# **React With Java**

>> Create a Java class which store the data members receive from React frontend – java object. (1)

>> Create a java class which process the object from above class. (2)

>> Calling the above class at frontend using **rpc** established in project (3)

<https://nextjs.org/learn/dashboard-app>

Maven is a Java build automation tool, and npm is a Node. js package manager and build automation tool. Both tools are intended to automate the process of developing, testing, and deploying software, and they both include a plethora of built-in functionality for common build tasks.

**Spring**

In layman’s terms, it’s an application framework that helps you build Java applications with all bells and whistles really fast.

**Spring Boot**

Spring Boot is a solution for creating stand-alone, production-grade Spring-based Applications that you can "just run" with very little configuration. In simple terms, it can help you get started in building a REST API server in less than 30 minutes.

Example:

User need to create volume group, creating java class which stores the user input, and using this object data another class which responsible to invoke the CLI by calling API call to CLI creates the volume group based on input given by user

1. class CreateVolumeGroupInput - with private data members
2. class CreateVolumeGroupTask – at volumes package
3. launch (task:await rpc.volumes.createVolumeGroup( new CreateVolumeGroupInput(name, size, type), 123) );

**REST API:**

* They follow a client-server model where one software program sends a request and the other responds with some data.
* REST APIs commonly use the HTTP protocol to send requests & receive responses.

# **Java Script ES6**

Refer the best module – [ES6 by JS](https://www.javascripttutorial.net/es6/)

To run js on cmd use command Node <file.js>

### **Basics**

* [let](https://www.javascripttutorial.net/es6/javascript-let/) – declare block-scoped variables using the let keyword. With in {} braces scoped.
* [const](https://www.javascripttutorial.net/es6/javascript-let/) – declare block-scoped variables using the const keyword. But the value can’t be changed but returns new variable with value.
* var – declare global-scoped when declare outside of the function or block {}, can be used though out the class anywhere.

typesof(var) -can returns the type of an element. Instanceof

Arrow functions const fun = (…args)=>{}; here this couldn’t work.

const a = [1,2,3,4];

.map is used to return the value in object/array >> console.log(array.map(ar => ar+1));//[2,3,4,5,]

.filter is used to return the filtered object values >> console.log(a.filter(ar => (ar/2 ===1)));//2

Const date = new Date();

date.getFullYear(); date.getDay(); date.getMonth()+1;

let current date = `${date}-${month}-${year}` //String literals

Tip01: The advantage of using static functions or properties in a class is that:

* They can be used to create functions/properties which need not be present in the instances. This helps to maintain some isolation in the codebase.
* They reduce code redundancy in some cases.

**Tip02**: to iterate though the array need to use “of” like, for I of array. To iterate through the object need to use “in” for I in object.

in > iterates through key in object, where “of ” > iterates through the value of array.

let language = "JavaScript";  
  
let text = "";  
for (let x of language) {  
text += x;  
}

for(let c in  charObj){

        if(charObj[c]%2 ==1){

            odd++;

        }

    }

If nominator is less than the denominator then it the mod result is nominator only.

onEvent(“id”, “event”, function());

### **Class and Objects**

Classes were introduced in [ECMAScript 2015](https://262.ecma-international.org/6.0/) (ES6) to provide a cleaner way to follow object-oriented programming patterns. Class is blueprint of an object.

Classes are developed on behalf of constructor functions. In this type of implementation its hard to add functions. Like need to use Pen.prototype.showprice = ()=>

function Pen(name, color, price) {

this.name = name;

this.color = color;

this.price = price;

}

const pen1 = new Pen("Marker", "Blue", "$3");

console.log(pen1);

We can re-create the above example with the help of the class keyword. Have a look at the below code:

class Pen {

constructor(name, color, price){

this.name = name;

this.color = color;

this.price = price;

}

showPrice(){

console.log(`Price of ${this.name} is ${this.price}`);

}

}

const pen1 = new Pen("Marker", "Blue", "$3");

pen1.showPrice();

Implement JavaScript inheritance using **extends** and **super – here mostly can use static types.**

Objects: Objects in everyday life have properties and “method” actions. Take, for instance, a fan. It's an object with humidity.

Creating an object without class:

const person = {

‘name’:'kamal',

age:30,

friends:[

'Shola','Ade','Ibraheem'

],

greet:function(){

alert('Hello World')

}

}

Console.log(Person.name);

For key properties– dot - must a variable or key

To Access a property > person.age;

To Delete a property> delete person.age;

To Modify a property or adding property(if new)> person.age = 30;

For without key properties– it is achievable by square brackets: To Access a property > person[‘name’] and person[“age”]; -> must string

function getAge(obj) {

*//console.log(obj.name);*

  return obj["age"];

}

*//bracket notation is used to retrive using string not variable*

Object Spread Operator

The object spread operator is a popular and powerful syntax in JavaScript. The spread operator takes all the key-value pairs of an object and copies the key name and value into a new object.

Object Destructuring

Object destructuring is an important feature in JavaScript that allows you to pull out values from an object and assign them to individual variables.

const person = { name: 'lawal', age: 39 };

const { person, age } = person;

console.log(name); // 'lawal'

console.log(age); // 39

### **Arrays**

It stores multiple values and elements in **one** variable. These values can be of any data type — meaning you can store a string, number, boolean, and other data types in one variable.

There are two standard ways to declare an array in JavaScript. These are either via the array constructor or the literal notation and using new Array.

// Using array constructor

let array = new array("John Doe", 24, true);

// Using the literal notation

let array = ["John Doe", 24, true];

const cars = new Array("Saab", "Volvo", "BMW");

Accessing an element => array[0]

Changing an element => array[0] = 10;

Array to String => array.to String();

array.length; array.sort();

adding an element: returns new length deleting an element  
array.push(45); -adds at end array.pop() – remove at end

array.unshift(77 ); - adds at beginning array.shift() – removes at beginning

array1.concat(array);

Splicing and Slicing Arrays

Array.splice(start, removeCount, newItem, newItem, newItem, ...)

let months = ["January", "February", "Monday", "Tuesday"];

let days = months.splice(2);

console.log(days); // ["Monday", "Tuesday"]

fruits.splice(2, 0, "Lemon", "Kiwi");- The splice() method can be used to add new items to an array.

const citrus = fruits.slice(1); - The slice() method slices out a piece of an array into a new array.

Array Destructuring: (… rest parameter)

function getScores() {

return [70, 80, 90, 100];

}

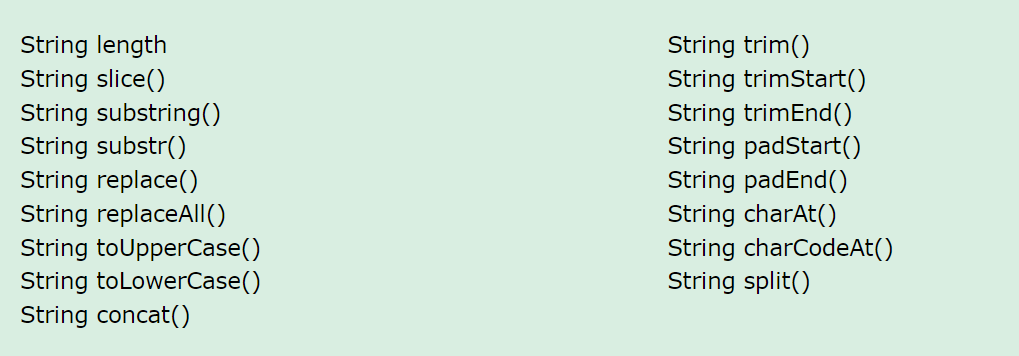
let [x, y, z] = getScores();

console.log(x); // 70

console.log(y); // 80

console.log(z); // 90

### **Strings**



### **Collections**

Map Object - the Map type that holds a collection of key-value pairs.

1. An object always has a default key like the [prototype](https://www.javascripttutorial.net/javascript-prototype/).
2. A key of an object must be a [string](https://www.javascripttutorial.net/javascript-string/) or a [symbol](https://www.javascripttutorial.net/es6/symbol/), you cannot use an object as a key.
3. An object does not have a property that represents the size of the map.

let userRoles = new Map([

[john, 'admin'],

[lily, 'editor'],

[peter, 'subscriber']

]);

Key is use din map methods=>map.set(john:’admin’), map.has(john);map.delete(john); map.clear()

let john = { name: 'John Doe' },

lily = { name: 'Lily Bush' },

peter = { name: 'Peter Drucker' };

let userRoles = new Map([

[john, 'admin'],

[lily, 'editor'],

[peter, 'subscriber'],

]);

for (const user of userRoles.keys()) { //can be used .values() {value to be returned} instead of .keys() or .entries {}

console.log(user.name);

}

for (let role of userRoles.values()) {

console.log(role);

}

for (const role of userRoles.entries()) {

console.log(`${role[0].name}: ${role[1]}`);

}

for (let [key, value] of chars.entries()) {

console.log(key === value);

}

chars.forEach(role => console.log(role.toUpperCase()));

Here are the main difference between a Map and a WeekMap:

* Elements of a WeakMap cannot be iterated.
* Cannot clear all elements at once.
* Cannot check the size of a WeakMap.

Set Object - the Set type that holds a collection of unique values.

let setObject = new Set();

let chars = new Set(['a', 'a', 'b', 'c', 'c']);

let size = chars.size;

chars.add('d');

let exist = chars.has('a');

console.log(exist);// true

chars.delete('d');

chars.clear();

for (let role of chars) {

console.log(role);

}

WeakSets

A WeakSet is similar to a Set except that it contains only objects. Since objects in a WeakSet may be automatically garbage-collected, a WeakSet does not have size property. Like a WeakMap, you cannot iterate elements of a WeakSet, therefore, you will find that WeakSet is rarely used in practice. In fact, you only use a WeakSet to check if a specified value is in the set.

Conditional loops and conditions:

If(i==10){  
}

Else if(i==11){

}

else{

}

while(i!=10){

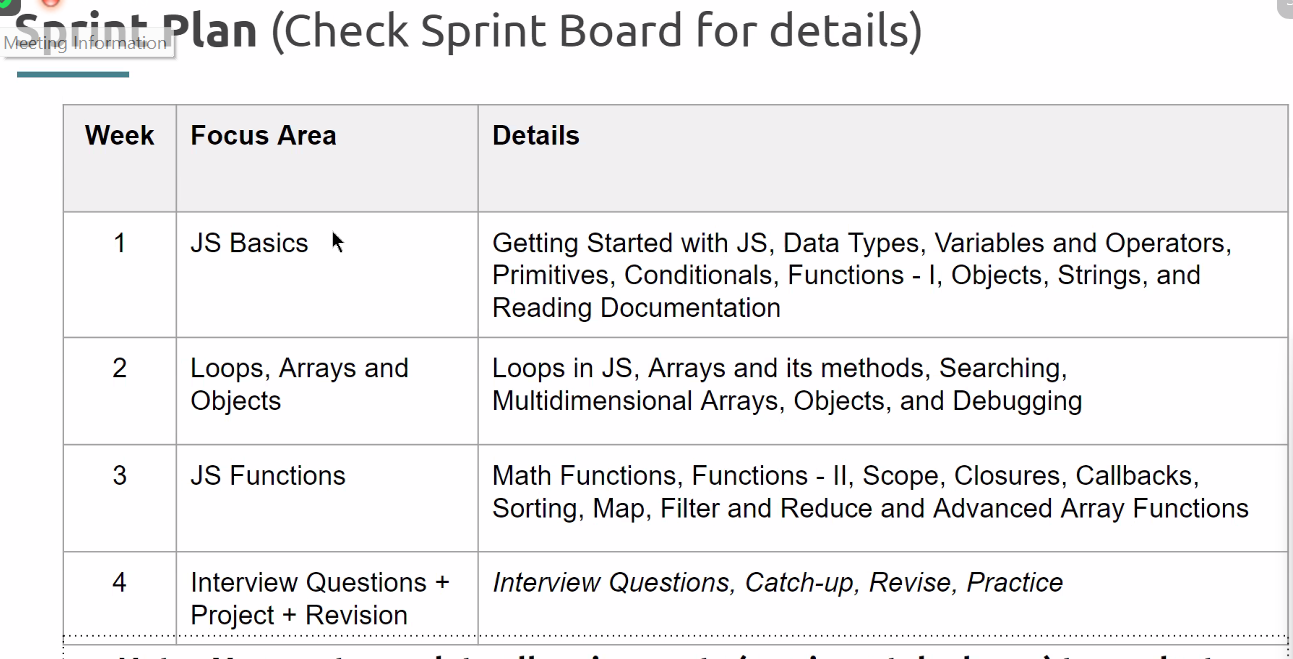
}

for(int i=0;i<10;i++){

}

switch(case){

}



incase requirement is returnig the mutlple values, then maping them into object is works, so the names must be same in object.

# JS Basics

### Getting Started with JS

# **React JS**

React components are JS functions which returns JSX code, these functions are render via dom and the a root is created using JS in index.html.

Refer the Git Hub repository - [**React\_Expenses\_Project**](https://github.com/sudhaScode/React_Expenses_Project)

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

### **Getting started with React**

**==========================================================**

To setup client by learning below commands

$mkdir client

$npx create-react-app expenses

$npm install {all dependencies like react, react-dom, and react-scripts, webpack webpack-dev-server webpack-cli --save}

$cd expenses

$npm start

===================================================================

setup server

$npm init -y

$npm install {all server dependencies like express cors nodemon}

$npm start

**==========================================================**

### **Components in React**

### **State, props, and Events**

* State management
* Components splitting and props
* Returning components conditionally

To send data from Parent to child can use JS variables inside the {} (JSX) at component calling**. [calling parent variable inside child component]**

To send data from child to parent can use event Listener functions with props at component calling, then calling that function inside of the child with object parameter(**props.parentfun(childdata)**), then the parent uses the objects inside it. **. [calling parent function inside child component, which handle the data send to that from child]**

Dynamic component return use achieved by useState hook and with ternary or conditional statements. And using .map can update list of objects for a component dynamically.

Dynamic style updating for components is achieved by inline style {{}} or className = {`exp+${with JS variable}`}

Dynamic styling React components

* >> **npm install –save styled-**components {import styled from ‘styled-components’;} const button = styled.button`css bricks`; inside back ticks for nested elements use &
* **Inline styling**
* **CSS Modules –** className = {props.styles}

<https://github.com/academind/react-complete-guide-code/tree/06-styling>

**Debugging React apps:** use console are result of npm start or npm run watch

**Styling:**

Inline CSS : style ={{}}

CSS-Module : className = {classes.button} [App.module.css]

Custom Error Handling : based on the value of the state, rendering a component

const submitHandler =(event)=>{

        event.preventDefault();

        if(userName.length === 0 || userAge.length === 0){

            setError({

                title:"Invalid input",

                message:"Please enter a valid name and age (non-empty values).",

            });

            setUsername(null);

            setUserage(null);

        }

        else if(+userAge<1){

            setError({

                title:"Invalid age",

                message: "  please enter a valid age > 0",

            });

            setUsername(null);

            setUserage(null);

        }

        else{

        const userData ={

            name:userName,

            age: userAge,

        }

        props.onAdd(userData);

        setUsername('');

        setUserage('');

    }

    }

**UI Cards:**  
it uses a props.children as data

UI cards are used to wrap the components in card components, it is possible by passing the dynamic classnames to extending the CLASS NAME of UI card component.

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

function Page(props){

    return (

        <div className={`${classes.card} ${props.className}`}>{props.children}</div>

    );

}

export default Page;

.card{

    background: rgb(63, 63, 63);

    box-shadow: 0 2px 8px rgba(0,0,0,0.26);

    border-radius: 10px;

}

Wrapping among individual components

<Page className={classes.data}> components </Page>

### **Fragments, Portals & Refs -Hooks, React-Context**

Hooks are helper functions.

Types of Hooks:

1. **State Hooks** – useState(); useReducer()

[useState](https://react.dev/reference/react/useState) declares a state variable that you can update directly.

[useReducer](https://react.dev/reference/react/useReducer) declares a state variable with the update logic inside a [reducer function.](https://react.dev/learn/extracting-state-logic-into-a-reducer)

1. **Context Hooks** – useContext()

Context lets a component [receive information from distant parents without passing it as props.](https://react.dev/learn/passing-props-to-a-component)

1. **Ref Hooks** – useRef(), useImaprativeHandle() - Refs let a component [hold some information that isn’t used for rendering](https://react.dev/learn/referencing-values-with-refs)

[useRef](https://react.dev/reference/react/useRef) declares a ref. You can hold any value in it, but most often it’s used to hold a DOM node.

[useImperativeHandle](https://react.dev/reference/react/useImperativeHandle) lets you customize the ref exposed by your component. This is rarely used.

1. **Effect Hooks –** useEffect(), Effects let a component [connect to and synchronize with external systems.](https://react.dev/learn/synchronizing-with-effects)

[useEffect](https://react.dev/reference/react/useEffect) connects a component to an external system.

1. **Performance Hooks -**useMemo(), useCallback() . A common way to optimize re-rendering performance is to skip unnecessary work.

To skip calculations and unnecessary re-rendering, use one of these Hooks:

* [useMemo](https://react.dev/reference/react/useMemo) lets you cache the result of an expensive calculation.
* [useCallback](https://react.dev/reference/react/useCallback) lets you cache a function definition before passing it down to an optimized component.

1. Resource Hooks – use(). Resources can be accessed by a component without having them as part of their state.

To read a value from a resource, use this Hook:

* [use](https://react.dev/reference/react/use) lets you read the value of a resource like a [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) or [context](https://react.dev/learn/passing-data-deeply-with-context).

**Other Hooks**

These Hooks are mostly useful to library authors and aren’t commonly used in the application code.

* [useDebugValue](https://react.dev/reference/react/useDebugValue) lets you customize the label React DevTools displays for your custom Hook.
* [useId](https://react.dev/reference/react/useId) lets a component associate a unique ID with itself. Typically used with accessibility APIs.
* [useSyncExternalStore](https://react.dev/reference/react/useSyncExternalStore) lets a component subscribe to an external store.

useState hook updates the value of the variable in next re-render, where useRef make updates the value instantly without waiting for re-render it don’t cause to rerender – no display of update value – when just to read .

const usernameRef =useRef();

ref ={usernameRef}

enteredName = usernameRef.current.value;

React.forwardRef(( props, ref)

|  |  |
| --- | --- |
| **useState()** | **useReducer()** |
| * The main state management “tool”. * Great for independent pieces of state/data. * Great if state updates are easy and limited to a few kinds of updates. * Causes a re-render. | * Great if you need to add more power. * Should be consider if you had related pieces of data/state. * Can be helpful if you have more complex state updates. |

<React.Fraagment>

</React.Fraagment>

const root = ReactDOM.createRoot(document.getElementById(‘root’));

root.render(<React.StrictMode> <App/></React.StrictMode>);

<https://github.com/academind/react-complete-guide-code/tree/09-fragments-portals-refs/code>

sideEffects:

useEffect: useEffect ((()=>{ //The code will run here},[//dependency])); When component mounted at least the useEffect blocked to code will runs , if dependency is given useeffect listens the change of the array elements, as per the change useeffect re-renders the updated components.

To make a component responsive with side effects(user interacts), based on current state of component.

**Cleanup** function in useEffect , use return at code block.

useEffect(()=>{

const identifier = setTimeout(()=>{

console.log(‘checking form validity’);

setFormInvalid(enteredEmail.includes(‘@’) && enteredPassword.trim().length>6);

}, 500);

return ()=>{

cosole.log(‘clean up!’);

clearTimeout(identifier);

};

}, [enteredEmail, enteredPassword]);

**useReducer:** One reducer equal to two useState hook, only one reducer is enough to use&set and validate the state

cosnt [state, dispatchFn] = useReducer(resucerFn, initialSate, initFn);

useCallback: this hook is used to improve the performance of an component by saving the implementation and invoking the implementation based on the state changes of component variables(keeping as dependencies[]). Normally used for functional changes.

usecallback(()=>{},[]);

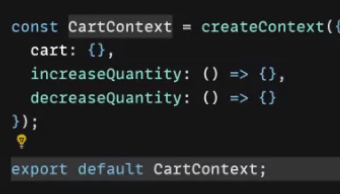
useMemo also a similar hook used to improve performance by optimizing complex calculation based on dependencies.

React.memo is used to trace the props changes send to the child component and if changes are made to the props values then the component re renders other wise it won’t allow to re render. It is used at exporting syntax. **export default React.memo(App);**

**React Context: Manage state (Shared state among componenets)**

**To create context:**

Example:



**To use context:**

Create a Reducer(state, action) function and pass to useReducer.

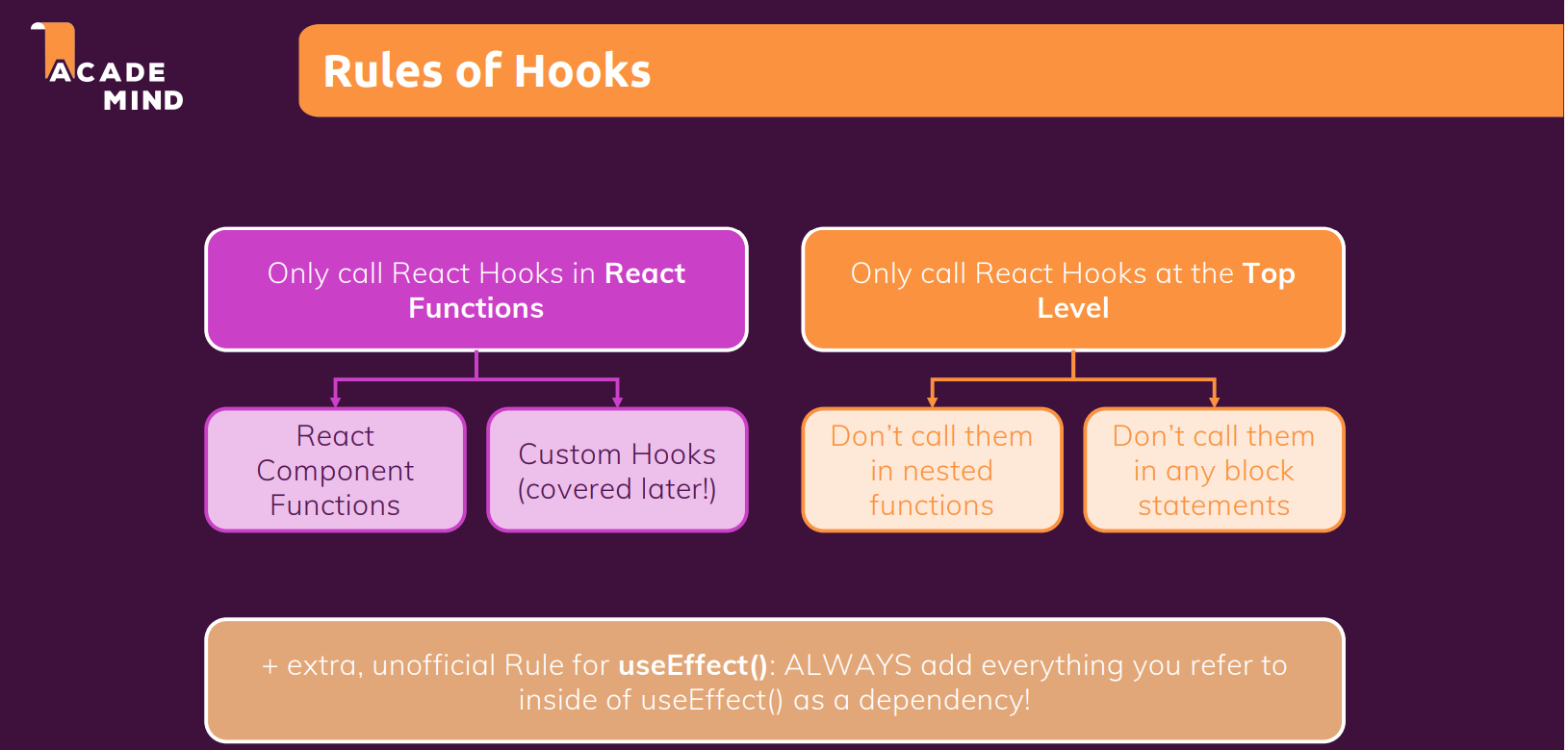
**const ctx = useContext(**React.createContext());

const   [cartSate, dispatchCartAction] =useReducer(cartReducer, defaultCartState);

**To update context:**

<CartContext.Provider> some code to share the global state share</ CartContext.Provider >

**const ctx = useContext(**React.createContext());



<https://github.com/academind/react-complete-guide-code/tree/10-side-effects-reducers-context-api>

**Backdrop and overlay in React application:**

**Add overlay id in index.html at public folder.**

import styles from './Modal.module.css';

import { Fragment } from 'react';

import ReactDOM from 'react-dom';

const Backdrop =props=>{

        return <div className={styles.backdrop} onClick ={props.onClose}></div>

};

const ModalOverlay  = props =>{

    return(

        <div className={styles.modal}>

            <div className={styles.content}>{props.children}</div>

        </div>

    );

};

const portalElement = document.getElementById('overlays');

const Modal = props=>{

    return(

    <Fragment>

        {ReactDOM.createPortal(<Backdrop onClose={props.onClose}/>, portalElement)}

        {ReactDOM.createPortal(<ModalOverlay>{props.children}</ModalOverlay>, portalElement)}

    </Fragment>

    );

};

export default Modal;

Using context hook getting the props of distant component is availed by following process: (Food- Order project)

Follow Up 1: create context

const Cartcontext = React.createContext({

    items:[],

    totalAmount:0,

    addItem: (item) =>{},

    removeItem: (id)=>{},

})

This Cartcontext works as a wrapper element.

Follow Up 2: create a component for provider, here the prop objects are handled

import Cartcontext from "./cart-context"

import { useReducer } from "react";

const defaultCartState = {

    items:[],

    totalAmount:0,

};

const cartReducer = (state, action) =>{

    if (action.type === 'ADD'){

        const updatedItems = state.items.concat(action.item);

        const updatedTotalAmount = state.totalAmount + action.item.price\*action.item.amount;

        return{

            items:updatedItems,

            totalAmount:    updatedTotalAmount

        };

    }

    return defaultCartState;

};

const CartProvider = props=>{

    const   [cartSate, dispatchCartAction] =useReducer(cartReducer, defaultCartState);

    const addItemToCartHandler =(item) =>{

        dispatchCartAction({type:'ADD', item: item })

    };

    const removeItemFromCartHandler =(id) =>{

        dispatchCartAction({type: 'REMOVE', id: id})

    };

    const cartContext = {

        items:cartSate.items,

        totalAmount: cartSate.totalAmount,

        addItem: addItemToCartHandler,

        removeItem: removeItemFromCartHandler,

    };

    return(

        <Cartcontext.Provider value={cartContext}>

            {props.children}

        </Cartcontext.Provider>

    );

}

export default CartProvider;

Follow Up 3: Fast forwarding the user input by useRef

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

import React from 'react';

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

    return(

        <div className={classes.input}>

            <label htmlFor={props.input.id}>

               {props.label}

            </label>

            <input ref ={ ref} id ={props.input.id} {...props.input}/>

        </div>

    );

});

export default Input;

Follow Up 4: get the user input from child to parent and update the context object:

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

import MealItemForm from './MealItemForm';

import { useContext } from 'react';

import Cartcontext from '../../../store/cart-context';

const MealItem = props =>{

    const cartCtx = useContext(Cartcontext);

    const price = `$${props.price}`;

    const onAddtoCartHandler = amount =>{

        cartCtx.addItem({

            id: props.id,

            name: props.name,

            amount:amount,

            price: props.price

        });

    };

    return(

        <li className={classes.meal}>

            <div>

                <h3>{props.name}</h3>

                <div className={classes.description}> {props.description}</div>

                <div className={classes.price}>{price}</div>

            </div>

            <div>

                <MealItemForm id = {props.id} onAddtoCart ={onAddtoCartHandler}/>

            </div>

        </li>

    );

}

export default MealItem;

Follow Up 5: render the context property in current component where needs to render:

import { useContext } from "react";

import CartIcon from "../Cart/CartIcon";

import styles from './HeaderCartButton.module.css';

import Cartcontext from "../../store/cart-context";

const HeaderCartButton = props =>{

    const cartCtx = useContext(Cartcontext);

    const numberofCartItems = cartCtx.items.reduce((current,item)=>{

        return current+item.amount},0);

    return(

        <button className={styles.button} onClick ={props.onClick}>

            <span className={styles.icon}>

                <CartIcon/></span>

            <span>Your Cart</span>

            <span className={styles.badge}>{numberofCartItems}</span>

        </button>

    );

};

export default HeaderCartButton;

**Conditional Rendering with logical && and Ternary operator can be implemented.**

### UI CARD

This type of components are used to wrap the react components to maintain consistency.

Create a card with props.childeren,

function Card (props){

    return (

        <div className = {`${styles.card} ${props.className}`}>

           {props.children}

        </div>

    );

}

.card{

    border-radius: 4px;

    background: wheat;

    box-shadow: 0 2px 8px rgba(160, 129, 129, 0.26);

}

Where above component is wrapped to component with claasName , the className contains css properties

### Class based components

class comp extends components{\

constructor(){

super();

this.state = { hasError:false};

}

render(){

return(

<h1>Class component<h1>

)  
 }

}

Class based components can’t use React Hooks.

<https://github.com/academind/react-complete-guide-code/tree/13-class-based-cmp>

### Database connectivity for React Apps

**Using the Star Wars API**

In the next lecture, you will be introduced to our demo backend that will be used in this course section: The Star Wars API.

I will use this page: <https://swapi.dev/>

Loading this page (and hence accessing this backend) **might fail** - if that is the case for you, you can **use this alternative**: <https://swapi.py4e.com/>

<https://github.com/academind/react-complete-guide-code/tree/14-sending-http-requests>

### **API FECTH CALL:**

This fetch call is implemented for routing components.

export async function loader(){

    const response = await fetch('http://localhost:8080/events');

    if(!response.ok){

     // return {isErorr: true, message:"Couldn't"}

     throw new Response(JSON.stringify({title: "error", message:"Couldn't load data"}),

     {status: 500})

    }

    else{

      //  const resData = await response.json();

        return response;

    }

}

API POST CALL:

This post call is implemented to get data from nested route component using request.formData() and sending to server.

export async function action({request, params}){

    const data = await request.formData();// getting data from router component where action property is initialized.

    const eventData = {

      title: data.get('title'),

      image: data.get('image'),

      date: data.get('date'),

      description: data.get('description')

    }

    console.log(eventData)

const response = await fetch('http://localhost:8080/events',  {

  method:'POST',

  headers: {'Content-Type': 'application/json'},

  body: JSON.stringify(eventData),

});

if(!response.ok){

  throw json({title:"Server is not responding",message:"Data can't sent" }, {status: 500} );

}

return redirect('/events');

}

Delete API Call:  
this delete API call implemented for route level action using submit hook

export async  function action({request , params}){

  const id = params.eventid; // it takes path id from route deffination

  console.log(id);

  const response = await fetch('http://localhost:8080/events/'+id, {method:request.method,});

  if(!response.ok){

    throw   json({message:"deleting is failed"},{status: 500});

  }

  return redirect('/events');

}

PUT/PATCH CALL:

export async function action({request, params}){

    const data = await request.formData();

    const method = request.method;

    const eventData = {

      title: data.get('title'),

      image: data.get('image'),

      date: data.get('date'),

      description: data.get('description')

    }

    console.log(method)

    let url =   'http://localhost:8080/events';

    if(method === 'PATCH'){

      const eventid = params.eventid;

      url ='http://localhost:8080/events/'+ eventid;

    }

const response = await fetch(url,  {

  method: method,

  headers: {'Content-Type': 'application/json'},

  body: JSON.stringify(eventData),

});

if(response.status === 422){

     return response;

}

if(!response.ok){

  throw json({title:"Server is not responding",message:"Data can't sent" }, {status: 500} );

}

return redirect('/events');

}

Custom Hooks:

<https://github.com/academind/react-complete-guide-code/tree/15-building-custom-react-hooks>

https://academind.com/tutorials/reactjs-a-custom-useform-hook/

const value = useCustom();

## Redux:

What is Redux?

A State management system for cross-component or app-wide state.

**Local State:**  
State that belongs to a single component.

Eg: listening to user input in a input filed; toggling a “show more ” details filed.

Should be managed component-internal with useState() / useReducer().

**Cross-Component State:**

State that effects multiple components.

Eg. Open/Closed state of a modal overlay.

Requires props chains/ props drilling./React context or Redux

**App-Wide State**

State that affects the entire app (most/all components)

Eg: user authentication status

Requires props chains/ props drilling. /React context or Redux

**Why Redux? React Context potential disadvantages:**

>>>Complex Setup by context

>>> Performance – not recommended for high frequency apps

**Central Data (State) Store:**

Indeed, the React Redux team now recommends the usage of an extra package called **Redux Toolkit** and another way of creating the Redux store.

**To create store**: import { configureStore } from '@reduxjs/toolkit';

>>> createStore, reducer function => state

* 1. Create a initial state with needed state variables
  2. Using the initial state update the value using reducer function

Reducer must be pure, side effect free, synchronous functions

import { configureStore } from '@reduxjs/toolkit';

const initialState = { counter: 0, showCounter: true}; // Initial state for your reducer

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

  if (action.type === 'increment') {

    return {

      counter: state.counter + 1,

      showCounter: state.showCounter

    };

  }

  if (action.type === 'increase'){

    return{

      counter: state.counter + action.amount,

      showCounter: state.showCounter

    }

  }

  if (action.type === 'decrement') {

    return {

      counter: state.counter - 1,

      showCounter: state.showCounter

    };

  }

  if (action.type === 'toggle'){

    return{

      showCounter: !(state.showCounter),

      counter: state.counter

    }

  }

  return state;

};

const rootReducer = {

  counter: counterReducer,

  // You can add more reducers here if needed

};

const store = configureStore({

  reducer: rootReducer,

  // Other store configuration options if needed

});

export default store;

**To update Stor,e:** , useDispatch() and dispatch ({action.type=== ‘incremanet’})

**To use Store:** import {useSelector, useDispatch} from 'react-redux';

>>> useSelector(state. counter. counter),

## **Redux Toolkit**

**To configure store:**

configureStore();

assign a object with reducer property

const store = configureStore({

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

  // Other store configuration options if needed

});

export const counterActions = counterSlice.actions;

export const authActions = authSlice.actions;

export default store;

**To create Store:**

Declare a initial state

Using createSlice implement the functions in reducers.

import { createSlice } from '@reduxjs/toolkit';

const initialAuthActionState = {isAuthenticated: false};

const authSlice = createSlice({

    name: "authentication",

    initialState: initialAuthActionState,

    reducers:{

      login(state){

        state.isAuthenticated = true;

      },

     logout(state){

      state.isAuthenticated = false;

     },

    }

  });

  export default authSlice;

**To use and update Store:**

useDisptach(), useSelector(),

Update

 const dispatch=useDispatch();

    const signinHandler = ()=>{

        dispatch(authActions.login());

    };

    const signoutHandler =()=>{

        dispatch(authActions.logout());

    }

Use

const auth = useSelector(state => state.auth.isAuthenticated);

The auth is the name that we give to reducer in configure store.

## Advanced Redux

* Handling async tasks with redux
* The Redux dev tools

**Redux and Async Code:**

Reducer must be pure, side effect free, synchronous functions and any aysnc API calls must not be run in reducers.

Instead of calling API calls in reducers functions implement inside components (like useEffects()) or inside action items.

Where should our logic (code) go?

|  |  |
| --- | --- |
| Synchronous, side-effect free code (i.e data transformation) | Asynchronous, side-effect code |
| Prefer Reducers  Avoid actions creators and components | Avoid Reducers  Prefer actions creators and components |

**Redux DevTools: extension**

## React Router – Single page application routing

Client-side routing – using React-Router package. -**npm install react-router-dom**

**What is Routing** - Changing the different URL as per client request.

**Defining Router:**

**Object based approach:**

Import createBrowserRouter =>>> const router = createBrowserRouter([\_PATH\_]);

const router = **createBrowserRouter**([

{ path :’/’, element : <HomePage/>}

{path: ‘/products’, element: <ProductPage/>}

]);

**<RouterProvider router ={router}/> -** it is interface component to call and render the nav component

**<Outlet/>** is responsible to rendering the route defined component

**To make dynamic paths use “:” colon >** ‘products/:poductId’

**Link** feature in react-router-dom, to replace anchor tag(<a><a/>, which cause a requesting html pages from server)

**NavLink -**Updated version of Link

<li>

            <NavLink to = "/"

            className={({isActive})=>isActive? classes.active:undefined} end >Home</NavLink>

            </li>

**Dynamic Routing**

**const params = useParams()** >> to use url endpoint id params.productId

const params = useParams();

<p>{params.id}</p>

**useNavigate –** to navigate programmatically

const navigate = useNavigate();

   const navigateHandler = ()=>{

      navigate('/events/new');

   }

**useNavigation –**

const navigation = useNavigation();

const isSubmitting = navigation.state === 'submitting';

**Relative and Absolute paths**

<Link to = "../.." relative="path">Back</Link>

In relative path the parent path navigate to child paths

Where absolute path every navigation is independent of parent.

**Index Routes**

const router = createBrowserRouter(

  [{

    path:'/',

    element:<RootLayout/>,

    errorElement: <ErrorPage/>,

    children:[

  //{path:'', element: <HomePage/>},

  {index:true, element: <HomePage/>},

  {path: 'events', element: <EventsPage/>},

  {path: 'events/eventDetails/:id', element:<EventDetailPage/>},

  {path:'events/new', element:<NewEventPage/>},

  {path: 'events/:id/edit', element:<EditEventPage/>}],

},

]);

**useLoader from route definition**

useLoader is used to get fetched data from server API at loader property.

const events = useLoaderData();

**useRouteLoaderData(“id”);** - when there is no element property in that path , using id property can get the loader data. Best use for the case when the neighbour loader data is need to use, then making the loader as parent can work. Argument is id property from route

 const data = useRouteLoaderData("event-detail");

**useActionLoader();** is used to get the response from backend API call retuned VALIDATION data

const data = useActionData(); // from backend

**useSubmit()** is used to pass the action and method to the respective delete call

function startDeleteHandler() {

    const procced = window.confirm("Are you sure?");

    if(procced){

    submit(null, {method:"delete"})

    }

  }

When dealing with router and its components , possibility of props- lifting state up working is low, because in routing a single component is mapped to different parent component. Because of that state up is only intended to specific parent component.

**Loader** to fetch the data from server and where **action** is used to post/patch data to server.

**formData()** is used to get the data from **form**

const data = await request.formData();

**useFetcher()** is used tom get the returned response from action function of the component, **useFetcher() to invoke action or loader without loading the page or route.**

const fetcher = useFetcher(); // to get respose of action

    const {data, state} = fetcher;

**redirect() is** used to navigate to a specific path.

return redirect('/events');

**defer() –** is used to improve the performance of the fetch API by using the loaded part of data. **asynchronous data loading and render**