**What is React?**

React is a **client-side javascript library for building user interfaces**. React makes building user interfaces simpler.

Alternatives: Angular.js, Vue.js

**Module 1: React Basics and Working with Components**

1) React is all about **building reusable components** (building blocks) of User Interface. When we open our react app, the code in the index.js file is executed by default. Using react, we can create our own custom HTML Elements.

2) To get started with creating react app, refer this: <https://reactjs.org/docs/create-a-new-react-app.html>

In react projects, in index.js file, these 2 lines are imported by default

import React from 'react';

import ReactDOM from 'react-dom/client';

Refer this: <https://stackoverflow.com/questions/34114350/react-vs-reactdom>

React and ReactDOM were only recently split into two different libraries. Prior to v0.14, all ReactDOM functionality was part of React. This may be a source of confusion, since any slightly dated documentation won't mention the React / ReactDOM distinction.

As the name implies, ReactDOM is the glue between React and the DOM. Often, you will only use it for one single thing: mounting with ReactDOM.render(). Another useful feature of ReactDOM is ReactDOM.findDOMNode() which you can use to gain direct access to a DOM element. (Something you should use sparingly in React apps, but it can be necessary.) If your app is "isomorphic", you would also use ReactDOM.renderToString() in your back-end code.

For everything else, there's React. You use React to define and create your elements, for lifecycle hooks, etc. i.e. the guts of a React application.

The reason React and ReactDOM were split into two libraries was due to the arrival of React Native. React contains functionality utilised in web and mobile apps. ReactDOM functionality is utilised only in web apps. [**UPDATE:** Upon further research, it's clear my ignorance of React Native is showing. Having the React package common to both web and mobile appears to be more of an aspiration than a reality right now. React Native is at present an entirely different package.]

3) When we create a react app, 3 folders are created: node\_modules (which consists of dependencies from the package.json file), public (which consists of static images etc) and a src file which consists of index.js file and app.js file

**Index.js**

import ReactDOM from 'react-dom/client';

// importing css would be an invalid syntax in plain javascript

import './index.css';

import App from './App';

//create a root object from reactDOM library

const root = ReactDOM.createRoot(document.getElementById('root'));

// attach the App Component inside the root component.

root.render(<App />);

**App.js**

import logo from './logo.svg';

import './App.css';

function App() {

  return (

    <div>

          Learn React

    </div>

  );

}

export default App;

4) Components are nothing but a function which may have some JS code and returns JSX code (Similar to HTML code). (Not necessarily JSX code…we can also use the React.createElement function. Actually, behind the scenes, React converts the JSX code to the createElement function). **JSX IS NOT HTML and JSX stands for JavaScript XML.**

**NOTE: React is declarative, which means we define the desired outcome (e.g. a target UI, in JSX in return statement of the component) and let the library (React) figure out the steps.**

**NOTE: We can access DOM APIs too. Document.getElementByID, getElementByClassName etc. but we should not use the DOM specific APIs. Use refs to get access to a specific dom element. (Don’t use id in react because when we use multiple components, then there will be same id in all those components. So use refs)**

**App.js**

import logo from './logo.svg';

import './App.css';

function App() {

  //we are returning JSX Code (Javascript XML Code). This is NOT HTML Code

  //react converts this JSX code into HTML code internally and then sends it to the browser.

  return (

    <p>

       Learn React

    </p>

  );

}

export default App;

5) Good practice: Inside the src folder: create a components folder for all the components (js and css files). This component folder will have multiple components. And also create an UI Folder which will contain common buttons, inputs, cards etc.

6) Accessing JS Code from JSX

function ExpenseListItemDate(){

    const expenseDate = new Date(2021, 2, 28);

    return (

        <div className="ExpenseListItemDate">

            <p className="ExpenseListItemDateMonth">March</p>

            <p className="ExpenseListItemDateYear">{expenseDate.getFullYear()}</p>

            <p className="ExpenseListItemDateDay">12</p>

        </div>

    )

}

4) Using props, we pass data from one component to another.

Passing the data from ExpenseList component to ExpenseListItem

import '../css/ExpenseList.css';

import ExpenseListItem from './ExpenseListItem';

function ExpenseList(){

    const expenses = [

        {

            month: 'March',

            year: Math.floor(Math.random() \* 5 + 2015),

            day: Math.floor(Math.random() \* 29),

            expenseItem: 'New TV',

            expensePrice: Math.random() \* 100 + 500;

        },

        {

            month: 'January',

            year: Math.floor(Math.random() \* 5 + 2015),

            day: Math.floor(Math.random() \* 29),

            expenseItem: 'New Laptop',

            expensePrice: Math.random() \* 100 + 500;

        },

        {

            month: 'September',

            year: Math.floor(Math.random() \* 5 + 2015),

            day: Math.floor(Math.random() \* 29),

            expenseItem: 'New Car',

            expensePrice: Math.random() \* 100 + 500;

        },

        {

            month: 'April',

            year: Math.floor(Math.random() \* 5 + 2015),

            day: Math.floor(Math.random() \* 29),

            expenseItem: 'Dinner',

            expensePrice: Math.random() \* 100 + 500;

        },

    ]

    //pass the expenses to expenseListItem component using props

    return (

        <div className="ExpenseList">

            <ExpenseListItem expenseItem = {expenses[0]}/>

            <ExpenseListItem expenseItem = {expenses[1]}/>

            <ExpenseListItem expenseItem = {expenses[2]}/>

            <ExpenseListItem expenseItem = {expenses[3]}/>

        </div>

    )

}

export default ExpenseList;

Accessing the data in ExpenseListItem:

import '../css/ExpenseListItem.css';

import ExpenseListItemDate from './ExpenseListItemDate';

import ExpenseListItemName from './ExpenseListItemName';

import ExpenseListItemPrice from './ExpenseListItemPrice';

function ExpenseListItem(props){

    return (

        <div className="ExpenseListItem">

            <div className="ExpenseListItemDateAndName">

                <ExpenseListItemDate day = {props.day} month = {props.month} year = {props.year}/>

                <ExpenseListItemName name = {props.expenseItem}/>

            </div>

            <ExpenseListItemPrice price = {props.expensePrice}/>

        </div>

    )

}

export default ExpenseListItem;

5) We can also make a custom html component (to use as a common wrapper), but to display elements inside the custom component, we would have to use special keyword: {props.children} and to use the className, we would have to use {props.className} in the component.js file.

Eg: card.js

function Card(props){

    return (

        <div className = {props.className + ' card'}>

            {props.children}

        </div>

    )

}

ExpenseListItem.js

function ExpenseListItem(props) {

    return (

        <Card className="ExpenseListItem">

            <div className="ExpenseListItemDateAndName">

                <ExpenseListItemDate day={props.expenseItem.day} month={props.expenseItem.month} year={props.expenseItem.year} />

                <ExpenseListItemName name={props.expenseItem.expenseItem} />

            </div>

            <ExpenseListItemPrice price={props.expenseItem.expensePrice} />

        </Card>

    )

}

6) The JSX code is then converted into the code that the browsers understand with the help of React Library. But in the modern projects, we need not import react for executing JSX code.

7) The reason for using a single wrapper in the return statement of JSX code is that when we run this code, the react library runs React.createElement(tagName, attributes, contentsOfTheTag) method. So, It must have a single wrapper. Thus the JSX code is not necessary but for making more readable code, we use JSX

import '../css/ExpenseAdder.css';

import React from 'react';

function ExpenseAdder(){

    return (

        <div className = "ExpenseAdder">

            <button className = "ExpenseAdderBtn">Add Expense</button>

        </div>

    )

    // The above return statement is equivalent to

    return React.createElement('div',

    {className: 'ExpenseAdder'},

    React.createElement('button', {className: 'ExpenseAdderBtn'}, 'Add Expense'));

}

export default ExpenseAdder;

8) Organising files: We create a separate folder UI which contains the general UI components which are not connected to specific features of the app like the card component we created for the expenses project.

**Module 2: React State and working with events**

1: Whenever you are writing functions that are triggered upon an event, that function’s name with “Handler”. (Eg: If you are writing a function that is triggered by the onclick event of a button, you can name the function clickHandler) **onClick** (This type of function should start with “on”) **= {clickHandler}**

**onShowCart = {showCartHandler}, onHideCart = {hideCartHandler} etc.,**

2: Whenever we run our react code, react runs all the component functions and it does not re-run the component codes again whenever an event happens like the button click event . So to update our page, we need a concept called state.

3: To call the component function again when we want to display our updated page, we use useState() method from the react library. The useState() function must be inside the main component function and it must not be inside the nested functions. The useState function takes an argument. The argument can be anything which we want to update. The useState function then returns an array of 2 elements. First element being the value of the argument which we want to change and the 2nd element is a function which updates the value and calls the component function again and so our page becomes updated.

import '../css/ExpenseAdder.css';

import React, {useState} from 'react';

function ExpenseAdder(){

    const [buttonText, setButtonText] = useState('Add Expense');

    const clickHandler = () => {

        if(buttonText === 'Add Expense')

            setButtonText('Delete Expense');

        else setButtonText('Add Expense');

        // button text is not updated yet. This will print the current buttonText

        // and not the updated button text

        console.log(buttonText);

        // after executing all the statements, React renders the component again

        // setButtonText is an asynchronous function which DOES NOT return a promise

    }

    return (

        <div className = "ExpenseAdder">

            <button onClick = {clickHandler} className = "ExpenseAdderBtn">{buttonText}</button>

        </div>

    )

}

export default ExpenseAdder;

4: What happens if we put multiple setState calls in our component? Read: <https://stackoverflow.com/questions/33613728/what-happens-when-using-this-setstate-multiple-times-in-react-component>.

React batches state updates that occur in event handlers and lifecycle methods. Thus, if you update state multiple times in a <div onClick /> handler, React will wait for event handling to finish before re-rendering.

**React batches all the synchronous state updates into one big update for performance optimizations. That is why react state update is not simultaneous.**

To be clear, this only works in React-controlled synthetic event handlers and lifecycle methods. State updates are not batched in AJAX and setTimeout event handlers, for example.

import '../css/ExpenseAdder.css';

import React, {useState} from 'react';

function ExpenseAdder(){

    console.log("Rendering...");

    const [buttonText, setButtonText] = useState('Add Expense');

    const clickHandler = () => {

        if(buttonText === 'Add Expense')

            setButtonText('Delete Expense');

        else setButtonText('Add Expense');

        // button text is not updated yet. This will print the current buttonText

        // and not the updated button text

        console.log(buttonText);

        // after executing all the statements, React renders the component again

        // setButtonText is an asynchronous function which DOES NOT return a promise

    }

    setTimeout(() => {

        setButtonText('Timeout');

    }, 10000)

    return (

        <div className = "ExpenseAdder">

            <button onClick = {clickHandler} className = "ExpenseAdderBtn">{buttonText}</button>

        </div>

    )

}

In the above example, the setButtonText() updates in the clickHandler and setTimeout function is not batched together.

function ExpenseAdder() {

    console.log("Rendering...");

    const [buttonText, setButtonText] = useState(1);

    const clickHandler = () => {

        // buttonText = 1

        setButtonText(buttonText + 1); // setButtonText(2);

        setButtonText(buttonText + 1); // buttonText is not updated yet.So setButtonText(2)

// button text is not updated yet. This will print the current buttonText

        // and not the updated button text

        console.log(buttonText); // prints 1

        // after executing all the statements, React renders the component again

        // setButtonText is an asynchronous function which DOES NOT return a promise

    }

    return (

        <div className="ExpenseAdder">

            <button onClick={clickHandler} className="ExpenseAdderBtn">{buttonText}</button>       </div>

    )

}

But if we use callbacks in setState() method…refer below code (**NOTE: Whenever our state depends on prev state, use callbacks inside the setState method which passes the prev state parameter**)

import '../css/ExpenseAdder.css';

import React, { useState } from 'react';

function ExpenseAdder() {

    console.log("Rendering...");

    const [buttonText, setButtonText] = useState(1);

    const clickHandler = () => {

        // buttonText = 1

        setButtonText(buttonText => buttonText + 1) //  update buttonText = 1 + 1

        setButtonText(buttonText => buttonText + 1) //  update buttonText = 2 + 1

        // React puts the updater functions in a queue.

        // Then, during the next render, it will call them in the same order:

        // a => a + 1 will receive 42 as the pending state and return 43 as the next state.

        // a => a + 1 will receive 43 as the pending state and return 44 as the next state.

        // button text is not updated yet. This will print the current buttonText

        // and not the updated button text

        console.log(buttonText); //prints 1

        // after executing all the statements, React renders the component again

        // setButtonText is an asynchronous function which DOES NOT return a promise

    }

    return (

        <div className="ExpenseAdder">

            <button onClick={clickHandler} className="ExpenseAdderBtn">{buttonText}</button>

        </div>

    )

}

export default ExpenseAdder;

**React puts your updater functions in a queue. Then, during the next render, it will call them in the same order:**

**a => a + 1 will receive 42 as the pending state and return 43 as the next state.**

**a => a + 1 will receive 43 as the pending state and return 44 as the next state.**

Refer: <https://stackoverflow.com/questions/50837670/reactjs-setstate-previous-state-is-the-first-argument-props-as-the-second-argum>

5. Why are we using const variables for the setState method when the value is being updated? We see that when we use the useState method, the component function is CALLED again, and then the useState method assigns the updated value to the new const variable. (Read more about state). The variable returned by the useState method is a STATE variable and normally variables disappear after function ends but state variable is preserved by the react.

6: Note that the setState method is asynchronous. React intentionally “waits” until all components call setState() in their event handlers before starting to re-render. This boosts performance by avoiding unnecessary re-renders. React batches (combines) all the state updates to avoid unnecessary re-render.

7: IMPORTANT RULE: Whenever our state depends on prev state, use callbacks inside the setState method which passes the prev state parameter. The reason for this is that React schedules state updates, and if we change states many times, we could be depending on an outdated state or an incorrect state.

8: Passing data from child to parent. Normally we transfer data from parent component to child component using props. But to send data in the reverse direction, we use the help of a function we write in the parent component and call that function in the child component using props and pass in the data as an argument to that function. (THIS IS CALLED “LIFTING THE STATE UP”)

**Module 3: Rendering lists and conditional content**

1: React is capable of rendering an array of JSX elements. Eg: In your return statement of functional components, the following will work

return (<div>{[<h1>Hello World1<h1/>, <h2>Hello World2<h2/>]}</div>)

This is same as

return(

<div>

<h1>Hello World1<h1/>

<h2>Hello World2<h2/>

</div>

)

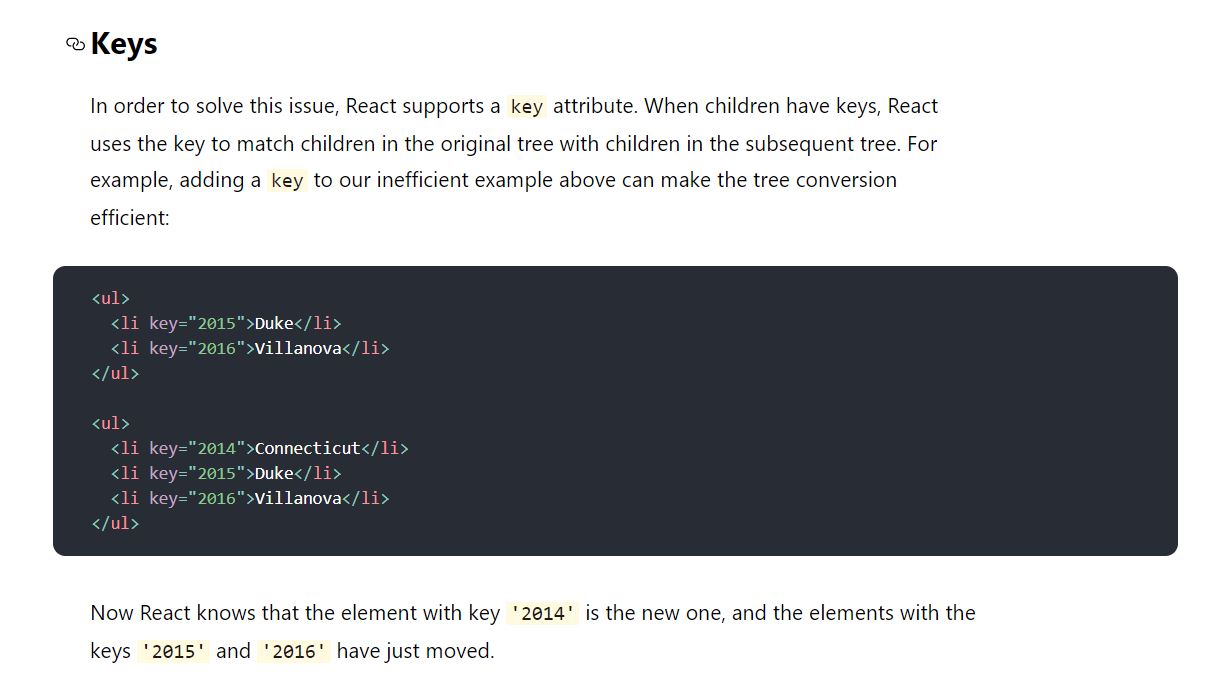
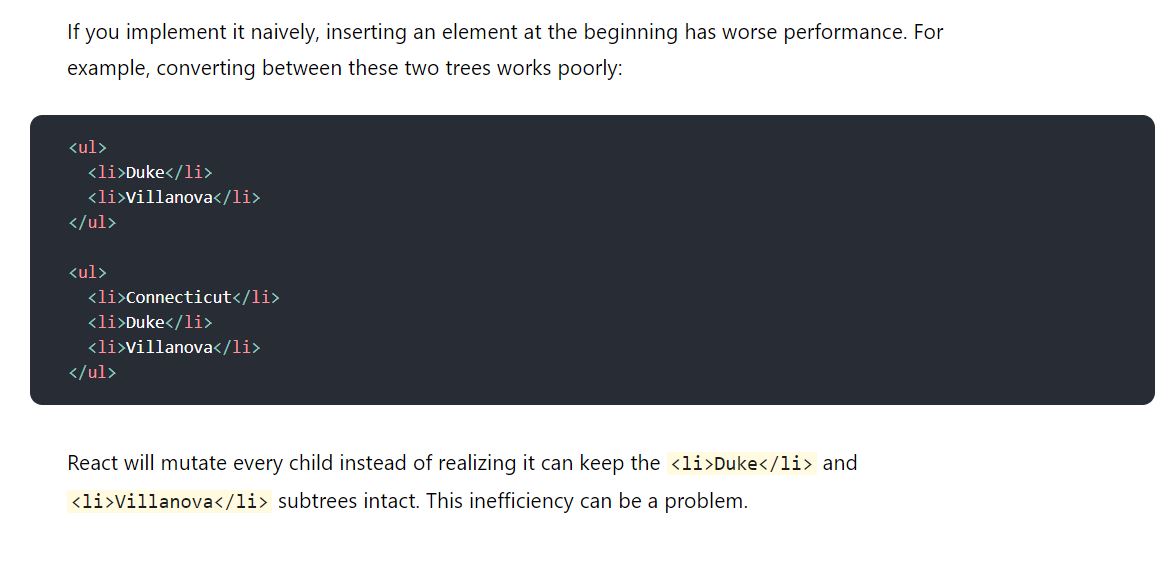
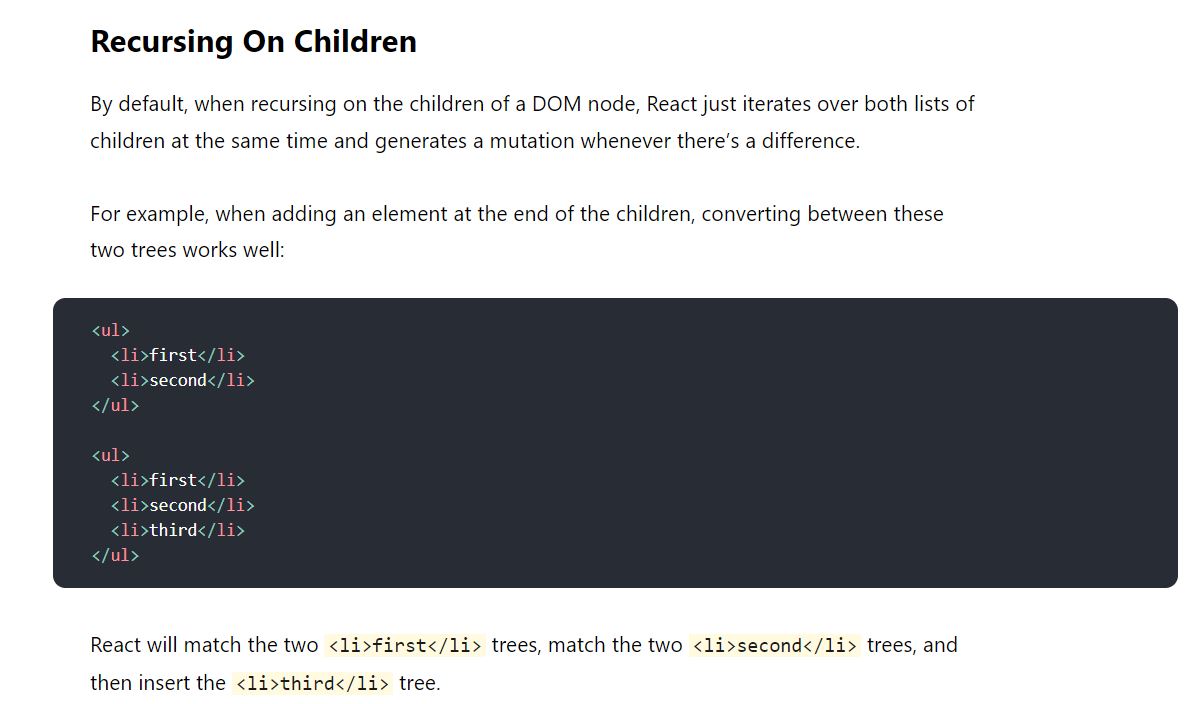
2: So if we need to display a dynamic array of objects, we can use array.map() method in the JSX code which takes in a **CALLBACK** function and applies that function to every element of the array and returns an array

Eg: arr1 = [1, 2, 3, 4]

arr1.map((x) => {return x\*x;}) will return [1, 4, 9, 16]

3: We should always add a key to individual elements in an array if we are displaying that array to boost performance. we add this key because when we do not add keys, react compares the existing tree to the new tree element by element and if the elements are not equal, then the modification can be expensive. Having a key helps react to know which elements have been added, modified and removed. (Chapter 5: Video 3)

Refer: <https://reactjs.org/docs/lists-and-keys.html>, <https://reactjs.org/docs/reconciliation.html#recursing-on-children>)



4: We can also store JSX code in variables. (Chapter 5: Video 4)

5: Setting styles dynamically using the style prop. The style prop requires an javascript object which will override the default style properties

<div className = ‘chart-bar\_\_fill’, style = {{height: 20px}}> </div>. The height of 20px will override the height in the css file. (Chapter 5: Video 8)

6: Always specify the button type (submit or button or reset) to avoid unnecessary problems.

**Module 6: Styling react components**

USING IDs for the HTML elements is a bad idea because if we create multiple components, then all the components will have the same id. Use Classes

Notice that when we import a CSS file in the parent component(say App.js file), it is applied to all its child js files (if we add imports of a css file to app.js file, then that css file would be imported to all the js files). So there is a chance to use same classname which is there in the global css file and cause unnecessary problems.

So, there are 2 options to prevent this: Use inline styles (CSS in JS files) or use **CSS modules**. Both of these provide unique class names so there wont be any clash. **Under the hood, what CSS Modules does is, it converts the classes of the CSS file and converts them into unique class names so that there is no chance of clashing with other class of same name.**

**Refer:** [**https://www.javascriptstuff.com/what-are-css-modules/**](https://www.javascriptstuff.com/what-are-css-modules/)

To use CSS Modules:

1. Change your css filename to filename.module.css (Eg: button.css to button.module.css)
2. In your JS File where you want to use the styles, write the import statement:

**import styles from ‘path to button.module.css’;**

**Now styles is a JS object. We can extract classes using styles.yourClassName or styles[“yourClassName”]**

**Module 7: Debugging React Apps**

1: Always lookout for browser console warnings and analyse code flow in case of errors.

2: Use breakpoints for analysing. Go to the sources tab and in your code and simply click on that line of code where you want your code to stop execution (You can also select multiple break points). Use features like step into next function and other features to debug.

3: Use react developer tools extension in the browser.

**Module 8: Practice Project**

1: A modal is a message box that is displayed on top of your screen. Modals put an overlay on the screen; therefore, they take visual precedence over all the other elements. (DO THE PRACTICE PROJECT)

**Module 9: Fragments Portals and Refs**

1: In bigger apps, we end up with tons of unnecessary divs (as wrappers) which add no meaning to the page just because we have to enclose our JSX code with a wrapper. (Video 2)

2: To solve this problem of reducing unnecessary wrapping tags, we create a component wrapper.js (Normally we create a Helpers folder in the components folder and we create wrapper.js file in that folder) and in that we return props.children as shown below (Video 3)

const Wrapper = (props) => {

    return props.children;

}

export default Wrapper;

Now we use this Wrapper component in place of unnecessary wrapping div tags so that there will be no unnecessary div tags in the html file. **This solves the problem of avoiding unnecessary enclosing tags.**

Now we need not do this in react, React also has empty wrapper component as shown below

return (

    <React.Fragment>

        <h2>Hi there!</h2>

        <p>This Works!<p>

    </React.Fragment>

)

This above approach will always work

return (

    <>

        <h2>Hi there!</h2>

        <p>This Works!<p>

    </>

)

The above may or may not work depending on the project setup.

So instead of unnecessary divs as wrappers, we use fragments or <>. (Video 4)

3: We use Portals to move our html code somewhere else (Functionality wise everything remains the same with or without portals) But if we use portals, we come up with more **semantically correct html code** (Refer: <https://reactjs.org/docs/portals.html>) .

// let's say the beloe jsx code shows the error popup when there is an invalid input.

// and the error popup spans the entire width and height of the browser. Then it makes sense

// to insert the errorModal near the root element in the DOM (functioning wise everything remains the same):

// in such cases, to move the error modal, we use portals

return (

    <React.Fragment>

        {!validInput && <ErrorModal/>}

        <inputForm/>

    </React.Fragment>

)

// create an element with id as error-modal near root first

return (

    <React.Fragment>

        {!validInput && ReactDOM.createPortal(<ErrorModal/>, document.getElementById('error-modal'))}

        <inputForm/>

    </React.Fragment>

)

4: We connect our html elements with Javascript code with useRef hook. But We should not (not CANNOT) manipulate the DOM using useRef.

Why should we use refs? We can get the job done using document.getElementById() right?

Refer this: <https://www.javascriptstuff.com/use-refs-not-ids/>

Using ids in react app is not a good idea because the idea of react is to create reusable components and we create multiple components, then every component will have the same id

Eg:

import '../css/ExpenseListItemPrice.css';

function ExpenseListItemPrice(props){

    const priceRef = useRef();

    //when this component renders, then priceRef will point to the ACTUAL DOM node of the paragraph element

    return (

        <div className="expense-list-item-price">

            <p ref = {priceRef}>{props.price}$</p>

        </div>

    )

}

export default ExpenseListItemPrice;

USE REFS WHEN WE NEED TO READ A VALUE. When we were using the state, the setState method of that input field was called whenever we changed the input (Unnecessary Re-rendering). So instead of this we use refs to avoid such re-rendering. We can modify values using refs but it is advised not to do so.

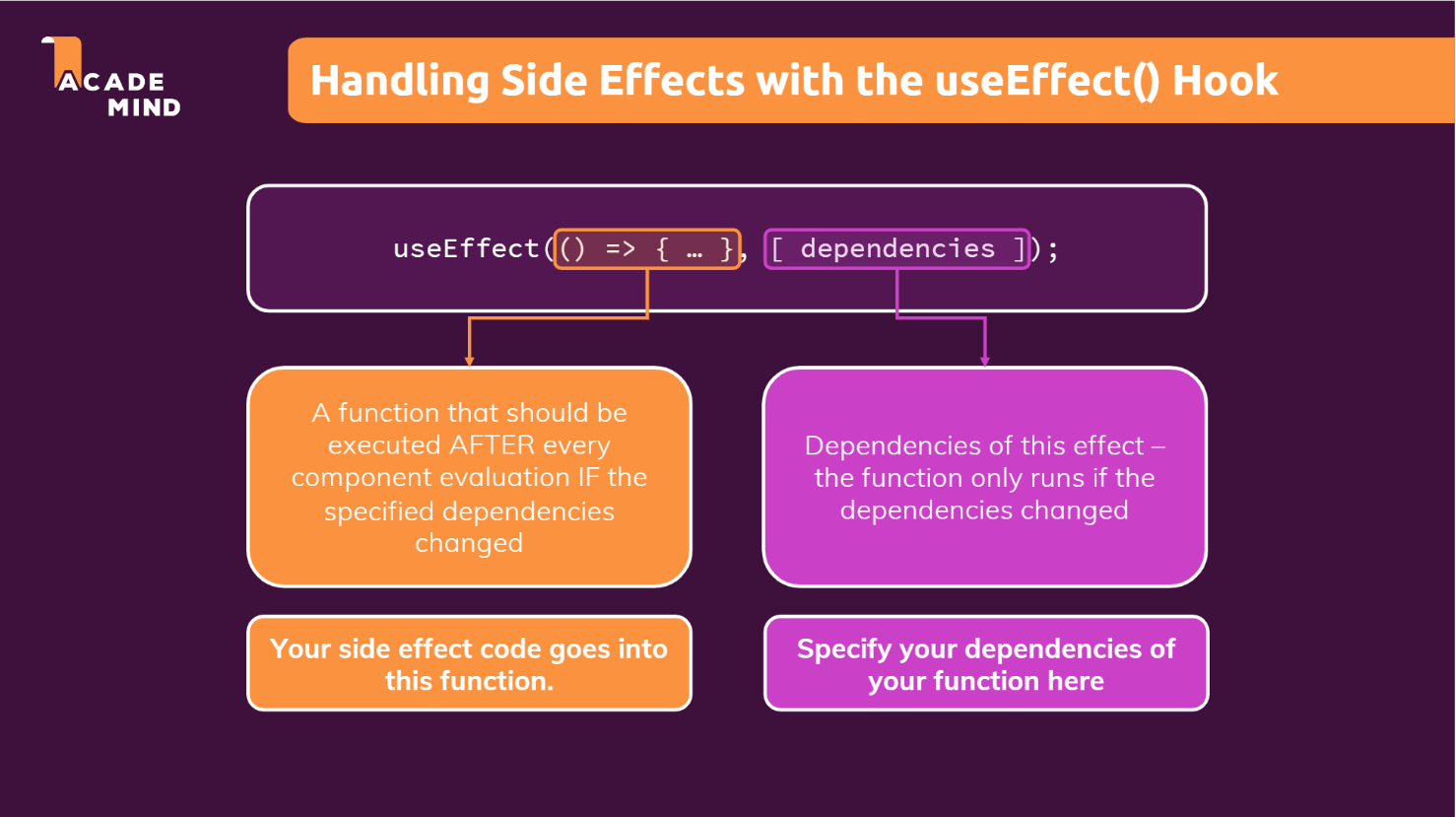
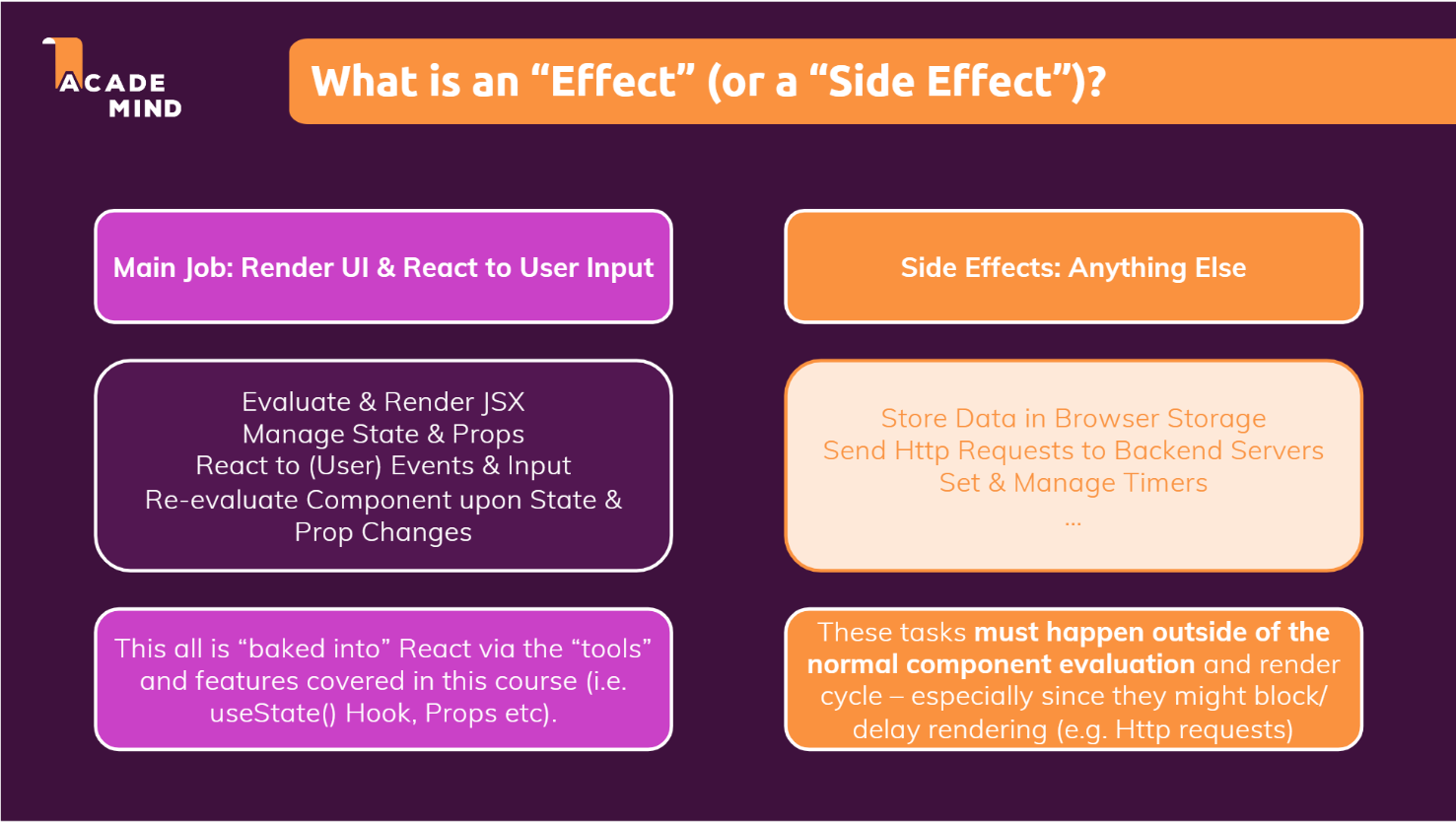
5: Controlled vs Uncontrolled Components:

In a controlled component, form data is handled by a React component (using state etc). The alternative is uncontrolled components, where form data is handled by the DOM itself (using refs or document.getElement api).

**Module 10 Advanced\_ Handling Side Effects, Using Reducers & Using the Context API**

1: When to use useEffect hook?: When we want some code to be executed **after the component renders** and **IF THE DEPENDENCIES SPECIFIED IN THE USEEFFECT CHANGES,** then we should use useEffect().

Suppose we have some api call in our component. Now everytime the component renders, a call to api would be made which might be undesirable. In such cases, we can use useEffect hook.



Notice the below code snippet. When we are logging in for the first time, we are storing data in the browser so that when we click the refresh button, we don’t log out. But the code goes in infinite loop when we are logged in, because storedUserInfo will be always 1 and this will cause infinite renders.

function App() {

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

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

  if(storedUserInfo === '1'){

    setIsLoggedIn(true);

  }

  const loginHandler = (email, password) => {

    // We should of course check email and password

    // But it's just a dummy/ demo anyways

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

    setIsLoggedIn(true);

  };

//other code

//return JSX code

}

This is where useEffect hook helps. useEffect hook takes in 2 parameters, 1st one is the function where we write our side effect code (For eg: Fetching data from the database) and 2nd one is array of dependencies. The side effect code will only run when our app starts for the first time and during re-render if our dependencies change.

**NOTE**: By using this Hook, you tell React that your component needs to do something after render. React will remember the function you passed (we’ll refer to it as our “effect”), and call it later after performing the DOM updates.

3: The 2nd parameter in the useEffect hook is the array of dependencies. The code on the useEffect hook will run when any one of the dependency changes. How to identify the dependencies? Write the dependencies according to the code that we have inside the callback of useEffect.

4:

useEffect(() => {

setFormIsValid(

        enteredEmail.includes('@') && enteredPassword.trim().length > 6

      )

    }, [enteredEmail, enteredPassword]);

Consider this above snippet. The setState method (setFormIsValid) is called everytime whenever one of the dependency change. Say instead of setFormIsValid method, we are sending http requests. Now we would be sending a lot of http requests because the state is changed upon every keystrokes (enteredEmail and enteredPassword are state variables here). This is not desirable. So we use a setTimeout function (which is an async function) which takes in 2 parameters (one is the code we want to execute and the other parameter is the time in milli seconds after which the function would be executed.

useEffect(() = > {

setTimeout(() => {

      setFormIsValid(

        enteredEmail.includes('@') && enteredPassword.trim().length > 6

  )}, 500);

}, [enteredEmail, enteredPassword]);

Now the setTimeout function will wait for 500 ms when one of the dependency change.

Now in useEffect method, we can return a function (we call this cleanup function) which is executed before any of the code in the useEffect method is executed and it will not be executed for the first time when the useEffect hook is executed. The cleanup function is also executed when the component is removed but the side effect code will not be executed.

So now we can clear out timer function in the cleanup function

We can say that the cleanup function contains the previous state of the side-effect code. We are clearing the previous timer in the cleanup function as shown in the code below.

  useEffect(() => {

//identifier stores the unique id of the timer. We use this id in the //clearTimeout function

    const identifier = setTimeout(() => {

      console.log("inside");

      setFormIsValid(

        enteredEmail.includes('@') && enteredPassword.trim().length > 6

      )

    }, 500);

    return () => {

      console.log("CLEANUP");

      clearTimeout(identifier); //clear the previous timer

    }

  }, [enteredEmail, enteredPassword]);

(VIDEO 6)

5: **If the dependency array is empty, the useEffect function will ONLY run the first time when the component renders and will NOT run again. And if we do not provide any dependencies, then the code in useEffect will run everytime when the component renders.**

6: Now consider this

 const [enteredEmail, setEnteredEmail] = useState('');

const [enteredPassword, setEnteredPassword] = useState('');

const [formIsValid, setFormIsValid] = useState(false);

useEffect(() => {

    const identifier = setTimeout(() => {

      setFormIsValid(

        enteredEmail.includes('@') && enteredPassword.trim().length > 6

      )

    }, 500);

    return () => {

      //clear out the previous timer

      clearTimeout(identifier);

    }

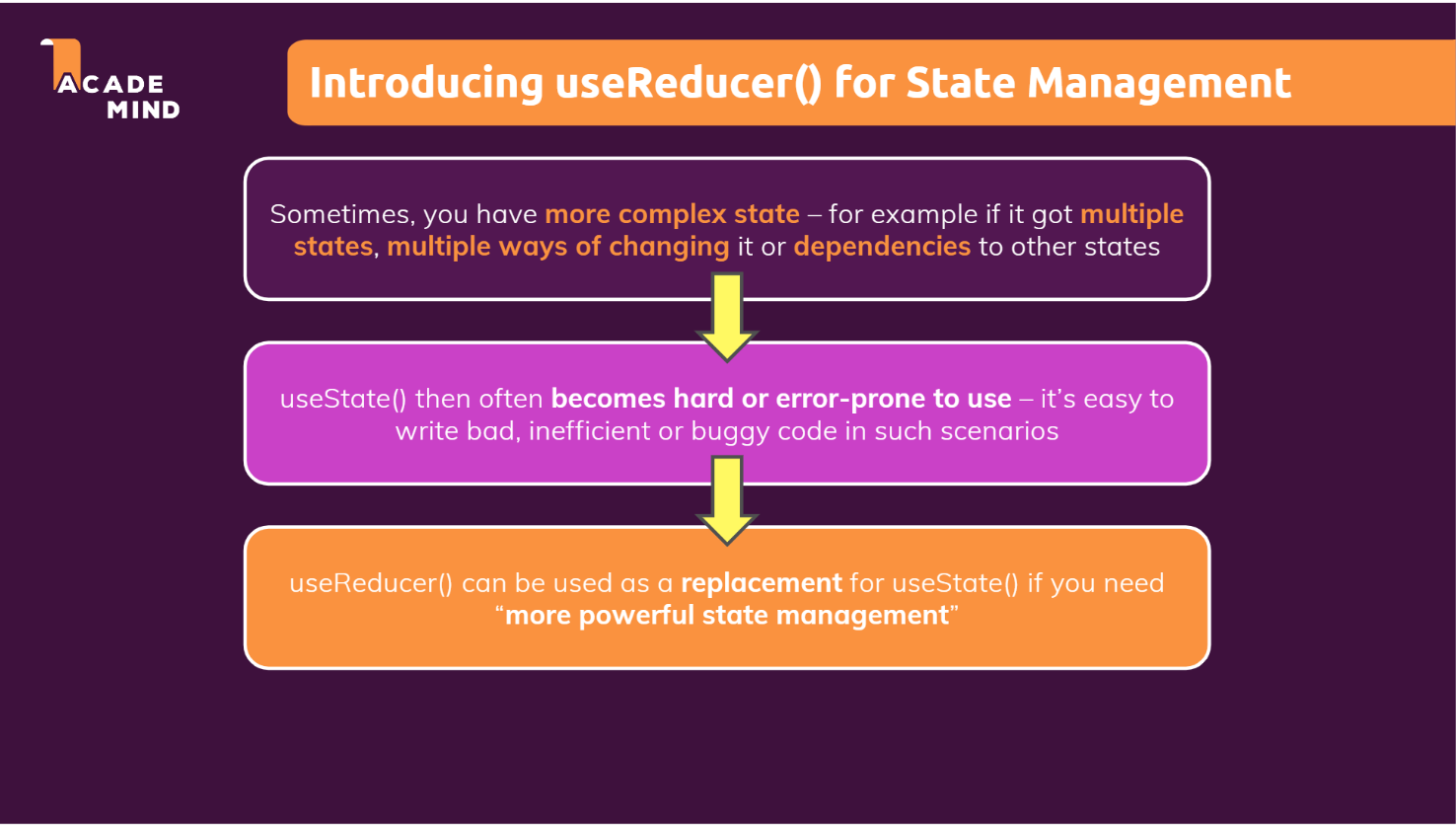
  }, [enteredEmail, enteredPassword]);

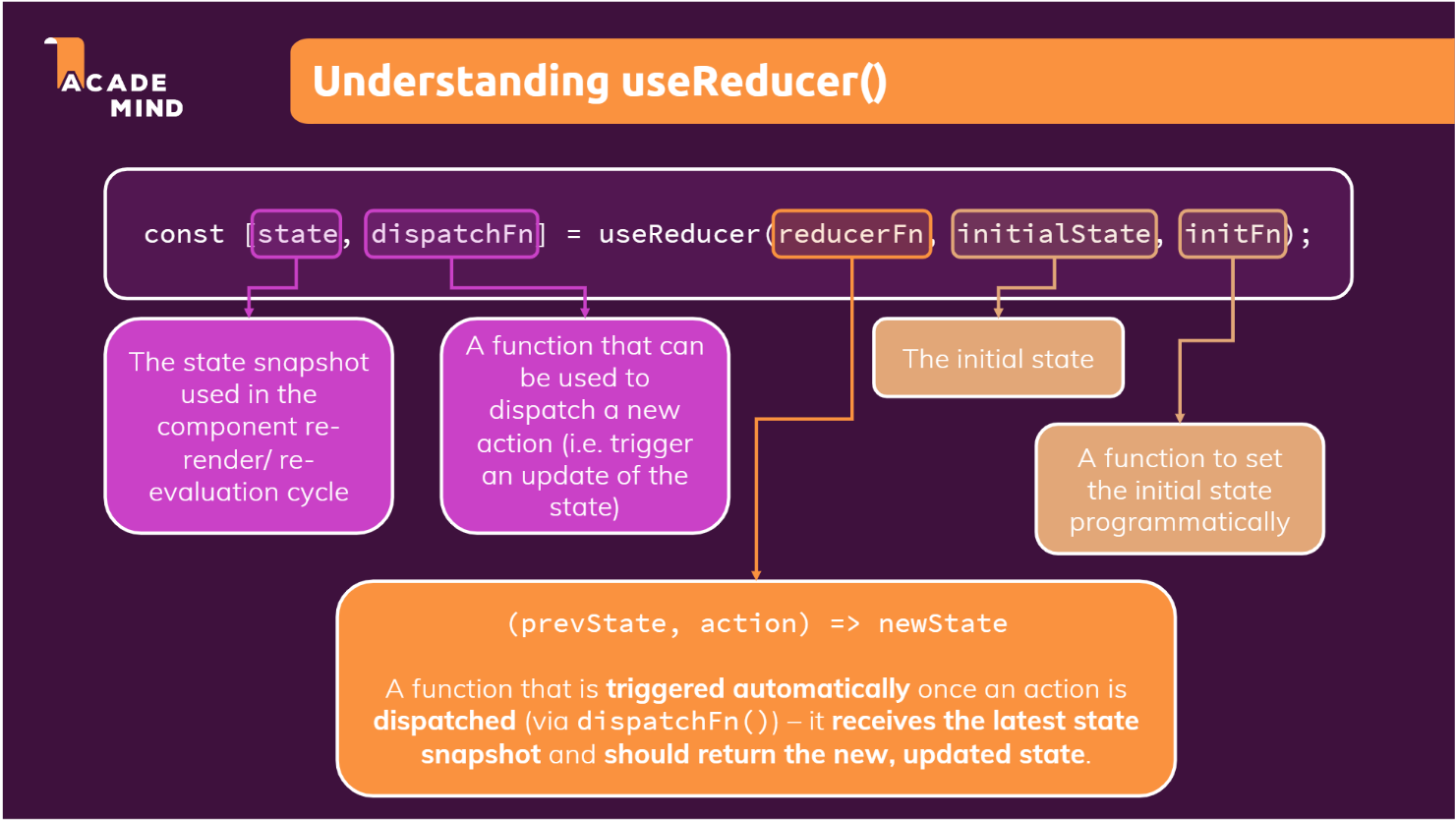
In the setFormIsValid, we are updating some state based on two other states (email and password). Now, we know that react “schedules” state updates. So we cannot guarantee that we are receiving the present states of enteredEmail and enteredPassword. So It is not correct way to write this type of code.

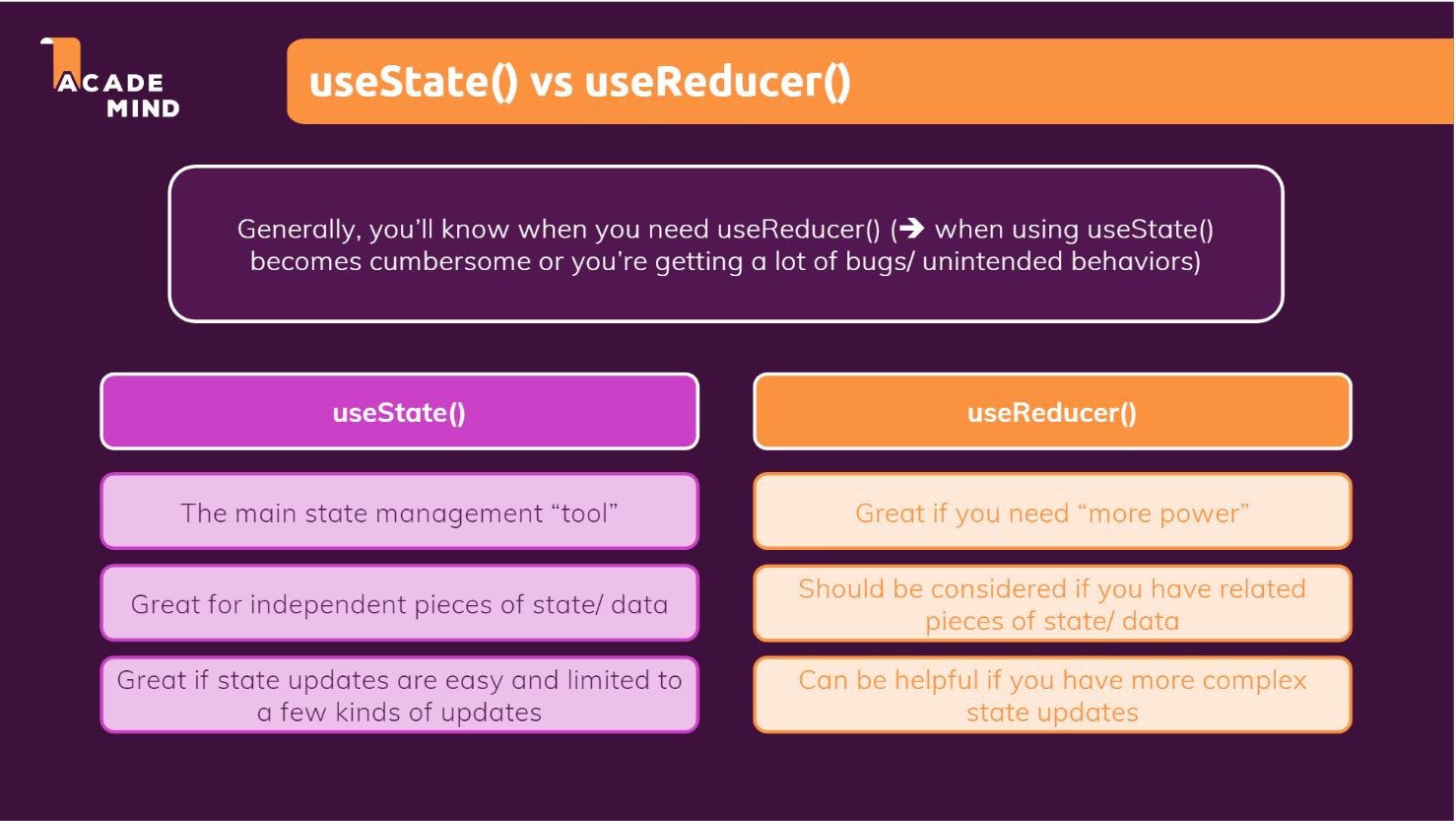
Now we can combine all this state variables (email, password, formIsValid) in an array of js objects and we use that array as state variable. That will work. But when states become more complex, we can use useReducer hook which is a powerful state management hook.

Use useReducer() when we have **multiple states, one state depending on other states or multiple ways of changing the state.**

7:







8: Use useReducer when you have complex state, we write the state updating logic in one place (i.e., the reducer function). (useState also uses useReducer)

const emailReducer = (state, action) => {

  if (action.type === 'USER\_INPUT') {

    return { value: action.val, isValid: action.val.includes('@') };

  }

  if (action.type === 'INPUT\_BLUR') {

    return { value: state.value, isValid: state.value.includes('@') };

  }

  return { value: '', isValid: false };

};

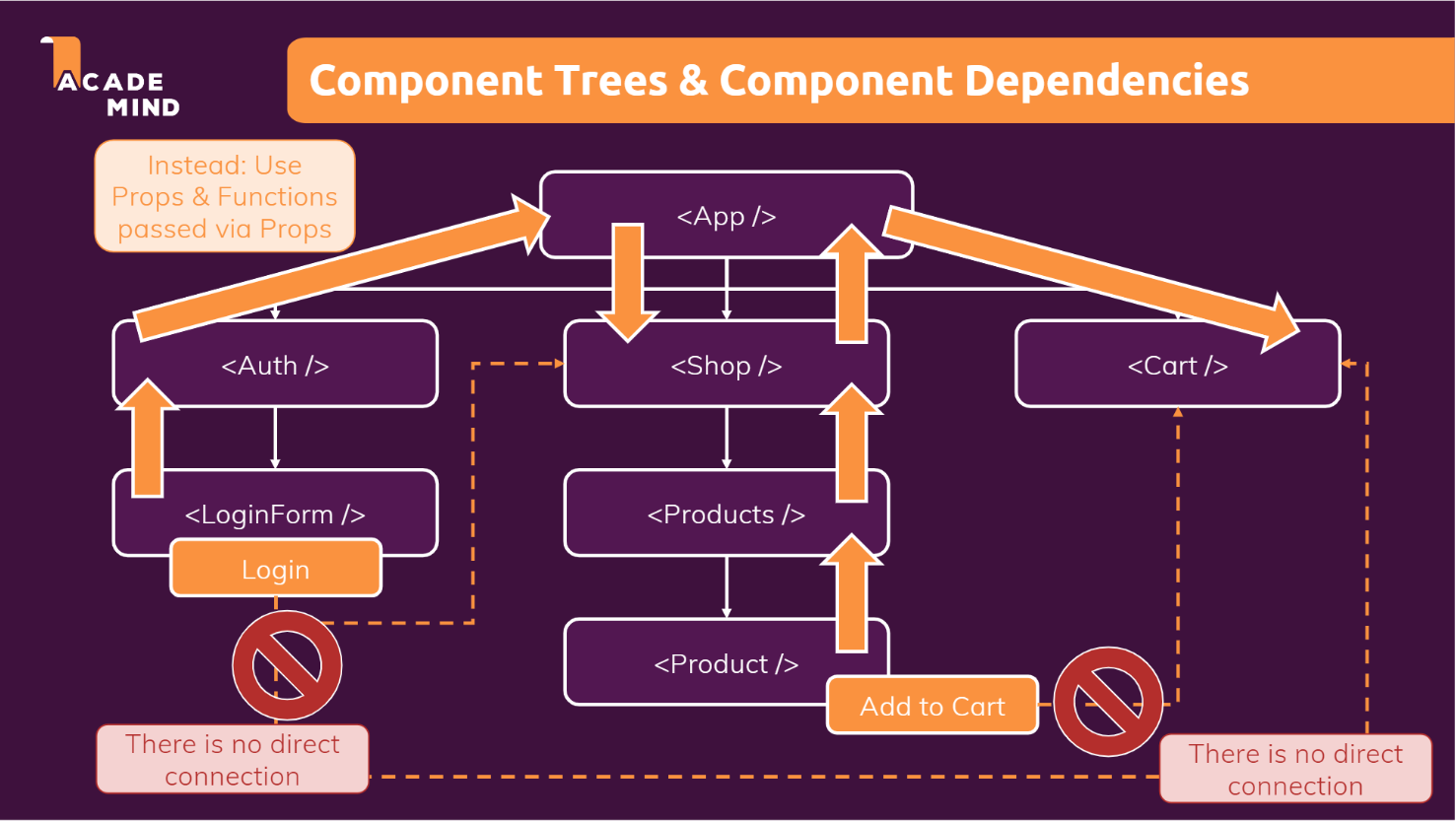
const [emailState, dispatchEmail] = useReducer(emailReducer, {

  value: '',

  isValid: null,

});

9: Many times we need to pass props through different components just so one component down the component tree can use it. We can avoid this by using React Context api. (VIDEO 12)



10: We usually create a folder named “context” or “store” to store our contexts. Suppose we want the authentication data to be available to several components. So in the folder ‘context’ or ‘store’, we create auth-context.js (We can name it AuthContext.js but that’s how we name our components). Note: We can name our files whatever we want…this is just the convention followed in most of the react projects.

import React, {createContext} from 'react';

const AuthContext = createContext({

    isLoggedIn: false

});

export default AuthContext;

Now we provide this data to the components using the Provider. When we provide this data to a component, the components’ children can also access it. Suppose we want to provide this authentication data to the whole app, We can simply wrap the components in the app.js file with AuthContext.Provider and pass a prop named value (It has to be value only) where we pass the current context value.

  return (

    <AuthContext.Provider value = {{isLoggedIn: false}}>

      <MainHeader isAuthenticated={isLoggedIn} onLogout={logoutHandler} />

      <main>

        {!isLoggedIn && <Login onLogin={loginHandler} />}

        {isLoggedIn && <Home onLogout={logoutHandler} />}

      </main>

    </AuthContext.Provider>

  );

}

Now using the provider, we provide the data (context). But to read the data, we use the “Consumer” function.

const Navigation = (props) => {

  return (

    <AuthContext.Consumer>

    {(context) => {

        <nav className={classes.nav}>

        <ul>

          {context.isLoggedIn && (

            <li>

              <a href="/">Users</a>

            </li>

          )}

          {context.isLoggedIn && (

            <li>

              <a href="/">Admin</a>

            </li>

          )}

          {context.isLoggedIn && (

            <li>

              <button onClick={props.onLogout}>Logout</button>

            </li>

          )}

        </ul>

      </nav>

    }}

    </AuthContext.Consumer>

  );

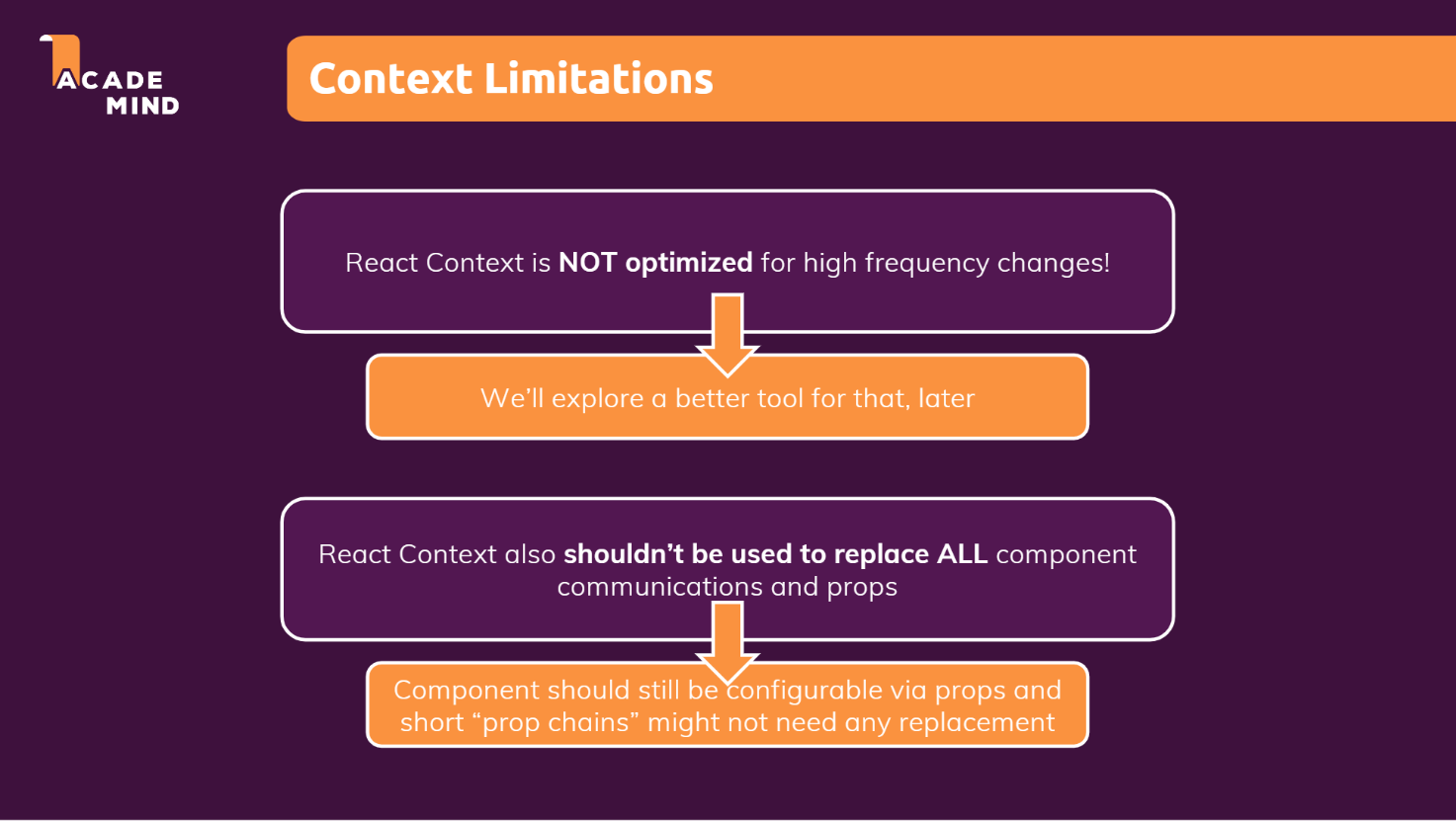
};

(VIDEO 13)

11: Instead of using the consumer function to get the context, we can also use the useContext hook

context = useContext(AuthContext);

we get the value we passed in the AuthContext.Provider (Video 14)



12: We can also pass function pointers in the context. (Note these functions or values which you define in the provider function need not be the in createContext method as the initial value) (Video 15)

13: But it is a good idea to include all the functions which you will be using through the function because the ide will give the auto-suggestion for the function or variable which we include in the createContext. To include a function in the createContext method, simply create a empty function like shown below

import React, {createContext} from 'react';

const AuthContext = createContext({

    isLoggedIn: false

onLogout: () => {}

});

export default AuthContext;

14: Say suppose we want the authentication details in all the components. So what we can do is, we can wrap the whole APP component with the AuthContextProvider custom component where we manage the state of the authentication.

15: Disadvantages of context: NOT optimised for high frequency changes.

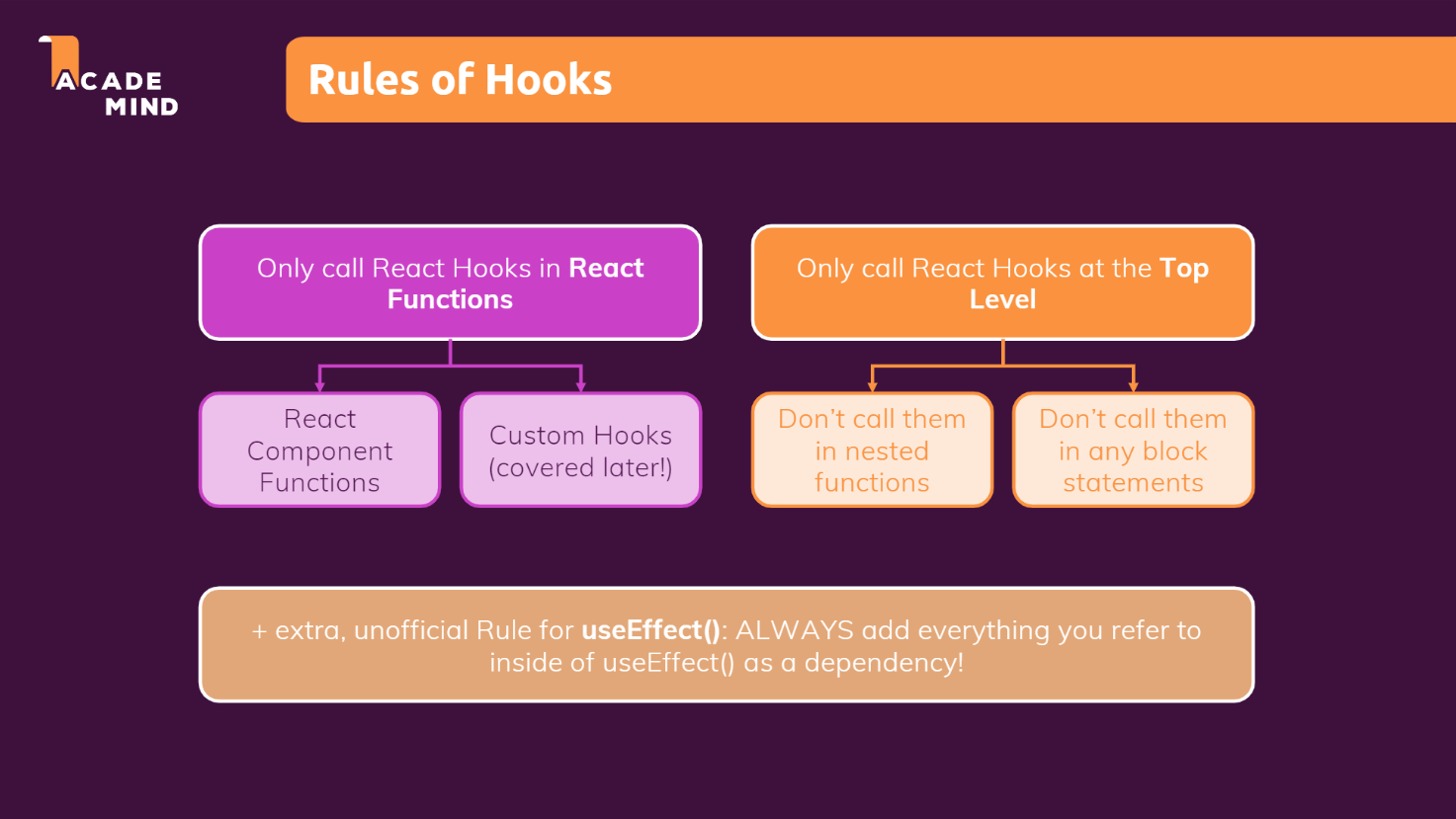
16: if the wrapping tag of provider gets rendered, the provider will render if the value prop changes. Otherwise the provider will not be rendered.

17: if we have to update the context, write functions for it in the createContext itself and create a state for the values used in the context.

**RULE OF HOOKS (React hooks always start with “use”)**

1. **Always use react hooks in “react function components or custom hooks”. By react function components, we mean the functions which return JSX code or React.createElement()**
2. **Always call react hooks at the top level (Don’t call them in nested functions, Don’t call them in any block statements (like in if block or else block).**
3. **Unofficial rule: Always add everything we refer to inside of useEffect in the dependency array (Even functions also, because they are also objects) (No need to pass state updating methods because React guarantees that the reference to state updating methods wont change after each rendering.**

(VIDEO 18)



**MODULE 11: Practice project (Food ordering App)**

1: We keep all our images, logo and related files in assets folder.

2: A modal is a message box that is displayed on top of your screen. Modals put an overlay on the screen; therefore, they take visual precedence over all the other elements.

3: Folder structure

**src/context**: context.js, Provider.js

**src/components:** Main components

**src/UI:** Card.js, Input.js, Modal.js, Button.js

**(note for modals, we created a modal and a backdrop (transparent black color with z index and inserted them into a div with id as overlays which is beside the root div using portal)**

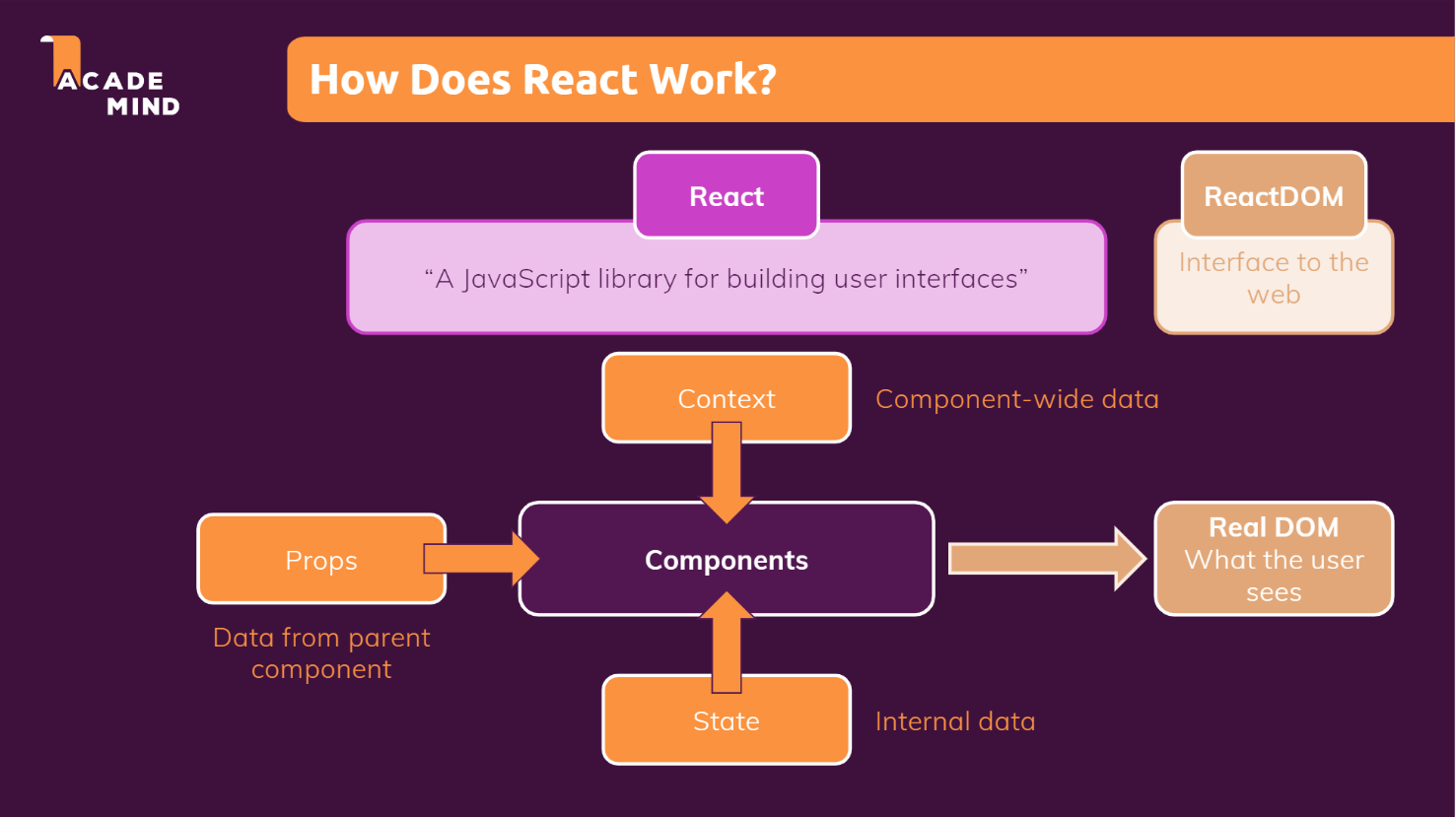
**src/Layout**: Header.js, Footer.js, Navbar.js etc

**Also, go through the project structure**

**MODULE 12: BEHIND THE SCENES**

1: REACT knows NOTHING ABOUT THE BROWSER. It is solely responsible for managing the components, props, state, context etc and React-DOM is responsible for rendering these components in the browser. React does not care whether the components contain HTML elements or any other components. React-DOM is responsible to work with the Real DOM to render the components.

Whenever there is any update in the components, React lets React-DOM know the changes and then React-DOM creates a virtual DOM, compares the newly created virtual DOM with the previous DOM and renders the component accordingly.

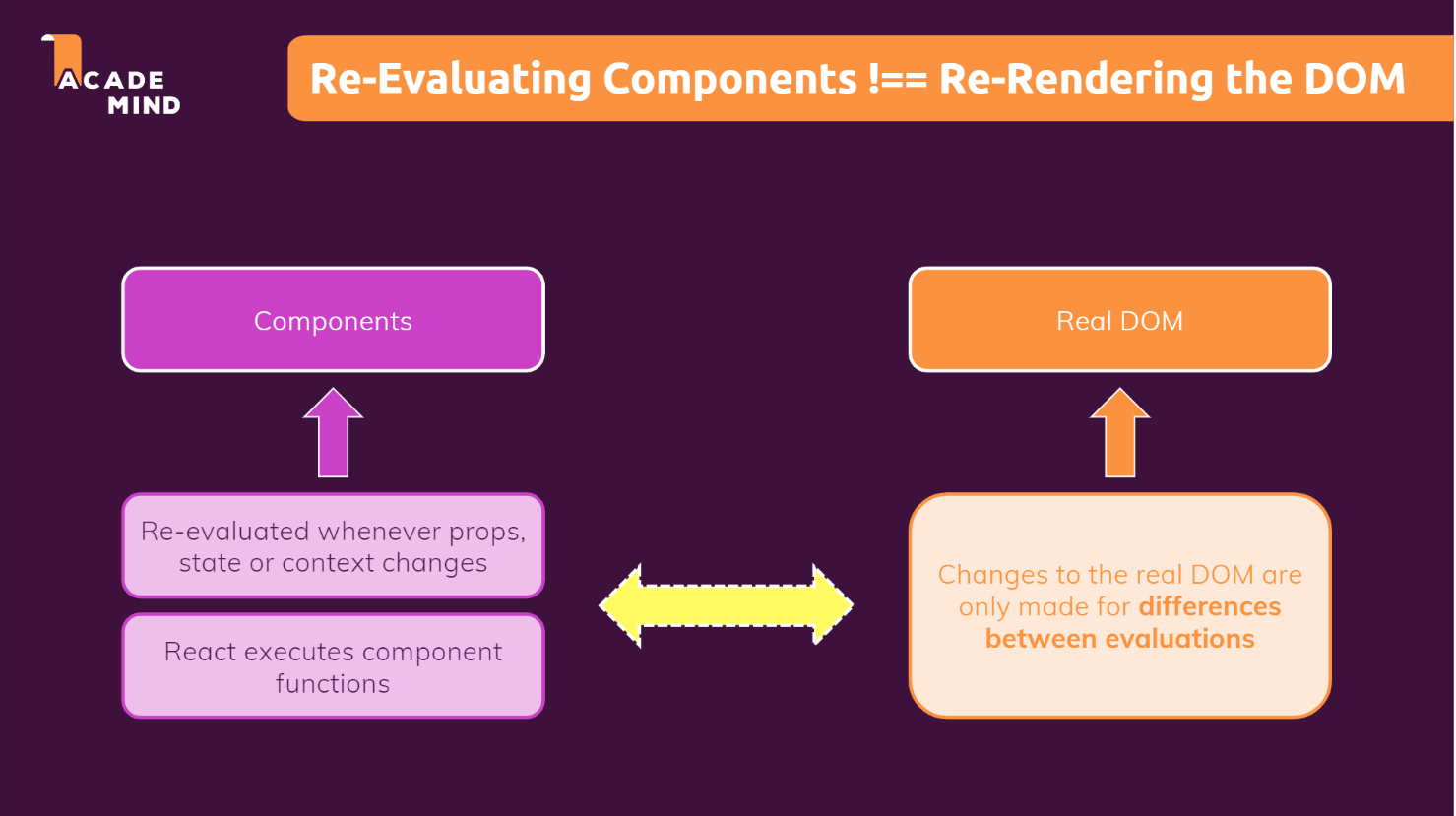


2: React uses a concept called virtual dom (Virtual dom is a light weight copy of actual dom but it does not have the power to change the document)

Graphical user interface, diagram, application

Description automatically generated

3: **Whenever state or props or context changes, the component re-renders. BUT re-evaluating the component is not the same as re-rendering the actual DOM. It does not mean that whenever the component will be evaluated, the actual DOM will re-render.**



The below info is taken from <https://medium.com/@happymishra66/virtual-dom-in-reactjs-43a3fdb1d130>

React-DOM does not update the real DOM directly but it updates the Virtual DOM. This causes a great performance benefit for ReactJS. Here, we will try to understand why updating the real DOM is slow and how updating Virtual DOM increase the performance?

**Why updating Real DOM is slow:**

Updating a DOM is not slow, it is just like updating any JavaScript object; then what exactly makes updating real DOM slow?

Let’s look at the below image from [html5Rocks](http://html5rocks/)to see how exactly browser renders a web page

Diagram

Description automatically generated

Rendering engines which are responsible for displaying or rendering the webpage on the browser screen parses the HTML page to create DOM. It also parses the CSS and applies the CSS to the HTML, thus creating a render tree, this process is called as *attachment*.

Layout process gives exact coordinates to each node of the render tree, where the node gets painted and displayed.

So when we do,

document.getElementById('elementId').innerHTML = "New Value"

Following thing happens:

1. The browser has to parse the HTML
2. It removes the child element of elementId
3. Updates the DOM with the “New Value”
4. Re-calculate the CSS for the parent and child
5. Update the layout i.e. each elements exact coordinates on the screen
6. paint the elements and display on the browser.

Recalculating the CSS and changing layouts uses complex algorithms and they affect the performance.

Thus updating a real DOM does not involve just updating the DOM but, it involves a lot of other processes.

**How Virtual DOM solves this problem?**

**What is virtual DOM?**

Virtual DOM is an in-memory representation of real DOM. It is a lightweight JavaScript object which is a copy of Real DOM.

Updating virtual DOM in ReactJS is faster because ReactJS uses

1. Efficient diff algorithm
2. Batched update operations
3. Efficient update of subtree only
4. Uses observable instead of dirty checking to detect the change

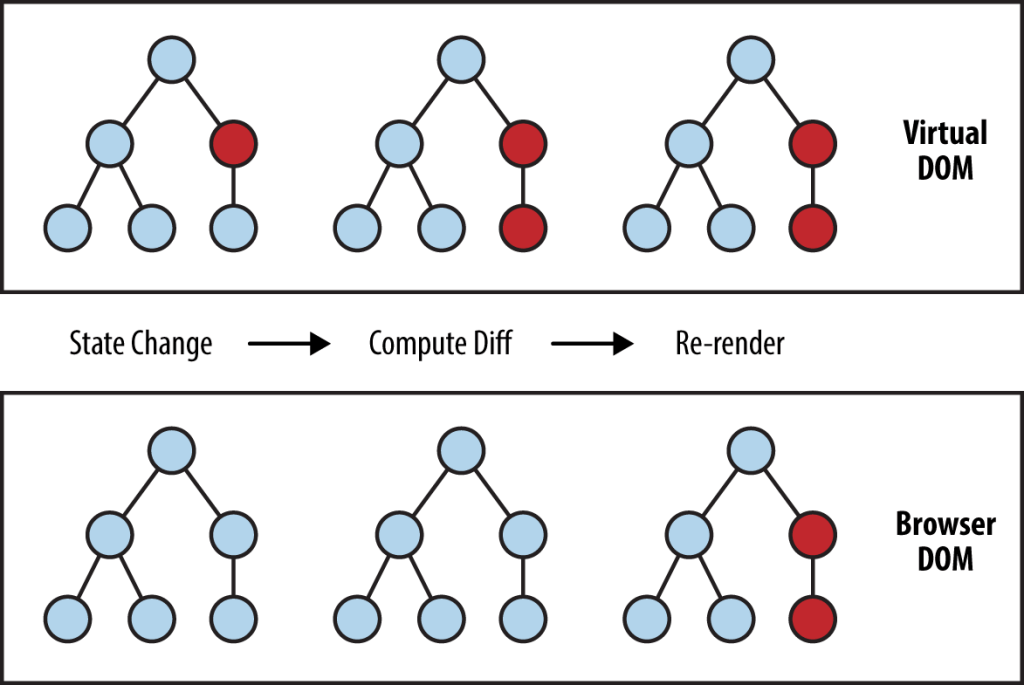
AngularJS uses dirty checking to find the models which have changed. This dirty checking process runs in cycle after a specified time. As the application grows, checking the whole model reduces the performance and thus makes the application slow.

ReactJS uses observable’s to find the modified components. Whenever setState() method is called on any component, ReactJS makes that component dirty and re-renders it.

Whenever setState() method is called, React-DOM creates the whole Virtual DOM from scratch. Creating a whole tree is very fast so it does not affect the performance. At any given time, ReactJS maintains two virtual DOM, one with the updated state Virtual DOM and other with the previous state Virtual DOM.

React-DOM then uses Diffing algorithm to find the changes between the current virtual dom and the previous virtual dom and marks the sub-tree of components that need re-rendering (Remember only the nodes which are updated will be re-rendered).

React-DOM syncs the changes between the updated virtual dom and the real dom. This process is called **Reconciliation. NOTE that React-DOM updates the real DOM in BATCHES (**updates to the real DOM are sent in batches, instead of sending updates for every single change in state.)

****

6: Changes to the real DOM are only made for the differences between previous component tree and present component tree. This is important for the performance.

React Hands over the Differences to React DOM

Then the ReactDOM is responsible for manipulating the actual DOM using the changes (This process is called reconciliation and react-DOM updates the actual-DOM in batches).

Your Components Change

React-DOM uses current virtual DOM and previous virtual DOM to know the differences. (This is known as Diffing.)

(ALL OF THIS FROM VIDEO 2)

7: In the elements tab in the developer tools, when a new html element is added, it will be flashed (VIDEO 3)

8: If we want the components not to **re-evaluate** again if the props of that component do not change, we can use **useMemo** function in the export statement. What useMemo hook does is whenever the parent component is re-evaluated, useMemo checks if the props to the child component is updated. If the props are not updated, then the child component also will not be rendered

import React from 'react'

function Component(props){

    console.log("Component will be re-evaluated only if the props change (Remember props equality is checked using references)")

    return <p>This is a component</p>

}

export default React.memo(Component);

But why is this not used for all components by default? When we use the memo function, react has to store the previous props and do the comparison between previous and current props. So we have to use the memo function depending on the component complexity and props complexity. And suppose we know that props are going to change anyways, so using useMemo will only increase the time complexity.

React memo can be very useful for a component that has many child components.

But if the props are objects, the component will be re-rendered because for objects, the references are different.

[1, 2, 3] === [1, 2, 3] will return false

9: To use memo function for the components which pass functions as prop, we use **useCallback hook**.

const memoizedCallback = useCallback(

    () => {

        if (flag) {

            //DO SOMETHING

        }

        else {

            //DO SOMETHING

        }

    },

    [flag]);

Now when the app runs the first time, the function inside the useCallback will be created and all this values will be stored. So if we are using some external defined variables in the useCallback hook, then mention it in the dependencies array because if we do not mention it in dependencies and the dependency is updated, react will use the **previous** value and **not the updated value.** This is the reason that we have to list down the dependencies in the dependency array.

in the 2nd argument of the useCallback hook, we pass the array of dependencies which tells react that after re-rendering of the component, if one of the dependency changes, a new function will be created again. Unless the dependencies change, react will return the same function created earlier. So we can use the useMemo hook since we have the same function object reference from the useCallback hook.

10: in a component, when we have many state updating functions, react simply batches (combines) them. So, no unnecessary re-rendering. That is why react waits and does not update state immediately.

11: Now we used the useCallback hook for using the same references for functions. For Objects we use useMemo hook

arr = useMemo(() => [1, 2, 3], [array\_of\_dependency]). Every time our app re-renders, it will compare the current and previous versions of dependencies and if they are same, the reference to the array will be same otherwise arr will have different reference.

NOTE THAT useCallback(fn, deps) is equivalent to useMemo(() => fn, deps)

**In short, when we use useCallback, when the component re-renders, the function in the useCallback will NOT be created again if the dependencies does not change. The old function will be used again. Same with useMemo.**

**MODULE 13: CLASS BASED COMPONENTS (INCOMPLETE)**

1: Class based components are alternative to functions.

2: Class-based components were used in the previous versions of react. Class based components **cannot use react hooks**

class CustomComponent{

render(){

return <h2>Class based Component</h2>

}

}

const CustomComponent = () => <h2>Function Based Component</h2>;

3: Now if we want to use props or other features of react, we have to inherit from the Component class

import {Component} from 'react';

class CustomComponent extends Component{

render(){

return <h2>{this.props}</h2>

}

}

Class based components can render functional components and vice-versa. (VIDEO 3)

To use states in classes, we have to initialise the state in the constructor method and state in classes is **ALWAYS AN OBJECT.**

import { Component } from "react";

class CustomComponent extends Component {

constructor() {

super();

this.state = {

count: 0,

flag: false

};

}

clickHander(){

this.setState((prevState) => {

return {

count: prevState.count + 1,

}

*//note that here the state will be merged NOT replaced*

*//therefor state will be {count:prevCount + 1, flag = false)*

*// not {oount: prevCount + 1} unlike useState hook in*

*// function based components where the state will be replaced*

});

}

render() {

return <button onClick = {this.clickHander.bind(this)}>{this.state.count}</button>

}

}

export default CustomComponent;

DID NOT UNDERSTAND ABOUT BIND HERE

(VIDEO 4)

4: Class based components have lifecycle methods like

1. componentDidMount()
2. componentDidUpdate()
3. componentWillUnmount()

ComponentDidMount(): Called once component is mounted (was evaluated and rendered) similar to useEffect hook with an **EMPTY DEPENDENCY ARRAY.**

ComponentDidUpdate(): Called when the component is updated (was evaluated and re-rendered) similar to useEffect with dependency array which consists of state which was changed to trigger the re-render.

ComponentWillUnmount(): Called right before the component is unmounted (Removed from DOM) similar to the **cleanup function in the useEffect hook.**

**GO THROUGH MODULE 13 again (MODULE 13 is not completed)**

**MODULE 14: SENDING HTTP REQUESTS**

1: **FRONTEND APPS SHOULD NOT BE DIRECTLY CONNECTED TO DATABASE,** because we can see the frontend code in the developer tools and **our database credentials will be exposed**.

We write backend code using nodejs, php, ASP.NET etc. that connects to the database and supplies necessary information to the frontend.

React App (In general any front end app)

Database (SQL or No-SQL)

Backend app (Nodejs, PHP etc.,)

2: Fetch api which is in-built is used to make http requests. Another commonly used api is axios. These apis return promises.

3: There are several ways to handle promises like callbacks, asyn-await, then-catch. (then-catch are the methods of promises)

4: callback functions are nothing but the functions which we want to execute after something happens. A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

const callback = () => console.log("Callback");

setTimeout(callback, 2000);

5: A promise has 2 possible outcomes: it will either be kept when the time comes, or it won’t. This is also the same for promises in JavaScript. When we define a promise in JavaScript, it will be resolved when the time comes, or it will get rejected.

A Promise is an object. There are 3 states of the Promise object:

**Pending**: Initial State, before the Promise succeeds or fails

**Resolved**: Completed Promise (If the promise is resolved, we get the value in then() method

**Rejected**: Failed Promise (Promise is rejected: We get the error in catch method)

const [flag, setFlag] = useState(true);

const promise = new Promise((resolve, reject) => {

if (flag) {

setTimeout(() => {

resolve("foo");

}, 2000);

}

else reject("Something is wrong");

});

promise.then((response) => console.log(response)).catch((error) => console.log(error));

6: Instead of then-catch, we can use async-await also.

7: **NOTE: async-await or then-catch can ONLY be used with promises. If we have an asynchronous code which does not return a promise, we have to convert it into an promise object.**

**8:**

const fetchMovieHandler = () => {

    const api = 'https://swapi.dev/api/films/';

    fetch(api).then((response) => response.json())

    .then((responseData) => console.log(responseData))

    .catch((error) => console.log(error))

  }

Is same as

  const fetchMovieHandler = async() => {

    const api = 'https://swapi.dev/api/films/';

    try{

      const response = await fetch(api);

      const responseData = await response.json();

      console.log(responseData);

    }

    catch (error){

      console.log(error);

    }

  }

**NOTE that the async function also returns a promise.**

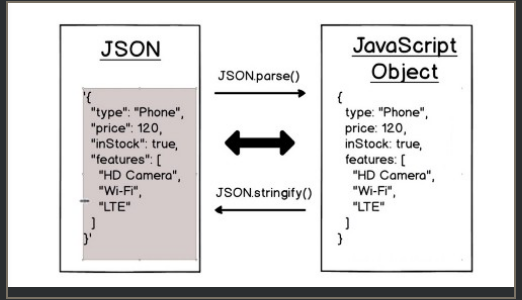
To handle errors with then, we use catch. To handle errors with async await, we use try-catch.

**NOTE that we are naming the function async in which we want to use the ‘await” keyword. (We cannot use await without async keyword)**

Why do we need async keyword? Is it not enough if we just include await? Here is the explanation on the stackoverflow: *“And this is why async was added as a function modifier: to avoid breaking existing code that used await as a variable name. Since no async methods already existed, no old code is invalidated, and in new code, the compiler can use the presence of the async tag to know that await should be treated as a keyword and not an identifier.”* **(**[**https://softwareengineering.stackexchange.com/questions/187492/why-do-we-need-the-async-keyword**](https://softwareengineering.stackexchange.com/questions/187492/why-do-we-need-the-async-keyword)**)**

9: Difference between JavaScript objects and JSON.

In JSON, the keys are wrapped in double quotes and values are either strings, numbers, objects, arrays, Boolean, null **BUT NOT functions**. In javascript objects keys need not be only strings and values can be functions also.



JSON format is language independent, so it is not a term only limited to javascript as opposed to object literal.

10:

const api = 'https://swapi.dev/api/films/';

    fetch(api).then((response) => {

      return response.json();

    }).then((responseData) => console.log(responseData));

Because we receive the response as **soon as all headers** have arrived. Calling .json() gets us another promise for the body of the http response that is yet to be loaded.

.json() method DOES NOT GIVE the response in json format. It takes a json object and gives us a javascript object.

11: lets say we are sorting an arr

Sort(arr)

Print(arr)

If the arr is too long, will the arr be printed before sorting Is completed? **NO. because these are synchronous code. We have to know which functions are asynchronous so that we can deal with them better.**

12: React “schedules” state updates to avoid unnecessary re-renders. It **DOES NOT MEAN THAT REACT WILL WAIT UNTIL ALL THE STATE UPDATING CODE IS FINISHED EXECUTING.**

If react had waited for all the state updating code to finish executing, we would not have seen 1, we would have seen directly 2 after 0. Here the component will be rendered a total of 3 times

const [val, setVal] = useState(0);

const fetchMovieHandler = async() => {

const api = 'https://swapi.dev/api/films/';

setVal(1);

const response = await fetch(api);

const responsedata = await response.json();

console.log(responsedata);

setVal(2);

};

return(

<div>

<button onClick={fetchMovieHandler}>Click Me!</button>

<p>{val}</p>

</div>)

13: lets say we need to fetch all the movies when the user goes to the page, before clicking the button. In this case we can use useEffect.

  useEffect(() => {

    fetchMovieHandler();

  }, []);

This code will be executed only once when the component mounts as the dependency array is empty. Now let’s say we want to execute fetchMovieHandler was a global function and we changed it using context methods. But it will not be re-executed because the dependency array is empty.

We learnt that we have to include any dependency which may change in the dependency array. Since **functions are also objects in js, we should include this in the dependency array. (NOTE that we need not include setState functions because react guarantees that setState will have the same reference)**

  useEffect(() => {

    fetchMovieHandler();

  }, [fetchMovieHandler]);

**This above code will cause infinite loop because the reference to the fetchMovieHandler changes after each re-rendering.**

To solve this problem, we can use useCallback hook so that reference does not change.

const fetchMovieHandler = useCallback(async() => {

    const api = 'https://swapi.dev/api/films/';

    try{

      setIsLoading(true);

      const response = await fetch(api);

      const responseData = await response.json();

      setIsLoading(false);

      setMovieList(

      responseData.results.map((movieInfo) => {

        return {

          id: movieInfo.episode\_id,

          title: movieInfo.title,

          openingText: movieInfo.opening\_crawl,

          releaseDate: movieInfo.release\_date

        }

      }));

    }

    catch (error){

      console.log(error);

    }

  }, [dependencies]);

14: We can also send POST requests using fetch api.

fetch(api, {

method: ‘POST’

body: JSON.stringify(movie)

headers: {‘Content-Type’: ‘application/json’}

});

In body field, we pass the data required (in most cases, we send the data in json format. JSON.stringify() takes a javascript object and converts into JSON

A response header is an HTTP header that can be used in an HTTP response and that doesn't relate to the content of the message. Response headers, like Age , Location or Server are used to give a more detailed context of the response.

Eg of a response header:

200 OK

Access-Control-Allow-Origin: \*

Connection: Keep-Alive

Content-Encoding: gzip

Content-Type: text/html; charset=utf-8

Date: Mon, 18 Jul 2016 16:06:00 GMT

Etag: "c561c68d0ba92bbeb8b0f612a9199f722e3a621a"

Keep-Alive: timeout=5, max=997

Last-Modified: Mon, 18 Jul 2016 02:36:04 GMT

Server: Apache

Set-Cookie: mykey=myvalue; expires=Mon, 17-Jul-2017 16:06:00 GMT; Max-Age=31449600; Path=/; secure

Transfer-Encoding: chunked

Vary: Cookie, Accept-Encoding

X-Backend-Server: developer2.webapp.scl3.mozilla.com

X-Cache-Info: not cacheable; meta data too large

X-kuma-revision: 1085259

x-frame-options: DENY

**NOTE:**

function App() {

const [val, setVal] = useState(0);

console.log(val);

const clickHandler1 = () => {

setVal((prevState) => prevState);

}

const clickHandler2 = () => {

setVal((prevState) => prevState + 1);

}

return (

<div className="App">

<h1>Hello CodeSandbox</h1>

<h2>Edit to see some magic happen!</h2>

<button onClick={clickHandler1}>Button 1</button>

<button onClick={clickHandler2}>Button 2</button>

</div>

);

}

**Ques:** I am trying to understand the useState hook in React. When the below component renders for the first time, it prints 0. When I click Button2, it prints 1. Now, when I click button1, after pressing button2, why is the App component being rendered again even though I am passing the same value? However when I click the button2 again, the component is not re-rendered again.

Yes, the app gets rendered, but the child components and useeffect code will not get rendered again.

Text

Description automatically generated

<https://github.com/facebook/react/issues/14810>

<https://stackoverflow.com/questions/24718709/reactjs-does-render-get-called-any-time-setstate-is-called>

**MODULE 15: CUSTOM HOOKS**

1: Custom hooks are just functions which has re-usable logic except that the custom hooks can use react-hooks unlike “normal” functions. Let’s say we have reusable state logic. We can’t write this logic in the normal function because that would violate the rule of hooks.

2: Do I have to name my custom Hooks starting with “use”? Please do. This convention is very important. Without it, we wouldn’t be able to automatically check for violations of rules of Hooks because we couldn’t tell if a certain function contains calls to Hooks inside of it. (Taken from the documentation)

3: CONVENTION: Create a separate folder “hooks” outside the components folder like the “store” folder which we used for storing the context files. Create files with the name “use-${your hook name}”. For e.g. use-counter or use-auth etc.,

import { useState, useEffect } from "react";

const useCounter = (flag = true) => {

    const [counter, setCounter] = useState(0);

//flag is set to true by default

    useEffect(() => {

        const interval = setInterval(() => {

            if(!flag)setCounter((prevCounter) => prevCounter - 1);

            else setCounter((prevCounter) => prevCounter + 1);

        }, 1000);

        return () => clearInterval(interval);

    }, [flag]);

    return counter;

}

export default useCounter;

//for forward counter use this, because flag is set to true by default

const counter = useCounter();

//for backward counter

const counter = useCounter(false);

4: When we use a component that uses a custom hook which has states, the component will re-render when the states in the custom hook are updated. State in the custom hooks is attached to the component in which we use the custom hook. In short, when the custom hook re-renders, the component which uses the custom hook will also re-render.

5: **Note: In the dependency array, always write every external objects or variables which we have used in the callback function.**

**Consider the code below:**

const {isLoading, error, sendRequest} = useHttp();

  const requestConfig = useMemo(() => {

    return {

      url: 'https://react-http-74835-default-rtdb.firebaseio.com/tasks.json'

    }

  }, []);

  const applyData = useCallback((data) => {

    const loadedTasks = [];

      for (const taskKey in data) {

        loadedTasks.push({ id: taskKey, text: data[taskKey].text });

      }

      setTasks(loadedTasks);

  }, []);

  useEffect(() => {

    sendRequest(requestConfig, applyData);

  }, [sendRequest, requestConfig, applyData]);

**If we write the request config function and apply data in the useEffect, then we can remove them from the dependency array**

useEffect(() => {

    const requestConfig = {

      url: 'https://react-http-74835-default-rtdb.firebaseio.com/tasks.json'

    }

    const applyData = (data) => {

      const loadedTasks = [];

      for (const taskKey in data) {

        loadedTasks.push({ id: taskKey, text: data[taskKey].text });

      }

      setTasks(loadedTasks);

    };

    sendRequest(requestConfig, applyData);

  }, [sendRequest]);

6:

    return {

        isLoading: isLoading,

        error: error,

        sendRequest: sendRequest

    };

Is same as

    return {

        isLoading,

        error,

        sendRequest

    };

7:

Const useHttp = () => {

return {

        isLoading: isLoading,

        error: error,

        sendRequest: sendRequest

    };

}

const {isLoading, error, sendRequest} = useHttp();

when destructing, if we want to use another name for sendRequest, we can do the following

const {isLoading, error, sendRequest: anotherName} = useHttp();

anotherName(); // anotherName refers to sendRequest method

**DID NOT UNDERSTAND THE BIND METHOD**

**MODULE 16: WORKING WITH FORMS**

1: When to validate forms?

1. When form is submitted
2. When a input is losing focus
3. On every keystroke. (VIDEO 3)

2: When submitting a form, if we do not trigger the preventDefault method, the browser will send a default http request to the server which is serving this website and the page will be reloaded. So we have to trigger this method to prevent this behaviour.

const formSubmitHandler = (event) => {

    event.preventDefault();

  }

3: While working with ref, do not use ref to modify the values because if we modify the values using ref, we are directly making changes to the DOM. And We should NOT change the DOM directly. It is the job of react to change the DOM.

To update the DOM, react first compares the present and previous Virtual DOM (Also known as diffing) (Remember Virtual DOM is a light-weight copy of the Real DOM except that it cannot manipulate the DOM) and hands over the changes to React-DOM which then updates the part of the DOM wrt these changes.

So using Ref, we SHOULD NOT manually modify the DOM. The following code should not be practiced.

  const formSubmitHandler = (event) => {

    event.preventDefault();

    const inputName = inputRef.current.value;

    console.log(inputName)

    inputRef.current.value = '';

  }

If we want to reset the user input are form submission, then using state would be better. If we want to just read the value on form submission, then use useRef.

(VIDEO 4)

4: Validating Input in the form.

We can validate in the client side., that would provide a good user experience. But we should also remember that the client side code which is run by the browser is editable in the dev-tools (in the sources tab). So it is best practice to validate the input in the server.

The following code will show the input validation when the form is submitted. (VIDEO 7, 8)

import { useState, useRef } from 'react';

const SimpleInput = (props) => {

  const [input, setInput] = useState('');

  const [enteredNameIsValid, setEnteredNameIsValid] = useState(true);

  const inputRef = useRef();

  const inputChangeHandler = (event) => {

    setInput(event.target.value);

  }

  const formSubmitHandler = (event) => {

    event.preventDefault();

    if(input.trim() === ''){

      setEnteredNameIsValid(false);

      return;

    }

    setEnteredNameIsValid(true);

    console.log(input);

  }

  const nameInputClasses = enteredNameIsValid? 'form-control': 'form-control invalid';

  return (

    <form onSubmit = {formSubmitHandler}>

      <div className= {nameInputClasses}>

        <label htmlFor='name'>Your Name</label>

        <input type='text' id='name' ref = {inputRef} onChange = {inputChangeHandler} value = {input} />

        {!enteredNameIsValid && <p>Entered Name is invalid</p>}

      </div>

      <div className="form-actions">

        <button type = "submit">Submit</button>

      </div>

    </form>

  );

};

Here enteredName is valid if enteredName.length != 0. But initially when our component starts, we set the enteredNameIsValid to true even though the input field is empty which is not correct.

Let’s say we have the following useEffect code in the same component.

  useEffect(() => {

    if(enteredNameIsValid){

      console.log("Entered Name is Valid");

    }

  }, [enteredNameIsValid]);

Then “Entered Name is Valid” will be the output when the component starts even though the input field is empty. This is not correct. To solve this, We can maintain another state which checks whether the input field is touched or not as shown below.

const SimpleInput = (props) => {

  const [input, setInput] = useState('');

  const [enteredNameIsValid, setEnteredNameIsValid] = useState(false);

  const [enteredNameTouched, setEnteredNameTouched] = useState(false);

  const inputRef = useRef();

  const inputChangeHandler = (event) => {

    setInput(event.target.value);

  }

  const formSubmitHandler = (event) => {

    event.preventDefault();

    setEnteredNameTouched(true);

    if(input.trim() === ''){

      setEnteredNameIsValid(false);

      return;

    }

    setEnteredNameIsValid(true);

    console.log(input);

  }

  useEffect(() => {

    if(enteredNameIsValid){

      console.log("Entered Name is Valid");

    }

  }, [enteredNameIsValid]);

  const nameInputClasses = !enteredNameTouched? 'form-control': enteredNameIsValid? 'form-control': 'form-control invalid';

  return (

    <form onSubmit = {formSubmitHandler}>

      <div className= {nameInputClasses}>

        <label htmlFor='name'>Your Name</label>

        <input type='text' id='name' ref = {inputRef} onChange = {inputChangeHandler} value = {input} />

        {!enteredNameIsValid && enteredNameTouched && <p>Entered Name is invalid</p>}

      </div>

      <div className="form-actions">

        <button type = "submit">Submit</button>

      </div>

    </form>

  );

};

export default SimpleInput;

5: Let’s say we want to validate the input when the input field has lost focus. We can do that using **onBlur** prop in the input tag.

  const inputBlurHandler = (event) => {

    setEnteredNameTouched(true);

    if(event.target.value === ''){

      setEnteredNameIsValid(false);

    }

  }

Now if the input is empty and the input has lost focus, the error will be shown. But when we start entering in the input field, the error wont go away because we are not validating it on every keystroke. So we can do this in the inputChangeHandler method.

  const inputChangeHandler = (event) => {

    setInput(event.target.value);

    setEnteredNameTouched(true);

    if(event.target.value === ''){

      setEnteredNameIsValid(false);

      return;

    }

    setEnteredNameIsValid(true);

  }

Now we can see that there is code duplication in inputChangeHandler, inputBlurHandler and formSubmitHandler which involves react hooks. Therefore we can use a custom hook here.

6: **formik** is a goodlibrary to manage forms.

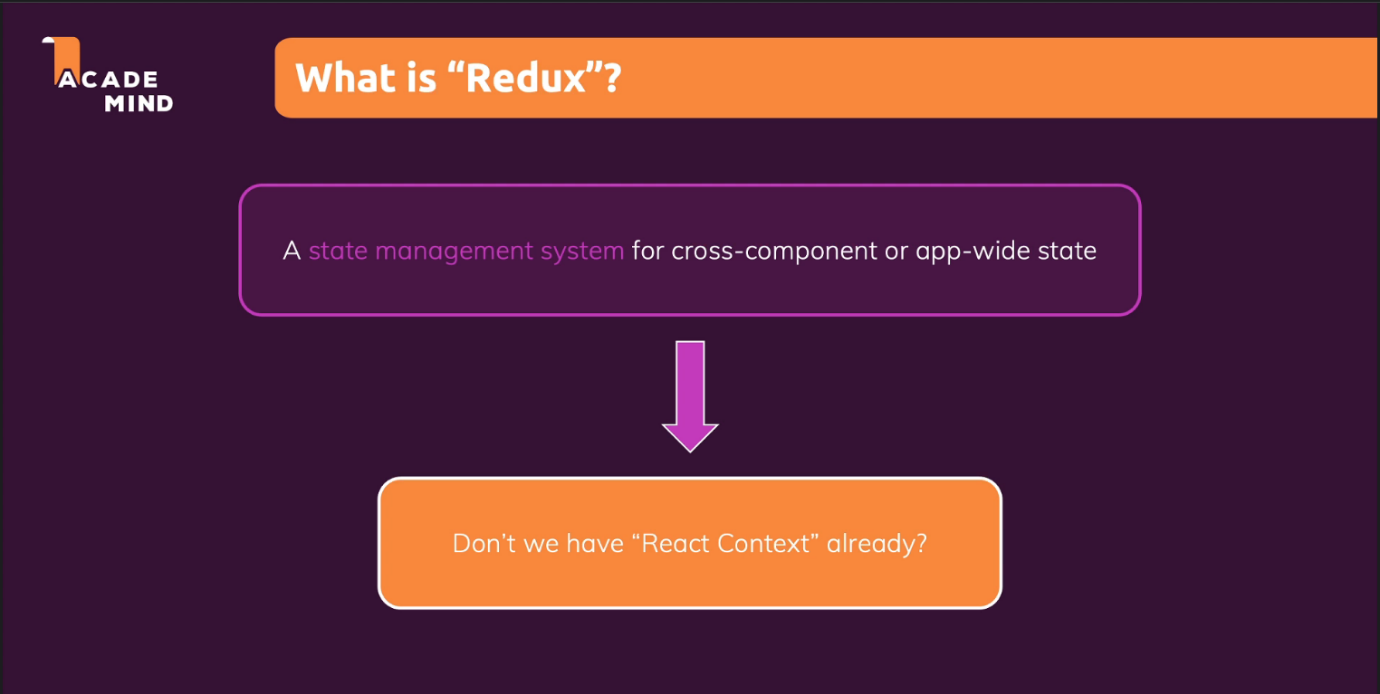
**Section 18: Diving into Redux (An alternative to context API)**

**Redux is a state management system for cross-component or app-wide state. Redux is based on flux architecture. Facebook recommends the flux architecture for creating SPAs.**

**About Flux Architecture:** [**https://dook.pro/blog/technology/39-redux-vs-react-context-which-one-is-the-right-winner**](https://dook.pro/blog/technology/39-redux-vs-react-context-which-one-is-the-right-winner)

**Diagram

Description automatically generated**

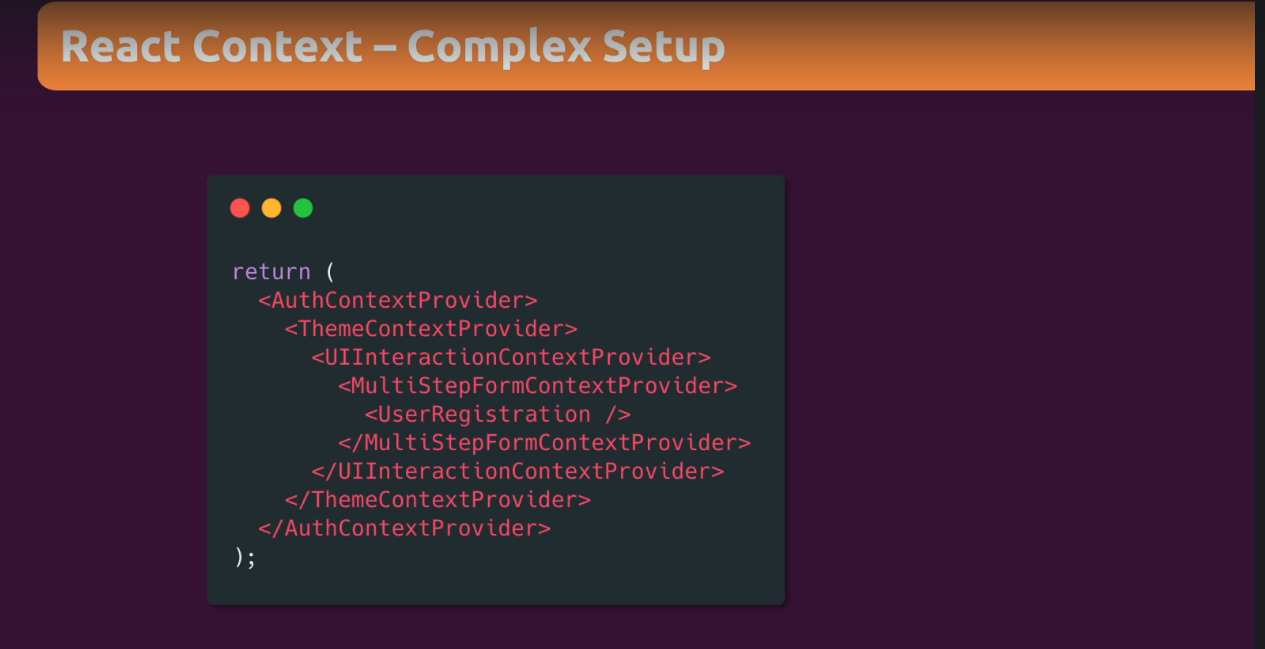
****

**Graphical user interface, diagram

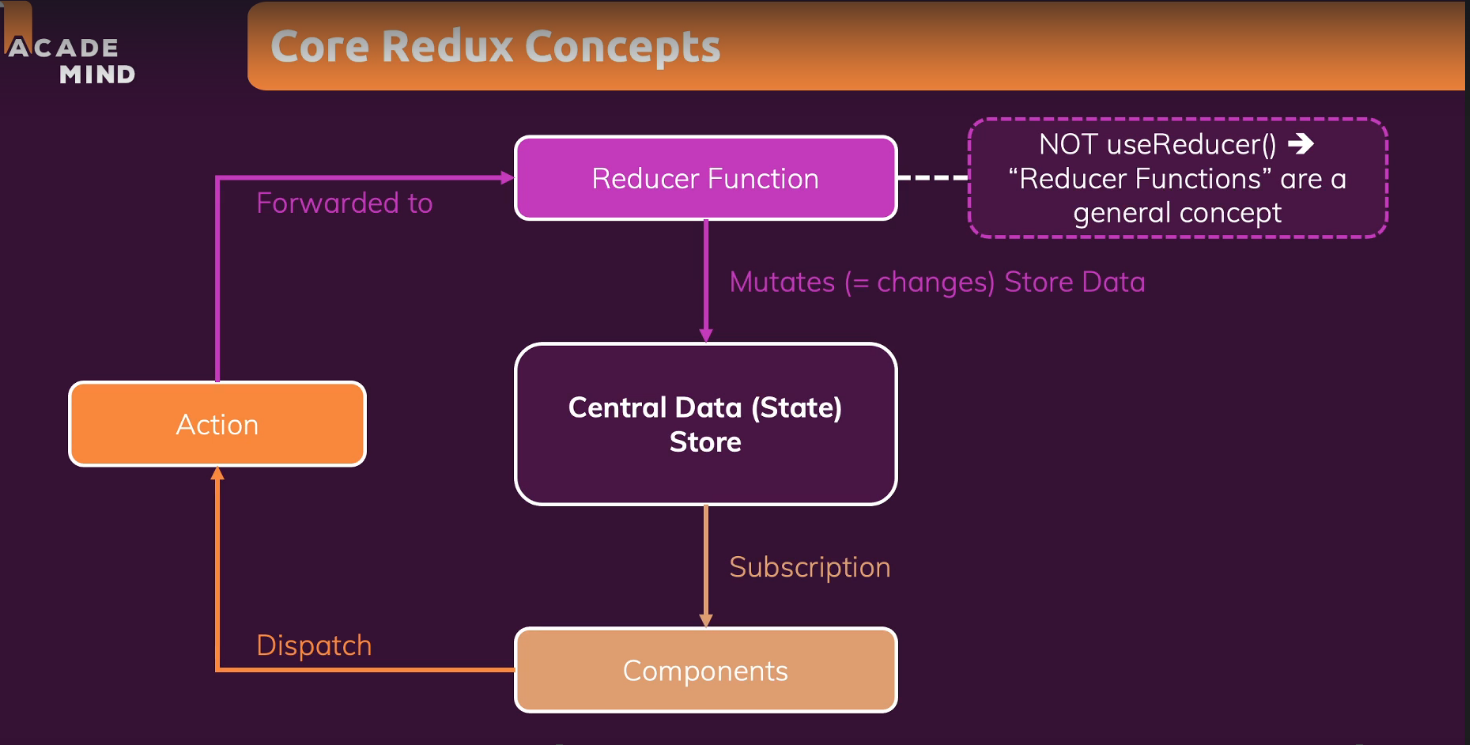
Description automatically generated**

**Context vs Redux**

|  |  |
| --- | --- |
| Context | Redux |
| **Context is not optimised for high frequency changes** | **Redux is optimised for high frequency changes** |
| **Using context api requires less setup** | **Using redux requires more setup (We have to create a store, reducers to change the state)** |
| **if we want only to pass props between parent and a deeply nested child and nothing else, then we should use context** | **If we also want to change the state frequently, then we should use redux** |
| **Context does not provide any dev tools** | **Redux provides redux-devtools to trace the state updates which makes debugging easier** |
|  |  |
|  |  |
|  |  |

****

1. In Redux, we have one store to manage the state of our application.
2. Components subscribe to the data. So whenever the data changes, the store notifies the components which have subscribed.
3. To manipulate the data inside the store, components use reducers and actions.



Note that the reducer function has to be pure functions (Which means that same input should lead to same output). **We must not send http requests, write to local storage, fetch from local storage etc. in reducers.**

const redux = require('redux');

// reducer function which will be called by redux library whenever an action is dispached or THE STORE is created for the first time.

// that is why, we have to initialise the state with the initial value.

const counterReducer = (state = {count: 0}, action) => {

    if(action.type === 'INCREMENT')

        return {

            count: state.count + 1

        }

    if(action.type === 'DECREMENT')

        return {

            count: state.count - 1

        }

    return state;

}

// create the store using the reducer

const store = redux.createStore(counterReducer);

// subscribe to the store. Whenever the data in the store changes, the method which has subscribed to the store will get notified.

store.subscribe(() => {

    // print the current state

    console.log(store.getState());

})

// dispatch an action.

store.dispatch({

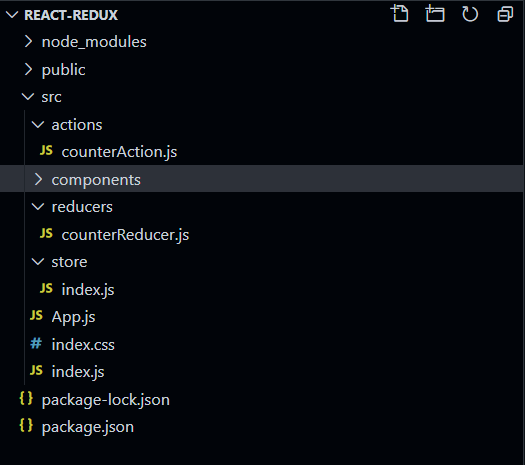
    type: 'INCREMENT'

})

**Redux in react:**

**Folder structure:**

1. **src/components**
2. **src/actions (Action functions for the store)**
3. **src/reducers (Reducer functions for the store)**
4. **src/store/index.js (The path where the store would be created)**

****

**src/store/index.js**

import {createStore} from 'redux'

import counterReducer from '../reducers/counterReducer'

const store = createStore(counterReducer);

export default store;

**src/reducers/counterReducer.js**

const initialState = {

    counter: 0

}

// REDUCER FUNCTION OF COUNTER.

// REMEMBER THAT REDUCER FUNCTIONS MUST BE PURE FUNCTIONS (NO HTTP REQUESTS, SAVING TO LOCAL STORAGE ETC)

const counterReducer = (state = initialState, action) => {

    if(action.type === 'INCREMENT'){

        return {

            counter: state.counter + 1

        }

    }

    if(action.type === 'DECREMENT'){

        return {

            counter: state.counter - 1

        }

    }

    return state;

}

export default counterReducer;

**src/actions/counterAction.js**

const incrementCounterAction = () => {

    return {

        type: 'INCREMENT'

    }

}

const decrementCounterAction = () => {

    return {

        type: 'DECREMENT'

    }

}

export {

    incrementCounterAction,

    decrementCounterAction

}

**src/components/counter.js**

**When an action is dispatched, useSelector() will do a reference comparison of the previous selector result value and the current result value. If they are different, the component will be forced to re-render. If they are the same, the component will not re-render.**

import classes from './Counter.module.css';

import {useSelector, useDispatch} from 'react-redux'

import {incrementCounterAction, decrementCounterAction} from '../actions/counterAction';

const Counter = () => {

  // subscribe to the counter of the state. Now whenever the

  // counter value changes, useSelector would render the component

  const counter = useSelector((state) => state.counter);

  // dispatch function to dispatch actions

  const dispatch = useDispatch();

  const incrementCounterHandler = () => {

    console.log("Dispatching increment action...")

    dispatch(incrementCounterAction())

  };

  const decrementCounterHandler = () => {

    dispatch(decrementCounterAction())

  }

  return (

    <main className={classes.counter}>

      <h1>Redux Counter</h1>

      <div className={classes.value}>{counter}</div>

      <button onClick={incrementCounterHandler}>Increment Counter</button>

      <button onClick={decrementCounterHandler}>Decrement Counter</button>

    </main>

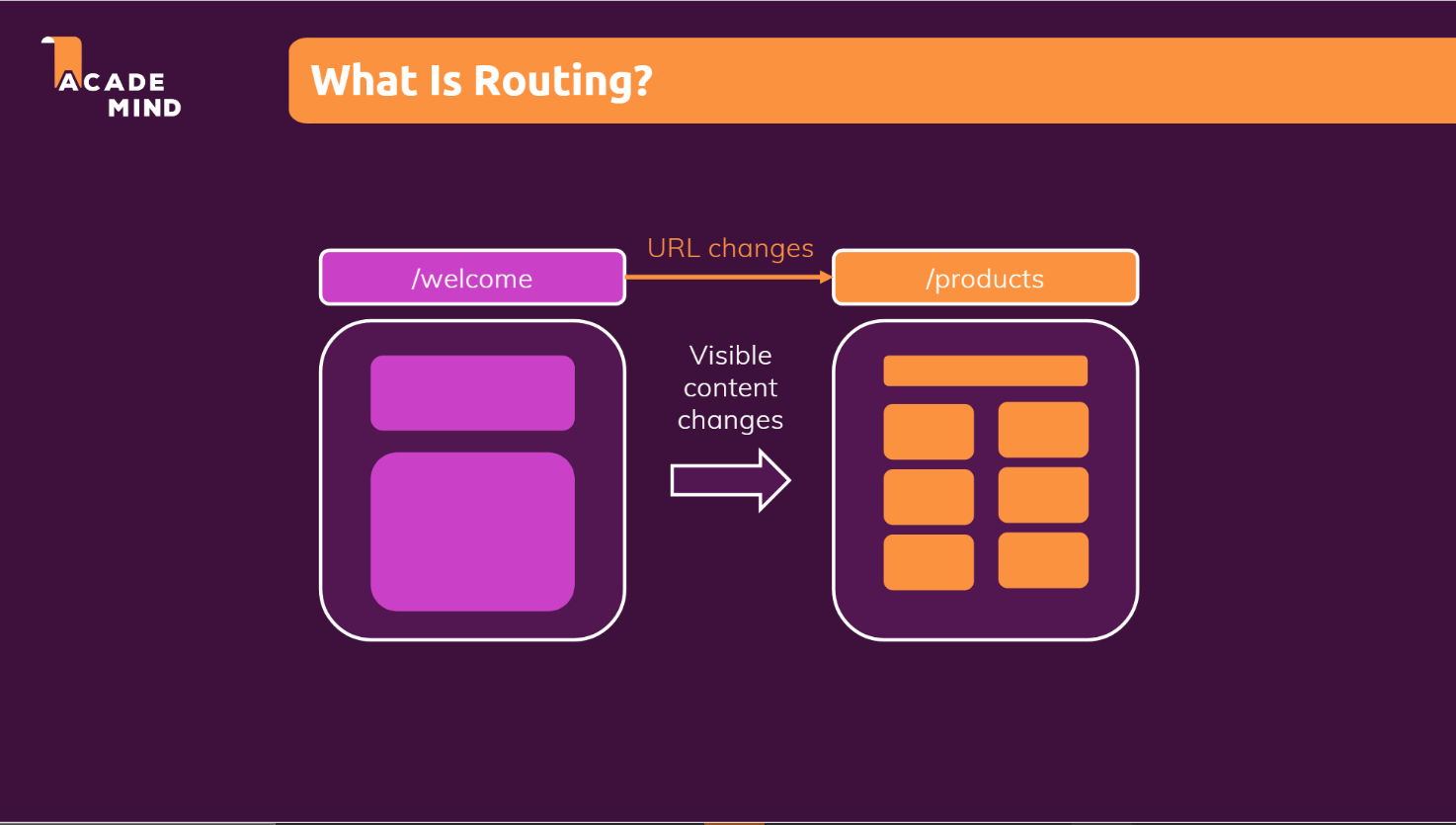
  );

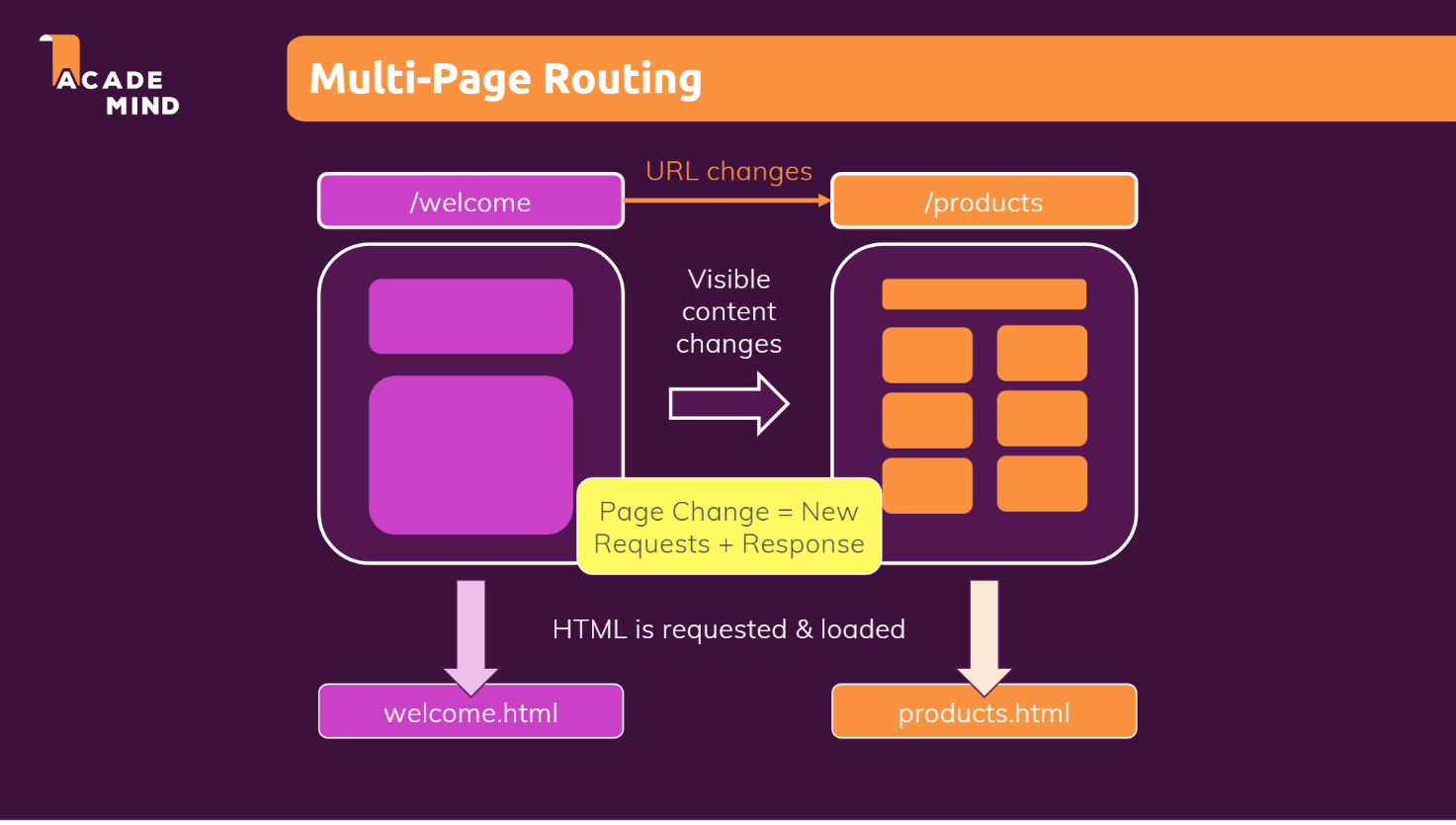
};

export default Counter;

**Section 19: Advanced Redux**

**Section 20: Building a Multi-Page SPA with React Router**

****

****

**Graphical user interface, text, application

Description automatically generated**

1. Install the **react-router-dom@5** (dom implies we are installing the router for web apps and @5 implies we are installing version 5)
2. To enable routing, we need to follow 2 steps
3. Wrap the app component in index.js with **BrowserRouter**

import React from 'react';

import ReactDOM from 'react-dom/client';

import './index.css';

import App from './App';

import { BrowserRouter } from 'react-router-dom'

const root = ReactDOM.createRoot(document.getElementById('root'));

// unless we wrap the App component with the BrowserRouter, we wont be able to use the Router functionality

root.render(

  <BrowserRouter>

      <App />

  </BrowserRouter>

);

1. In the App component, for different routes, use the **route** component as shown below

import './App.css';

import {Route} from 'react-router-dom';

import Home from './routes/Home';

import Products from './routes/Products'

function App() {

  return (

    <div>

      <Route path="/home">

        <Home/>

      </Route>

      <Route path = "/products">

        <Products/>

      </Route>

    </div>

  );

}

export default App;

**Working with links**

We have to wrap our links with the **Link** component from the react-router library. Otherwise using href trigger a get request and **all the state would be lost**

**Wrong approach:**

export default function Header() {

    // The href attribute would trigger a page refresh

    // which would reset the application states.

    // However the link and navlink of react-router doesn't trigger a page refresh.

    // Since React is used to create single page applications most of the time

    // make sure you choose Link or Navlink when working with routing

    return (

        <header>

            <nav>

                <ul>

                    <li>

                        <a href="/home">Home</a>

                    </li>

                    <li>

                        <a href="/products">Products</a>

                    </li>

                </ul>

            </nav>

        </header>

    )

}

**Correct Approach:**

In the below approach, we don’t generate a new request when we click the link and all the state is preserved.

import {Link} from 'react-router-dom'

export default function Header() {

    // The href attribute would trigger a page refresh

    // which would reset the application states.

    // However the link and navlink of react-router doesn't trigger a page refresh.

    // Since React is used to create single page applications most of the time

    // make sure you choose Link or Navlink when working with routing

    return (

        <header>

            <nav>

                <ul>

                    <li>

                        <Link to="/home">Home</Link>

                    </li>

                    <li>

                        <Link to="/products">Products</Link>

                    </li>

                </ul>

            </nav>

        </header>

    )

}

**Extracting Params from link**

**App.js (We want to extract the productID in the productDetails component)**

import './App.css';

import { BrowserRouter, Route } from 'react-router-dom';

import Home from './routes/Home';

import Products from './routes/Products'

import ProductDetails from './routes/ProductDetails'

import Header from './components/layout/Header'

// unless we wrap the component with the BrowserRouter, we wont be able to use the Router functionality

function App() {

  return (

    <BrowserRouter>

      <Header />

      <Route path="/home">

        <Home />

      </Route>

      <Route path="/products/:productID">

        <ProductDetails/>

      </Route>

      <Route path="/products">

        <Products />

      </Route>

    </BrowserRouter>

  );

}

export default App;

**Note:** When we enter the url: /products/productid, **BOTH THE COMPONENTS: ProductDetails and Products would be rendered. It is not like only the first component which matches with the route gets rendered. To handle this, we use Switch component in react-router-dom library (Refer below).**

import './App.css';

import { BrowserRouter, Route, Switch } from 'react-router-dom';

import Home from './routes/Home';

import Products from './routes/Products'

import ProductDetails from './routes/ProductDetails'

import Header from './components/layout/Header'

// unless we wrap the component with the BrowserRouter, we wont be able to use the Router functionality

function App() {

  return (

    <BrowserRouter>

      <Header />

      <Switch>

        <Route path="/home">

          <Home />

        </Route>

        <Route path="/products/:productID">

          <ProductDetails />

        </Route>

        <Route path="/products">

          <Products />

        </Route>

      </Switch>

    </BrowserRouter>

  );

}

export default App;

**ProductDetails.js**

import {useParams} from 'react-router-dom'

export default function ProductDetails(){

    // use react-router-dom's useParams hook to get access to the params

    const {productID} = useParams();

    return (

        <div>

            {productID}

        </div>

    )

}

**Switch and exact**

import './App.css';

import { BrowserRouter, Route, Switch } from 'react-router-dom';

import Home from './routes/Home';

import Products from './routes/Products'

import ProductDetails from './routes/ProductDetails'

import Header from './components/layout/Header'

// unless we wrap the component with the BrowserRouter, we wont be able to use the Router functionality

// Notice the order of productDetails component and products component. If products component is before productDetails component,

// then for all the paths which start with "/products", only the products component would be rendered.

// SO THE ORDERING OF THE COMPONENTS IS ALSO IMPORTANT

// But there is a prop called exact which can be used in switch component to match the exact path so that we wont have to worry about the ordering of the components

function App() {

  return (

    <BrowserRouter>

      <Header />

      <Switch>

        <Route path="/home">

          <Home />

        </Route>

        <Route path="/products/:productID">

          <ProductDetails />

        </Route>

        <Route path="/products">

          <Products />

        </Route>

      </Switch>

    </BrowserRouter>

  );

}

export default App;

**So the above code with exact prop can be written like this:**

import './App.css';

import { BrowserRouter, Route, Switch } from 'react-router-dom';

import Home from './routes/Home';

import Products from './routes/Products'

import ProductDetails from './routes/ProductDetails'

import Header from './components/layout/Header'

// unless we wrap the component with the BrowserRouter, we wont be able to use the Router functionality

// Notice the order of productDetails component and products component. If products component is before productDetails component,

// then for all the paths which start with "/products", only the products component would be rendered.

// SO THE ORDERING OF THE COMPONENTS IS ALSO IMPORTANT

// But there is a prop called exact which can be used in switch component to match the exact path so that we wont have to worry about the ordering of the components

function App() {

  return (

    <BrowserRouter>

      <Header />

      <Switch>

        <Route exact path="/home">

          <Home />

        </Route>

        <Route exact path="/products">

          <Products />

        </Route>

        <Route exact path="/products/:productID">

          <ProductDetails />

        </Route>

      </Switch>

    </BrowserRouter>

  );

}

export default App;

**Redirecting**

import './App.css';

import { BrowserRouter, Route, Switch, Redirect } from 'react-router-dom';

import Home from './routes/Home';

import Products from './routes/Products'

import ProductDetails from './routes/ProductDetails'

import Header from './components/layout/Header'

// if the path is '/', then the user should be redirected to '/home'

function App() {

  return (

    <BrowserRouter>

      <Header />

      <Switch>

        <Route exact path="/home">

          <Home />

        </Route>

        <Route exact path="/">

          <Redirect to='/home'/>

        </Route>

        <Route exact path="/products">

          <Products />

        </Route>

        <Route exact path="/products/:productID">

          <ProductDetails />

        </Route>

      </Switch>

    </BrowserRouter>

  );

}

export default App;

**Adding Not found Page**

import './App.css';

import { BrowserRouter, Route, Switch, Redirect } from 'react-router-dom';

import Home from './routes/Home';

import Products from './routes/Products'

import ProductDetails from './routes/ProductDetails'

import Error from './routes/Error'

import Header from './components/layout/Header'

// suppose any of the paths do not match, then we have to render an error component. The symbol \* is used to match any url

// and it should be placed in the LAST.

function App() {

  return (

    <BrowserRouter>

      <Header />

      <Switch>

        <Route exact path="/home">

          <Home />

        </Route>

        <Route exact path="/">

          <Redirect to='/home'/>

        </Route>

        <Route exact path="/products">

          <Products />

        </Route>

        <Route exact path="/products/:productID">

          <ProductDetails />

        </Route>

        <Route path="\*">

          <Error/>

        </Route>

      </Switch>

    </BrowserRouter>

  );

}

export default App;

**Implementing Programmatic Navigation**

Till now we have navigated to different paths using Link component. But there are many usecases where we want to navigate programmatically like when a form is submitted, the user should be redirected to the home page…etc. using **useHistory** hook.

import { useEffect } from 'react';

import { useHistory } from 'react-router-dom';

import QuoteForm from '../components/quotes/QuoteForm';

const NewQuote = () => {

  const history = useHistory();

  const addQuoteHandler = (quoteData) => {

    // history.push adds the page in the brower history stack and redirect to quotes page

    history.push('/quotes');

    // there is another method called replace in history object which replaces the current page in the browser's history.

    // That means, when we hit the back button, we wont go to the form page

    // history.replace('/quotes')

  };

  return <QuoteForm onAddQuote={addQuoteHandler} />;

};

export default NewQuote;