React tutorial - Basics

Introduction

Lexicon

NAME	DESCRIPTION	USAGE
WEBPACK	A webpack is a module bundler	It is responsible for combining all your JavaScript into a single file that can be used by a web browser.
MODULES	Contains JavaScript code.	We split JavaScript programs into separate modules that can be imported when needed.
NPM	Node Package Manager	Registry that contains over 800,000 code packages free to download/use.
JSX	JavaScript Syntax Extension	Used to code html into JavaScript components.

Files tree structure

NAME	DESCRIPTION	NOTE
PACKAGE.JSON/	Where you install dependencies from npm.	React is listed as a dependency.
NODES_MODULES/	Source code for your dependencies.	
PUBLIC/	For static files and html shell for your website.	Your react app will be mounted in the div with root id.
SRC/	Where you will code your app.	
INDEX.JS	Initial entry point that tells the react app how to start up.	It bundles all the CSS with the final app code. Load your code into the div with root id.
APP.JS	First main component.	

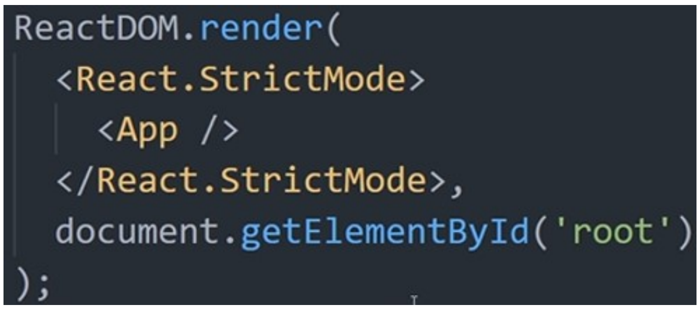
Behind the scene

React manages a webpack config.

React methods

Render

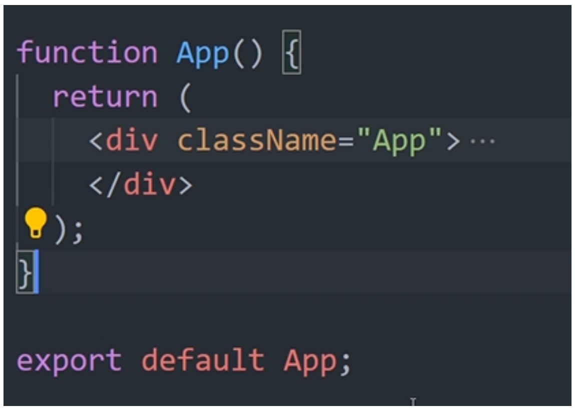
The StrictMode detects unsafe code automatically.



```
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

React components

Defines a function and export it as default.

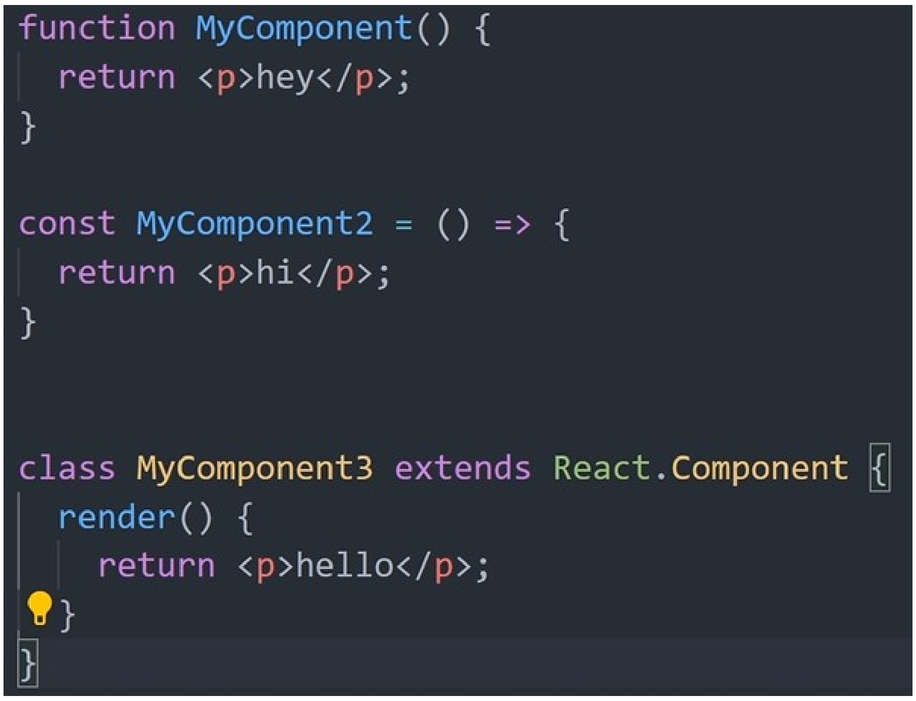


```
function App() {
  return (
    <div className="App">⋯
    </div>
  );
}

export default App;
```

Components

Reusable pieces of UI that are composed together as a tree to represent a complete frontend application.

It can be a function, a function expression or a class

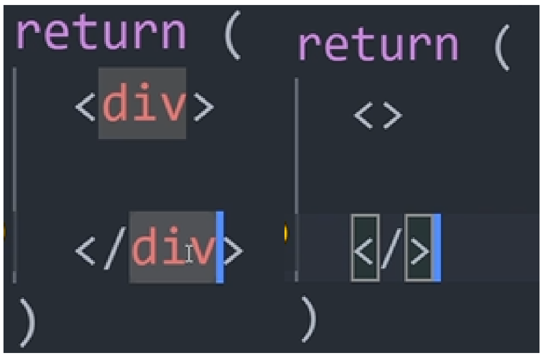


```
function MyComponent() {
  return <p>hey</p>;
}

const MyComponent2 = () => {
  return <p>hi</p>;
}


class MyComponent3 extends React.Component {
  render() {
    return <p>hello</p>;
  }
}
```

Functions

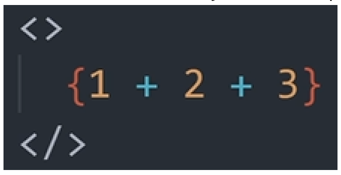
Return

- Returns JSX.
- Has to return a parent element (div here) or a fragment.



```
return (          return (
  <div>             <>

  </div>            </>
)                 )
```

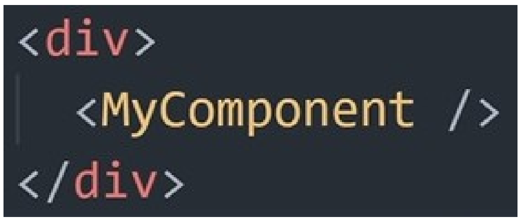
- You can write html inside the return function.
- You can write javascript inside the return function using braces.



```
<>
  {1 + 2 + 3}
</>
```

Usage

Once a component is defined, it can be declared or used in other parts of the UI just like regular html elements.



```
<div>
  <MyComponent />
</div>
```

Props

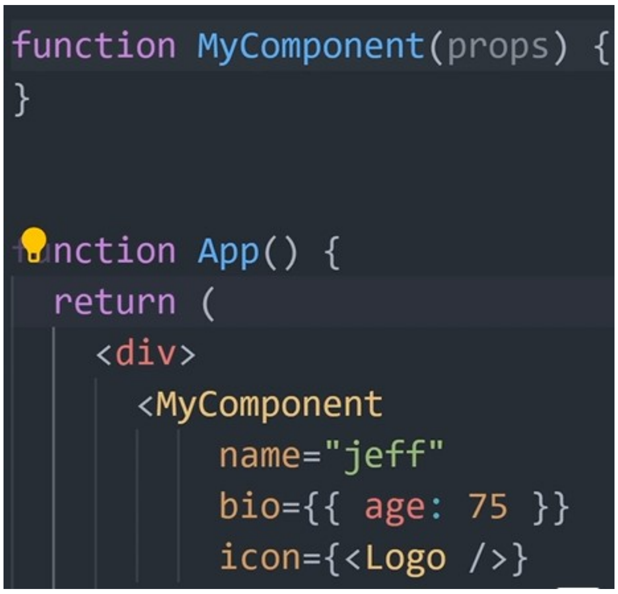
Props are immutble.

Used to pass data into the component from the parent.

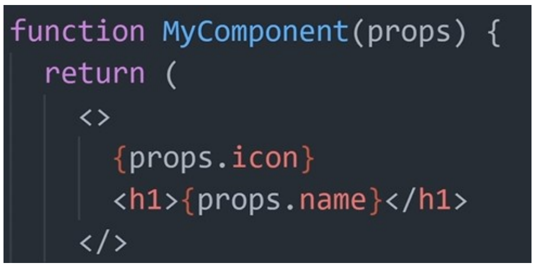
Every functional component has a prop argument that can accept external data.

A prop might be a value, an object or a component.

Components can pass props from parent to child but not from child to parent.

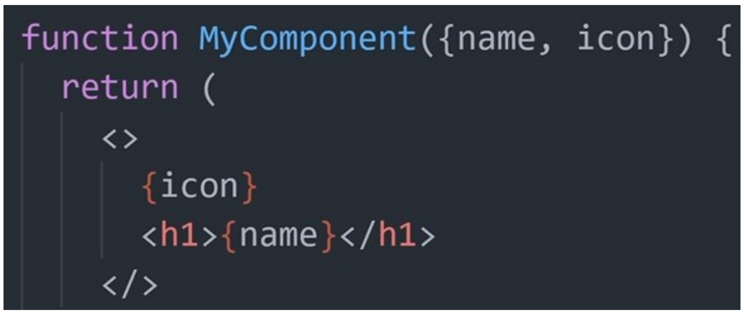


```
function MyComponent(props) {
}


function App() {
  return (
    <div>
      <MyComponent
          name="jeff"
          bio={{ age: 75 }}
          icon={<Logo />}
```



```
function MyComponent(props) {
  return (
    <>
      {props.icon}
      <h1>{props.name}</h1>
    </>
```

Destructuring props

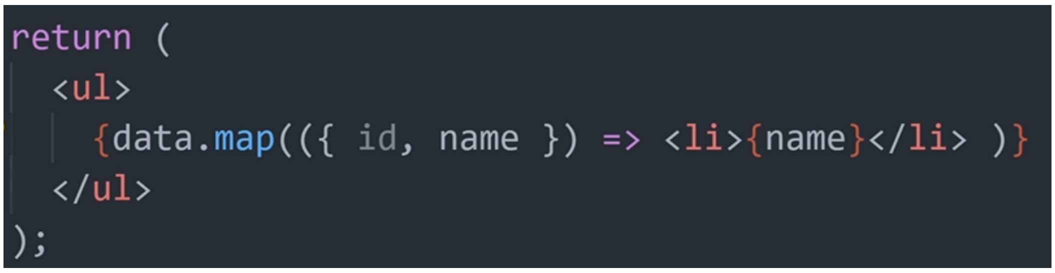


```
function MyComponent({name, icon}) {
  return (
    <>
      {icon}
      <h1>{name}</h1>
    </>
```

Virtual DOM

When the prop data changes, the library knows how to efficiently re-render each component using reconciliation using an internal mechanism called the Virtual DOM.

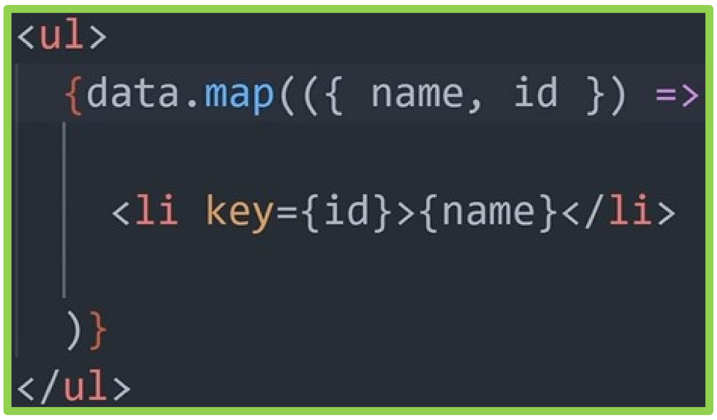
Loops & KEYS



```
return (
  <ul>
    {data.map(({ id, name }) => <li>{name}</li> )}
  </ul>
);
```

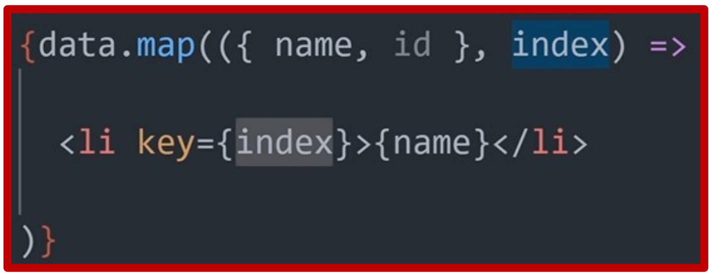
This will return an error because react needs a key for each unique item.

Providing a key allows react to keep track of each unique item, so that it can be re-rendered individually when the value in the array changes in the future.



```
<ul>
  {data.map(({ name, id }) =>

    <li key={id}>{name}</li>

  )}
</ul>
```

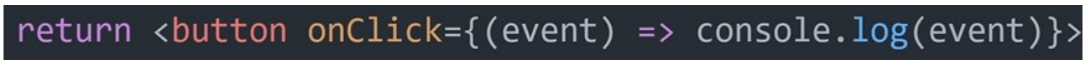
Don't use the index of the array map, because it doesn't allow react to optimize properly performance.



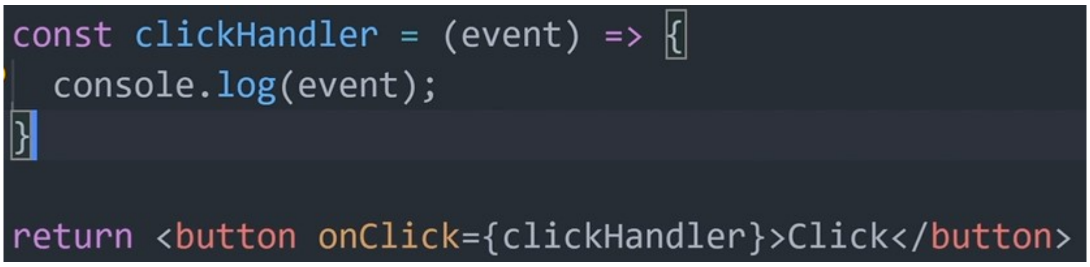
```
{data.map(({ name, id }, index) =>

  <li key={index}>{name}</li>

)}
```

Events

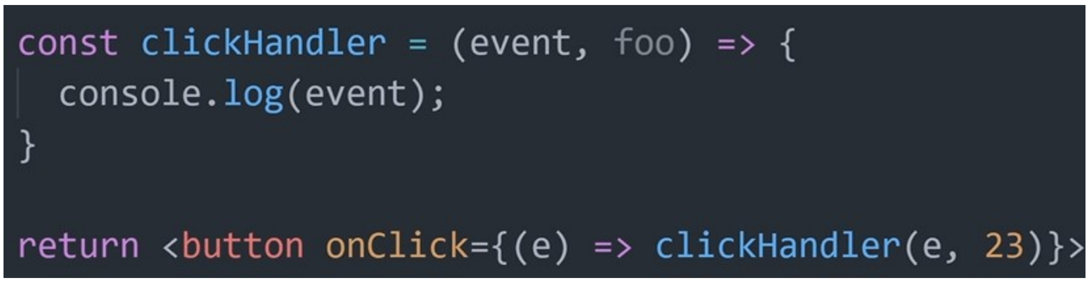
Here, (event) is the function that handles the onClick event.
The arrow then tells what the function does.



```
return <button onClick={(event) => console.log(event)}>
```

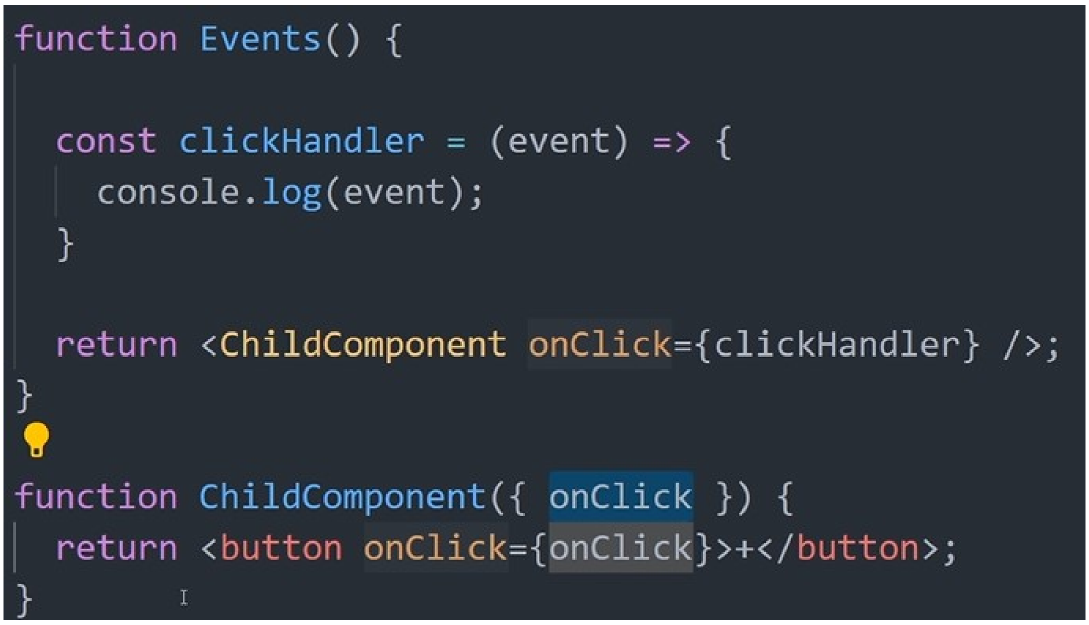


```
const clickHandler = (event) => {
  console.log(event);
}

return <button onClick={clickHandler}>Click</button>
```



```
const clickHandler = (event, foo) => {
  console.log(event);
}

return <button onClick={(e) => clickHandler(e, 23)}>
```

To reuse code or a handler method in other components, we can pass our event handler function as a prop to a child component.

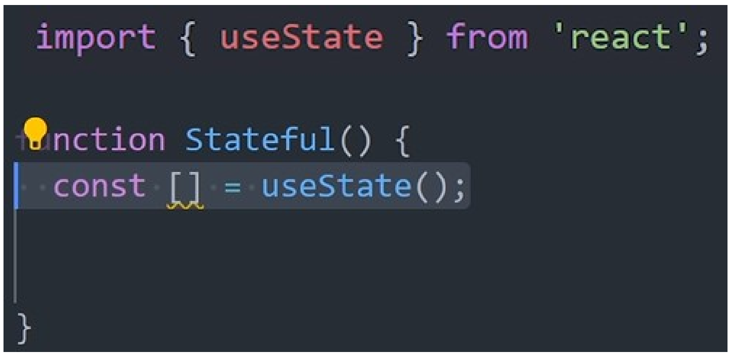


```
function Events() {

  const clickHandler = (event) => {
    console.log(event);
  }

  return <ChildComponent onClick={clickHandler} />;
}

function ChildComponent({ onClick }) {
  return <button onClick={onClick}>+</button>;
}
```

State

We use props to share data between components, but how do we actually modify that data ?

React provide a hook called useState.
A hook is a function that can be called in the top level of your component to use different features of the framework.



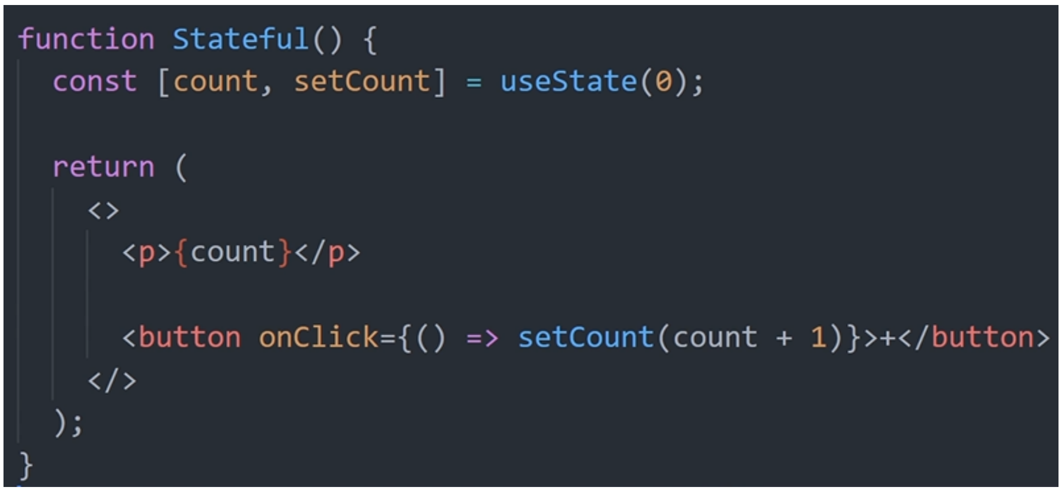
```
import { useState } from 'react';

 nction Stateful() {
  const [] = useState();


}
```

useState

Creates a statefull value, and whenever it changes it will automatically re-render any component that depends on it.

It takes a default value as an argument and return 2 values in an array.

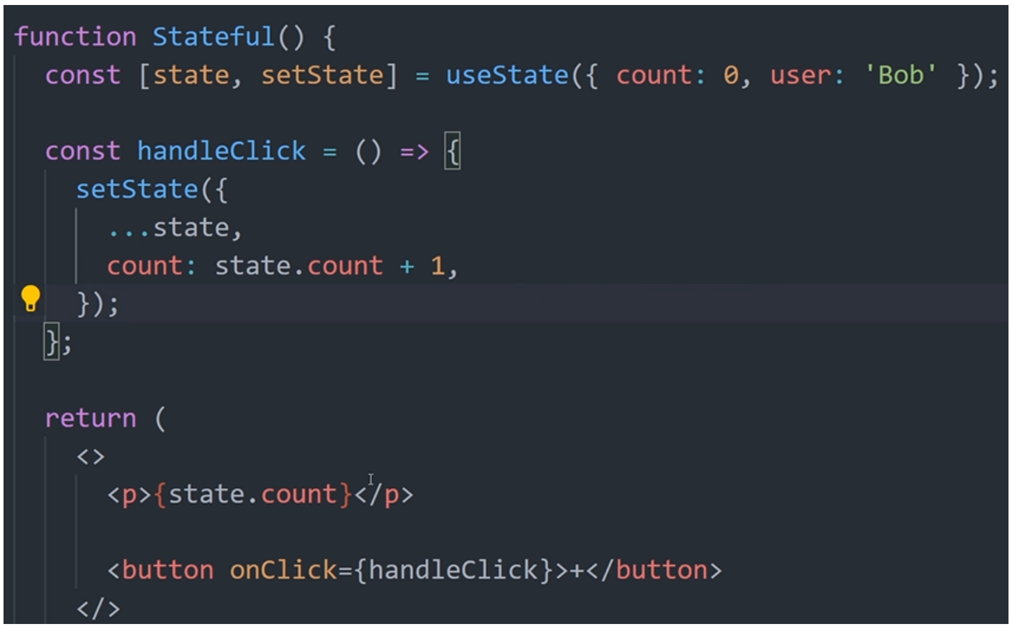
It is common to destructure these values,
with the first value to be the value you want to use in the UI,
and the second value to be a function that you can use to render a different value.



```
function Stateful() {
  const [count, setCount] = useState(0);

  return (
    <>
      <p>{count}</p>

      <button onClick={() => setCount(count + 1)}>+</button>
    </>
  );
}
```

When you work with an array or object with useState, when you set a new value it will completely override the old one.

If you want to merge a new array into an array or object, It's common to use the spread syntax, to merge the old value into a new object.

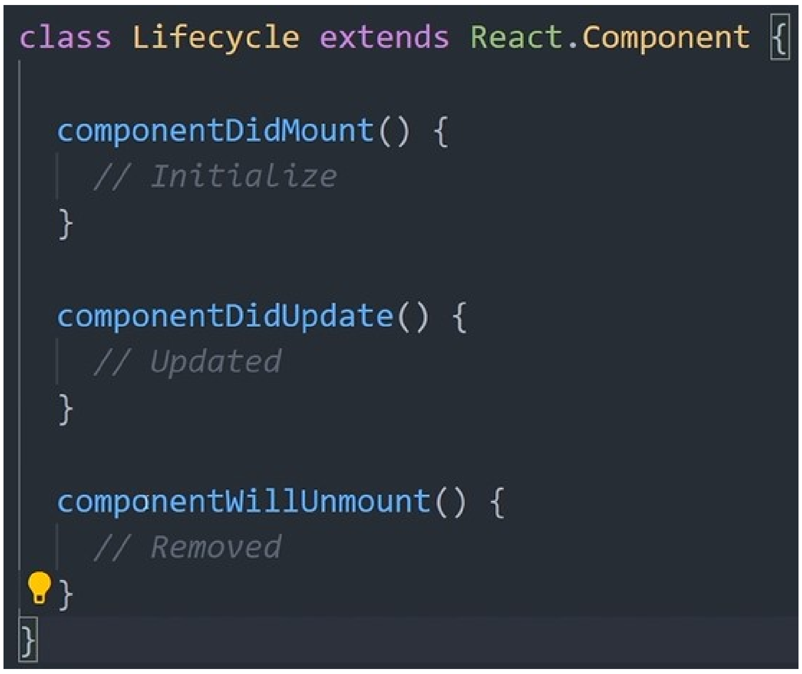


```
function Stateful() {
  const [state, setState] = useState({ count: 0, user: 'Bob' });

  const handleClick = () => {
    setState({
      ...state,
      count: state.count + 1,
    });
  };

  return (
    <>
      <p>{state.count}</p>

      <button onClick={handleClick}>+</button>
    </>
```

Lifecycle & Effects

Firstly: component is mounted (created)
Secondly: component might be updated (state changes)
Thirdly: component might be unmounted (removed)

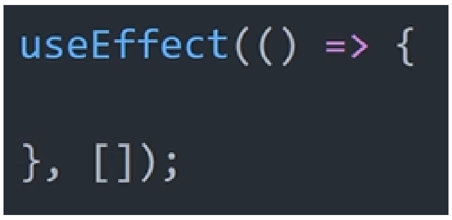
When using class components, we can use different methods to tap in those lifecycle (for example, we initialize a connection to a DB then destroy the component when the data was fetched):



```
class Lifecycle extends React.Component {

  componentDidMount() {
    // Initialize
  }

  componentDidUpdate() {
    // Updated
  }

  componentWillUnmount() {
    // Removed
  }
}
```

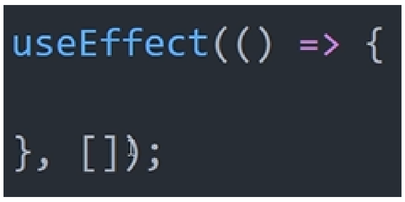
Function components don't have access to these lifecycle methods, instead it uses the hook useEffect.

First argument is the code you want to run and second argument is when you want that code to run.



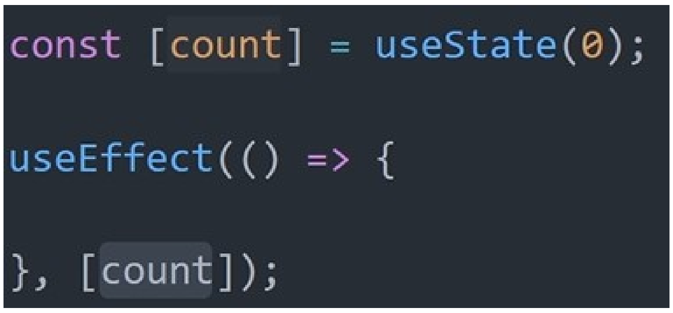
```
useEffect(() => {

}, []);
```

Runs whenever the component is first initialized (equivalent to componentDidMount).



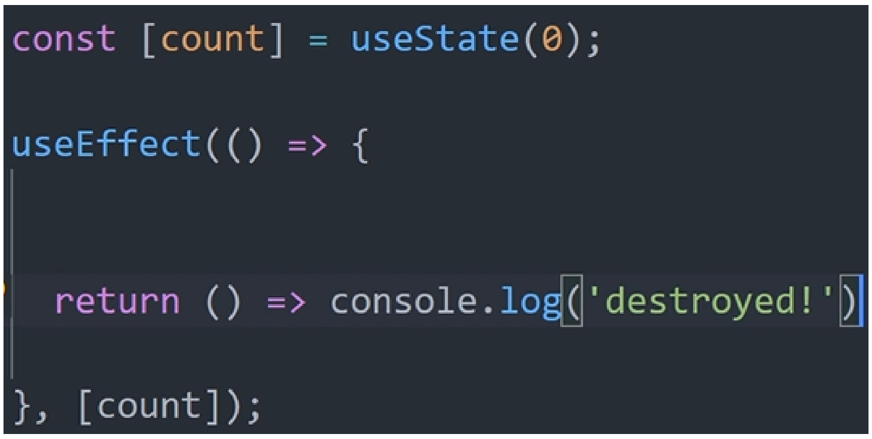
```
useEffect(() => {

}, []);
```

Runs whenever the data changes (equivalent to componentDidUpdate).



```
const [count] = useState(0);

useEffect(() => {

}, [count]);
```

Normally the useEffect function doesn't have a return value. However, if you return another function, that function will run when the component is destroyed.

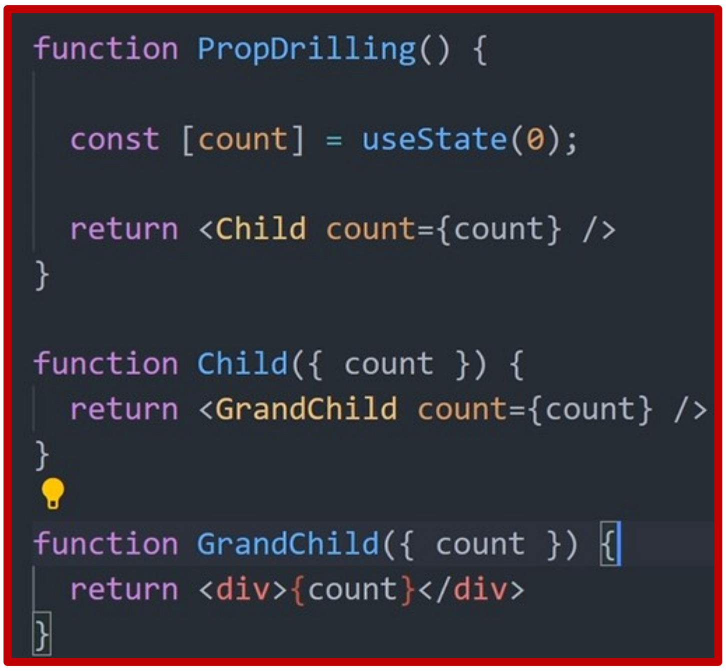


```
const [count] = useState(0);

useEffect(() => {


  return () => console.log('destroyed!')

}, [count]);
```

Context

Share reactive data anywhere in the component tree.

This is called prop drilling and is not used.



```
function PropDrilling() {

  const [count] = useState(0);

  return <Child count={count} />
}

function Child({ count }) {
  return <GrandChild count={count} />
}

function GrandChild({ count }) {
  return <div>{count}</div>
}
```

Context API allows you to place data somewhere in the component tree, then all children can have access to those datas without access it with a prop.

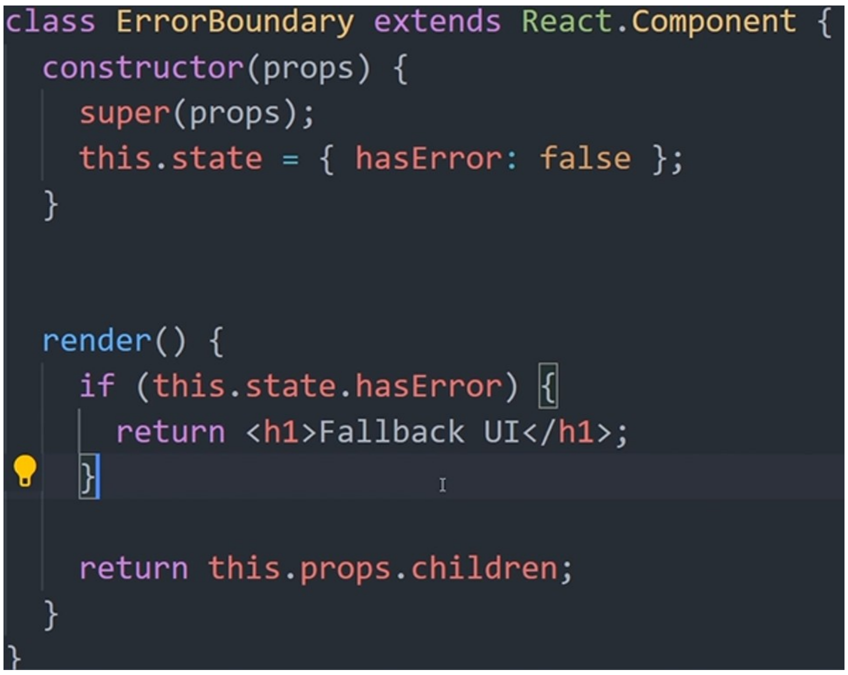
We use the createContext function with the data we want to share (CountContext here).

https://reactjs.org/docs/context.html#when-to-use-context

Error Boundaries

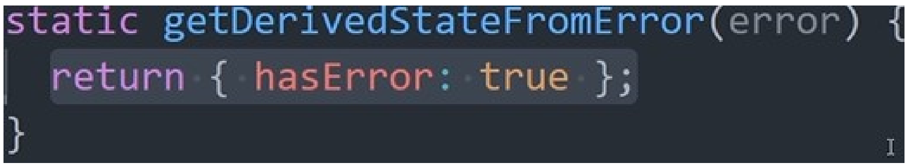
If a component fails we can catch the error and return something about it, it prevents the application to crash.

We'll create an ErrorBoundary class which will detect if the state of the component has an error or not.



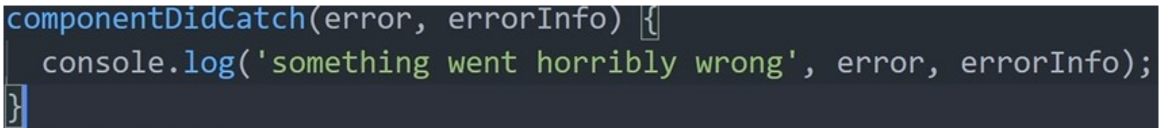
```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }


  render() {
    if (this.state.hasError) {
      return <h1>Fallback UI</h1>;
    }

    return this.props.children;
  }
}
```

To detect an error and update the state, we implement the getDerivedStateFromError method:



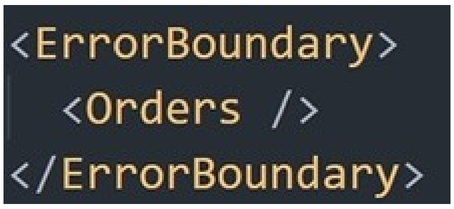
```
static getDerivedStateFromError(error) {
  return { hasError: true };
}
```

We can also run a side effect when the error occurs:



```
componentDidCatch(error, errorInfo) {
  console.log('something went horribly wrong', error, errorInfo);
}
```

Now we can implement our ErrorBoundary component wherever we want and it will catch errors instead of crashing our app.



```
<ErrorBoundary>
  <Orders />
</ErrorBoundary>
```

Suspense & Lazy

Suspense lets components “wait” for something before rendering.

Lazy lets you define a component that is loaded dynamically (when needed).

**Suspense only supports one use case: loading components dynamically with React.lazy. In the future, it will support other use cases like data fetching.*

Lazy, imports a component only when we need to use that component.

```
const SomeComponent = React.lazy(() => import('./SomeComponent'));
```

With React Suspense, you can declare that a component may "suspend" while it is waiting for data to be loaded. Here while OtherComponent is loading we’re displaying the spinner component.

```
// Displays <Spinner> until OtherComponent loads
<React.Suspense fallback={<Spinner />}>
  <div>
    <OtherComponent />
  </div>
</React.Suspense>
```

Routes

Used to navigate through your website. Consider looking at the S&L chapter.

```
import React, { Suspense, lazy } from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </Suspense>
  </Router>
);
```

Animations

https://www.framer.com/motion/

Good practices

Components

Export one component per file.

Use function components and not class components.

Starts with a Capital letter.

Keys

Loops & KEYS

Commands

Create

```
$> npx create-react-app
```

Start

```
$> npm start
```

Build

Located in build/static/js/main.xxxx.js

The random number is so each new build can invalidate anything that has been cached in the browser.

```
$> npm run build
```

Extras

- Static site : Gatsby
- Server-side rendering : next.js
- Animations : React-spring
- Forms : Formik
- State management: redux, flux, mobx, recoil, xstate ...
- Mobile : React native

Made by :

Matthias Brat

Socials:

github.com/matthiasbrat

stackoverflow.com/users/17921879/matthias-b

matthiasbrat.dev