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.

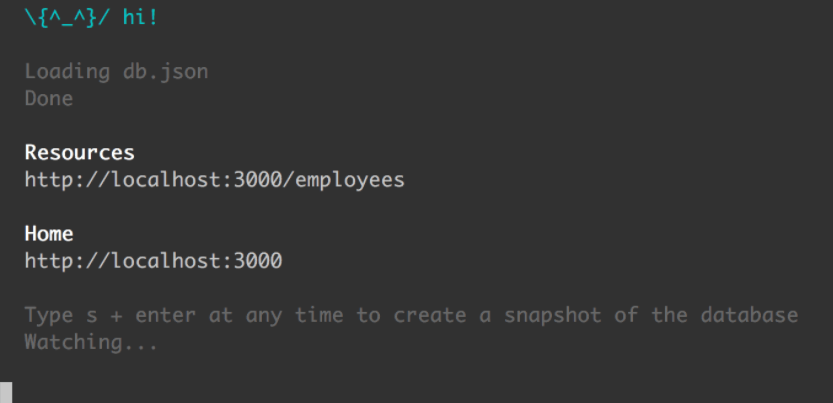
1. "employees": [
2. {
3. "id": 1,
4. "first\_name": "Sebastian",
5. "last\_name": "Eschweiler",
6. "email": "sebastian@javatpoint.com"
7. },
8. {
9. "id": 2,
10. "first\_name": "Steve",
11. "last\_name": "Palmer",
12. "email": "steve@javatpoint.com"
13. },
14. {
15. "id": 3,
16. "first\_name": "Ann",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

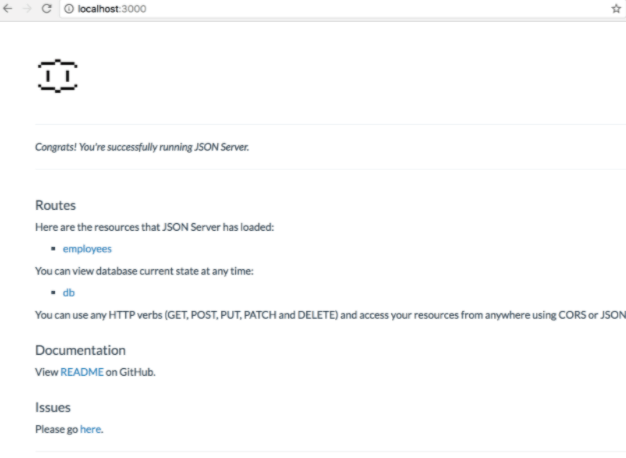
After this step, all we need to do is to run the server through the Terminal command.

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.

# JSON Placeholder

In this article, you are going to learn about JSON Placeholder and everything associated with it. But before moving further, you need to understand more about it. JSON or JavaScript Object Notation serves a lot of functionalities. One such functionality offered by JSON is the Placeholder. As the name suggests, the JSON Placeholder can generate a place to hold the data. This data is usually fake, although it appears genuine. It is done to fill the gap of testing methods, and primarily web development-related domains use it most of the time.

## What is JSON Placeholder?

[JSON](https://www.javatpoint.com/json-tutorial) Placeholder is a fake REST API that is primarily used for prototyping and testing. You can call it a **web developer's image placeholder.** JSON Placeholder is an online service that can be used when you need fake data to prototype or test some fake data. The code for JSON Placeholder can be run from anywhere under the support of JSONP and CORS. The primary use of JSON Placeholder is to fake a server, sharing the code, and many such REST API uses are associated with it.

## Why is JSON Placeholder used?

Most developers experience when they put their hands on a new library and try some exploring methods or hacking a prototype or tutorial. Hence, to test out new libraries and frameworks, they need some data for testing or exploring the aspects. They also refrain from the idea of using some API that is public because it usually takes more time to register a client and understand how things work in such a complex API rather than having to focus on the task he needs to do. In all such instances, there arises a need for some data that is available to be tested and can be faked using the REST API. This is the main reason why most developers use JSON Placeholder to gain insights. This is where the Placeholder comes in handy, like that of image placeholders mostly needed by web developers.

### Features of JSON Placeholder

1. It doesn't need any registrations.
2. It needs almost no configuration.
3. It forms the basic API automatically.
4. It shares many relationships with the data.
5. It can incorporate cross-domains like CORS and JSONP
6. It is highly supported for some requests like getting, POST, PATCH, PUT, etc.
7. It is greatly compatible with different JavaScript frameworks and libraries like **Backbone, AngularJS,** etc.

### Installation

To use JSON Placeholder, you need to install it on your local system using the following command given below.

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

1. $ npm install -g jsonplaceholder
2. $ jsonplaceholder

## How to use JSON Placeholder

Below are some code snippets in jQuery that show the things that can be done using JSON Placeholder. Since Github incorporates jQuery loading, you must pick up the code and paste the code snippets into your console. Let's see what the aspect where it is applicable is.

* **Getting a resource**

1. fetch('https://jsonplaceholder.typicode.com/posts/1')
2. .then((response) =**>** response.json())
3. .then((json) =**>** console.log(json));

**Output**

{

id: 1,

title: '...',

body: '...',

userId: 1

}

* **Listing Resource**

1. fetch('https://jsonplaceholder.typicode.com/posts')
2. .then((response) =**>** response.json())
3. .then((json) =**>** console.log(json));

**Output**

[

{ id: 1, title: '...' /\* ... \*/ },

{ id: 2, title: '...' /\* ... \*/ },

{ id: 3, title: '...' /\* ... \*/ },

/\* ... \*/

{ id: 100, title: '...' /\* ... \*/ },

];

* **Listing**

1. {
2. id: 1,
3. title: '...',
4. body: '...',
5. userId: 1
6. }

**Output**

{

id: 1,

title: '...',

body: '...',

userId: 1

}

* **Creating a resource**

1. fetch('https://jsonplaceholder.typicode.com/posts', {
2. method: 'POST',
3. body: JSON.stringify({
4. title: 'foo',
5. body: 'bar',
6. userId: 1,
7. }),
8. headers: {
9. 'Content-type': 'application/json; charset=UTF-8',
10. },
11. })
12. .then((response) =**>** response.json())
13. .then((json) =**>** console.log(json));

**Output**

{

id: 101,

title: 'foo',

body: 'bar',

userId: 1

}

#### **Note: The resources may not get updated by the server but JSON Placeholder fakes as if it is updated.**

* **Resource Updating**

1. fetch('https://jsonplaceholder.typicode.com/posts/1', {
2. method: 'PUT',
3. body: JSON.stringify({
4. id: 1,
5. title: 'foo',
6. body: 'bar',
7. userId: 1,
8. }),
9. headers: {
10. 'Content-type': 'application/json; charset=UTF-8',
11. },
12. })
13. .then((response) =**>** response.json())
14. .then((json) =**>** console.log(json));

**Output**

{

id: 1,

title: 'foo',

body: 'bar',

userId: 1

}

* **Resource Patching**

1. fetch('https://jsonplaceholder.typicode.com/posts/1', {
2. method: 'PATCH',
3. body: JSON.stringify({
4. title: 'foo',
5. }),
6. headers: {
7. 'Content-type': 'application/json; charset=UTF-8',
8. },
9. })
10. .then((response) =**>** response.json())
11. .then((json) =**>** console.log(json));

**Output**

{

id: 1,

title: 'foo',

body: '...',

userId: 1

}

* **Resource Deletion**

1. fetch('https://jsonplaceholder.typicode.com/posts/1', {
2. method: 'DELETE',
3. });

* **Resource Filtration**

1. // This will return all the posts that belong to the first user
2. fetch('https://jsonplaceholder.typicode.com/posts?userId=1')
3. .then((response) =**>** response.json())
4. .then((json) =**>** console.log(json));

* **Nested Resource listing**

1. fetch('https://jsonplaceholder.typicode.com/posts/1/comments')
2. .then((response) =**>** response.json())
3. .then((json) =**>** console.log(json));

## JSON Placeholder API Integration

To understand the API integration of JSON Placeholder, let's create a **'Todo' list** based on the data fetched. To add new data from a definite source, all you need is to integrate the Falcon platform. Falcon is not, but a data integration method makes any API integration easier. Therefore, the components you will use include API to fetch data, GraphQL for extension formatting, Queries to present the data as and when required, and the components that will allow you to render the frontend.

To move ahead on creating the application, you need to add everything to the client/src and server/src. It is recommended to use NPM packages so that functionalities can be reused. Therefore, the following steps would sum up the creation of the application.

First of all, you need to create a file with the source server/src/falcon-JSON placeholder-api/index.js. In this file, you will be requesting the API by creating the fetch requests from the source. Add the following code in the source as shown below.

1. const { ApiDataSource } = require('@deity/falcon-server-env');
2. module.exports = class JsonPlaceholderApi extends ApiDataSource {
3. async todoList(\_, { pagination }) {
4. const query = {};
5. if (pagination) {
6. query.\_limit = pagination.perPage;
7. query.\_start = pagination.page;
8. }
9. const todos = await this.get('todos', query);
10. return {
11. items: todos
12. }
13. }
14. async todo(\_, { id }) {
15. return this.get(`todos/${id}`);
16. }
17. };

The next steps involve adding GraphQL schema, and to do that; you need to create two files.

1. |-server
2. |-src
3. |-falcon-todos-extension
4. index.js
5. schema.graphql

Notice the naming convention give for the extension. It would be best to rename it based on the data type obtained from the source, unlike the API package. This is done to ensure that the source can be swapped at any point in time, but the extension remains the same along with address server/src/falcon-todos-extension/index.js.

1. extend type Query {
2. todoList(pagination: PaginationInput): TodoList! @cache(ttl: 15)
3. todo(id: ID!): Todo! @cache(ttl: 15)
4. }
5. type TodoList {
6. items: [Todo!]!
7. pagination: Pagination
8. }
9. type Todo {
10. userId: Int
11. id: ID
12. title: String
13. completed: Boolean
14. }

In the above code snippet, you can observe that each query should be uniquely defined, so accept and return go hand in hand. Another instance to note here is that the two queries are named todoList and todo. Now, you need to look at the server/src/falcon-jsonplaceholder-api/index.js, and you might notice that some of the methods share the same name, and the methods are automatically bound due to the same name.

The next step involves pagination. It is a technique to pass the input to the PaginationInput of the todoList. It is not defined here. It is solely defined in the **@diety/falcon-data/src/Pagination/Pagination.ts.** This source directory contains bits of information perPage and page. Consider the pagination technique as shown in the code below.

1. export type PaginationInput = {
2. perPage: number;
3. page: number;
4. ;

Now, you have to map these queries using the parameters present in the directory server/src/falcon-jsonplaceholder-api/index.js.

1. if (pagination) {
2. query.\_limit = pagination.perPage;
3. query.\_start = pagination.page \* pagination.perPage;
4. }
5. return this.get('todos', query);

The result can be viewed through the link given below.

1. https://jsonplaceholder.typicode.com/todos?\_limit=[perPage]&\_start=[page]

The next step includes the API extension. Now that you have added the files, you need to make sure falcon-server uses them properly and ensures all the resources are mapped together. It is done in the server/config/default.json file.

1. {
2. ...
3. "apis": {
4. ...
5. "jsonplaceholder": {
6. "package": "./src/falcon-jsonplaceholder-api/index.js",
7. "config": {
8. "host": "jsonplaceholder.typicode.com",
9. "protocol": "https"
10. }
11. }
12. },
13. "extensions": {
14. ...
15. "falcon-todos-extension": {
16. "package": "./src/falcon-todos-extension",
17. "config": {
18. "api": "jsonplaceholder"
19. }
20. }
21. }

In the above-given code snippet, you define the JSON for the API and pass this data to the extension config "api": "jsonplaceholder". Within the API, you might also observe a host and protocol defined, and they automatically get used by the function present in the directory server/src/falcon-jsonplaceholder-api/index.js.

Now, the final task involves testing your GraphQL. Since you are already set to test the extension, you need to have a falcon server running on port 4000, the default port. It would be best if you put down the queries here so that the results are returned.

## Final Thoughts

Regarding the application of JSON Placeholder, you might have across the fact that it has CORS and JSONP support with an integrated environment to be used anytime and from anywhere. JSON Placeholder is a dynamic testing machine that can be faked to test out certain aspects of the application that need some temporary data to be tested. Its sole purpose is to create prototypes for the testing environment that need some sample data to get followed up. Hence, JSON Placeholder creates various prototypes to serve this purpose with ease.