**React Hooks Tutorial for Beginners: what you will learn**

In the following tutorial you'll learn:

how to use React hooks

how the same logic would be implemented in React class components

**React Hooks Tutorial for Beginners: requirements**

To follow along with the tutorial you should have a basic understanding of:

* ECMAScript 2015 (arrow functions, destructuring, classes)
* React

**React Hooks Tutorial for Beginners: setting up the project**

If you want to follow along with the examples make sure to configure a React development environment. Run:

npx create-react-app exploring-hooks

and you're good to go!

(You should have one of the latest version of Node.js for running npx).

**React Hooks Tutorial for Beginners: in the beginning there was setState**

I won't go too deep here, I assume you're already using React in your project but let me do a quick recap.

React is a library for building user interfaces and one of its perks is that the library itself imposes a strict data flow to the developer. Do you remember jQuery? With jQuery it's almost impossible to clearly structure a project, let alone defining how the data should flow across the UI. It's hard to keep track of what function is changing what piece of UI.

The same applies to plain JavaScript: even if with self-disclipine and practice it's possibile to come up with a well structured project (thinking about the module pattern), good luck tracking state and interactions between functions (it can be done but it's hard without external help, see Redux).

These problems has been somewhat eased by React: by enforcing a clear structure (container and functional components) and a strict data flow (components react to state and props change) now its easier than before to create well reasoned UI logic.

So the theory in React is that a piece of UI can "react" in response to a state change. The basic form for expressing this flow was an ES6 class up until now. Consider the following example, an ES6 class extending from React.Component, with an internal state:

import React, { Component } from "react";

export default class Button extends Component {

constructor() {

super();

this.state = { buttonText: "Click me, please" };

this.handleClick = this.handleClick.bind(this);

}

handleClick() {

this.setState(() => {

return { buttonText: "Thanks, been clicked!" };

});

}

render() {

const { buttonText } = this.state;

return <button onClick={this.handleClick}>{buttonText}</button>;

}

}

As you can see from the code above the component's internal state gets mutated by setState when clicking the button. The text's button in turns reacts to this change and gets the updated text.

A more concise version of the component can be expressed by removing the constructor and leveraging the ECMAScript class field proposal:

import React, { Component } from "react";

export default class Button extends Component {

state = { buttonText: "Click me, please" };

handleClick = () => {

this.setState(() => {

return { buttonText: "Thanks, been clicked!" };

});

};

render() {

const { buttonText } = this.state;

return <button onClick={this.handleClick}>{buttonText}</button>;

}

}

So, in the beginning there was setState (and still it will be). But keep calm. The style above is perfectly fine and ES6 classes in React won't go away anytime soon.

But now with React hooks it's possible to express the flow internal state change -> UI reaction without using an ES6 class.

Follow me into the next section ...

**React Hooks Tutorial for Beginners: updating the state in React ... without setState**

So what options do we have for managing the internal state in React now that setState and classes are not a need anymore?

Enter the first and most important React hook: useState. useState is a function exposed by the react package. You will import that function at the top of your files as:

import React, { useState } from "react";

By importing useState in your code you're signaling the intent to hold some kind of state inside your React component. And more important, that React component shouldn't be an ES6 class anymore. It can be a pure and simple JavaScript function. This is the most appealing thing of this hooks story.

After importing useState you'll pick an array containing two variables out of useState, and the code should go inside your React component:

const [buttonText, setButtonText] = useState("Click me, please")

Confused by this syntax? It's ES6 destructuring. The names above can be anything you want, it doesn't matter for React. Anyway I advise using descriptive and meaningful variable names depending on the state's purpose.

The argument passed to useState is the actual starting state, the data that will be subject to changes. useState returns for you two bindings:

the actual value for the state

the state updater function for said state

So the previous example, a button component, with hooks becomes:

import React, { useState } from "react";

export default function Button() {

const [buttonText, setButtonText] = useState("Click me, please");

return (

<button onClick={() => setButtonText("Thanks, been clicked!")}>

{buttonText}

</button>

);

}

For calling the setButtonText state updater inside the onClick handler you can use an inline arrow function. But if you prefer using a regular function you can do:

import React, { useState } from "react";

export default function Button() {

const [buttonText, setButtonText] = useState("Click me, please");

function handleClick() {

return setButtonText("Thanks, been clicked!");

}

return <button onClick={handleClick}>{buttonText}</button>;

}

Must be honest, I fancy regular functions more than arrow functions, unless I have specific requirements. Readability improves a lot. Also, when I write code I think always of the next developer that will mantain that code. And my code should be readable.

React hooks, that's it! I could end this post here but not before showing you how to fetch data with hooks.

Head over the next section!

**React Hooks Tutorial for Beginners: in the beginning there was componentDidMount (and render props)**

Data fetching in React! Do you remember the old days of componentDidMount? You would slap fetch(url) in componentDidMount and call it a day. Here's how to fetch an array of data from an API for rendering out a nice list:

import React, { Component } from "react";

export default class DataLoader extends Component {

state = { data: [] };

componentDidMount() {

fetch("http://localhost:3001/links/")

.then(response => response.json())

.then(data =>

this.setState(() => {

return { data };

})

);

}

render() {

return (

<div>

<ul>

{this.state.data.map(el => (

<li key={el.id}>{el.title}</li>

</ul>

</div>

);

}

}

You could even use async/await in componentDidMount, with some caveats. But most of the asynchronous logic in my projects would live outside React components. There are yet a couple of shortcomings in the above code. The rendered list is fixed but with a render prop we can easily pass children as a function. The refactored component would look like the following:

import React, { Component } from "react";

export default class DataLoader extends Component {

state = { data: [] };

componentDidMount() {

fetch("http://localhost:3001/links/")

.then(response => response.json())

.then(data =>

this.setState(() => {

return { data };

})

);

}

render() {

return this.props.render(this.state.data);

}

}

And you would consume the component by providing a render prop from the outside:

<DataLoader

render={data => {

return (

<div>

<ul>

{data.map(el => (

<li key={el.id}>{el.title}</li>

))}

</ul>

</div>

);

}}

/>

Even this pattern (born for providing a nicer alternative to mixins and HOCs) has its shortcomings. And that's I guess the exact reason which lead React engineers to come up with hooks: provide a better ergonomics for encapsulating and reusing logic in React.

So impatient as I am, one of the first thing I wanted to try with hooks was data fetching. But what hook I'm supposed to use for fetching data? Would the component still use the render prop pattern?

Let's see into the next section!

**React Hooks Tutorial for Beginners: fetching data with useEffect**

I thought data fetching with React hooks shouldn't look so different from useState. A quick glance at the documentation gave me an hint: useEffect could be the right tool for the job.

I read: "useEffect serves the same purpose as componentDidMount, componentDidUpdate, and componentWillUnmount in React classes, but unified into a single API"

Bingo! It's amazing, isn't it? With this knowledge in hand I refactored the first version of Dataloader for using useEffect. The component becomes a function and fetch gets called inside useEffect. Moreover, instead of calling this.setState I can use setData (an arbitrary function extracted from useState):

import React, { useState, useEffect } from "react";

export default function DataLoader() {

const [data, setData] = useState([]);

useEffect(() => {

fetch("http://localhost:3001/links/")

.then(response => response.json())

.then(data => setData(data));

});

return (

<div>

<ul>

{data.map(el => (

<li key={el.id}>{el.title}</li>

))}

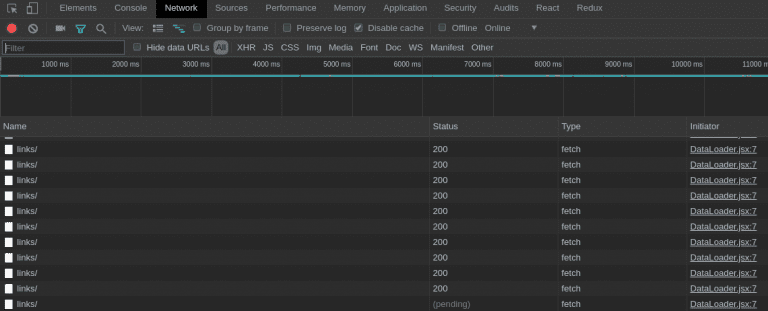
</ul>

</div>

);

}

At this point I thought "what could be wrong?" and I launched the app. This is what I saw in the console:

[](https://www.valentinog.com/blog/static/11c6d28ed166196b3a0bb8996fa8cb72/e5715/react-hooks-useEffect-loop-fetch.png)

It was clearly my fault because I've already got an hint of what was going on:

"useEffect serves the same purpose as componentDidMount, componentDidUpdate, and componentWillUnmount"

componentDidUpdate! componentDidUpdate is a lifecycle method running every time a component gets new props, or a state change happens. That's the trick. If you call useEffect like I did you would see an infinite loop. And for solving this "bug" you would need to pass an empty array as a second argument to useEffect:

//

useEffect(() => {

fetch("http://localhost:3001/links/")

.then(response => response.json())

.then(data => setData(data));

}, []); // << super important array

//

I wish this info got the visibility it deserves rather than being at the end of this page: [Using the Effect Hook](https://reactjs.org/docs/hooks-effect.html). But even with this info I wouldn't suggest rewriting all your React components to use hooks for fetching. A lot could still change in the near future.

Anyway, **useEffect replaces componentDidMount, componentDidUpdate, and componentWillUnmount**, which I think is a nice thing for both experts developers and newcomers to React.

**React Hooks Tutorial for Beginners: can I use render props with React hooks?**

Of course! But there's no point in doing that. Our DataLoader component becomes:

import React, { useState, useEffect } from "react";

export default function DataLoader(props) {

const [data, setData] = useState([]);

useEffect(() => {

fetch("http://localhost:3001/links/")

.then(response => response.json())

.then(data => setData(data));

}, []); // << super important array

return props.render(data)

}

And you would consume the component by providing a render prop from the outside as we did in the previous example.

But again, there's no point in this refactoring because React hooks were born for a reason: sharing logic between components, and we'll see an example in the next section.

**React Hooks Tutorial for Beginners: your first custom React hook**

Instead of HOCs and render props, we can encapsulate our logic in a React hook and then import that hook whenever we feel the need. In our example we can create a custom hooks for fetching data.

A custom hook is a JavaScript function whose name starts with "use", according to the React documentation. Easier done than said. Let's make a useFetch hook then:

// useFetch.js

import { useState, useEffect } from "react";

export default function useFetch(url) {

const [data, setData] = useState([]);

useEffect(() => {

fetch(url)

.then(response => response.json())

.then(data => setData(data));

}, []);

return data;

}

And this is how you would use the custom hook:

import React from "react";

import useFetch from "./useFetch";

export default function DataLoader(props) {

const data = useFetch("http://localhost:3001/links/");

return (

<div>

<ul>

{data.map(el => (

<li key={el.id}>{el.title}</li>

))}

</ul>

</div>

);

}

This is what make hooks so appealing: finally we have a nice, standardized, and clean way for encapsulating and sharing logic.

NOTE: I didn't account for fetch errors in the code above, do your homeworks!

**React Hooks Tutorial for Beginners: can I use async/await with useEffect?**

When playing with useEffect I wanted to try async/await inside the hook. Let's see our custom hook for a moment:

// useFetch.js

import { useState, useEffect } from "react";

export default function useFetch(url) {

const [data, setData] = useState([]);

useEffect(() => {

fetch(url)

.then(response => response.json())

.then(data => setData(data));

}, []);

return data;

}

For refactoring to async/await the most natural thing you would do is probably:

// useFetch.js

import { useState, useEffect } from "react";

export default function useFetch(url) {

const [data, setData] = useState([]);

useEffect(async () => {

const response = await fetch(url);

const data = await response.json();

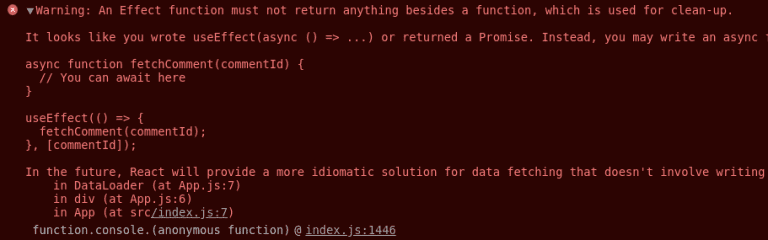
setData(data);

}, []);

return data;

}

Makes sense right? Then I opened the console and React was screaming at me:

[](https://www.valentinog.com/blog/static/e93fff71c0f696b3ff16e90f676f624e/e5715/react-hooks-async-await.png)

"Warning: An Effect function must not return anything besides a function, which is used for clean-up." Followed by a complete explanation of what I was doing wrong. How nice!

Turns out you cannot return a Promise from useEffect. JavaScript async functions always return a promise and useEffect should exclusively return another function, which is used for cleaning up the effect. That is, if you were to start setInterval in useEffect you would return a function (we have a closure there) for clearing up the interval.

So for making React happy we could rewrite our asynchronous logic like so:

// useFetch.js

import { useState, useEffect } from "react";

export default function useFetch(url) {

const [data, setData] = useState([]);

async function getData() {

const response = await fetch(url);

const data = await response.json();

setData(data);

}

useEffect(() => {

getData();

}, []);

return data;

}

and your custom hook will work again.

**React Hooks Tutorial for Beginners: wrapping up**

React hooks are a nice addition to the library. Born as an RFC in November 2018 they caught up quickly and landed in React 16.8. Think of React hooks as encapsulated states living outside your React components.

React hooks make render props and HOCs almost obsolete and provide a nicer ergonomics for sharing logic. With React hooks you can reuse common pieces of logic between React components.

React ships with a bunch of pre-defined hooks. The most important are useState and useEffect. useState makes possible to use local state inside React components, without resorting to ES6 classes.

useEffect replaces componentDidMount, componentDidUpdate, and componentWillUnmount providing a unified API. There are a lot of other hooks and I suggest reading through the official documentation for learning more.

It's easy to foresee where React is going: functional components all over the place! But even then we will have 3 ways for expressing components in React:

* functional components
* class components
* functional components with hooks

I can see a lot of convenience in hooks and I'm happy with the API they provide. It's amazing how React is evolving, the community seems to find always a clever way for solving harder problems.

**React Hooks Tutorial for Be**