In this tutorial, you'll use the `useEffect` and `useState` Hooks to fetch and display information in a sample application, using JSON server as a local API for testing purposes. You'll load information when a component first mounts and save customer inputs with an API. You'll also refresh data when a user makes a change and learn how to ignore API requests when a component unmounts. By the end of this tutorial, you'll be able to connect your React applications to a variety of APIs and you'll be able to send and receive real-time data.

Prerequisites

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.22.0 and npm version 6.14.6. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.
- A React development environment set up with Create React App, with the non-essential boilerplate removed. To set this up, follow **Step 1 — Creating an Empty Project** of the How To Manage State on React Class Components tutorial. This tutorial will use `api-tutorial` as the project name.
- You will be using React components and Hooks in this tutorial, including the `useState` and `useEffect` Hooks. You can learn about components and Hooks in our tutorials How To Manage State with Hooks on React Components and How To Handle Async Data Loading, Lazy Loading, and Code Splitting with React.
- You will also need a basic knowledge of JavaScript and HTML, which you can find in our How To Build a Website with HTML series and in How To Code in JavaScript. Basic knowledge of CSS would also be useful, which you can find at the Mozilla Developer Network.

Step 1 — Creating a Project and a Local API

In this step, you'll create a local REST API using JSON server, which you will use as a test data source. Later, you'll build an application to display a grocery list and to add items to the list. JSON server will be your local API and will give you a live URL to make `GET` and `POST` requests. With a local API, you have the opportunity to prototype and test components while you or another team develops live APIs.

By the end of this step, you'll be able to create local mock APIs that you can connect to with your React applications.

On many agile teams, front-end and API teams work on a problem in parallel. In order to develop a front-end application while a remote API is still in development, you can create a local version that you can use while waiting for a complete remote API.

There are many ways to make a mock local API. You can create a simple server using Node or another language, but the quickest way is to use the JSON server Node package. This project creates a local REST

API from a JSON file.

To begin, install `json-server`:

```
$ npm install --save-dev json-server
```

When the installation is finished, you'll receive a success message:

```
Output
+ json-server@0.16.1
added 108 packages from 40 contributors and audited 1723 packages in 14.505s

73 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

`json-server` creates an API based on a JavaScript object. The keys are the URL paths and the values are returned as a response. You store the JavaScript object locally and commit it to your source control.

Open a file called `db.json` in the root of your application. This will be the JSON that stores the information you request from the API:

```
$ nano db.json
```

Add an object with the key of `list` and an array of values with an `id` and a key of `item`. This will list the item for the grocery list. The key `list` will eventually give you a URL with an endpoint of `/list`:

api-tutorial/db.json

```
{
  "list": [
    { "id": 1, "item": "bread" },
    { "id": 2, "item": "grapes" }
  ]
}
```

In this snippet, you have hard-coded `bread` and `grapes` as a starting point for your grocery list.

Save and close the file. To run the API server, you will use `json-server` from the command line with an argument point to the API configuration file. Add it as a script in your `package.json`.

Open `package.json`:

```
$ nano package.json
```

Then add a script to run the API. In addition, add a `delay` property. This will throttle the response, creating a delay between your API request and the API response. This will give you some insights into how the application will behave when waiting for a server response. Add a `delay` of `1500` milliseconds. Finally, run the API on port `3333` using the `-p` option so it won't conflict with the `create-react-app` run script:

api-tutorial/package.json

```
{
  "name": "do-14-api",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^4.2.4",
    "@testing-library/react": "^9.3.2",
    "@testing-library/user-event": "^7.1.2",
    "react": "^16.13.1",
    "react-dom": "^16.13.1",
    "react-scripts": "3.4.3"
  },
  "scripts": {
    "api": "json-server db.json -p 3333 --delay 1500",
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": "react-app"
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  },
  "devDependencies": {
    "json-server": "^0.16.1"
  }
}
```

Save and close the file. In a new terminal or tab, start the API server with the following command:

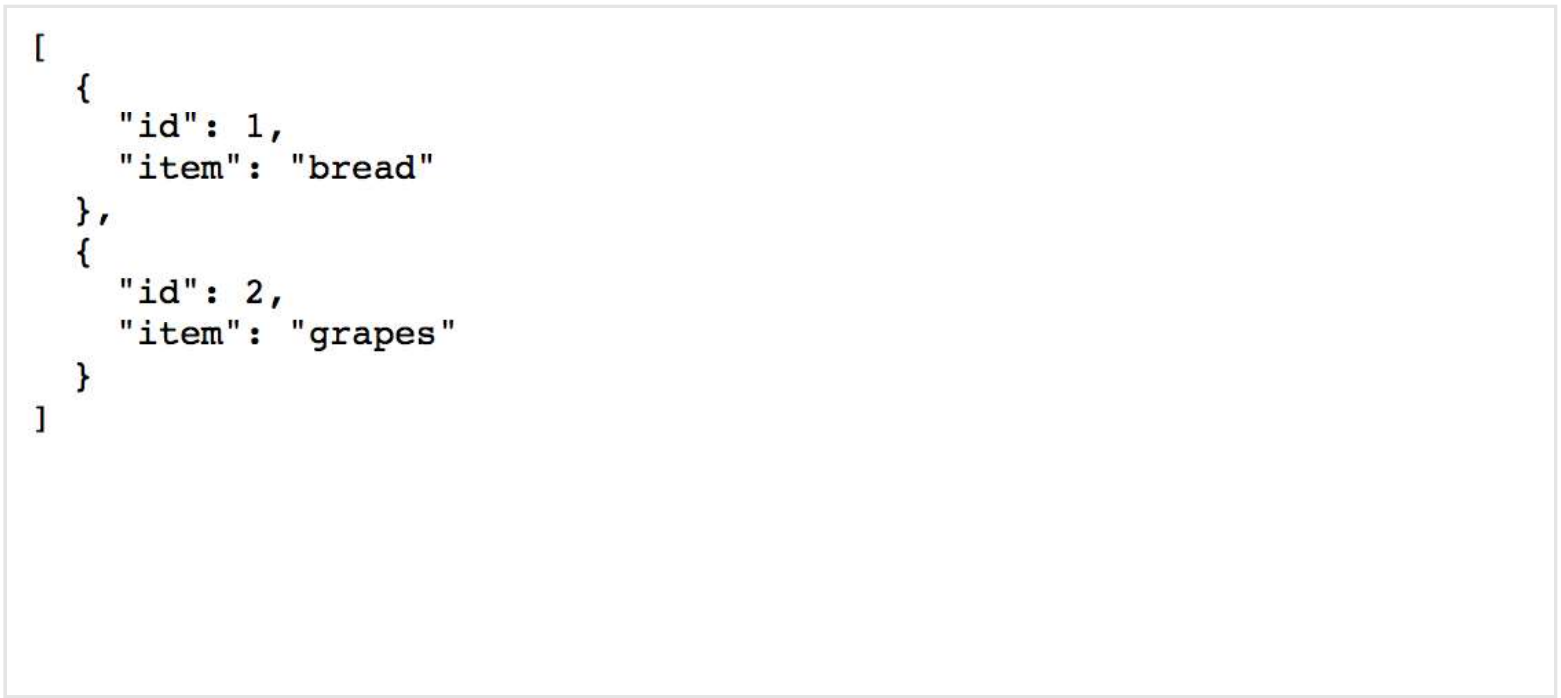
```
$ npm run api
```

Keep this running during the rest of the tutorial.

When you run the command, you will receive an output that lists the API resources:

```
Output
> json-server db.json -p 3333


  \{^_^}/ hi!

  Loading db.json
  Done

  Resources
  http://localhost:3333/list

  Home
  http://localhost:3333

  Type s + enter at any time to create a snapshot of the database
```

Open `http://localhost:3333/list` and you'll find the live API:



When you open an endpoint in your browser, you are using the `GET` method. But `json-server` is not limited to the `GET` method. You can perform many other REST methods as well. For example, you can `POST` new items. In a new terminal window or tab, use `curl` to `POST` a new item with a type of `application/json`:

```
$ curl -d '{"item":"rice"}' -H 'Content-Type: application/json' -X POST http://localhost:3333/list
```

Note that you must stringify the content before you send it. After running the `curl` command, you'll receive a success message:

```
Output
{
  "item": "rice",
  "id": 3
}
```

If you refresh the browser, the new item will appear:

The `POST` request will also update the `db.json` file. Be mindful of the changes, since there are no barriers to accidentally saving unstructured or unhelpful content as you work on your application. Be sure to check any changes before committing into version control.

In this step, you created a local API. You learned how to create a static file with default values and how to fetch or update those values using RESTful actions such as `GET` and `POST`. In the next step, you'll create

services to fetch data from the API and to display in your application.

Step 2 — Fetching Data from an API with `useEffect`

In this step, you'll fetch a list of groceries using the `useEffect` Hook. You'll create a service to consume APIs in separate directories and call that service in your React components. After you call the service, you'll save the data with the `useState` Hook and display the results in your component.

By the end of this step, you'll be able to call web APIs using the Fetch method and the `useEffect` Hook. You'll also be able to save and display the results.

Now that you have a working API, you need a service to fetch the data and components to display the information. Start by creating a service. You can fetch data directly inside any React component, but your projects will be easier to browse and update if you keep your data retrieval functions separate from your display components. This will allow you to reuse methods across components, mock in tests, and update URLs when endpoints change.

Create a directory called `services` inside the `src` directory:

```
$ mkdir src/services
```

Then open a file called `list.js` in your text editor:

```
$ nano src/services/list.js
```

You'll use this file for any actions on the `/list` endpoint. Add a function to retrieve the data using the `fetch` function:

api-tutorial/src/services/list

```
export function getList() {
  return fetch('http://localhost:3333/list')
    .then(data => data.json())
}
```

The only goal of this function is to access the data and to convert the response into JSON using the `data.json()` method. `GET` is the default action, so you don't need any other parameters.

In addition to `fetch`, there are other popular libraries such as Axios that can give you an intuitive interface and will allow you to add default headers or perform other actions on the service. But `fetch` will work for most requests.