Websites : github.com/enaqx/awesome-react

* React:

React is a library which provides modern, reactive user interfaces for the web. It provides component based approach.

**Features of react:**

* It is a library not framework.
* Virtual DOM: update the DOM in most efficient way, which avoids layout thrash(restricting the layout) and in mobile devices saves battery and cpu.
* By default it follows one-way binding
* Angular is template-centric whereas react is JavaScript-centric
* It has mvc in a single component and each component can be independent of each other. Form, accordion, button can be different components.

**What is Componenet? And why one should use it?**

Components are the basic building block which we require to create user Interface.

React components are independent and reusable code. They are the building blocks of any React application. Components serve the same purpose as JavaScript functions, but work individually to return JSX code as elements for our UI.

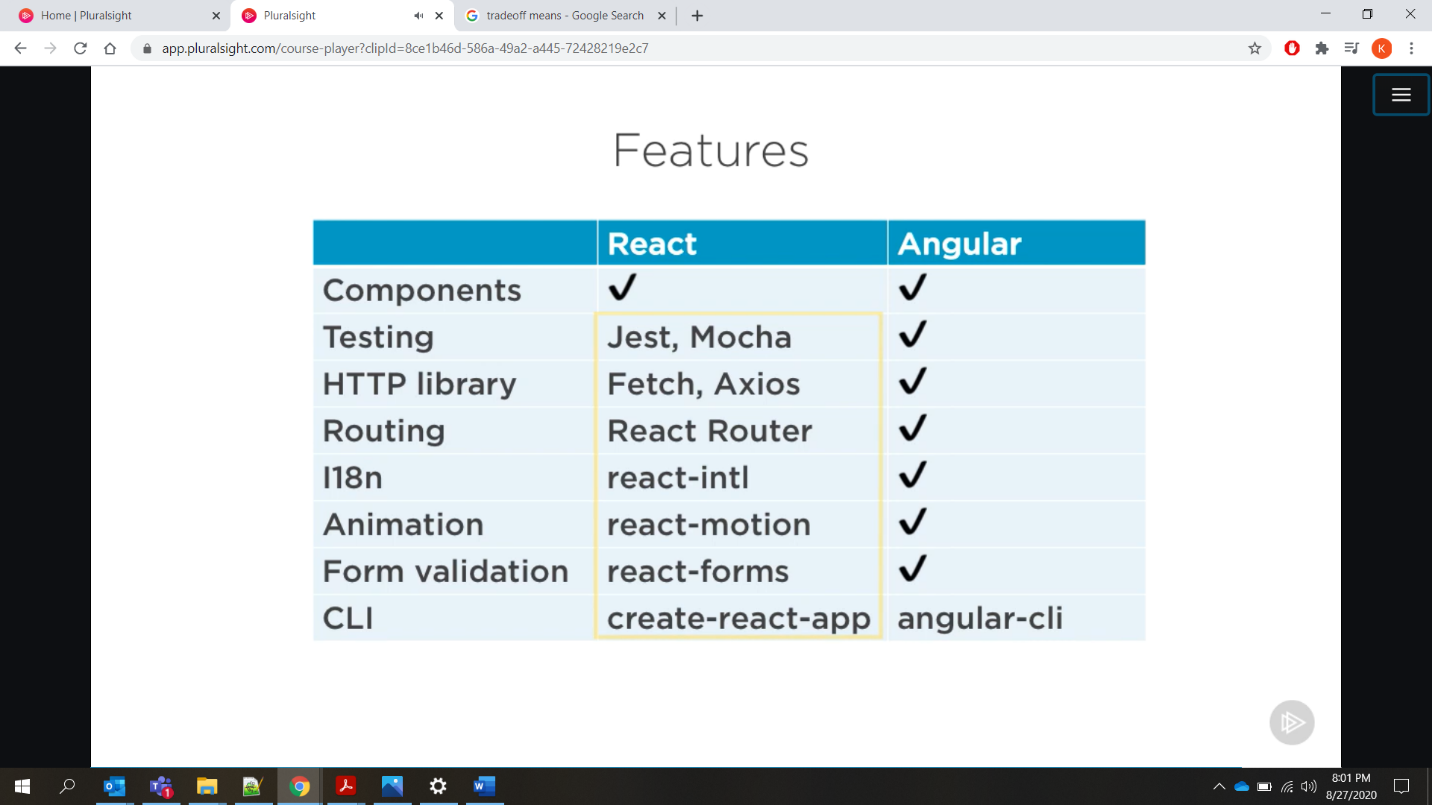
Benefits:

- Reusability

- Separation of concerns

**React components:**

* Function component
* Class component



**Why not web compoents**

* Spotty browser support- need polyfills to work in many browsers
* JS libraries keeps innovating
* Only run in the browser, not on mobiles

**Why React?**<https://jscomplete.com/why-react>

**Virtual DOM:**  
[https://reactjs.org/docs/virtual-dom](https://reactjs.org/docs/faq-internals.html%23:~:text=The%20virtual%20DOM%20(VDOM)%20is,This%20process%20is%20called%20reconciliation.&text=They%20may%20also%20be%20considered,virtual%20DOM)

**Event in react take function reference not function invocation**

* The public folder contains static files such as index. html, javascript library files, images, and other assets, etc. which you don't want to be processed by webpack.

**Tree Reconciliation:**

React provides a declarative API so that you don’t have to worry about exactly what changes on every update. Reconciliation is the process through which React updates the DOM. When a component’s state changes, React has to calculate if it is necessary to update the DOM. It does this by creating a virtual DOM and comparing it with the current DOM. In this context, the virtual DOM will contain the new state of the component.

* <https://reactjs.org/docs/reconciliation.html>
* <https://css-tricks.com/how-react-reconciliation-works/>

**To mount and unmount the react container just change the key.**

**CustomHooks**

* Use “use” before naming a custom hook.

**CustomHook Rules:**

* Don’t call hooks inside loops or conditions

**JSX:**

<https://reactjs.org/docs/introducing-jsx.html>

**JSX Restrictions:**

All attributes names can’t be similar as that of html.

For Ex. we use className instead of class.

* **Key** is necessary in react list because it update the list efficiently. To fin out which element changed and not the whole list.
* **It is best practice** to not mutate or update state directly when array comes into picture. Because array is reference based.

NEVER mutate this.state directly, as calling setState() afterwards may replace the mutation you made. Treat this.state as if it were immutable.

setState() does not immediately mutate this.state but creates a pending state transition. Accessing this.state after calling this method can potentially return the existing value.

There is no guarantee of synchronous operation of calls to setState and calls may be batched for performance gains.

setState() will always trigger a re-render unless conditional rendering logic is implemented in shouldComponentUpdate(). If mutable objects are being used and the logic cannot be implemented in shouldComponentUpdate(), calling setState() only when the new state differs from the previous state will avoid unnecessary re-renders.

**Components & JSX Cheat Sheet**

Components are the **core building block of React apps**. Actually, React really is just a library for creating components in its core. A typical React app therefore could be depicted as a **component tree** - having one root component ("App") and then an potentially infinite amount of nested child components.

Each component needs to return/ render some **JSX** code - it defines which HTML code React should render to the real DOM in the end.

**JSX is NOT HTML** but it looks a lot like it. Differences can be seen when looking closely though (for example className in JSX vs class in "normal HTML"). JSX is just syntactic sugar for JavaScript, allowing you to write HTMLish code instead of nested React.createElement(...) calls.

When creating components, you have the choice between**two different ways:**

1. **Functional components** (also referred to as "presentational", "dumb" or "stateless" components - more about this later in the course) => const cmp = () => { return <div>some JSX</div> } (using ES6 arrow functions as shown here is recommended but optional)
2. **class-based components** (also referred to as "containers", "smart" or "stateful" components) => class Cmp extends Component { render () { return <div>some JSX</div> } }

We'll of course dive into the difference throughout this course, you can already note that you should use 1) as often as possible though. It's the best-practice.

**Props & State**

props  and state  are **CORE concepts** of React. Actually, only changes in props  and/ or state  trigger React to re-render your components and potentially update the DOM in the browser (a detailed look at how React checks whether to really touch the real DOM is provided in section 6).

**Props**

props  allow you to pass data from a parent (wrapping) component to a child (embedded) component.

**Example:**

**AllPosts Component:**

1. const posts = () => {
2. return (
3. <div>
4. <Post title="My first Post" />
5. </div>
6. );
7. }

Here, title  is the custom property (prop ) set up on the custom Post  component. We basically replicate the default HTML attribute behavior we already know (e.g. <input type="text">  informs the browser about how to handle that input).

**Post Component:**

1. const post = (props) => {
2. return (
3. <div>
4. <h1>{props.title}</h1>
5. </div>
6. );
7. }

The Post  component receives the props  argument. You can of course name this argument whatever you want - it's your function definition, React doesn't care! But React will pass one argument to your component function => An object, which contains all properties you set up on <Post ... /> .

{props.title}  then dynamically outputs the title  property of the props  object - which is available since we set the title  property inside AllPosts  component (see above).

**State**

Whilst props allow you to pass data down the component tree (and hence trigger an UI update), state is used to change the component, well, state from within. Changes to state also trigger an UI update.

**Example:**

**NewPost Component:**

1. class NewPost extends Component { // state can only be accessed in class-based components!
2. state = {
3. counter: 1
4. };
6. render () { // Needs to be implemented in class-based components! Needs to return some JSX!
7. return (
8. <div>{this.state.counter}</div>
9. );
10. }
11. }

Here, the NewPost  component contains state . Only class-based components can define and use state . You can of course pass the state  down to functional components, but these then can't directly edit it.

state  simply is a property of the component class, you have to call it state  though - the name is not optional. You can then access it via this.state  in your class JSX code (which you return in the required render()  method). Whenever state  changes (taught over the next lectures), the component will re-render and reflect the new state. The difference to props  is, that this happens within one and the same component - you don't receive new data (props ) from outside!

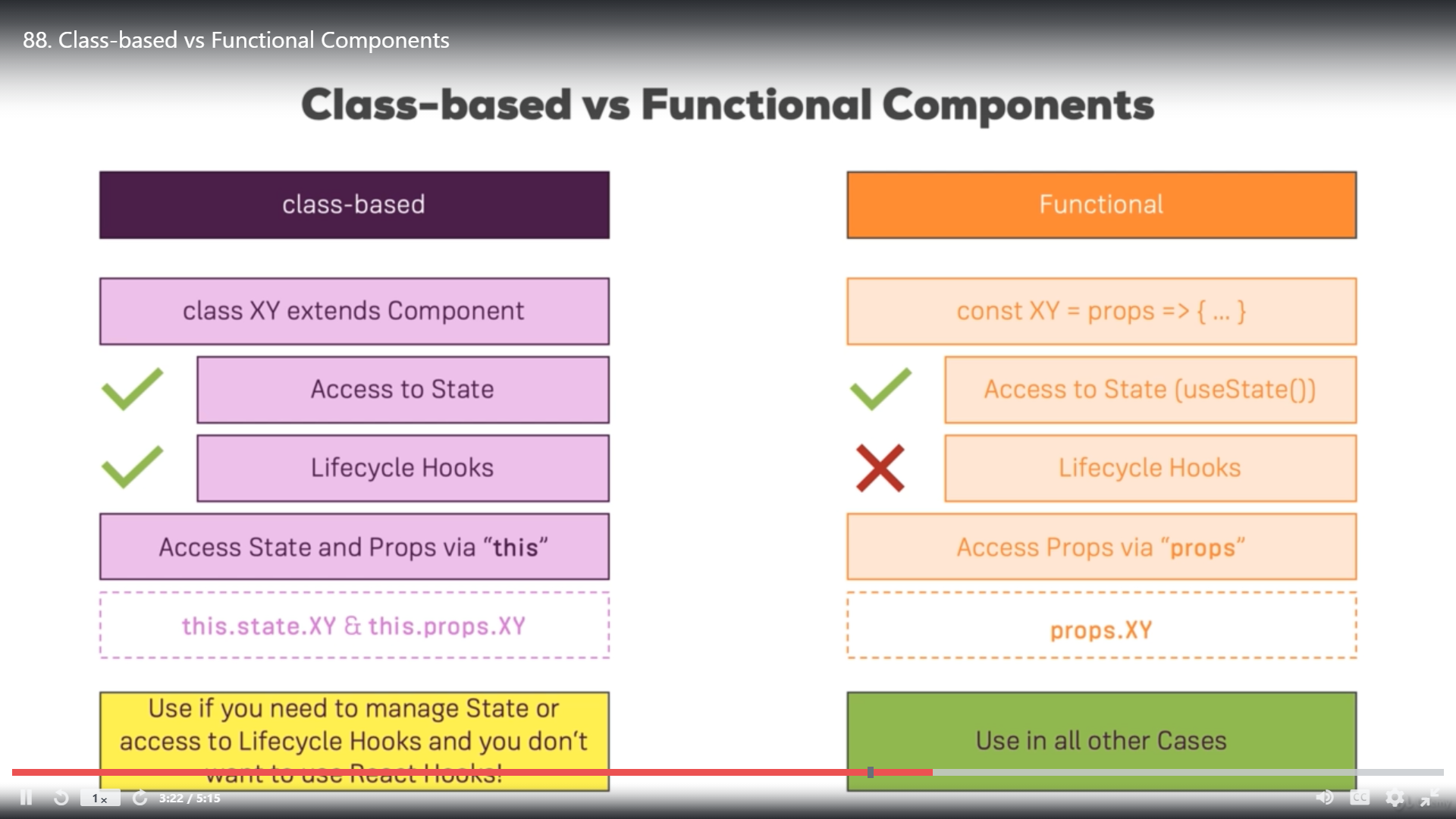
**Difference between class based component and function based components:**

Class based setState automatically merges the object passed as a parameter with the existing state but function based component replaces the existing state with the new object passed as a param.

A component with states defined in it is also know as smart component, container component,

stateful component. And component without state is known as dumb component.

Presentational component is a component in which we do not manage state.



* **CSS Modules:**

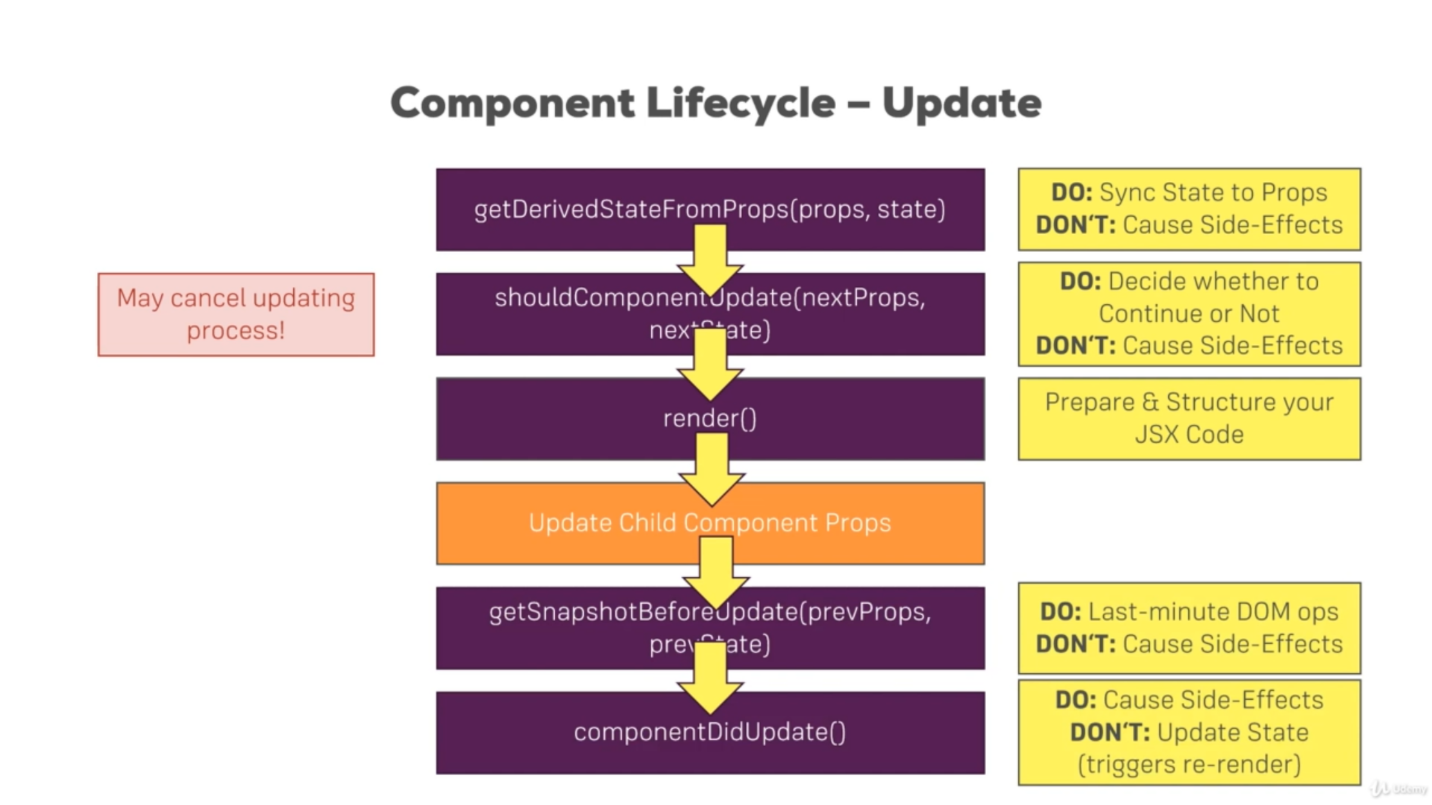
<https://medium.com/nulogy/how-to-use-css-modules-with-create-react-app-9e44bec2b5c2>

* **ErrorBoundary: (React 16+) only works in prod**<https://reactjs.org/docs/error-boundaries.html>

Use error boundary only when you know error might occur and you can’t handle it.

* **Lifecycle hooks: (**<https://reactjs.org/docs/react-component.html>**)**

getDerivedStateFromProps is part of both creation and update lifecycle.



Functional component uses useEffect React hook to maintain state like update lifecycle hooks in Class component. It contains componentDidMount and ComponentDidUpdate.

* useEffect(() => {

// http call

return () => {

console.log(‘last statement call when component is destroyed’);

}

}, [])

Above http call happens only one time after the component creation only when empty array is passed and return statement call happens when component is destroyed.

* useEffect(() => {

// http call

return () => {

console.log(‘last statement call’);

}

})

when 2nd param is not passed then useEffect will run for every update cycle. And return(cleanUp) statement happens before http call is made.

* And when 2nd argument is passed inside array then useEffect will run only when argument passed is updated.
* PureComponent: It already has condition check shouldComponentUpdate() when it checks all the updated props properties with the old ones. If any props properties changes then only it will render the component otherwise not.
* **React.memo** is a higher order component.

If your component renders the same result given the same props, you can wrap it in a call to React.memo for a performance boost in some cases by memoizing the result. This means that React will skip rendering the component, and reuse the last rendered result.

React.memo only checks for prop changes. If your function component wrapped in React.memo has a useState or useContext Hook in its implementation, it will still rerender when state or context change.

* **High order components:**

**To Which Events Can You Listen?**

You can find a list of supported events here:

<https://reactjs.org/docs/events.html#supported-events>

* **Block statement doesn’t work inside {} as jsx expression.**
* **When previous state/old state is involved in setState method then it is best practice to pass prevState as param in setState method. This will save our state without any inconsistency**

this.setState((prevState, props) => {

return {

persons: persons,

changeCouter: prevState.changeCouter + 1

}

})

* Always update state Immutably

const newArr = […oldArr];

this statement will copy and create a new Array.