Hyperiondev

TASK

React - Fetching Data

Visit our website

Introduction

WELCOME TO THE REACT - FETCHING DATA TASK!

You have already learnt to use the Fetch API to get data from an external source like a web API. In this task, you will learn to create a React app that interacts with a third-party API to fetch data.

WHERE DO WE GET DYNAMIC DATA FROM?

Dynamic applications rely on dynamic data. Where and how this data is stored will depend largely on the decisions you as a web developer makes with your development team for each project.

Usually, if your web app is dependent on data that you need to gather, store and update regularly, you will decide to use some or other type of database. This database could be hosted on your own web servers or on a PaaS cloud. You and your development team will be responsible for choosing and designing your database solution and for writing the code that interacts with this database. You will learn to do this in future tasks.

If your web app isn't heavily reliant on data that needs to be stored and updated by you, then you may opt for a simpler solution (such as storing json files on your webserver) for keeping small amounts of uncomplicated data that isn't often changed.

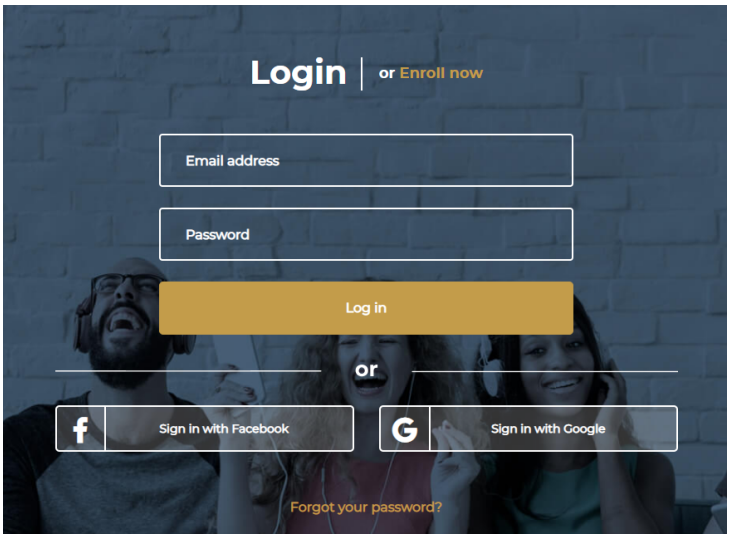
Whenever you need to expose data that is stored and managed by you or your development team, you need to be able to write the back-end code that will expose this data to the front-end of your app. You usually do this by creating a custom Restful API. You will learn to do this using Express soon.

However, it is also becoming more common for web applications to access data that is stored, updated and maintained by a third party. If this is the case, this third party will usually provide a Restful API that you can use to retrieve this data. Before learning to do this, you first need to understand what a Restful API is.

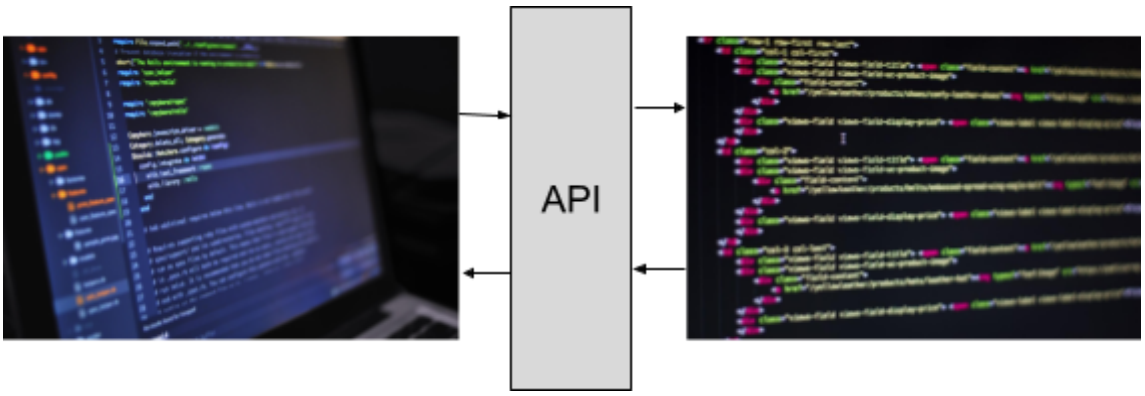
WHAT IS AN API?

An application programming interface (API) is an interface that sits between two applications or modules of code.

Let us illustrate this using an example: Imagine you are asked to create a web page that implements the login functionality provided by Facebook. As illustrated below, this should work similarly to the page you encountered when you first registered with HyperionDev.



To get the Facebook login functionality to work, you don't want to write the code needed to login from scratch, you just want to use the code Facebook has already written to do this. You also don't necessarily want to know exactly how Facebook wrote their code or line for line how it works. You just want to know what you need to do in order to get their code to work with yours. This is where the Facebook web API comes in. The web API serves as an interface between your code and Facebook's code. It hides/abstracts the complexity of Facebook's code and just gives you the information you need to be able to interface/interact with it.



When coding using JavaScript, you will encounter many APIs. The client-side JavaScript APIs that you will encounter can be broadly grouped into two categories:

- Third-party APIs: These are APIs that allow you to use code written by third parties like Facebook, Google, or Twitter. Facebook's Graph API is an example of a third party web API.
- Browser APIs: These APIs abstract functionality provided by your browser. For example, you could use the Geolocation API which allows you to interface with code that works on a lower level on your browser and machine to get information about where your computer is. The API allows you to use this code without knowing exactly how the code works.

We now know what an API is, but what is a Restful API? To understand that, you first need to know what a web service is.

WEB SERVICES

Web services are basically modules of code that can be accessed over the web. IBM defines web services as "self contained, self-describing, modular applications that can be published, located and invoked across the web". The purpose of a web service is to provide some sort of functionality to the requester over the web. There are multiple basic ways that web services can work: E.g. SOAP is an old standard that is now largely replaced by RESTful web services.

For sake of interest, there's a new web service paradigm called GraphQL, which is slowly replacing REST. Just goes to show how no technology will be relevant forever. However, as it stands REST is currently by far the most popular standard.

WHAT IS A RESTFUL API?

Restful web services allow us to access Restful APIs. RESTful APIs are based on the REST architecture. This is a client/server architecture that uses a stateless communication protocol like HTTP.

According to **Oracle**, RESTful web services are based on the following principles:

- RESTful web services expose resources using URIs.

- Resources are manipulated using PUT, GET, POST, and DELETE operations.
    - PUT creates a new resource
    - DELETE deletes a resource.
    - GET retrieves the current state of a resource.

- POST transfers a new state onto a resource.

- Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON and others.

- Interaction with resources is stateless. State information is exchanged using techniques like URI rewriting, cookies, hidden form fields and embedding state information in response messages.

FETCHING DATA WITH REACT

From the above, it should be clear that we can use HTTP Get requests to access data exposed by Restful APIs. There are several ways of fetching data. One way is using the Fetch API.

As the name suggests, the Fetch API provides an interface for asynchronously fetching resources. The `fetch()` method takes one mandatory argument, the path to the resource you want to fetch. It returns a Promise that resolves to the Response to that request, whether it is successful or not.

Take note:

Remember that a promise basically allows the interpreter to carry on with other tasks while a function is executed without waiting for that function to finish. This is because the promise, 'promises' to let you know the outcome of the function once it is finished.

The basic structure of a promise is:

```
doSomething().then(successCallback, failureCallback);
```

A promise uses callbacks to execute different functions based on whether the desired action succeeded or failed.

With React, asynchronous calls needed to load data from a remote endpoint are usually made in the **componentDidMount** lifecycle method. The componentDidMount method is invoked immediately after a component is mounted.

Consider the example provided in the **React documentation** below. In the code example we make a fetch() call to a fictitious API

(`fetch("https://api.example.com/items")`) that will supposedly return the following json:

```
{ "items": [ { "id": 1, "name": "Apples",  "price": "$2" },
{ "id": 2, "name": "Peaches", "price": "$5" } ] }
```

Consider the `fetch()` call in the componentDidMount method. Notice that the promise returned by the `fetch()` method call is consumed by the first `.then` statement. Here the response received from fetching the data from the API is passed as an argument to the `res.json()` method. This method returns a promise that resolves with the result of parsing the res text as JSON. This result is used to set the state of the component we are creating. `this.state.items` will therefore now contain `[ { "id": 1, "name": "Apples",  "price": "$2" }, { "id": 2, "name": "Peaches", "price": "$5" }]`.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      error: null,
      isLoaded: false,
      items: []};
  }

  componentDidMount() {
    fetch("https://api.example.com/items")
      .then(res => res.json())
      .then(
        (result) => {
          this.setState({
            isLoaded: true,
            items: result.items });
        },
        // Note: it's important to handle errors here instead of a catch()
block so that
        // we don't swallow exceptions from actual bugs in components.
        (error) => {
          this.setState({
            isLoaded: true,
            error
          });
        });
  }

  render() {
```

```
    const { error, isLoaded, items } = this.state;
    if (error) { return <div>Error: {error.message}</div>;
    } else if (!isLoaded) { return <div>Loading...</div>;
    } else { return (
        <ul>
          {items.map(item => (
            <li key={item.name}>
              {item.name} {item.price}
            </li>
          ))}
        </ul>
      );}}
  }
```

API KEYS

Often when you want to make use of a third-party API, you will be granted an API key. An API key is a string of seemingly jumbled letters and numbers. It's actually a password that gives your app access to the said API.

In the compulsory task below, you are going to obtain an API key to use a third-party API. For this task, you can hardcode the key into your program where you make the `fetch()` call.

Here is an example of what the URL for the Open Weather API in a fetch() call looks like when hardcoded:

```
`http://api.openweathermap.org/data/2.5/weather?q=${city},${country}&appid=this_is_an_api_key`
```

Note that this isn't secure. If you publish your code on GitHub, anyone will be able to see the key and use the API as you. To help alleviate this, API keys and calls to third-party APIs are usually handled by the back-end of your web app. You will learn to do this soon. For now, don't worry about storing the API key in your React app.

It is good practice if you want to hide the key from your public code, to do the following:

1. Add a file called '.env' on your root folder with key/pairs entries.
   E.g.: `WEATHER_API_KEY=<yourKey>`

2. Now you access the key stored in '.env' from anywhere in your React code by using the process.env variable.

   Below is an example of how the same Open Weather API URL mentioned before would look if you use an environment variable instead of hardcoding the API key:

```
`http://api.openweathermap.org/data/2.5/weather?q=${city},${country}&ap
pid=${process.env.WEATHER_API_KEY}`
```

3. Add .env to your .gitignore file so that the .env file that stores your API key isn't pushed to GitHub.
4. Backup the .env file somewhere private. From here on out it's required to run your app and won't be available on GitHub.

Compulsory Task 1

Follow these steps:

- Create a React app that will display the definition and example usage of any word typed by the user.
- Use the Merriam Webster Dictionary API: **https://www.dictionaryapi.com/**
- Consult the **documentation** of the API in order to get an understanding of how to use the API. You will need to, among other things, obtain an API key.

Optional Bonus Task

Follow these steps:

- If you have hardcoded the API key into your fetch() call URL, modify your project to use an .env file and environment variables instead.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.

REFERENCE

React.js. (2020). Getting Started – React. Retrieved 6 August 2020, from **https://reactjs.org/docs/getting-started.html**