Using JSON Server and Postman to Mock Client/Server Communication

- Due No Due Date
- Points 1
- Submitting a website url

[(https://github.com/learn-co-curriculum/phase-1-using-json-server-and-postman)](https://github.com/learn-co-curriculum/phase-1-using-json-server-and-postman) [(https://github.com/learn-co-curriculum/phase-1-using-json-server-and-postman/issues/new)](https://github.com/learn-co-curriculum/phase-1-using-json-server-and-postman/issues/new)

Learning Goals

- Set up JSON Server as a mock backend
- Use Postman to mimic frontend responses
- Practice the client/server request/response cycle

Introduction

In typical full-stack applications, a frontend (the client) and a backend (the server) work together. The frontend initiates communication, often either asking for data or sending some data to be stored. The backend is actively listening for these requests, and when one is received, it will do some work for us and send a response back. This response may include requested data, or it could include a confirmation that data was stored. This request/response cycle is a critical piece of web development and the backbone of most modern websites.

In the next lessons, we'll start to explore the first half of this request/response cycle — initiating requests from the frontend. Before we start practicing in JavaScript, though, it would be helpful if we could explore how this cycle works. Luckily, we have some tools that can mimic both frontend requests and backend responses. For the frontend, we have **[Postman](https://www.postman.com/downloads/)** [(https://www.postman.com/downloads/)](https://www.postman.com/downloads/), an app that can be used to build requests without writing code. For the backend, we have **[JSON Server](https://www.npmjs.com/package/json-server)** [(https://www.npmjs.com/package/json-server)](https://www.npmjs.com/package/json-server), a Node application that mimics the behavior of a full backend server.

Combined, we can practice sending requests from Postman to the JSON server and see how the server responds.

Review: What is JSON Again?

JSON, JavaScript Object Notation, is a *data interchange format*. We use JSON to send structured data between frontends and backends. There are a few formats available to handle this task, but JSON has some specific advantages that make it a great choice for our purposes:

- It is human-readable. JSON data is stored as a `String` , but structured in a way that looks very similar to a JavaScript object.
- It is easy to convert into a JavaScript object. JavaScript has built-in methods for turning objects into JSON and vice versa. Very handy!
- Despite having JavaScript in the name, the format is compatible with many programming languages. Languages like Ruby and Python have their own methods for handling JSON and converting it into object-like data structures.

Below is an example of what a piece of JSON looks like when sent from client to server (or vice versa):

```
'{ "name": "Annie Easley", "occupation": "Computer Scientist" }'
```

The data above is a `String` , but you can see that it contains what looks like key/value pairs. Notice that the keys and values are both wrapped in quotes while other characters, `{` , `}` , and `:` , are not. This is required syntax for JSON. All text-based data, even keys, must be wrapped in quotes within the larger `String` . Numbers are the only exception to this.

When working with JSON, the outside quotes are not always shown. Instead of a single line like above, we may see JSON like this:

```
{
  "name": "Annie Easley",
  "occupation": "Computer Scientist"
}
```

If you are curious about JSON syntax, there are many JSON validators online like **[this one](https://jsonlint.com/) (https://jsonlint.com/)** that will confirm if your JSON is formatted correctly.

What is JSON Server?

JSON Server is a freely available Node package that can turn a JSON file on your computer into mock data storage. When JSON Server is running, we can send requests to get data from storage or add data to it, as though we were talking to a server with a database.

A huge benefit of JSON Server is that we don't have to spend much time setting the mock server up, allowing us to focus on developing the frontend of an application first.

Setting up JSON Server

To start using JSON Server, we need to install it, then provide a basic set of data to practice with.

First, we'll install JSON Server globally on your machine:

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

With the command above, you should now be able to spin up a mock server from any directory on your computer. Alternatively, if you remove the `-g` option from this command but are in a folder with a `package.json` file, `json-server` will be added as a dependency in the file.

> **Note**: For users of the **Live Server VSCode extension (https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer)**, you'll need to do a bit of extra configuration so that the `json-server` plays nicely with Live Server in future lessons. Follow the steps in **this gist (https://gist.github.com/ihollander/cc5f36c6447d15dea6a16f68d82aacf7)** (you'll only need to do this once), then come back to this lesson.

Next, we'll need to create a file that will act as our data storage.

```
$ touch db.json
```

Open this file in your text editor and paste in the following content:

```
{
  "articles": [
    {
      "id": 1,
      "title": "Example Article",
      "content": "This is an example."
    },
    {
      "id": 2,
      "title": "Second Article",
      "content": "This is also an example."
    }
  ]
}
```

Here, we've created one top-level key, `"articles"` , in our JSON, which points to an array. This array contains two elements, both objects with three keys: `"id"` , `"title"` , and `"content"` . Our first goal will be to access this data.

Start the Server

To start JSON Server, run the following command from the same directory that `db.json` is in:

```
$ json-server --watch db.json
```

When run, you'll see some messaging about how to access our JSON data. By default, JSON Server will start up on port `3000` . You should see a notice that you can access the server at `http://localhost:3000` .

Open your browser and paste this URL in. If the server is running correctly, you should be presented with a page of information provided by JSON Server. On this page, you'll see a **Resources** section that lists one resource: `/articles` . The server has read the `db.json` file and found our `articles` key, turning it into a resource. Click `/articles` and you will be navigated to a new page, `http://localhost:3000/articles` . Instead of a page of info, you'll see the value associated with `articles` in our data, an array containing two objects:

```
[
  {
    "id": 1,
    "title": "Example Article",
    "content": "This is an example.",
  },
  {
    "id": 2,
    "title": "Second Article",
    "content": "This is also an example.",
  },
];
```

> **Note**: It's recommended that you use a browser extension such as **JSON Viewer (https://chrome.google.com/webstore/detail/json-viewer/gbmdgpbipfallnflgajpaliibnhdgobh?hl=en-US)** to format the JSON string in the browser.

We can go even further — notice the `"id"` key that is listed. Instead of just going to `/articles` , we can append the value of `"id"` to the end of the URL:

```
http://localhost:3000/articles/1
```

Now, instead of an array, we get the object inside of it:

```
{
  "id": 1,
  "title": "Example Article",
  "content": "This is an example."
}
```

Neat! So what is happening? We won't go into too much detail, but JSON server is following **RESTful conventions (https://en.wikipedia.org/wiki/Representational_state_transfer)**. By providing `/articles` followed by `/1` in our URL, JSON Server knows we're asking for a resource called `articles` , and within that resource, we're asking for whatever data has an ID of `1` . The `articles` content we store in our JSON file could be in any order. JSON Server will look through and match the request to an ID and return *that* content. If we change to `2` , we'll get the other data we stored in `articles` .

Leave JSON server running and we'll move on to the next tool, Postman.

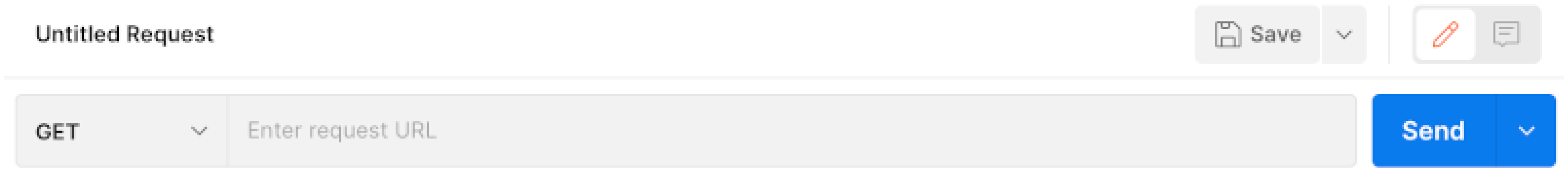
What is Postman?

As we mentioned, Postman is an application that allows us to mock up frontend requests without writing any JavaScript. With Postman, we can practice sending requests to our JSON Server.

Setting up Postman

To get the Postman app, head over to https://www.postman.com/downloads/ (https://www.postman.com/downloads/) and click **Download the App**. There is a web version of Postman, but this will not work with our `localhost` server.

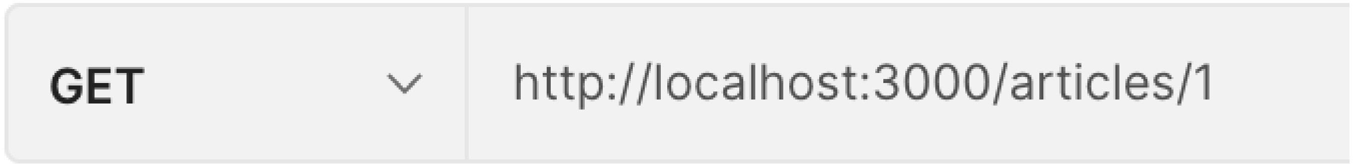
Once it's downloaded and installed, open the app. You should see a screen inviting you to create an account or sign in. At the bottom of that screen, click the "Skip and go to the app" link. On the next screen, you should see a "Get Started" section on the right side; click the first option: "Create a request". You should then see an input field starting with **GET** and containing the placeholder text *Enter request URL*.



We're now ready to send requests to our server.

Retrieving Data from our JSON Server using Postman

Here, we'll write in the URL we previously used to get our JSON server data, `http://localhost:3000/articles/1` .



Once entered, hit the **Send** button. If everything is working, you should see the same article data from earlier, an object with three keys: `"id"` , `"title"` , and `"content"` . You're now performing the full request/response cycle using our tools! Let's explore what is happening.

When you click **Send** on Postman, you send a request to the URL you provided. This is a **GET** request — a request for data from a resource. Our JSON server is actively listening for these requests. If you look at your terminal where JSON server is running, you will see that the server has recognized your GET request, displaying something similar to this:

```
GET /articles/1 200 25.666 ms
```

JSON Server sees that this is a GET request. It also notes that the request is specifically for `/articles/1` . `200` is a **HTTP status code (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status)** that indicates the request was received, accepted and responded to successfully. `25.666 ms` is the amount of time it took to complete the request, in milliseconds. Back in Postman, we can see the response from our server in the lower panel and confirm we received what we expected.

Depending on what we need, we can change out the details of our request. Imagine we are building a local news site containing many articles. Instead of requesting just the article with an ID of `1` , we might just send a request for `/articles` and get everything available from the server. In complex webpages, we may send requests to both depending on what page is being accessed — we might have an index page of all article titles, and when a title is clicked, we'd send a request for a single article.

Sending Data to our JSON Server from Postman

We've now seen how a GET HTTP request works, so let's move on to a POST request. POST requests are used when we want to *send data to* a server.

> **Note:** There are a couple of other HTTP request options for sending data — PUT and PATCH — but for simplicity, we'll just focus on POST requests for now.

Continuing our news site example, let's say we've written a new article and want to add it to the site. A POST request allows us to send the contents of this new article, along with any other details we want to include, like the title. As long as we've structured the request correctly, JSON Server will receive the request, recognize it as a POST request and attempt to store the article information in the 'database.'

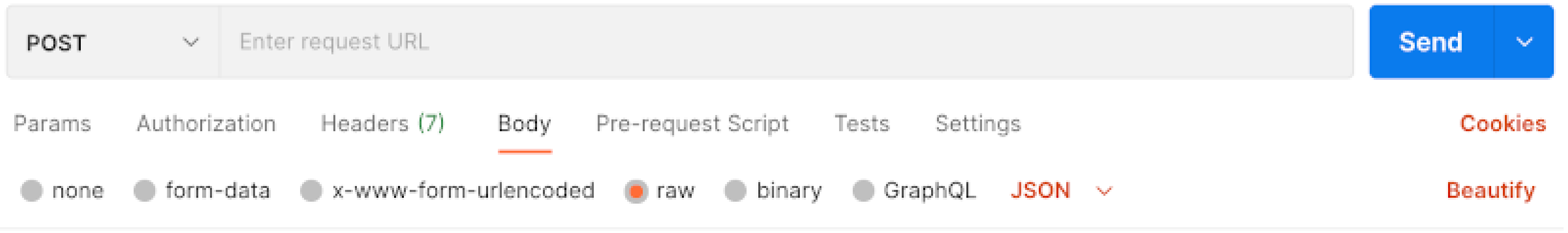
To send data to our server, first, we need to switch our Postman request from GET to POST. Click on GET beside the URL bar to display a drop-down menu of HTTP request options and switch over to POST.

Second, we need to adjust the URL we're using. In this particular case, we're sending content that should become a *new* article. Because of this, we don't want to use a specific ID value in the URL. Instead, we'll send a request to `/articles` :

```
http://localhost:3000/articles
```

When JSON Server receives the request, it'll recognize it as a POST request and automatically add it to the appropriate resource. It'll also assign an ID for us, so we don't need to worry about including one.

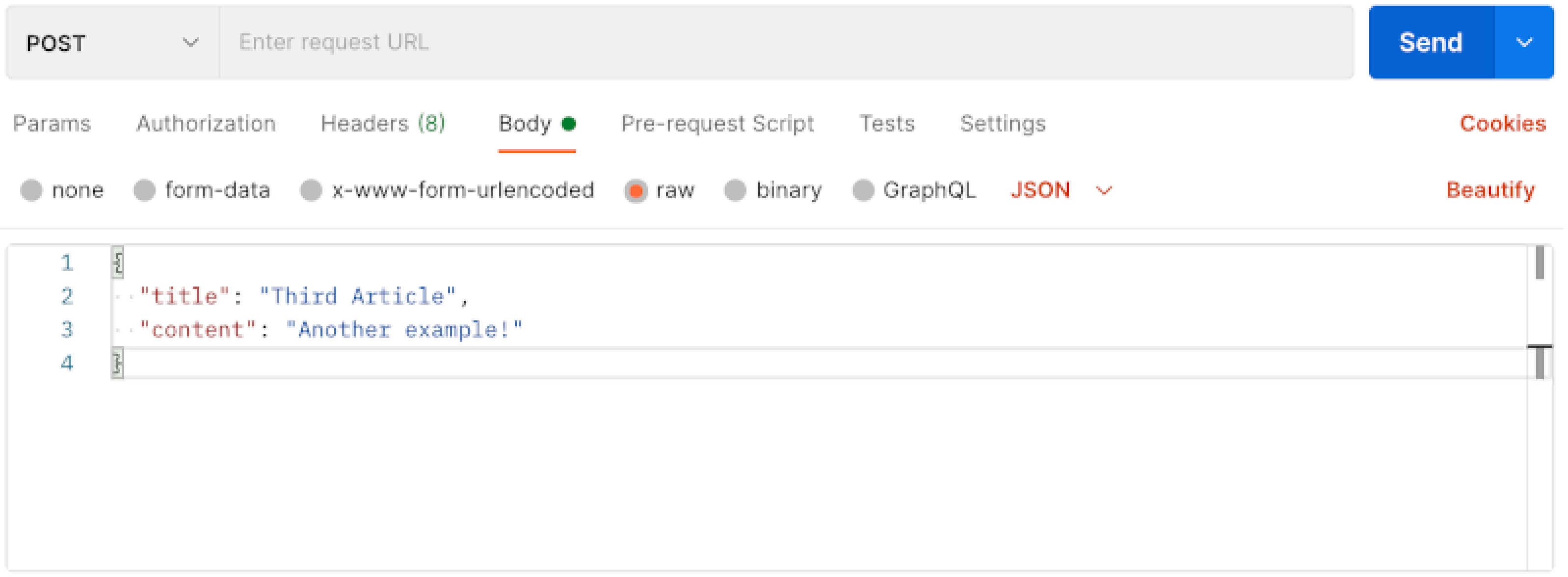
Finally, before we can send our request, we need to provide the data we want to send. In Postman, just below the URL bar, click the **Body** tab, then choose the **raw** option, and select **JSON** from the drop-down menu.



In the code box just below these options, write in the following JSON:

```
{
  "title": "Third Article",
  "content": "Another example!"
}
```

Note that we don't need to wrap the contents in quotes and left out the ID key/value. Postman will handle these for us.



```
{
  "title": "Third Article",
  "content": "Another example!"
}
```

When ready, click **Send**. In the terminal, we should see JSON Server recognizing the request. In Postman, we'll see the server's response in the lower panel:

```
{
  "title": "Third Article",
  "content": "Another example!",
  "id": 3
}
```

Typically, after a successful POST request, the server will send back the new data as a response. In this case, it sent back what we sent *and* included the newly assigned ID.

As one final confirmation, navigate to your `db.json` file and open it up. You should see that the file has changed to include your newly submitted content! Congratulations, you've persisted data to `db.json` !

Submit Using CodeGrade

Once you're done, be sure to commit and push your code up to GitHub, then submit the assignment using CodeGrade. Even though this lab does not have tests, it must still be submitted through CodeGrade in order to be marked as complete in Canvas.

Conclusion

Although we haven't learned how to build our own backends yet, JSON server can act as a placeholder, enabling us to learn the first half of the request/response cycle without having to worry about a backend. Also, because JSON Server follows RESTful conventions that are widely used throughout the internet, we'll start to become familiar with how server resources *should* be structured long before we create our own.

We encourage you to try creating your own resources in `db.json` . A few quick notes about setting resources up:

- Any top-level key/value pair (like `articles` in our example) will be automatically treated as a resource we can retrieve and send data to.
- The value should be either an array or an object. Try both to see how they differ!

With JSON Server, you'll now be able to design frontends that persist data!

Resources

- **JSON Server (https://www.npmjs.com/package/json-server)**
- **Postman (https://www.postman.com/downloads/)**
- **HTTP Status Codes (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status)**
- **Representational State Transfer (REST) (https://en.wikipedia.org/wiki/Representational_state_transfer)**