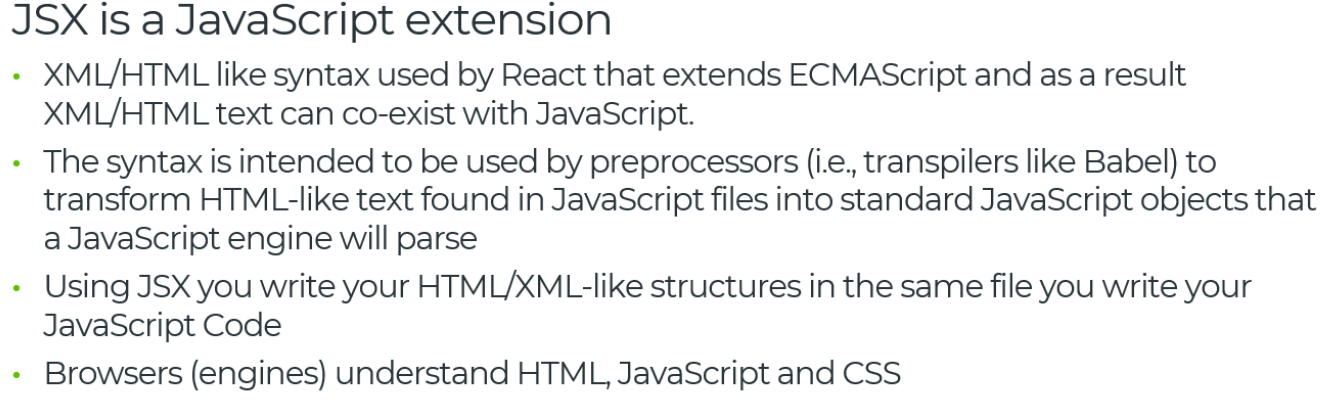
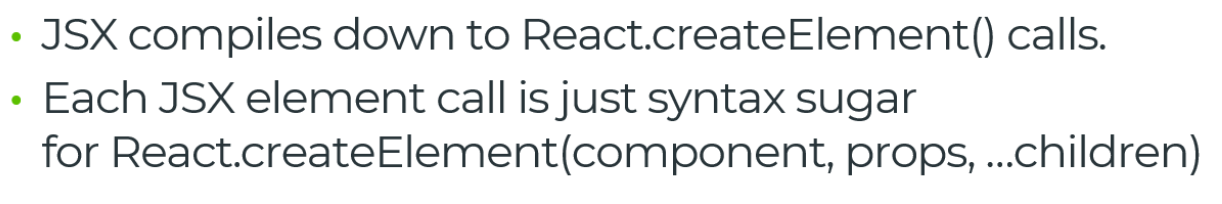
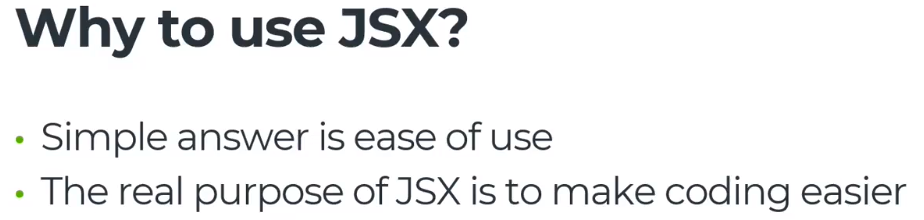
**REACT**

Create-react-app install react and all the third-party libraries we need – a lightweight Dev. Server, Webpack for bundling our files, Babel for compiling our js code and other tools. Babel takes JSX code and converts it to JS that browsers can understand. JSX e JS extension.

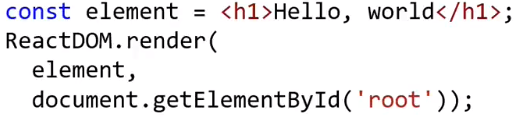
**JSX**



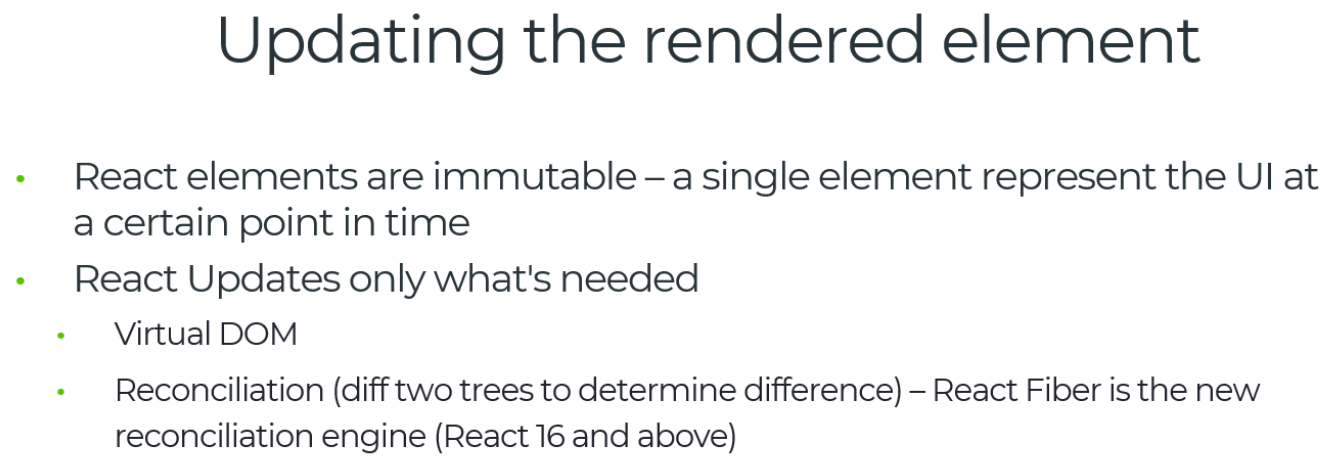




След като сме написали нашето приложение на Реакт, то образува едно дърво и това дърво се вкарва като ноудове в реалния ДОМ на браузъра. Това става посредством React Renderer:

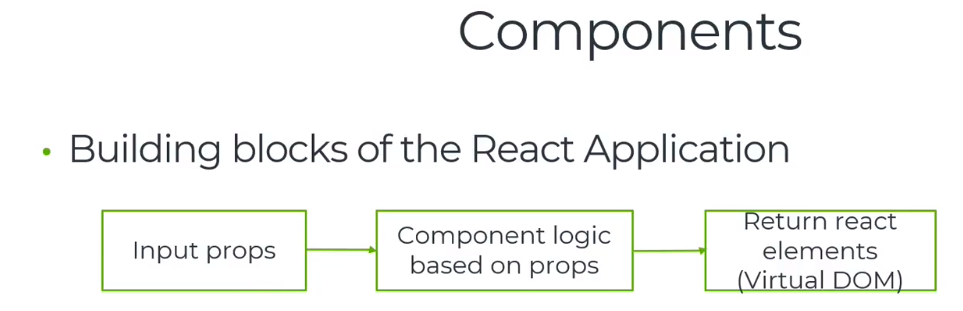


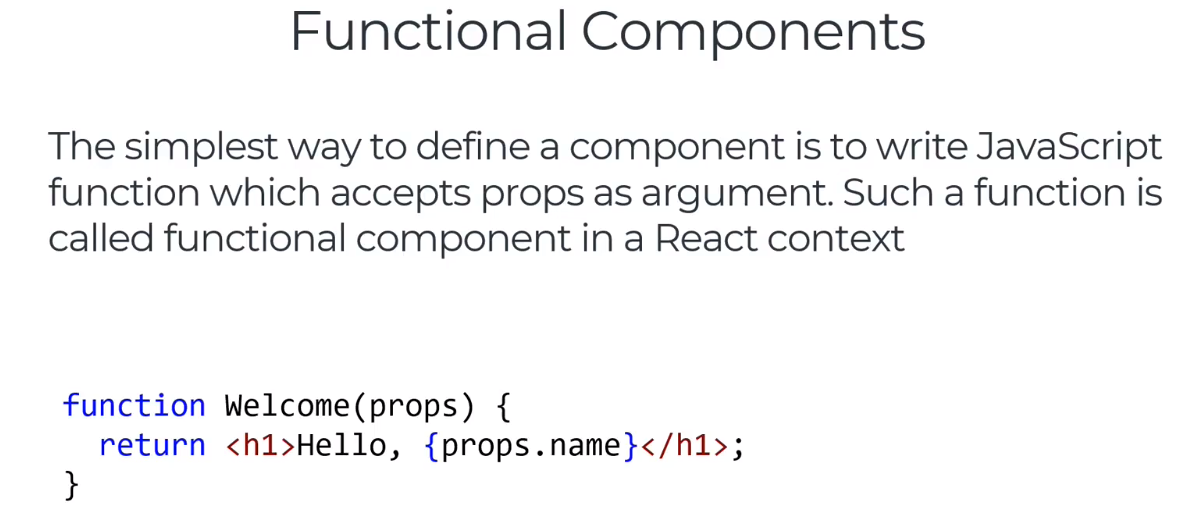
С този код можем да вкараме JSX в реалния DOM.



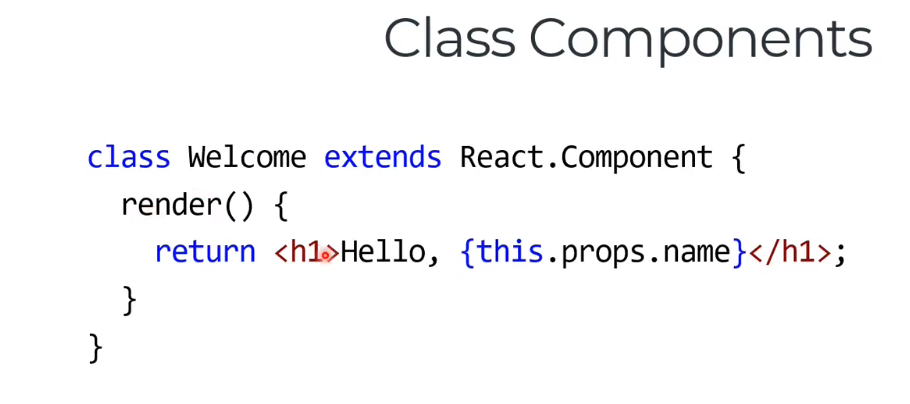
**COMPONENTS**

Те са основният стройтелен материал на Реакт. Комп. Имат като вход пропъртита (props), съответно компонентът вътрешно си има някаква логика, скрита за външния свят и като резултат, комп. Връщат Реакт ноудовете, който Реакт може да манипулира и инджектва за нас

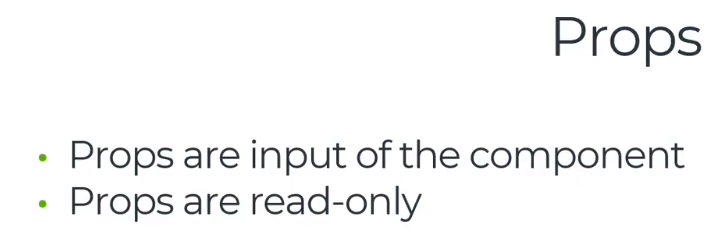




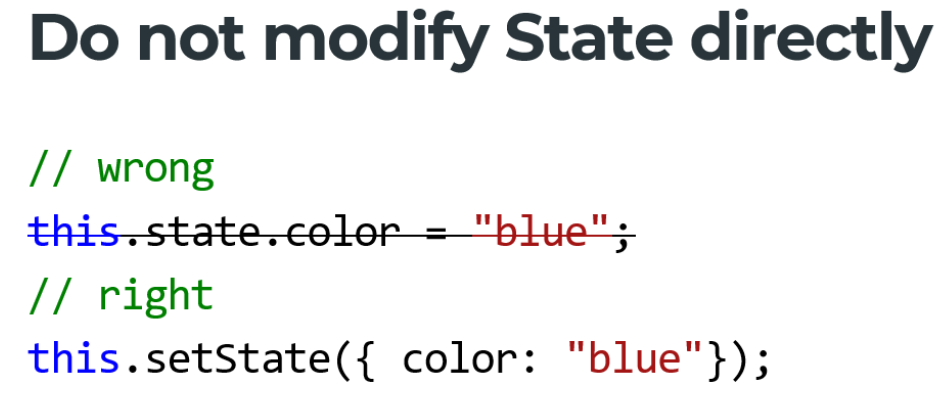
Функц. Комп връщат Реакт Ноуд, чрез JSX



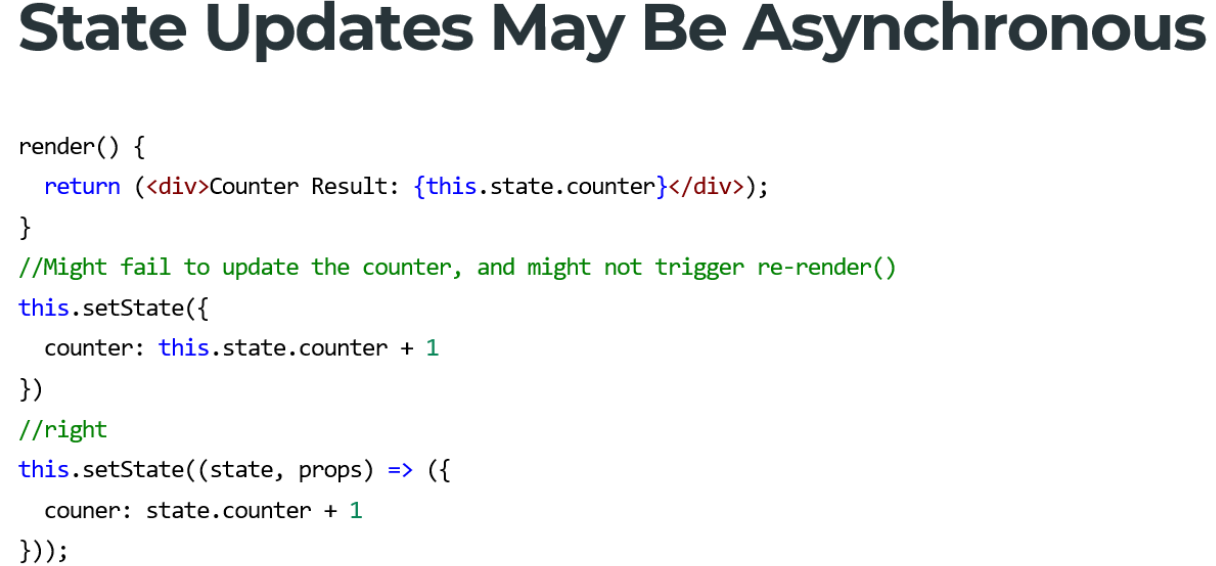
Компонентът не може да променя пропъртитата, той може само да променя стейта си. Напр. ако кафе-машината е един компонент, а проп. Са напр. количеството захар и вида напитка, който си избираме.



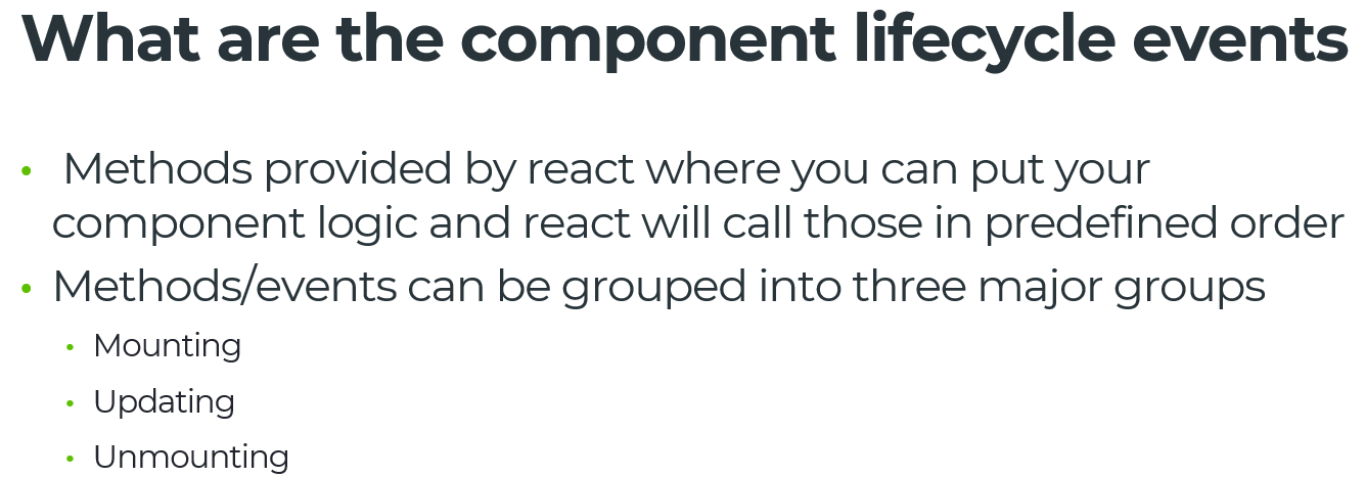
State is plain JS object where you store property values which belong to the component. Проп. Управляват компонента отвън – задава спрямо каква конфигурация, какви пропъртита да свърши нещо. А стейта е нещо, което служи вътрешно на самия компонент. В примера с кафе машината, вътр. Състояние е смилането на кафето, сипването на мляко, пусни 5гр. захар и т.н.

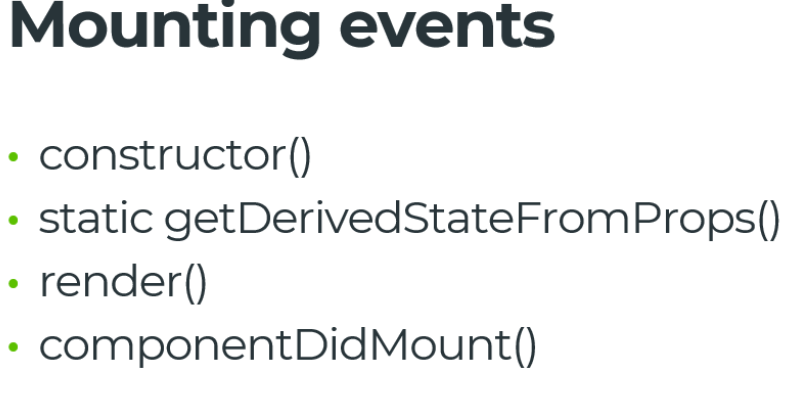
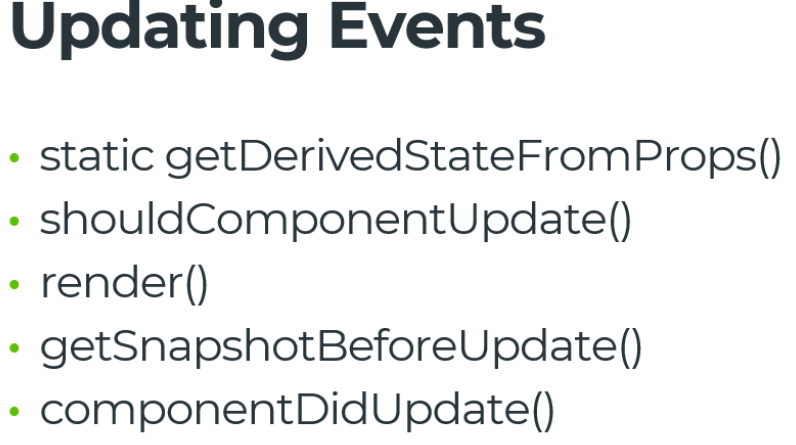
Do not modify state directly. 

State updates are asynchronous и ако разчитате на стейта веднага, след като сте го модифицирали, може да изпаднем в некоректно състояние на стейта. Затова setState има втори аргумент, който е callback func, който се извиква, след като е аплайнат стейта:

State updates may be asynchronous. 

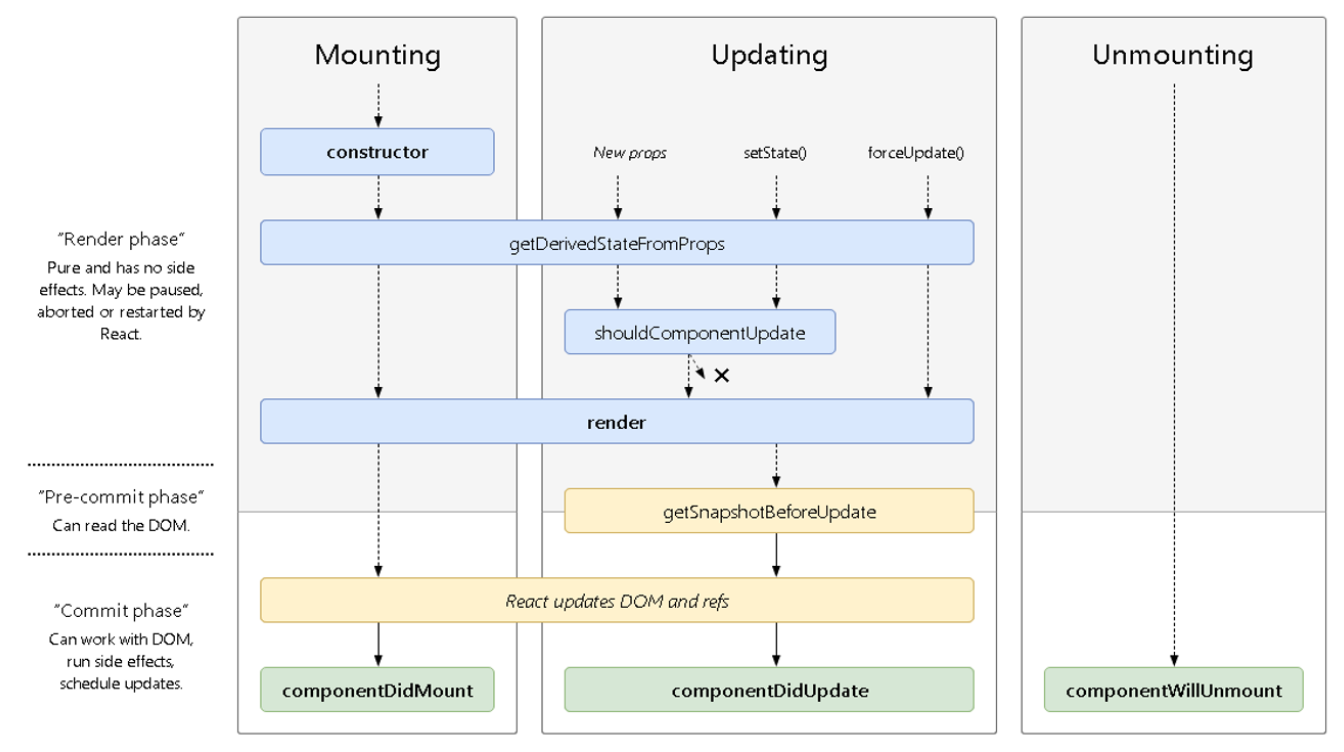
State updates are merged

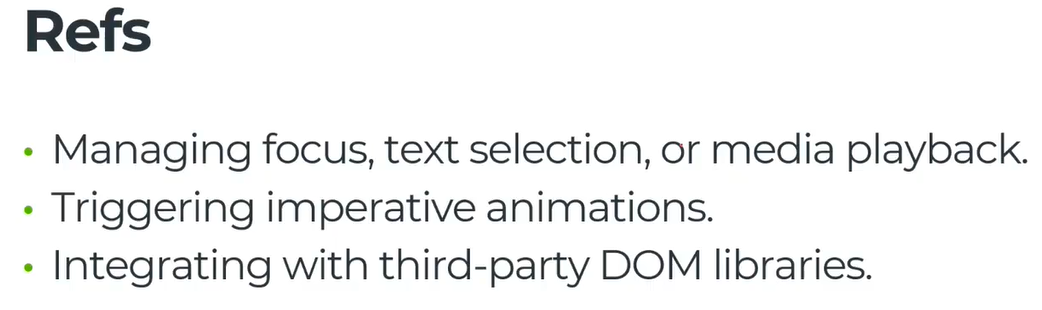


ComponentDidMount() – Node-a след като бъде създаден в Shadow DOM, React преценява кога да го inject към реалния ДОМ. Тогава се хвърля този евент. Т.е. при него вече сме сигурни, че нашият компонент е вътре в реалния ДОМ. Това е евентът, който се ползва обикновено, за да заредим някакви данни от сървъра

ShouldComponentUpdate() – с него можем да прекъснем всички lifecycle events надолу. Там може да се извършва някаква валидация външна и на базата на нея нещо да се рендерира или не. Връща true or false



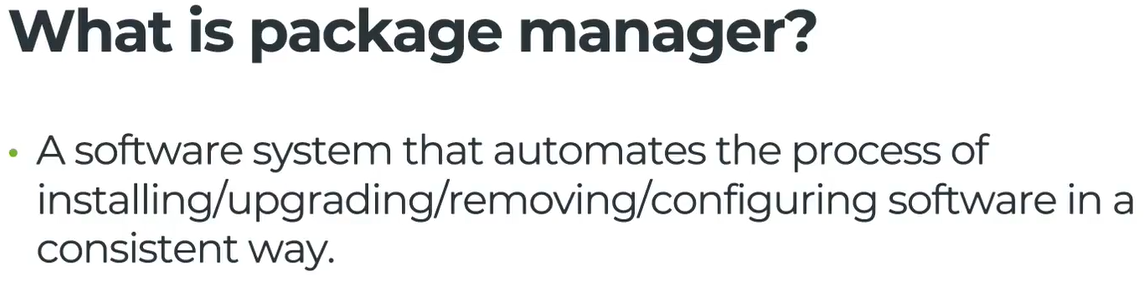


**Functional vs OOP**

At Facebook, we use React in thousands of components, and we haven’t found any use cases where we would recommend creating component inheritance hierarchies.

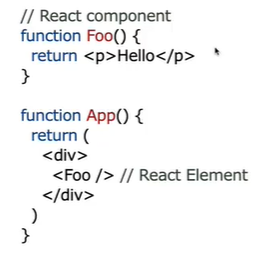
Props and composition give you all the flexibility you need to customize a component’s look and behavior in an explicit and safe way. Remember that components may accept arbitrary props, including primitive values, React elements, or functions.

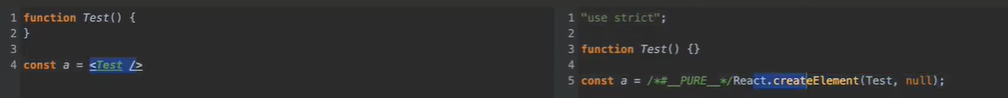
If you want to reuse non-UI functionality between components, we suggest extracting it into a separate JavaScript module. The components may import it and use that function, object, or a class, without extending it.



NPM is package manager for Node.js. Той ни позволява да инсталираме JS библиотеки, без ние да мислим за това. Конфигурацията на NPM stave v package.json

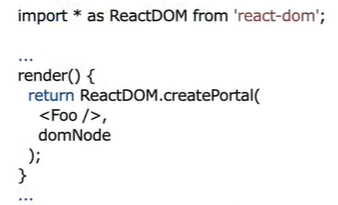
Каква е разликата между Реакт Компонент и Реакт Елемент?



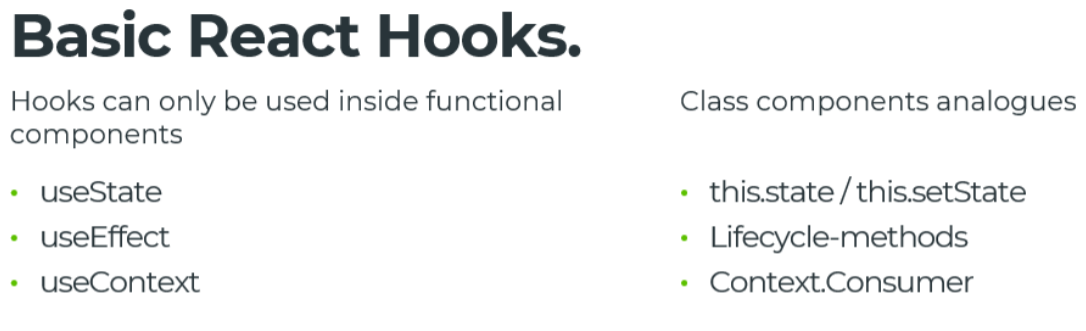


Function Test() e React Component, a const a = <Test /> e React element – т.е. зависи как го илзваме

**CreatePortal:** Имаме възможност от наш компонент, който е много надълбоко в нашия апп, да му подадем елемент (Foo />), който да бъде рендериран в domNode елемент от ДОМ. Може да се ползва напр. когато искаме да покажем някакъв попъп. Друг вариант е, когато имаме някакъв топ левъл елемент, в който да добавяме някакви неща и той да се грижи за тях



**Redux** has one way direction flow

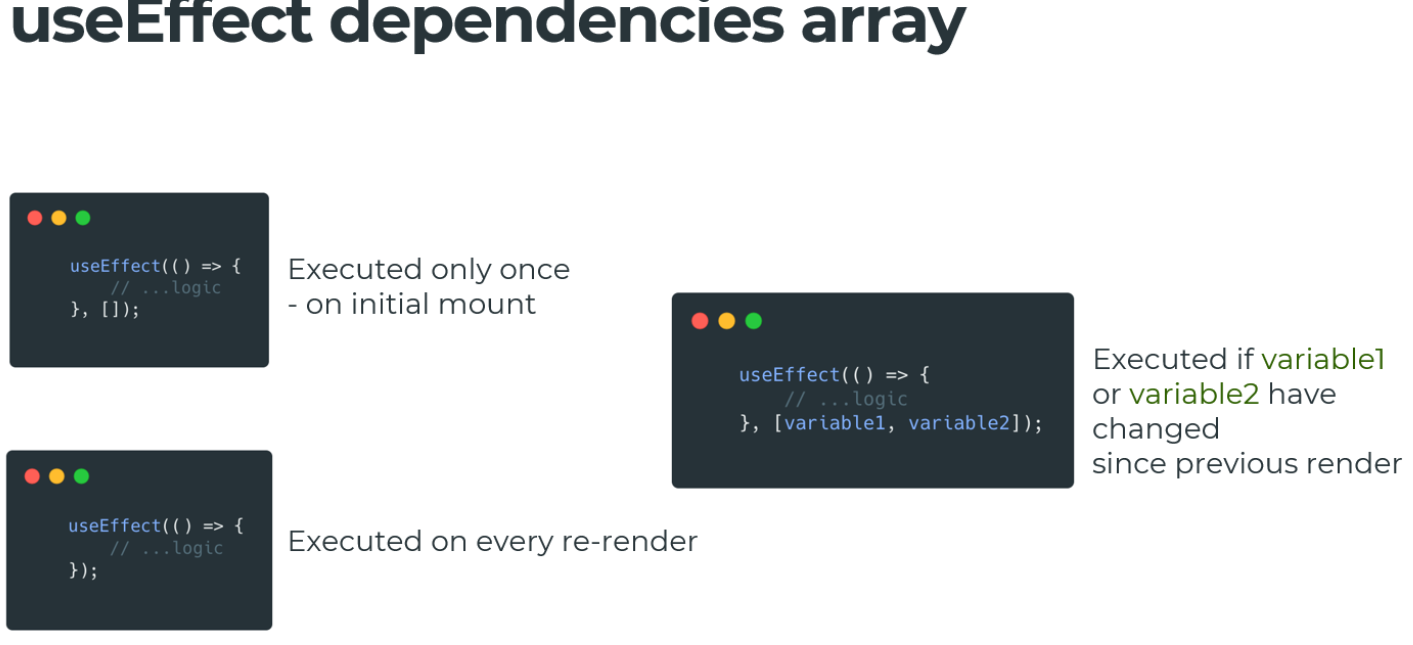




When to **useState? -** For every value that must be persistent between rerenders. For every value that must trigger a rerender. …for nearly everything

Differences between **useState and this.state**

* **useState** can be used multiple times (better state distribultion, allows for rerender **“bail-out”** when the same value is being set).
* No **callback** parameter

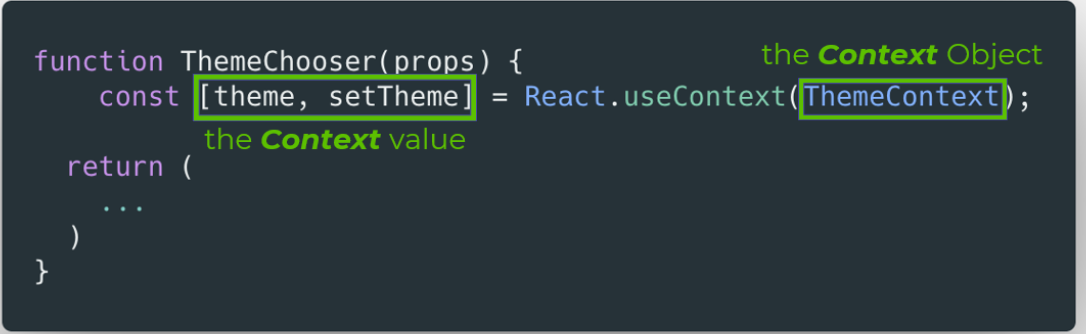


When to **useEffect?**

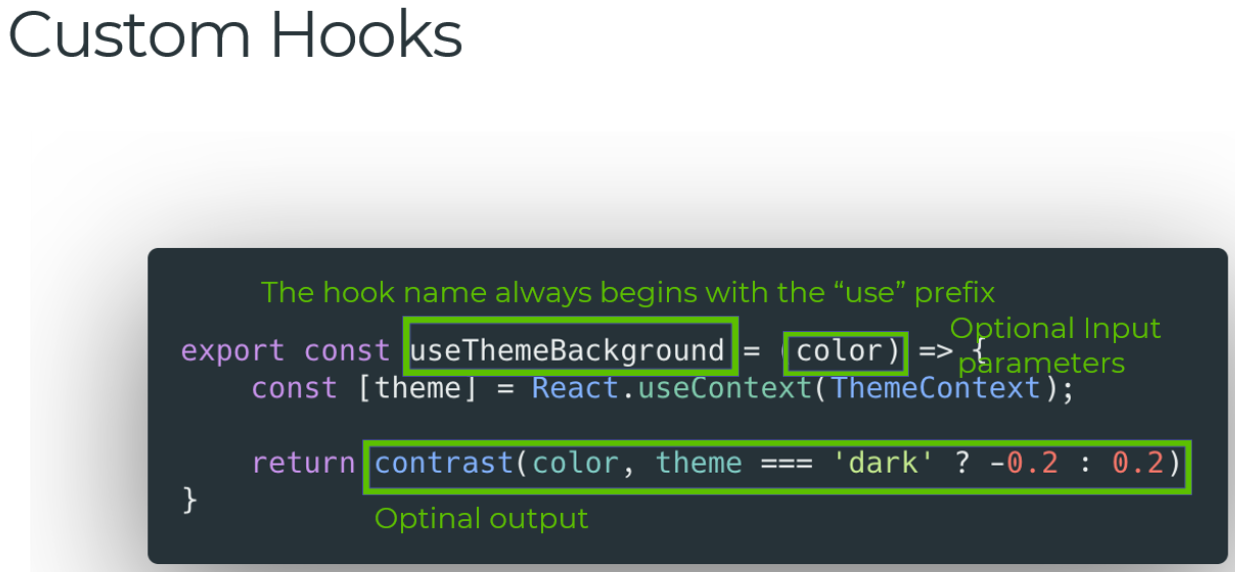
* When working with the DOM
* When fetching data
* Whenever you must execute **side-effects** based on the current rendered tree.

Differences between **useEffect and componentDidMount/Update**

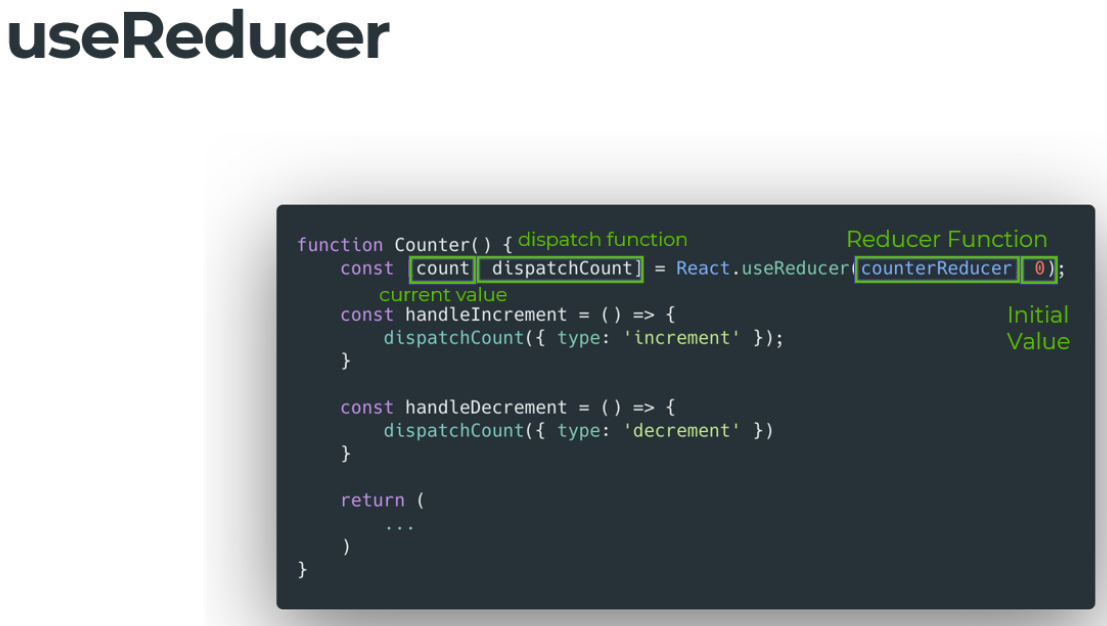
* useEffect is asyncronous
* useEffect accepts a trigger dependencies array
* useEffect combines – componentDidMout, componentWillUnmount, componentDidUpdate, componentWillUpdate(deprecated)



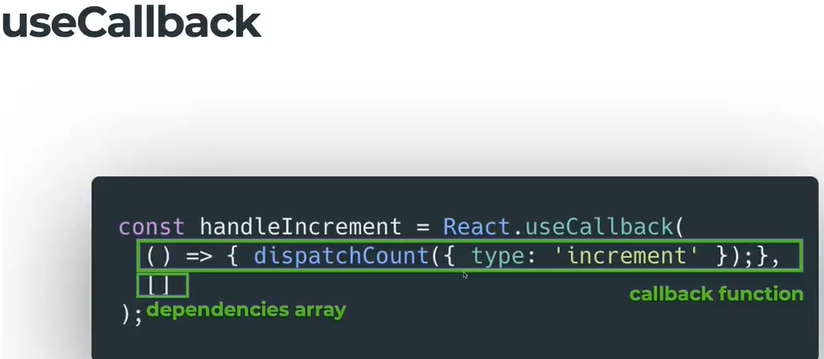
When to **useContext? –**Whenever you want to access the parent context(avoids prop-drilling, removes the neccessity of HOCs, simplifies VIrtualDOM)



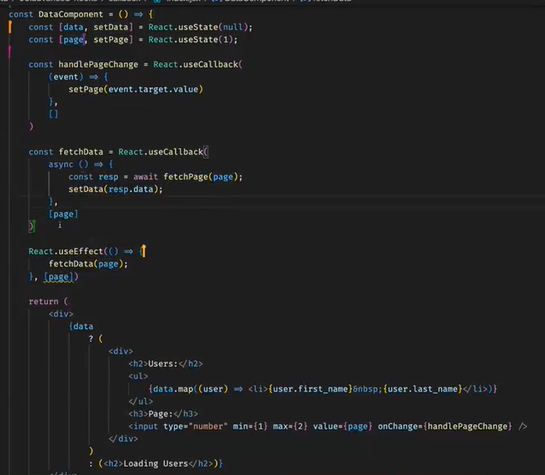
When should you create a custom hook? –When you using more than two react hooks in combination with each other mutliple times –When you want to provide common functionality which depends on react hooks



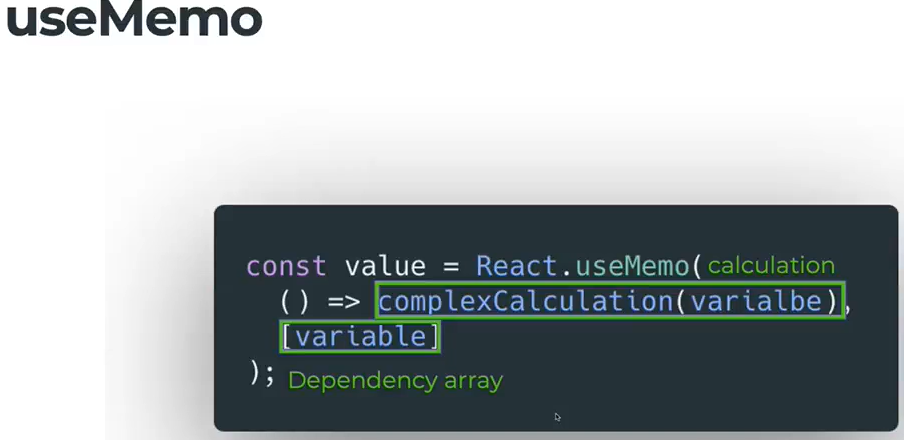
When to **useReducer? –** A recommended approach is to always prefer **useReducer to useState.** Most state variables need more than a ‘set’ function. Easier test. Extensibility



When to **useCallback? –** use it for every callback funtion isnide of your component. Don’t worry about premature optimization. Prevents unnecessary rerender. If the function begins with ‘handle’ it most probably should be wrapped in **useCallback**. Приема два параметъра – първият е функция, която искаме да се изпълни, вторият е подобен на този на useEffect, само че тук в dependency array задаваме кога да се направи нова инстанция на тази функция, а не кога да се изпълни както при useEffect



Функцията async() => {..} да се промени само когато се промени page. А useEffect искаме да се промени, когато се промени fetchData. A fetchData се променя, когато се промение page.

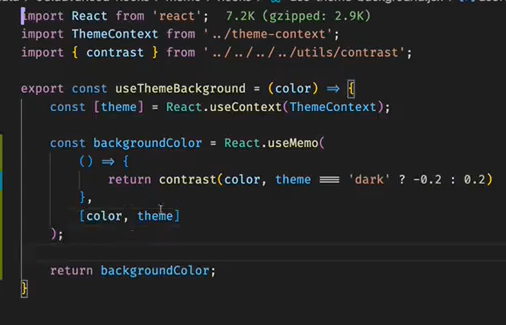


Много прилича на **useCallback()**. Като първи параметър му даваме функция, но без параметри и някакво изчиследние, което искаме да се случи и отново array s dependencies, който упоменава кога тази сметка трябва да се повтори.

When to **useMemo? –** for every complex calculation that can be avoided during rerender. Don’t worry about premature optimizastion. Prevents unnecessary rerenders.

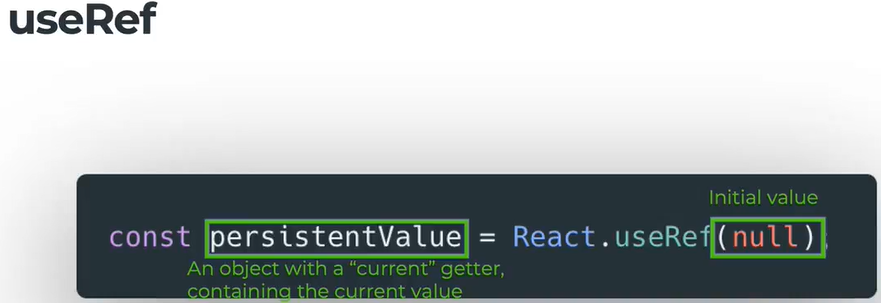
Когато правим някакви сложни сметки – математически, факторие, обикаляме дървета, обръщаме масиви и когато не искаме тази сметка да се случва на всеки render.

Може би **useMemo** има много повече използване от **useCallback**



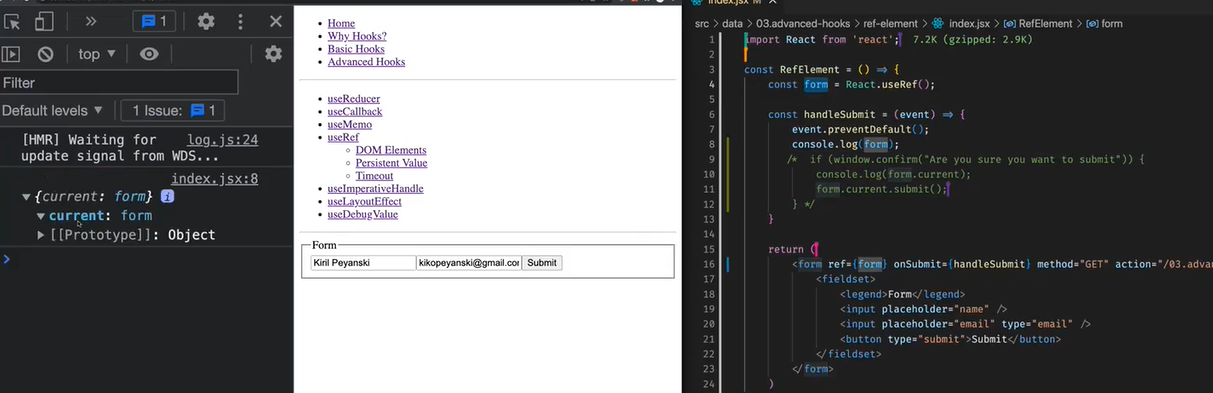
В повечето случай каквото се ползва в useMemo, това трябва да сложим в dep. Array.

useMemo връща резултат, а useCallback дефинира просто нова функция всеки път когато депенд. Се промени.



Резултатът от useRef винаги е обект, в който има current, където всъщност е стойността, която търсим – това е getter-a. Това е state, който не trigger state update

When to useRef? –for getting a refenrece to a DOM element(не искаме да пазим DOM елементи в state) –use useRef in combination with userImperativeHandle to get a reference to a React Component –for persisting any value between rerenders, което е аналогично на Class Fields –for maintaing timeouts



В случая form служи и за getter and setter