PROJECT **Create a Real-time Bitcoin Price Tracker in React**

Introduction

Suppose you want to create a standalone React.js component that interacts with an API that you can plug into your own app as a subcomponent. In this guide, we will take a look at building a Bitcoin (BTC) price tracker that is fully automated (no interaction with a user) and updates the price dynamically every three seconds from Coindesk's public API. We will take a look at the requirements of calling an API, recursive calls after a time interval, and binding data to a component's state for rendering.

Setup

First, create a React.js component called `BTCTracker` that makes use of the jQuery library (for API calls later on).

JAVASCVRIPT CODE:

```
import React from 'react';
import $ from 'jquery';

export default class BTCTracker extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      price: 0.00,
      lastFetch: ""
    }
  }

  render() {
    return (
      <div>
        <h1>
          BTC Price: {this.state.price}
          <small>
            {this.state.lastFetch}
          </small>
        </h1>
      </div>
    );
  }
}
```

This component maintains two state variables, namely:

1. `price`: The current price of Bitcoin against the US Dollar
2. `lastFetch`: The timestamp when it was last fetched from the API

These values are rendered in the component's UI and should change with every call of `setState()`.

Fetching from an API

Next, create a function to fetch data from a third-party API. In this case, you'll be using Coindesk's public API, which returns data in the following format:

CODE JSON:

```
{
  time: {
    updated: "Sep 16, 2020 18:43:00 UTC",
    updatedISO: "2020-09-16T18:43:00+00:00",
    updateduk: "Sep 16, 2020 at 19:43 BST"
  },
  disclaimer: "This data was produced from the CoinDesk Bitcoin Price Index
(USD). Non-USD currency data converted using hourly conversion rate from
openexchangerates.org",
  chartName: "Bitcoin",
  bpi: {
    USD: {
      code: "USD",
      symbol: "&#36;",
      rate: "11,091.2212",
      description: "United States Dollar",
      rate_float: 11091.2212
    },
    GBP: {
      code: "GBP",
      symbol: "&pound;",
      rate: "8,436.3156",
      description: "British Pound Sterling",
      rate_float: 8436.3156
    },
    EUR: {
      code: "EUR",
      symbol: "&euro;",
      rate: "9,395.0075",
      description: "Euro",
      rate_float: 9395.0075
    }
  }
}
```

Assuming that the returned value can be stored in a variable called `response`, you should set your state values to the following based on the response's structure:

1. `price` to `response.bpi.USD.rate` to get the USD rate
2. `lastFetch` to `response.time.updated` to get the last time this data was updated

Create a function called `fetch()` that contains the following:

CODE JAVASCRIPT:

```
fetch() {
  var context = this;

  window.setTimeout(function() {
    $.ajax({
      url: "https://api.coindesk.com/v1/bpi/currentprice.json",
      dataType: "json",
      method: "GET",
      success: function(response) {
        context.setState({
          price: response.bpi.USD.rate,
          lastFetch: response.time.updated
        });
      }
    });
  }, 3000);
}
```

The first thing you notice is that a `context` variable is used to refer to `this`. This is so your code can still refer to the component within the anonymous functions called in `setTimeout` or the one assigned to `success`. Remember

that you still have to invoke `setState`, which is a function of a React component to update the state values.
Second, `window.setTimeout(f, s)` is called where `f` is a function and `s` is the number of milliseconds until the function fires. In this case, `f` is set to be a function that invokes the jQuery ajax call to the API and `s` is set to 3000 (approximately three seconds).
Third, the function passed to `success` will accept `response` in the form of JSON, which is the JSON data returned by the API. It is assumed to be JSON due to the `dataType: "json"` that you pass to jQuery's `ajax()` call.
The final thing the `success` function will do is invoke `fetch()` again as called by `context`, which is a reference to the instance of the component. As a result, this rescursive strategy will call `fetch()` every three seconds, thus automating the call to the external API.

Calling fetch() the First Time

Since the user won't invoke `fetch()`, the method has to be called once `BTCTracker` is loaded. You can use the lifecycle method `componentDidMount()` to call any logic once a React.js component is loaded. In this case, `BTCTracker`'s `componentDidMount()` will simply call `fetch()`:

CODE JAVASCRIPT:

```
componentDidMount() {
  this.fetch();
}
```

Overall Code

The final code of the component looks like the following:

CODE JAVASCRIPT:

```
import React from 'react';
import $ from 'jquery';

export default class BTCTracker extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      price: 0.00,
      lastFetch: ""
    }
  }

  componentDidMount() {
    this.fetch();
  }

  fetch() {
    var context = this;

    window.setTimeout(function() {
      $.ajax({
        url: "https://api.coindesk.com/v1/bpi/currentprice.json",
        dataType: "json",
        method: "GET",
        success: function(response) {
          context.setState({
            price: response.bpi.USD.rate,
            lastFetch: response.time.updated
          });
        }
      });
    }, 3000);
  }
  render() {
    return (
      <div>
        <h1>
          BTC Price: {this.state.price}
          <small>
            {this.state.lastFetch}
          </small>
        </h1>
      </div>
    );
  }
}
```

Try mounting the component on your page and start tracking Bitcoin's rate!

CODE JAVASCRIPT:

```
ReactDOM.render(
  <BTCTracker />,
  document.getElementById("react-root")
);
```

The price should refresh every three seconds.

Conclusion

There might be cases where you need a React.js component to perform autonomously without any user interaction. This allows information to be dynamically rendered, allowing a richer user experience, and provides more options for information communicated back to the user. This is done through recursive calls within small time intervals. Since data is often taken from a third-party API, this approach allows the creation of more information-driven, self-serving components for your interface, such as dashboards with aggregated information from various sources.

As an exercise, try to integrate the other attributes returned by Coindesk's API, such as the rate of Bitcoin against the British Pound and the Euro.

Source: https://app.pluralsight.com/guides/create-a-real-time-bitcoin-price-tracker-in-reactjs