What is JSON

JSON is an acronym for JavaScript Object Notation, is an open standard format, which is lightweight and text-based, designed explicitly for human-readable data interchange. It is a language-independent data format. It supports almost every kind of language, framework, and library.

In the early 2000s, JSON was initially specified by Douglas Crockford. In 2013, JSON was standardized as ECMA-404, and RCF 8259 was published in 2017.

JSON is an open standard for exchanging data on the web. It supports data structures like objects and arrays. So, it is easy to write and read data from JSON.

In JSON, data is represented in key-value pairs, and curly braces hold objects, where a colon is followed after each name. The comma is used to separate key-value pairs. Square brackets are used to hold arrays, where each value is comma-separated.

What is JSON

* JSON stands for JavaScript Object Notation.
* JSON is an open standard data-interchange format.
* JSON is lightweight and self-describing.
* JSON originated from JavaScript.
* JSON is easy to read and write.
* JSON is language independent.
* JSON supports data structures such as arrays and objects.

Features of JSON

* Simplicity
* Openness
* Self-Describing
* Internationalization
* Extensibility
* Interoperability

Why do we use JSON?

Since JSON is an easy-to-use, lightweight language data interchange format in comparison to other available options, it can be used for API integration. Following are the advantages of JSON:

* **Less Verbose**: In contrast to XML, JSON follows a compact style to improve its users' readability. While working with a complex system, JSON tends to make substantial enhancements.
* **Faster**: The JSON parsing process is faster than that of the XML because the DOM manipulation library in XML requires extra memory for handling large XML files. However, JSON requires less data that ultimately results in reducing the cost and increasing the parsing speed.
* **Readable**: The JSON structure is easily readable and straightforward. Regardless of the programming language that you are using, you can easily map the domain objects.
* **Structured Data**: In JSON, a map data structure is used, whereas XML follows a tree structure. The key-value pairs limit the task but facilitate the predictive and easily understandable model.

JSON Data Types

Following are the most commonly used JSON data types.

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Description** | **Example** |
| String | A string is always written in double-quotes. It may consist of numbers, alphanumeric and special characters. | "student", "name", "1234", "Ver\_1" |
| Number | Number represents the numeric characters. | 121, 899 |
| Boolean | It can be either True or False. | true |
| Null | It is an empty value. |  |

JSON Objects

In JSON, objects refer to dictionaries, which are enclosed in curly brackets, i.e., { }. These objects are written in key/value pairs, where the key has to be a string and values have to be a valid JSON data type such as string, number, object, Boolean or null. Here the key and values are separated by a colon, and a comma separates each key/value pair.

For example:

1. {"name" : "Jack", "employeeid" : 001, "present" : false}

JSON Arrays

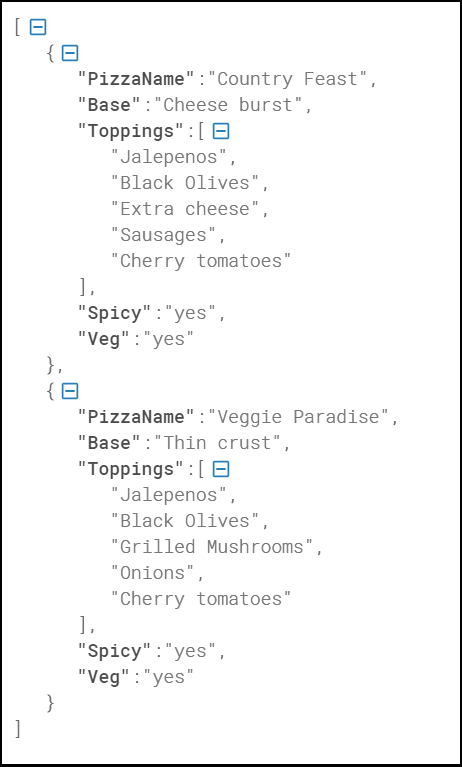
In JSON, arrays can be understood as a list of objects, which are mainly enclosed in square brackets [ ]. An array value can be a string, number, object, array, boolean or null.

For example:

1. [{
2. "PizzaName" : "Country Feast",
3. "Base" : "Cheese burst",
4. "Toppings" : ["Jalepenos", "Black Olives", "Extra cheese", "Sausages", "Cherry tomatoes"],
5. "Spicy" : "yes",
6. "Veg" : "yes"
7. },
9. {
10. "PizzaName" : "Veggie Paradise",
11. "Base" : "Thin crust",
12. "Toppings" : ["Jalepenos", "Black Olives", "Grilled Mushrooms", "Onions", "Cherry tomatoes"],
13. "Spicy" : "yes",
14. "Veg" : "yes"
15. }
16. ]

In the above example, the object "Pizza" is an array. It contains five objects, i.e., PizzaName, Base, Toppings, Spicy, and Veg.

Output:



JSON Vs XML

JSON stands for JavaScript Object Notation, whereas XML stands for Extensive Markup Language. Nowadays, JSON and XML are widely used as data interchange formats, and both have been acquired by applications as a technique to store structured data.

Difference between JSON and XML

|  |  |
| --- | --- |
| JSON is easy to learn. | XML is quite more complex to learn than JSON. |
| It is simple to read and write. | It is more complex to read and write than JSON. |
| It is data-oriented. | It is document-oriented. |
| JSON is less secure in comparison to XML. | XML is highly secured. |
| It doesn't provide display capabilities. | It provides the display capability because it is a markup language. |
| It supports the array. | It doesn't support the array |
| Example :  [  {  "name" : "Peter",  "employed id" : "E231",  "present" : true,  "numberofdayspresent" : 29  },  {  "name" : "Jhon",  "employed id" : "E331",  "present" : true,  "numberofdayspresent" : 27  }  ] |  |

JSON vs XML

Before knowing about the differences between JSON and XML, we should be aware of the definition of json and xml.

What is json?

JSON stands for **JavaScript object notation**. JSON has been derived from javascript, where javascript is a programming language. It was originally created to hold the structured data that could be used in javascript. JSON became so popular that it is used for data for all kinds of applications. It is the most popular way of sending the data for Web APIs.

**Basic data types supported by json are:**

* **Strings**: Characters that are enclosed in single or double quotation marks.
* **Number**: A number could be integer or decimal, positive or negative.
* **Booleans**: The Boolean value could be either true or false without any quotation marks.
* **Null**: Here, null means nothing without any quotation marks.

In addition to basic data types, json has arrays and objects.

Arrays

Arrays are the lists that are represented by the square brackets, and the values have commas in between them. They can contain mix data types, i.e., a single array can have strings, Boolean, numbers.

**For example:**

**Example 1:** [1, 2, 7.8, 5, 9, 10];

**Example 2:** ["red", "yellow", "green"];

**Example 3:** [8, "hello", null, true];

In the above, example 1 is an array of numbers, example 2 is an array of strings, and example 3 is an array of mix data types.

Objects

Objects are JSON dictionaries that are enclosed in curly brackets. In objects, keys and values are separated by a colon ':', pairs are separated by comma. Keys and values can be of any type, but the most common type for the keys is a string.

For example: {"red" : 1, "yellow" : 2, "green" : 3};

Nesting

Nesting involves keeping the arrays and objects inside of each other. We can put the arrays inside objects, objects inside arrays, arrays inside arrays, etc. We can say that json file is a big object with lots of objects and arrays inside.

**Example:**

1. {
2. "song" :
3. {
4. "title" :  "Hey Dude";
5. "artist": "The Beatles";
6. "musicians": ["John Lennon", "Paul McCratney", "Ringo Starr"];
7. }
8. }

In the above code, the song starts with a curly bracket. Therefore, a song is an object. It contains three key-value pairs wherein title, artist and musicians are the keys.'

What is XML?

XML stands for an extensible markup language. It is like HTML, where HTML stands for Hypertext Markup language. HTML is used for creating websites, whereas XML can be used for any kind of structured data.

XML has two ways of handling data, i.e., Tags and Attributes. The tags work as HTML. The start tags start with the <\_> and end with the </\_>. The start and end tags must match. The names must only be letters, numbers, and underscore, and the tag name must start with a letter only.

**For example:**

<title> Hello World </title>

Nested Tags

When we put the tag inside of another tag that creates the nested data.

**For example:**

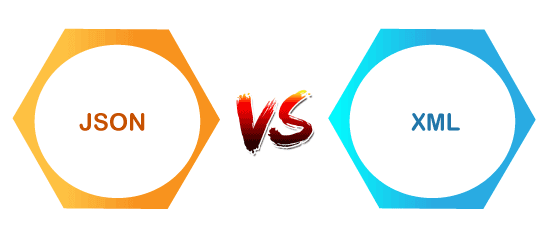
1. **<color>**
2. **<red>** 1 **</red>**
3. **<yellow>** 2 **</yellow>**
4. **<green>** 3 **</green>**
5. **</color>**

As we can observe in the above code that inside the color tag, we have three more tags, i.e., red, yellow, and green.

Similarities between the json and XML.

* **Self-describing:** Both json and xml are self-describing as both xml data and json data are human-readable text.
* **Hierarchical:** Both json and xml support hierarchical structure. Here hierarchical means that the values within values.
* **Data interchange format:** JSON and XML can be used as data interchange formats by many different programming languages.
* **Parse:** Both the formats can be easily parsed.
* **Retrieve:** Both formats can be retrieved by using HTTP requests. The methods used for retrieving the data are GET, PUT, POST.

Differences between the json and XML.



**The following are the differences between the json and xml:**

|  |  |
| --- | --- |
| **JSON** | **XML** |
| JSON stands for javascript object notation. | XML stands for an extensible markup language. |
| The extension of json file is .json. | The extension of xml file is .xml. |
| The internet media type is application/json. | The internet media type is application/xml or text/xml. |
| The type of format in JSON is data interchange. | The type of format in XML is a markup language. |
| It is extended from javascript. | It is extended from SGML. |
| It is open source means that we do not have to pay anything to use JSON. | It is also open source. |
| The object created in JSON has some type. | XML data does not have any type. |
| The data types supported by JSON are strings, numbers, Booleans, null, array. | XML data is in a string format. |
| It does not have any capacity to display the data. | XML is a markup language, so it has the capacity to display the content. |
| JSON has no tags. | XML data is represented in tags, i.e., start tag and end tag. |
|  | XML file is larger. If we want to represent the data in XML then it would create a larger file as compared to JSON. |
| JSON is quicker to read and write. | XML file takes time to read and write because the learning curve is higher. |
| JSON can use arrays to represent the data. | XML does not contain the concept of arrays. |
| It can be parsed by a standard javascript function. It has to be parsed before use. | XML data which is used to interchange the data, must be parsed with respective to their programming language to use that. |
| It can be easily parsed and little bit code is required to parse the data. | It is difficult to parse. |
| File size is smaller as compared to XML. | File size is larger. |
| JSON is data-oriented. | XML is document-oriented. |
| It is less secure than XML. | It is more secure than JSON. |

JSON Example

JSON example can be created by object and array. Each object can have different data such as text, number, boolean etc. Let's see different JSON examples using object and array.

JSON Object Example

A JSON object contains data in the form of key/value pair. The keys are strings and the values are the JSON types. Keys and values are separated by colon. Each entry (key/value pair) is separated by comma.

The **{** (curly brace) represents the JSON object.

1. {
2. "employee": {
3. "name":       "sonoo",
4. "salary":      56000,
5. "married":    **true**
6. }
7. }

JSON Array example

The **[** (square bracket) represents the JSON array. A JSON array can have values and objects.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

Let's see the example of JSON array having values.

1. ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"]

Let's see the example of JSON array having objects.

1. [
2. {"name":"Ram", "email":"Ram@gmail.com"},
3. {"name":"Bob", "email":"bob32@gmail.com"}
4. ]

JSON Example 1

1. {"employees":[
2. {"name":"Shyam", "email":"shyamjaiswal@gmail.com"},
3. {"name":"Bob", "email":"bob32@gmail.com"},
4. {"name":"Jai", "email":"jai87@gmail.com"}
5. ]}

The XML representation of above JSON example is given below.

1. **<employees>**
2. **<employee>**
3. **<name>**Shyam**</name>**
4. **<email>**shyamjaiswal@gmail.com**</email>**
5. **</employee>**
6. **<employee>**
7. **<name>**Bob**</name>**
8. **<email>**bob32@gmail.com**</email>**
9. **</employee>**
10. **<employee>**
11. **<name>**Jai**</name>**
12. **<email>**jai87@gmail.com**</email>**
13. **</employee>**
14. **</employees>**

JSON Example 2

1. {"menu": {
2. "id": "file",
3. "value": "File",
4. "popup": {
5. "menuitem": [
6. {"value": "New", "onclick": "CreateDoc()"},
7. {"value": "Open", "onclick": "OpenDoc()"},
8. {"value": "Save", "onclick": "SaveDoc()"}
9. ]
10. }
11. }}

The XML representation of above JSON example is given below.

1. **<menu** id="file" value="File"**>**
2. **<popup>**
3. **<menuitem** value="New" onclick="CreateDoc()" **/>**
4. **<menuitem** value="Open" onclick="OpenDoc()" **/>**
5. **<menuitem** value="Save" onclick="SaveDoc()" **/>**
6. **</popup>**
7. **</menu>**

JSON Object

JSON object holds key/value pair. Each key is represented as a string in JSON and value can be of any type. The keys and values are separated by colon. Each key/value pair is separated by comma.

The curly brace **{** represents JSON object.

Let's see an example of JSON object.

1. {
2. "employee": {
3. "name":       "sonoo",
4. "salary":      56000,
5. "married":    **true**
6. }
7. }

In the above example, employee is an object in which "name", "salary" and "married" are the key. In this example, there are string, number and boolean value for the keys.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

JSON Object with Strings

The string value must be enclosed within double quote.

1. {
2. "name":       "sonoo",
3. "email":      "sonoojaiswal1987@gmail.com"
4. }

JSON Object with Numbers

JSON supports numbers in double precision floating-point format. The number can be digits (0-9), fractions (.33, .532 etc) and exponents (e, e+, e-,E, E+, E-).

1. {
2. "integer": 34,
3. "fraction": .2145,
4. "exponent": 6.61789e+0
5. }

JSON Object with Booleans

JSON also supports boolean values *true* or *false*.

1. {
2. "first": **true**,
3. "second": **false**
4. }

JSON Nested Object Example

A JSON object can have another object also. Let's see a simple example of JSON object having another object.

1. {
2. "firstName": "Sonoo",
3. "lastName": "Jaiswal",
4. "age": 27,
5. "address" : {
6. "streetAddress": "Plot-6, Mohan Nagar",
7. "city": "Ghaziabad",
8. "state": "UP",
9. "postalCode": "201007"
10. }
11. }

JSON Array

JSON array represents ordered list of values. JSON array can store multiple values. It can store string, number, boolean or object in JSON array.

In JSON array, values must be separated by comma.

The **[** (square bracket) represents JSON array.

JSON Array of Strings

Let's see an example of JSON arrays storing string values.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"]

JSON Array of Numbers

Let's see an example of JSON arrays storing number values.

1. [12, 34, 56, 43, 95]

JSON Array of Booleans

Let's see an example of JSON arrays storing boolean values.

1. [**true**, **true**, **false**, **false**, **true**]

JSON Array of Objects

Let's see a simple JSON array example having 4 objects.

1. {"employees":[
2. {"name":"Ram", "email":"ram@gmail.com", "age":23},
3. {"name":"Shyam", "email":"shyam23@gmail.com", "age":28},
4. {"name":"John", "email":"john@gmail.com", "age":33},
5. {"name":"Bob", "email":"bob32@gmail.com", "age":41}
6. ]}

JSON Multidimensional Array

We can store array inside JSON array, it is known as array of arrays or multidimensional array.

1. [
2. [ "a", "b", "c" ],
3. [ "m", "n", "o" ],
4. [ "x", "y", "z" ]
5. ]

JSON Comments

JSON doesn't support comments. It is not a standard.

But you can do some tricks such as adding extra attribute for comment in JSON object, for example:

1. {
2. "employee": {
3. "name":       "Bob",
4. "salary":      56000,
5. "comments":    "He is a nice man"
6. }
7. }

Here, "comments" attribute can be treated as comment.

JSON Compare

In this article, we are going to learn about the comparison between JSON. This tutorial is solely intended to work on comparing different chunks of data present in the JSON format. We might opt for using different local as well as online tools. The comparison is done not only to check if the dependencies of the data are relevant, but also for multiple things that we are going to discuss in this article. In a complete sense, we would be learning what factors the comparison is usually based on. We might also discuss some of the tools that are available to locally compare the JSON and also some online tools that are based out on it. Let's proceed.

The [JSON](https://www.javatpoint.com/json-tutorial) compare is a superset of **JSONLint validator**, a tool that is used to compare different sorts of JSON data. JSON compare serves similar features like JSONLint in an advanced manner. This tool allows us to take input directly and validate the JSON code. After this, we can upload the JSON data pile and validate it even in a standalone environment or multiple batches. Meanwhile, you can simultaneously upload multiple files and compare them using a different set of commands present in the JSON Compare tool and later merge two JSON objects.

There are multiple ways of comparison done using the JSON compare tool. This tool has mainly **3 modes** that cover the entire set. They are:

**Simple:** This mode enables us to input or copy paste and validate.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

**Batch:** This mode enables us to upload multiple batches and proceed with validation.

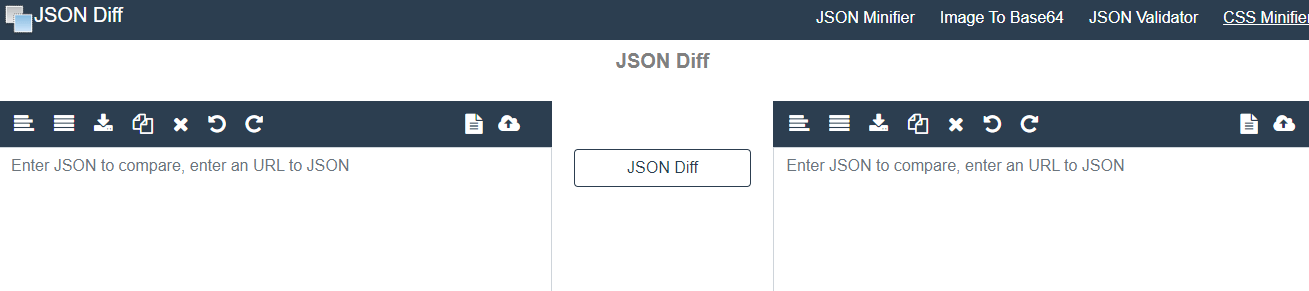
**Diff:** This mode enables us to directly input, copy-paste, merge and validate two separate JSON objects.

Let's look at them one by one. But before proceeding any further, we need to install all of these modes to compare our JSON data.

1. npm install -g json-simple
3. npm install -g json-batch
5. npm install -g json-diff

The above commands will install all the dependencies of the modes used in the JSON Compare.

Among these three modes, most developers prefer using diff mode since the simple command uniquely picks up and merges two JSON objects, batch, and merge them with ease. There are multiple domain websites for using diff. Consider an example image as shown.



As shown in the image above, the diff mode offers different methods of comparing two JSON objects. We can sort them, copy-paste them, merge, and compare them. Such compare is validated automatically as soon as we write or paste our JSON objects on both sides. After writing or pasting, the JSON Diff button can easily manipulate accordingly.

Consider a function that is used to compare two JSON objects and then return the JSON object with a list of all possible differences like the below example.

1. var madrid =
2. '{"type":"team","description":"Good","trophies":[{"ucl":"10"},
3. {"copa":"5"}]}';
5. var barca =
6. '{"type":"team","description":"Bad","trophies":[{"ucl":"3"}]}';

If we run the compare mode diff like (**madrid, barca**), the returned object would look something like this.

1. {"description" : "Bad", "trophies":[{"ucl":"3"}, {"copa":"5"}]};

It is quite evident from the above example that the compare command diff returns the different objects present in both the defined object. It is simple as that. It needs no further explanation since the diff command automatically compares like the generic sort or compares functions used in the programming paradigms.

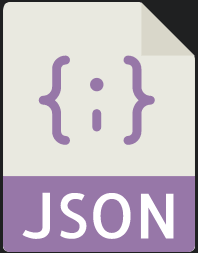
Advantages of using JSON diff

1. JSON diff is comparatively a proficient mode than simple or batch mode.
2. The comparison parameter for using diff is low and easily manageable.
3. The validation done by the JSON diff tool is highly efficient and reliable.
4. JSON diff checks all the constraints strictly and validates them in multiple batches reducing the time required for comparison.

Summary

In this article, we came across the fact of using JSON compare to use it in comparing various sorts of objects present in the JSON data set. We also learned about online tools that reduce the cost of infrastructure required for comparison. Later, we discussed the installation of different modes of JSON compare tool with which we can perform simple to complex tasks depending upon the requirement. We also discussed JSON diff mode that is highly used and has properties of both simple and batch mode. We also discussed with an example how the diff mode compares two objects in JSON and rolls out the difference among the two objects through a function call.

JSON Beautifier



JSON or **JavaScript Object Notation** is a format for open standard file and data interchange format that uses human-readable text to capture and transfer data objects consisting of some attributes i.e. arrays and value pairs or serializable values. JSON is very popular these days since the frameworks of every [JavaScript](https://www.javatpoint.com/javascript-tutorial) library or any other programming paradigm uses it because of its **simplicity** and **readability**. JSON is a very common data format used for storing a diverse range of data in various types of applications. For example, a responsive server-based web application.

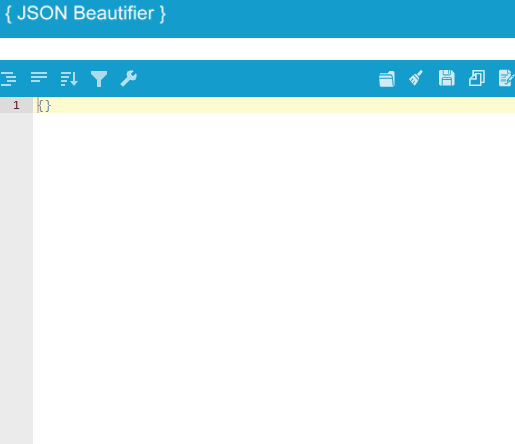
Additionally, while dealing with data of various types, it turns quite hectic for humans to manipulate the JSON data. The problem takes a more drastic turn when the huge application that might be highly dependent on JSON becomes difficult to debug whenever a conflict occurs. It becomes difficult for humans to figure out where the bug exists while moving through all the heavy piles of JSON. This problem needs immediate remedy to cure. That's the reason why JSON beautifier.

What is JSON beautifier?

[JSON](https://www.javatpoint.com/json-tutorial) beautifier is a tool that helps developers to control the format of data in JSON files. It is a **free tool** offered by the JSON online website to curb the need of reducing bugs from massive JSON data. The JSON beautifier recognizes the bad patterns of the raw JSON data and constructs them again to make them ready to be easily readable. This is thereby beneficial for the developer's community to save time and money for users. Developers do not need to pay any single penny to view their standalone JSON data of the application they have been working on. They can easily compile and test the JSON data that is being processed. We can consider it as an all-in-one web-based tool that serves multiple features. Moreover, there's no need for any credit cards or any hidden charges involved while using them. We can easily access the online tool through the browser itself. Also, no plug-in or device-specific configurations are required to use this online tool and have access to all sorts of available browsers today.

The JSON beautifier also comes along with JSON **formatter** that is a cross-collaborative tool to integrate the JSON data by organizing them into specific criteria. Be it data which is an integer, string, or anything, the beautifier tool handles all these kinds easily. Whenever we have a pile of JSON data with us that are hard to read and understand, we can opt for using this beautifier to arrange the data into readable and beautiful outlay. The JSON beautifier looks something like this.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)



In the above image, the blank space represents the **editable** or **unorganized JSON** that we would be writing or pasting from our source repository or file. The above options pop up as soon as the cursor moves along. We can compile it automatically or debug it as and when required.

Consider the below JSON data that is unsorted or not properly arranged.

1. {
2. "Company": {
3. "Employee": {
4. "FirstName": "Sonoo",
5. "LastName": "Jaiswal",
6. "ContactNo": "1234567890",
7. "Email": "sonoojaiswal@javatpoint.com",
8. "Address": {
9. "City": "Noida",
10. "State": "Uttar Pradesh",
11. "Zip": "123456"
12. }
13. }
14. }
15. }

The above pile of JSON data seems to be confusing with brackets and parameters. It is visibly difficult to generalize which bracket is enclosed with what parameters. Using JSON beautifier, we can easily make it readable, sort it accordingly and also edit if the data is not right. The beautified JSON data after using JSON online beautifier looks like this

1. {
2. "Company": {
3. "Employee": {
4. "FirstName": "Sonoo",
5. "LastName": "Jaiswal",
6. "ContactNo": "1234567890",
7. "Email": "sonoojaiswal@javatpoint.com",
8. "Address": {
9. "City": "Noida",
10. "State": "Uttar Pradesh",
11. "Zip": "123456"
12. }
13. }
14. }
15. }

It is quite visible that using the JSON beautifier makes the data look amazing and easy to understand. The brackets appear balanced with parameters. Similarly, all sorts of such unorganized or improper format of data while working with large projects can be easily manipulated to understand using the JSON beautifier.

Summary

We also saw how unorganized data or unsorted data can be easily beautified using this utility tool. The web interface of the online application is quite easy to understand along with in-built features of sorting the unsorted array, minimizing the complexities while working with the huge piles of data in JSON formatter. We also saw an example of how the JSON beautifier balances the parenthesis present in the JSON data format. This makes it easier for any developer to carry out operations and meanwhile edit the JSON data while beautifying it with ease.

JSON Server

Most developers have heard of the term JSON. It abbreviates for "**JavaScript Object Notation**". JSON is nothing but an open standard file format and data interchange format. It uses human-readable text to store and transfer data objects. It generally consists of two human-readable attributes i.e. **value pairs** and **arrays.** JSON is a very common data format having different applications. For example, a web application communicating with a server uses JSON.

In the similar sense, [JSON](https://www.javatpoint.com/json-tutorial) Server is no different. JSON Server is a Node Module that you can use to create demo **REST JSON** services within a short span of minutes. All we need to do is have a JSON file as sample data. Let's learn more about how to install and set up a JSON server in our system and alongside we'll try to create a **REST API**. Let's learn more about it.

Installation

To get started with the installation of the JSON server, all we need to do is to open the Terminal of our system and type the following command.

1. npm install -g json-server

The next step involves the creation of a database with JSON. For the sake of learning, we have created an example database as shown below.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. //db.json

4. "posts": [
5. { "id": 1, "title": "json-server", "author": "typicode" }
6. ],
7. "comments": [
8. { "id": 1, "body": "some comment", "postId": 1 }
9. ],
10. "profile": { "name": "typicode" }

We are almost done. The next step is to start the JSON Server and to do that use the below command.

1. json server --watch db.json

Now, if we move to http://localhost:3000/posts/1, we'll get something like this.

1. { "id": 1, "title": "json-server", "author": "typicode" }

This output signifies that our JSON server is working fine for the above attributes and is having some values in them. We can now easily create a request for the JSON Server. But, before moving further, it is good to take some key factors into account. Those key factors are:

1. If we make some requests like **POST, DELETE, PUT**, or **PATCH**, the changes are directly reflected and safely stored in the db.json file.

2. The requested JSON body should be object enclosed, just like the **GET** For example:

1. {"name": Foobar}

3. The **ID** values are not generally mutable. Any value of the ID present in the body of the PUT and PATCH request is usually ignored by default. The value set in a POST request is only taken into consideration if not already taken.

4. The PATCH, POST and PUT request should always include a **Content-Type: application/json** header to be used in the JSON request body. Else, it will return the **2XX** status code without updating the changes made to the data.

Routes

Routes are nothing but the pathways through which the requests are processed in a pipeline. A specific route serves a specific request so that dependencies are not disturbed.

Therefore, based on the previous **db.json** file various default routes can be added with the command **--routes** at the terminal. Some default routes are shown below.

Singular Routes

1. GET    /profile
2. POST   /profile
3. PUT    /profile
4. PATCH  /profile

Plural Routes

1. GET    /posts
2. GET    /posts/1
3. POST   /posts
4. PUT    /posts/1
5. PATCH  /posts/1
6. DELETE /posts/1

Creating REST API

**Representational State Transfer** (REST) is a software architectural style that uses a subset of **HTTP**. It is generally used to create an interactive application based on the available services offered by the web. All those applications that follow these web services guidelines are thereby termed **RESTful**.

Now let's create a new JSON file with the name db.json. This file would contain the sample data that should be exposed by the REST API. For all the objects contained inside the JSON structure, **CRUD** (**Create Read Update Delete**) endpoints are automatically created. Let's take a look at the sample db.json file.

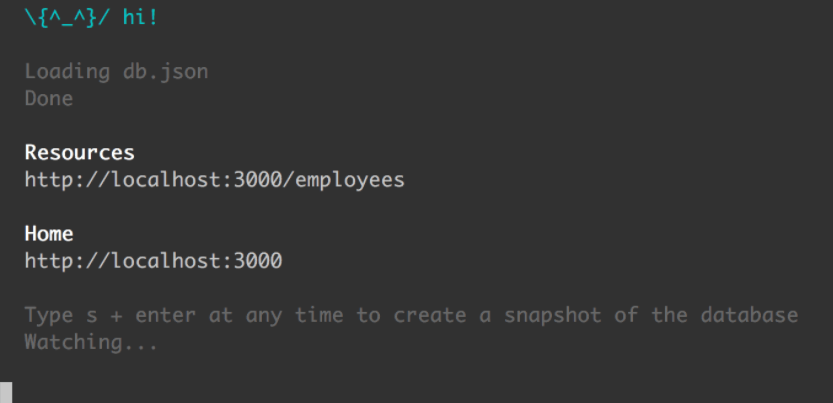
1. "employees": [
2. {
3. "id": 1,
4. "first\_name": "Sebastian",
5. "last\_name": "Eschweiler",
6. "email": "sebastian@javatpoint.com"
7. },
8. {
9. "id": 2,
10. "first\_name": "Steve",
11. "last\_name": "Palmer",
12. "email": "steve@javatpoint.com"
13. },
14. {
15. "id": 3,
16. "first\_name": "Ann",
17. "last\_name": "Smith",
18. "email": "ann@javatpoint.com"
19. }
20. ]
21. }

The above sample **db.json** structure consists of an objecting employee which has 3 assigned data sets. Each of the employee objects consists of four properties like **id, first\_name, last\_name,** and **email**.

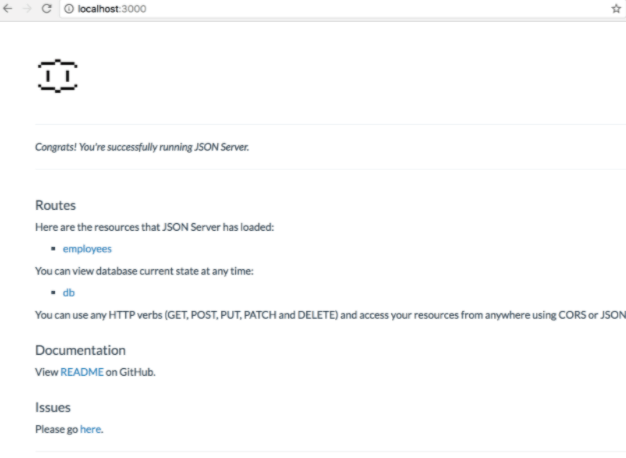
After this step, all we need to do is to run the server through the Terminal command.

1. json server --watch db.json

The next step is to pass a parameter over the file containing the JSON structure. Also, we are making use of the watch parameter because this makes us sure that the server is being started in watch mode and the changes and updates are accordingly exposed to the API. The following console shoots after we start the server.



Now, if we run the URL http://localhost:3000/employees in the browser environment, we might see the following results.



From the above image, we can see that the correct recognition of the resource employees. Now, when we click on the employees link and use the **HTTP GET** request to the http://localhost:3000/employees , it will show us the following JSON result.

1. [
2. {
3. "id": 1,
4. "first\_name": "Sebastian",
5. "last\_name": "Eschweiler",
6. "email": "sebastian@javatpoint.com"
7. },
8. {
9. "id": 2,
10. "first\_name": "Steve",
11. "last\_name": "Palmer",
12. "email": "steve@javatpoint.com"
13. },
14. {
15. "id": 3,
16. "first\_name": "Ann",
17. "last\_name": "Smith",
18. "email": "ann@javatpoint.com"
19. }

This action would create various routed endpoints like we discussed previously and on inspection, we would find the following results.

1. GET    /employees
2. GET    /employees/{id}
3. POST   /employees
4. PUT    /employees/{id}
5. PATCH  /employees/{id}
6. DELETE /employees/{id}

The above endpoints signify that we can now make DELETE, PATCH, GET, POST, and PUT requests and any changes would certainly be automatically saved to the **db.json** file. Meanwhile, while creating a PATCH, PUT or POST request, never forget to include a **Content-Type: application/json** header because the JSON would seek content in the requested body. Else, it might result in the **200 OK** without changes being made to the data.

Furthermore, there's another instance of extending our URL with more parameters like filtering the parameters by modifying the existing URL with the following set of the parameter using **http://localhost:3000/employees?first\_name=Sebastian**. In the similar sense, other requests can also be filtered easily and would be automatically reflected and saved to the db.json file. This is how we successfully created a RESTful API. We can test it, deploy it with ease using various tools like **Netlify, Heroku**, or **Postman**. Let's learn some extra actions that can be carried out with the temporary API we have created.

Extra Options

**Static file server**

The JSON server that we have created can be used to serve our HTML. CSS and JS compatibility but simply creating a **./public** directory or use the command **--static** so that a different static file directory is set.

1. mkdir **public**
2. echo 'hello world' > **public**/index.html
3. json-server db.json
4. json-server db.json --**static** ./some-other-dir

**Port Switch/ Alternative Port**

The request processing through the localhost can be shifted or switched to another port using the below command to our existing JSON Server. We just need to use the **--port number** flag. For example:

1. json-server --watch db.json --port 3004

**Middleware Module**

If we want any validation, authentication, or behavioral changes for the API or the project you want to create by creating a module in combination with other **Express** middleware. Consider the below instance.

1. npm install json-server --save-dev
2. // server.js
3. **const** jsonServer = require('json-server')
4. **const** server = jsonServer.create()
5. **const** router = jsonServer.router('db.json')
6. **const** middlewares = jsonServer.defaults()
8. server.use(middlewares)
9. server.use(router)
10. server.listen(3000, () => {
11. console.log('JSON Server is running')
12. })

Run the above file using,

1. node server.js

The path provided by us for the json.Server.router function in the above code snippet is relative to the directory from where we would be launching the node process. If we run the above code snippet from another directory, it is highly recommended to use an absolute path:

1. **const** path = require('path')
2. **const** router = jsonServer.router(path.join(\_\_dirname, 'db.json'))

For any other in-memory database, we simply need to pass an object to **jsonServer.router()** function in the existing Express project.

Data Mocking of JSON Server

As of now, we learned to expose data by creating a fake API manually in a JSON file. However, if we need large data sets, this manual way can result in hectic and absurd. Therefore, to cope with this issue, we can use **Faker.js** because it can handle large amounts of data for JSON Server. It does that by generating fake data when integrating with JSON and API. This can be done by using the following steps.

1. npm init

Next, install Faker.js by using the command:

1. npm install faker

Faker.js will successfully be installed in the **node\_modules** folder. Now, create another file in the employees.js and insert the below given sample JavaScript code.

1. // employees.js
2. var faker = require('faker')
3. function generateEmployees () {
4. var employees = []
5. **for** (var id = 0; id < 50; id++) {
6. var firstName = faker.name.firstName()
7. var lastName = faker.name.lastName()
8. var email = faker.internet.email()
9. employees.push({
10. "id": id,
11. "first\_name": firstName,
12. "last\_name": lastName,
13. "email": email
14. })
15. }
16. **return** { "employees": employees }
17. }

In the above code snippet, we are implementing the function **generateEmployees()** to generate JSON-based object containing the data of 50 employees. Thus, to obtain fake data for the first name, last name, and the email that we are using, we would be using the methods **faker.name.firstName(), faker.name.lastName()** and **faker.internet.email()**.

Moreover, the JSON Server requires exporting the general **employees()** function in the above code which is done by using the method.

1. module.exports = generateEmployees

Having added that export, we would now be able to pass the employees.js file directly to the json-server command by

1. Json-server employees.js

The work is done. We have successfully exposed REST API that will give us the access to all the 50 employees data sets by the means of Faker.js.

Summary

In this tutorial, we learned about the JSON Server from scratch to advance. We tried to blend in the use cases of it and eventually learned its implementation along with setting up the local environment, creating requests, etc. We also learned about routes that are quite important because we can use them to create pathways for data to move to our files and then trace back the requests like **POST, DELETE,** etc. In the later section, we learned to set up middleware with Express environment followed by mocking up the JSON Server with Faker.js. We can therefore jump to conclusion that JSON Server covers tremendous application in the development since most of the developers use this format for establishing data manipulation, access, and storage mechanism and carrying out overall data-driven paradigms.