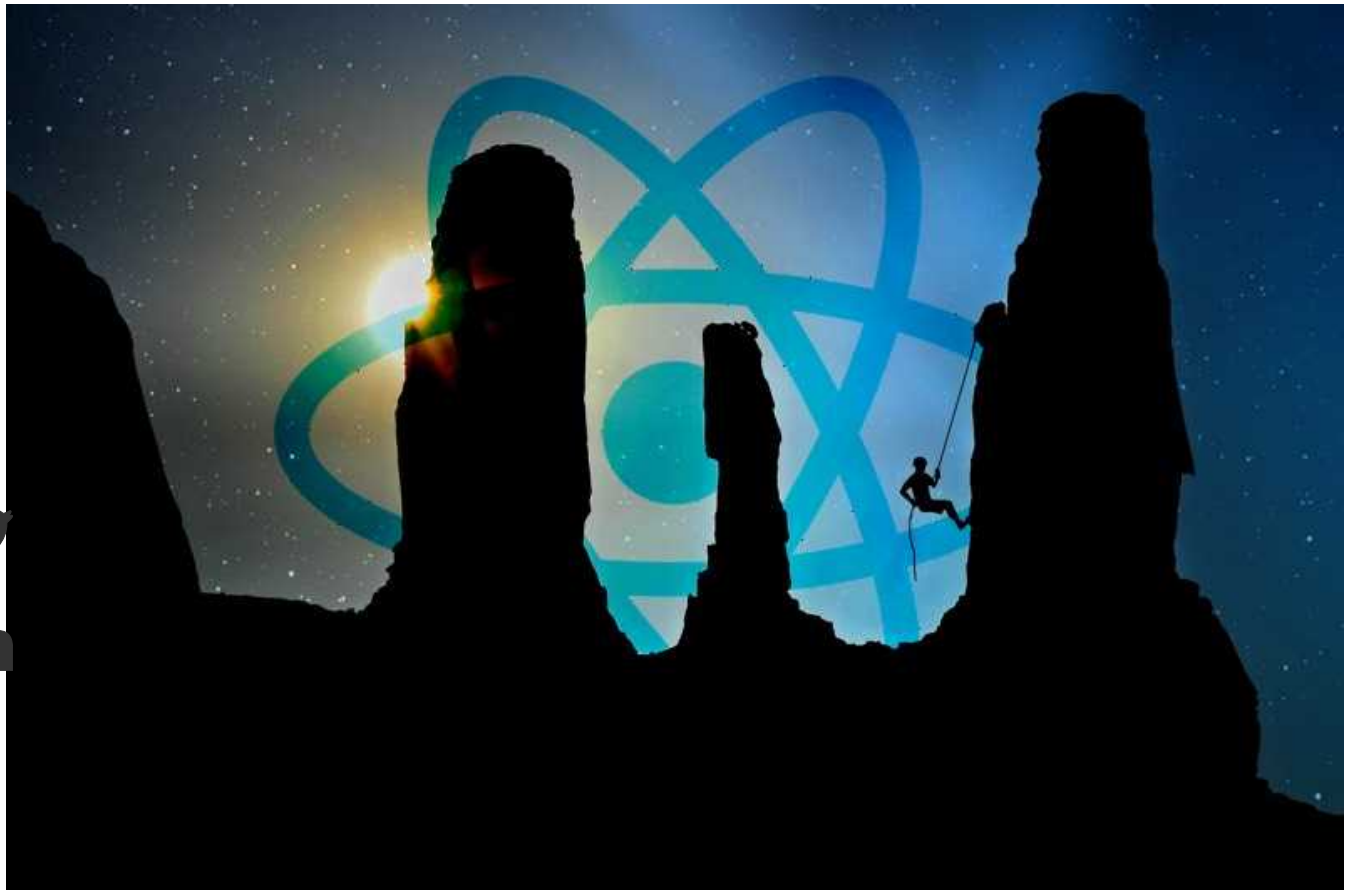
How to fetch data in React

JULY 06, 2018 BY ROBIN WIERUCH - EDIT THIS POST



Newcomers to React often start with applications that don't need data fetching at all. Usually they are confronted with Counter, Todo or TicTacToe applications. That's good, because data fetching adds another layer of complexity to your application while taking the first steps in React.

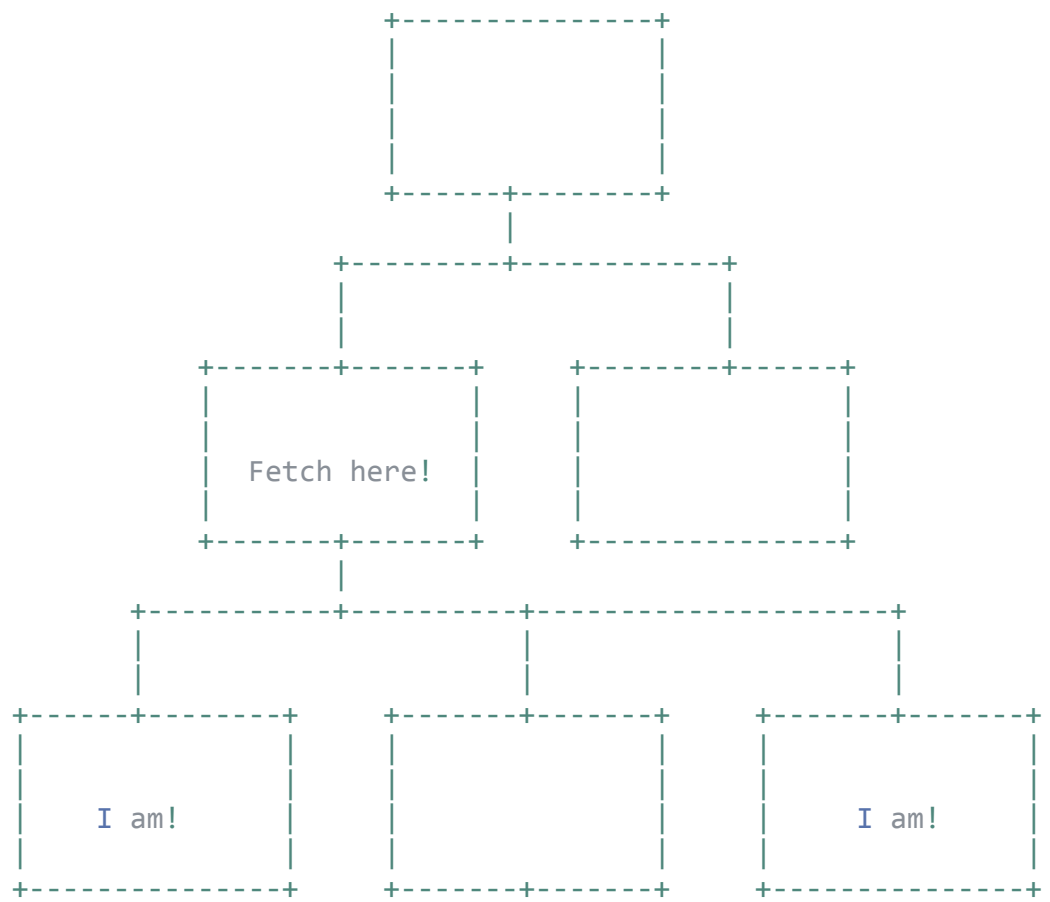
However, at some point you want to request real world data from an own or a third-party API. The article gives you a walkthrough on how to fetch data in React. There is no external state management solution, such as Redux or MobX, involved to store your fetched data. Instead you will use React's local state management.

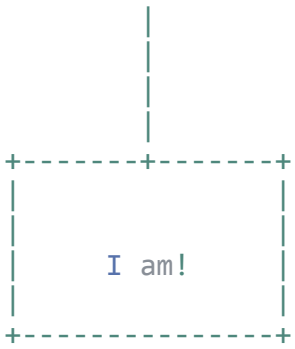
TABLE OF CONTENTS

WHERE TO FETCH IN REACT'S COMPONENT TREE?

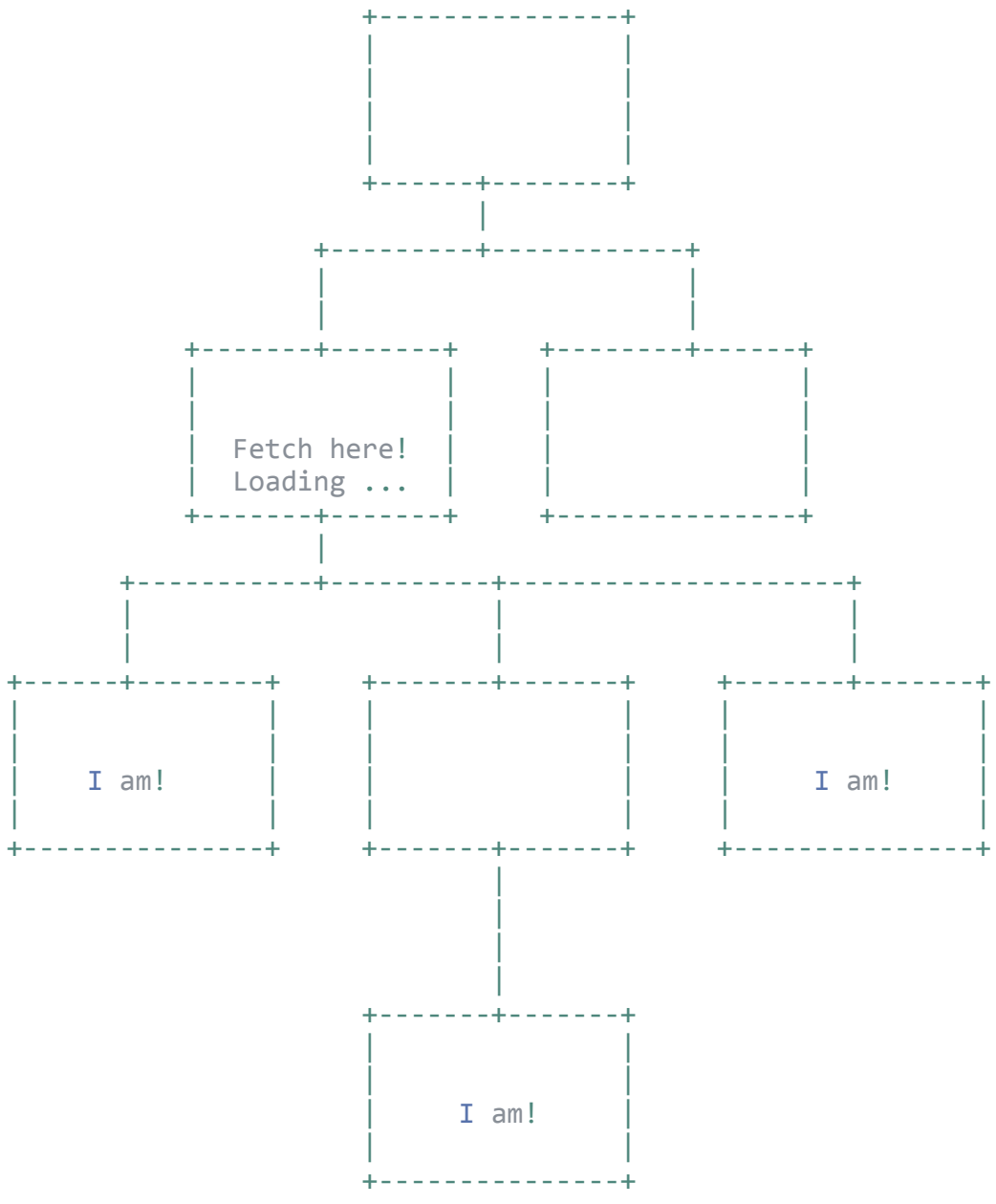
Imagine you already have a component tree that has several levels of components in its hierarchy. Now you are about to fetch a list of items from a third-party API. Which level in your component hierarchy, to be more precise, which specific component, should fetch the data now? Basically it depends on three criteria:

1. Who is interested in this data? The fetching component should be a common parent component for all these components.

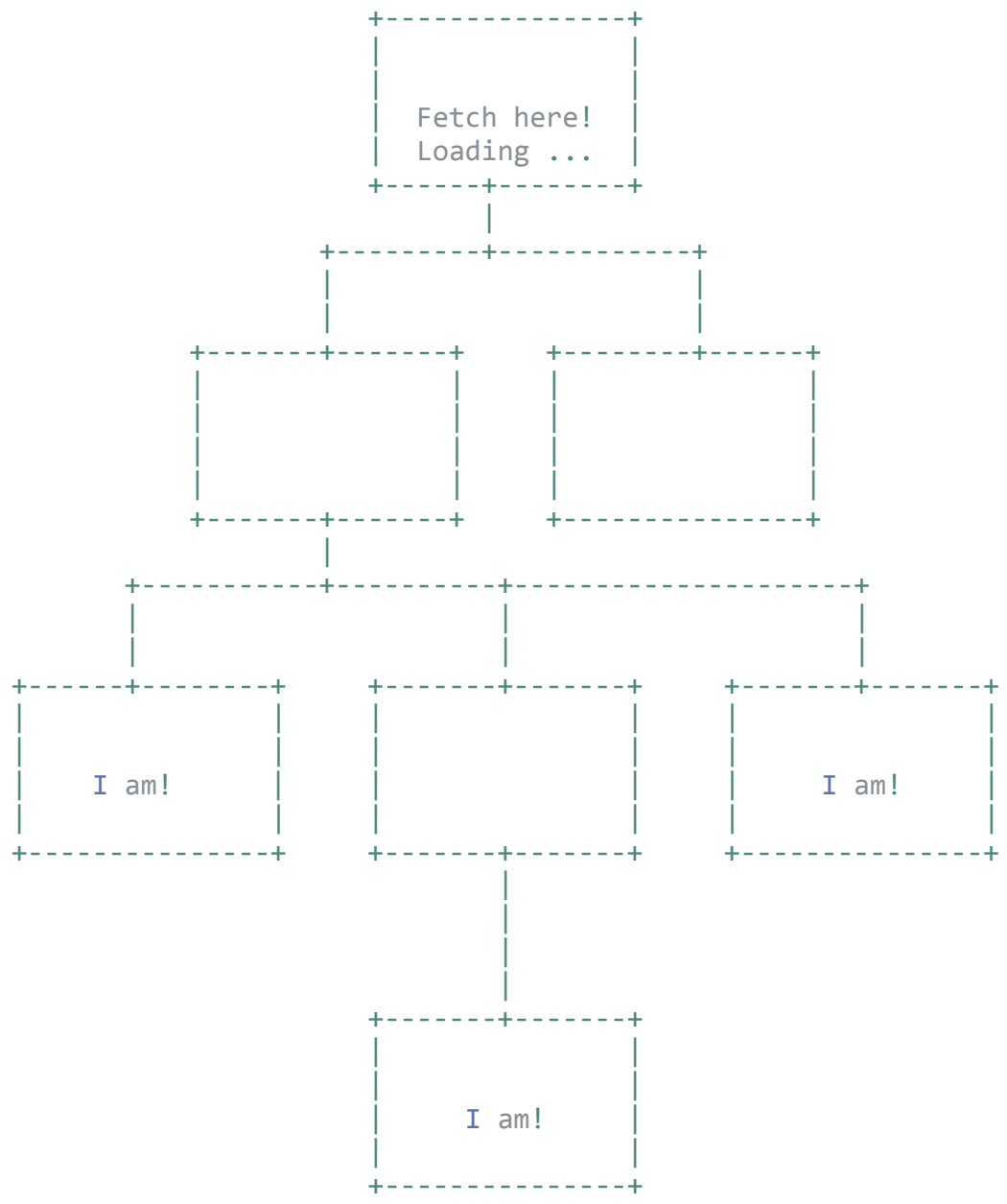




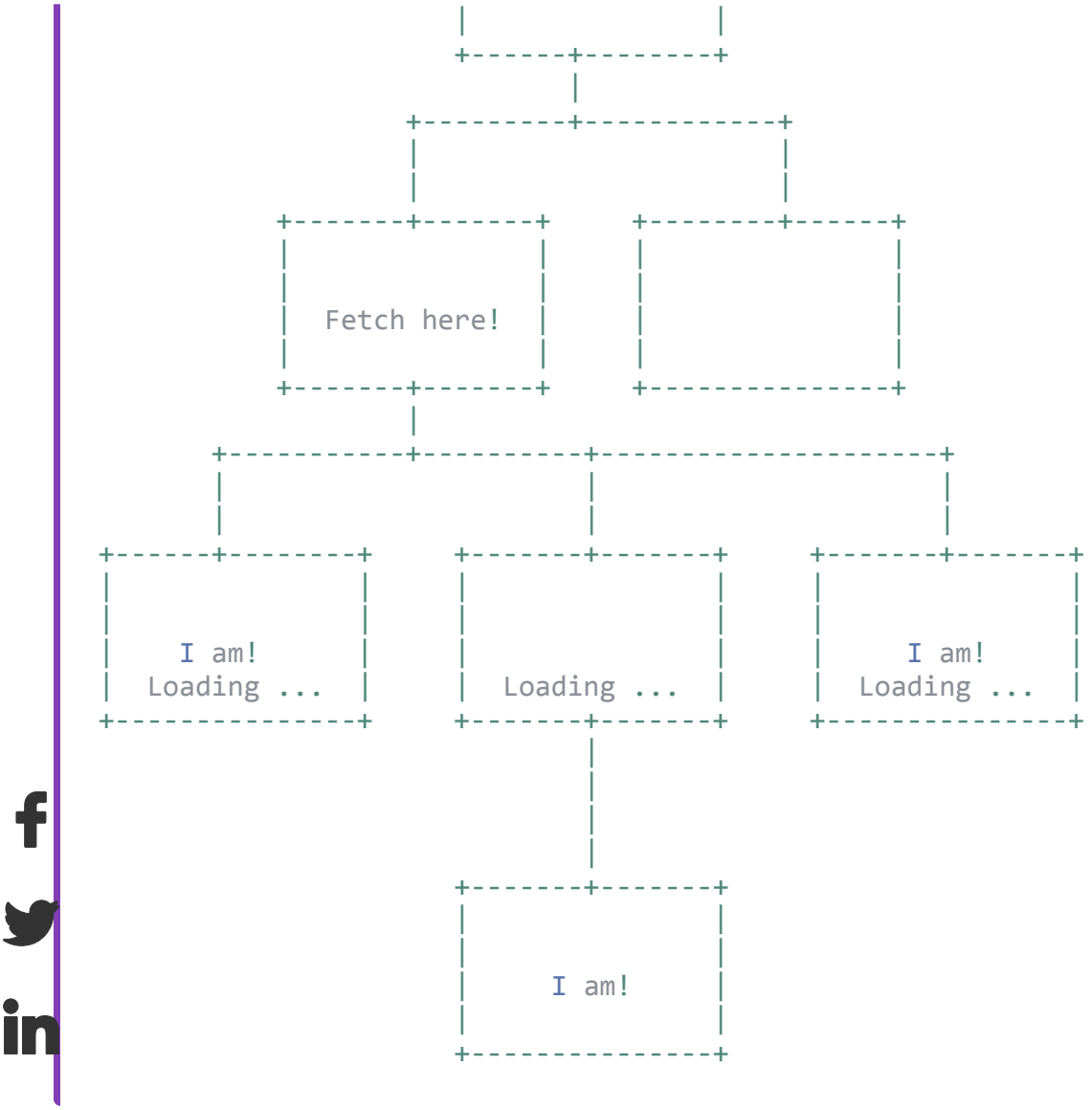
2. Where do you want to show a conditional loading indicator (e.g. loading spinner, progress bar) when the fetched data from the asynchronous request is pending? The loading indicator could be shown in the common parent component from the first criteria. Then the common parent component would still be the component to fetch the data.



2.1. But when the loading indicator should be shown in a more top level component, the data fetching needs to be lifted up to this component.



2.2. When the loading indicator should be shown in child components of the common parent component, not necessarily the components that need the data, the common parent component would still be the component to fetch the data. The loading indicator state could then be passed down to all child components that would be interested to show a loading indicator.



3. Where do you want to show an optional error message when the request fails? Here the same rules from the second criteria for the loading indicator apply.

That's basically everything on **where to fetch the data** in your React component hierarchy. But when should the data be fetched and how should it be fetched once the common parent component is agreed on?

HOW TO FETCH DATA IN REACT?

React's ES6 class components have lifecycle methods. The `render()` lifecycle method is mandatory to output a React element, because after all you may want to display the fetched data at some point.

There is another lifecycle method that is a perfect match to fetch data: `componentDidMount()`. When this method runs, the component was already rendered once with the `render()` method,

but it would render again when the fetched data would be stored in the local state of the component with `setState()`. Afterward, the local state could be used in the `render()` method to display it or to pass it down as props.

The `componentDidMount()` lifecycle method is the best place to fetch data. But how to fetch the data after all? React's ecosystem is a flexible framework, thus you can choose your own solution to fetch data. For the sake of simplicity, the article will showcase it with the native fetch API that comes with the browser. It uses JavaScript promises to resolve the asynchronous response. The most minimal example to fetch data would be the following:

```
import React, { Component } from 'react';

class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      data: null,
    };
  }

  componentDidMount() {
    fetch('https://api.mydomain.com')
      .then(response => response.json())
      .then(data => this.setState({ data }));
  }

  ...
}

export default App;
```

That's the most basic React.js fetch API example. It shows you how to get JSON in React from an API. However, the article is going to demonstrate it with a real world third-party API:

```
import React, { Component } from 'react';

const API = 'https://hn.algolia.com/api/v1/search?query=';
const DEFAULT_QUERY = 'redux';

class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      hits: [],
    };
  }
```

```
  componentDidMount() {
    fetch(API + DEFAULT_QUERY)
      .then(response => response.json())
      .then(data => this.setState({ hits: data.hits }));
  }

  ...
}

export default App;
```

The example uses the Hacker News API but feel free to use your own API endpoints. When the data is fetched successfully, it will be stored in the local state with React's `this.setState()` method. Then the `render()` method will trigger again and you can display the fetched data.

```
...

class App extends Component {
 ...

  render() {
    const { hits } = this.state;

    return (
      <ul>
        {hits.map(hit =>
          <li key={hit.objectID}>
            <a href={hit.url}>{hit.title}</a>
          </li>
        )}
      </ul>
    );
  }
}

export default App;
```

Even though the `render()` method already ran once before the `componentDidMount()` method, you don't run into any null pointer exceptions because you have initialized the `hits` property in the local state with an empty array.

Note: If you want to get to know data fetching with a feature called React Hooks, checkout this comprehensive tutorial: How to fetch data with React Hooks?

WHAT ABOUT LOADING SPINNER AND ERROR HANDLING?

Of course you need the fetched data in your local state. But what else? There are two more properties that you could store in the state: loading state and error state. Both will improve your user experience for end-users of your application.

The loading state should be used to indicated that an asynchronous request is happening. Between both `render()` methods the fetched data is pending due to arriving asynchronously. Thus you can add a loading indicator during the time of waiting. In your fetching lifecycle method, you would have to toggle the property from false to true and when the data is resolved from true to false.

```
...

class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      hits: [],
      isLoading: false,
    };
  }

  componentDidMount() {
    this.setState({ isLoading: true });

    fetch(API + DEFAULT_QUERY)
      .then(response => response.json())
      .then(data => this.setState({ hits: data.hits, isLoading: false }));
  }

  ...
}

export default App;
```

In your `render()` method you can use React's conditional rendering to display either a loading indicator or the resolved data.

```
...

class App extends Component {
  ...

  render() {
    const { hits, isLoading } = this.state;

    if (isLoading) {
      return <p>Loading ...</p>;
    }
```

```
    return (
      <ul>
        {hits.map(hit =>
          <li key={hit.objectID}>
            <a href={hit.url}>{hit.title}</a>
          </li>
        )}
      </ul>
    );
  }
}
```

A loading indicator can be as simple as a Loading... message, but you can also use third-party libraries to show a spinner or pending content component. It is up to you to signalize your end-user that the data fetching is pending.

The second state that you could keep in your local state would be an error state. When an error occurs in your application, nothing is worse than giving your end-user no indication about the error.

```
...

class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      hits: [],
      isLoading: false,
      error: null,
    };
  }

  ...

}
```

When using promises, the `catch()` block is usually used after the `then()` block to handle errors. That's why it can be used for the native fetch API.

```
...

class App extends Component {

  ...

  componentDidMount() {
    this.setState({ isLoading: true });
```

```
    fetch(API + DEFAULT_QUERY)
      .then(response => response.json())
      .then(data => this.setState({ hits: data.hits, isLoading: false }))
      .catch(error => this.setState({ error, isLoading: false }));
  }

  ...

}
```

Unfortunately, the native fetch API doesn't use its catch block for every erroneous status code. For instance, when a HTTP 404 happens, it wouldn't run into the catch block. But you can force it to run into the catch block by throwing an error when your response doesn't match your expected data.

```
...

class App extends Component {

  ...

  componentDidMount() {
    this.setState({ isLoading: true });

    fetch(API + DEFAULT_QUERY)
      .then(response => {
        if (response.ok) {
          return response.json();
        } else {
          throw new Error('Something went wrong ...');
        }
      })
      .then(data => this.setState({ hits: data.hits, isLoading: false }))
      .catch(error => this.setState({ error, isLoading: false }));
  }

  ...

}
```

Last but not least, you can show the error message in your `render()` method as conditional rendering again.

```
...

class App extends Component {

  ...
```

```
  render() {
    const { hits, isLoading, error } = this.state;

    if (error) {
      return <p>{error.message}</p>;
    }

    if (isLoading) {
      return <p>Loading ...</p>;
    }

    return (
      <ul>
        {hits.map(hit =>
          <li key={hit.objectID}>
            <a href={hit.url}>{hit.title}</a>
          </li>
        )}
      </ul>
    );
  }
}
```

That's all about the basics in data fetching with plain React. You can read more about managing the fetched data in React's local state or libraries such as Redux in The Road to Redux.

HOW TO FETCH DATA WITH AXIOS IN REACT

As already mentioned, you can substitute the native fetch API with another library. For instance, another library might run for every erroneous requests into the catch block on its own without you having to throw an error in the first place. A great candidate as a library for fetching data is axios. You can install axios in your project with `npm install axios` and then use it instead of the native fetch API in your project. Let's refactor the previous project for using axios instead of the native fetch API for requesting data in React.

```
import React, { Component } from 'react';
import axios from 'axios';

const API = 'https://hn.algolia.com/api/v1/search?query=';
const DEFAULT_QUERY = 'redux';

class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
```

```
      hits: [],
      isLoading: false,
      error: null,
    };
  }

  componentDidMount() {
    this.setState({ isLoading: true });

    axios.get(API + DEFAULT_QUERY)
      .then(result => this.setState({
        hits: result.data.hits,
        isLoading: false
      }))
      .catch(error => this.setState({
        error,
        isLoading: false
      }));
  }

  ...
}

export default App;
```

As you can see, axios returns a JavaScript promise as well. But this time you don't have to resolve the promise two times, because axios already returns a JSON response for you. Furthermore, when using axios you can be sure that all errors are caught in the `catch()` block. In addition, you need to adjust the data structure slightly for the returned axios data.

The previous example has only shown you how to get data in React from an API with a HTTP GET method in React's componentDidMount lifecycle method. However, you can also actively request data with a button click. Then you wouldn't use a lifecycle method, but your own class method.

```
import React, { Component } from 'react';
import axios from 'axios';

const API = 'https://hn.algolia.com/api/v1/search?query=';
const DEFAULT_QUERY = 'redux';

class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      hits: [],
      isLoading: false,
      error: null,
    };
  }
```

```
  getStories() {
    this.setState({ isLoading: true });

    axios.get(API + DEFAULT_QUERY)
      .then(result => this.setState({
        hits: result.data.hits,
        isLoading: false
      }))
      .catch(error => this.setState({
        error,
        isLoading: false
      }));
  }

  ...
}

export default App;
```

But that's only the GET method in React. What about writing data to an API? When having axios in place, you can do a post request in React as well. You only need to swap the `axios.get()` with a `axios.post()`.

HOW TO TEST DATA FETCHING IN REACT?

So what about testing your data request from a React component? There exists an extensive React testing tutorial about this topic, but here it comes in a nutshell. When you have setup your application with create-react-app, it already comes with Jest as test runner and assertion library. Otherwise you could use Mocha (test runner) and Chai (assertion library) for these purposes as well (keep in mind that the functions for the test runner and assertions vary then).

When testing React components, I often rely Enzyme for rendering the components in my test cases. Furthermore, when it comes to testing asynchronous data fetching, Sinon is helpful for spying and mocking data.

```
npm install enzyme enzyme-adapter-react-16 sinon --save-dev
```

Once you have your test setup, you can write your first test suite for the data request in React scenario.

```
import React from 'react';
import axios from 'axios';
```

```
import sinon from 'sinon';
import { mount, configure} from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';

import App from './';

configure({ adapter: new Adapter() });

describe('App', () => {
  beforeAll(() => {

  });

  afterAll(() => {

  });

  it('renders data when it fetched data successfully', (done) => {

  });

  it('stores data in local state', (done) => {

  });
});
```

Whereas one test case should show that the data is rendered in the React component successfully after the data fetching, the other test verifies that the data is stored in the local state. Perhaps it is redundant to test both cases, because when the data is rendered it should be stored in the local state as well, but just for the sake of demonstrating it you will see both use cases.

Before all tests you want to stub your axios request with mocked data. You can create your own JavaScript promise for it and use it later to have fine-grained control over its resolve functionality.

```
...

describe('App', () => {
  const result = {
    data: {
      hits: [
        { objectID: '1', url: 'https://blog.com/hello', title: 'hello', },
        { objectID: '2', url: 'https://blog.com/there', title: 'there', },
      ],
    }
  };

  const promise = Promise.resolve(result);

  beforeAll(() => {
    sinon
      .stub(axios, 'get')
      .withArgs('https://hn.algolia.com/api/v1/search?query=redux')
```

```
      .returns(promise);
  });

  afterAll(() => {
    axios.get.restore();
  });

  ...
});
```

After all tests you should have sure to remove the stub from axios again. That's it for the asynchronous data fetching test setups. Now let's implement the first test:

```
...

describe('App', () => {
  ...

  it('stores data in local state', (done) => {
    const wrapper = mount(<App />);

    expect(wrapper.state().hits).toEqual([]);

    promise.then(() => {
      wrapper.update();

      expect(wrapper.state().hits).toEqual(result.data.hits);

      done();
    });
  });

  ...
});
```

In the test, you start to render the React component with Enzyme's `mount()` function which makes sure that all lifecycle methods are executed and all child components are rendered. Initially you can have an assertion for your hits being an empty array in the local state of the component. That should be true, because you initialze your local state with an empty array for the hits property. Once you resolve the promise and trigger your component's rendering manually, the state should have changed after the data fetching.

Next you can test whether everything renders accordingly. The test is similar to the previous test:

```
...

describe('App', () => {
  ...
```

```
  it('renders data when it fetched data successfully', (done) => {
    const wrapper = mount(<App />);

    expect(wrapper.find('p').text()).toEqual('Loading ...');

    promise.then(() => {
      wrapper.update();

      expect(wrapper.find('li')).toHaveLength(2);

      done();
    });
  });
});
```

In the beginning of the test, the loading indicator should be rendered. Again, once you resolve the promise and trigger your component's rendering manually, there should be two list elements for the requested data.

That's essentially what you need to know about testing data fetching in React. It doesn't need to be complicated. By having a promise on your own, you have fine-grained control over when to resolve the promise and when to update the component. Afterward you can conduct your assertions. The previous shown testing scenarios are only one way of doing it. For instance, regarding the test tooling you don't necessarily need to use Sinon and Enzyme.

HOW TO FETCH DATA WITH ASYNC/AWAIT IN REACT?

So far, you have only used the common way for dealing with JavaScript promises by using their `then()` and `catch()` blocks. What about the next generation of asynchronous requests in JavaScript? Let's refactor the previous data fetching example in React to async/await.

```
import React, { Component } from 'react';
import axios from 'axios';

const API = 'https://hn.algolia.com/api/v1/search?query=';
const DEFAULT_QUERY = 'redux';

class App extends Component {
  ...

  async componentDidMount() {
    this.setState({ isLoading: true });

    try {
```

```
      const result = await axios.get(API + DEFAULT_QUERY);

      this.setState({
        hits: result.data.hits,
        isLoading: false
      });
    } catch (error) {
      this.setState({
        error,
        isLoading: false
      });
    }
  }

  ...
}

export default App;
```

Instead of `then()` you can use the async/await statements when fetching data in React. The async statement is used to signalize that a function is executed asynchronously. It can be used on a (React) class component's method too. The await statement is used within the async function whenever something is executed asynchronously. So the next line is not executed before the awaited request resolves. Furthermore, a try and catch block can be used to catch the error in case the request fails.

HOW TO FETCH DATA IN HIGHER-ORDER COMPONENTS?

The previously shown ways to fetch data can be repetitive when using it in a lot of components. Once a component mounted, you want to fetch data and show conditional loading or error indicators. The component so far can be split up into two responsibilities: showing the fetched data with conditional renderings and fetching the remote data with storing it in local state afterward. Whereas the former is only there for rendering purposes, the latter could be made reusable by a higher-order component.

Note: When you are going to read the linked article, you will also see how you could abstract away the conditional renderings in higher-order components. After that, your component would only be concerned displaying the fetched data without any conditional renderings.

So how would you introduce such abstract higher-order component which deals with the data fetching in React for you. First, you would have to separate all the fetching and state logic into a higher-order component.

```
const withFetching = (url) => (Component) =>
  class WithFetching extends React.Component {
    constructor(props) {
      super(props);

      this.state = {
        data: null,
        isLoading: false,
        error: null,
      };
    }

    componentDidMount() {
      this.setState({ isLoading: true });

      axios.get(url)
        .then(result => this.setState({
          data: result.data,
          isLoading: false
        }))
        .catch(error => this.setState({
          error,
          isLoading: false
        }));
    }

    render() {
      return <Component { ...this.props } { ...this.state } />;
    }
  }
```

Except for the rendering, everything else within the higher-order component is taken from the previous component where the data fetching happened right in the component. In addition, the higher-order component receives an url that will be used to request the data. If you need to pass more query parameters to your higher-order component later on, you always can extend the arguments in the function signature.

```
const withFetching = (url, query) => (Comp) =>
  ...
```

In addition, the higher-order component uses a generic data container in the local state called `data`. It is not aware anymore of the specific property naming (e.g. hits) as before.

In the second step, you can dispose all of the fetching and state logic from your `App` component. Because it has no local state or lifecycle methods anymore, you can refactor it to a functional stateless component. The incoming property changes from the specific `hits` to the generic `data` property.

```
const App = ({ data, isLoading, error }) => {
  if (!data) {
    return <p>No data yet ...</p>;
  }

  if (error) {
    return <p>{error.message}</p>;
  }

  if (isLoading) {
    return <p>Loading ...</p>;
  }

  return (
    <ul>
      {data.hits.map(hit =>
        <li key={hit.objectID}>
          <a href={hit.url}>{hit.title}</a>
        </li>
      )}
    </ul>
  );
}
```

Last but not least, you can use the higher-order component to wrap your App component.

```
const API = 'https://hn.algolia.com/api/v1/search?query=';
const DEFAULT_QUERY = 'redux';

...

const AppWithFetch = withFetching(API + DEFAULT_QUERY)(App);
```

Basically that's it to abstract away the data fetching in React. By using higher-order components to fetch the data, you can easily opt-in this feature for any component with any endpoint API url. In addition, you can extend it with query parameters as shown before.

HOW TO FETCH DATA IN RENDER PROPS?

The alternative way of higher-order components are render prop components in React. It is possible to use a render prop component for declarative data fetching in React as well.

```
class Fetcher extends React.Component {
  constructor(props) {
    super(props);
```

```
    this.state = {
      data: null,
      isLoading: false,
      error: null,
    };
  }

  componentDidMount() {
    this.setState({ isLoading: true });

    axios.get(this.props.url)
      .then(result => this.setState({
        data: result.data,
        isLoading: false
      }))
      .catch(error => this.setState({
        error,
        isLoading: false
      }));
  }

  render() {
    return this.props.children(this.state);
  }
}
```

Then again you would be able to use the render prop component the following way in your App component:

```
const API = 'https://hn.algolia.com/api/v1/search?query=';
const DEFAULT_QUERY = 'redux';

...

const RenderPropApproach = () =>
  <Fetcher url={API + DEFAULT_QUERY}>
    {({ data, isLoading, error }) => {
      if (!data) {
        return <p>No data yet ...</p>;
      }

      if (error) {
        return <p>{error.message}</p>;
      }

      if (isLoading) {
        return <p>Loading ...</p>;
      }

      return (
        <ul>
          {data.hits.map(hit =>
            <li key={hit.objectID}>
```

```
              <a href={hit.url}>{hit.title}</a>
            </li>
          )}
        </ul>
      );
    }}
  </Fetcher>
```

By using React's children property as render prop, you are able to pass all the local state from the Fetcher component. That's how you can make all the conditional rendering and the final rendering within your render prop component.

HOW TO FETCH DATA FROM A GRAPHQL API IN REACT?

Last but not least, the article should shortly mention GraphQL APIs for React. How would you fetch data from a GraphQL API instead of a REST API (which you have used so far) from a React component? Basically it can be achieved the same way, because GraphQL is not opinionated about the network layer. Most GraphQL APIs are exposed over HTTP whether it is possible to query them with the native fetch API or axios too. If you are interested in how you would fetch data from a GraphQL API in React, head over to this article: A complete React with GraphQL Tutorial.

. . .

You can find the finished project in this GitHub repository. Do you have any other suggestions for data fetching in React? Please reach out to me. It would mean lots to me if you would share the article to others for learning about data fetching in React.

HOW TO TEST AXIOS IN JEST BY EXAMPLE

Every once in a while we need to test API requests. Axios is one of the most popular JavaScript libraries to fetch data from remote APIs . Hence, we will use Axios for our data fetching example...

SETUP POSTGRESQL WITH SEQUELIZE IN EXPRESS

Eventually every Node.js project running with Express.js as web application will need a database. Since most server applications are stateless, in order to scale them horizontally with multiple server...

How to support me

Contact Me Privacy & Terms