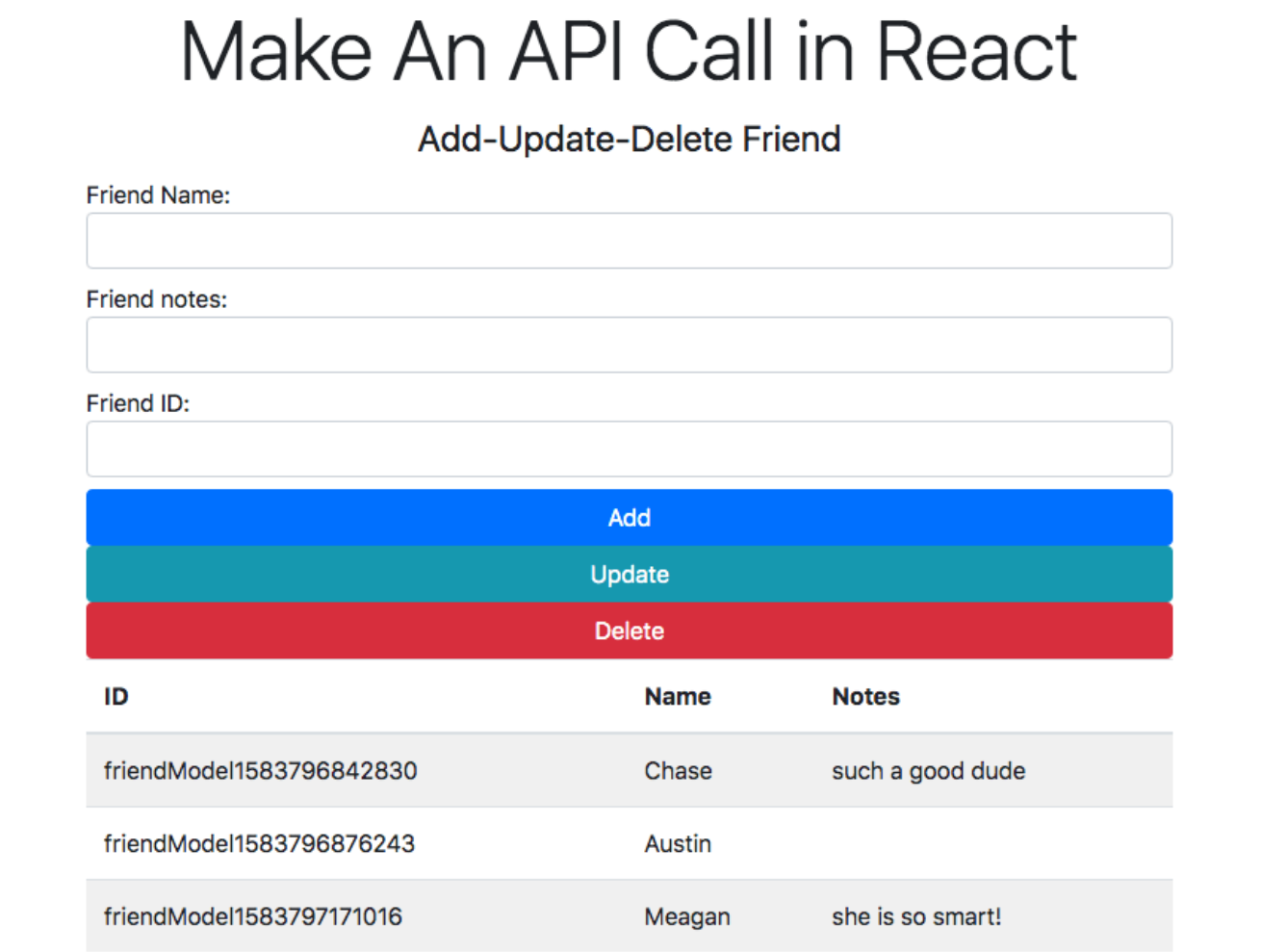


How To Use an API with ReactJS

Last Updated on September 14, 2020 by Jarrett Retz 4 Comments

Table of Contents [show]

Intro

A client recently asked me, “Do you know how to use APIs?”.

Frankly, I hesitated when answering because I thought,

“What do *you mean* by API?”.

There wasn’t enough context for me to figure out what kind of Application Programming Interface (API) she was talking about, and you can learn more about the various types of APIs here.

Consider this, my favorite definition from the article linked above concerning different types of APIs is;

> *“…an API provides interactions between one software and another, but not users.*

Given a broad definition, asking a developer if they know how to use APIs is like asking a chef if they know to use the stove.

On rare occasions, they may not.

However, truthfully it depends on the type of stove and what they are using it for.

If you grant me the comparison: using an API depends on the API and what I am using it for.

Web APIs

RapidAPI is a platform for accessing web-based APIs.

The most popular type of web API is a Representational state transfer API or RESTful API for short. To be brief, it follows an architecture that uses predefined and stateless operations to access web resources.

Using web APIs requires the use of HTTP requests. Different types of requests include GET, POST, DELETE, PATCH, etc. If you have written some code and it needs to communicate with code somewhere else, it will send an HTTP request over the internet. Next, let's discuss the HTTP request types just mentioned. You will notice that the types are matched to verbs that are similar to their function:

- **GET:** Used to request data from an endpoint
- **POST:** Sends data to an endpoint
- **DELETE:** Remove data from an endpoint.
- **PATCH**: Update a record or data value at an endpoint.

These types of interactions are external from the local environment, and therefore, the APIs that are providing the data or services are called **external APIs.** It is difficult to fathom how useful it is to have the capability to access external APIs for various data and services.

API Keys

As previously mentioned, external APIs offer data, services, or both. Having access to the API can be valuable, therefore, you may need to buy a subscription to use it. Don't worry, there are still plenty of free APIs.

Furthermore, the hypothetical external API does not have unlimited resources, so letting software call the API an unlimited amount of times at no cost could render it useless. Consequently, API providers will issue secret API-keys to monitor usage over time. Using RapidAPI allows the developer to only deal with one key (RapidAPI's key), otherwise, developers would need to have API-keys for all of their APIs. Most of the time API-keys need to be kept secret, however, some services will issue public API keys.

Soon we will learn how to use the RapidAPI dashboard to simplify our understanding of how these parts work together, but first, let's talk about React!

What is React?

> *A JavaScript library for building user interfaces*
>
> **ReactJS.org**

The ReactJS library is useful for handling the relationship between **views**, **state**, and **changes in state**.

Views are what the user sees rendered in the browser. **State** refers to the data stored by different views that typically rely on who the user is, or what the user is doing. For example, when you sign into a website it may show your user profile (view) with your name (state).

The state data may change user-to-user but the view remains the same. Expanding on our example, let's suppose the website is an e-commerce site. If you are signed in, the website may have a *Products You Might Like* section. However, if you are not signed in, rendering that view may not make sense. ReactJS helps developers manage that kind of scenario with ease.

The sections below will be a brief introduction to ReactJS. For a more comprehensive tutorial, you can visit the ReactJS.org tutorial page.

Components

Views are created inside of components. There two types of components: class and function. Below is a class component:

```
1.  class ExampleComponent extends React.Component {
2.    constructor(props) {
3.      super(props);
4.      this.state = { };
5.    }
6.
7.    render() {
8.      return (
9.        <div>
10.         <h1>This is a view created by a class component</h1>
11.       </div>
12.     );
13.   }
14. }
```

Notice that we declare a state object in the constructor. This where we can store data to be used inside the view. Speaking of the view, it is rendered to the page inside of `render()`.

A function component may look like:

```
1.  const ExampleComponent = (props) => {
2.      const [stateVariable, setStateVariable] = useState('');
3.
```

```
    return (
        <div>
            <h1>This is a function component view</h1>
        </div>
    )
}
```

Unlike the class component, our function returns the view and does not need a render function. Also, the state is controlled by the React hook function `useState`.

Later on, we will use class components in the example app.

Props

We can pass data or functions down into components through **props**. Here's an example:

```
import React, { useState } from 'react'
import ReactDOM from 'react-dom'

const ParentComponent = () => {
    const [stateVariable, setStateVariable] = useState('this is the starting value for the variable');

    return (
        <div>
            <h1>This is a function component view</h1>
            <ChildComponent exampleProp={stateVariable} />
        </div>
    )
}

const ChildComponent = (props) => {
    return (
        <div>
            <h2>{props.exampleProp}</h2>
        </div>
    )
}
```

```
24.  ReactDOM.render( <ParentComponent />,
     document.getElementById('app') );
```

In the above example, the string `'this is the starting value for the variable'` is rendered to the page in replace of `{props.exampleProp}`.

The child component is rendered inside of the parent component and can inherit data passed down to through properties.

This a confusing concept at first because it does not read procedurally, but it will make more sense when we start using actual data. It's a big reason why React can be so powerful.

JSX

JSX looks like an HTML templating language but is more powerful. We already saw how JSX can embed data into a view using curly brackets during our previous example. It looks very similar to HTML but has many differences. A few are:

- passing custom props as attributes on an element tag
- Instead of assigning CSS classes with `class=`, JSX uses `className=`. Camel casing is used for event listeners as well (I.e `onClick, onHover`).
- closing tags are different depending on the elements. Below are a few examples.

HTML input and image tags:

```
<img class="" src="" alt="" >
```

```
<input type="text" required>
```

JSX input, and image tags:

```
<img className="" src="" alt="" />
```

```
<input type="text" required />
```

In addition to the above subtle changes, JSX tags can be self-closing. A valid JSX element could be.

```
<div className="empty-div" />
```

You can learn more about JSX on React's documentation page.

Side Effects

Side effects occur when a function goes outside of its scope or it is affected by something outside of the function body. **Making an HTTP request and saving the result to the components state is a side effect.**

Lifecycle Methods

In React, components have methods that are executed during the different phases of the components. These methods are called lifecycle methods.

It takes practice to figure out where and how certain lifecycle methods should be used. Regardless, the most important for HTTP requests is `componentDidMount` with class components and `useEffect` with functional components.

Grabbing the examples above, let's add the methods we are talking about to the components.

Class component:

```
import React from 'react'
import ReactDOM from 'react'

class ExampleComponent extends React.Component {
    constructor(props) {
        super(props);
        this.state = { };
    }

    componentDidMount() {
        // send HTTP request
        // save it to the state
    }

    render() {
        return (
            <div>
                <h1>This is a view created by a class component</h1>
            </div>
        );
    }
}

ReactDOM.render( <ExampleComponent />, document.getElementById('app') );
```

Function component:

```
import React, { useState, useEffect } from 'react'
import ReactDOM from 'react-dom'

const ExampleComponent = (props) => {
    const [stateVariable, setStateVariable] = useState('');

    useEffect(() => {
        // send HTTP request
        // save response to variable
    }, [])

    return (
        <div>
            <h1>This is a function component view</h1>
        </div>
    )
```

```
  }

  ReactDOM.render( <ExampleComponent />,
document.getElementById('app') );
```

These methods are used because they are called immediately after the component is mounted to the DOM (Document Object Model).

Here an example of how this works:

1. User visits webpage
2. The component is mounted to the DOM (webpage)
3. The `componentDidMount` or `useEffect` method is called sending off the HTTP request
4. The webpage gives an indication it is loading data
5. The data is received from the external API and added to state (side effect)
6. The component renders with the data that was fetched.

That was a bit of a crash course, and I do not expect you to have a full understanding of React. However, in this circumstance, the best way to learn is to build it! Therefore, let's get started!

Prerequisites

Let's make sure that you are fully prepared to advance and build out API calls using ReactJS. Below are a few prerequisites:

- RapidAPI account. It's free to sign-up and the API we use in the example below is free as well!
- An internet connection and a browser that isn't Internet Explorer. IE has some compatibility issues that can be easily avoided. I recommend Firefox or Google Chrome.
- A reputable text editor. I use Visual Studio Code, however, another popular example is Sublime Text.
- A general understanding of HTML, CSS, and Javascript.

- Knowledge of how to use the command-line on your operating system. The commands are simple and should be similar across systems, however, there might be a slight variation. Looking up the appropriate command should be quick and easy.
- NodeJS and NPM installed locally. There are installation instructions on NodeJS.org.

NPM stands for Node Package Manager. Unsurprisingly, it is the package manager for ReactJS and NodeJS packages. We will use it to download React and other dependencies.

How to Fetch/Call an API with React

View Repository on Github

In this example, we will learn how to send HTTP requests to a database to get, create, update and delete external data using an API on RapidAPI with React. Then, we will learn how to display this data on the webpage. Finally, I will point you towards more advanced examples so you can continue to grow!

1. Create a Basic Project Structure

Make a new folder. I named mine `react-api-call`.

Open up your text editor inside of the new folder and navigate into the new folder with your terminal.

Create the following folders:

- public
- src

Inside `public` create the file `index.html` and add the following code to it.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initia
    <title>Call Api</title>

    <!-- Load bootstrap -->
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/b
integrity="sha384-
Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9If
crossorigin="anonymous">

</head>
<body>
    <div id="App"></div>

  <!-- from https://reactjs.org/docs/add-react-to-a-website.h
  <!-- Load React. -->
  <!-- Note: when deploying, replace "development.js" with
"production.min.js". -->
  <script src="https://unpkg.com/react@16/umd/react.developme
crossorigin></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.d
crossorigin></script>

  <!-- Load our React component. -->
  <script src="js/app.js"></script>

  <!-- Load bootstrap js and dependencies -->
  <script src="https://code.jquery.com/jquery-3.4.1.slim.min.
integrity="sha384-
J6qa4849blE2+poT4WnyKhv5vZF5SrPo0iEjwBvKU7imGFAV0wwj1yYfoRSJoZ
crossorigin="anonymous"></script>
  <script
src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/po
integrity="sha384-
Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfoo
crossorigin="anonymous"></script>
  <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/boo
integrity="sha384-
```

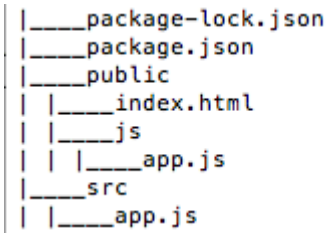
```
    wfSDF2E50Y2D1uUdj0O3uMBJnjuUD4Ih7YwaYd1iqfktj0Uod8GCExl3Og8ifw
    crossorigin="anonymous"></script>
  </body>
  </html>
```

There is not much going on in this HTML file. We created a container for the app with an id of 'App' and loaded in:

- Bootstrap for styling
- React and React-DOM in the body Javascript
- React component (not created yet)

Next, inside of the `public` folder, create the folder `js`. This will be an empty folder.

After that, inside of `src` folder create the empty file `app.js`. Your project structure should look like



The above picture does not include the `node_modules` folder that should be in the project root as well.

2. Add React Component

Back in the terminal run these two commands:

1. `npm init -y`: Creates an npm package in our project root
2. `npm install babel-cli@6 babel-preset-react-app@3`: Installs the packages we need to convert JSX to HTML

Because JSX isn't compatible with vanilla Javascript & HTML there needs to be a way to compile it into something that is. We installed Babel packages above to handle the translation.

In `src/app.js` add the following code:

```
'use strict';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      friends: [],
      name: '',
      id: '',
      notes: ''
    };

    this.create = this.create.bind(this);
    this.update = this.update.bind(this);
    this.delete = this.delete.bind(this);
    this.handleChange = this.handleChange.bind(this);
  }

  componentDidMount() {
    // get all entities - GET

  }

  create(e) {
    // add entity - POST
    e.preventDefault();

  }

  update(e) {
    // update entity - PUT
    e.preventDefault();

  }

  delete(e) {
    // delete entity - DELETE
```

```
38.       e.preventDefault();
39.
40.     }
41.
42.     handleChange(changeObject) {
43.       this.setState(changeObject)
44.     }
45.
46.     render() {
47.       return (
48.           <div className="container">
49.             <div className="row justify-content-center">
50.               <div className="col-md-8">
51.                 <h1 className="display-4 text-center">Make An
    API Call in React</h1>
52.                 <form className="d-flex flex-column">
53.                   <legend className="text-center">Add-Update-
    Delete Friend</legend>
54.                   <label htmlFor="name">
55.                     Friend Name:
56.                     <input
57.                       name="name"
58.                       id="name"
59.                       type="text"
60.                       className="form-control"
61.                       value={this.state.name}
62.                       onChange={(e) => this.handleChange({
    name: e.target.value })}
63.                       required
64.                       />
65.                   </label>
66.                   <label htmlFor="notes">
67.                     Friend notes:
68.                     <input
69.                       name="notes"
70.                       id="notes"
71.                       type="test"
72.                       className="form-control"
73.                       value={this.state.notes}
74.                       onChange={(e) => this.handleChange({
    notes: e.target.value })}
75.                       required
76.                       />
77.                   </label>
78.                   <label htmlFor="id">
79.                     Friend ID:
```

```
                    <input
                      name="id"
                      id="id"
                      type="text"
                      className="form-control"
                      value={this.state.id}
                      onChange={(e) => this.handleChange({ id:
    e.target.value })}
                      />
                  </label>
                  <button className="btn btn-primary"
    type='button' onClick={(e) => this.create(e)}>
                    Add
                  </button>
                  <button className="btn btn-info"
    type='button' onClick={(e) => this.update(e)}>
                      Update
                  </button>
                  <button className="btn btn-danger"
    type='button' onClick={(e) => this.delete(e)}>
                      Delete
                  </button>
                </form>
              </div>
            </div>
          </div>
      );
    }
  }

  let domContainer = document.querySelector('#App');
  ReactDOM.render(<App />, domContainer);
```

Notice we have inserted empty functions to fill with APIs calls and bound them to the component in the `constructor` function. Also, we initialized our state and defined variables that we need to keep track of.

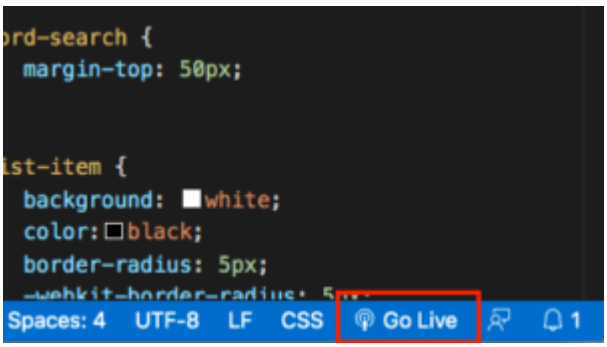
Taking a look at the JSX, you notice that we are using camel-case (camelCase) to define attributes and properties. Furthermore, we are controlling the inputs by assigning them to values in state and passing an `onChange` handler.

Next, we need to set up a live server to make the development process easier to understand.

If you are using VSCode you can download the plugin Live Server. This allows you to watch your changes update automatically.

If you are not using VSCode there is most likely an easy way to get the same functionality with your text editor with the help of your favorite search engine.

After installing the plugin in VSCode, select the 'Go Live' button which will start the application on a port locally (i.e http://localhost:5050). The browser gives you the option to select the directory to run the server in choose `public`.



Although the site is live, nothing will display because we are not compiling our React code in the `src` directory.

In the terminal run:

```
npx babel --watch src --out-dir public/js --presets react-app/prod
```

The command will automatically compile the code in `src` and place it into the appropriately named folder in `public/js`.

The output in your terminal should be:

```
src/app.js -> public/js/app.js
```

Now, you should see our React file code appearing in the browser.

3. Add API Calls

Our interaction with an API depends on that specific API. It may only offer certain types of HTTP methods, endpoints, and parameters. Whatever the case we need instructions on what that API expects our requests to look like. I have chosen to use the **FaiRESTdb API** for this example so I can demonstrate the four main types of API calls that we should know (GET, POST, PUT, DELETE).

Let's explore the dashboard tool.

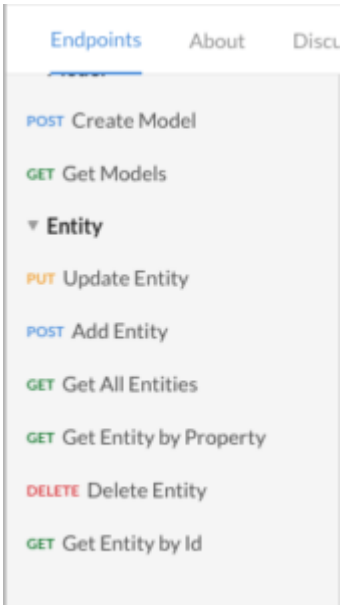
Open Up FaiRESTdb API on RapidAPI

Search for the API on the marketplace or follow the link to the API dashboard. In the dashboard you can find:

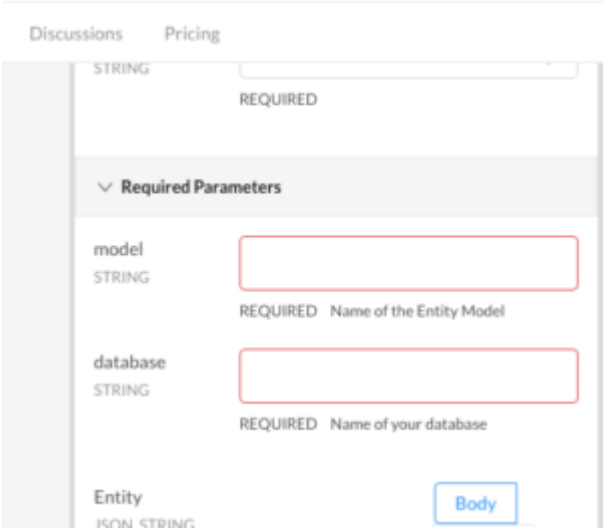
- endpoint details

- pricing
- documentation
- community discussions

The different types of endpoints that are available can be found on the left side of the interactive dashboard.



Depending on which endpoint is selected, a definition and parameter information will be available in the center of the dashboard.



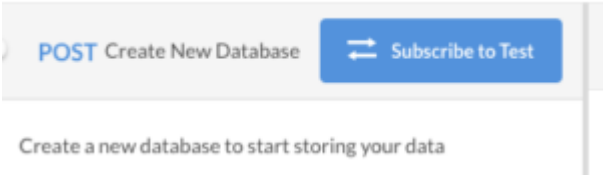
Finally, the right side of the dashboard builds code snippets based on the endpoint and the parameters that we enter. In this way, the dashboard works from left-to-right.

In addition to ready-to-use code snippets, the right side of the dashboard displays sample response objects that are important to planning our application. Most APIs allow us to test endpoints and receive real response data right in the API dashboard.

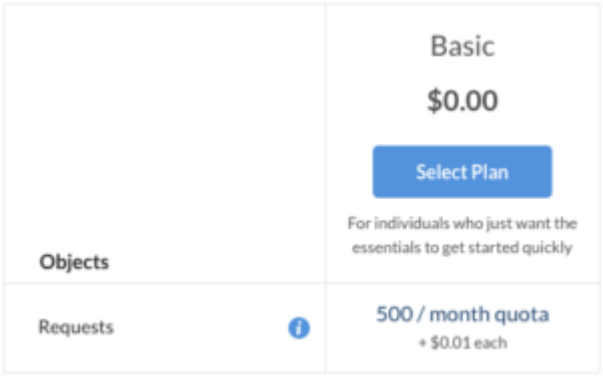
Response objects consist of the data the API returns to our request. Response objects can be JSON, XML, audio/video etc. However, the API we are using will respond in a JSON format. JSON stands for Javascript Object Notation and is a lightweight data exchange framework.

We can create a database for our sample application in the dashboard after subscribing because the tests are real requests.

Click the blue '**Subscribe to Test**' button on the dashboard.



The pricing panel shows us the available plans and quota restrictions for the API. Notice, we get 500 calls a month, then it's $0.01 for each subsequent call. You may think that would be difficult to keep track of, however, RapidAPI has a developer dashboard that allows you to track your API calls!

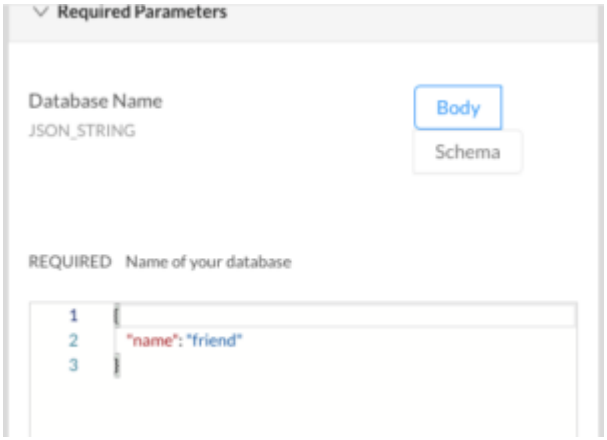


Select the Basic plan.

Create Database

Now that we can use the API, head back to the dashboard and select the **Create Database** endpoint. Our example will need a database to communicate with and we can create one here in the dashboard.

Change the name value in the request to 'friend'.



Hit the '**Test Endpoint**' button. A response object should appear on the right side of the dashboard telling you that the database was created successfully!

Now that we have a database, we need to make a friend model. Instead of setting up another function, for the sake of simplicity, we will create a basic model with the dashboard.

Create a Model

Just like we did for the database, we need to select the endpoint, fill in the parameters, and send the request.

Select the **Create Model** endpoint and add the appropriate parameters.

Select the **Test Endpoint** button. The response should be similar to:

Great! Now it's time to add the API code to our React component!

Grab Code Snippets

The process for retrieving the code snippets is as follows:

1. Select the endpoint
2. Change parameters and schema

3. Select the desired language and library
4. Copy the code

Let's do this for the **Add Entity (POST), Get All Entities (GET), Update Entity (PUT)**, and **Delete Entity (DELETE)** endpoints.

Remember, the database name is 'friend', the model name is 'friendModel', and the schema needs to match the friend model schema we set up.

For this example, I am using the (JavaScript) fetch library.

I've edited the snippets a little bit so they can receive JSON responses and send JSON requests. Furthermore, I made React specific changes to show how we are going to getting the variables from the component state for the request.

Here's what the snippets should be for each endpoint (make sure you are substituting your RapidAPI key):

Add Entity:

```
// creates entity
fetch("https://fairestdb.p.rapidapi.com/friend/friendModel",
{
  "method": "POST",
  "headers": {
    "x-rapidapi-host": "fairestdb.p.rapidapi.com",
    "x-rapidapi-key": "apikey",
    "content-type": "application/json",
    "accept": "application/json"
  },
  "body": JSON.stringify({
    name: this.state.name,
    notes: this.state.notes
  })
})
.then(response => response.json())
```

```
  .then(response => {
    console.log(response)
  })
  .catch(err => {
    console.log(err);
  });
```

Get All Entities:

```
// read all entities
fetch("https://fairestdb.p.rapidapi.com/friend/friendModel",
{
  "method": "GET",
  "headers": {
    "x-rapidapi-host": "fairestdb.p.rapidapi.com",
    "x-rapidapi-key": "apikey"
  }
})
.then(response => response.json())
.then(response => {
  this.setState({
    friends: response
  })
})
.catch(err => { console.log(err);
});
```

Update Entity:

```
// this will update entries with PUT
fetch("https://fairestdb.p.rapidapi.com/friend/friendModel",
{
  "method": "PUT",
  "headers": {
    "x-rapidapi-host": "fairestdb.p.rapidapi.com",
    "x-rapidapi-key": "apikey",
    "content-type": "application/json",
    "accept": "application/json"
  },
  "body": JSON.stringify({
    _id: this.state.id,
    name: this.state.name,
    notes: this.state.notes
  })
```

```
  })
  .then(response => response.json())
  .then(response => { console.log(response);
  })
  .catch(err => { console.log(err); });
```

Delete Entity:

```
// deletes entities
fetch(`https://fairestdb.p.rapidapi.com/friend/friendModel/_i
, {
  "method": "DELETE",
  "headers": {
    "x-rapidapi-host": "fairestdb.p.rapidapi.com",
    "x-rapidapi-key": "apikey"
  }
})
.then(response => response.json())
.then(response => {
  console.log(response);
})
.catch(err => {
  console.log(err);
});
```

IMPORTANT: Comment out the `componentDidMount` function after the code is added. This will stop the **Get All Entities** function from firing every time we make and edit and the app rerenders.

Next, add the snippets to the application inside the appropriate function. We want the app to retrieve all the entities when the component loads, therefore, the **Get All Entities** snippet goes in the `componentDidMount` function.

Add Snippets to Functions

For the snippets to work, I have added the necessary state variables that the functions need.

After adding the snippets to their corresponding functions, and modifying JSX, `src/app.js` should look like:

```
'use strict';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      friends: [],
      name: '',
      id: '',
      notes: ''
    };

    this.create = this.create.bind(this);
    this.update = this.update.bind(this);
    this.delete = this.delete.bind(this);
    this.handleChange = this.handleChange.bind(this);
  }

//   componentDidMount() {
//     // get all entities - GET
//     fetch("https://fairestdb.p.rapidapi.com/friend/friendM
//       "method": "GET",
//       "headers": {
//         "x-rapidapi-host": "fairestdb.p.rapidapi.com",
//         "x-rapidapi-key": API_KEY
//       }
//     })
//     .then(response => response.json())
//     .then(response => {
//       this.setState({
//         friends: response
//       })
//     })
//     .catch(err => { console.log(err);
//     });
//   }

  create(e) {
    // add entity - POST
    e.preventDefault();

    // creates entity
```

```
      fetch("https://fairestdb.p.rapidapi.com/friend/friendModel
        "method": "POST",
        "headers": {
          "x-rapidapi-host": "fairestdb.p.rapidapi.com",
          "x-rapidapi-key": API_KEY,
          "content-type": "application/json",
          "accept": "application/json"
        },
        "body": JSON.stringify({
          name: this.state.name,
          notes: this.state.notes
        })
      })
      .then(response => response.json())
      .then(response => {
        console.log(response)
      })
      .catch(err => {
        console.log(err);
      });
    }

    update(e) {
      // update entity - PUT
      e.preventDefault();

      // this will update entries with PUT
      fetch("https://fairestdb.p.rapidapi.com/friend/friendModel
          "method": "PUT",
          "headers": {
              "x-rapidapi-host": "fairestdb.p.rapidapi.com",
              "x-rapidapi-key": API_KEY,
              "content-type": "application/json",
              "accept": "application/json"
          },
          "body": JSON.stringify({
              _id: this.state.id,
              name: this.state.name,
              notes: this.state.notes
          })
          })
          .then(response => response.json())
          .then(response => { console.log(response);
          })
          .catch(err => { console.log(err); });
    }
```

```
89.
90.    delete(e) {
91.      // delete entity - DELETE
92.      e.preventDefault();
93.      // deletes entities
94.
fetch(`https://fairestdb.p.rapidapi.com/friend/friendModel/_id
{
95.        "method": "DELETE",
96.        "headers": {
97.          "x-rapidapi-host": "fairestdb.p.rapidapi.com",
98.          "x-rapidapi-key": API_KEY
99.        }
100.     })
101.     .then(response => response.json())
102.     .then(response => {
103.       console.log(response);
104.     })
105.     .catch(err => {
106.       console.log(err);
107.     });
108.   }
109.
110.   handleChange(changeObject) {
111.     this.setState(changeObject)
112.   }
113.
114.   render() {
115.     return (
116.         <div className="container">
117.           <div className="row justify-content-center">
118.             <div className="col-md-8">
119.               <h1 className="display-4 text-center">Make An AI
React</h1>
120.               <form className="d-flex flex-column">
121.                 <legend className="text-center">Add-Update-De
Friend</legend>
122.                 <label htmlFor="name">
123.                   Friend Name:
124.                   <input
125.                     name="name"
126.                     id="name"
127.                     type="text"
128.                     className="form-control"
129.                     value={this.state.name}
130.                     onChange={(e) => this.handleChange({ name
```

```
      })}
131.                     required
132.                     />
133.                 </label>
134.                 <label htmlFor="notes">
135.                   Friend notes:
136.                   <input
137.                     name="notes"
138.                     id="notes"
139.                     type="test"
140.                     className="form-control"
141.                     value={this.state.notes}
142.                     onChange={(e) => this.handleChange({ notes
      })}
143.                     required
144.                     />
145.                 </label>
146.                 <label htmlFor="id">
147.                   Friend ID:
148.                   <input
149.                     name="id"
150.                     id="id"
151.                     type="text"
152.                     className="form-control"
153.                     value={this.state.id}
154.                     onChange={(e) => this.handleChange({ id: e
155.                     />
156.                 </label>
157.                 <button className="btn btn-primary" type='butt
      => this.create(e)}>
158.                   Add
159.                 </button>
160.                 <button className="btn btn-info" type='button
      this.update(e)}>
161.                     Update
162.                 </button>
163.                 <button className="btn btn-danger" type='butto
      this.delete(e)}>
164.                     Delete
165.                 </button>
166.               </form>
167.             </div>
168.           </div>
169.         </div>
170.     );
171.   }
```

```
172.   }
173.
174.   let domContainer = document.querySelector('#App');
175.   ReactDOM.render(<App />, domContainer);
```

A few things to note:

- all the calls are from the same form, which can cause user experience issues, but for this example, it can work
- **componentDidMount** is commented out

Now, we can add, update, and delete friends from our application. Also, we can view the records that we are making using the RapidAPI dashboard to send requests and inspect the response.

How do we know that is work?

HTTP Status Codes

HTTP status codes provide information about a particular request. For example;

- 200 – the request has succeeded
- 201 – request successfully created something
- 400 – bad request (syntax, parameters)
- 401 – unauthorized
- 404 – page not found
- 500 – server error

Those are some of the most common. In our app, when we add check the status of requests in the developer tools.

In the **Network** tab, you can find the request that was just sent from the browser. Let's try it out.

In the app add a friend to the database.

Open up the developer tools (I am using Google Chrome) and go to the **Network** tab.

Notice the request's name was 'friendModel' and the status was 200! If a request fails, the browser may highlight the request red so it is easier to find.

We are golden in terms of sending HTTP requests with React. However, what if we want to view this data in our application? It would be much easier to make edits and see all of our friends!

How to Display API data in React componentDidMount

Previously, we added code to `componentDidMount` that retrieves the records in our database. However, we did not have a way to display that data on the page, so we commented it out.

Let's create a new component that receives the response data as a **prop**, and displays that data when the page loads.

Above the App component in `app.js` add the component:

```
'use strict'


class Friends extends React.Component {
    render() {
        return (
            <table>
                <thead>
                    <tr>
                        <th>ID</th>
                        <th>Name</th>
                        <th>Since</th>
                    </tr>
                </thead>
                <tbody>
                    {this.props.friends &&
this.props.friends.map(friend => {
                        return <tr>
                            <td>{friend._id}</td>
                            <td>{friend.name}</td>
                            <td>{friend.since}</td>
                        </tr>
                    })}
                </tbody>
            </table>
        );
    }
}

// App component
```

The above component is similar to App, however, it waits for `this.props.friends` to be defined before rendering the table rows. This is necessary because the parent component, App, needs to fetch the data. Therefore, the data will not be available when the component first mounts.

Next, add this component as a child of App and pass the property `friends` with the value `this.state.friends`.

Inside the JSX code of the App component, and underneath the closing `</form>` tag, add the following line.

```
<Friends friends={this.state.friends} />
```

Next, uncomment the `componentDidMount` function.

Finally, the friends that we have in the database will appear as rows in the table when the page loads!

This will make it easier to add, remove, and edit our friend list.

Further Considerations

User Experience

There are many ways that the form could be improved, for example, it could:

- display success and error messages
- clear the input after submitting
- only allow the user to submit fields that are used by the specific API call

However, this is one of the most basic ways to make a React component and I chose this method so we could focus on the important details of building components and making API calls. Not only that, I wanted you to understand React at a basic level before using an advanced project builder.

That being said, for a small project this could be fixed up to look—and behavior—nicer with little effort.

Security

Despite instructing you to put your API-key in the front-end code for this example that is a bad idea, because it can be found if someone was looking for it. Therefore, whether you build a React app this way, or whether you use Create-React-App, if the React code is in the front-end it should not have an API-key.

This might leave you wondering, so what do I do? One example is to use a framework like Next.js that uses server-side rendering or a static site render like GatsbyJS. Unfortunately, even those solutions do not cover all scenarios so make sure you read up on what you are trying to do before you put your API key in a compromising position.

In the next section, I cover two examples of solutions for securing your app for API calls.

React API Call Examples

> *Build a Dictionary App with the WordsAPI (JavaScript)*

The above tutorial doesn't use React but utilizes Netlify to make secure API requests without having to set up a backend server. Netlify offers serverless function capabilities that are not shipped with the frontend code, so API secrets can be added as environment variables later.

Reviewing the above tutorial would introduce you to an easy solution to securing third-party API requests.

> *How to Create an App with Kairos Facial Recognition API in Javascript*

If you are looking for more of a challenge the tutorial above is the answer. The frontend of the app is built with Create-React-App (CRA) which has been a popular choice for many when building larger React applications.

In addition to CRA, the example application sets up a backend NodeJS server that makes secure HTTP requests to a third-party API. This is another very common pattern in creating React applications and is why stacks like the M.E.R.N become popular (MongoDB, Express, React, Node).

Here's another example using OpenWeatherMap:

> *How To Create a Weather App with React (OpenWeatherMap API)*

And with Google Maps:

> *How to use the Google Maps API in React*

Conclusion

It's difficult to build a React application without adding features that involve sending HTTP requests. There are many APIs that make our lives easier and it would be foolish not to utilize them! This introduction to React and API calls was brief so I hope you can sharper your skills with one of the tutorials I suggested above.

If you have any questions please leave a comment!

Related

- How to display API data with Axios using React
- How to fetch data from API using GraphQL and React
- 10 React Starter Project Ideas to Get You Coding
- How to Add an API to RapidAPI
- How to use RapidAPI

FAQ

How to display API data using Axios with React?

Set-up the app, add Axios API call with API Key, and transform Axios Response data. Read more here: https://rapidapi.com/blog/axios-react-api-tutorial/

What is right way to do API call in React JS?

As best place and practice for external API calls is React Lifecycle method componentDidMount(), where after the execution of the API call you should update the local state to be triggered new render() method call, then the changes in the updated local state will be applied on the component view.

How to fetch data from a GraphQL API in React?

Set up the application, fetch GraphQL data and display the response data. See more here: https://rapidapi.com/blog/graphql-react-fetch-data/

5 / 5 (2 votes)

« API vs Microservices [What's the Difference?]

Build an IP Scanner Tool in Python with geoPlugin IP Geolocation API »

Related Blog Posts

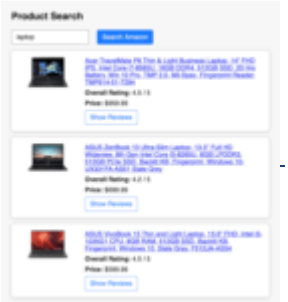
Flutter vs. React Native: Which Do Developers' Use More

[How to build a REST API with Node.js – Part 1]

[React API Authentication & Authorization]

[How to use WordPress with React (WordPress React API Tutorial)]

[How to use an E-mail API with JavaScript]



[How to use Amazon Product/Review APIs in JavaScript]

Filed Under: JavaScript API Tutorials, React API Tutorials, REST API Tutorials Tagged With: how to, how to use an api, javascript, react, react.js, reactjs

Jarrett Retz

Jarrett is a Web and Automation Application Developer based in Spokane, WA. He is well versed in NodeJS, React, Django, Python, MongoDB, third-party APIs (i.e Sendgrid, Stripe), PostgreSQL, HTML5, CSS3, and cloud-computing resources.

Learn more about Jarrett on his website.

Search this website

Search

Comments

min says
SEPTEMBER 28, 2020 AT 9:13 AM

Thanks for this tutorial!

I ran into an issue when trying to display Friends:
Uncaught (in promise) SyntaxError: Unexpected token { in JSON at position 435

Using 'TestEndpoint' for "GET Get All Entities" returns status 200, but error in the body:

"ERROR":{1 item
"message":"src property must be a valid json object"
}

Any tips for how to resolve?

Reply

Jim says
JANUARY 8, 2021 AT 3:28 PM

Hi Min,
I got this exact same error.
Were you ever able to resolve it?

Reply

Team OrbitSolve says
JANUARY 11, 2021 AT 9:35 PM

This error is resolved now. Please try again

Reply

OrbitSolve says
JANUARY 11, 2021 AT 9:36 PM

This error is resolved now. Please try again.

Reply

Leave a Reply

Comment

Name *

Email *

Post Comment

APIs mentioned in this article

Connect to the FaiRESTdb API

Learn

How to use an API

API Glossary

API Testing

API Management

For Developers

For API Providers

About

Team

Jobs

Contact Us

API Directory

Press Room

Privacy Policy

Terms of Use