VALENTINO GAGLIARDI / HIRE

Last updated: February 7, 2020 by Valentino Gagliardi - 6 minutes read

How To Use Async Await in React (componentDidMount Async)

How to use Async Await in React? In this post we'll see how to fix regeneratorRuntime and and how to handle errors with Fetch and async/await.

Do you want to use `async/await` in React? create-react-app supports `async/await` out of the box. But if you work with your own webpack boilerplate you may hit **regeneratorRuntime is not defined**.

In the following post you'll see how to use `async/await` in React and how to fix such error. You'll learn how to:

- fix regeneratorRuntime is not defined
- use `async/await` in React with Fetch
- handle errors with Fetch and `async/await`

Disclaimer

React is changing fast and and the method exposed here could be already obsolete by the time you'll see this article, originally wrote in June 2018. Take the post with a grain of salt. In particular you may want to investigate **React Hooks, which have they're own way for fetching data**.

How To Use async/await in React: what is async/await?

`async/await` in JavaScript is nothing more than syntactic sugar over Promises.

Why so? Are JavaScript Promises not enough? Promises are fine, yet in some situations you may end up with a long chain of then/catch.

I would argue, if you find yourself in that position better you simplify the code, But `async/await` helps writing asynchronous code in a way that looks synchronous.

It makes your code cleaner and readable. Plus you can use try/catch for proper error handling. `async/await` is convenient and clean: at some point you may want to introduce it in your React components too.

Let's see how.

How To Use Async Await in React: an example with Promises

Before starting off make sure to have a React development environment. To make one you can follow this tutorial of mine: How to set up React, webpack, and babel or you can also use **create-react-app**.

Let's say you want to **fetch data from an API**. It's standard React stuff, you put the API call in `componentDidMount` and that's it:

```
import React, { Component } from "react";
import ReactDOM from "react-dom";

class App extends Component {
  constructor() {
    super();
    this.state = { data: [] };
  }
```

```
  componentDidMount() {
    fetch(`https://api.coinmarketcap.com/v1/ticker/?limit=10`
      .then(res => res.json())
      .then(json => this.setState({ data: json }));
  }

  render() {
    return (
      <div>
        <ul>
          {this.state.data.map(el => (
            <li>
              {el.name}: {el.price_usd}
            </li>
          ))}
        </ul>
      </div>
    );
  }
}

export default App;

ReactDOM.render(<App />, document.getElementById("app"));
```

(This component is a contrived example: there's no error handling. Let's assume we're in the happy path and nothing goes wrong with our fetch call).

NOTE: you can write the very same component as a function with the `useEffect` hook.

If you run webpack-dev-server with:

```
npm start
```

you'll see the code working as expected. Nothing fancy right? Can we do better? Would be nice to use `async/await` in `componentDidMount` right? Let's try!

How To Use Async Await in React: using the async/await syntax

Supported since version 7.6.0, `async/await` is widely used in Node.js. On the front-end it's another story. `async/await` is not supported by older browsers. (I know, who cares about IE?).

Anyway, using `async/await` in React requires no magic. But where to put `async/await` in a React component? Used mostly for data fetching and other initialization stuff `componentDidMount` is a nice place for `async/await`.

Here are the steps to follow:

- put the `async` keyword in front of your functions
- use `await` in the function's body
- catch any errors

Now, **create-react-app supports `async/await` out of the box**. But if you have a webpack boilerplate you may hit an error (more in a minute). Now let's apply `async/await` to our React component. Open up `App.js` and adjust `componentDidMount`:

```
import React, { Component } from "react";
import ReactDOM from "react-dom";

class App extends Component {
  constructor() {
    super();
    this.state = { data: [] };
  }

  async componentDidMount() {
    const response = await fetch(`https://api.coinmarketcap.
    const json = await response.json();
    this.setState({ data: json });
  }

  render() {
    return (
      <div>
        <ul>
          {this.state.data.map(el => (
            <li>
              {el.name}: {el.price_usd}
            </li>
          ))}
        </ul>
      </div>
    );
  }
}

export default App;

ReactDOM.render(<App />, document.getElementById("app"));
```

No error catching, again, let's assume nothing goes wrong with our fetch call. Take a look at the browser's console: **regeneratorRuntime is not defined?**.

What's that? The key for fixing that error are **babel preset env and babel plugin transform runtime**. Make sure to install them:

```
npm i @babel/preset-env @babel/plugin-transform-runtime @babe
```

Open up `.babelrc` and update the configuration as follow:

```
{
  "presets": [
    [
      "@babel/preset-env",
      {
        "targets": {
          "browsers": [
            ">0.25%",
            "not ie 11",
            "not op_mini all"
          ]
        }
      }
    ],
    "@babel/preset-react"
  ],
  "plugins": [
    "@babel/plugin-transform-runtime"
  ]
}
```

Save the file (run the build again if you don't have webpack-dev-server) and check out the browser. It **should work**! But we're not done yet. How about errors? What happens if the user goes offline or the **API goes down**? In the next section we'll se **how to handle errors with Fetch** and `async/await`.

How To Use Async Await in React: handling errors

The example we saw so far doesn't handle errors. Granted, **in real world apps you would decouple fetch calls from the view**. Moreover, the Fetch API has some caveats when it comes to handling errors.

Let's experiment with our component. Induce an error by removing "coinmarketcap" from the url:

```
async componentDidMount() {
    const response = await fetch(`https://api.com/v1/ticker/
    const json = await response.json();
    this.setState({ data: json });
}
```

Run the code and check the console. You get:

```
TypeError: NetworkError when attempting to fetch resource //

Uncaught (in promise) TypeError: Failed to fetch // in Chrom
```

There is an uncaught error of course. Let's catch it. Add a `try/catch` block:

```
async componentDidMount() {
    try {
      const response = await fetch(`https://api.com/v1/ticke
      const json = await response.json();
      this.setState({ data: json });
    } catch (error) {
      console.log(error);
    }
}
```

And re-run the code. You'll see the error logged to the console. So here's the **first thing: wrap Fetch in a `try/catch` to handle network errors**. Now let's try another thing. Check this out:

```
async componentDidMount() {
    try {
      const response = await fetch(`http://httpstat.us/500`)
    } catch (error) {
      console.log(error);
    }
}
```

What do you see in the console? **Nothing. No errors, no sign of life. Why? Fetch will only reject a Promise if there is a network error**. In **case of 404 or 500 you'll see no errors. That means you must check the response object:**

```
async componentDidMount() {
    try {
      const response = await fetch(`http://httpstat.us/500`)
      if (!response.ok) {
        throw Error(response.statusText);
      }
    } catch (error) {
      console.log(error);
    }
}
```

Now the error is logged to the console as expected. **At this point you can show the user some error message.**

How To Use Async Await in React: wrapping up

Used mostly for data fetching and other initialization stuff `componentDidMount` is a nice place for `async/await` in React. Here are the steps you need to follow for using `async/await` in React:

- configure babel
- put the async keyword in front of `componentDidMount`
- use await in the function's body
- make sure to catch eventual errors

If you use Fetch API in your code be aware that it has some caveats when it comes to handling errors.

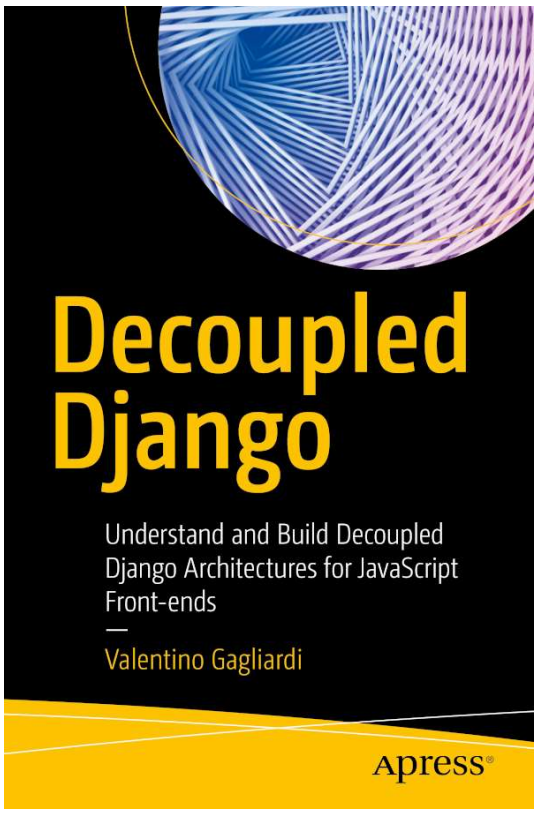
Thanks for reading and stay tuned!

STAY UPDATED

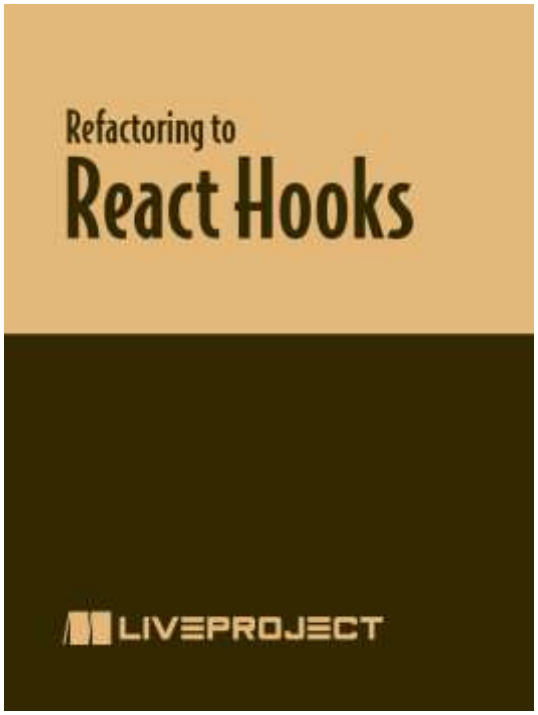
Be the first to know when I publish new stuff.

COURSES AND BOOKS

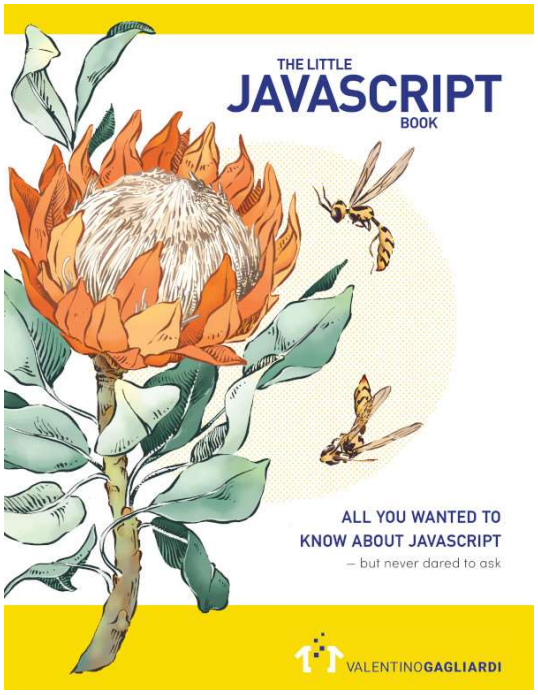
Decoupled Django



Refactoring to React Hooks



The Little JavaScript Book



Hi! I'm Valentino! I'm a freelance consultant with a wealth of experience in the IT industry. I spent the last years as a frontend consultant, providing advice and help, coaching and training on JavaScript and React. Let's get in touch!

More from the blog:

- React's useReducer with Redux Toolkit. Why not?
- 5 tips for dealing with untested React codebases
- Testing React Components with react-test-renderer, and the Act API
- Socket.IO, React and Node.js: Going Real-Time
- Demystifying Object.is and prevState in React useState
- Writing truly reusable React hooks, one test at a time
- React Hooks Tutorial: useState, useEffect, useReducer
- React Context API is not a state management tool