JSON  
 <http://www.json.org/>

<http://www.copterlabs.com/blog/json-what-it-is-how-it-works-how-to-use-it/>

WHAT IS JSON?

[JSON](http://en.wikipedia.org/wiki/JSON) is short for JavaScript Object Notation, and is a way to store information in an organized, easy-to-access manner. In a nutshell, it gives us a human-readable collection of data that we can access in a really logical manner.

### Storing JSON Data

As a simple example, information about me might be written in JSON as follows:

|  |  |
| --- | --- |
| 12345 | **var** jason **=** {  "age" **:** "24",  "hometown" **:** "Missoula, MT",  "gender" **:** "male"  }; |

This creates an object that we access using the variable jason. By enclosing the variable’s value in curly braces, we’re indicating that the value is an object. Inside the object, we can declare any number of properties using a "name": "value" pairing, separated by commas. To access the information stored in jason, we can simply refer to the name of the property we need. For instance, to access information about me, we could use the following snippets:

|  |  |
| --- | --- |
| 12 | document.write('Jason is ' jason.age); *// Output: Jason is 24*  document.write('Jason is a ' jason.gender); *// Output: Jason is a male* |

### Storing JSON Data in Arrays

A slightly more complicated example involves storing two people in one variable. To do this, we **enclose multiple objects in square brackets,**which signifies an array. For instance, if I needed to include information about myself and my brother in one variable, I might use the following:

|  |  |
| --- | --- |
| 12345678910 | **var** family **=** [{  "name" **:** "Jason",  "age" **:** "24",  "gender" **:** "male"  },  {  "name" **:** "Kyle",  "age" **:** "21",  "gender" **:** "male"  }]; |

[**view raw**](https://gist.github.com/jlengstorf/2760314/raw/gistfile1.js)[**gistfile1.js**](https://gist.github.com/jlengstorf/2760314#file-gistfile1-js) hosted with ❤ by **[GitHub](https://github.com/)**

To access this information, we need to access the array index of the person we wish to access. For example, we would use the following snippet to access info stored in family:

|  |  |
| --- | --- |
| 12 | document.write(family[1].name); *// Output: Kyle*  document.write(family[0].age); *// Output: 24* |

### Nesting JSON Data

Another way to store multiple people in our variable would be to**nest objects.**To do this, we would create something similar to the following:

|  |  |
| --- | --- |
| 123456789101112 | **var** family **=** {  "jason" **:** {  "name" **:** "Jason Lengstorf",  "age" **:** "24",  "gender" **:** "male"  },  "kyle" **:** {  "name" **:** "Kyle Lengstorf",  "age" **:** "21",  "gender" **:** "male"  }  } |

[**view raw**](https://gist.github.com/jlengstorf/2760326/raw/gistfile1.js)[**gistfile1.js**](https://gist.github.com/jlengstorf/2760326#file-gistfile1-js) hosted with ❤ by **[GitHub](https://github.com/)**

Accessing information in nested objects is a little easier to understand; to access information in the object, we would use the following snippet:

|  |  |
| --- | --- |
| 123 | document.write(family.jason.name); *// Output: Jason Lengstorf*  document.write(family.kyle.age); *// Output: 21*  document.write(family.jason.gender); *// Output: male* |

RSS

1. **RSS** stands for Really Simple Syndication. It's an easy way for you to keep up with news and information that's important to you, and helps you avoid the conventional methods of browsing or searching for information on websites.

<http://www.oracle.com/technetwork/articles/java/json-1973242.html>

**Listing 1. Example of JSON representation of an object**

JSON is often used in Ajax applications, configurations, databases, and RESTful web services. All popular websites offer JSON as the data exchange format with their RESTful web services.

# [What is JSON and why would I use it?](http://stackoverflow.com/questions/383692/what-is-json-and-why-would-i-use-it)

<http://stackoverflow.com/questions/383692/what-is-json-and-why-would-i-use-it>

**JSON (JavaScript Object Notation) is a lightweight format that is used for data interchanging.** It is also a subset of JavaScript's Object Notation (the way objects are built in JavaScript)

An example of where this is used is web services responses. In the 'old' days, web services used XML as their primary data format for transmitting back data, but since JSON appeared (*The JSON format is specified in*[*RFC 4627*](http://tools.ietf.org/html/rfc4627)*by Douglas Crockford*), it has been the preferred format because it is much more **lightweight**

You can find a lot more info on one of Crockford's sites [here](http://www.json.org/)

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.

Here is an example of JSON data:

{

"firstName": "John",

"lastName": "Smith",

"address": {

"streetAddress": "21 2nd Street",

"city": "New York",

"state": "NY",

"postalCode": 10021

},

"phoneNumbers": [

"212 555-1234",

"646 555-4567"

]

}

## JSON in JavaScript

When working with JSON and JavaScript, you may be tempted to use the eval function to evaluate the result returned in the callback, but this is not suggested due to security reasons (malicious data can be sent from the server to the client and then evaled in the client script with harmful effects).

Therefore, one must always try to use Crockford's script that checks for a valid JSON before evaluating it. Link to the script explanation is found [here](http://www.json.org/js.html) and here is a [direct link](https://github.com/douglascrockford/JSON-js/blob/master/json2.js) to the js file.

Example on how to use the JSON Parser (with the json from the above code snippet):

//The callback function that will be executed once data is received from the server

var callback = function (result) {

var johnny = JSON.parse(result);

//Now, the variable 'johnny' is an object that contains all of the properties

//from the above code snippet (the json example)

alert(johnny.firstName + ' ' + johnny.lastName); //Will alert 'John Smith'

};

The JSON Parser also offers another very useful method, stringify. This method accepts a JavaScript object as a parameter, and outputs back a string with JSON format. This is useful for when you want to send data back to the server:

var anObject = {name: "Andreas", surname : "Grech", age : 20};

var jsonFormat = JSON.stringify(anObject);

//The above method will output this: {"name":"Andreas","surname":"Grech","age":20}

The above two methods (parse and stringify) also take a second parameter, which is a function that will be called for every key and value at every level of the final result, and each value will be replaced by result of your inputted function. (More on this [here](http://www.json.org/js.html))

Btw, for all of you out there who think JSON is just for JavaScript, check out [this post](http://simonwillison.net/2006/Dec/20/json/) that explains and confirms otherwise.

# SON vs XML: How JSON Is Superior To XML

<https://www.udemy.com/blog/json-vs-xml/>

## Extensibility

With JSON, you are limited to only storing classical data like text and numbers. However, XML allows you to store any data type you can come up with. The ability to extend the attributes of the data stored in XML files is what allows it to be more flexible than JSON. However, it also makes it more difficult to read. This makes XML more extensible, but that may not be a good thing. This depends on the type of information you are trying to transfer. Documents require extensibility to manage images, charts, graphs, and other elements of formatting. However, classical data does not require this extensibility and can benefit from the simplicity of JSON. This [course about APIs](https://www.udemy.com/introduction-to-web-apis/?tc=blog.jsonvsxml.text.p&utm_source=blog&utm_medium=udemyads&utm_content=post15108&utm_campaign=content-marketing-blog&xref=blog) covers the extensibility of XML and the use of JSON in much more detail.

**JSON**

**Pro:**

* Simple syntax, which results in less "markup" overhead compared to XML.
* Easy to use with JavaScript as the markup is a subset of JS object literal notation and has the same basic data types as JavaScript.

**Con:**

* Simple syntax, only a handful of different data types is supported.

**XML**

**Pro:**

* Generalized markup, it is possible to create "dialects" for any kind of purpose
* [XML Schema](http://en.wikipedia.org/wiki/XML_Schema_%28W3C%29) for datatype, structure validation. Makes it also possible to create new datatypes.
* [XSLT](http://en.wikipedia.org/wiki/XSLT) for transformation into different output formats.
* [XPath](http://en.wikipedia.org/wiki/Xpath) / [XQuery](http://en.wikipedia.org/wiki/Xquery) for extracting information (which makes getting information in deeply nested structures much easier then with JSON).

**Con:**

* Relatively wordy compared to JSON (results in more data for the same amount of information).

<http://www.mkyong.com/tutorials/java-json-tutorials/>

**JSON** (JavaScript Object Notation), is a simple and easy to read and write data exchange format. It’s popular and implemented in countless projects worldwide, for those don’t like XML, JSON is a very good alternative solution.

In this series of Java JSON tutorials, we focus on three popular third party Java libraries to process JSON data, which are **Jackson**, **Google Gson** and **JSON.simple**

# JSON.Simple Example – Read And Write JSON

<http://www.mkyong.com/java/json-simple-example-read-and-write-json/>

#### 1. JSON.simple Dependency

**JSON.simple** is available at Maven central repository, just declares following dependency in your pom.xml file.

**<dependency>**

**<groupId>**com.googlecode.json-simple**</groupId>**

**<artifactId>**json-simple**</artifactId>**

**<version>**1.1**</version>**

**</dependency>**

#### 2. Write JSON to file

In below example, it write JSON data via JSONObject and JSONArray, and save it into a file named “**test.json**“.

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** org.json.simple.JSONArray;

**import** org.json.simple.JSONObject;

**public** **class** JsonSimpleExample {

**public** **static** **void** main(String[] args) {

JSONObject obj = **new** JSONObject();

obj.put("name", "mkyong.com");

obj.put("age", **new** Integer(100));

JSONArray list = **new** JSONArray();

list.add("msg 1");

list.add("msg 2");

list.add("msg 3");

obj.put("messages", list);

**try** {

FileWriter file = **new** FileWriter("c:**\\**test.json");

file.write(obj.toJSONString());

file.flush();

file.close();

} **catch** (IOException e) {

e.printStackTrace();

}

System.out.print(obj);

}

}

**Output** – See content of file named “**test.json**“.

**{**

"age":100,

"name":"mkyong.com",

"messages":**[**"msg 1","msg 2","msg 3"**]**

**}**

<https://code.google.com/p/json-path/>

# [@RequestBody and @ReponseBody spring](http://stackoverflow.com/questions/11291933/requestbody-and-reponsebody-spring)

Example:

<http://stackoverflow.com/questions/11291933/requestbody-and-reponsebody-spring>

Using a JavaScript-library like JQuery, you would post a JSON-Object like this:

{ "firstName" : "Elmer", "lastName" : "Fudd" }

Your controller method would look like this:

// controller

@ResponseBody @RequestMapping("/description")

public Description getDescription(@RequestBody UserStats stats){

return new Description(stats.getFirstName() + " " + stats.getLastname() + " hates wacky wabbits");

}

// domain / value objects

public class UserStats{

private String firstName;

private String lastName;

// + getters, setters

}

public class Description{

private String description;

// + getters, setters, constructor

}

Now if you have [Jackson](http://jackson.codehaus.org/) on your classpath (and have an [<mvc:annotation-driven>](http://static.springsource.org/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-config-enable) setup), Spring would convert the incoming JSON to a *UserStats* object from the post body (because you added the @Request Body annotation) and it would serialize the returned object to JSON (because you added the @ResponseBodyannotation). So the Browser / Client would see this JSON result:

{ "description" : "Elmer Fudd hates wacky wabbits" }

See this previous answer of mine for a complete working example: <http://stackoverflow.com/a/5908632/342852>

Note: RequestBody / ResponseBody is of course not limited to JSON, both can handle multiple formats, including plain text and XML, but JSON is probably the most used format.

@RequestBody is on the parameter, @ResponseBody is on the method. important difference!

Intvw questions

<http://www.globalguideline.com/interview_questions/Questions.php?sc=JSON>