React is a JavaScript library for building user interfaces.

React makes building complex, interactive and reactive user interfaces simpler.

React is all about “Components”. Because all user interfaces in the end are made up of components.

Why use Components??

* Reusability, Separation of Concerns (don’t do many in one place).

React allows you to create **re-usable and reactive components consisting of HTML and JavaScript (and CSS).**

In react we will just create a html file with a div which will be having class “root” and with the help of this div we will create our own JSX which is also called as JavaScript Extensible Markup Language.

JSX is actually like HTML just creating it with the Help of JavaScript.

**index.js:**

import ReactDOM from ‘react-dom’; -> Imported React-dom library

import ‘./index.css’; -> Imported css file

import App from ‘./app.js’;

ReactDOM.render(<App />, document.getElementById(‘root’));

* Render Method on the ReactDOM object the second argument is a default Javascript DOM API which we are calling on the global document object. The first argument is a component (javaScript File). React renders HTML to the web page by using a function called render().

**App.js:**

In app.js we write a function where we write a JSX code and export it.

It is a first JSX component in our project which we always import in index.js to start our react app.(Root Component)

Body ->

const App = () => {

return(

<div>

<h2> Creating a JSX file </h2>

</div>

);

}

export default App;

here we can inherit other files too like making other components and importing it here and calling it in JSX code and using it. This help us to code React projects efficiently.

**Components and it’s Important Features:**

Components are just a JavaScript function it returns JSX code but it is just a JavaScript function.

**Rule in React:**

1. Lowercase elements are built in HTML elements and Uppercase elements are the elements defined by the developer.
2. Components can only return one div element. But inside of that one div we can make multiple div.
3. We can also include ‘./filename.css’ in Components
4. Here class are called as className = “nameofclass”;
5. The piece of code written in Javascript if we want to use it in JSX we can call it by using curly brackets

Like for Example:

Code in JavaScript-> Const expenseDate = new Date(2021,2,28);

Code in JSX -> <h2>{expenseDate.toISOString())</h2>

// using ISO to returns a string in simplified extended ISO format.

1. Let’s consider we have created an object in an components and now using JSX we pass that value

For Example:

Object->

const expenses = {

title = “Hello World”;

amount = 129.99;

date = new Date(2022,2,24);

}

And if we pass it from App.js we can pass the values to the child component by <Expenses items={expenses}/>

And in expense.js we can call title by using props.items[0].title

JSX ->

To pass values in ExpenseItem component we can write this piece of code ->

<ExpenseItem title={expense.title} amount={expense.amount} date={expense.date}></ExpenseItem>

In ExpenseItem Component->

During the call of arrow function you should it like this

Const ExpenseItem = (props) => {

Return(

<div>

<div>{props.toISOString()}</div> -> for Date printing taking input from App.js taking data from parent component to child component

);}

1. For date we might want to only show day or year or month to the user.

to do that we can make an const like

const month = props.date.toLocaleString(‘en-US’,{month: ‘long’});

const day = props.date.toLocaleString(‘en-US’,{day: ‘2-digit’});

const year = props.date.getFullYear();

1. Making an Card.js which in generally in React means a kind of container look with rounded corners, drop shadows and elements like these.

Create a **Card.js** file with **Card.css** file which will inherit the css code.

Card.js ->

import ‘./Card.css’;

const Card = (props) => {

const classes = ‘card’ + props.className; // Anything we receive from outside as a css file we add it in this card.js file with card’s css file’s data

return <div classname=”classes”>{props.children}</div>;

}

export default Card;

and in Other Component where we want to use Card.js we will there use it in the place of first div in that return JSX function. Out of the Box we can’t use the custom components as wrappers. To make it work we will take props in **Card.js’s** function and can call **{props.children}** between the div where children is the reserved name and the value of this special children prop will always be the content between the opening and the closing tags between the inherited component.

Why to use Card.js??

* To avoid code duplication in css files

1. We have imported ‘react-dom’ which is actually a part of ‘react’ this both modules combines and make react. With the help of reacrt-dom we can write the HTML or JSX code in JS files

We can also use ‘react’ import and can make an JSX code in an different way too.

For Example:

Return React.createElement(‘div’,{This will be the className or contraints we can say..},React.createElement(‘h2’,{},”Let’s get started!”),

React.createElement(‘Expenses’,{items={expenses}));

Actually ‘react-dom’ convert our JSX code internally into this code only. It just happens at the back of the compiling.

**State and Working with Events.**

1. We can add event listener in JSX code by using onClick={functionname} or any other event listener most of the eventlistener starts form letter on.
2. We cannot use a temp for any variable used in JSX like in our app we have tried to use title = props.title; and changed the value afterwards but this will not work because react does not work like that.
3. To make it use we import react specific function

i.e. import React, {useState} from ‘react’;

inside our component function we call useState(); it cannot be called outside of the function or any nested function (function inside the function).

So to change the title we can write the useState like this

const [title,setTitle] = useState(props.title);

const clickHandler = () => {

setTitle = ‘Updated!’;

}

in return block

<h2>{title}</h2>

Wherever both title and setTitle values are used these functions will be called again. It actually doesn’t change the value right away but instead schedules the state update.

Each component div or custom div is on the per component Instance Basis i.e. separate state for a separate div or custom div.

1. To take the input from the user we can use for example <input type=’text’ onChange={titleChangeHandler} />

And in function we just take input from argument in the form of event and we can target it using event.target.value

1. We can also use one state for multiple state where we declare

const [userInput, setUserInput] = useState({

enteredTitle: ‘’;

enteredAmount: ‘’;

enteredDate: ‘’;

});

const titleChangeHandler = (event) => {

setUserInput({

…userInput -> using spread operator to save the userInput date so that it will not get lost

enteredTitle: event.target.value,

})

}; -> If we use this approach we may be depending on an outdated or incorrect state snapshot because we have scheduled a lot of state updates at the same time

OR

Const titleChangeHandler = (event) => {

setUserInput((prevState) => {

return { …prevState, enteredTitle: event.target.value};

});

}; -> If we use this approach React will guarantee that the snapshot it gives you here will always be the latest state snapshot keeping all scheduled state updates in mind.

* We use this function form whenever our state update depends on the previous state.

1. If we use event.preventDefault(); the page will not reload during the submission of form.
2. Child to Parent Communication

We can create our own eventHandler in JSX like we have done in our project

<ExpenseForm onSaveExpenseData={saveExpenseDataHandler} />

Const saveExpenseDataHandler = (enteredExpenseData) => {

Const expenseData = {

…enteredExpenseData,

Id: Math.random().toString();

};

};

And in ExpenseForm.js file we have taken a props and after the function of expenseForm-> submitHandler we have declared a statement->

Props.onSaveExpenseData(expenseData);

1. Stateless or presentational or dumb are the components were we don’t use any useState or make any other function other than JSX return function(because it doesn’t have any internal state It’s just there to output some data) . Stateful or smart are the components were we output some data but make a use of useState and sent the data up to the parent or sent the data down to the child.

**Rendering Lists & Conditional Content**

1. We can use map to do the same function work repeatedly because what maps actually do is to do the same work form each and every object in an array.

For Example:

In Expense.js now we have converted all the hardcoded data into a map function where it do return JSX code for each array object.

{ props.item.map((expense) => (

<ExpenseItem

Title = {expense.title}

Amount = {expense.amount}

Date = {expense.date}

/>

))}

1. To make it dynamic we have used useState to update the array list. Initually we have renamed our initually array (Expenses) to an DUMMY\_EXPENSES and have taken it outside of the function. Now we make a useState function for it. Which we already as addExpenseHandler so no we update this function to add New Expenses in the array by ->

Const addExpenseHandler = (expense) => {

setExpense((prevExpenses) => {

return [expense, …prevExpenses];

});

};

1. We were getting some of the warnings in the console because React was warning us about the bugs not settled and data may loses and it is because React can’t say if the code which it have added is unique or not. So to overcome this problem we have add an key item in an map JSX code in an Expenses.js component so now React can surely say that the added Expense is unique and will not throw any warning in an Console.

New Map:

{props.items.map((expense) => (

<expenseItem

Key={expense.id}

Title={expense.title}

Amount={expense.amount}

Date={expense.date}

/>

))}

1. Whenever we do not have any Expense object in an array according to the year it will cause a problem because nothing will be displayed on the screen so to overcome this problem we can add ternary expression in an map method in expense.js

But before this we have added filteredExpenses = props.items.filter((expense) => {

Return expense.date.getFullYear().toString() === filteredYear;

});

* Which returns the objects according to the filtered date only…

Now map method will be

{<filteredExpenses.length === 0 ? ( <p> No expenses found. </p>)

:filteredExpense.map((expense) => (

<ExpenseItem

key={expense.id}

title={expense.title}

amount={expense.amount}

date={expense.date}

/>}

Or

{<filteredExpenses.length === 0 && ( <p> No expenses found. </p>)/>}

{<filteredExpenses.length > 0 && filteredExpense.map((expense) => (

<ExpenseItem

key={expense.id}

title={expense.title}

amount={expense.amount}

date={expense.date}

/>}

Or

We can make a function for it like this ->

let expensesContent = <p>No Expenses Found</p>;

  if (filteredExpenses.length > 0) {

    expensesContent = filteredExpenses.map((expense) => (

      <ExpenseItem

        key={expense.id}

        title={expense.title}

        amount={expense.amount}

        date={expense.date}

      />

    ));

  }

And JSX return we will just write {expensesContent}

1. In NewExpense.js add useState to add one more feature of Add New Expense button.

Adding Elements ->

Const [isEditing, setIsEditing] = useState(false);

Const startEditingHandler = () => {

setIsEditing(true);

};

Const stopEditingHandler = () => {

setIsEditing(false);

};

And now in JSX

Return (

<div className=”new-expense”>

{!isEditing && (

<button onClick={startEditingHandler}>Add New Expense</button>

)}

{isEditing && (

<ExpenseForm

onSaveExpenseData={saveExpenseDataHandler}

onCancle={stopEditingHandler}

/>

)}

</div>

);

And In ExpenseForm.js component just add cancel button in new-expense\_\_actions div ->

<div className=”new-expense\_\_action”>

<button type=”button” onClick={props.onCancel}>

Cancel

</button>

<button type=”submit”>Add Expense</button>

</div>

1. Adding Chart in the application. Create a Chart Folder. And create two files as chart.js and chartBar.js and their corresponding css files.

Return (

<div className=’chart’>

{props.dataPoints.map((datapoint) => (

<ChartBar

Key={datapoint.label}

Value={datapoint.value}

maxValue={null}

label={datapoint.label}

/>

))}

</div>

);

In chartBar.js

Const ChartBar = (props) => {

Let barFillHeight = ‘0%’;

If(props.maxvalue > 0){

carFillHeight = Math.round((props.value / props.maxValue) \* 100) + ‘%’;

}

Return <div className=”chart-bar”>

<div className=”chart-bar\_\_inner”>

<div className=”chart-bar\_\_fill” style={{height: barFillHeight}}></div>

</div>

<div className=”chart-bar\_\_label”>{props.label}</div>

</div>

};

1. To give value to Chart we create a ExpensesChart in Expenses folder

Const ExpensesChart = (props) => {

Const chartDataPoints = [

{label: ‘Jan’,value: 0},

{label: ‘Feb’,value: 0},

{label: ‘Mar’,value: 0},

{label: ‘Apr’,value: 0},

{label: ‘May’,value: 0},

{label: ‘Jun’,value: 0},

{label: ‘Jul’,value: 0},

{label: ‘Aug’,value: 0},

{label: ‘Sep’,value: 0},

{label: ‘Oct’,value: 0},

{label: ‘Nov’,value: 0},

{label: ‘Dec’,value: 0},

];

For (const expense of props.expenses) {

Const expenseMonth = expense.date.getMonth();

chartDataPoints[expenseMonth].value += expense.amount;

}

Return <Chart dataPoints={chartDataPoints} />

And Now in Chart.js update the maxValue feature

Create the function in Chart->

Const dataPointValues = props.dataPoints.map(datapoint => datapoint.value);

Const totalMaximum = Math.max(…dataPointValues);

And in maxValue={totalMaximum}

And Now add <ExpensesChart expenses={filteredExpenses}/> in Expenses.js

**STYLING REACT COMPONENTS**

1. We can directly add styles to the input or any component by using style={{JavaScript code with conditional and style(css)}}

<label style={{ color: !isValid ? "red" : "black" }}>Course Goal</label>

1. We can also make style in css file like we have done in our project

<div className={`form-control ${!isValid ? "invalid" : ""}`}>

And we have added the code in CSS file with invalid and valid

1. Styled Components: we have to do npm install --save styled-components in the terminal first. We do this to avoid multiple class same names styles. We have changed the button.js

Form

import React from 'react';

import './Button.css';

const Button = props => {

  return (

    <button type={props.type} className="button" onClick={props.onClick}>

      {props.children}

    </button>

  );

};

export default Button;

To

import styled from "styled-components";

const Button = styled.button`

  font: inherit;

  padding: 0.5rem 1.5rem;

  border: 1px solid #8b005d;

  color: white;

  background: #8b005d;

  box-shadow: 0 0 4px rgba(0, 0, 0, 0.26);

  cursor: pointer;

  &:focus {

    outline: none;

  }

  &:hover,

  &:active {

    background: #ac0e77;

    border-color: #ac0e77;

    box-shadow: 0 0 8px rgba(0, 0, 0, 0.26);

  }

`;

export default Button;

Adding styled component to the dynamic code

Import styled from ‘styled-components’;

const FormControl = styled.div`

  margin: 0.5rem 0;

  & label {

    font-weight: bold;

    display: block;

    margin-bottom: 0.5rem;

  }

  & input {

    display: block;

    width: 100%;

    border: 1px solid ccc;

    font: inherit;

    line-height: 1.5rem;

    padding: 0 0.25rem;

  }

  & input:focus {

    outline: none;

    background: #fad0ec;

    border-color: #8b005d;

  }

  &.invalid input {

    color: red;

    background: #ffd7d7;

  }

  &.invalid label {

    color: red;

  }

`;

Change div into <FormControl className={!isValid && “invalid”}>

…Code in between

</formControl>

OR

const FormControl = styled.div`

  margin: 0.5rem 0;

  & label {

    font-weight: bold;

    display: block;

    margin-bottom: 0.5rem;

    color: ${(props) => (props.invalid ? "red" : "black")};

  }

  & input {

    display: block;

    width: 100%;

    border: 1px solid ${(props) => (props.invalid ? "red" : "#ccc")};

    background: ${(props) => (props.invalid ? "#ffd7d7" : "")};

    font: inherit;

    line-height: 1.5rem;

    padding: 0 0.25rem;

  }

  & input:focus {

    outline: none;

    background: #fad0ec;

    border-color: #8b005d;

  }

`;

In JSX

<FormControl invalid={!isValid}>

1. Using CSS Modules

Just make another CSS file with css code and in JS file

Import styles from ‘./filename.module.css’;

Whereever we need to give class name to a div we have to give in this way ->

<div

        className={`${styles["form-control"]} ${!isValid && styles.invalid}`}

      >

**Diving Deeper working with ref’s and Portal**

1. Make a Wrapper.js file which will only return props.children and that can be used as an div instead of div and Actually this method is inbuild in React which can be used like <> </> and <Fragment></Fragment>
2. We can also have side by side div’s because having nested div’s is not a good practice. So to do this we create a div in index.html for example like in our project we have done we have created two div’s with an id=”backdrop-root” and id=”overlay-root”

Import reactDOM from ‘react-dom’;

Make an another own function of each of these constituents and in main return do {ReactDom.createPortal(<FunctionName onClick={props.onConfirm}/>,document.getElementById(‘backdrop-root’))}

1. We actually only use ref’s to take a data from user not to set anything. It is quite powerful and it will sure take less space than useState

It takes import React , {useRef} from ‘react’;

It actually replaces code

Const [enteredUsername, setEnteredUsername] = useState(‘’);

Const [enteredAge, setEnteredAge] = useState(‘’);

Const nameInputRef = useRef();

Const ageInputRef = useRef();

In JSX code ->

Value={enteredUsername}

onChange={usernameChangeHandler}

by -> ref={ nameInputRef }

value={enteredAge}

onChange={ageChangeHandler}

by-> ref = {ageInputRef}

change in addUserHandler we just replace everything by taking input from ref calling two constants

const enteredName = nameInput.current.value;

const enteredUserAge = ageInputRef.current.value;

and change the if conditions according to it

and at last

nameInput.current.value = “”;

ageInputRef.current.value =””;

**Handling Effects, Reducer and Context**

1. UseEffect can be used to handling side effect like we have used It in our project in Login.js

useEffect(() => {

const identifier = setTimeout(() => {

console.log(“checking for validity!”);

setFormIsValid(

enteredEmail.includes(“@”) &&

enteredEmail.include(“.”) &&

enteredPassword.trim().length > 6

);

}, 500);

Return () => {

Console.log(“Cleanup!”);

clearTimeout(identifier);

};

},[enteredEmail, enteredPassword]);

useEffect will take an pair where first one will be the function when the second one where there is an array of dependencies when these array of dependencies will be changes then this function will be executed

1. The function of useEffect with empty array [ ] will be executed at the very end of the component and only once when the webpages gets reloaded or restarts.
2. A normal useEffect function with no [ ] specified will be executed on every movement in an component
3. The useEffect with return function which is also called as cleanup function will be only executed when the total function will be executed at the very first the return code will not be executed.Whenever the dependencies changes this return code will be executed where we cleartimeout of the main function with every change in dependencies. This help us with to avoid sending too many http requests to backend server with every keystroke.
4. Whenever we want to depend on another state to set current state we should use useReducer

Const [state,dispatchFn] = useReducer(reducerFn, initualState, initFn);

State -> the state snapshot used in the component rerender/re-evaluation cycle

dispatchFn -> A function that can be used to dispatch a new action (i.e. trigger an update of the state)

reducerFn -> (prevState, action) => newState -> A function that is triggered automatically once an action is dispatched (via dispatchFn()) -> it receives the latest state snapsshot and should return the new, updated state.

initualState -> The initial state

initFn -> A function to set the initial state programmatically.

In our Login.js we can set useReducer in the place of useState of enteredEmail and emailIsValid.

**Outside of Login function**

Const emailReducer = (state,action) => {

If(action.type === “USER\_INPUT\_EMAIL”){

Return {

Value: action.val,

isValid: action.val.includes(“@”);

};

If(action.type ===”INPUT\_BLUR\_EMAIL”){

Return{

Value: state.value,

isValid: state.value.includes(“@”);

};

Return {value: “ “,isValid: false};

};

// Here state will be storing the previous value and action will be storing the current value which is passed by an object in the Input function

**Inside Login function**

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

value: “”,

isValid:undefined,

});

Const emailChangeHandler = (event) => {

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

};

Const validateEmailHandler = () => {

dispatchEmail({type:”INPUT\_BLUR”});

};

In JSX return code change value by {emailState.value} and emailState.isValid to check the validity of input.

1. When to use useState() and useReducer()

Generally, you will know when you need useReducer() (-> when using useState() becomes cumbersome or you’re getting a lot of bugs/ unintended behaviours)

useState()

1] The main state management “ tool ”.

2] Great for independent pieces of state/data.

3] Great if state updates are easy and limited to a few kinds of updates.

useReducer()

1] Great if you need “more power”.

2] Should be considered if you have related pieces of state/data.

3] Can be helpful if you have more complex state updates.

1. UseContext is actually a js file which is created for the availability of the information from a js component which is not related directly and need to go up to parent first and access it afterwards.
2. To access the data in context js file we have to wrap all the components JSX code which need it into the <fileNameofcontext.Provider> which will provide the data to the context file like we have used in our my-app4.

<AuthContext.Provider

      value={{

        isLoggedIn: isLoggedIn,

        onLogout: logoutHandler,

        onLogin: loginHandler,

      }}

Through this we can provide data to the context file when even we want to change it.

1. To access it we have two methods first one is a little bit lengthly and we can easily make an mistake here.

<AuthContext.Consumer>

{(ctx) => {

return (

 <nav className={classes.nav}>

<ul>

 {ctx.isLoggedIn && (

 <li>

<a href="/">Users</a>

 </li>

 )}

{ctx.isLoggedIn && (

<li>

<a href="/">Admin</a>

</li>

 )}

{ctx.isLoggedIn && (

 <li>

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

</li>

)}

</ul>

</nav>

);

}}

</AuthContext.Consumer>

1. To use it more Elegently we can import the file and include useContext and it it

Const ctx = useContext(fileName);

And replace all the props with ctx.

1. In further the videos we have moved all the code from App.js to auth-context file with adding the export function to it. We to make it possible we have to add

<AuthContextProvider>

    <App />

  </AuthContextProvider>

To index.js file to provide data from anywhere to anywhere.

1. Rules of React HOOKS.

1] We Only call React Hook in React Functions

We are allowed to use React hooks in React Component Functions and Custom Hooks

2] Only call React Hooks at the Top Level

1) Don’t call them in nested functions

2) Don’t call them in any block statements

3] + extra, unofficial Rule for useEffect(): ALWAYS add everything you refer to inside of useEffect() as a dependency!

1. useImperativeHandle which is used to focus on something or scroll we use useRef to send the data from input.js to this file where we have created a JSX code to make an box of input. And here we are going to make an function activate which will add focus on the current scenario if the condition is not true or not matched. We have to Make this function as React.forwardRef because we are taking the input from props and as well as refs too. And the useImperative function is written like this

const LoginJSX = React.forwardRef((props, ref) => {

  const inputRef = useRef();

  const activate = () => {

    inputRef.current.focus();

  };

  useImperativeHandle(ref, () => {

    return {

      focus: activate,

    };

  });

A LOOK BEHIND THE SCENE OF REACT.

1. How React and React-dom actually works is React-dom actually cares about what should be shown onto the screen. It’s react-dom which works with the JSX code. React only cares about the component in which it is not interested in what do it contains it is only Interested in in which position the state, props and context is being changed. React makes an virtual tree with the sub-tree which and whenever the state changes tree also changes so that it is being informed to the react-dom which do not re-evaluate the whole page like React it just update the new tree.

1. Export default React.memo(App); will we ensure that whenever the primitive type data does not changes which is being passed and if we use useCallback to ensure that the function, object and component are equals then we can use React.memo which will ensure if it is changed or not like it will save the previous function, object or component and will compare the new one and if there is a difference between them then it will be re-evaluated otherwise it will not be re-evaluated.
2. Const toggleParagraphHandler = useCallback(() => {

if(allowToggle){

setShowParagraph((prevShowParagraph) => ! prevShowParagraph), []);

this is how we use useCallback which is used to save and compare the previous snapshot of the component.

1. Const { items } = props;

useMemo(() => {

console.log(‘Items sorted’);

return items.sort((a,b) =>a + b;

},[items]);

This alone will not work because the araay we are sending is always a different one so to confirm it we have to also use useMemo there too.

Const listItem = useMemo(() => [5, 3, 1,10, 9], []);

Now this will ensure that it will be not changed.

**Class Method to write a component in React/ JavaScript**

1. We can convert the simple function for example

const User = (props) => {

  return <li className={classes.user}>{props.name}</li>;

};

To

import { Component } from "react";

import classes from "./User.module.css";

class User extends Component {

  componentWillUnmount() {

    console.log("User will unmount!");

  }

  render() {

    return <li className={classes.user}>{this.props.name}</li>;

  }

1. Here we cannot use hooks which is why classes method to write the useState, useEffect, useContext is a little different we can write useContext in classes too. But it is unstable.

import { Fragment, Component } from "react";

import classes from "./UserFinder.module.css";

import Users from "./Users";

import UsersContext from "../store/users-context";

class UserFinder extends Component {

  static contextType = UsersContext;

  constructor() {

    super();

    this.state = {

      filteredUsers: [],

      searchTerm: "",

    };

  }

  componentDidMount() {

    this.setState({

      filteredUsers: this.context.users,

    });

  }

  componentDidUpdate(prevProps, prevState) {

    if (prevState.searchTerm !== this.state.searchTerm) {

      this.setState({

        filteredUsers: this.context.users.filter((user) =>

          user.name.includes(this.state.searchTerm)

        ),

      });

    }

  }

  searchChangeHandler(event) {

    this.setState({ searchTerm: event.target.value });

  }

  render() {

    return (

      <Fragment>

        <div className={classes.finder}>

          <input type="search" onChange={this.searchChangeHandler.bind(this)} />

        </div>

        <Users users={this.state.filteredUsers} />

      </Fragment>

    );

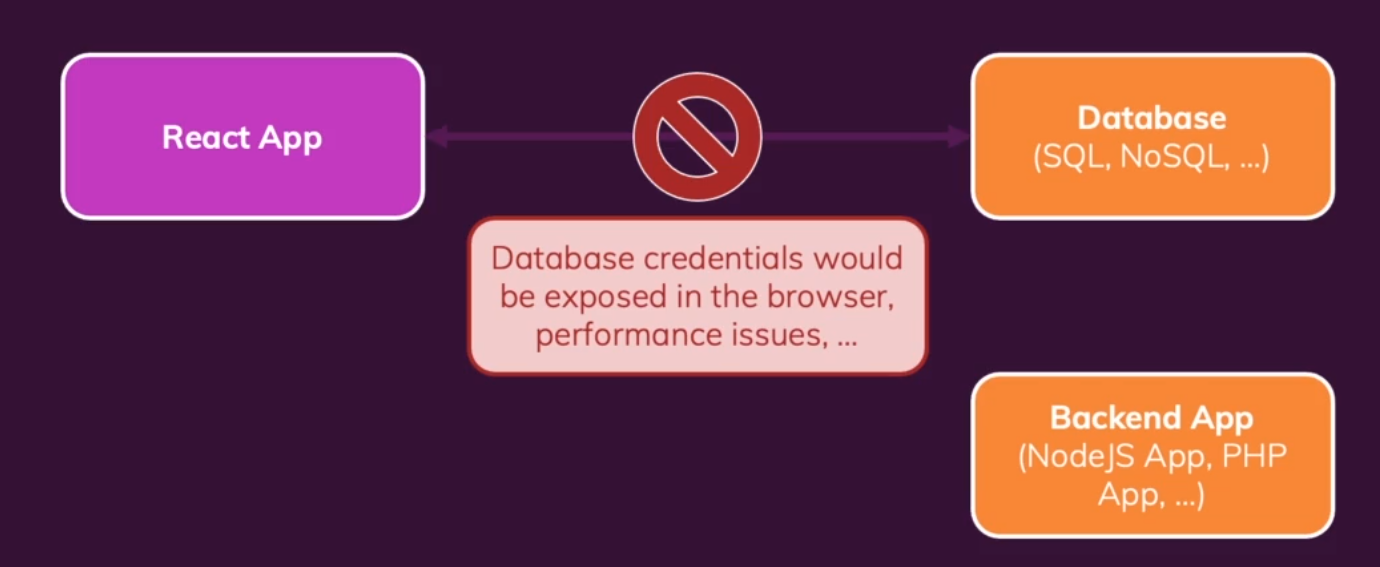
  }

}

1. Here we have used this.setState to setState just like useState. Here we have to setState by just accessing the single object which is mendatory in Classes to have and change the properties.
2. ComponentdidUpdate is used as useEffect and as the mount is used to set an data.

Sending an HTTP Requests

1. We cannot access any data directly through Database (SQL, NoSQL, Mango Js) because of Security issues and many more reasons so to access the data we use Backend App like NodeJS App, PHP App,…
2. We can send Http requests froms axios which is very simple to do.
3. But we have used fetch api which is built into the browsers. It allows us to fetch data and also send data.
4. So now we can say that there server api’s that will help us do talking to the database



1. const addMovieHandler = async (movie) => {

    const response = await fetch(

<https://swapi.dev/api/films/).then((response)> => {

return response.json();

}).then((data) => {

Const transformedMovies = data.results.map(movieData => {

Return {

Id: movieData.episode\_id,

Title: movieData.title,

openingText: movieData.opening\_crawl,

};

setMovies(transformedMovies);

}

So now here we have fetched data from swapi.dev where there is a data about it. And we have created a transformationMovies and set the data according to it because we have named it different in different file.And we have setMovies useState which will state the movies whenever we click on the button.

1. We can write the function fetchMoviesHandler also as

Async function fetchMoviesHandler() {

Const response = await fetch(‘https://swapi.dev/api/films/’);

Const data = await response.json();

And the same transformedMovies function.

This is a asynchronous function which will help us use data whenever the data is available.

1. Loading screen

Const [isLoading, setIsLoading] = useState(false);

And at the very start of the function of fetchMovieHandler() we will write setIsLoading(true); which will show the loading text or screen or anything that we have set as an loading.

And at the very end we will set our setIsLoading to false because loading is done at that time and we are ready to show the loaded data.

1. We will create the JSX code according to isLoading

<section>

{!isLoading && movies.length > 0 && <MoviesList movies={movies} />}

{!isLoading && movies.length === 0 && <p>Found no movies.</p>}

{isLoading && <p>Loading…</p>}

</section>

Firstly this code will show us No Movies found but when we click on fetchMovies button all the movies will be shown us after the loading…

1. Now whenever the website throw an error of any type like an 404 which shows that the request is fulfilled but the response which we have got is not expected by an server so the Loading… will be only shown on the screen no error code or anything will be shown so to show that we will create ->

Const [error, setError] = useState(null);

Here we are working with async await so we will use try-catch else if we could be using .then then we would be using catch.

Here we have created an error of fetching which will give us data but in transformedMovies function it will not give us error.message so to generate it we use response.ok

If(!response.ok) {

Throw new Error(‘something went wrong’);

}

And we wrap all this data into the try block and in catch block.

we setError(error.message);

and setIsLoading(false);

in JSX code we write {!isLoading && error &&<p>error</p>}

1. Now to show the data fetched at the very start we will use useEffect and call the function fetchMoviesHandler(); in that like

useEffect(() => {

fetchMoviesHandler();

}, []);

This will only load at the very start of the application and will new run again afterwards. But the problem will arise whenever we use it can create certein bugs if our code in that function is use some external state. To avoid any problem we use useCallback.

Now we will take all the code inside of the usecallback(async() => …code…, [])

1. Now If we want to add movies and that cannot be possible with the star war api so we just create a realtime firebase which will create a database for us which we can use to get or post the data.

Now the POST function be like =>

Async addMovieHandler = (movie) => {

Const response = await fetch(‘…Firebase..Link…’,{

Method: ‘POST’

Body: JSON.stringify(movie),

Headers: {

‘Content-Type’: ‘application/json’

}

Const data = await response.json();

Console.log(data);

});

Here the method will be post because we want to set the data in the database if we wanted to get the data then we could have used ‘get’, and to store the data there we will convert it into JSON by JSON.stringify(movie)

And set the headers as ‘Content-Type’: ‘application/json’.

And now the code we have written before in transformedMovies will not work because now we will get the data in Object so we have to configure it according to it.

Now we will create

Const loadedMovies = [];

For(const key in data) {

loadedMovies.push({

id: key,

title: data[key].title,

openingText: data[key].openingText,

releaseDate: data[key].releaseDate,

});

}

setMovies(loadedMovies);

CUSTOM HOOKS

1. Now custom hooks are created by naming the function starting with useName just like useCounter we have created before.

Import {useState, useEffect} from ‘react’;

Const useCounter = () => {

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

useEffect(() => {

const interval = setInterval(() => {

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

},1000);

Return () => clearInterval(interval);

},[]);

Return counter;

};

Export default useCounter;

And in ForwardCounter.js file we just call the useCounter hook and return <Card>{counter}<Card>;

Note: Above code is only for forwardCounter.

1. Now to create the useCounter hook for both the js files we will just update the file a little bit.

Import {useState, useEffect} from ‘react’;

Const useCounter = (forward = true) => {

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

useEffect(() => {

const interval = setInterval(() => {

if(forward){

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

}else{

setCounter((prevCounter) => prevCounter - 1);

}

},1000);

Return () => clearInterval(interval);

},[forward]);

Return counter;

};

Export default useCounter;

Here we hae initually assigned it as true if we do not assign any value to it.

And now we will go inside of the BackwardCounter.js file which will go ahead and call const counter = useCounter(false);

1. Now we are operating on another project where we are building the hook for sending and receiving the data from an server. So now we will create the use-http.js file where we have created a useHttp function and exported it.

Const useHttp = (applydata) => {

const [isLoading, setIsLoading] = useState(false);

  const [error, setError] = useState(null);

  const sendRequest = useCallback ( async (requestConfig) => {

    setIsLoading(true);

    setError(null);

    try {

      const response = await fetch(

        requestConfig.url, {

method: requestConfig.method ? requestConfig.method : ‘GET’,

headers: requestConfig.headers ? requestConfig.headers : {},

body: requestConfig.body ? JSON.stringify(requestConfig.body) : null

      );

      if (!response.ok) {

        throw new Error('Request failed!');

      }

      const data = await response.json();

applyData(data);

    } catch (err) {

      setError(err.message || 'Something went wrong!');

    }

    setIsLoading(false);

  }, [applyData]);

Return {

isLoading: isLoading,

error: error,

sendRequest: sendRequest,

};

Here we have changed the function in such a way that the function takes two argument where 1st one gives us the data we need like url and 2nd one gives us the function which do some particular task.And at last we have returned isLoading, error and sendRequest.

Now in App.js we will send the data and create the function

Const transformTasks = useCallback(tasksObj => {

Const loadedTasks = [];

For(const taskKey in taskObj){

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

}

setTasks(loadedTasks);

}, []);

Const { isLoading, error, sendRequest: fetchtasks} = useHttp({transformTasks);

useEffect(() => {

fetchTasks({url: ‘….. LINK …..’});

}, [fetchTasks]);

1. Now in NewTask.js where we add new tasks to the DataBase.

We include useHttp and

Const NewTask = (props) => {

Const createTask = (taskData) => {

const generatedId = taskData.name;

const createdTask = { id: generatedId, text: taskText };

props.onAddTask(createdTask);

}

Const {isLoading, error, sendRequest: sendTaskRequest} = useHttp(createTask);

Const enterTaskHandler = useCallback(async (taskText) => {

sendTaskRequest({url: ‘https://react-http-igjrgjejgijj/tasks.json’,method: ‘POST’, headers: {

‘Content-Type: ‘application/json’

},

Body: {text: taskText},

}), []);

WORKING WITH FORMS AND USER

1. Here we are going to see what’s form and what is soo hard about it

So when we submit a form or try to fill a form there are two possibilities

The first one is that **one or more inputs are invalid** and

So now for that we have output input-specific error message & highlight problematic inputs and ensure form can’t be submitted / saved.

The second one is that **All inputs are valid and Allow form to be submitted / saved.**

In form we can check input given by user in three points and that is

1. When form is submitted.

* Allows the user to enter a valid value before warning him/her
* Avoid unnecessary warnings but maybe present feedback “too late”

1. When a input is losing focus.

* Allows the user to enter valid value before warning him/her
* Very useful for untouched forms

1. On every keystroke.

* Warns user before he/she had a chance of entering valid values
* If applied only on invalid inputs, has the potential of providing more direct feedback.

1. Now we will try to update things when we submit the form. And to do that we will include onSubmit={formSubmissionHandler} on the form element and in that we can use useState properties if we want to also manipulate the after string like if the user have submitted the form and now he want to insert another name then automatically the input section will be empty. Or we will useRef if only we want the values from it.
2. Now we have just create an if statement where we have checked if the input is empty or not by enteredname.trim() === ‘’ and if it is then return it.
3. To add new functionalities

We will set an new state

Const [enteredNameIsValid, setenteredNameIsValid] = useState(true);

And now set it false in that if condition

And in jsx code we will add {!enteredNameIsValid && <p className=”error-text”>Name must not be empty.</p>

And include the nameInputClasses = enteredNameIsVAlid ? ‘form-control’ : ‘form-control invalid’;

And add className={nameInputClasses} in that div

1. But Initually we have set our useState as true which is not quite actually true or good. It makes more sense to make it false at start and to avoid the first instance showing error we will manage it by creating an new useState

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

And

Const nameInputIsInvalid = !enteredNameIsValid && enteredNameTouched;

Now change nameInputIsInvalid with the div and the error-text

{nameInputIsInvalid && (<p className=’error-text’>Name must not be empty.</p>)}

And setEnteredNameTouched to true in formSubmissionForm.

1. Now we will work on onBlur={nameInputBlurHandler}

Const nameInputBlurHandler = event => {

setEnteredNameTouched(true);

include the if statement and return;

};

1. Now we change the input on every keystroke

Const nameInputChangeHandler = event => {

setEnteredName(event.target.value);

if(event.target.value.trim() !== ‘ ’) {

setEnteredNameIsValid(true);

}

};

1. Now to clean our code we create a const nameInputChangeHandler in the place of enteredNameIsValid = enteredName.trim() !== ‘ ‘;

And we change the position of the nameInputIsInvalid after the nameInputChangeHandler and in the

formSubmissionHandler we setEnteredNameTouched(false);

1. Now we make an const [formIsValid, setFormIsValid] = useState(false);

And useEffect(() => {

If(enteredNameIsValid) {

setFormIsValid(true);

} else {

setFormIsValid(false);

}

}, [enteredNameIsValid]);

Or

Let formIsValid = false;

If (enteredNameIsValid) {

formIsValid = true;

}

1. Now we create hook instead of create function

Const useInput = validateValue => {

Const [enteredValue, setEnteredValue] = useState(‘ ‘);

Const [isTouched, setIsTouched] = useState(false);

Const valueIsValid = validateValue(enteredValue);

Const hasError = !valueIsValid && isTouched;

Const valueChangeHandler = event => {

setEnteredValue(event.target.value);

};

Const inputBlurHandler = event => {

setIsTouched(true);

};

Const reset = () => {

setEnteredValue(‘ ‘);

setIsTouched(false);

};

Return {

Value: enteredValue, hasError, isValid: valueIsValid, valueChangeHandler,inputBlurHandler,reset

};

now in SimpleInput.js

const {

value: enteredName,

hasError: nameInputHasError,

isValid: enteredNameIsValid,

valueChangeHandler: nameChangeHandler,

inputBlurHandler: nameBlurHandler,

reset: resetNameInput,

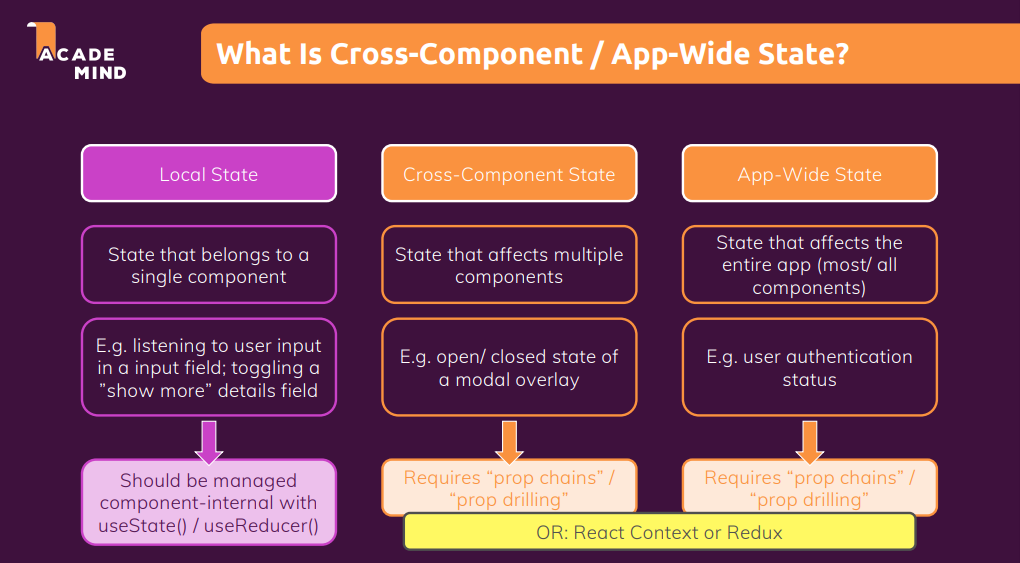
} = useInput(value => value.trim() !== ‘ ‘);

REDUX

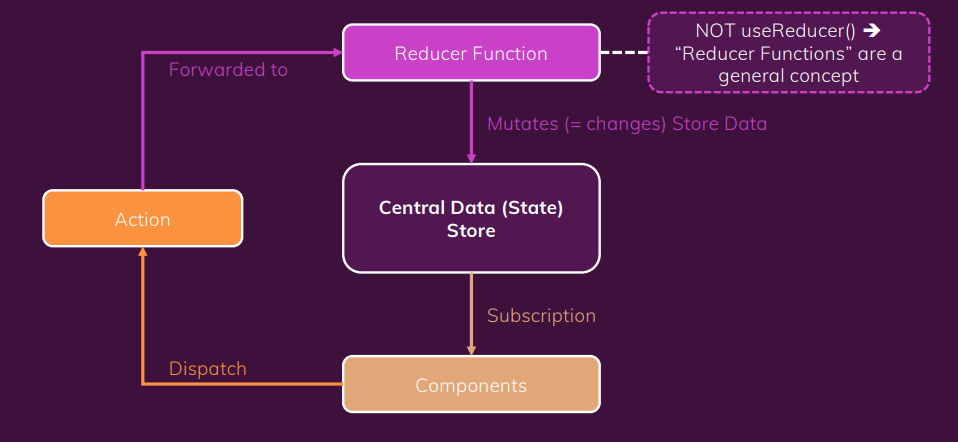
1. Redux -> A state management system forcross-component or app-wide state.

To include Redux in js file we use

const redux = require("redux");



we knew that redux uses Reducer as a part where it stores data in it.



Now, to create the simple Redux function we can just call the method

const store = redux.createStore(counterReducer);

and Here is the function of Redux which we have included in createStore.

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

  if (action.type === "increment") {

    return {

      counter: state.counter + 1,

    };

  }

  if (action.type === "decrement") {

    return {

      counter: state.counter - 1,

    };

  }

  return state;

};

const counterSubscriber = () => {

  const latestState = store.getState();

  console.log(latestState);

};

store.subscribe(counterSubscriber);

this will just call the function once but we have not created a function that can call subscribe method.

But we can also use dispatch like this

store.dispatch({ type: "increment" });

store.dispatch({ type: "decrement" });

which will give the output as

{ counter: 1 }

{ counter: 2 }

const initialCounterState = { counter: 0, showCounter: true };

const counterSlice = createSlice({

  name: "counter",

  initialState: initialCounterState,

  reducers: {

    increment(state) {

      state.counter++;

    },

    decrement(state) {

      state.counter--;

    },

    increase(state, action) {

      state.counter = state.counter + action.payload;

    },

    decrease(state, action) {

      state.counter = state.counter - action.payload;

    },

    toggleCounter(state) {

      state.showCounter = !state.showCounter;

    },

  },

});

The Reducer function.

const store = configureStore({

  reducer: { counter: counterSlice.reducer, Auth: authSlice.reducer },

});

export const counterActions = counterSlice.actions;

export const authActions = authSlice.actions;

The configureStore for other functions to access.

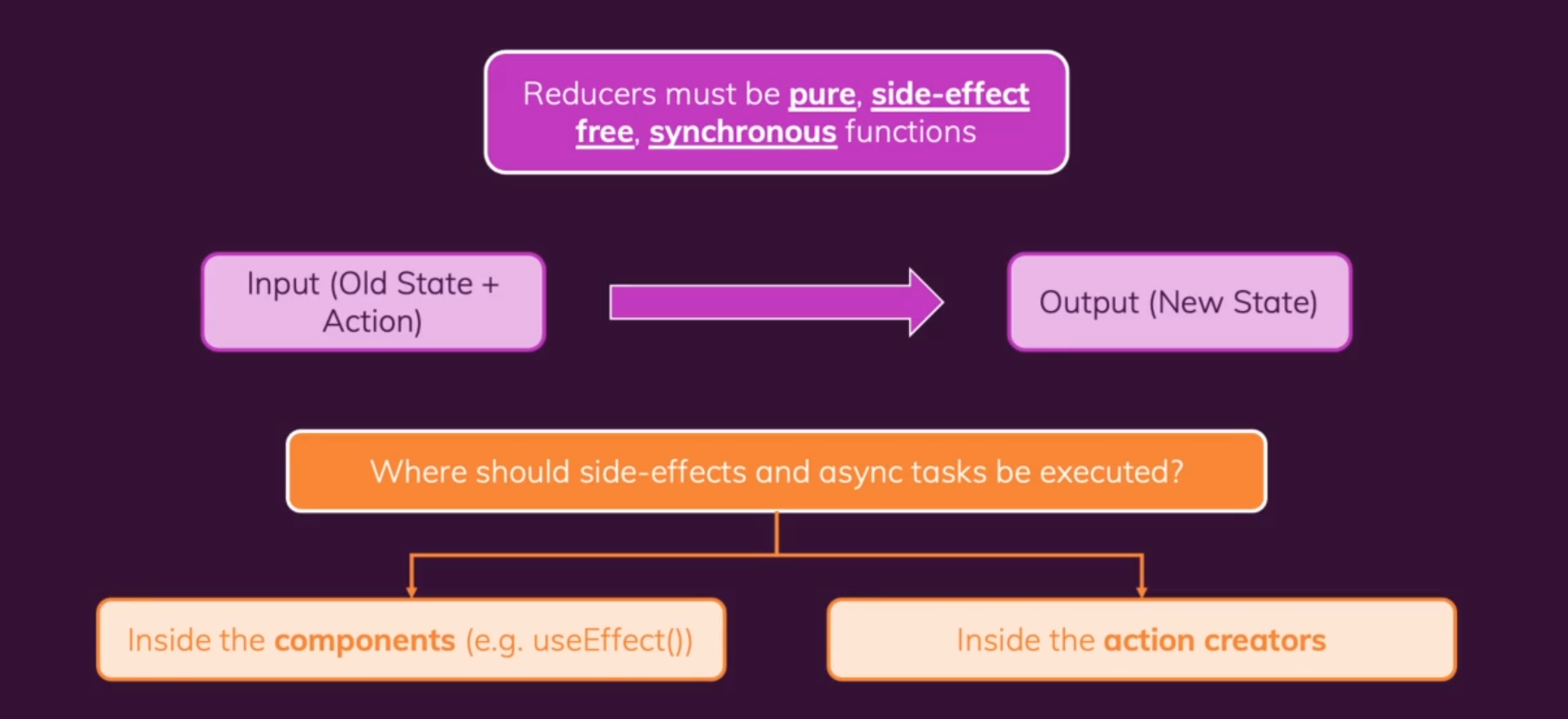
In Counter.js file we can

const dispatch = useDispatch();

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

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

se



1. Now we will fetch the data from the realtime database which is done by using the useEffect. Here we have used notification.js to show the status of the website like is the data successfully sent to realtime database or is it sending or it was a success or it failed to send the data.

useEffect(() => {

    const sendCartData = async () => {

      dispatch(

        btnActions.showNotification({

          status: "pending",

          title: "Sending...",

          message: "Sending cart data!!",

        })

      );

      const response = await fetch(

        "https://practice-http-de3f6-default-rtdb.firebaseio.com/cart.json",

        {

          method: "PUT",

          body: JSON.stringify(cart),

        }

      );

      if (!response.ok) {

        throw new Error("Sending cart data failed.");

      }

      const responseData = await response.json();

      dispatch(

        btnActions.showNotification({

          status: "success",

          title: "success!",

          message: "Sending cart data successfully!!",

        })

      );

    };

    if (isInitial) {

      isInitial = false;

      return;

    }

    sendCartData().catch((error) => {

      dispatch(

        btnActions.showNotification({

          status: "error",

          title: "Error!",

          message: "Sending cart data failed!!",

        })

      );

    });

  }, [cart, dispatch]);

1. But we have a alternative way to fetch the data too and that is using thunk

Thunk -> A function that delays an action until later

An action creater function that does not return the action itself but another function which eventually return the action.

