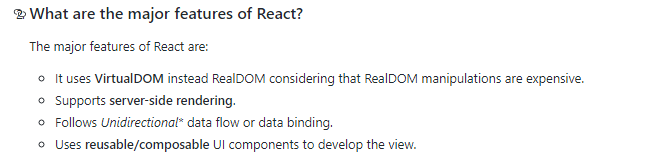
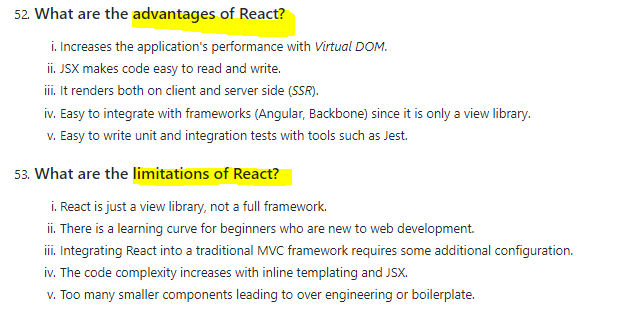
React at its core is just a component creation library

# **React**

[React](https://facebook.github.io/react/) is a JavaScript library for building user interfaces. It is the view layer for web applications. It lets you compose complex UIs from small and isolated pieces of code called “components”.

At the heart of all React applications are **components**. A component is a self-contained module that renders some output.





**JSX**

**JSX is a XML-like syntax extension to ECMAScript (the acronym stands for JavaScript XML). Basically it just provides syntactic sugar for the React.createElement() function, giving us expressiveness of JavaScript along with HTML like template syntax.**

Fundamentally, JSX just provides syntactic sugar for the React.createElement(component, props, ...children) function. The JSX code:

<MyButton color="blue" shadowSize={2}>

Click Me

</MyButton>

compiles into:

React.createElement(

MyButton,

{color: 'blue', shadowSize: 2},

'Click Me'

)

You can also use the self-closing form of the tag if there are no children. So:

<div className="sidebar" />

compiles into:

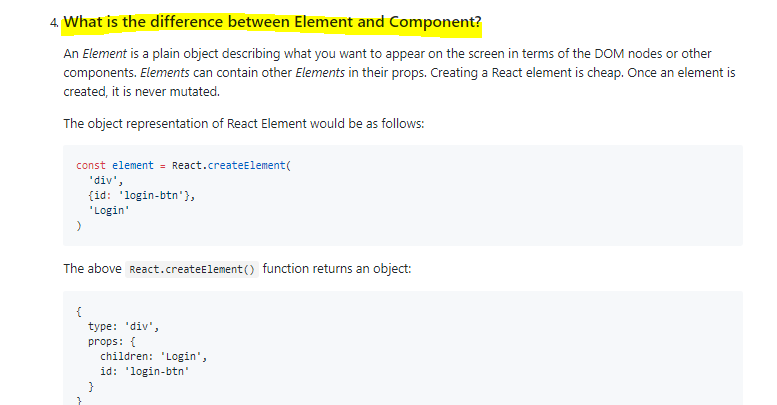
React.createElement(

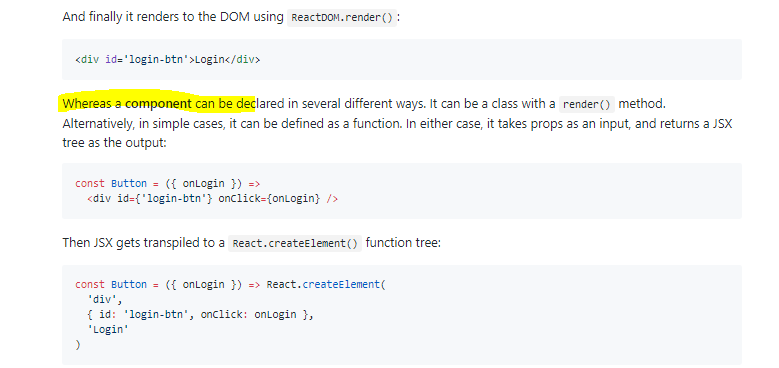
'div',

{className: 'sidebar'},

null

)

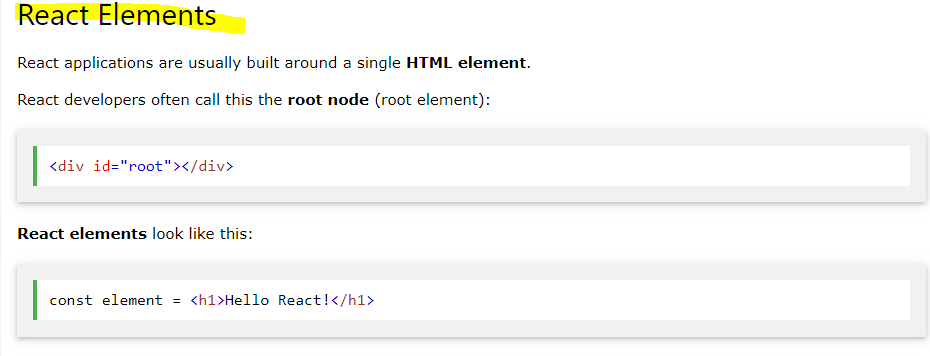


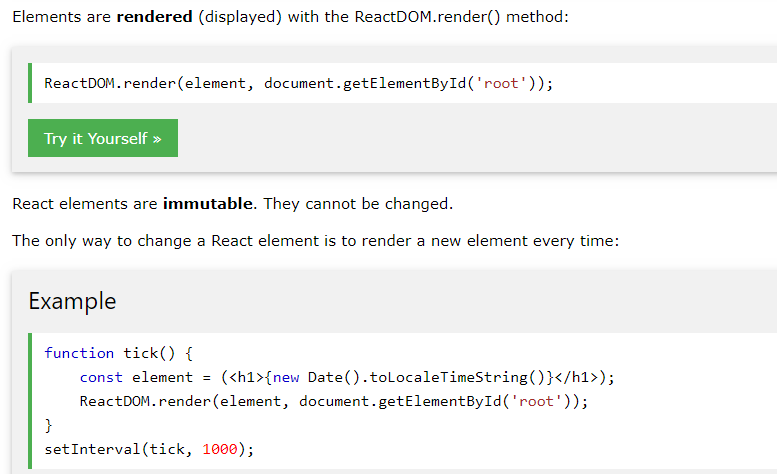


Just like HTML, JSX tags can have a tag names, attributes, and children. If an attribute is wrapped in curly braces, the value is a JavaScript expression.

Note that JSX does not use quotes around the HTML text string.

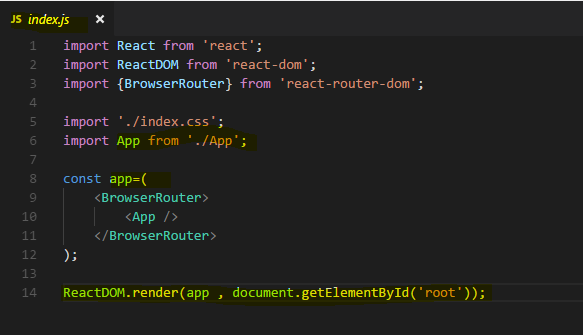








**For example entry point of react application which is served by server**

**And here’s the root component ‘App’ getting render to id=root.** 

**This index.js file is the entry point for bundler/builder(like webpack) for making a ‘bundle.js’ file out of whole react application and attaching the bundle.js file to index.html**

**Typically, you render one root component, the app component but you can name it whatever you want. One root component, in our case it is the app component named app, and in there you would nest all the other components your application might need and of course these components can then also be nested into each other but all the way up to the top, you only have one root component.**

**Babel is a preprocessor.**

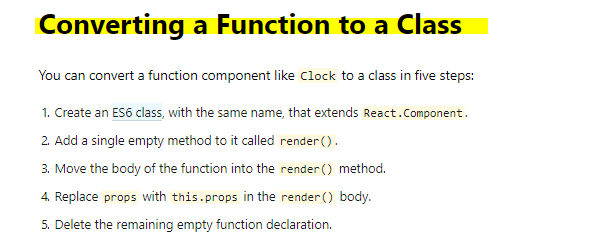


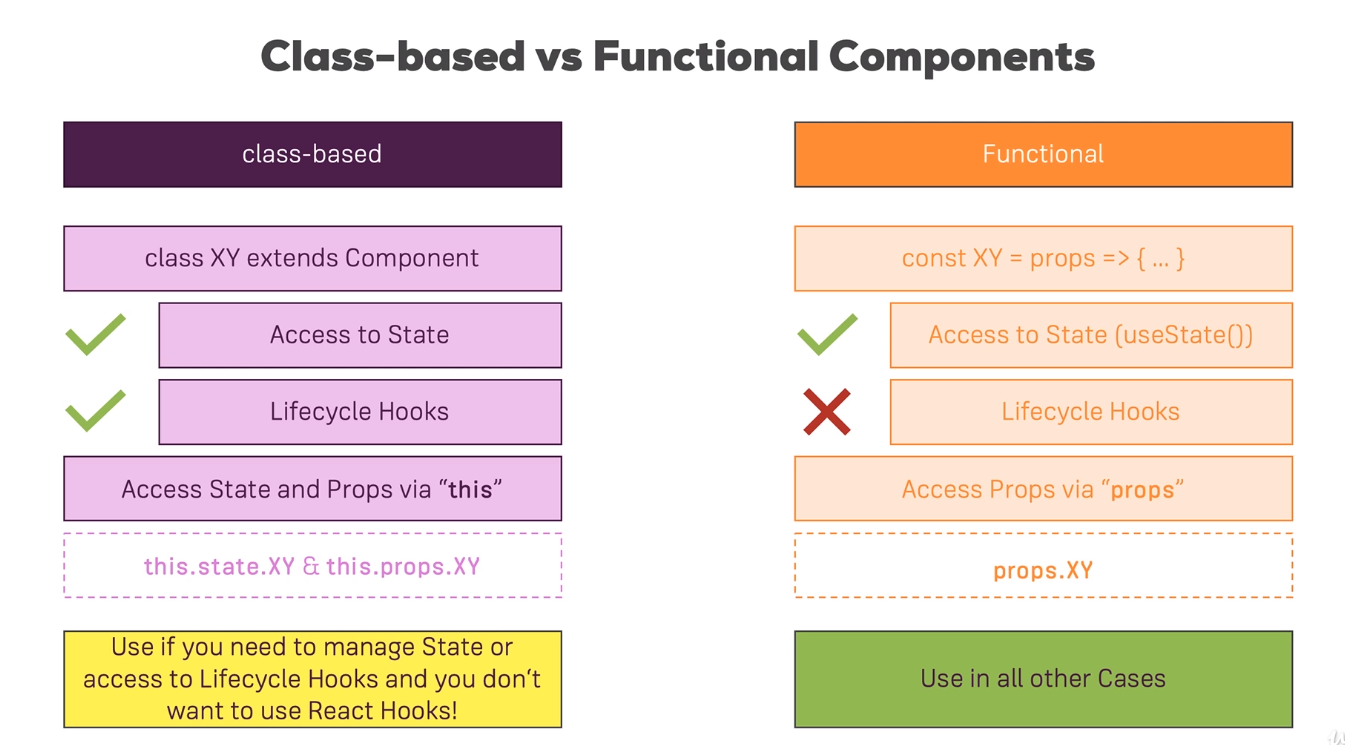
[**Why does React require Babel and Webpack to work?**](https://stackoverflow.com/questions/43175140/why-does-react-require-babel-and-webpack-to-work)

React doesn't "need" babel or webpack but the library is built on the concept of using ES6 javascript syntax and JSX (essentially HTML in JS).

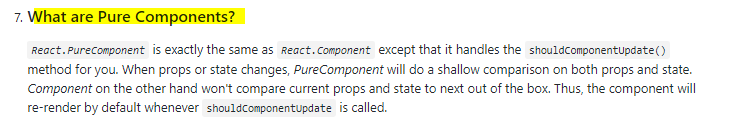
React however can be used without [ES6](https://facebook.github.io/react/docs/react-without-es6.html) and [JSX](https://facebook.github.io/react/docs/react-without-jsx.html) which would remove the initial need for Babel but you would lose the potential benefits of ES6 and JSX.

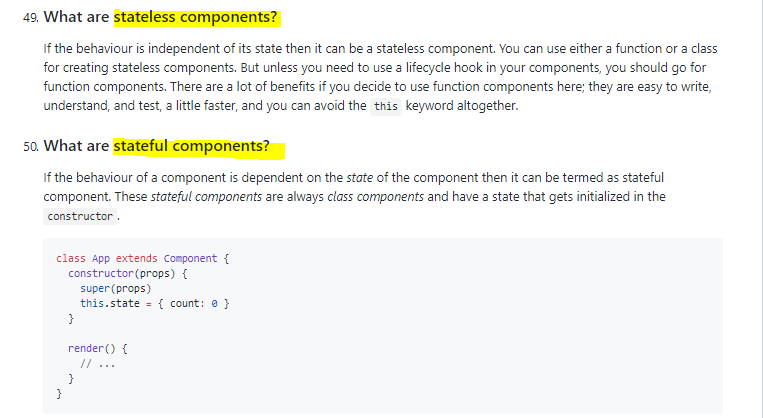
## **Component**

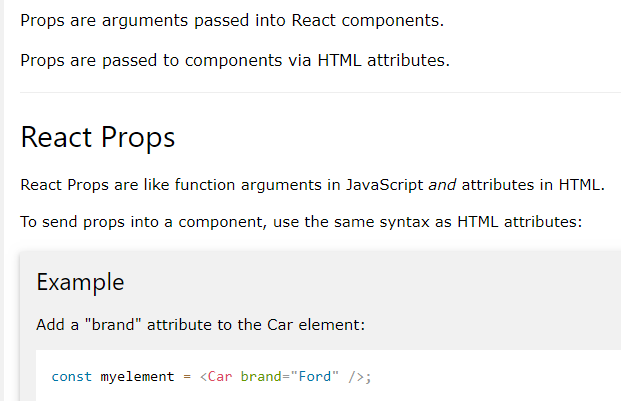


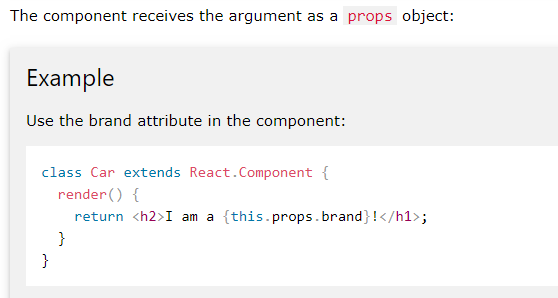


**presentational component?** It is a functional component that does not manage state

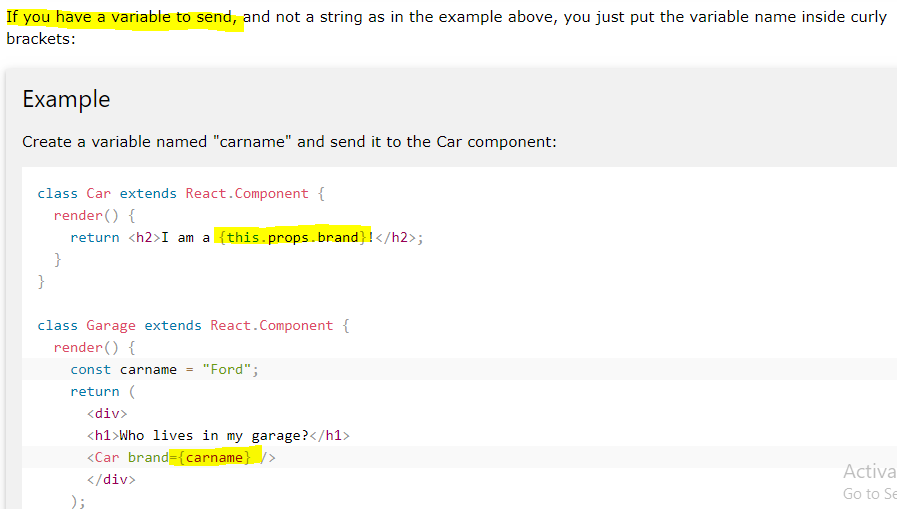




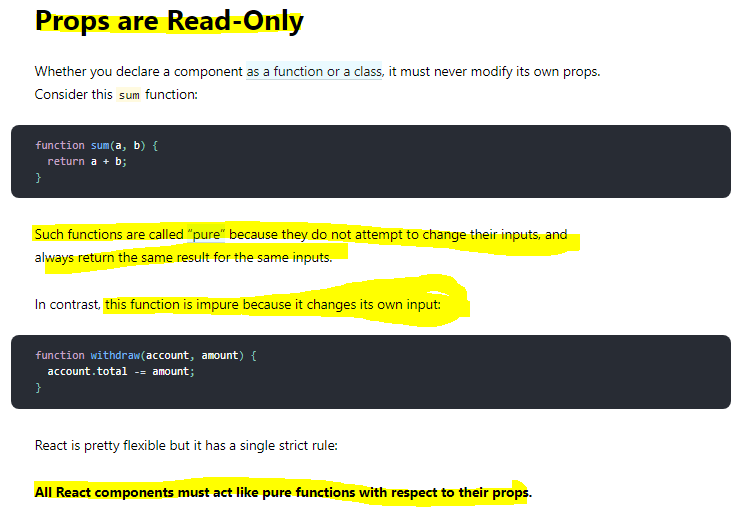
**React Props**

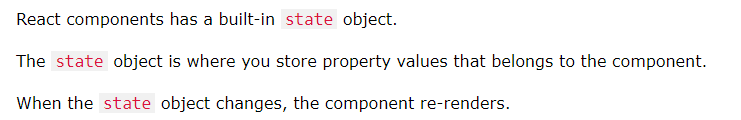


**For functional component it would be props.brand not this.props.brand.**

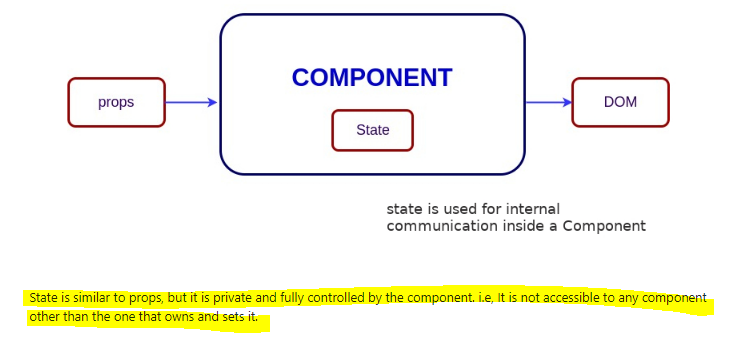




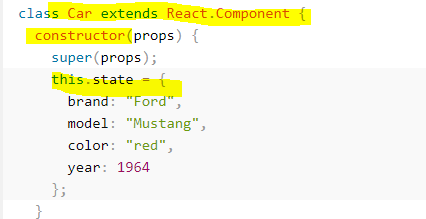


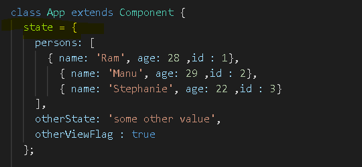
**React State**

State of a component is an object that holds some information that may change over the lifetime of the component. We should always try to make our state as simple as possible and minimize the number of stateful components.

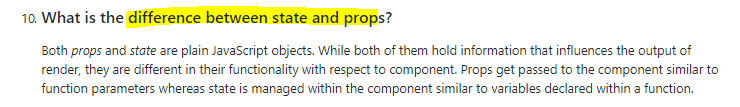


**There are two ways we can initialize or define state object-**

The state object is initialized in the constructor: 

**Directly use state object outside constructor -** 

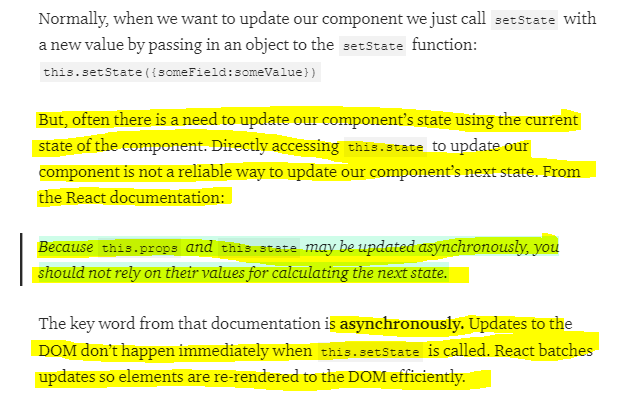
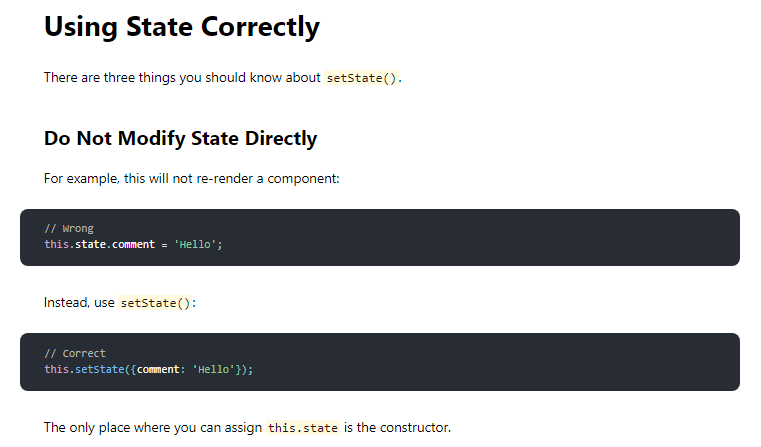


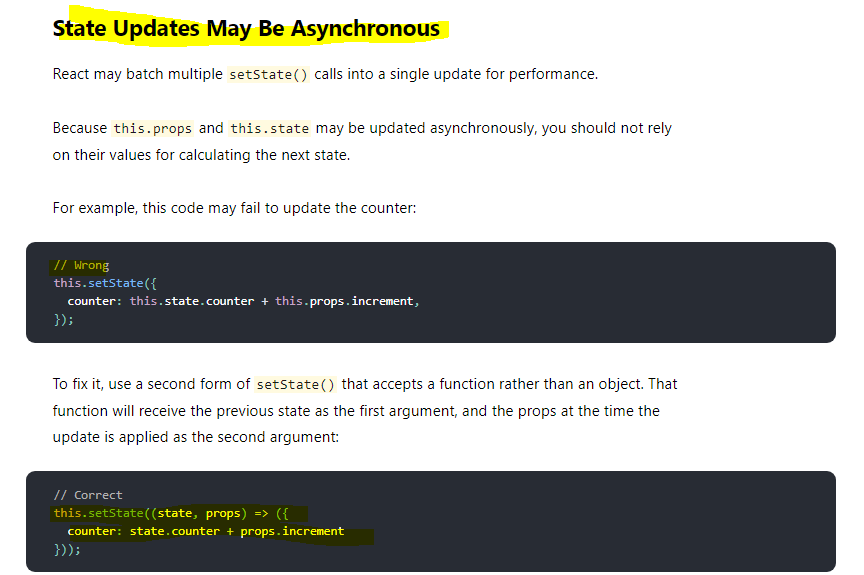


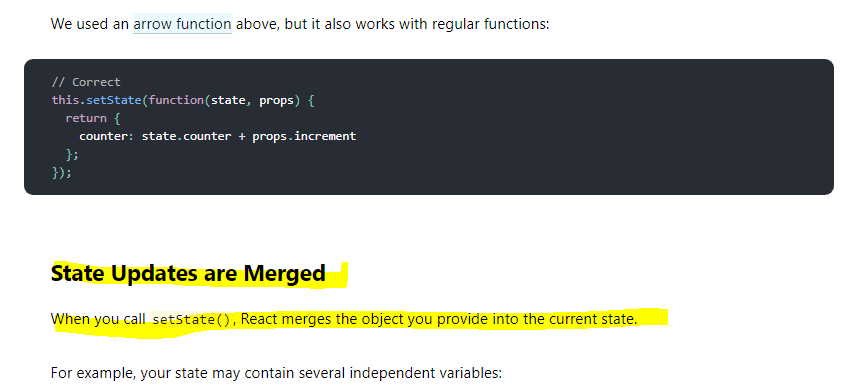
### **setState - Setting the State**

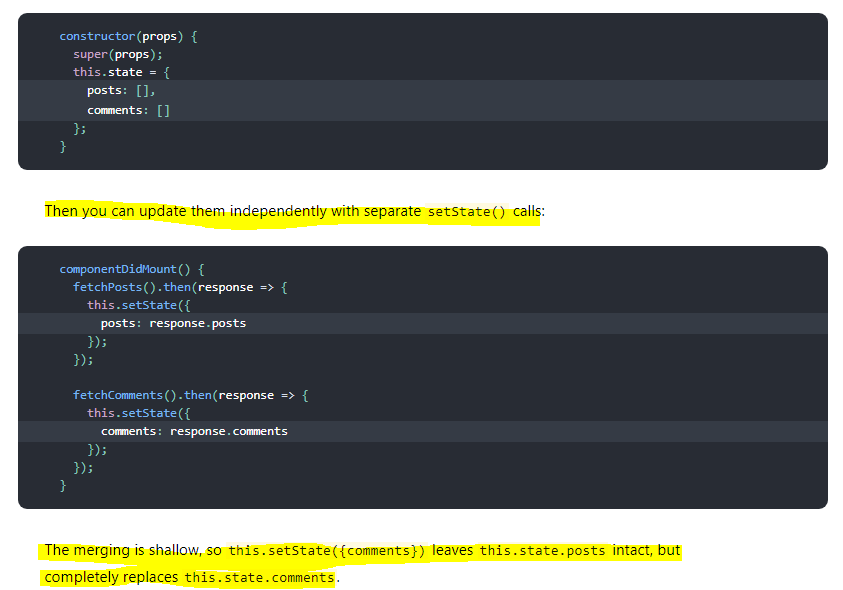
Always use the setState() method to change the state object, it will ensure that the component knows its been updated and calls the render() method (and all the other lifecycle methods).

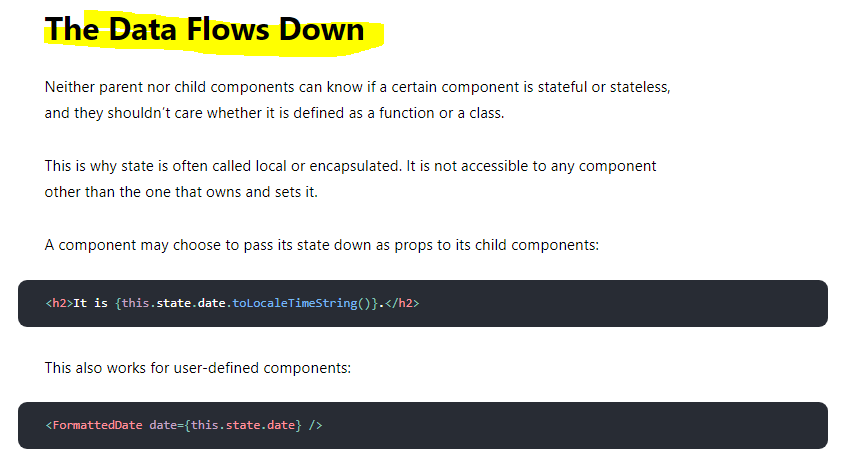


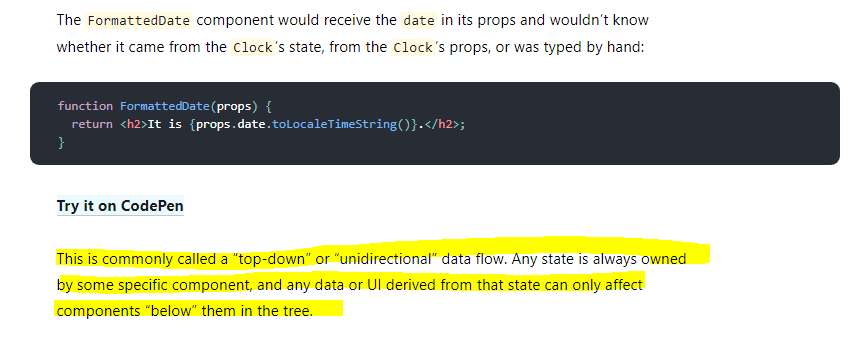
**Setting state correctly** 

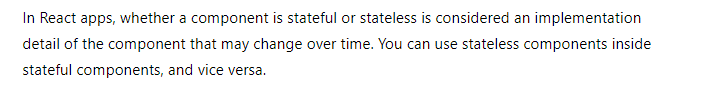


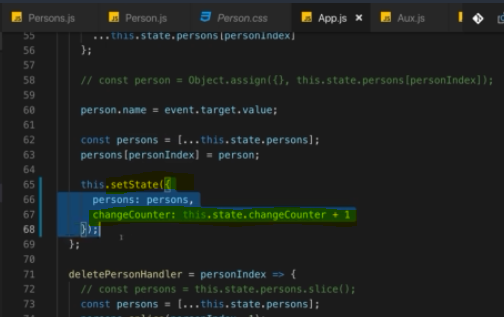












We use setState to update their state.

Now the good news is we're using set state correctly here, bad news is you can use it incorrectly but I will show you how and how to avoid it.

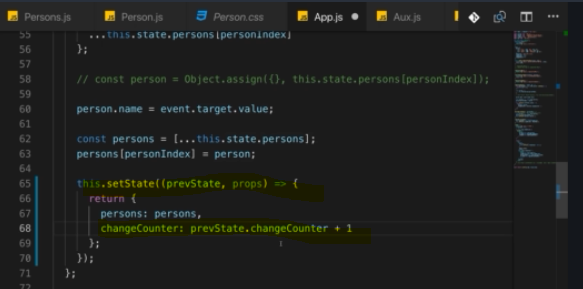
now of course this above code always depends on the old state, right?

Behind the scenes, set state does not immediately triggeran update of the state of this component in a re-render cycle, instead it's basically scheduled by React and React will then perform the state update and the re-render cycle when it has the available resources to do that.

You call set state synchronously here but it's not guaranteed to execute and finish immediately and therefore, this state when used for a state update is not guaranteed to be the latest state or the previous state on which you depend, it could be an older state.

**Set state does not only take a Javascript object, it also works when you pass in a function, so you can use either syntax.**

optional syntax where you actually receive two arguments and I'm using an anonymous arrow function here, where the first argument is your old state and I'll name it prev state for previous state and the second argument are your current props in case you need those. Then you have your function body and in that function body, you should return this new state object, like this. Now of course there, you are now not referring to this state but you can refer to previous state change counter and here, React guarantees you that this will be the actual previous state as you would expect it for this set state update.

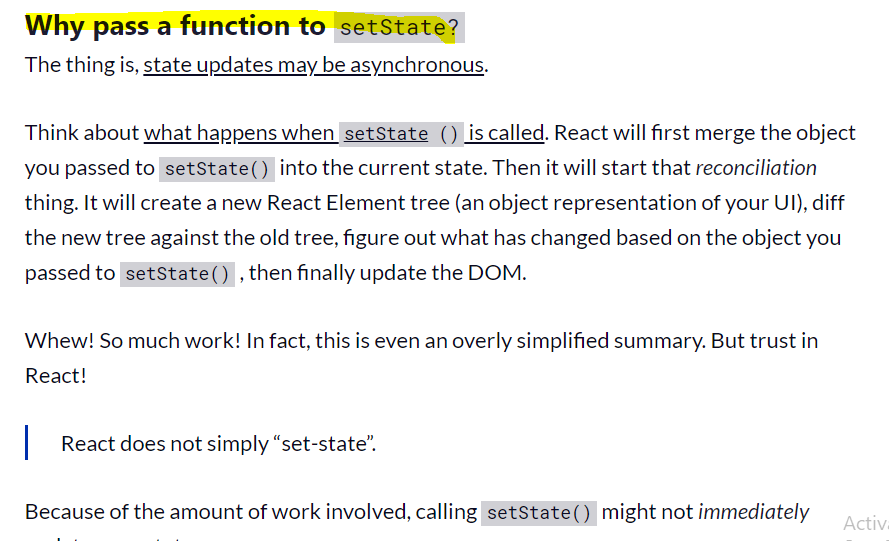


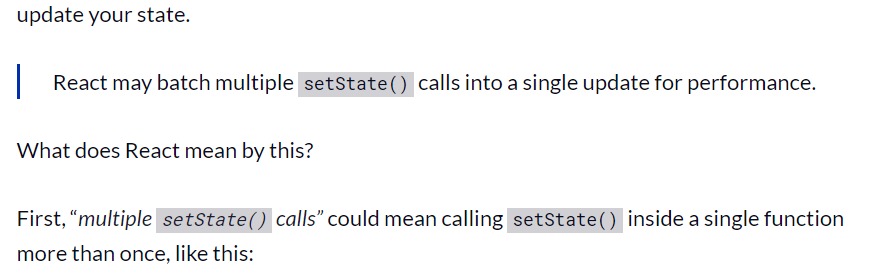
**this is the better way and the recommended way of updating the state when you're depending on old state. Super important to keep that in mind, it's an important pattern, not an optional solution but really the best practice for state updates that depend on the old state.**

**If you know you’re going to use setState to update your component and you know you’re going to need the current state or the current props of your component to calculate the next state, passing in a function as the first parameter of this.setState instead of an object is the recommended solution.**

#### **Another very good explanation**

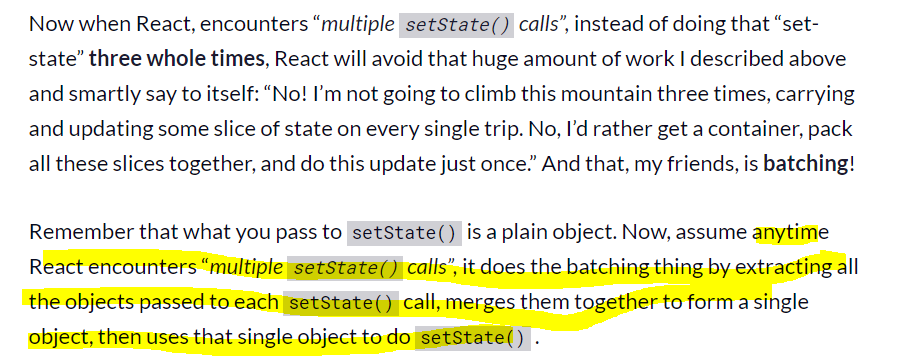
<https://www.freecodecamp.org/news/functional-setstate-is-the-future-of-react-374f30401b6b/>



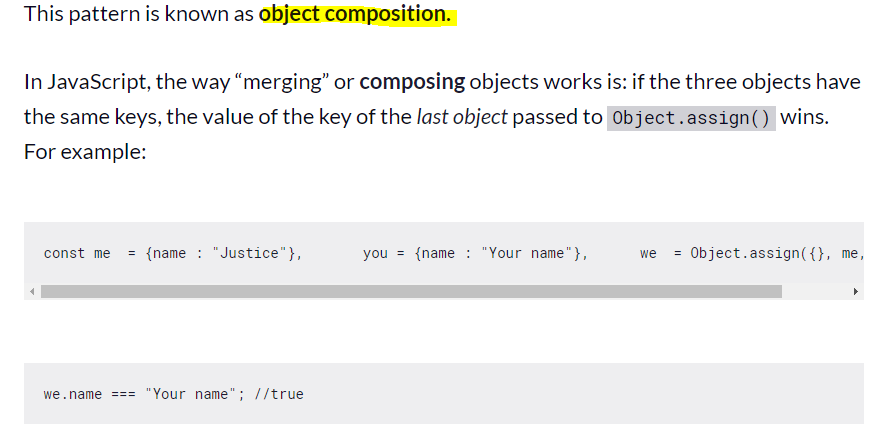


// multiple setState()

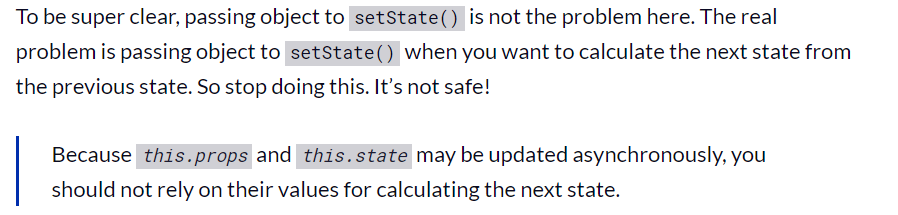
callsincreaseScoreBy3 () { this.setState({score : this.state.score + 1}); this.setState({score : this.state.score + 1}); this.setState({score : this.state.score + 1});}



const singleObject = Object.assign( {}, objectFromSetState1, objectFromSetState2, objectFromSetState3);

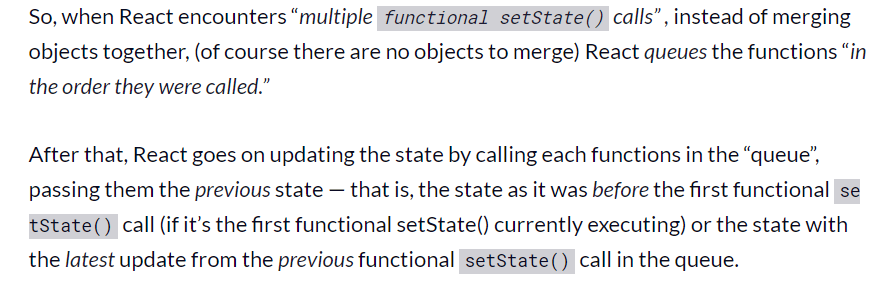


Thus, if you call setState() with an object multiple times — passing an object each time — React will **merge**. Or in other words, it will **compose** a new object out of the multiple objects we passed it. And if any of the objects contains the same key, the value of the key of the last object with same key is stored.



**When you do functional setState…**

**Updates will be queued and later executed in the order they were called.**



**// multiple functional setState call**

**increaseScoreBy3 () { this.setState( (state) => ({score : state.score + 1}) ), this.setState( (state) => ({score : state.score + 1}) ), this.setState( (state) => ({score : state.score + 1}) ) }}**

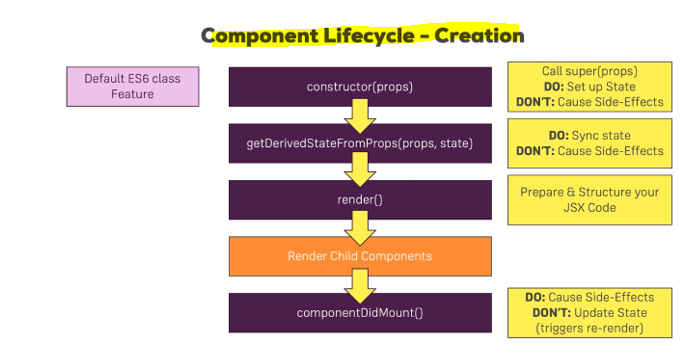
### **Component Lifecycle hooks/methods(only available in Class-based components)**

**it's only available in class-based components. For functional component handling lifecycle of component we have react hooks.**

We've got these methods which we can add to any class-based component and React will execute them for us and they will run at different points of time and we can do different things with them.

for example

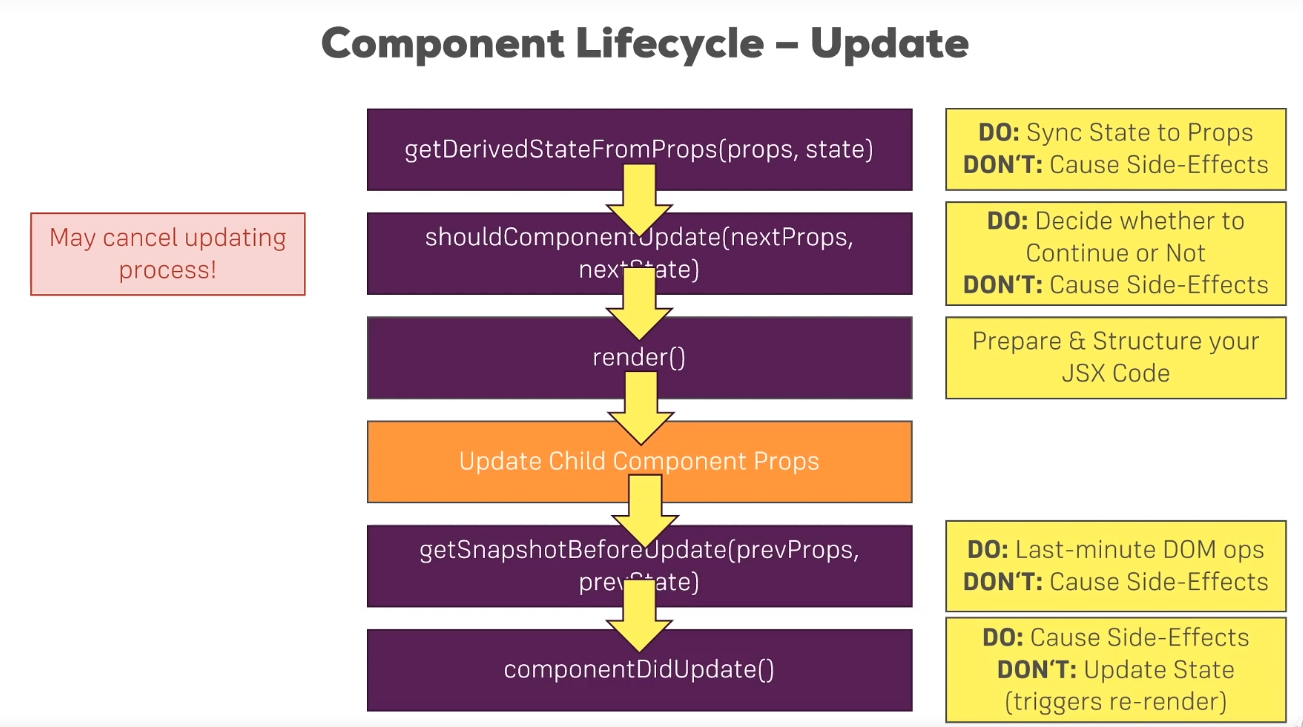
fetch data from the web or to do some cleanup work before a component is removed from the DOM

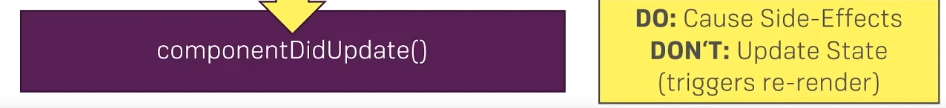




**here, you can cause side effects. That is a typical hook you would use for making an HTTP request to get new data from the web.**

**What you shouldn't do in here is update the state, so don't call set state in here unless it's in, let's say the then block of a promise after you sent an HTTP request but don't call set state in here synchronously. So** **you can definitely set up some code that executes in the future which then updates the state, for example when the response from the server is back but don't do it right away when componentDidMount runs that you immediately call set state because that will trigger a re-render cycle and that is bad for performance.**



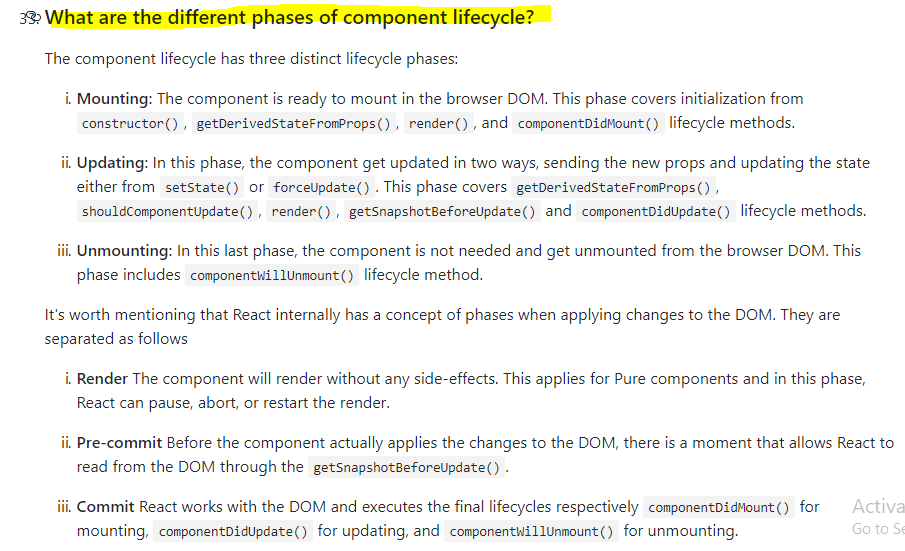


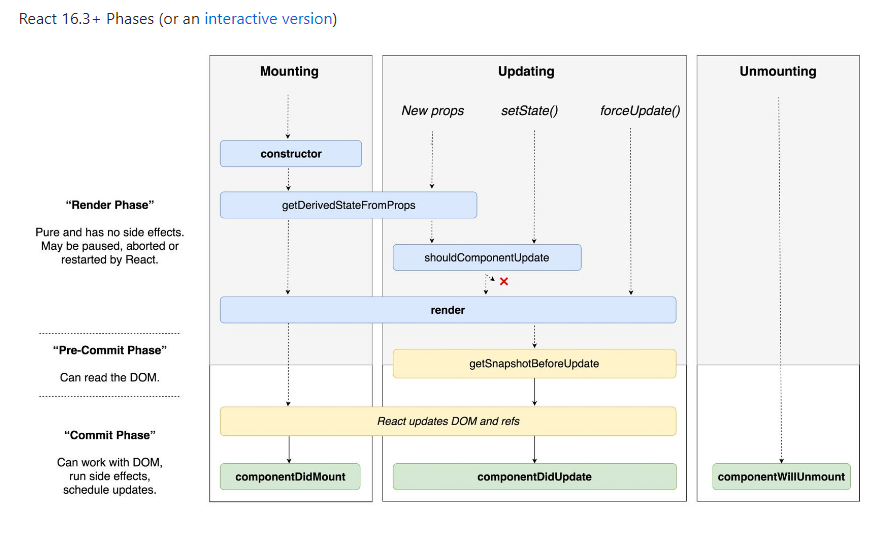
once we're done with the update, componentDidUpdate is called. A lifecycle hook that signals that you are now done with the updating, that the render method has been executed

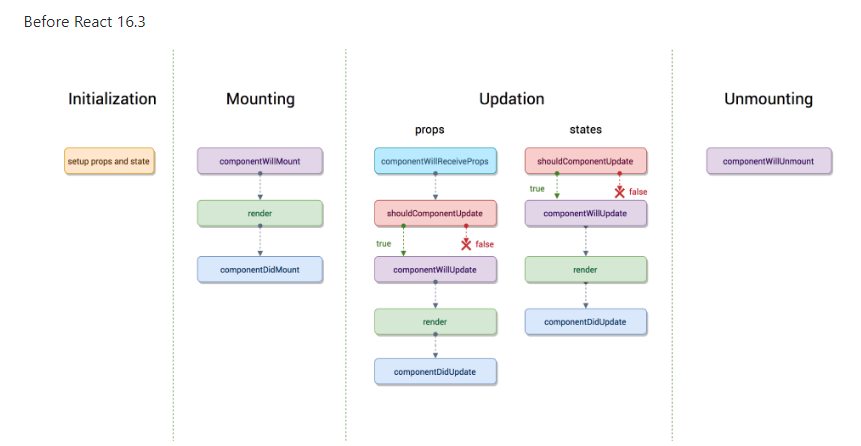
and here you can now cause side effects, so here you could now make an HTTP request, though you'll have to watch out to not enter an infinite loop here

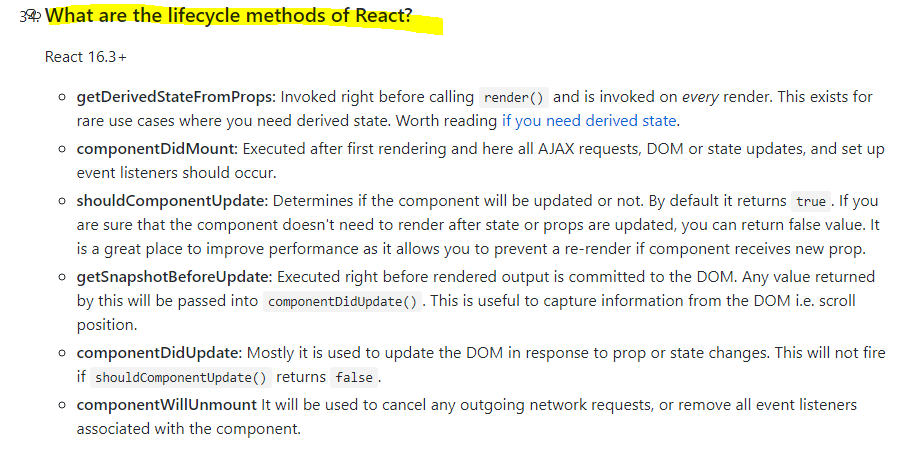
if you make an HTTP request and you get back a response and you then update your component and then this cycle starts .this is a typical problem you'll be facing. Now what you shouldn't do here outside of the, let's say then block of a promise of an HTTP request is updating the state with set state.

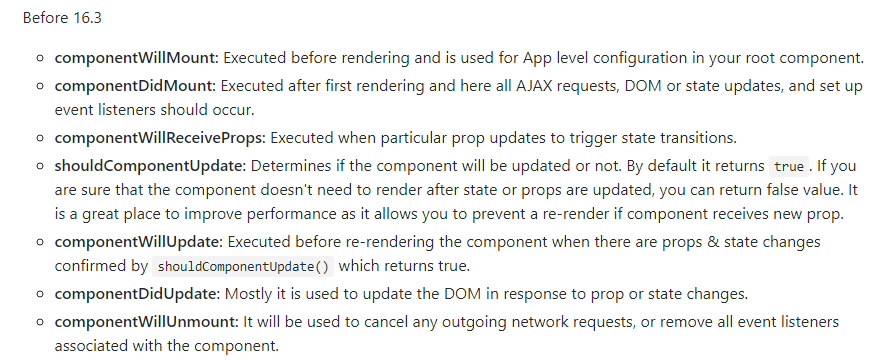
It's fine to do it as a result of some async task you're kicking off here but you should not call it synchronously in componentDidUpdate because that will simply lead to an unnecessary re-render cycle.





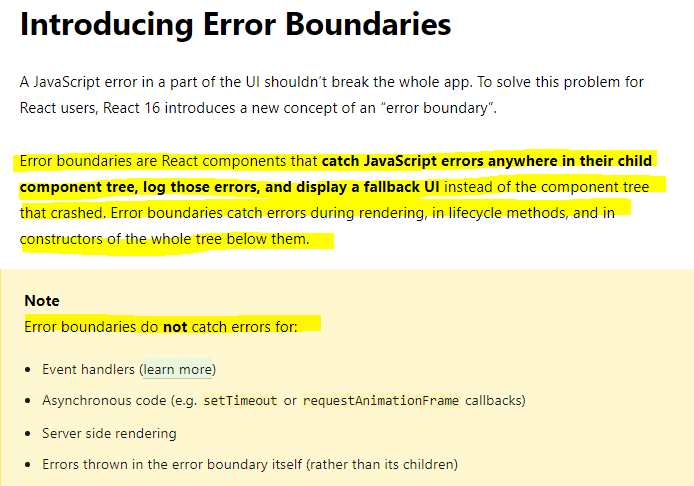






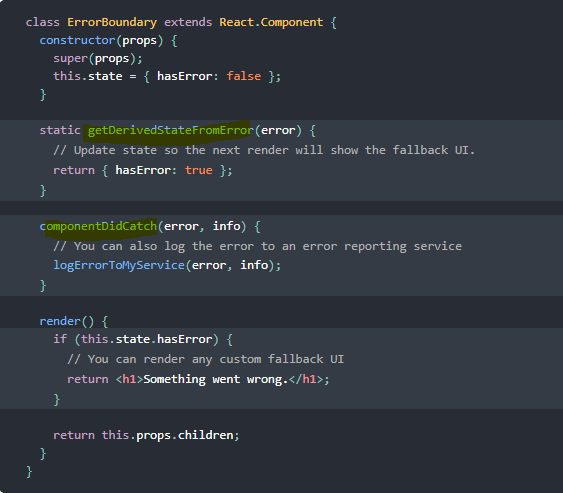
# **Error Boundaries**

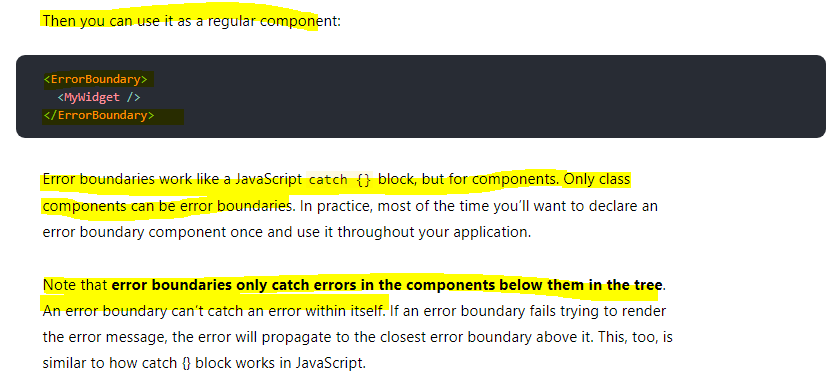
In the past, JavaScript errors inside components used to corrupt React’s internal state and cause it to [emit](https://github.com/facebook/react/issues/4026) [cryptic](https://github.com/facebook/react/issues/6895) [errors](https://github.com/facebook/react/issues/8579) on next renders. These errors were always caused by an earlier error in the application code, but React did not provide a way to handle them gracefully in components, and could not recover from them.



**A class component becomes an error boundary if it defines either (or both) of the lifecycle methods**[**static getDerivedStateFromError()**](https://reactjs.org/docs/react-component.html#static-getderivedstatefromerror)

**or [componentDidCatch()](https://reactjs.org/docs/react-component.html" \l "componentdidcatch). Use static getDerivedStateFromError() to render a fallback UI after an error has been thrown. Use componentDidCatch() to log error information.**





# Context

**Context was introduced by React and it helps us handle cases like this, where you need certain data, certain state in multiple components and you don't want to pass that state across multiple layers of components just to get it from component A at the top to component D at the very bottom when the components B, C in between don't really care about it.**

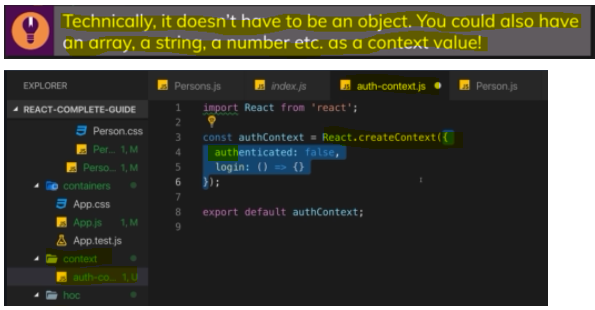
**Context provides a way to pass data through the component tree without having to pass props down manually at every level.**

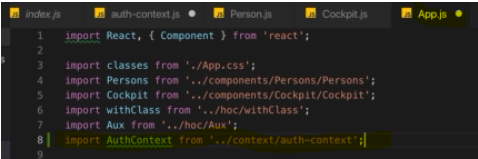
In a typical React application, data is passed top-down (parent to child) via props, but this can be cumbersome for certain types of props (e.g. locale preference, UI theme) that are required by many components within an application. Context provides a way to share values like these between components without having to explicitly pass a prop through every level of the tree.

**When to Use Context**

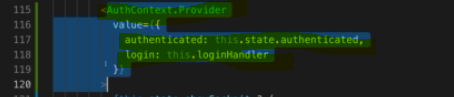
Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language.

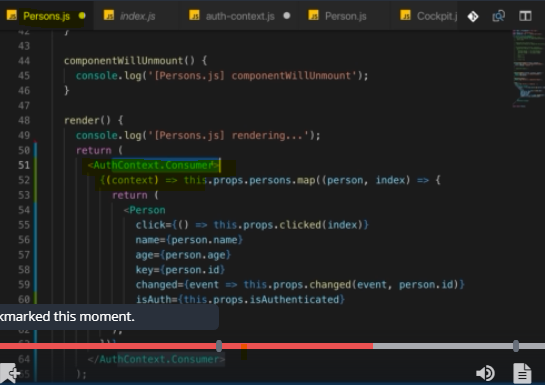
## **Example –**

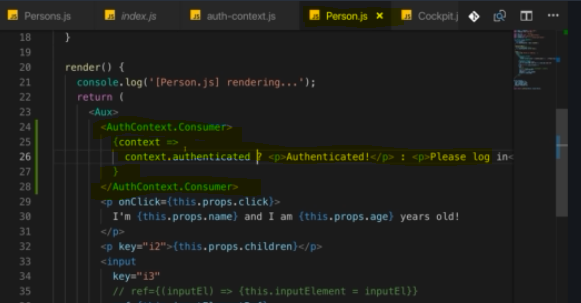


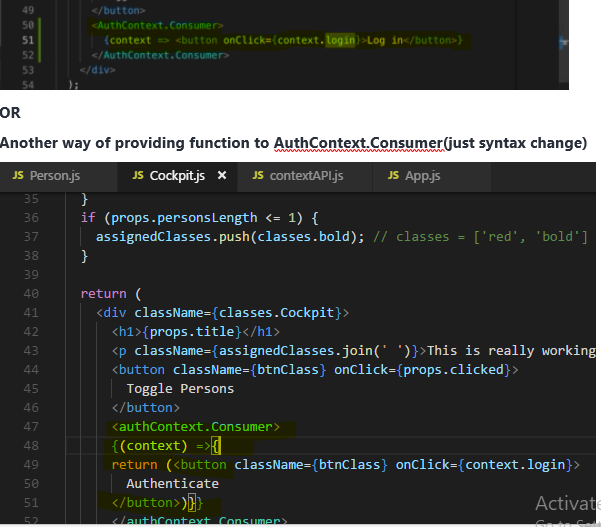


**React will re-render when state or props change. So only changing something in a context object would not cause a re-render cycle and therefore this is not enough. Hence I still manage my authentication status in the state of this component  but I then also store the current state in that authenticated prop of the object I am passing as a value to the authContext and since this effectively is a prop of the authContext provider, this will update whenever this state updates.**









## **contextType and useContext() -Better way of Using ContextAPI( for class-based and functional based component)**

**For class-based component**

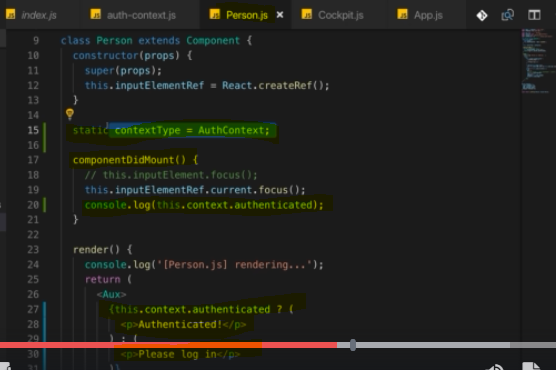
Previous ContextAPI approach code a bit verbose and can be tricky to wrap your head around when you see it the first time, **it also only gives you access to the context in your JSX code here, where you render authContext consumer.**

Thankfully, React 16.6 added another way of using context.

You can add a special static property named **contextType**.

**This allows us to get access to our context even in places like componentDidMount where we previously couldn't.**

**And you can of course also use this context in your render function.**

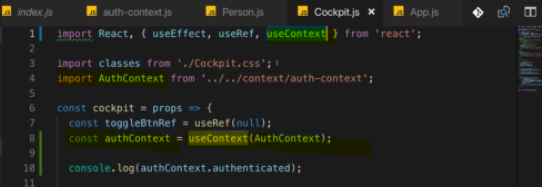


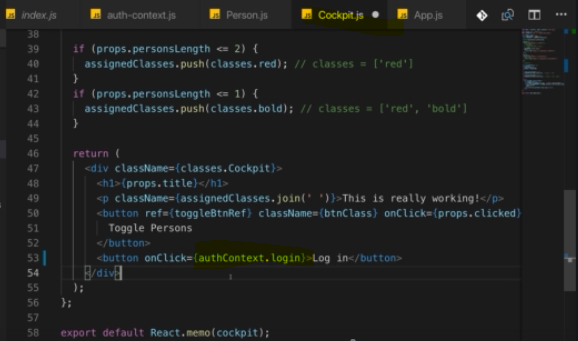
**Now this allows React to automatically connect this component here, this class-based component to your context behind the scenes and it gives you a new property in this component, the this.context property.**

**For Functional Component**

Thankfully, React has you covered with hooks.

**You can import the useContext hook and this now allows you to get access to your context anywhere in your functional component function body**





**So use context is to functional components what static context type is to class-based components you could say.**

# **Prop-types**

you're building a library that you want to distribute to other people, let's say via npm or you're working in a bigger team, then you might have scenarios where people start using your component incorrectly because they don't know that age should be a number, maybe they try to pass in a string. you can do this by importing or by installing an extra package,

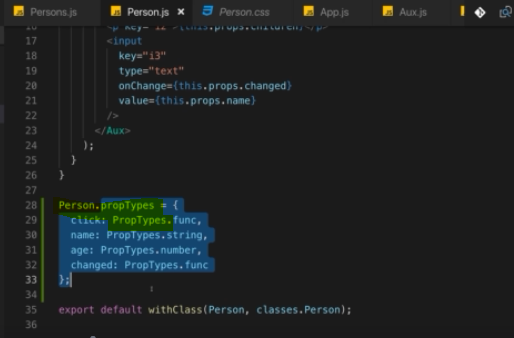


That is provided by the React community or by the official React team actually and it's not built into React core but it is part of React so to say, it's just an extra installation away because it's optional to use.

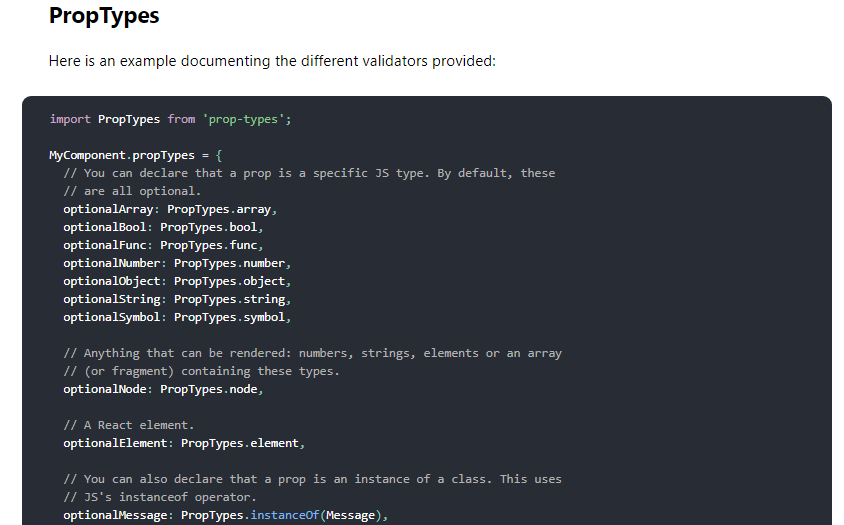
Now lets use it, first import it in your component



Then use it like this.



on your component you can add an extra property and now this works in both functional and class-based components.



**Default prop value**

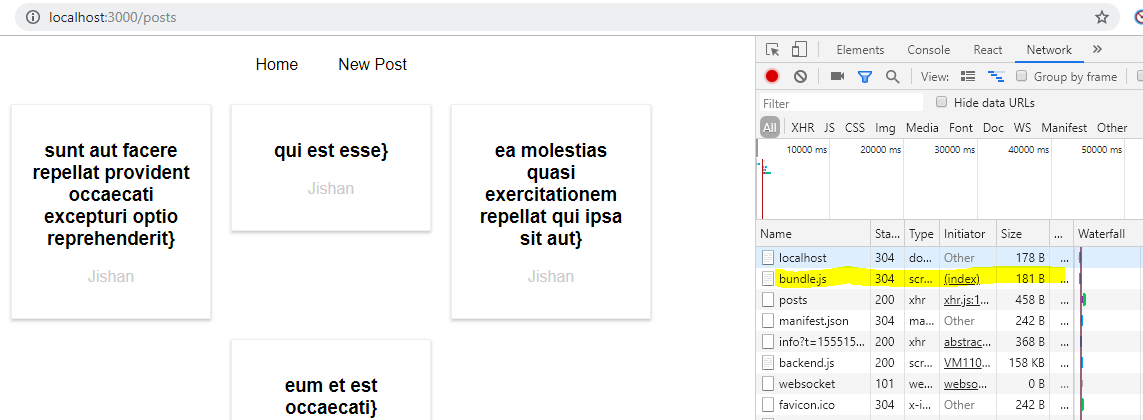
# **Code-Splitting(Lazy Loading)**

**Loading Routes Lazily (Loading Component Asynchronously)**

**The technique of downloading only what you need is known as code splitting or lazy loading and there you would essentially want to make sure that in your component, you're only loading the component once you need it.**

**Bundling is great, but as your app grows, your bundle will grow too. Especially if you are including large third-party libraries. You need to keep an eye on the code you are including in your bundle so that you don’t accidentally make it so large that your app takes a long time to load.**

**To avoid winding up with a large bundle, it’s good to get ahead of the problem and start “splitting” your bundle.** [Code-Splitting](https://webpack.js.org/guides/code-splitting/) is a feature supported by bundlers like Webpack and Browserify (via [factor-bundle](https://github.com/browserify/factor-bundle)) which can create multiple bundles that can be dynamically loaded at runtime.



if we have a look at all the requests once we load the page

this contains all our source code, it's relatively big because we're in development mode, it will be much smaller once you are shipping this for production.

**Still loading the entire bundle with all the code of our application up front can be bad if we have a big application with distinct features and distinct areas in the app where a user might never visit a certain area. Like in our app, we have the NewPost.**

**if the user never visits new post, loading the code responsible for that component doesn't make a lot of sense.  It should only be loaded if the user actually navigates to /newposts ,**

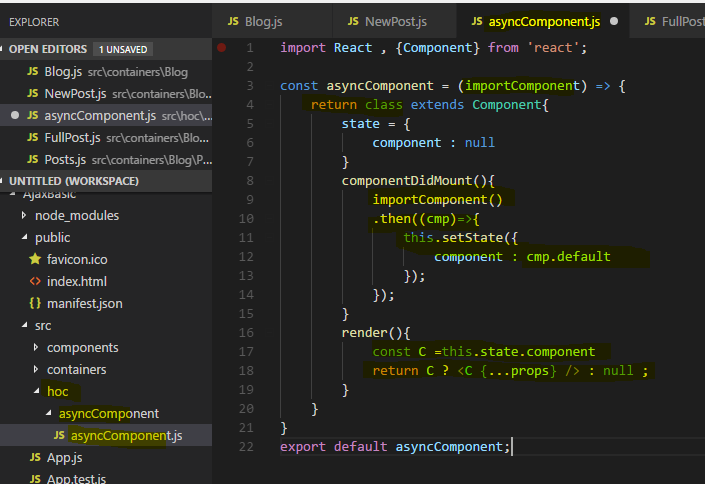
**otherwise new posts and all potential children are never needed.**

**So why should we download the code up front? Would it be better to not download it and hence have a smaller upfront chunk to download and instead download the code responsible for this component and its children when needed.**

**To implement code splitting or lazy loading, we have create react app and react route for and that's important. This technique will work for react router for and for create react app**

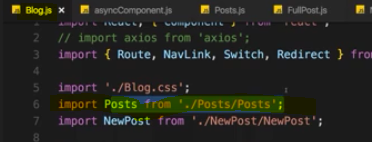
**because code splitting depends heavily on the webpack configuration you are using, it is an advanced concept after all.**

**for this to work in this setup, we need a higher order component**.



**There I expect to get my import component argument which will in the end be a function**

**I want to load this dynamically now Now the thing is whenever you are importing something like this here, with import something from somewhere**



you basically inform webpack, the build tool which gets used behind the scenes about this dependency and it will include it in the global bundle, this is its job.

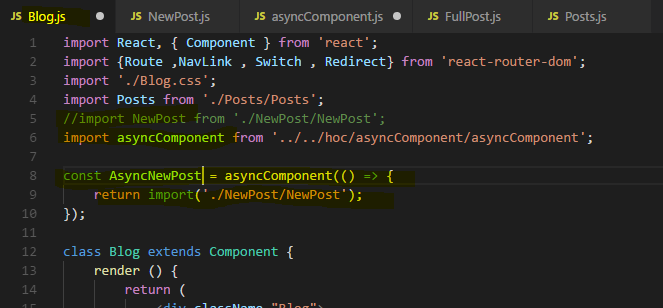
**Now for a lazy loading, this is exactly the opposite of what we want to do, we don't want to include it in the bundle, we want to load it when needed.**

**Still webpack needs to be able to dynamically prepare some extra bundle for this potentially loaded code.**

**I'll use the import keyword as a function.**

**This is a special syntax, the dynamic import syntax which means whatever comes between the parentheses here is only imported when that function here is executed and that function here will only be executed once we render AsyncNewPost to the screen**.

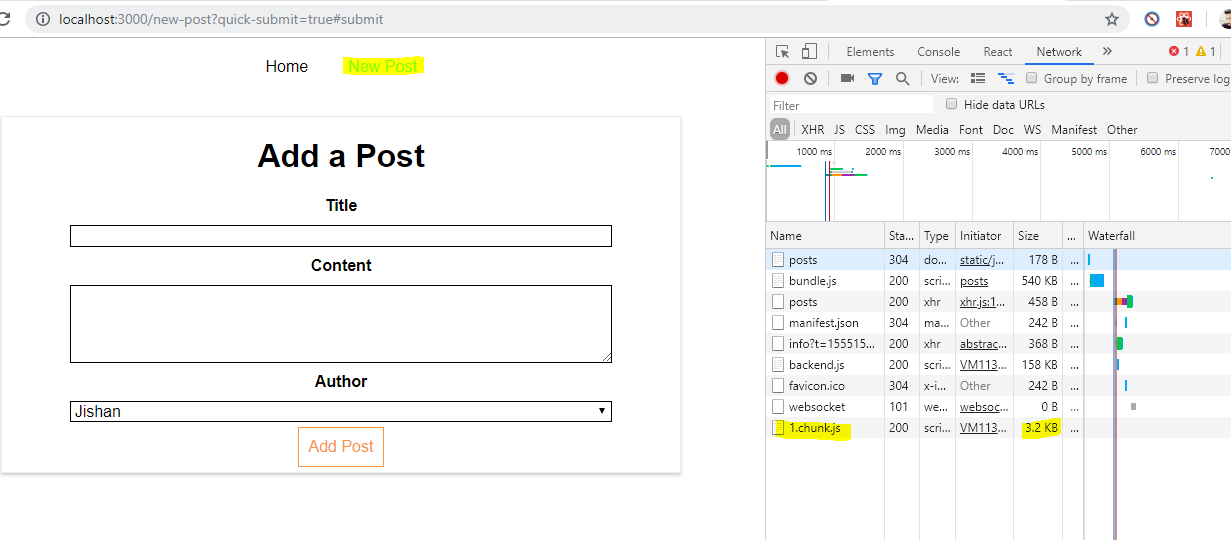
now I'm only importing this when this constant gets used somewhere.



**Eventually this will be a component because keep in mind AsyncComponent returns a component, we have a higher order component. It returns a class with a render method so this is a valid component.**

**This component will eventually render some dynamically loaded component and we decide which component it should be with the function we passed to AsyncComponent.**

**Now if we save this and we go back to our application, watch that path on the bottom right ,when I click on new post. Once I click there you'll see that this 1chunk.js file was loaded which is very small.**

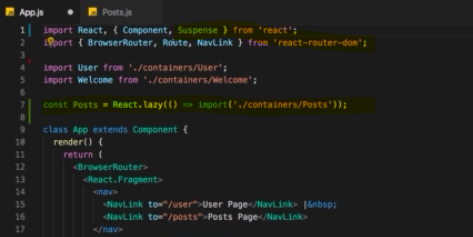


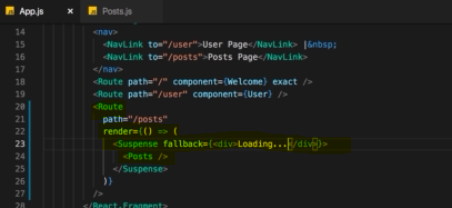
**But it didn't add it to the main bundle, instead it's prepared to load it when needed when we actually include AsyncNewPost which we only do when navigating to /newpost. This is how you load components asynchronously,**

## Lazy Loading with React(16.6 or above)

**if you are using the latest react version or react 16 or higher then you have a new way of lazy loading because react 16.6 adds a new method on the react object.**

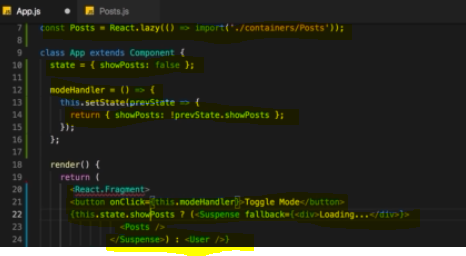
**The lazy method which you can use to load your data your components asynchronously. Which means only when they are needed.**

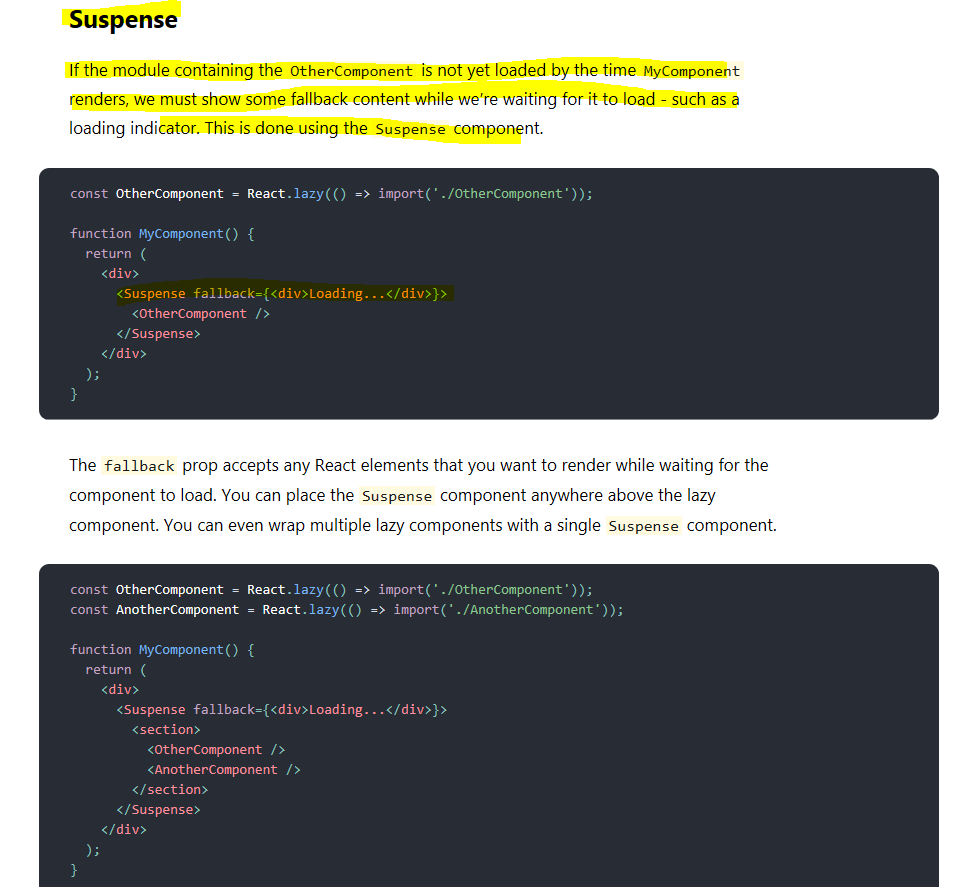




**Code-Splitting is not just useful for routing**

We could of course also have a scenario where we don't use that browser router instead let's say I have a simple button here where I say toggle mode . And now I just want to render either the user or to post page.



**Suspense**

# What is Virtual DOM? How reacts Updates the DOM ? What is reconciliation?

The Virtual DOM (VDOM) is an in-memory representation of Real DOM. The representation of a UI is kept in memory and synced with the "real" DOM. It's a step that happens between the render function being called and the displaying of elements on the screen. This entire process is called reconciliation.

**React basically keeps two copies of the DOM. It has the old virtual DOM and then the re-rendered one, the re-rendered one is the one which gets created when the render method is called.**

**React makes a comparison. It compares the old virtual DOM to the new one and it checks if there are any differences. If it can detect differences, it reaches out to the real DOM and updates it and**

**even then, it doesn't re-render the real DOM entirely, it only changes it in the places where differences were detected,**

**for example if a button text changed, it will only update that text and not re-render the whole button, leave alone the whole DOM.**

**Reconciliation**

When a component's props or state change, React decides whether an actual DOM update is necessary by comparing the newly returned element with the previously rendered one. When they are not equal, React will update the DOM. This process is called reconciliation.

virtual DOM like the real DOM is too a simple js object.React always maintains two virtual DOMs one with the initial state and the other with the updated state of UI. And further the real DOM is updated based on the changes in the two. For this comparison React uses the **diffing algorithm** instead of state of the art algorithms .

