JSON in a Nutshell

JavaScript Object Notation

Salim Bouzitouna & Reda Bendraou

JSON => JavaScript Object Notation

- A Subset of ECMA-262 Third Edition (a standard for script prog. Languages)
  - Developed by **Douglas Crockford**
- Language Independent.
- Text-based.
- Light-weight.
- Easy to parse.

JSON, first Example (Google Map markers)

```
{
  "markers": [
    {
      "name": "Rixos The Palm Dubai",
      "position": [25.1212, 55.1535],
    },
    {
      "name": "Shangri-La Hotel",
      "location": [25.2084, 55.2719]
    },
    {
      "name": "Grand Hyatt",
      "location": [25.2285, 55.3273]
    }
  ]
}
```

JSON

avaScript Object Notation

- Why It’s called JavaScript Object Notation? =>because it is based on the Object Literal notation of JavaScript:

```
var myObject = {
        myString: "my string value",
        myInt: 2,
        myBool: false
}
```

- A JSON string can be converted directly to a JavaScript Object

Character Encoding / Mime Type

- Default: UTF-8.
- UTF-16 and UTF-32 are allowed.
- MIME Media Type =>`application/json`

Versionless

- JSON has no version number.

- No revisions to the JSON grammar are anticipated.

- JSON is very stable.

JSON Is Not...

- a document format.

- a markup language.

- Dependent on the JavaScript Language.

- a general serialization format.
  - No cyclical/recurring structures.
  - No invisible structures.
  - No functions.

Data Interchange

- JSON is a simple, common representation of data.
- Communication between servers and browser clients.
- Communication between peers.
- Language independent data interchange.

JSON: well-suited for data transfer

- It's simultaneously human- and machine-readable format;
- It has support for Unicode, allowing almost any information in any human language t be communicated;
- The self-documenting format that describes structure and field names as well as specific values;
- The strict syntax and parsing requirements that allow the necessary parsing algorith to remain simple, efficient, and consistent;
- The ability to represent the most general computer science data structures: records, lists and trees.

JSON Data Types

JSON Data Types

- Strings
- Numbers
- Booleans
- Objects
- Arrays
- **`null`**

Strings

- Sequence of 0 or more Unicode characters
- No separate character type
  - A character is represented as a string with a length of 1
- Wrapped in "double quotes"
- Backslash escapement

```
{
"address": "123 fake st",
"city": "Portland",
"state": "Oregon"
}
```

Numbers

- Integer
- Real
- Scientific
- No octal or hex
- No **`NaN`** or **`Infinity`**
    - Use **`null`** instead

JSON Number: Example

- In JSON, Integers, Decimals, and Exponents are all supported **number** types. This also includes negative numbers. So, for "zipcode", we have an integer. "latitude" and "longitude" are both decimals. "longitude" is a negative number. "largenumber" is an exponent.

```
{
"address": "123 fake st",
"city": "Portland",
"state": "Oregon",
"zipcode": 12345,
"latitude": 45.5339475,
"longitude": -122.7040751,
"largenumber": 16176633332384900000,
"residential": false,
"business": null
}
```

Booleans & null

- A boolean can either be true or false. It is important that your true or false in JSON starts with a lowercase letter, and not uppercase. It must be all lowercase characters.
  - **`true`**
  - **`False`**
    - "residential": false

- null =>A value that isn't anything. In JSON, null must always be all lowercase characters.
    - "business": null

Object

- Objects are unordered containers of key/value pairs
- Objects are wrapped in { }
- , separates key/value pairs
- : separates keys and values
- Keys are strings
- Values are JSON values

Object

```
{"name":"Jack B. Nimble","at large":
true,"grade":"A","level":3,
"format":{"type":"rect","width":1920,
"height":1080,"interlace":false,
"framerate":24}}
```

Object (with a better formatting)

```
{
    "name":         "Jack B. Nimble",
    "at large": true,
    "grade":        "A",
    "format": {
        "type":           "rect",
        "width":          1920,
        "height":         1080,
        "interlace": false,
        "framerate": 24
    }
}
```

Array

- Arrays are ordered sequences of values
- Arrays are wrapped in **[]**
- , separates values
- JSON does not talk about indexing.
    - An implementation can start array indexing at 0 or 1.

Examples

```
["Sunday", "Monday",
  "Tuesday", "Wednesday",
  "Thursday", "Friday",
  "Saturday"]
```

```
[
     [0, -1, 0],
     [1, 0, 0],
     [0, 0, 1]
]
```

JSON Array: Examples

ay of String

```
udents": [
ne Thomas",
b Roberts",
bert Bobert",
omas Janerson"
```

Array of Numbers

```
{
"scores": [
93.5,
66.7,
87.6,
92
]
}
```

Array of Boolean

```
{
"answers": [
true,
false,
false,
true
]
}
```

Array of Objects

```
{
"test": [
{
"question": "The sky is blue",
"answer": true
},
{
"question": "The earth is flat.",
"answer": false
},
{
"question": "A cat is a dog.",
"answer": false
}
]
}
```

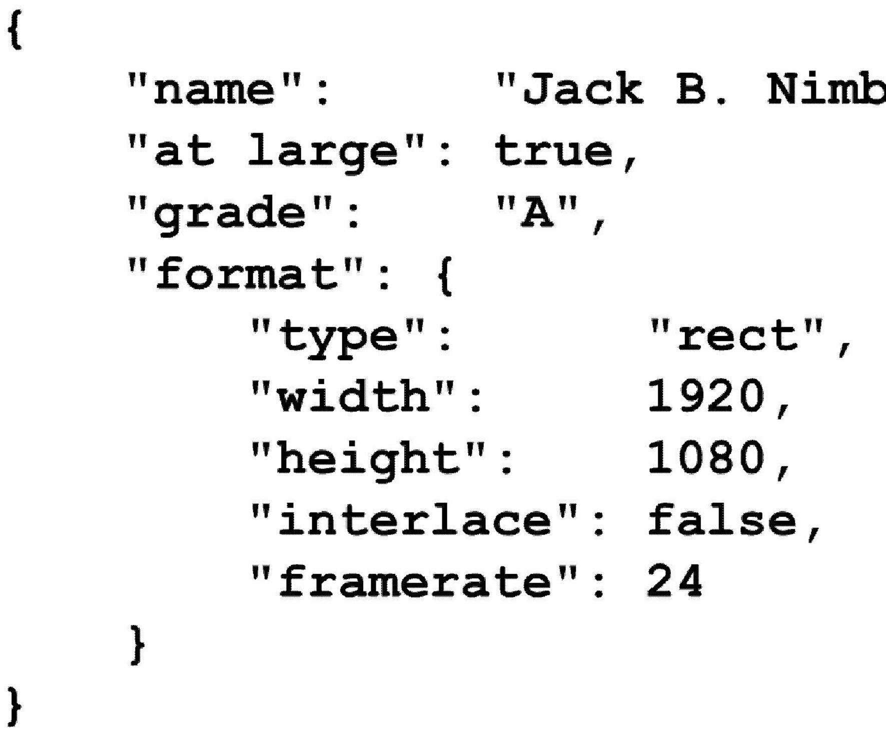
Array of Arrays

```
{
        "tests": [
                [
                        true
                        fals
                        fals
                        fals
                ],
                [
                        true
                        true
                        true
                        true
                        fals
                ],
                [
                        tru
                        fals
                        true
                ]
        ]
}
```

Exercise : XML to JSON

- Convert this XML Structure to JSON

```
<Contacts>
<Contact type="Personne" id="a4" conjoint="a2">
                        <nom>Forestier</nom>
                        <prenom>Madeleine</prenom>
                        <adresses>
                                    <adresse type="domicile">
                                                <rue>rue Fontaine</rue>
                                                <numero>12</numero>
                                                <ville>Paris</ville>
                                                <code_postal>75009</code_postal>
                                                <pays>France</pays>
                                    </adresse>
                        </adresses>
                        <email>madeleine.forestier@belami.fr</email>
                        <telephones>
                                    <telephone type="domicile"  de="19h00" a="23h00">0144274428</telephone>
            <telephone type="mobile"  de="08h00" a="23h00">0644274428</telephone>
                        </telephones>
            </Contact>
</Contacts>
```



```
{
    "name":        "Jack B. Nimb
    "at large": true,
    "grade":       "A",
    "format": {
        "type":        "rect",
        "width":       1920,
        "height":      1080,
        "interlace": false,
        "framerate": 24
    }
}
```

Exercise: XML to JSON

```
{
 "Contacts":{
  "Contact":{
   "-type":"Personne",
   "-id":"a4",
   "-conjoint":"a2",
   "nom":"Forestier",
   "prenom":"Madeleine",
   "adresses":{
    "adresse":{
     "-type":"domicile",
     "rue":"rue Fontaine",
     "numero":"12",
     "ville":"Paris",
     "code_postal":"75009",
     "pays":"France"
    }
   },
   "email":"madeleine.forestier@belami.fr",
   "telephones":{
    "telephone":[
     {
      "-type":"domicile",
      "-de":"19h00",
      "-a":"23h00",
      "#text":"0144274428"
     },
     {
      "-type":"mobile",
      "-de":"08h00",
      "-a":"23h00",
      "#text":"0644274428"
     }
    ]
   }
  }
 }
}
```

JSON Vs.XML

JSON Vs. XML

- Please, stop comparing them! Each language can be appropriate for a given use scenario

=> http://www.yegor256.com/2015/11/16/json-vs-xml.html

```
loyees":[
 "firstName":"John", "lastName":"Doe" },
 "firstName":"Anna", "lastName":"Smith" },
 "firstName":"Peter", "lastName":"Jones" }
```

```
<employees>
    <employee>
        <firstName>John</firstName> <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName> <lastName>Smith</lastName
    </employee>
    <employee>
        <firstName>Peter</firstName> <lastName>Jones</lastNa
    </employee>
</employees>
```

JSON Vs. XML

JSON	XML
JSON stands for JavaScript Object Notation.	XML stands for eXtensible Markup Language.
JSON is simple to read and write.	XML is less simple than JSON.
JSON is easy to learn.	XML is less easy than JSON.
JSON is data-oriented.	XML is document-oriented.
JSON doesn't provide display capabilities.	XML provides the capability to display data because it is a markup language.
JSON supports array.	XML doesn't support array.
JSON is less secured than XML.	XML is more secured.
JSON can be parsed by a standard JavaScript function. Very practical	XML has to be parsed with an XML parser. This is harder
JSON supports only text and number data type.	XML support many data types such as text, number, images, charts, graphs etc. Moreover, XML offeres options for transferring the format or structure of the data with actual data.

JSON Vs. XML...Both are:

- "self describing" (human readable)
- Hierarchical (values within values)
- Can be parsed and used by lots of programming language
- Can be fetched with an XMLHttpRequest

JSON Vs. XML…XML Strength

- Attributes and namespaces
- Xpath
- Schema
- Display formatting with CSS or XSL
- We will see later some JSON initiatives to overcome this!

JSON Vs. XML…Who is using it?

- The following major public APIs uses XML only: Amazon Product Advertising API.

- The following major APIs use JSON only: Facebook Graph API, Google Maps API, Twitter API, AccuWeather API, Pinterest API, Reddit API, Foursquare API.

- The following major APIs use both XML and JSON: Google Cloud Storage, Linkedin API, Flickr API

Source : http://www.cs.tufts.edu/comp/150IDS/final_papers/tstras01.1/FinalReport/FinalReport.html

Arguments against JSON

- JSON Doesn't Have Namespaces.
- JSON Has No Validator.
- JSON Is Not Extensible.
- JSON Is Not XML.
- No way to specify comments (should be added as an element of the object, ex. “comment”:”bla bla”)

JSON Doesn't Have Namespaces

- Every object is a namespace. Its set of keys is independent of all other objects, even exclusive of nesting.

- JSON uses context to avoid ambiguity, just as programming languages do.

Namespace

- http://www.w3c.org/TR/REC-xml-names/
- In this example, there are three occurrences of the name title within the markup, and the name alone clearly provides insufficient information to allow correct processing by a software module.

```
<section>
    <title>Book-Signing Event</title>
    <signing>
        <author title="Mr" name="Vikram Seth" />
        <book title="A Suitable Boy" price="$22.95" />
    </signing>
    <signing>
        <author title="Dr" name="Oliver Sacks" />
        <book title="The Island of the Color-Blind"
              price="$12.95" />
    </signing>
</section>
```

Namespace

```
{"section":
    "title": "Book-Signing Event",
    "signing": [
        {
            "author": { "title": "Mr", "name": "Vikram Seth" },
            "book": { "title": "A Suitable Boy",
                      "price": "$22.95" }
        }, {
            "author": { "title": "Dr", "name": "Oliver Sacks" },
            "book": { "title": "The Island of the Color-Blind",
                      "price": "$12.95" }
        }
    ]
}}
```

- `section.title`
- `section.signing[0].author.title`
- `section.signing[1].book.title`

JSON Has No Validator

- Being well-formed and valid is not the same as being correct and relevant.

- Ultimately, every application is responsible for validating its inputs. This cannot be delegated.

- A YAML validator can be used.

- JSON Answer=> **JSON Schema**

JSON Schema

- JSON Schema is a vocabulary that allows you annotate and validate JSON files

- Describes your existing data format

- Clear, human- and machine-readable documentation

- Complete structural validation, useful for
  - automated testing
  - validating client-submitted data

JSON Schema

- The latest Internet-Drafts at the IETF are the draft-wright-json-schema*-01 documents, which correspond to the draft-06 meta-schemas.

- Published on 2017-04-15.

- JSON Schema is not really adopted right now and having it as a Draft version gives a clear indication that may be this is too early to invest in learning this specification=> see example next slide to have an Idea

- http://json-schema.org/documentation.html

JSON Schema: Example

Schema: Type

```
'type": "object",
'properties": {
 "first_name": { "type":
 "string" },
 "last_name": { "type":
 "string" },
 "birthday": { "type":
 "string", "format": "date-
 time" },  "address": {
  "type":
  "object"
```

JSON file: Instance

```
{
  "first_name": "George",
  "last_name": "Washington",
  "birthday": "22-02-1732",
  "address": {
    "street_address": "3200 Mount Vernon Memorial Highwa
    "city": "Mount Vernon",
    "state": "Virginia",
    "country": "United States"
  }
}
```

JSON Schema Example

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": ""http://foo.bar/schemas/address.json""
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",

  "properties": {

    "id": {

      "description": "The unique identifier for a product",
      "type": "integer"
    },

    "name": {
      "description": "Name of the product",
      "type": "string"
    },

    "price": {
      "type": "number",
      "minimum": 0,
      "exclusiveMinimum": true
    }
  },

  "required": ["id", "name", "price"]
}
```

$schema indicates the JSON Schema Draft version used to specify this schema

title gives a title to the schema

description describes the content of the schema

type indicates that you are defining a JSON object

properties defines the properties of the object

required to indicate the set of mandatory properties

minimum min value expected

exclusiveMinimum means > than the min value indicated in the minimum property.

JSON Shema Examples: String

Declaration	Correct	wrong
{ "type": "string", "minLength": 2, "maxLength": 3 }	"AB"	"ABBB" "A"
{ "type": "string", "pattern": "^(\\([0-9]{3}\\))?[0-9]{3}-[0-9]{4}$" }	"555-1212" "(888)555-1212"	"(800)FLOWERS"

JSON Shema Examples: number

Declaration	Correct	wrong
{ "type": "number" }	42 42,0	"42"
{ "type" : "number", "multipleOf" : 10 }	10 20	25
{ "type": "number", "minimum": 0, "maximum": 100, "exclusiveMaximum": true }	0 10 99	100 101

JSON Shema Examples: Object

Declaration	Correct	wrong
{ "key" : "value", "another_key" : "another_value" }	{ "key" : "value", "another_key" : "another_value" }	"{ 0.01 : "cm" 1 : "m", 1000 : "km" } "Not an object"

{
"type": "object",
"properties": {
"number": { "type": "number" },
"street_name": { "type": "string" },
"street_type": { "type": "string",
"enum": ["Street", "Avenue",
"Boulevard"]
} } }
	{ "number": 1600,
	"street_name": "Pennsylvania",
	"street_type": "Avenue" }
		{ "number": "1600", "street_name": "Pennsylvania", "street_type": "Avenue" }

{
"type": "object",
"properties": {
"name": { "type": "string" },
"email": { "type": "string" },
"address": { "type": "string" },
"telephone": { "type": "string" }
},
"required": ["name", "email"]
}
	{ "name": "William Shakespeare",
	"email": "bill@stratford-upon-avon.co.uk" }
		{ "name": "William Shakespeare",
		"address": "Henley Street,
		Stratford-upon-Avon,
		Warwickshire, England", }

JSON Shema Examples: Array

Declaration	Correct	wrong
{ "type": "array", "items": { "type": "number" } }	[1, 2, 3, 4, 5]	[1, 2, "3", 4, 5] "

{
"type": "array",
"items": [
{
"type": "number"
},
{
"type": "string"
},
{
"type": "string",
"enum": ["Street", "Avenue","Boulevard"]
},
{
"type": "string",
"enum": ["NW", "NE", "SW", "SE"]
}
]
}
	[1600, "Pennsylvania", "Avenue", "NW"]
	[10, "Downing", "Street"]
		["Palais de l'Élysée"]
		[24, "Sussex", "Drive"]

JSON Shema Examples

Type	Declaration	Correct	wrong
Booléen	{ "type": "boolean" }	true false	0 1
null	{ "type": "null" }	null	False 0

JSON is Not Extensible

- It does not need to be.
- It can represent any non-recurrent data structure as is.
- JSON is flexible. New fields can be added to existing structures without obsoleting existing programs.

JSonPath

JSonPath

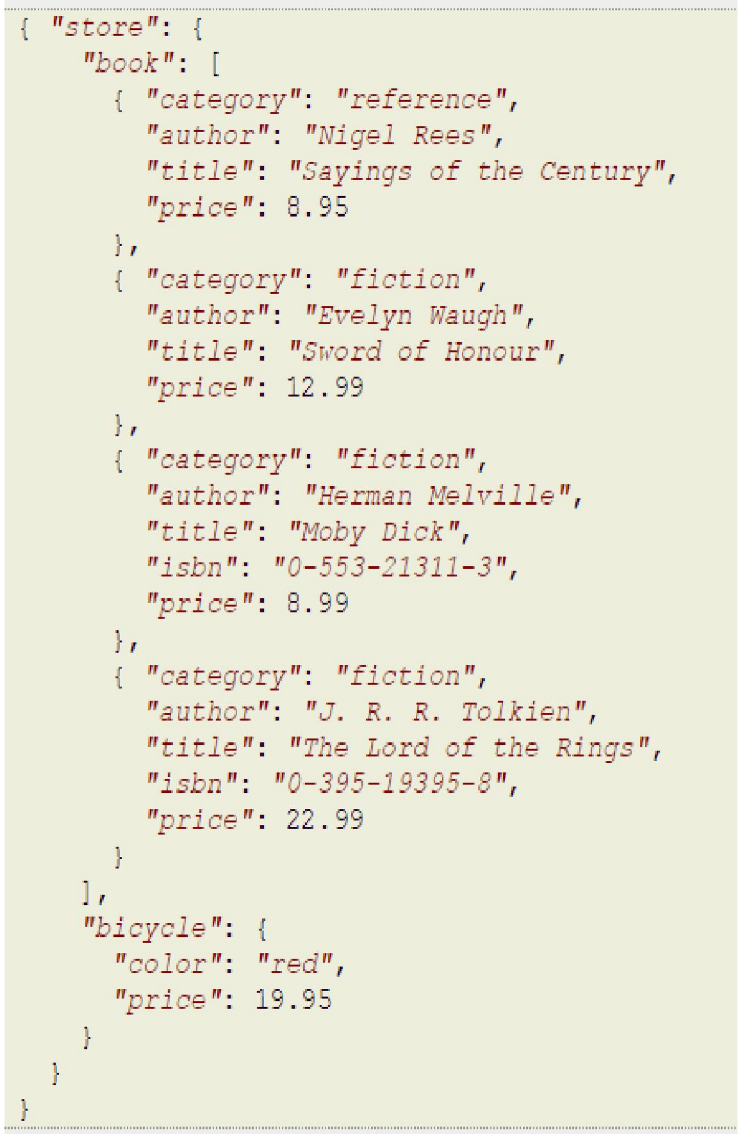
- **JsonPath** = equivalent to Xpath in XML
-
  - ❑ **Only for extracting data, doesn’t alter the JSON file**
  - ❑ **Simple and easy to learn**

Operateurs JsonPath

XPath	JSONPath	Description
/	$	the root object/element
.	@	the current object/element
./.	. or []	child operator
..	n/a	parent operator
//	..	recursive descent. JSONPath borrows this syntax from E4X.
*	*	wildcard. All objects/elements regardless their names.
@	n/a	attribute access. JSON structures don't have attributes.
[]	[]	Child operator or array index. This operator can used to select a field that may contain special characters that need to be quoted.
|	[,]	Union operator in XPath results in a combination of node sets. JSONPath allows alternate names or array indices as a set.
n/a	[start:end:step]	array slice operator borrowed from ES4.
[]	?()	applies a filter (script) expression.
n/a	()	script expression, using the underlying script engine.
()	n/a	grouping in Xpath.

Syntaxe (1)

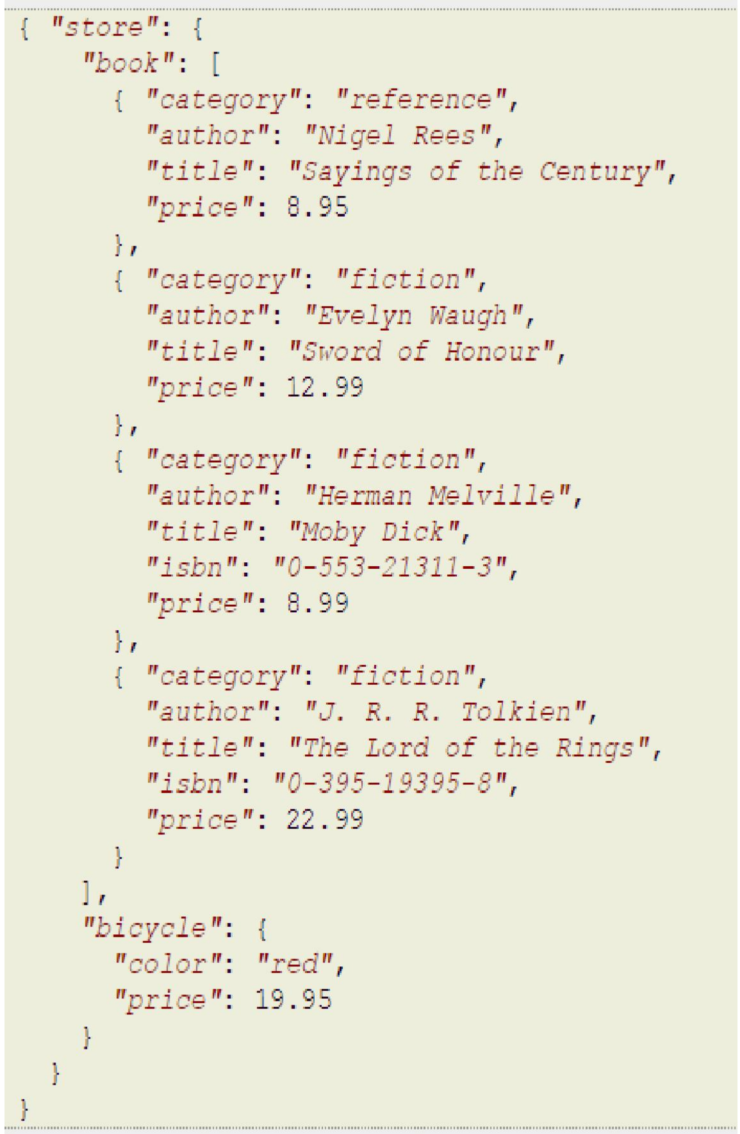
XPath	JSONPath	Result
/store/book/author	**$.store.book[*].author**	the authors of all books in the store
//author	**$..author**	all authors
/store/*	**$.store.***	all things in store, which are some books and a red bicycle.
/store//price	$.store..price	the price of everything in the store.



```
{ "store": {
    "book": [
      { "category": "reference",
        "author": "Nigel Rees",
        "title": "Sayings of the Century",
        "price": 8.95
      },
      { "category": "fiction",
        "author": "Evelyn Waugh",
        "title": "Sword of Honour",
        "price": 12.99
      },
      { "category": "fiction",
        "author": "Herman Melville",
        "title": "Moby Dick",
        "isbn": "0-553-21311-3",
        "price": 8.99
      },
      { "category": "fiction",
        "author": "J. R. R. Tolkien",
        "title": "The Lord of the Rings",
        "isbn": "0-395-19395-8",
        "price": 22.99
      }
    ],
    "bicycle": {
      "color": "red",
      "price": 19.95
    }
  }
}
```

Syntaxe (2)

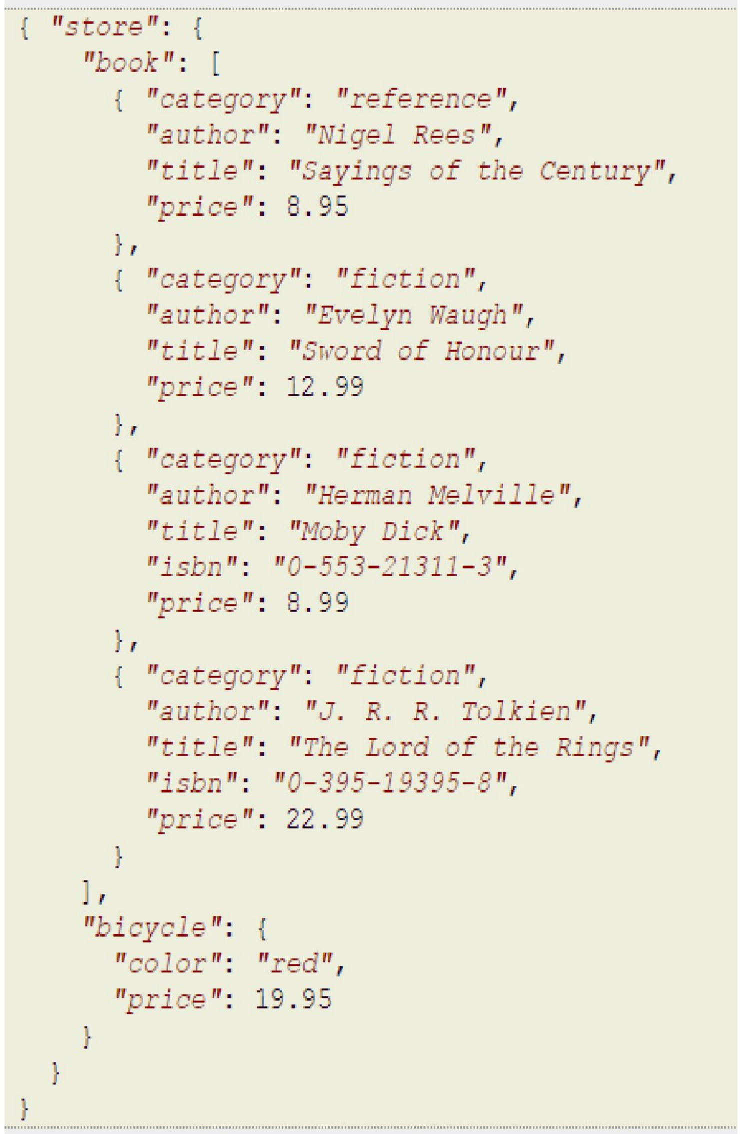
XPath	JSONPath	Result
//book[3]	**$..book[2]**	the third book
/store//price	**$.store..price**	the price of everything in the store.
//*	**$..***	all Elements in XML document. All members of JSON structure.



```
{ "store": {
    "book": [
      { "category": "reference",
        "author": "Nigel Rees",
        "title": "Sayings of the Century",
        "price": 8.95
      },
      { "category": "fiction",
        "author": "Evelyn Waugh",
        "title": "Sword of Honour",
        "price": 12.99
      },
      { "category": "fiction",
        "author": "Herman Melville",
        "title": "Moby Dick",
        "isbn": "0-553-21311-3",
        "price": 8.99
      },
      { "category": "fiction",
        "author": "J. R. R. Tolkien",
        "title": "The Lord of the Rings",
        "isbn": "0-395-19395-8",
        "price": 22.99
      }
    ],
    "bicycle": {
      "color": "red",
      "price": 19.95
    }
  }
}
```

Syntaxe (3)

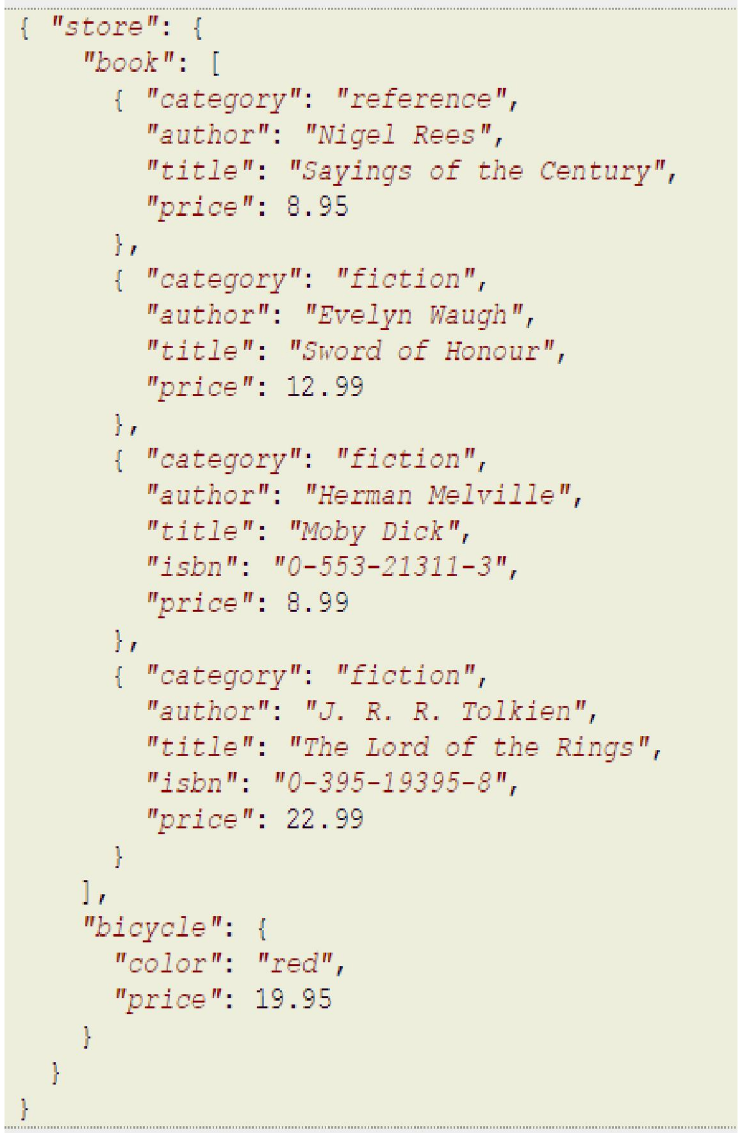
XPath	JSONPath	Result
ook[last()]	$..book[(@.length-1)] $..book[-1:]	the last book in order.
ook[position()<3]	$..book[0,1] $..book[:2]	the first two books
ook[isbn]	$..book[?(@.isbn)]	filter all books with isbn number
ook[price<10]	$..book[?(@.price<10)]	filter all books cheapier than 10



```
{ "store": {
    "book": [
      { "category": "reference",
        "author": "Nigel Rees",
        "title": "Sayings of the Century",
        "price": 8.95
      },
      { "category": "fiction",
        "author": "Evelyn Waugh",
        "title": "Sword of Honour",
        "price": 12.99
      },
      { "category": "fiction",
        "author": "Herman Melville",
        "title": "Moby Dick",
        "isbn": "0-553-21311-3",
        "price": 8.99
      },
      { "category": "fiction",
        "author": "J. R. R. Tolkien",
        "title": "The Lord of the Rings",
        "isbn": "0-395-19395-8",
        "price": 22.99
      }
    ],
    "bicycle": {
      "color": "red",
      "price": 19.95
    }
  }
}
```

Syntaxe (4)

XPath	JSONPath	Result
ook[last()]	**$..book[(@.length-1)]**	the last book in order.
book[first()] | book[last()]	**$..book[0,-1:]**	the first and the last book in order.
book[position() mod =0]	**$..book[0::2]**	the last book in order.



```
{ "store": {
    "book": [
      { "category": "reference",
        "author": "Nigel Rees",
        "title": "Sayings of the Century",
        "price": 8.95
      },
      { "category": "fiction",
        "author": "Evelyn Waugh",
        "title": "Sword of Honour",
        "price": 12.99
      },
      { "category": "fiction",
        "author": "Herman Melville",
        "title": "Moby Dick",
        "isbn": "0-553-21311-3",
        "price": 8.99
      },
      { "category": "fiction",
        "author": "J. R. R. Tolkien",
        "title": "The Lord of the Rings",
        "isbn": "0-395-19395-8",
        "price": 22.99
      }
    ],
    "bicycle": {
      "color": "red",
      "price": 19.95
    }
  }
}
```

JSON Tools

JSON Tools

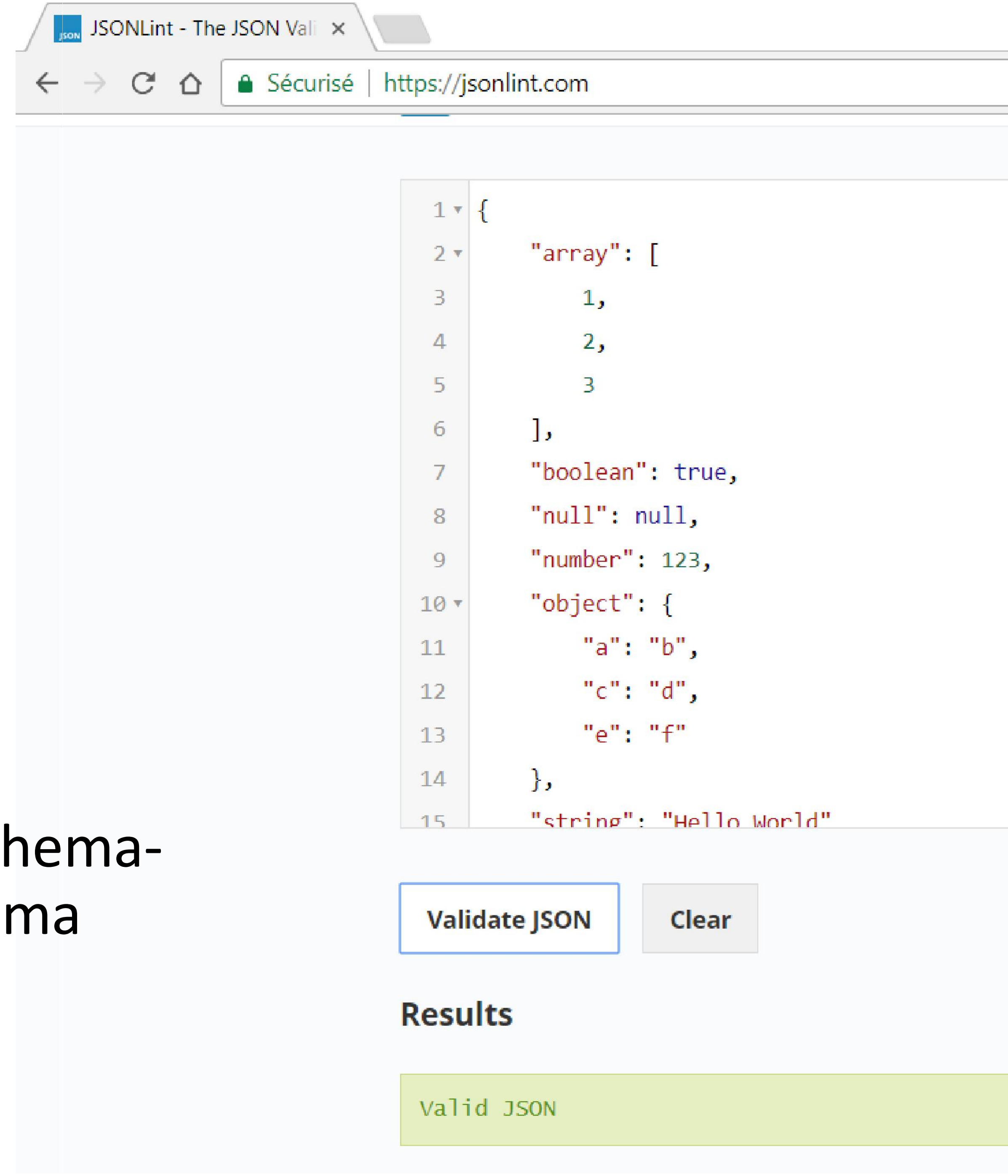
- For Debugging, any extension/pluging/tool that comes with popular Web Browsers (exp. Firebug for Mozilla)

- Mainly will depend in which context you are using JSon (Ajax, REST API, etc)

JSON Tools

- For validating your JSON files JSONLint

Other tools: WJElement (C), json-schema-validator (Java), Json.NET, json-schema (Python), php-json-schema (PHP),

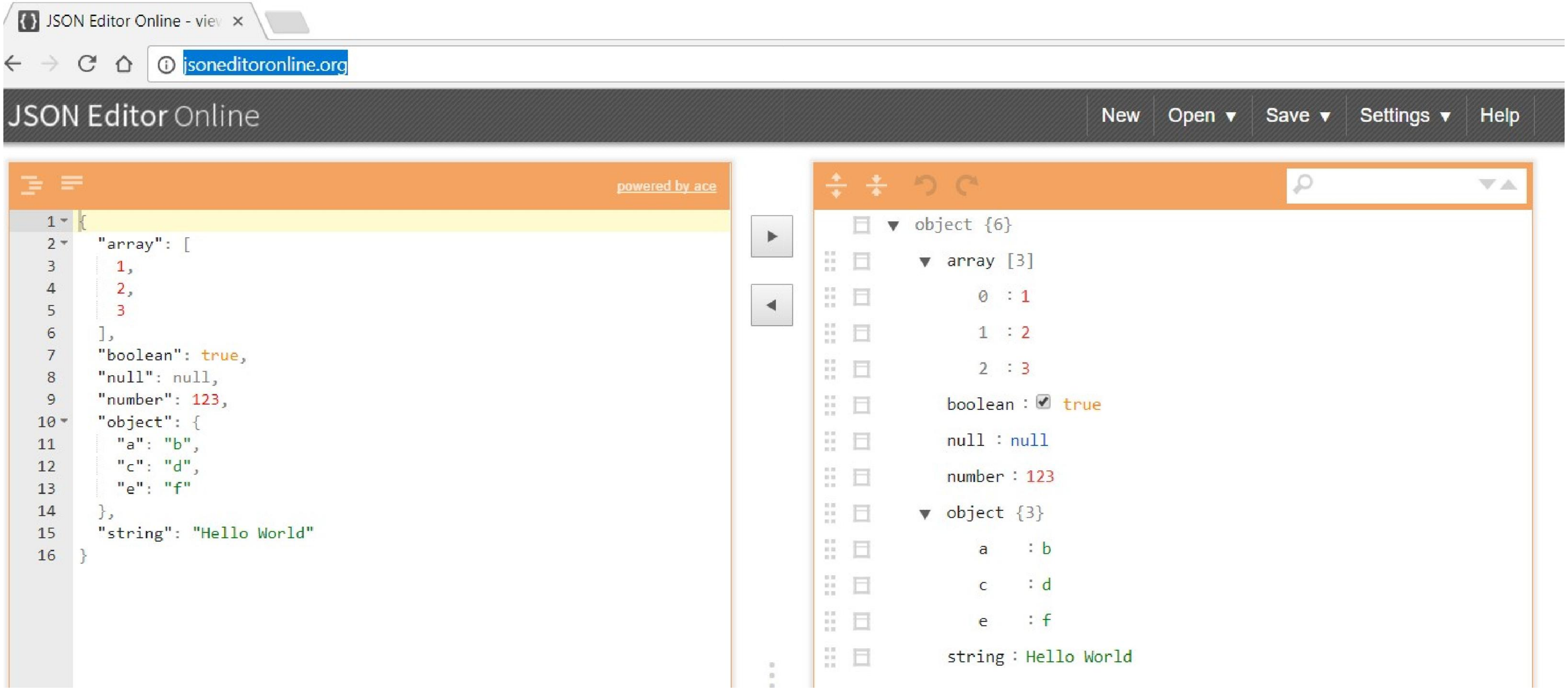


```
{
    "array": [
        1,
        2,
        3
    ],
    "boolean": true,
    "null": null,
    "number": 123,
    "object": {
        "a": "b",
        "c": "d",
        "e": "f"
    },
    "string": "Hello World"
```

```
Valid JSON
```

JSON online Editor

- For a tree view, for formatting your JSon files=> online editor: http://jsoneditoronline.org/



```
{
  "array": [
    1,
    2,
    3
  ],
  "boolean": true,
  "null": null,
  "number": 123,
  "object": {
    "a": "b",
    "c": "d",
    "e": "f"
  },
  "string": "Hello World"
}
```

```
▼ object {6}
    ▼ array [3]
        0  : 1
        1  : 2
        2  : 3
    boolean : ☑ true
    null : null
    number : 123
    ▼ object {3}
        a    : b
        c    : d
        e    : f
    string : Hello World
```