API = Application programming interface.

* SetTimeout is a web API

MOI for capital city

promises callback these are asynchronous programing async/await.

① Date

* What is Javascript ⇒ Javascript is a single threaded, non blocking, asynchronous concurrent language.

(meaning of single threaded that it can only do one thing at a time)

it has a call stack, an event loop and a callback queue and some other APIs

V8 is the javascript runtime which has a call stack and a heap.

Heap is used for memory allocation and the stack holds the execution context.

* Synchronous programing ⇒ synchronous code or progmaing is a step by step code they are executed one after an other. until the function 1 will not be executed, function 2 can not be executed it will wait for the function 1 is compiled.

Ex:

```
console.log(1)
console.log(2)
console.log(3)
```

```
function1 () {
setTimeout (()=>{
  console.log(5) }, 100)
```

output: 1, 2, 3, 5

* Asynchronous programing ⇒ In Javascript Asynchronous is a non-blocking architecture, so the execution of one task is not dependent on another task. the task can run simultaneously. in Javascript asynchronous programing is commonly achived using callback, promises, and more recently async/await.

Ex using callback

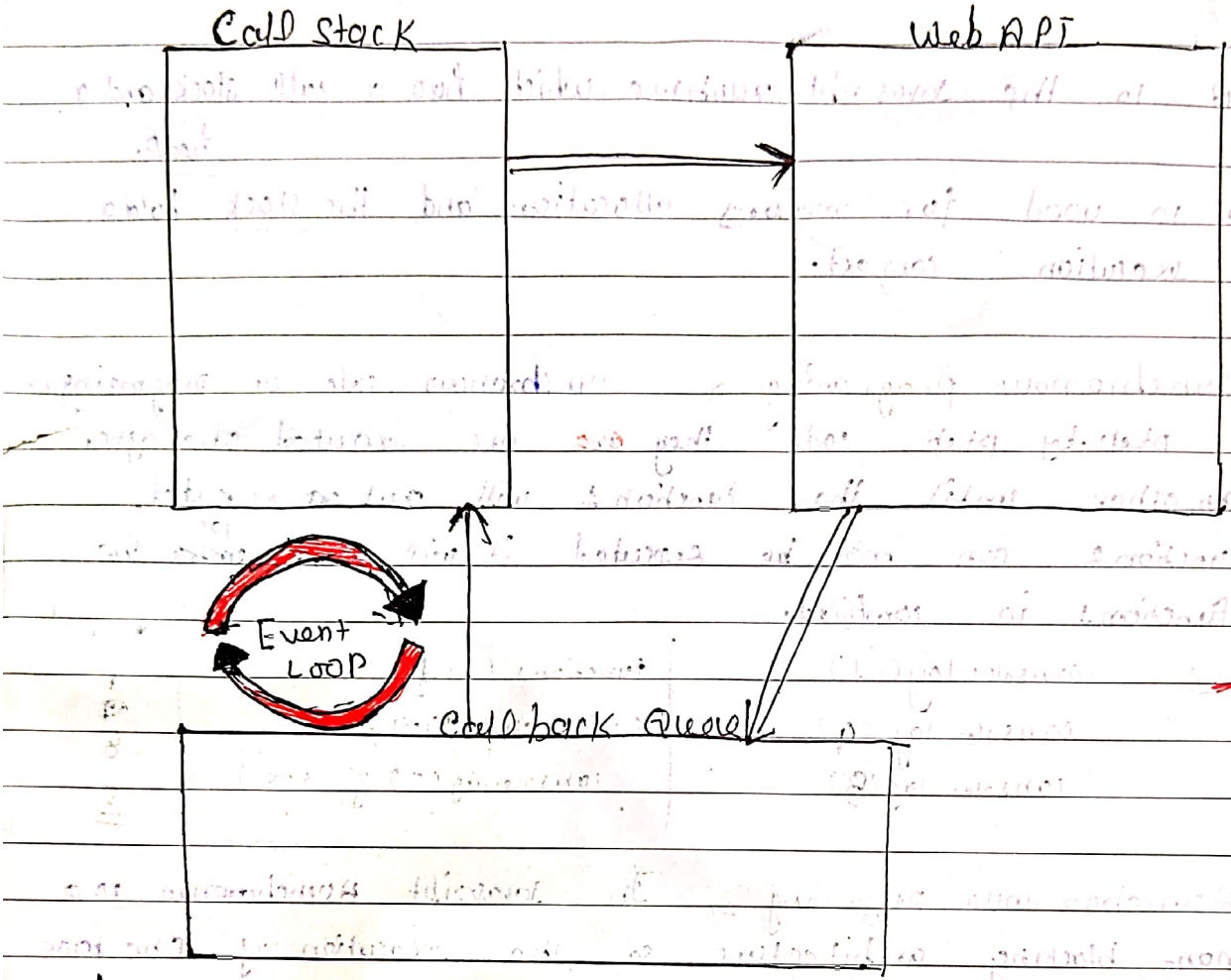
```
console.log(1);
function problem() {
  setTimeout (()=>{
  console.log(2)
  }, 1000)
  console.log(3)
```

output: 1, 3, 2

Date

Event Loop →

~~It javascript event loop is a runtime model~~ that is responsive for executing code, collecting and processing events, and executing queued sub-tasks.



- In javascript event loop is single threaded runtime model, it means that only one piece of code can be executed at a time.

it is responsible for executing code, collecting and processing events, and executing queued sub-tasks.

Event loop allows javascript to appear to be multi-threaded by executing code in non-blocking fashion.

Date

To handle asynchorous tasks, Javascript relies on the event loop. it consist of two main component (i) the call stack and task Queue (also known eventqueue).

① Call stack ⇒ the call stack is data structure that keeps track of the currently executing function. whenever a function is called it is pushed onto the top of the stack. and when a function is completed it is popped off the stack.

② Task Queue (Event Queue) ⇒ Task Queue or event Queue that holds tasks or events that are ready to be proceed. these tasks can include user interaction (click, keyboard events) timers, network responses, and other asynchronous operations

x when a Javascript program start, the event loop starts executing code from call stack. when the callstack is empty, the event loop check the event queue for any event that need to be processed. if there is any Event in the queue, the event loop excutes the corresponding callback functions. and the process repeat again and again until all the program are executed

Orther words =

the event loop is powerful tool that allows Javascript to handle asynchorous events such as network request and user input, without blocking the main thread.
this makes Javascript ideal for building web application that are responsive and interactive.

④

Date

* Call back function In Javascript call back function is a function that is passed as an argument to another function, and it will executed at a later point of time. the purpose of a call back function is to specify what should happen after a certain operation. other word में कुछ वाली definition बोल दो.

Use case the primary purpose of a callback function is to allow asynchronous operation to be executed in a non-blocking manner.

ex:-
```
function userdata( name, lastname, callback ){
  let intro =
    'myname is' + name + 'and lastname is' + lastname + '.';
  callback( intro )
}

userdata('suhail', 'khan' function( intro ) {
    console.log( intro )
  })
```

output my name is suhail khan.

what are

Date

(5)

Date

Async await ⇒ Async / await is a modern syntax introduced in javascript to simplify asynchronous promising and make it appear more synchronous. it is built on top of promises and provide a way to write asynchronous code that looks and behaves like synchronous code.

type case and how works=

ASYNC the async keyward is used to defing an asynchronous function. And asynchronus function return a promise implicitly.

await inside an async function await keyword is used to pause the execution of the function until a promise is resolved or rejected.

Ex

```
async function makeApiCall() {

  try {
    const res = await fetch(url)
    console.log(res.json())   // promise देगा console में
    const data = await res.json()  Resolve कराया यहां पर
    console.log(data)
  } catch(err) {
    console.log(err)
  }
```

makeAPI call()

Most important

Date

countries

* (arr) दिए है country को Select करने में उसकी cities आना चाहिए ।

```
const [ country, setCounty ] = useState( )

return (
    <div>
```

//1st

```
< select / Select
    value = { county }
    onchange = { (e) => {
    console.log ( e.target.value )
    setCounty ( [ e.taget.value) /
    } ) >
{ countries.map(( item, index ) => {
return (
    <option key = { index } value = { index } >
        { item.name }
    </option >
    ) }) }.        </ select >
```

//2nd

```
< select >
{ countries[ country ] &&
    countries[ country ].cities.map(( item, index) => {
return < option value = { index } > { item } </option >

}) }
</ select >

</div>
)
```

INDIA — mumbai, Delhi

PAKISTAN — Karachi, Lahore

API = Application programming interface

Closures

Date

* what are closures with examples? this is a feature in

closure is a feature in javascript that allows inner function to access the outer scope of a function even the outer function has returned

Ex:-

```
function outer() {
    let x = "masai school";
    function inner() {
        console.log(x)
    }
    return inner
}
let ans = outer();
console.log(Ans)
```

Output = masai school

On other page

it is useful for a vaiety of programing tasks. such as -

i) creating a private variables and functions.

ii) implementing callback and event handlers.

* Generate random number

```
const randomNumber = Math.floor(Math.random()*100) + 1;
console.log(randomNumber)
```

callback queue and

time which has a call

memory allocation

JavaScript is an object-oriented language.

H.O.F

Date

* Higher order function

A higher order function is a function that takes one or more function as an argument and return a function as its result.

In other words we can say that H.O.f either accepts a function as an argument or return a functi as its result or both the case.

ex:-

```
function H.O.f( x, operation ) {
    return operation( x );
}

function double( x ) {
    return x * 2
}

function plus( x ) {
    return x + 2
}

console.log( H.O.f( 4, double ) )
console.log( H.O.f( 4, plus ) )
```

H.O.f	normal
	i) it will return output as v

let, const, var

Date

(2)

i) var → variable declared with var are function-scoped. this means they are accessible with in the function in which they are declared. if variable declared outside of function it becomes global variable.

Note: variable declared with var can be redeclared and updated.

ii) let = variable declared with let has block-scoped. it means it can be accessible inside the block in which they are declared. (ex: conditional statement or loop) variable declared with let can be reassigned, but cannot be redeclared with in the same block.

const = it also has block-scoped. the value can not be ~~change or~~ reassigned or redecled.

ex

```
function find( ) {
  var x = 1;
  let y = 2;
  const z = 3;
  if ( true ) {
    var x = 10
    let y = 11
    const z = 12
    console.log( x, y, z )   output = 10, 11, 12
  }

  console.log ( x, y, z )    output = 10, 2, 3
}

find ( )
```

MAP | ForEach | Reduce

Date

* map :- it is used iterate over an array and transform each element into a new value it returns a new array with the transform elements. without modifying the original array.

ex :- const arr = [1, 2, 3]
const newArray = arr.map((ele) => ele*2)
console.log(newArray)

output = [2, 4, 6]

* forEach ⇒ it is used to iterate over an array and execute a provided function for each element of the array. it does not return a new array but it can modify the original array.

* const arr = [1, 2, 3]
arr.forEach((ele) => console.log(ele*3))
output = 3, 6, 9

* Reduce - Reduce is used to accumulate the elements of an array into a single value.
it takes a callback function that takes two arguments i.e accumulat and current element the call back function return the updated acc value which is passed to the next iteration. the final accumulated value is the return value of reduce.

const arr = [1, 2, 3]
const sum = arr.reduce((accumulat, element) => accumulat + element, 0)
console.log(sum) ⇒ output = 6

Hooks in React hooks are generally functions that allow you to use React features like state and life cycle method inside functional components.

there are several inbuilt hooks in react.
i) useState, useEffect, useContext, and useMemo.

Why we use ⇒ Hooks provide a way to write more concise and reusable code in React functional component, and we can simplify the process of managing state and side effects.

useState →

useEffect it allows you to perform side effects in your components. these side effects are fetching data, directly updating the dom and timers, etc.

```
function Timer() {
const [count, setCount] = useState(0)
useEffect(() => {
setTimeout(() => (count+1))
}, [])
});
return <h1> I rendered {count} times </h1>
```

NOTE = Any null value of operator will always return type of object.

Date

* useRef → it is hook to allows you to creat a mutable refrence that persists across renders. useRef return a ref object, which has a current property that can be set & read. We can use useRef for manage fo and access Dom elements.

```
function myComponent ( ) {
    const inputRef = useRef(null)

    function handleClick ( ) {
      input.current.focus()
    }

    return (
        <div>
      <input type="text" ref={inputRef} />
      <button onClick={handleClick}> focus</
      </div>
    )
    }
```

useState | useEffect

console.log(NaN === NaN) = false

typeof(NaN) = Number

(4)

Date

Use State useState is a hook that allows you to add state to functional components. it takes an initial state value and returns an array with two value (1) the current state value and a function that can be used to the state. when the state is updated, React will re-render the component with new value.

Ex:-

```
const Counter = () => {
  const [count, setCount] = useState(0)

  const handleUpdate = () => {
    setCount(count+1)
  }

  return (
    <div>
      <p> count is {count} </p>
      <button onClick={handleUpdate}> count++
      </button>
    </div>
  );
};
```

Rough

```
y = 3*2
  6.

i = 3
i<=3
3<=3
0 = 0+1*2
  0

0 = 0+2*2
0 = 0+4
0 = 4
0 = 0+3*3
y = 6
```

Ex:- var x = 100
function test(){
var x = i = 10
}
test()
console.log(x, i)

(2)

HOISTING

Date

Hoisting :- It is a javascript behaviours where variable and functions decleration are moved to the top of their scope before code execution. we can say that function and variables can be used before they are decleared.

Ex:-

```
var x
console.log(x)
x = 1;
output = (undefined)
```

```
foo()
function foo(){
console.log("Hello")
}
= (output- Hello)
```

* Rest operator ⇒ the rest operator is an improved way to handle functions parameter the rest parameter syntax allows us to represent an indifinite number of argument as an array

* with the help of rest piarameter a function can be called with any number of argument no matter how it was defined.

Ex:-

```
function fun( a, b, ... c ){
console.log( ${a} ${b} )  output: Aman, mrupul
console.log( c )  [ Rohit, Ravi, Raj ]
}

fun( Aman, mrupul, Rohit, Ravi, Raj )
```

Date

Spread operator ⇒ In JavaScript spread operator allows us to expand an array into individual array element. to use the spread operator three dots (...) should be proceed by the array name.

ex:- let arr1 = [1, 2, 3, 4]
let arr2 = [5, 6, 7, 8]
let arr3 = [...arr1, ...arr2]
console.log(arr3)
output = [1, 2, 3, 4, 5, 6, 7, 8]

Date: Page:

callback function ⇒

when a function passed an argument to an other function this is called callback function. and it is executed inside the function to which it is passed.

purpose ⇒ primary purpose of a callback function is to allow asynchronous operation to to be executed in a non-blocking

output = Data received data

Which kind of state (i)

(ii)
(iii)
(iv)
(v)

Date

* what is Redux?

Redux is a state management liberary for javascript application.

We can use redux together with React.js and also used with other view libraries such as Angular or Vue.js.

* Redux helps manage the state of an application in a central location called a store which is plan javascript object.

* what is JSX ⇒

JSX stands for Javascript XML. JSX allows us to write HTML in React. JSX makes it easier to write and add HTML in React.

ex:-

```
const name = "manaj";
const element = <h1> Hello, {name} </h1>;
const name = "manaj"

return (
    <div>
        <h1> Hello! {name} </h1>
    </div>
)
```

* core principal of redux =

Redux is predictible state management library for javascript. the core principals of redux are follow as =

① Single-source of truth ⇒ the state of an entire redux application is stred in a single object tree called the store. it makes easy to manage and update the state as well as to debug the application.

② State is read only ⇒ the state in redux application is read only, which means that it can not be modified directly. instead changes to the state are made by dispatching "actions" that describe what happend in the application.

(3) changes are made with pure function ⇒ in Redux the state is updated by pure function called "reducer". these reducer takes the previous state and an action as input. and return a new state object.

* pure function

A pure function is a function that always return the same output given the same input, and has no side effect. this a pure function does not modify anything outside of its own scope, and does not rely on any external state or variables.

Ex :-

Reducer is pure function =

Date

Classes

In javascript classes are syntactical sugar for creating object and constructor function. classes allows you to define a blue print for creating objects with a set of properties and methods.

Ex

```
class Animal {
constructor ( name, sound ) {
    this.name = name;
    this.sound = sound;
} speak ( ) {
    console.log( ' ${ this.name } says ${ this.sound } ' );
}
}
const dog = new Animal( 'bruck', 'roor' )
dog.speak ( )
```

* What are redux DevTool's most essential features?

① Time - Travelling debugger

② Action log

③ State inspector

④ Time-travelling - console

⑤ multiple store support

Take number as input from user and generate it to random number

Date

```
import React from 'react'

const GenratorNumber = ( ) => {
  const [ number, setnumber ] = useState("")
  const [ random , setrandom ] = useState ( null )

  const handleSubmit = ( e ) => {
    e.preventDefault ( )

    const randomnumber = Math.floor( Math.random()
    setrandom ( randomnumber )          * 100 ) + 1;
      setnumber ("")
  };

  return (
      <div>
      <form onSubmit = { handleSubmit } >

      <input type = "number"  value = { number }
      onChange = { (e) => setnumber(e.target.value) } />

      <input type = "submit" value = "submit" />
      <hr />
      <p> random number is : { number } </p> }

      </div>

export default GenratorNumber
```

Date

* Constructor function

Constructor function is special type of function that is used to create new obj with a set of properties and methods. When a constructor function called with new keyword, it creates a new instance of the ~~project~~ obj and assign the properties and method to that instance.

Thub rule ⇒ start with capital or uppercase.

Ex:

```
function person ( name, age ) {
    this.name = name;
    this.age = age;
    this.sayHello = function (-) {
    console.log(' Name is ${this.name} and age ${this.age}.')
    }
}

const John = new person ('Subail', 26 )
John.sayHello()
```

output = Name is Subail and age is 26

* Factory function

⇒ It is a function that creates and return new object without using the new keyword. defined with a lowercase.

Ex:

```
function createUser (name, age) {
    return {
        name,
        age,
        sayHello () {
        console.log(' name is ${this.name} age ${this.age}.')
        }
    }
}
const John = createUser ('Subail', 20)
John.sayHello();
```

Date

React React is a popular javascript library for building user interface. It is developed by facebook. and released in 2013.
React follow a component-based structure where the the UI is broken down into smaller reusable components, to making it easier to manage and maintain.

node.js = Basically node.js is a runtime environment developed by Ryan Dahl in 2009. It allows developers to run javascript on the server-side as opposed to just in the browser. node.js comes with a built in package manager called npm. which allows developers to easily install and manage third party pkg and modules.

Used by = Netflix, LinkedIn, and Walmart.

Async/await basically async/await simplifies the process even further. An async function return a promise and with in the promise function, you can use the await keyword to wait for the result of an asynchronous operation. When using await, the function pauses execution until the asynchronous operation is complete and then it resumes execution.

= Async/await allows you to write code that looks more synchronous and easier to read.

* we use promises to avoid asynchronous nature of JavaScript code

promises

Date

* promises ⇒ promises are a features in JavaScript that provide a way to handle asynchronous operations.

A Promise represents a value that may not be available yet, but will be resolved some in the future.

A promise can be in one of three states

(1) it has 3 states ✓

Pending ⇒ initial state, not rejected, not fulfilled wait for asyncronus operation to complete

(2) Fulfilled ⇒ it means the asyncronus operation has completed successfully with a resolved value we can use it by .then() method.

(3) Rejected ⇒ it means the asyncronus operation has failed. this means that the promise has reason for Rejection.

we can access it by using .catch() method

```
const my promise = new promise ( ( resolve, rejected ) => {
    setTimeout( () => {
        const number = math.random() ;
        if( number < 0.5 ) {
            resolve ( number )
        } else {
            rejected ( new error (" value is greater "
        }
    }, 1000 ) } ) ;
my promise.then( ( result ) => {
    console.log (" fulfilled : ${ result } ' )
} ). catch ( ( error ) => {
    console.log ( " rejected : ${ error } )
```

Most important example

use this all time

Date

```
function upperdata( ) {
return new Promise((resolve, rejected) => {
```

new keyword used to create new instance of promise is a constructor function.

~~setTimeout(( ) => {~~

~~const num = Math. random( )~~

~~if~~

```
setTimeout( ( ) => {
const number = Math.floor( Math. random ( ) * 10)

{ number < 10 ? resolve(number) : reject ( new Error(" number is grater")

}, 1000);

}) };

upperdata( ). then (( result) => {
console.log(' fulfilled : ${ result}');

}) . catch (( error ) => {
console. log( ' rejected : ${ error}');

})
```

Most important example
use this all time

Date

function updateData( ) {
return new Promise((resolve, rejected) => {
new keyword used to create new instance of promise is a constructor function.

~~setTimeout( ) => {~~
~~const num = Math. random( )~~
~~if~~

setTimeout(( ) => {
const number = Math. floor(Math. random ( ) * 10)

{ number < 10 ? resolve(number) : reject (new Error(" number is grater")}

}, 1000) ;

}) } ;

updateData( ). then((result) => {
console.log(' fulfilled : ${ result}');

}) . catch((error) => {
console. log(' rejected : ${ error}') ;

})

Date

promises example

~~function promise(res, rej)~~

```
function promise(time) {
    return new promise((res, rej) => {

        if(typeof time !== 'number') {
        rej("argument is not a number")
        return
        }

        setTimeout(() => {
            res() or res({status: 200})
        }, time)
    })
}

promise(2000)
.then((res) => console.log("printed after 2 sec"))
.catch((err) => console.log(err))
```

2nd way

we can also use with async await.

```
async function test() {
    try {
        console.log("waiting for result")
        let res = await promise(2000)
        console.log(res)
        if(res.status === 200) {
            return true
        }
    }
    catch(err) {
        console.log(err)
    }
}
test().then((res) => {console.log(res)})
```

Date

```
function sleep ( time ) {
  return new promise ((res, rej) => {
  if (typeof time ! == 'number' ) {
    rej (" rejected")
  return
  }

    setTimeout ( ( ) => {
    res ( {status: 200} )
    }, time)

  })
  }
  setTimeout ( ( ) => {console.log ('first' ) }, 0 );

  promise . then ( res => console.log ( 'Promise' ) ) . catch ( )
```

out put =

promise

first

Important

* undefined == null ⇒ true Not check type on check value because both are false value.

* undefined === null ⇒ false Not same as the null.

* [ ] == [ ] or [ ] === [ ] ⇒ false.
  (i) two arrays are distinct object in memory.
  (ii) two arrays are separate objects with difference reference.

Date

* Three basic object attributes in javascript are =

(i) class, prototype, object's extensible flag.

(ii) class, native object, and interfaces and object extensible flag.

* console.log ("123 Hello") ;
  (print)

* webpack is a module bundler.

Q4

Date

Memoization = memoization is a process that allows us to cache the values of expensive function calls so that the next time function is called with the same arguments, the cached value is returned rather than having to re-compute the function.

Example

Date

Example

How will you exicute javascript after pageload

① window.onload() ② document.onload()

Dom content loaded

~~Use index~~

create index() command in the shell to create a single index

Use memo

it is a react hook that we can use to wrap function within a component we can use this to ensure that the values within that function are re-computed only when one of its dependencies change.

Date

React.memo ⇒ it is a higher order function components in React that is used to optimize the performance of functional components by reducing unnecessary re-renders.

when a functional component is wrapped with the React.memo, React will optimize the result of the component rendering based on its props. if the props haven't change since last rendering, react will reuse the cached result instead of re-rendering the component.

Note:- it can improve the performance of our application by reducing the number of unnecessary re-renders.

example:- import React, {useState, useMemo} from 'react'

function App() {
const [count, setCount] = useState(0)
const [item, setItem] = useState(5)

const multiCount = useMemo(function multiCount() {
console.log("multiCount rendering")
return count * 2
}, [count])

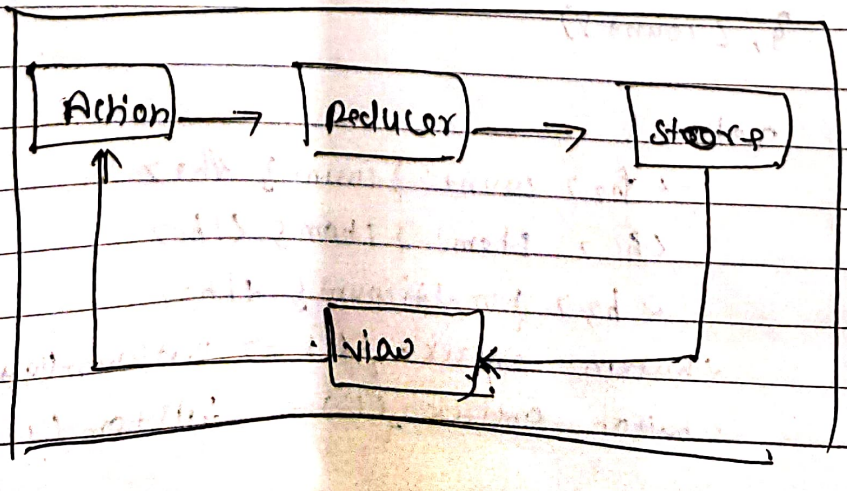
return (
<h2> count: {count} </h2>
<h1> Item: {item} </h1>
<h3> {multiCount} </h3>
<button onClick={() => setCount(count+1)} </button>
<button onClick={() => setItem(item+5)}
</button>

Date

useCallback useCallback is hook in React that is used to memoize a function. Similar to how useMemo memoizes a value, when the function is memoized with useCallback the same function instance is returned on every render as long as its dependencies have not changed.

Ex→

```
import { useState, useCallback } from react;

function myCom() {
    const [count, setCount] = useState(0)

    const handleClick = useCallback(() => {
        setCount(count + 1)
    }, [count]);

    return (
        <div> {
            <p>count: {count}</p>
            <button onClick={handleClick}>Increment</button>
        </div>
    )
}
```



Date

Redux Top Level Directories

(1) Components ⇒ Used for dumb components unware of redux (do not aware of redux store)

(2) Containers ⇒ Used for smart components connected to redux.

(3) Actions ⇒ Used for all actions creators, where file names correspond to part of the app.

(4) Reducers ⇒ Used for all reducers, where file name correspond to state key

(5) Store ⇒ Used for store initialization.

(6) API / utils / selectors

Actions ⇒ Action are a plain javascript object that contains information. Action are only javascript source for the store. Action have a type field that tells what kind of action to perform and all other fields contain information or data.

Example

```
                                    payload
export const getData = () => {

    return { type: GET_BUILDS, }
}
                                    payload

export const getRequest = ( ) => {

    return { type: GET_REQUEST }
}
```

Pure function ⇒

Date

Redyer

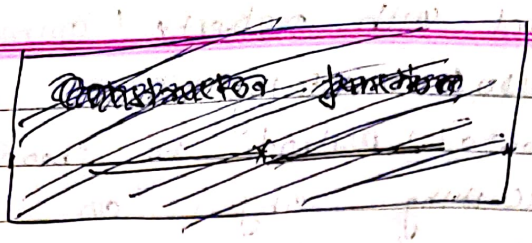
A reducer is a pure function that takes the current or initial state of the applicat and an action as argument, and return a new state that reflects the changes specified by the action.

Ex:-

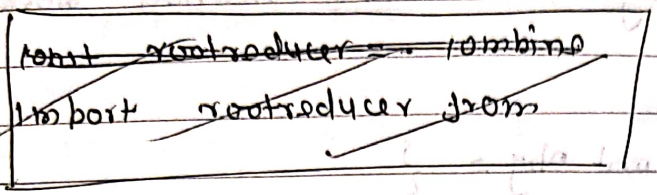
```
const initialState = {
    isLoading : false,
    isError : false,
    data : []
}

export const reducer = ( state = initialState, {type, payload} ) => {
    switch( type ) {
        case GET_REQUEST :
        return { ...state, isLoading : true, isError : false }

        case GET_SUCCESS :
        return { ...state, isLoading : false, data : payload }

        default :
        return state;
    }
```

Store :- Store is an object that holds the application state and provides a way to dispatch actions to update the state. In redux there is only a single store for the entire application.

Date



* example



```
const rootReducer = combineReducer({
    authReducer,
    todoReducer
});

export const store =
```

```
() => { const store = legacy_createStore(AppReducer,
                        applyMiddleware(thunk));

export { store }
```

① uuid ⇒ (universal unique identifiers) this library generates RFC-compliant UUIDs as strings, which can be used as unique keys in React components.

```
import { v4 as uuidv4 } from 'uuid';

export const MyComponent = () => {
    const uniqID = uuidv4()
    return <div key={uniqID}> {uniqID} </div>;
};
```

② shortid ⇒ this library generates short, unique, and non-sequential ID strings.

```
import shortId from 'shortid';

export const MyComponent = () =>
    const uniqId = shortid.generate();
    return <div key={uniqId}> {uniqId} </div>;
```

Date

Create an object with property (animal: dog) and method say Hello (console.log(" This is dog")) use the object.create() method to create a new o using the existing object as the prototype the newly created object. add property to the new object (legs: 4) and method work (console. Barkin

example

① let Animal obj = {
Animal: 'dog',
SayHello: function () {
console.log('this is dog')
}
}

add new key to Animal obj - here
Animal obj. color = " red"

②
Create new obj using animal obj

const newobj = object.create(animal obj)
newobj.legs = 4;
newobj. work = function () {
console.log('Barking')
}

console.log(Animal obj. Animal): output = dog

console.log (newobj.legs) = 4

~~console.log(~~
newobj. work() = Barking
animal obj. sayHello() = this is dog

Date

Event loop ⇒ An events loop in Javascript is a single thread that handles all the asynchronous callbacks, such as user input, network request, and other events.

the execution flow of javascript is based on event loop.

there is an endless loop, where the javascript engine waits for tasks, execute them and then waiting for more tasks.

function currying

currying in javascript transform a function with multiple arguments into a nested series of function, each function taking a single argument.

(i) it helps you avoid passing the same variable multiple times.

(ii) it helps you to create a higher order function.

Ex:-

```
function add(a){
return function(b){
    return a+b
}
}
console.log(add(2)(3));
```

API =

Date

WEB API (API)

Decorators in React

Decorators help

the aim of decorator to add functionality to an object dynamically, without changing its original implemention.

* it help you take an existing class component, or function of a class component and modify it.

---

context API it is a features in react that allows you manage global state and share data between component without having to pass props through all the intermediate components.

with the context API, you can create a context object that holds a value, and then provide that value to all the component that need it.

this can be a more efficient and organized way manage state in larger applications.

Ex (1) create context

```
import React, { createContext, useState } from "react"
const ThemeContext = createContext()
export const ThemeProvider = ({ children }) => {
const [ theme, setTheme ] = useState("light")
```

Date

~~Throttling~~

~~Throttling~~ ⟶ ✗ ⟶

const toggleTheme = ( ) => {

setTheme(theme === "light" ? "dark" : "light");

}

② (provide)

return (

<ThemeContext.Provider value={{theme, toggleTheme}}>

{children}

</ThemeContext.Provider>

)

}

③ use

import ThemeContext from "./Theme"

import useContext from "react";

const App = ( ) => {

const { theme, toggleTheme } = useContext(ThemeContext);

return (

<div style={{ bgC: theme === "dark" ? "black" : "white",

}}>

<h1> theme : {theme} </h1>

<button onClick={toggleTheme}> toggle </button>

export default App

interview से पहले chat gpt पर details में देखना है

Date

* Debouncing ⇒ Debouncing is a technique used in web Development to improve the performance of event handlers that are triggered frequently such as scroll, resize, or input events.

* the purpose of debouncing is to prevent a function from being called multiple times in quick succession, which can be effect our application performance and unexpected behaviours.

Note ÷ Note provided by javascript throttling and debo

ex:-

```
let count = 0
function getData ( ) {
  console.log(" fetching data " + count++)
}

function Debouncing( call, delay ) {
  let timer
  return function(... args) {
    if( timer)
      clearTimeout( timer)
    timer = setTimeout ( () => {
      call()
    }, delay);
  }
}

const main = Debouncing (getData, 1000)
```

Html

~~<input type "text" id "search" onchange{~~

```
<input type = "text" id = "search" oninput={ main()} />
```

Date

* Throttling ⇒ Throttling is a technique in which, no matter how many times the user click or fires the event, the attached function will be executed only once in a given time interval.

(ii)

Used

Debouncing and throttling both are used to enhance the website performance by limiting the number of times the events are triggered.

Ex:

```
const newFun = myThrottle(() => {
    document.getElementById("idbtn").disabled = false;
    console.log("user clicked..!!")
}, 5000)
```

```
const myThrottle = (fn, d) => {
return function (...args) {
    document.getElementById("myId").disabled = true;
    setTimeout(() => {
        fn();
    }, d);
}
}
```

Html

```
<button id="id" onclick={newFun()} />
```

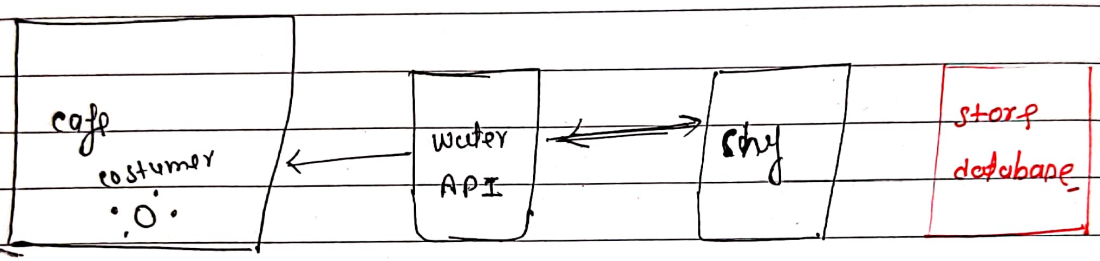
Date _____

API → Application programming interface ⇒ An API is a software intermediary that allows two or more application or system to talk each other.

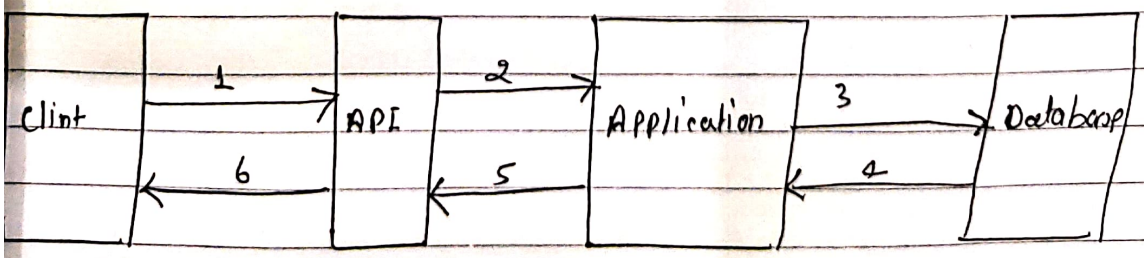
Types of API =

(a) private ⇒ used within an organization
(b) partner ⇒ used with business partner
(c) public ⇒ used by third prty developer.

Example



Web API ⇒



Explicitly and implicitly

Date

In javascript, there are two ways in which a function can return a value.

Explicit Return →

An explicit return occurs when a function uses the return statement to explicitly specify the value to be returned. When the return statement is encountered the value after the 'return' keyword is returned as the result of the function.

ex:
```
function add(a, b) {
  return a+b
}
let result = add(2,4)
console.log(result) ⇒ 6
```

Implicit return ⇒

Implicit return occurs when a function doesn't not use the 'return' explicitily, but the result of the function is still returned.
This happen when the function body consist of a single expression, and that expression is implicity returned as the result.

```
const multiply = (a, b) ⇒ a*b

const rslt = multiply(3,4)
console.log(result) = 12
```

Note - it is commonly used with the arrow function and syntax to write shorter and more expressive code.

⑲ this, call, apply, bind in JavaScript

apply, call, bind method are used to set the this context and optionally pass argument to the fun

Rule-01 By default this keyword point at the global object. which in the browser is the wi object.

ex:-

```
function sayHello() {
    console.log("byee", this)
}
```

output byee window ऑब्जेक्ट

```
sayHello();
```

Rule-2 the implicit binding of this =

when a method is called as a property of an then this implicity refers to the parent ob if a function attached to an object is called "object.functionname() syntax", the points to the object to which the function is attached.

ex:-

```
var John = {
    name : "John Doe",
    age : 30
    sayname : function () {
        console.log("my name is", this.name)
    },

    sayHello : function () {
        console.log("Hello! from", this.name)
    },
}

John.sayname() : => my name is John Do
John.sayHello() : => Hello! from John Doe
```

Rule # 3 call / apply

Date _____

when a function is called using the call or apply methods, then 'this' refers to the value passed as the first argument to the call and apply.

Ex:
```
function sayHello ( ) {
    console.log ("Hello !", this)
}

var John = {
    name: "Love",
    age: 30
};

var James = {
    name: "Lovly",
    age: 30
}

sayHello.call ( John ) => Hello! {name: "Love", age: 30}
sayHello.apply ( James ) => Hello! {name: "Lovly", age: 30}
```

NOT

* **bind** bind creates a new function hard bound to the object that we have specified.

Ex:
```
function sayHello ( ) {
    console.log (" Hello", this, this.name);
}

var John = {
    name: " suhail"
    age: 30
}

var newfunction = sayHello.bind ( John )

newfunction ( ) : => Output-
Hello ! { name: "suhail", age: 30} John.
```

(8) which of the following are not ...
(i) variable (ii) function (iii) object

Date

Closure

* A closure is a mechanism in javascript in which the inner function has access to the outer function variable.

Ex:

```
function outer ( ) {
    var a = 10

    return function inner ( b ) {
        return a + b
    }
}

var x = outer ( ),
console.log ( x ( 10 ) )
```

```
( function ( a ) {
return ( function ( ) {
    console.log (a)
    a = 6 ;
} ) ( )
} ) ( 21 )
```

output = 21
output = 10

*

Currying =

note = in currying we can pervating some function p

```
function outer (a) {
return function inner ( b ) {
    return a + b
}
}

console.log ( outer (10) (20) )
```

output (30)

Note = Base Object does not have a

(3)

Date

* when we need to remember previous value

Important

```
function calculator ( initial value = 0) {
  var value = initialvalue
  function add (val) {
  value = value + val
  return value
  }
  return { add }
}

var cal = calculator( )
console.log( cal. add(10)) => output to
console.log ( cal. add(10)) output 20
```

* IIFI

Exx

```
( function ( ) {
setTimeout ( ) => console.log (1), 2000 )
  console.log (2);
setTimeout ( ) => console.log (3), 0 );
console.log (4)
}) ( );
```