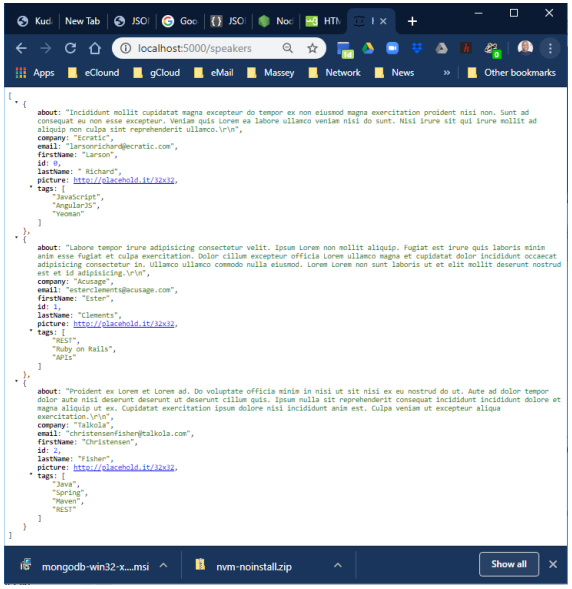
You should see the following:



Create and Deploy a Stub JSON API

Step 3: Access the RESTful JSON API

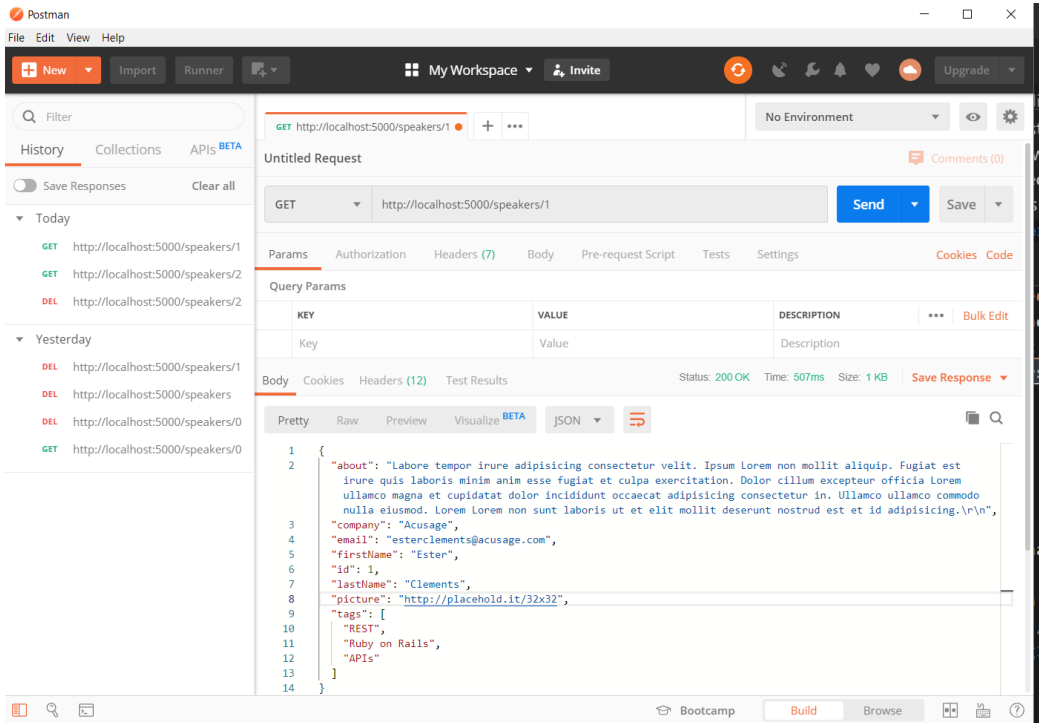
- Visit **http://localhost:5000/speakers** in your Chrome or Firefox browser.
- With JSON pretty-printing provided by the JSONView extension; you should see all the speakers from our Stub JSON API as shown in this screenshot:



- Adding the **ID**number gets the speaker with that id, e.g., **http://localhost:5000/speakers/2**

Using Postman for more JSON API Testing

- The Web browser has limited testing functionality — it can only send HTTP GET requests.
- Postman provides the ability to fully test a RESTful API — it can send **HTTP GET, POST, PUT,** and **DELETE** requests and set **HTTP Headers**.



JSON in JavaScript

JSON Serialization

- Applications need to serialize (or flatten) their information into JSON in order to produce data for other applications in a platform-neutral manner.
- Serialisation is achieved with JSON.stringify()
- JSON.stringify() serializes from JSON Object to JSON document.

JSON De-serialization

- An application must also be able to deserialize (or unflatten) JSON data consumed from external sources into data structures for use by that application.

- De-serialisation is achieved with JSON.parse()
- JSON.parse() deserializes to JSON Object from the JSON document.

The JSON Stringifier/Parser Object

- Was originally developed by Douglas Crockford;
- Been part of the JavaScript library as of ECMAScript 5 in 2009;
- Can’t be instantiated; and
- Has no other functionality.

Example 1: JSON Serialization with Simple JavaScript Data Types

```
// filename: basic-data-types-stringify.js
function simpleTypeStringify(){
    let age = 39; // Integer
    let serialisedAge = 'age = ' + JSON.stringify(age) + '\n';
    console.log(serialisedAge); alert(serialisedAge);

    let fullName = 'Jacinda Adern'; // String
    let serialisedFullName = 'fullName = ' +
        JSON.stringify(fullName) + '\n';
    console.log(serialisedFullName); alert(serialisedFullName);

    let tags = ['json', 'rest', 'api', 'oauth']; // Array
    let serialisedTagsArray = 'tags = ' + JSON.stringify(tags) + '\n';
    console.log(serialisedTagsArray); alert(serialisedTagsArray);

    let reqistered = true; // Boolean
    let serialsedBoolean = 'registered = ' +
        JSON.stringify(reqistered) + '\n';
    console.log(serialsedBoolean); alert(serialsedBoolean);

    let speaker = {
            firstName: 'Jacinda',
            lastName: 'Adern',
            email: 'jacinda@adern.com',
            about: 'Incididunt mollit cupidatat
                magna excepteur do tempor ex non ...',
            company: 'Ecratic',
            tags: ['json', 'rest', 'api', 'oauth'],
            registered: true
        };
```

```
    let serialisedSpeaker = 'speaker = ' + JSON.stringify(speaker);
    console.log(serialisedSpeaker); alert(serialisedSpeaker);
}
```

Click to run example

View the results in both the alerts and the Console of the Developer Tools for your browser.

Example 2: Stringifying with pretty-print

JSON.stringify() doesn't do anything with the scalar types (Number, String, Boolean).
However, with the speaker Object, JSON.stringify() initially generates a valid, yet unattractive, JSON String.
JSON.stringify() has other parameters that enhance serialization.

In the Mozilla Developer Network (MDN) JavaScript Guide, JSON.stringify() method signature is as follows:

```
JSON.stringify(value[, replacer [, space]])
```

The parameter list is as follows:

value (required)

The JavaScript value to serialize.

replacer (optional)

Either a function or an array. If a function is provided, the stringify() method invokes the replacer function for each key/value pair in an Object.

space (optional)

Indentation—either a Number or String. If a Number is used, this value specifies the number of spaces used for each indentation level.

Example 2: Stringifying with pretty-print

```
// filename: obj-literal-stringify-params.js
function serializeSpeaker(key, value) {
    return (typeof value === 'string' || Array.isArray(value)) ? undefined : value;
}

let speaker = {
    firstName: 'Jacinda',
    lastName: 'Adern',
    email: 'Jacinda@adern.com',
    about: 'Incididunt mollit cupidatat magna excepteur do tempor ex non ...',
    company: 'Adern Ltd',
    tags: ['json', 'rest', 'api', 'oauth'],
    registered: true
};

function prettyPrint(){
    // Pretty Print.
    let serialisedSpeaker = 'Speaker (pretty print):\n' + JSON.stringify(speaker, null, 2) + '\n';
    console.log(serialisedSpeaker); alert(serialisedSpeaker);

    // Pretty print and filter out Strings and Arrays.
    serialisedSpeaker = 'Speaker without Strings and Arrays:\n' +
    JSON.stringify(speaker, serializeSpeaker, 2);
    console.log(serialisedSpeaker); alert(serialisedSpeaker);
}
```

Click to run example

View the results in both the **alert()** and the **console.log()** of the Developer Tools for your browser.

The first **JSON.stringify()** call pretty-prints the JSON output with an indentation level of 2. The second call uses the **serializeSpeaker()** function as a **replacer** (JavaScript functions are treated as expressions and can be passed as parameters). **serializeSpeaker()** checks the type of each value and returns undefined for *Strings* and *Arrays*. Otherwise, this function returns the value "as is".

Example 3: JSON Deserialization Using eval()

- Originally, JavaScript developers used the **eval()** function to parse JSON.
- **eval()** takes a String parameter that could be a JavaScript expression, a statement, or a sequence of statements.
- Consider the following example:

```
// filename: eval-parse-kd.js

function parseEval() {
    let x = '{ "sessionDate": "2019-10-06T13:30:00.000Z" }';

    let parseWithEval = 'Parse with eval(): ' + eval('(' + x + ')').sessionDate + '\n';
    console.log(parseEval); alert(parseWithEval);

    let parseWithJSON = 'Parse with JSON.parse(): ' + JSON.parse(x).sessionDate;
    console.log(parseWithJSON); alert(parseWithJSON);
}
```

Click to run example

View the results in both the **alert()** and the **console.log()** of the Developer Tools for your browser.

In this case, both **eval()** and **JSON.parse()** work the same and parse the date properly. *So what's the problem?*

Example 3: JSON Deserialization Using eval()

Let's look at another example with a JavaScript statement embedded in the String:

```
function parseWithEval2(){
    let x = '{ "sessionDate": new Date() }';

    let parsedWithEval = 'Parse with eval(): ' + eval('(' + x + ')').sessionDate + '\n';
    console.log(parsedWithEval); alert(parsedWithEval);

    let parsedWithJSON = 'Parse with JSON.parse(): ' + JSON.parse(x).sessionDate;
    console.log(parsedWithJSON); alert(parsedWithJSON);
}
```

Click to run example

View the results in both the **alert()** and the **console.log()** of the Developer Tools for your browser.

- You should observe the following from this example:

- *We passed in text that contains a JavaScript statement,* ***new Date()****, and* ***eval()*** *executes that statement.*
- *Meanwhile,* ***JSON.parse()*** *correctly rejects the text as invalid JSON.*

- **Security Vulnerability:** Although we passed in only a fairly innocuous statement to create a Date, *someone else could pass in malicious code and* ***eval()*** *would still execute it.*
  - *Even though* ***eval()*** *can be used to parse JSON, it is considered a bad/unsafe practice because it opens the door to any valid JavaScript expression, leaving your application vulnerable to attacks.*
  - *Because of this security issue, the eval() function has been deprecated (for parsing JSON) in favor of JSON.parse()*

Example 4: JSON Deserialization with an Object and JSON.parse()

The following code uses JSON.parse() to deserialize a JSON String into a speaker JavaScript Object:

```
function deserialiseSpeaker() {
    let json = '{' +  // Multi-line JSON string.
        '"firstName": "Jacinda",' +
        '"lastName": "Adern",' +
        '"email": "Jacinda@adern.com",' +
        '"about": "Incididunt mollit cupidatat magna excepteur do tempor ex non ...",' +
        '"company": "Adern Ltd",' +
        '"tags": [' + '"json",' + '"rest",' + '"api",' + '"oauth"' + '],' +
        '"registered": true' +
    '}';

    // De-serialize JSON string into speaker object.
    let speaker = JSON.parse(json);

    // Print 2nd speaker object.
    let firstName = 'speaker.firstName = ' + speaker.firstName;
    console.log(firstName); alert(firstName);
}
```

Click to run example

View the results in both the **alert()** and the **console.log()** of the Developer Tools for your browser.

Note:JSON.parse() takes a JSON String as input and parses it into a fully functional Java- Script Object. We're now able to access the speaker Object's data members.

Topic Summary

In this topic, we have covered the following:

- The basics of JSON;
- Modelling JSON data with JSON Editor Online;
- Deploying and testing a Stub JSON API;
- Simple conversion between JavaScript and JSON;

We've covered "just enough" of several technologies for you to understand core concepts and build simple applications. But we've just scratched the surface of JavaScript, Node.js,and related technologies

Core Reference for this Topic

Main resource for this topic is the book:

Tom Marrs (2017). *JSON at Work: Practical Data Integration for the Web.* Chapters 1 and 2: pp. 3--55. O'Reilly Media, Inc. ISBN: 978-1-449-35832-7