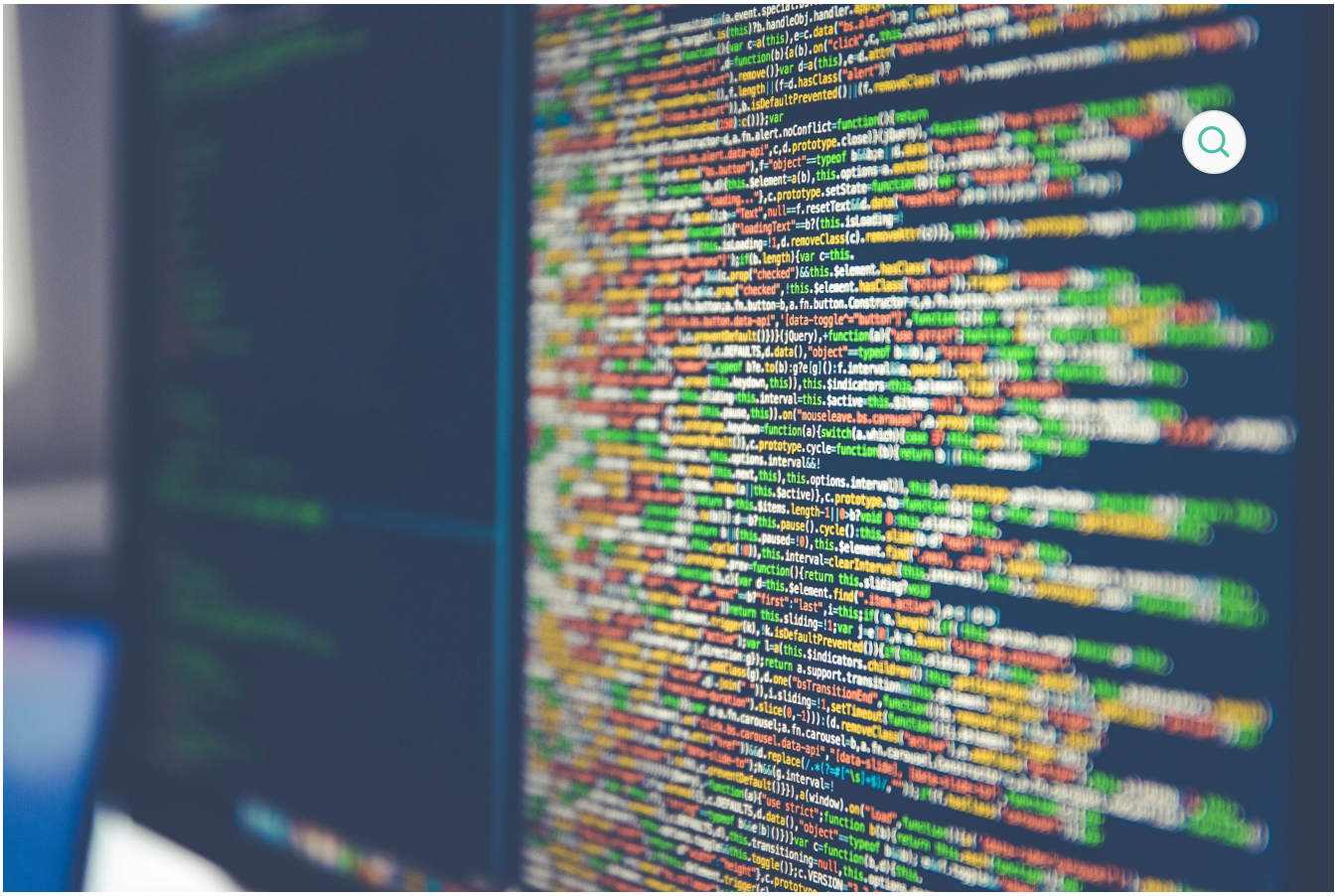
Build a Realtime PWA with React

Create an app to display the current and past price information of BTC, LTC, and ETH

Yomi

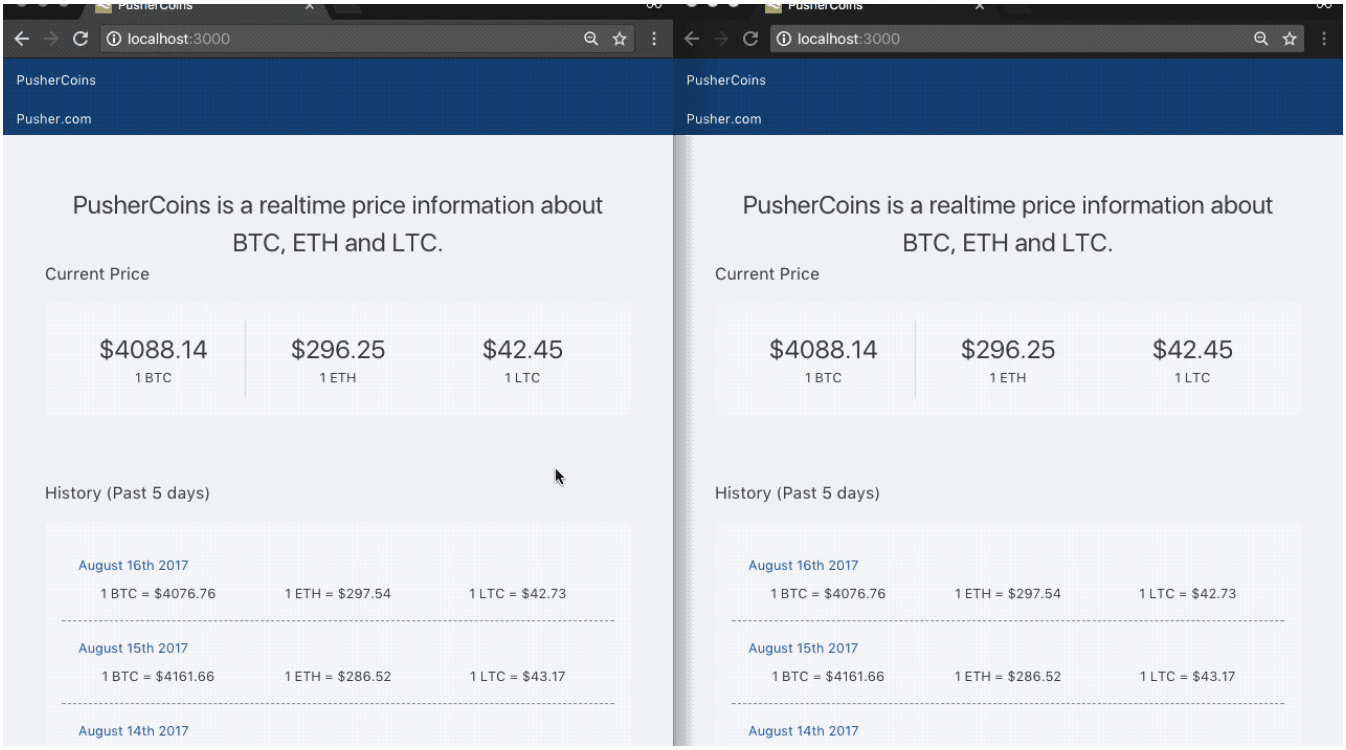
Apr 8, 2018 · 15 min read



Progressive Web Apps (PWAs) are experiences that combine the best of the web and apps. They use service workers, HTTPS, a manifest file and an app shell architecture to deliver native app experiences to web applications.

In this tutorial, we'll build a PWA called PusherCoins. PusherCoins displays the current and past price information about Bitcoin (BTC), Litecoin (LTC), and Ethereum (ETH) using data from cryptocompare.com. A demo can be seen below. The current Bitcoin,

Ethereum, and Litecoin prices are updated every 10 seconds, changed in real-time, and seen across other connected clients connected via Pusher.



. . .

Building a PWA with Create React App

We're going to be building a realtime PWA with the help of create-react-app.

It's common for developers who are just getting into React to have a difficult time getting set up and configuring their apps. `create-react-app` eliminates all of that by allowing developers to build React apps with little or no build configuration. All you have to do to get a working React app is to install the npm module and run a single command.

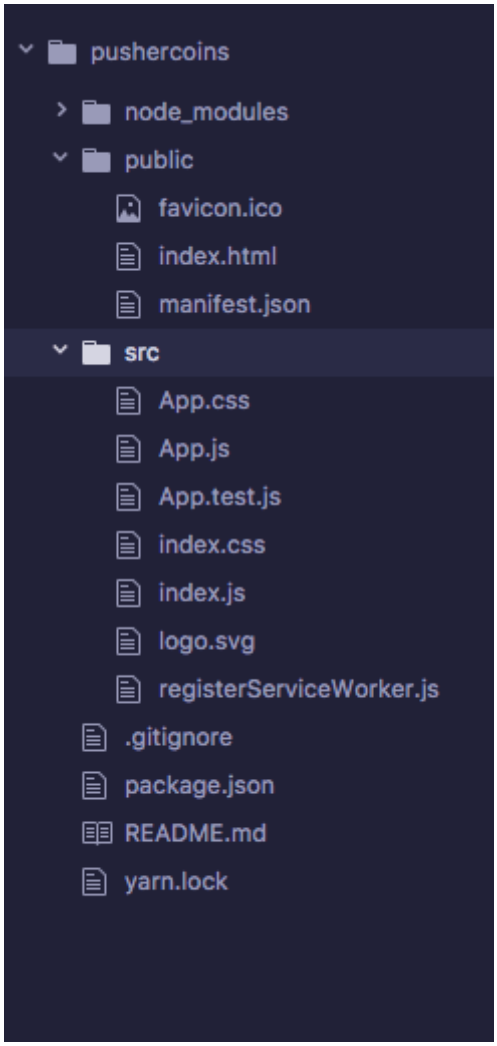
More importantly, the production build of `create-react-app` is a fully functional Progressive Web Application. This is done with the help of the `sw-precache-webpack-plugin`, which is integrated into the production configuration.

Let's get started with building the React app. Install the `create-react-app` tool with the following command:

```
npm install -g create-react-app
```

Once the installation process has been completed, you can now create a new React app by using the command `create-react-app pushercoins`.

This generates a new folder with all the files required to run the React app and a service worker file. A manifest file is also created inside the `public` folder.



The `manifest.json` file in the `public` folder is a simple JSON file that gives you, the ability to control how your app appears to the user and define its appearance at launch.

```
{
  "short_name": "PusherCoins",
  "name": "PusherCoins",
  "icons": [
    {
      "src": "favicon.ico",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "android-chrome-512x512.png",
```

```
12            "sizes": "512x512",
13            "type": "image/png"
14          }
15        ],
16        "start_url": "./index.html",
17        "display": "standalone",
18        "theme_color": "#000000",
19        "background_color": "#ffffff"
20      }
```

manifest.json hosted with ♡ by GitHub view raw

We notify the app of the `manifest.json` file by linking to it in line 12 of the `index.html` file.

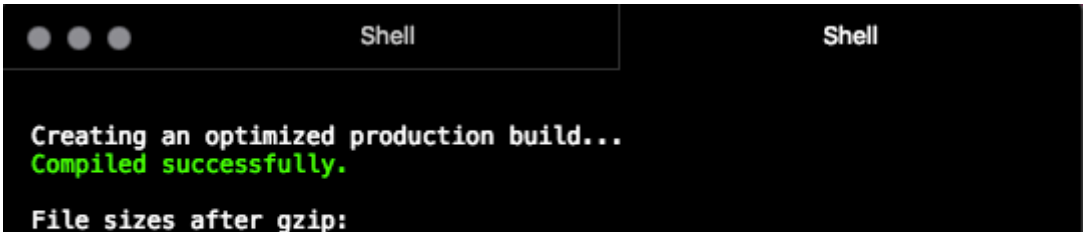
```
<link rel="manifest" href="%PUBLIC_URL%/manifest.json">
```

Next up, let's go through the `registerServiceWorker.js` file and see how the service worker file works. The service worker file can be seen in the `src` folder on GitHub.

The service worker code basically registers a service worker for the React app. We first check if the app is being served from localhost via the `isLocalhost` const value that will either return a truthy or falsy value. The `register()` function helps to register the service worker to the React app only if its in a production mode and if the browser supports Service workers. The `unregister()` function helps to unregister the service worker.

Let's find out if the service worker really works. To do that we'll need to prepare the React app for production as the service worker code only works in production mode. The `npm run build` command helps with that.

This command builds the app for production to the `build` folder and correctly bundles React in production mode and optimizes the build for the best performance. It also registers the service worker. Run the command and the output from the terminal should look like something below:

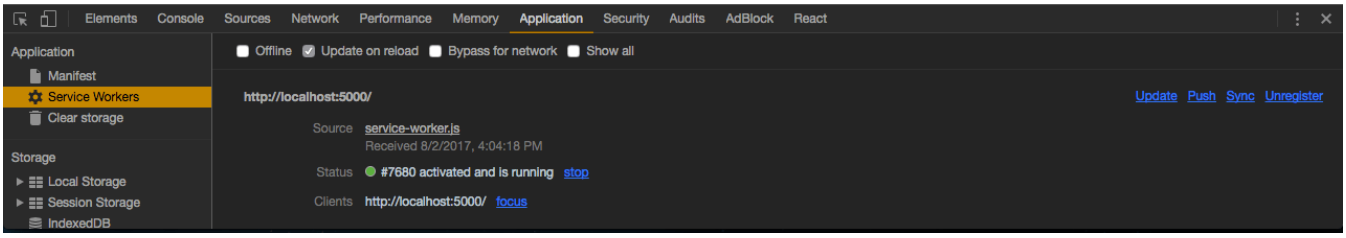


```
  48.25 KB  build/static/js/main.120eaed3.js
  288 B     build/static/css/main.cacbacc7.css

The project was built assuming it is hosted at the server root.
To override this, specify the homepage in your package.json.
For example, add this to build it for GitHub Pages:

  "homepage" : "http://myname.github.io/myapp",

The build folder is ready to be deployed.
You may serve it with a static server:

  yarn global add serve
  serve -s build
```

We get to see the size of the files in our React app and most importantly how to run the app with the aid of a static server. We are going to use the serve npm package to, wait for it, serve(😀) the React app. Therefore, use the following commands to install serve on your computer and also set up a static server for the app:

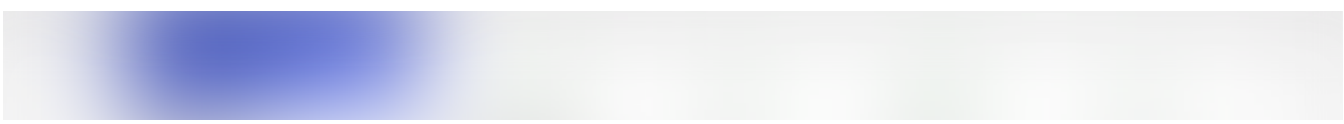
```
npm i serve -g

serve -s build
```

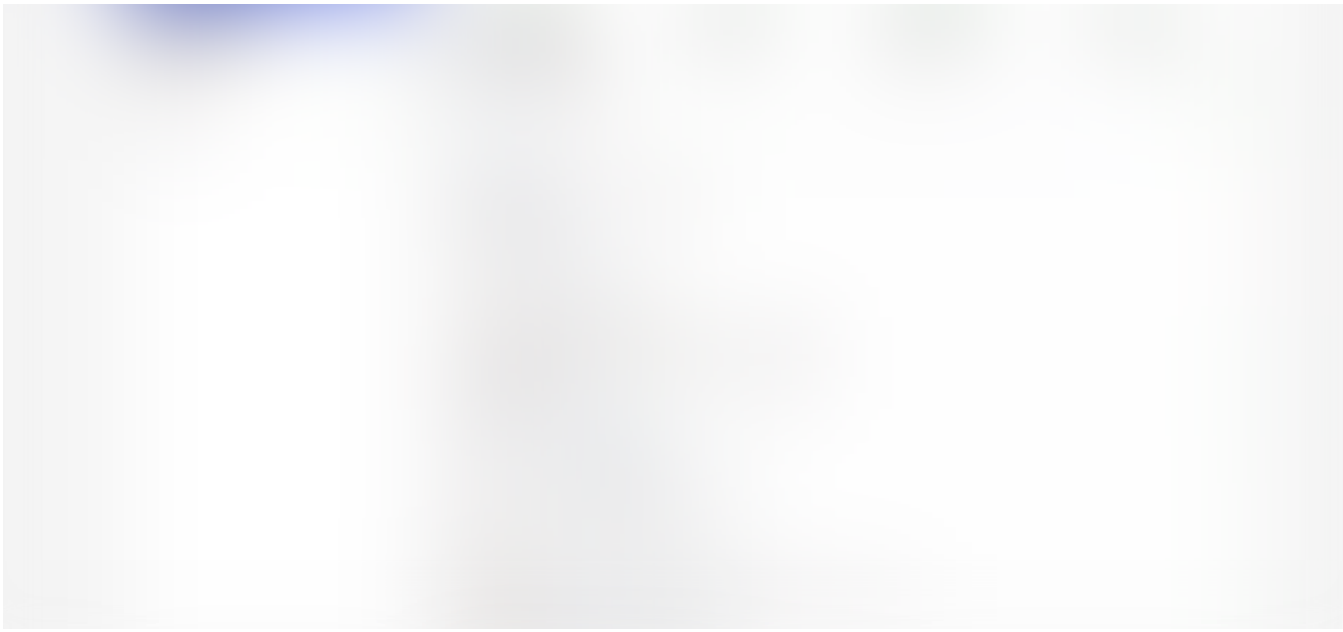
Your application should be up and running at http://localhost:5000. So how do we check if a site is a PWA? We can do that by checking the service worker section in the 'Application' tab in the Developer tools.



We could also check by using the Lighthouse tool. Lighthouse is an open-source, automated tool for improving the quality of web pages. It can perform audits on performance, accessibility and Progressive Web A6pps. Lighthouse is currently available as an extension on Google Chrome only and as an npm package.

I used the Lighthouse extension to generate a report for the newly created React app in production and got the following result.





The React app got a score of 91 out of 100 for the PWA section, which isn't that bad. All audits were passed but one related to HTTPS, which cannot be implemented right now because the app is still on a local environment.

Now that we know how to check if an app is a PWA, let's go ahead to build the actual app. As we'll be building this PWA with React, it's very important that we think in terms of React components.

Therefore, the React app would be divided into three components.

1. `History.js` houses all the code needed to show the past prices of BTC, ETH, and LTC.
2. `Today.js` houses all the code needed to show the current price of BTC, ETH, and LTC.
3. `App.js` houses both `History.js` and `Today.js`

Alright, let's continue with building the app. We'll need to create two folders inside the `src` folder, `Today` and `History`. In the newly created folders, create the files `Today.js`, `Today.css` and `History.js`, `History.css` respectively. Your project directory should look like the one below.

Before we get started on the `Today` and `History` components, let's build out the app shell.

> An app shell is the minimal HTML, CSS and JavaScript required to power the user interface and when cached offline can ensure instant, reliably good performance to users on repeat visits. You can read more about app shells here.

Open up the `App.js` file and replace with the following code:

```
// Import React and Component
    import React, { Component } from 'react';
    // Import CSS from App.css
    import './App.css';
    // Import the Today component to be used below
    import Today from './Today/Today'
    // Import the History component to be used below
    import History from './History/History'

    class App extends Component {
      render() {
        return (
          <div className="">
              <div className="topheader">
                  <header className="container">
                      <nav className="navbar">
                          <div className="navbar-brand">
                              <span className="navbar-item">PusherCoins</span>
                          </div>
                          <div className="navbar-end">
                              <a className="navbar-item" href="https://pusher.com" target="_bla
                          </div>
                      </nav>
                  </header>
              </div>
              <section className="results--section">
                  <div className="container">
                      <h1>PusherCoins is a realtime price information about<br></br> BTC, ETH a
                  </div>
                  <div className="results--section__inner">
                      <Today />
                      <History />
                  </div>
              </section>
          </div>
```

```
        );
      }
    }

    export default App;
```

app.js hosted with ♡ by GitHub view raw

The `App.css` file should be replaced with the following:

```
.topheader {
  background-color: #174c80;
}
.navbar {
  background-color: #174c80;
}
.navbar-item {
  color: #fff;
}
.results--section {
  padding: 20px 0px;
  margin-top: 40px;
}
h1 {
  text-align: center;
  font-size: 30px;
}
```

app.css hosted with ♡ by GitHub view raw

We'll also be using the Bulma CSS framework, so add the line of code below to your `index.html` in `public` folder:

```
<link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.4.3/css/bulma.min.css">
```

. . .

Create the React Components

Next up, open up the `Today.js` file as we'll soon be writing the code for that component. So what does this component do?

It's responsible for getting the current prices of Bitcoin, Ethereum, and Litecoin from the Cryptocompare API and displaying it on the frontend. Let's write the code.

The first thing we do is import React and its Component module using ES6 `import`, we also import axios. Axios is used to make API requests to the Cryptocompare API and can be installed by running `npm install axios` in your terminal

```
import React, { Component } from 'react';
    import './Today.css'
    import axios from 'axios'
```

The next thing to do is create an ES6 class named `Today` that extends the component module from `react`.

```
class Today extends Component {
        // Adds a class constructor that assigns the initial state values:
        constructor () {
            super();
            this.state = {
                btcprice: '',
                ltcprice: '',
                ethprice: ''
            };
        }
        // This is called when an instance of a component is being created and inserted into th
        componentWillMount () {
            axios.get('https://min-api.cryptocompare.com/data/pricemulti?fsyms=BTC,ETH,LTC&tsym
                .then(response => {
                    // We set the latest prices in the state to the prices gotten from Cryptocu
                    this.setState({ btcprice: response.data.BTC.USD });
                    this.setState({ ethprice: response.data.ETH.USD });
                    this.setState({ ltcprice: response.data.LTC.USD });
                })
                // Catch any error here
                .catch(error => {
                    console.log(error)
                })
        }
        // The render method contains the JSX code which will be compiled to HTML.
        render() {
            return (
                <div className="today--section container">
                    <h2>Current Price</h2>
```

```
                    <div className="columns today--section__box">
                        <div className="column btc--section">
                            <h5>${this.state.btcprice}</h5>
                            <p>1 BTC</p>
                        </div>
                        <div className="column eth--section">
                            <h5>${this.state.ethprice}</h5>
                            <p>1 ETH</p>
                        </div>
                        <div className="column ltc--section">
                            <h5>${this.state.ltcprice}</h5>
                            <p>1 LTC</p>
                        </div>
                    </div>
                </div>
            )
        }
    }

    export default Today;
```

today.js hosted with ♡ by GitHub view raw

In the code block above, we imported the `react` and `component` class from React. We also imported `axios` which will be used for API requests. In the `componentWillMount` function, we send an API request to get the current cryptocurrency rate. The response from the API is what will be used to set the value of the state.

Let's not forget the CSS for the component. Open up `Today.css` and type in the following CSS code:

```
.today--section {
        margin-bottom: 40px;
        padding: 0 50px;
    }
    .today--section h2 {
        font-size: 20px;
    }
    .today--section__box {
        background-color: white;
        padding: 20px;
        margin: 20px 0;
        border-radius: 4px;
        box-shadow: 0 1px 2px 0 rgba(0, 0, 0, 0.05);
    }
    .btc--section {
```

```
        text-align: center;
        border-right: 1px solid #DAE1E9;
    }
    .btc--section h5 {
        font-size: 30px;
    }
    .eth--section {
        text-align: center;
        border-right: 1px solid #DAE1E9;
    }
    .eth--section h5 {
        font-size: 30px;
    }
    .ltc--section {
        text-align: center;
    }
    .ltc--section h5 {
        font-size: 30px;
    }
    @media (max-width: 480px) {
        .eth--section {
            border-right: none;
        }
        .btc--section {
            border-right: none;
        }
        .today--section {
            margin-top:  50px;
        }
    }
```

today.css hosted with ♡ by GitHub view raw

The next step is to write the code for `History.js`. This component is responsible for showing us the prices of BTC, ETH, and LTC from the past five days. We'll be using the `axios` package as well as the `moment` package for formatting dates. Moment.js can be installed by running `npm install moment` in your terminal. Open up the `History.js` file, the first thing we do is import React and its Component module using ES6 `import`, we also import axios and `Moment.js`.

```
import React, { Component } from 'react';
    import './History.css'
    import axios from 'axios'
    import moment from 'moment'
```

Like we did in the `Today.js` component, we'll create an ES6 class named `History` that extends the component module from `react` and also create some functions which will be bound with `this`.

```
class History extends Component {
      constructor () {
        super();
        this.state = {
            todayprice: {},
            yesterdayprice: {},
            twodaysprice: {},
            threedaysprice: {},
            fourdaysprice: {}
        }
        this.getBTCPrices = this.getBTCPrices.bind(this);
        this.getETHPrices = this.getETHPrices.bind(this);
        this.getLTCPrices = this.getLTCPrices.bind(this);
      }
      // This function gets the ETH price for a specific timestamp/date. The date is passed in
      getETHPrices (date) {
          return axios.get('https://min-api.cryptocompare.com/data/pricehistorical?fsym=ETH&tsy
      }
      // This function gets the BTC price for a specific timestamp/date. The date is passed in
      getBTCPrices (date) {
          return axios.get('https://min-api.cryptocompare.com/data/pricehistorical?fsym=BTC&tsy
      }
      // This function gets the LTC price for a specific timestamp/date. The date is passed in
      getLTCPrices (date) {
          return axios.get('https://min-api.cryptocompare.com/data/pricehistorical?fsym=LTC&tsy
      }
    }
```

history.js hosted with ♡ by GitHub view raw

As seen in the code block above, we have defined state values that will hold the price information about BTC, ETH, and LTC for the past five days. We also created functions that return API requests to Cryptocompare. Now, let's write the code that utilizes the functions above, stores the various prices in the state, and renders them.

It's important to note that Cryptocompare currently does not have an API endpoint that allows you to get a date range of price information. You'd have to get the timestamp of the past five days and then use them individually to get the required data you want. A workaround will be to use moment.js to get the timestamp of the particular day you

want using the `.subtract` method and `.unix` method . So for example, to get a timestamp of two days ago, you'd do something like:

```
moment().subtract(2, 'days').unix();
```

Okay, so let's continue with the rest of the code and write out the functions that get the values for the past 5 days.

```
// This function gets the prices for the current date.
    getTodayPrice () {
        // Get today's date in timestamp
        let t = moment().unix()
        // axios.all is used to make concurrent API requests. These requests were the functions
        axios.all([this.getETHPrices(t), this.getBTCPrices(t), this.getLTCPrices(t)])
            .then(axios.spread((eth, btc, ltc) => {
                let f = {
                    date: moment.unix(t).format("MMMM Do YYYY"),
                    eth: eth.data.ETH.USD,
                    btc: btc.data.BTC.USD,
                    ltc: ltc.data.LTC.USD
                }
                // Set the state of todayprice to the content of the object f
                this.setState({ todayprice: f });
            }));
    }
    // This function gets the prices for the yesterday.
    getYesterdayPrice () {
        // Get yesterday's date in timestamp
        let t = moment().subtract(1, 'days').unix();
        // axios.all is used to make concurrent API requests. These requests were the functions
        axios.all([this.getETHPrices(t), this.getBTCPrices(t), this.getLTCPrices(t)])
            .then(axios.spread((eth, btc, ltc) => {
                let f = {
                    date: moment.unix(t).format("MMMM Do YYYY"),
                    eth: eth.data.ETH.USD,
                    btc: btc.data.BTC.USD,
                    ltc: ltc.data.LTC.USD
                }
                // Set the state of yesterdayprice to the content of the object f
                this.setState({ yesterdayprice: f });
            }));
    }
    // This function gets the prices for the two days ago.
    getTwoDaysPrice () {
```

```
        // Get the date for two days ago in timestamp
        let t = moment().subtract(2, 'days').unix();
        // axios.all is used to make concurrent API requests. These requests were the functions
        axios.all([this.getETHPrices(t), this.getBTCPrices(t), this.getLTCPrices(t)])
            .then(axios.spread((eth, btc, ltc) => {
                let f = {
                    date: moment.unix(t).format("MMMM Do YYYY"),
                    eth: eth.data.ETH.USD,
                    btc: btc.data.BTC.USD,
                    ltc: ltc.data.LTC.USD
                }
                // Set the state of twodaysprice to the content of the object f
                this.setState({ twodaysprice: f });
            }));
    }
    // This function gets the prices for the three days ago.
    getThreeDaysPrice () {
        // Get the date for three days ago in timestamp
        let t = moment().subtract(3, 'days').unix();
        // axios.all is used to make concurrent API requests. These requests were the functions
        axios.all([this.getETHPrices(t), this.getBTCPrices(t), this.getLTCPrices(t)])
            .then(axios.spread((eth, btc, ltc) => {
                let f = {
                    date: moment.unix(t).format("MMMM Do YYYY"),
                    eth: eth.data.ETH.USD,
                    btc: btc.data.BTC.USD,
                    ltc: ltc.data.LTC.USD
                }
                // Set the state of threedaysprice to the content of the object f
                this.setState({ threedaysprice: f });
            }));
    }
    // This function gets the prices for the four days ago.
    getFourDaysPrice () {
        // Get the date for four days ago in timestamp
        let t = moment().subtract(4, 'days').unix();
        // axios.all is used to make concurrent API requests. These requests were the functions
        axios.all([this.getETHPrices(t), this.getBTCPrices(t), this.getLTCPrices(t)])
            .then(axios.spread((eth, btc, ltc) => {
                let f = {
                    date: moment.unix(t).format("MMMM Do YYYY"),
                    eth: eth.data.ETH.USD,
                    btc: btc.data.BTC.USD,
                    ltc: ltc.data.LTC.USD
                }
                // Set the state of fourdaysprice to the content of the object f
                this.setState({ fourdaysprice: f });
```

```
            }));
        }
        // This is called when an instance of a component is being created and inserted into the DC
        componentWillMount () {
            this.getTodayPrice();
            this.getYesterdayPrice();
            this.getTwoDaysPrice();
            this.getThreeDaysPrice();
            this.getFourDaysPrice();
        }
```

So we have five functions above, they basically just use `moment.js` to get the date required and then pass that date into the functions we first created above, to get the price information from Cryptocompare. We use `axios.all` and `axios.spread`, which is a way of dealing with concurrent requests with callbacks. The functions will be run in the `componentWillMount` function.

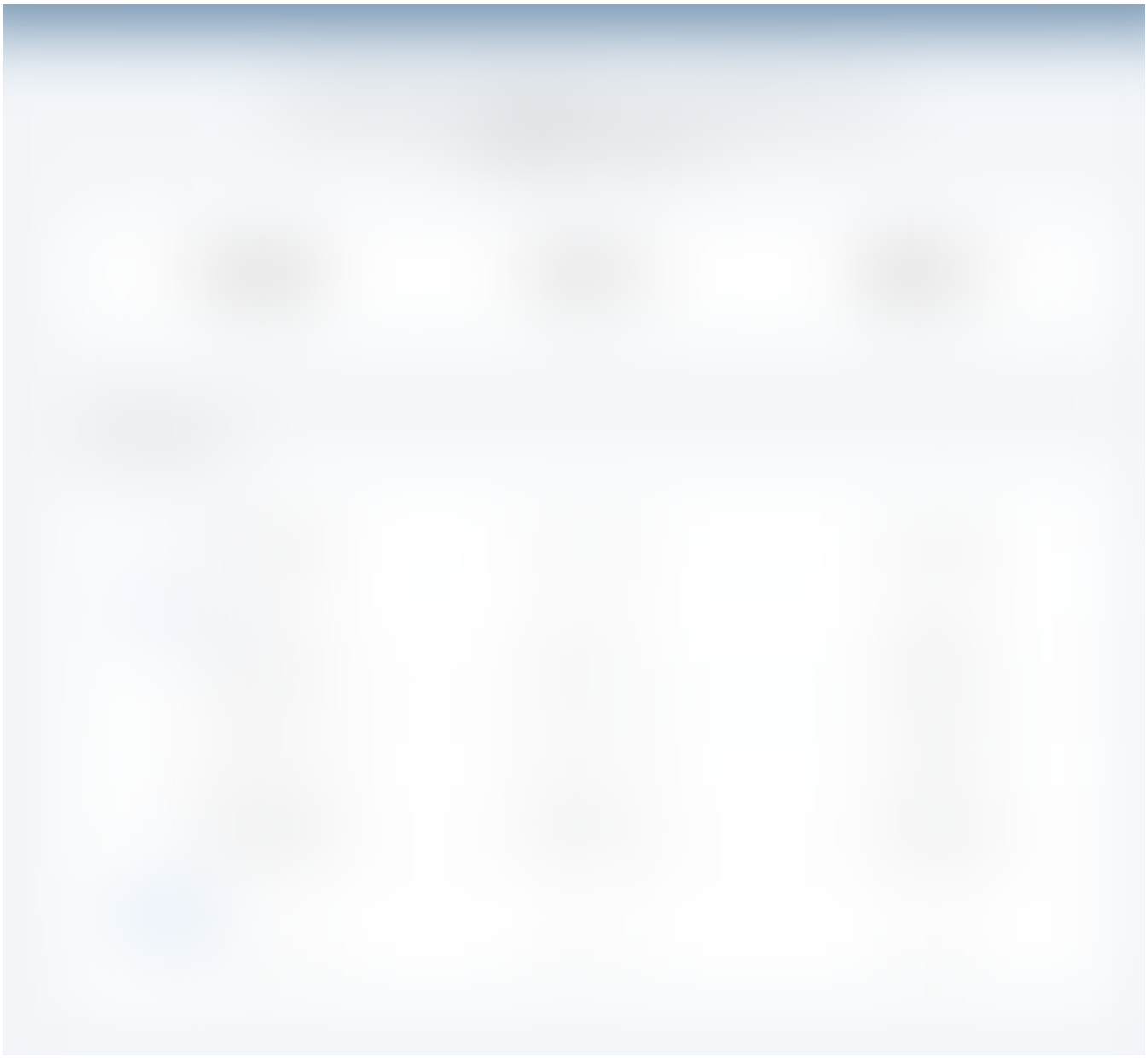
Finally, for `History.js`, we'll write the render function:

```
render() {
            return (
                <div className="history--section container">
                    <h2>History (Past 5 days)</h2>
                    <div className="history--section__box">
                        <div className="history--section__box__inner">
                            <h4>{this.state.todayprice.date}</h4>
                            <div className="columns">
                                <div className="column">
                                    <p>1 BTC = ${this.state.todayprice.btc}</p>
                                </div>
                                <div className="column">
                                    <p>1 ETH = ${this.state.todayprice.eth}</p>
                                </div>
                                <div className="column">
                                    <p>1 LTC = ${this.state.todayprice.ltc}</p>
                                </div>
                            </div>
                        </div>
                        <div className="history--section__box__inner">
                            <h4>{this.state.yesterdayprice.date}</h4>
                            <div className="columns">
                                <div className="column">
                                    <p>1 BTC = ${this.state.yesterdayprice.btc}</p>
                                </div>
                                <div className="column">
```

```
                    <p>1 ETH = ${this.state.yesterdayprice.eth}</p>
                </div>
                <div className="column">
                    <p>1 LTC = ${this.state.yesterdayprice.ltc}</p>
                </div>
            </div>
        </div>
        <div className="history--section__box__inner">
            <h4>{this.state.twodaysprice.date}</h4>
            <div className="columns">
                <div className="column">
                    <p>1 BTC = ${this.state.twodaysprice.btc}</p>
                </div>
                <div className="column">
                    <p>1 ETH = ${this.state.twodaysprice.eth}</p>
                </div>
                <div className="column">
                    <p>1 LTC = ${this.state.twodaysprice.ltc}</p>
                </div>
            </div>
        </div>
        <div className="history--section__box__inner">
            <h4>{this.state.threedaysprice.date}</h4>
            <div className="columns">
                <div className="column">
                    <p>1 BTC = ${this.state.threedaysprice.btc}</p>
                </div>
                <div className="column">
                    <p>1 ETH = ${this.state.threedaysprice.eth}</p>
                </div>
                <div className="column">
                    <p>1 LTC = ${this.state.threedaysprice.ltc}</p>
                </div>
            </div>
        </div>
        <div className="history--section__box__inner">
            <h4>{this.state.fourdaysprice.date}</h4>
            <div className="columns">
                <div className="column">
                    <p>1 BTC = ${this.state.fourdaysprice.btc}</p>
                </div>
                <div className="column">
                    <p>1 ETH = ${this.state.fourdaysprice.eth}</p>
                </div>
                <div className="column">
                    <p>1 LTC = ${this.state.fourdaysprice.ltc}</p>
                </div>
```

```
                            </div>
                        </div>

                    </div>
                </div>
            )
        }
    }

    export default History;
```

We can now run the `npm start` command to see the app at http://localhost:3000.



We can quickly check to see how the current state of this app would fare as a PWA. Remember we have a service worker file which currently caches all the resources

needed for this application. So you can run the `npm run build` command to put the app in production mode, and check its PWA status with Lighthouse.

We got a 91/100 score. Whoop! The only audit that failed to pass is the HTTPS audit which cannot be implemented right now because the app is still on a local server.

Our application is looking good and is quite fast (interactive at < 3s), let's add real-time functionalities by adding Pusher.

. . .

Make It Real-time With Pusher

By using Pusher, we can easily add real-time functionalities to the app. Pusher makes it simple to bind UI interactions to events that are triggered by any client or server. Let's set up Pusher.

Log into your dashboard (or create a new account if you're a new user) and create a new app. Copy your `app_id`, `key`, `secret`, and `cluster` and store them somewhere as we'll be needing them later.

We'll also need to create a server that will help with triggering events to Pusher and we'll create one with Node.js. In the root of your project directory, create a file named `server.js` and type in the following code:

```
// server.js
    const express = require('express')
    const path = require('path')
    const bodyParser = require('body-parser')
    const app = express()
    const Pusher = require('pusher')

    //initialize Pusher with your appId, key, secret and cluster
    const pusher = new Pusher({
        appId: 'APP_ID',
        key: 'APP_KEY',
        secret: 'APP_SECRET',
        cluster: 'YOUR_CLUSTER',
        encrypted: true
    })

    // Body parser middleware
    app.use(bodyParser.json())
    app.use(bodyParser.urlencoded({ extended: false }))

    // CORS middleware
    app.use((req, res, next) => {
        // Website you wish to allow to connect
        res.setHeader('Access-Control-Allow-Origin', '*')
        // Request methods you wish to allow
        res.setHeader('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT, PATCH, DELETE')
        // Request headers you wish to allow
        res.setHeader('Access-Control-Allow-Headers', 'X-Requested-With,content-type')
        // Set to true if you need the website to include cookies in the requests sent
        // to the API (e.g. in case you use sessions)
        res.setHeader('Access-Control-Allow-Credentials', true)
        // Pass to next layer of middleware
        next()
    })

    // Set port to be used by Node.js
    app.set('port', (5000))

    app.get('/', (req, res) => {
        res.send('Welcome')
    })

```

```
// API route in which the price information will be sent to from the clientside
app.post('/prices/new', (req, res) => {
    // Trigger the 'prices' event to the 'coin-prices' channel
    pusher.trigger( 'coin-prices', 'prices', {
        prices: req.body.prices
    });
    res.sendStatus(200);
})

app.listen(app.get('port'), () => {
    console.log('Node app is running on port', app.get('port'))
})
```

This is a simple Node.js server that uses Express as its web framework. Pusher is initialized with the dashboard credentials, and the various API routes are also defined. Don't forget to install the packages in use:

```
npm install express body-parser pusher
```

We'll also need to add a line of code to the `package.json` file so as to allow API proxying. Since we will be running a backend server, we need to find a way to run the React app and backend server together. API proxying helps with that.

To tell the development server to proxy any unknown requests (`/prices/new`) to your API server in development, add a `proxy` field to your `package.json` immediately after the `scripts` object:

```
"proxy": "http://localhost:5000"
```

We only need to make the current price real-time and that means we'll be working on the `Today` component, so open up the file. The Pusher JavaScript library is needed, so run `npm install pusher-js` to install that.

The first thing to do is import the `pusher-js` package:

```
import Pusher from 'pusher-js'
```

In the `componentWillMount` method, we establish a connection to Pusher using the credentials obtained from the dashboard earlier:

```
// establish a connection to Pusher
    this.pusher = new Pusher('APP_KEY', {
        cluster: 'YOUR_CLUSTER',
        encrypted: true
    });
    // Subscribe to the 'coin-prices' channel
    this.prices = this.pusher.subscribe('coin-prices');
```

We need a way to query the API every 10 seconds to retrieve the latest price information. We can use the `setInterval` function to send an API request every 10 seconds and then send the result of that API request to Pusher so that it can be broadcasted to other clients.

Before we create the setInterval function, let's create a simple function that takes in an argument and sends it to the backend server API:

```
sendPricePusher (data) {
   axios.post('/prices/new', {
       prices: data
   })
     .then(response => {
         console.log(response)
     })
     .catch(error => {
         console.log(error)
     })
}
```

Let's create the `setInterval` function. We will need to create a `componentDidMount` method so we can put the interval code in it:

```
componentDidMount () {
    setInterval(() => {
        axios.get('https://min-
api.cryptocompare.com/data/pricemulti?fsyms=BTC,ETH,LTC&tsyms=USD')
            .then(response => {
                this.sendPricePusher (response.data)
            })
            .catch(error => {
                console.log(error)
```

```
            })
        }, 10000)
    }
```

So right now, the app queries the API every 10 seconds and sends the data to Pusher, but we still haven't made the app real-time. We need to implement the real-time functionality so that other clients/users connected to the application can see the price change in real-time. That will be done by using Pusher's bind method.

Inside the `componentDidMount` method, add the code below, immediately after the `setInterval` function:

```
// We bind to the 'prices' event and use the data in it (price
information) to update the state values, thus, realtime changes
this.prices.bind('prices', price => {
   this.setState({ btcprice: price.prices.BTC.USD });
   this.setState({ ethprice: price.prices.ETH.USD });
   this.setState({ ltcprice: price.prices.LTC.USD });
 }, this);
```

The code block above, listens for data from Pusher, since we already subscribed to that channel and uses the data it gets to update the state values, thus, real-time changes. We now have a Progressive Real-time App! See a demo below:

. . .

Offline Strategies

Right now, if we were to go offline, our application would not be able to make API requests to get the various prices. So how do we make sure that we still able to see some data even when the network fails?

One way to go about it would be to use Client Side Storage. How would this work? We'll simply use localStorage to cache data.

localStorage makes it possible to store values in the browser which can survive the browser session. It is one type of the Web Storage API, which is an API for storing key-value pairs of data within the browser. It has a limitation of only storing strings. That means any data being stored has to be *stringified* with the use of `JSON.stringify`

It's important to note that there are other types of client-side storage, such as Session Storage, Cookies, IndexedDB, and WebSQL. Local storage can be used for a demo app like this, but in a production app, it's advisable to use a solution like IndexedDB which offers more features like better structure, multiple tables and databases, and more storage.

The goal will be to display the prices from `localStorage`. That means we'll have to save the results from various API requests into the `localStorage` and set the state to the values in the `localStorage`. This will ensure that when the network is unavailable and API requests are failing, we would still be able to see some data, albeit cached data. Let's do just that. Open up the `Today.js` file and edit the code inside the callback function of the API request to get prices with the one below:

```
    axios.get('https://min-api.cryptocompare.com/data/pricemulti?fsyms=BTC,ETH,LTC&tsyms=USD')
        .then(response => {
            this.setState({ btcprice: response.data.BTC.USD });
            localStorage.setItem('BTC', response.data.BTC.USD);

            this.setState({ ethprice: response.data.ETH.USD });
            localStorage.setItem('ETH', response.data.ETH.USD);

            this.setState({ ltcprice: response.data.LTC.USD });
            localStorage.setItem('LTC', response.data.LTC.USD);
        })
        .catch(error => {
```

```
            console.log(error)
        })
```

We are essentially storing the values gotten from the API request to the `localStorage`. With our values now in the `localStorage`, we'll need to set the state values to the saved values in `localStorage`. Inside the `componentDidMount` method, before the `setInterval` code, add the following code:

```
if (!navigator.onLine) {
        this.setState({ btcprice: localStorage.getItem('BTC') });
        this.setState({ ethprice: localStorage.getItem('ETH') });
        this.setState({ ltcprice: localStorage.getItem('LTC') });
    }
```

navigator.js hosted with ♡ by GitHub view raw

The code above is only executed when the browser is offline. We can check for internet connectivity by using `navigator.onLine`. The `navigator.onLine` property returns the online status of the browser. The property returns a boolean value, with `true` meaning online and `false` meaning offline.

Let's now implement localStorage for `History.js` too. We'll need to save the values from the API in these functions (`getTodayPrice(), getYesterdayPrice(), getTwoDaysPrice(), getThreeDaysPrice(), getFourDaysPrice()`) to the `localStorage`.

```
// getTodayPrice()
    localStorage.setItem('todayprice', JSON.stringify(f));
    this.setState({ todayprice: f });

    // getYesterdayPrice()
    localStorage.setItem('yesterdayprice', JSON.stringify(f));
    this.setState({ yesterdayprice: f });

    // getTwoDaysPrice()
    localStorage.setItem('twodaysprice', JSON.stringify(f));
    this.setState({ twodaysprice: f });

    // getThreeDaysPrice()
    localStorage.setItem('threedaysprice', JSON.stringify(f));
    this.setState({ threedaysprice: f });

    // getFourDaysPrice()
```

```
18      localStorage.setItem('fourdaysprice', JSON.stringify(f));
19      this.setState({ fourdaysprice: f });
```

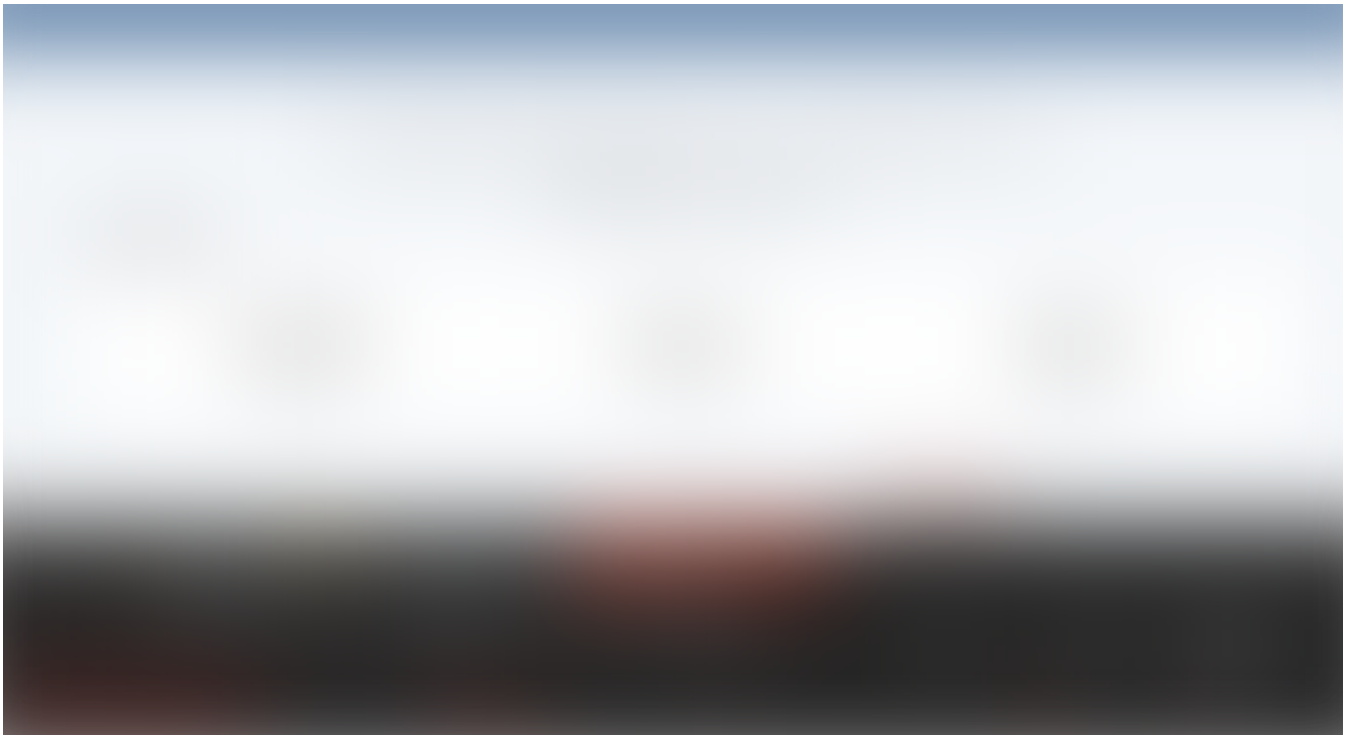
local.js hosted with ♡ by GitHub view raw

We are essentially storing the values gotten from the API request to the `localStorage`. With our values now in the `localStorage`, we'll also need to set the state values to the saved values in `localStorage` like we did in the `Today` component. Create a `componentDidMount` method and add the following code inside the method:

```
componentDidMount () {
        if (!navigator.onLine) {
            this.setState({ todayprice: JSON.parse(localStorage.getItem('todayprice')) });
            this.setState({ yesterdayprice: JSON.parse(localStorage.getItem('yesterdayprice')) }
            this.setState({ twodaysprice: JSON.parse(localStorage.getItem('twodaysprice')) });
            this.setState({ threedaysprice: JSON.parse(localStorage.getItem('threedaysprice')) }
            this.setState({ fourdaysprice: JSON.parse(localStorage.getItem('fourdaysprice')) });
        }
    }
```

today.js hosted with ♡ by GitHub view raw

Now our application will display cached values when there's no internet connectivity.



It's important to note that the app is time sensitive. Time-sensitive data are not really useful to users when cached. What we can do is, add a status indicator warning the

user when they are offline, that the data being shown might be stale and an internet connection is needed to show the latest data.

. . .

Deploy the App to Production

Now that we're done building, let's deploy the app to production and carry out a final Lighthouse test. We'll be using now.sh for deployment, `now` allows you to take your JavaScript (Node.js) or Docker powered websites, applications, and services to the cloud with ease. You can find installation instructions on the site. You can also use any other deployment solution, I'm using Now because of its simplicity.

Prepare the app for production by running the command below in the terminal:

```
npm run build
```

This builds the app for production to the `build` folder. Alright, so the next thing to do is to create a server in which the app will be served. Inside the `build` folder, create a file named `server.js` and type in the following code:

```
const express = require('express')
    const path = require('path')
    const bodyParser = require('body-parser')
    const app = express()
    const Pusher = require('pusher')

    const pusher = new Pusher({
        appId: 'APP_ID',
        key: 'YOUR_KEY',
        secret: 'YOUR SECRET',
        cluster: 'YOUR CLUSTER',
        encrypted: true
    })

    app.use(bodyParser.json())
    app.use(bodyParser.urlencoded({ extended: false }))
    app.use(express.static(path.join(__dirname)));

    app.use((req, res, next) => {
        // Website you wish to allow to connect
```

```
        res.setHeader('Access-Control-Allow-Origin', '*')
        // Request methods you wish to allow
        res.setHeader('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT, PATCH, DELETE')
        // Request headers you wish to allow
        res.setHeader('Access-Control-Allow-Headers', 'X-Requested-With,content-type')
        // Set to true if you need the website to include cookies in the requests sent
        // to the API (e.g. in case you use sessions)
        res.setHeader('Access-Control-Allow-Credentials', true)
        // Pass to next layer of middleware
        next()
    })

    app.set('port', (5000))

    app.get('/', (req, res) => {
        res.sendFile(path.join(__dirname + '/index.html'));
    });

    app.post('/prices/new', (req, res) => {
        pusher.trigger( 'coin-prices', 'prices', {
            prices: req.body.prices
        });
        res.sendStatus(200);
    })

    app.listen(app.get('port'), () => {
        console.log('Node app is running on port', app.get('port'))
    })
```

server.js is hosted with ♡ by GitHub view raw

This is basically the same code we wrote in the `server.js` file in the root of the project directory. The only addition here is that we set the home route to serve the `index.html` file in the `public` folder. Next up, run the command `npm init` to create a `package.json` file for us and lastly install the packages needed with the command below:

```
npm install express body-parser pusher
```

You can now see the application by running `node server.js` inside the `build` folder and your app should be live at http://localhost:5000.

Deploying to `now` is very easy, all you have to do is run the command `now deploy` and `now` takes care of everything, with a live URL automatically generated.

If everything goes well, your app should be deployed and live now, in this case, https://build-zrxionqses.now.sh/. Now automatically provisions all deployments with SSL, so we can finally generate the Lighthouse report again to check the PWA status. A live Lighthouse report of the site can be seen here.

. . .

App Install

One of the features of PWAs is the web app install banner. So how does this work? A PWA will install a web app install banner only if the following conditions are met:

- Has a web app manifest file with:
- A `short_name` (used on the home screen)
- A `name` (used in the banner)
- A 144x144 png icon (the icon declarations must include a mime type of `image/png`)
- A `start_url` that loads
- A service worker registered on your site
- Is served over HTTPS (a requirement for using service worker).

- Is visited at least twice, with at least five minutes between visits.

The `manifest.json` file in the `public` folder meets all the requirements above, we have a service worker registered on the site and the app is served over HTTPS at https://build-zrxionqses.now.sh/.

. . .

Conclusion

In this tutorial, we've seen how to use ReactJS, Pusher and service workers to build a realtime PWA. We saw how service workers can be used to cache assets and resources so as to reduce the load time and also make sure that the app works even when offline.

We also saw how to use localStorage to save data locally for cases when the browser loses connectivity to the internet.

The app can be viewed live here and you can check out the GitHub repo here. See if you can change stuff and perhaps make the app load faster!

JavaScript React Pwa Cryptocurrency