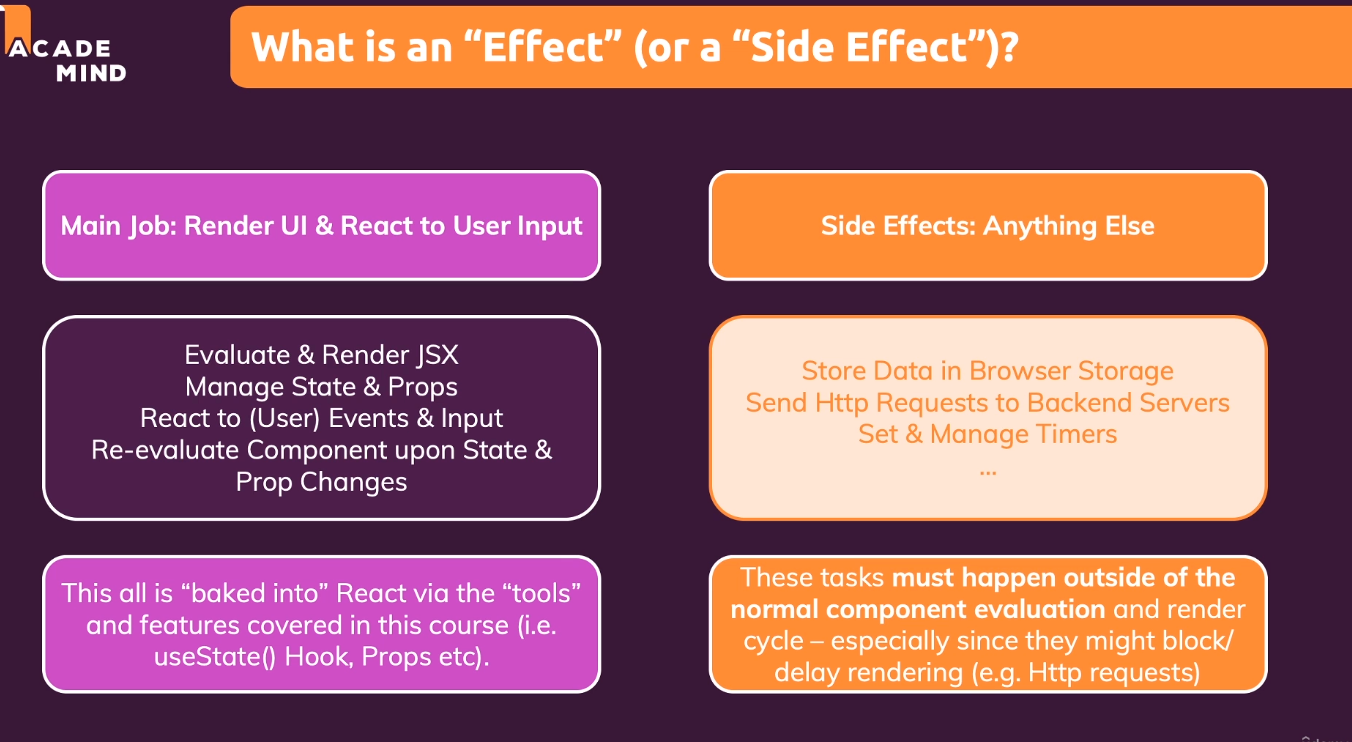
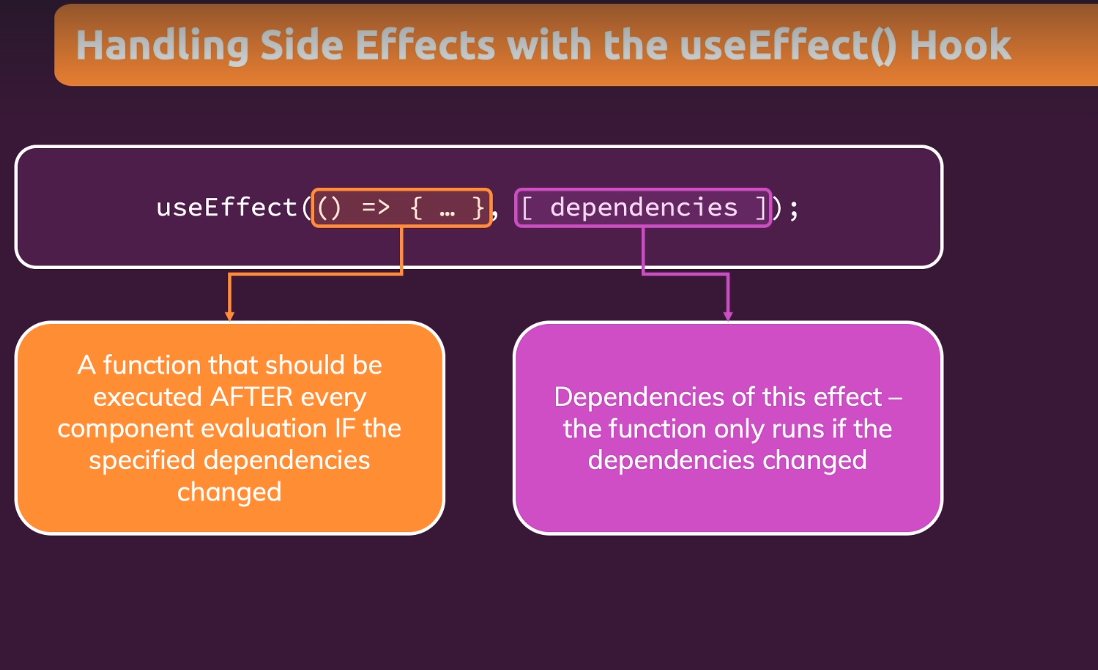
What is a side effect?

Main Job of react is to render and re-render JSX when state changes.

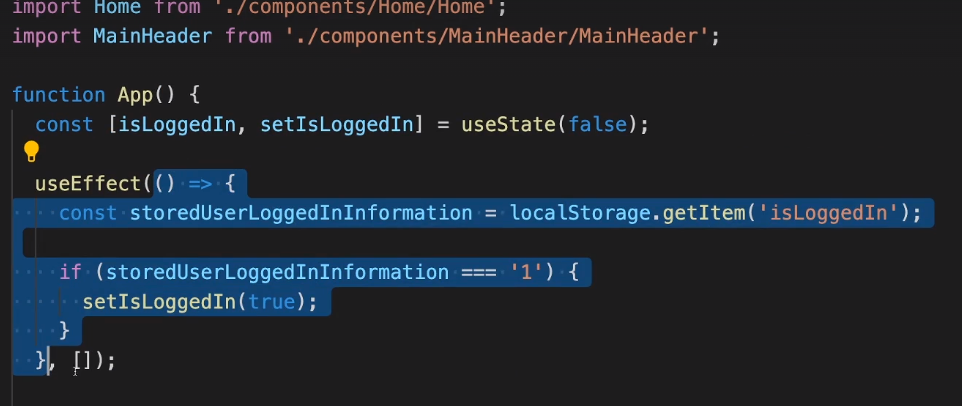


Side effects should not directly go into the functions as this can result in infinite loop or unexpected behaviour of the component. It can even lead to unnecessary multiple http calls.

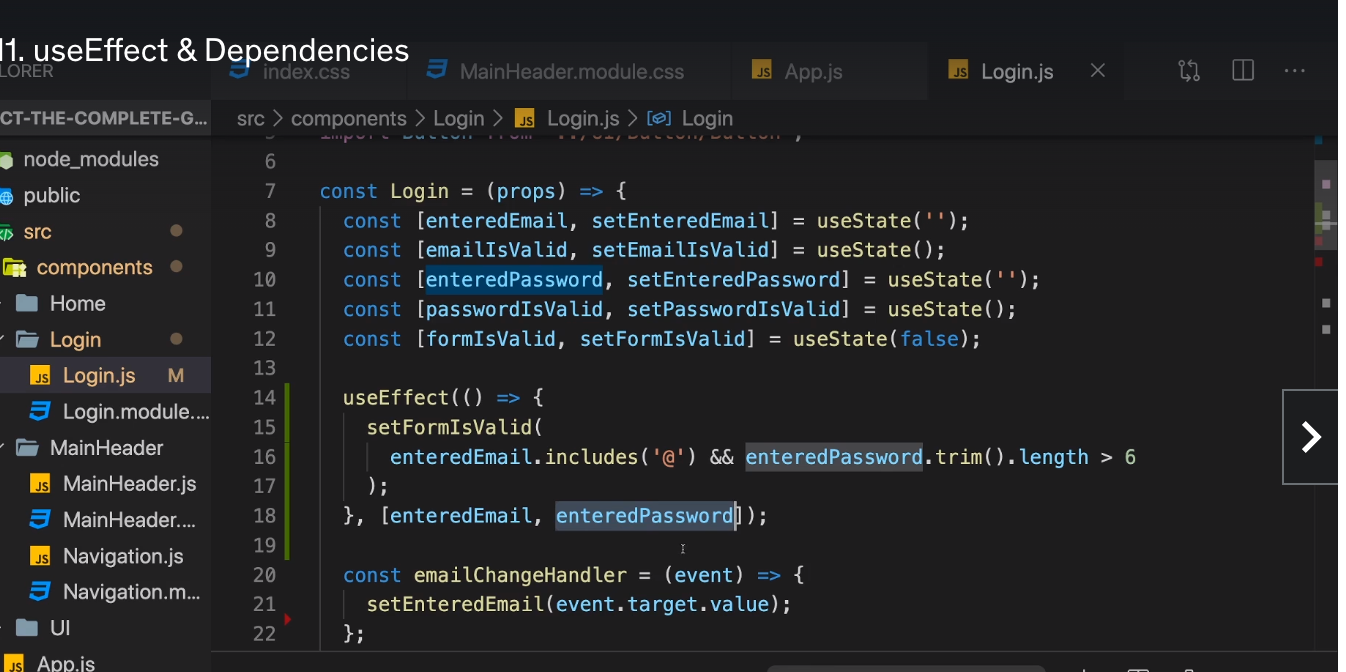
useEffect() Hook is used to handle the scenario.



Following is an example of useEffect() hook in the below example if we do not use useEffect() hook, the component would be re-rendered because setIsLoggedIn() state is getting called everytime the component is refreshed. Hence, in order to resolve this we use useEffect().



useEffect with dependencies ➖



setFormIsValid() needs to be called whenever the values of enteredEmail and enterPassword changes. So enteredEmail and enteredPassword are dependencies in this case.

**What to add & Not to add as Dependencies**

In the previous lecture, we explored useEffect() dependencies.

You learned, that you should add "everything" you use in the effect function as a dependency - i.e. all state variables and functions you use in there.

That is correct, but there are a **few exceptions** you should be aware of:

* You **DON'T need to add state updating functions** (as we did in the last lecture with setFormIsValid): React guarantees that those functions never change, hence you don't need to add them as dependencies (you could though)
* You also **DON'T need to add "built-in" APIs or functions** like fetch(), localStorage etc (functions and features built-into the browser and hence available globally): These browser APIs / global functions are not related to the React component render cycle and they also never change
* You also **DON'T need to add variables or functions** you might've **defined OUTSIDE of your components** (e.g. if you create a new helper function in a separate file): Such functions or variables also are not created inside of a component function and hence changing them won't affect your components (components won't be re-evaluated if such variables or functions change and vice-versa)

So long story short: You must add all "things" you use in your effect function **if those "things" could change because your component (or some parent component) re-rendered.** That's why variables or state defined in component functions, props or functions defined in component functions have to be added as dependencies!

Here's a made-up dummy example to further clarify the above-mentioned scenarios:

* import { useEffect, useState } from 'react';
* let myTimer;
* const MyComponent = (props) => {
* const [timerIsActive, setTimerIsActive] = useState(false);
* const { timerDuration } = props; // using destructuring to pull out specific props values
* useEffect(() => {
* if (!timerIsActive) {
* setTimerIsActive(true);
* myTimer = setTimeout(() => {
* setTimerIsActive(false);
* }, timerDuration);
* }
* }, [timerIsActive, timerDuration]);
* };

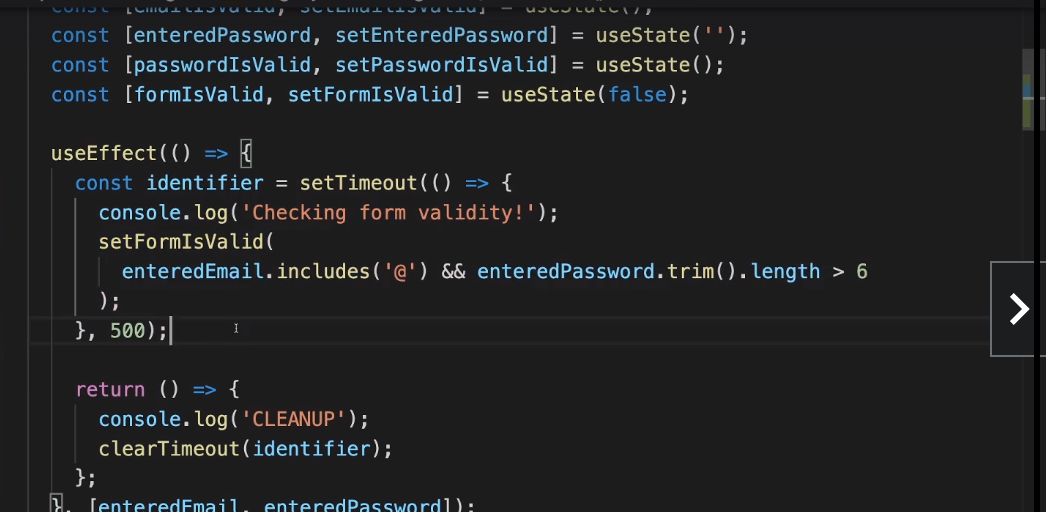
In this example:

* timerIsActive is **added as a dependency** because it's component state that may change when the component changes (e.g. because the state was updated)
* timerDuration is **added as a dependency** because it's a prop value of that component - so it may change if a parent component changes that value (causing this MyComponent component to re-render as well)
* setTimerIsActive is **NOT added as a dependency** because it's that **exception**: State updating functions could be added but don't have to be added since React guarantees that the functions themselves never change
* myTimer is **NOT added as a dependency** because it's **not a component-internal variable** (i.e. not some state or a prop value) - it's defined outside of the component and changing it (no matter where) **wouldn't cause the component to be re-evaluated**
* setTimeout is **NOT added as a dependency** because it's **a built-in API** (built-into the browser) - it's independent from React and your components, it doesn't change

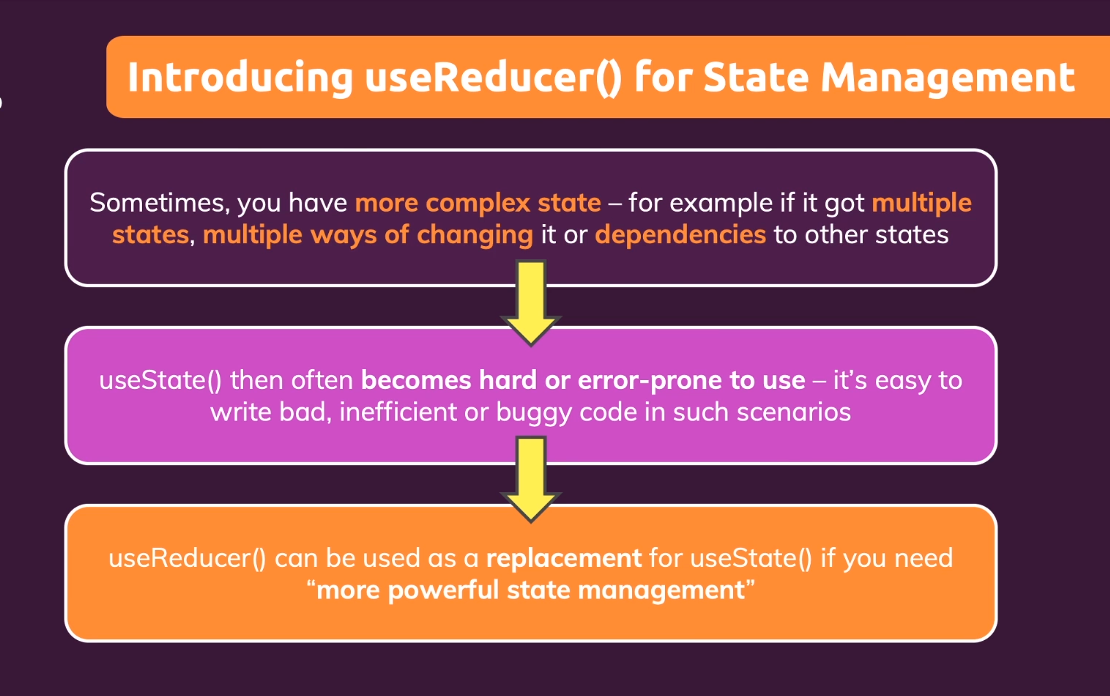
Debouncing can be used to improve the performance and avoid unnecessary re-rendering of the component.

useEffect also has something called cleanup function which gets called after the first time useEffects runs in the browser.

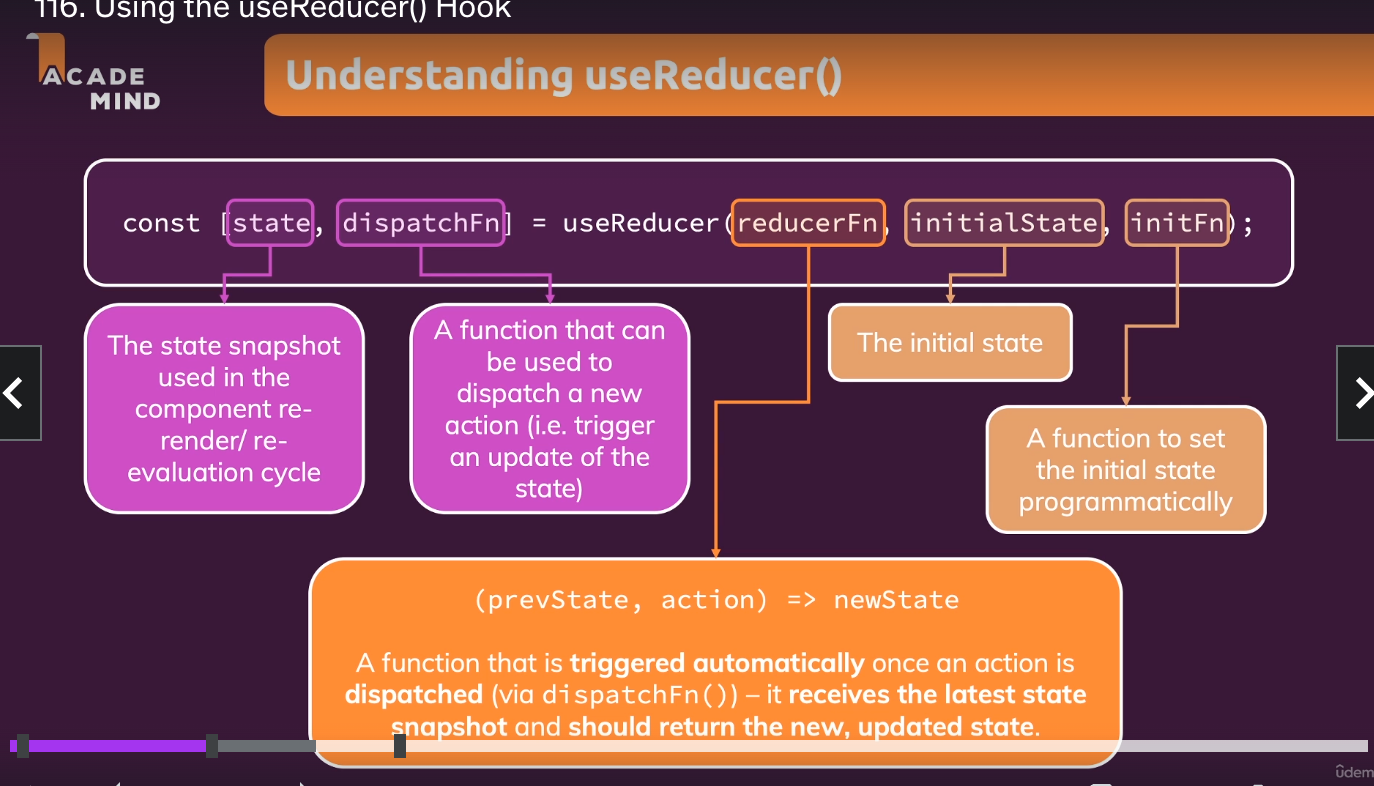
Following example explains the use of clean up function.



useReducer() ➖



Updating a state which depends on another state .



For example if we want to keep a track of different states of email validations we can use useReducer as below:-

emailReducer function should be outside the component function.

Const emailReducer = (state, action) => {

if(action.type===’USER\_INPUT’){

Return {value: action.val, isValid: action.val.includes(‘@’)};

}

Return {value: ‘’, isValid:false}

f(action.type===’INPUT\_BLUR’){

Return {value: state.value, isValid: state.value.includes(‘@’)};

}

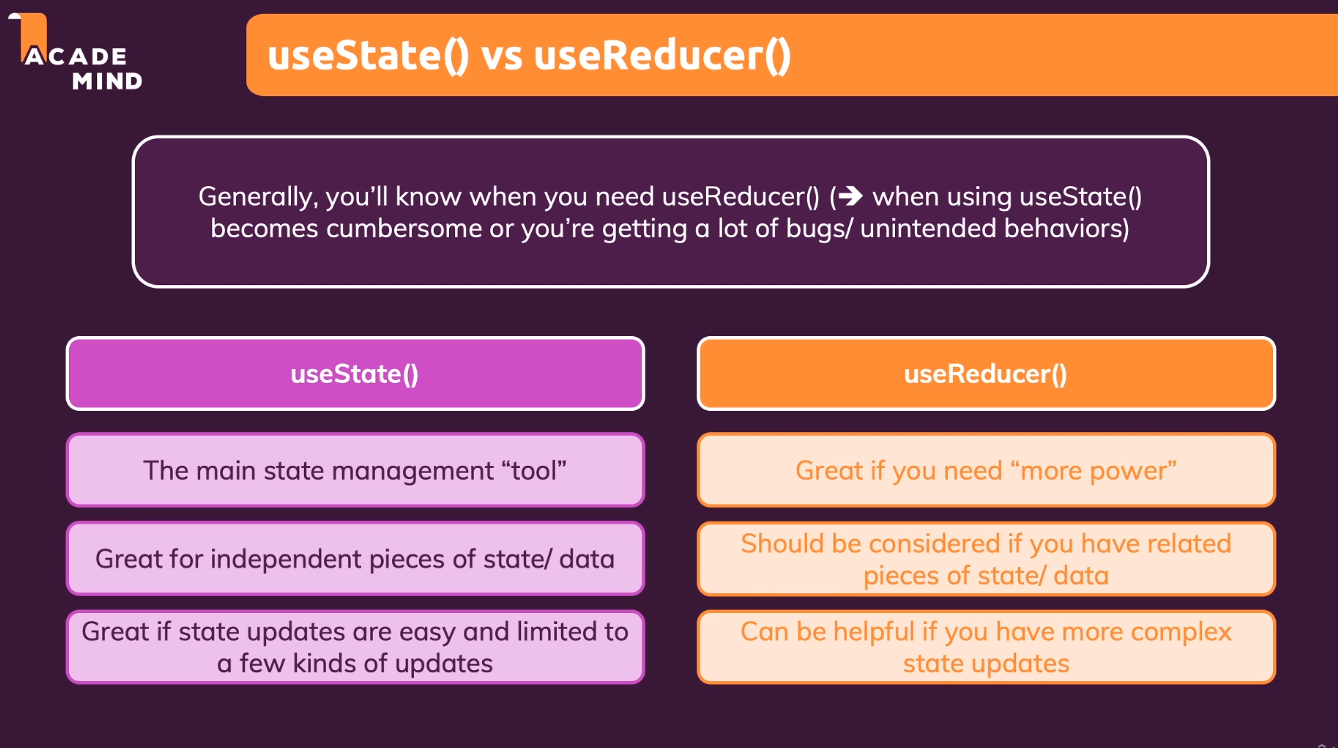
Return {value: ‘’, isValid:false}

}

Const emailChangeHandler = (event) => {

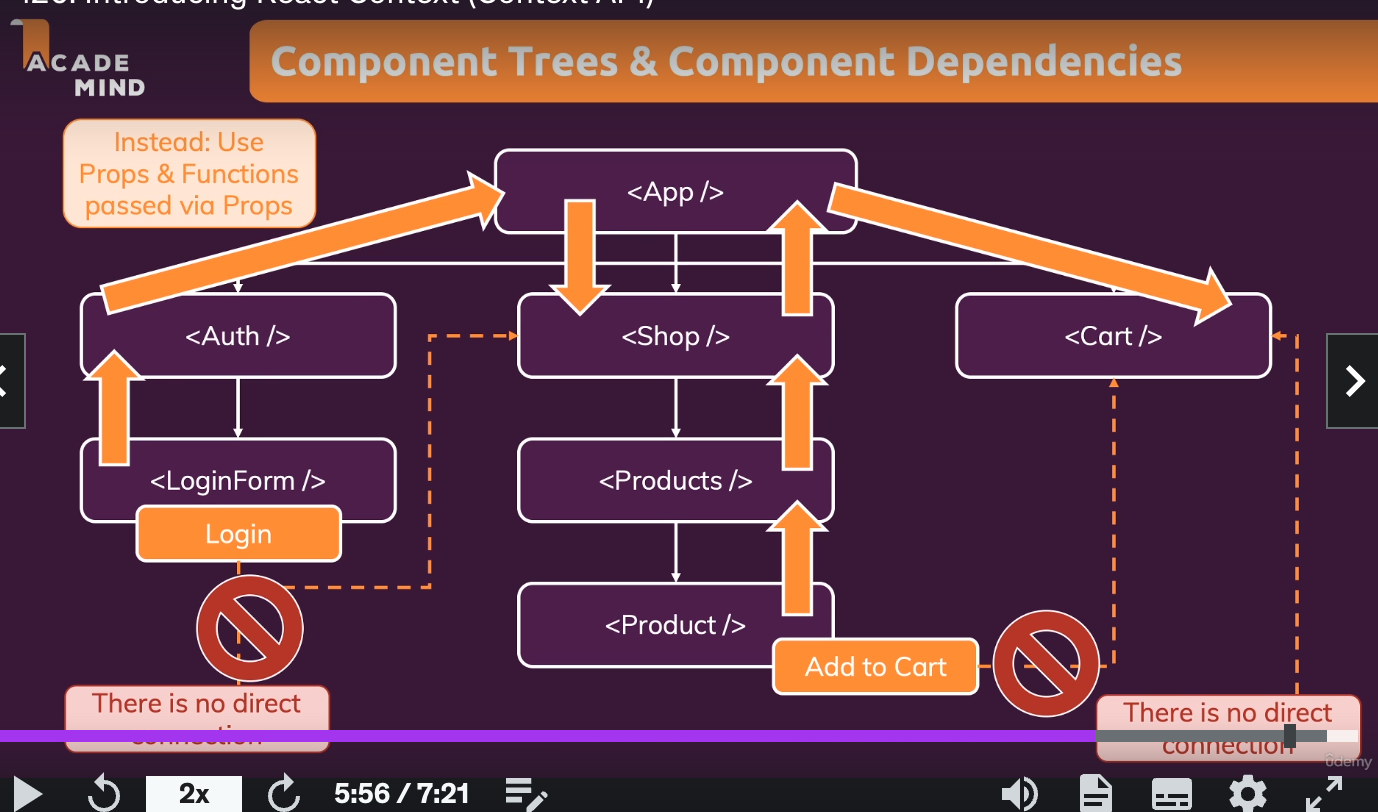
dispatchEmail({type:’USER\_INPUT’, val: event.target.value}); }

Const [emailState, dispatchEmail] = useReducer(emailReducer);



**Introduction React Context API :-**

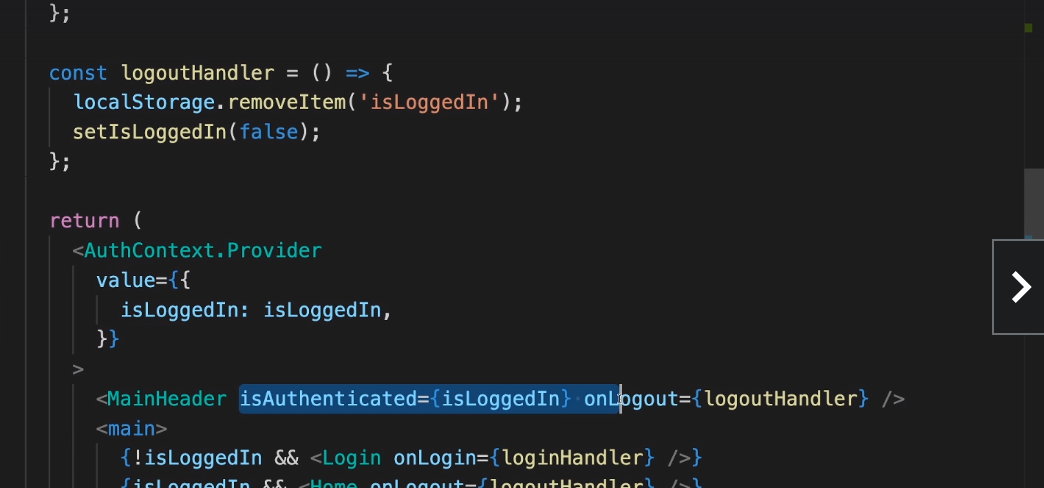
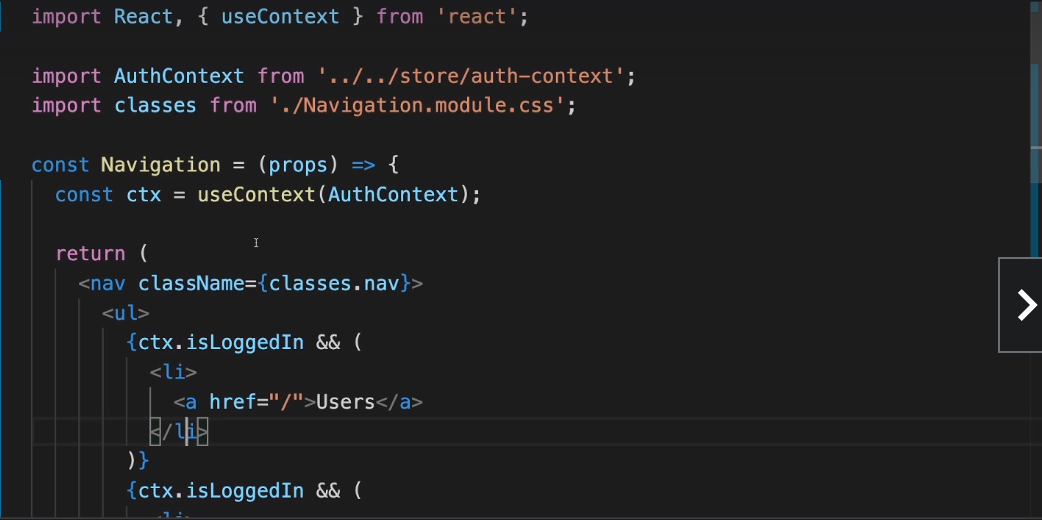
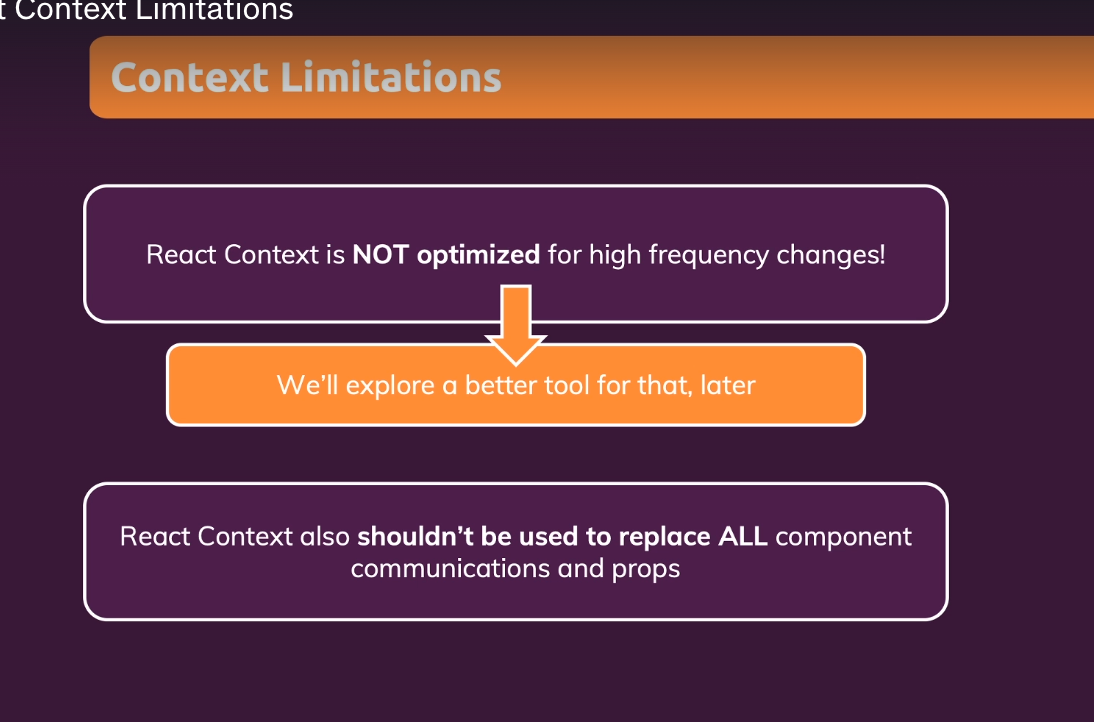
Bigger problem needs to be addressed in these scenarios.



Passing data between components is difficult if we have a nested tree of components.

React context is used to share props and functions without building such a prop chain.

Steps to use context API:-

1. Step one create a folder with any name usually store and create a context.js file ( like auth-context.js)
2. Example code for context file can be as follows. In the below example we want to manage isLoggedIn state throughout the application.
3. Wrap a provider wherever we want to listen to AuthContext. We also need to pass value to the provider.
4. 
5. Now we need to listen to the provider. We can do that using a hook or using <AuthContext.Consumer>
6. We will use react hook called useContext() hook if we want to consume the isLoggedIn State.
7. 
8. We can also pass functions to value ={} in provider.
9. We can import all the state management and related items to a context file which will enable state management.
10. 
11. Auth context provider sample code is as follows:-

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

const AuthContext = React.createContext({

isLoggedIn: false,

onLogout: () => {},

onLogin: (email, password) => {}

});

export const AuthContextProvider = (props) => {

const [isLoggedIn, setIsLoggedIn] = useState(false);

useEffect(() => {

const storedUserLoggedInInformation = localStorage.getItem('isLoggedIn');

if (storedUserLoggedInInformation === '1') {

setIsLoggedIn(true);

}

}, []);

const logoutHandler = () => {

localStorage.removeItem('isLoggedIn');

setIsLoggedIn(false);

};

const loginHandler = () => {

localStorage.setItem('isLoggedIn', '1');

setIsLoggedIn(true);

};

return (

<AuthContext.Provider

value={{

isLoggedIn: isLoggedIn,

onLogout: logoutHandler,

onLogin: loginHandler,

}}

>

{props.children}

</AuthContext.Provider>

);

};

export default AuthContext;

1. Ref can be passed to custom components using forwardRefs.