FRONTEND

1) Javascript
2) Javascript Type Conversions
3) Template Literals (String Interpolations)
4) Arrow Functions
5) We can change key:value of a constant object/array
6) Entry Control Loop – Exit Control Loop
7) Mutable v/s Immutable
8) Spread Operator
9) Checking Mutability with Object
10) Comparing Mutable Datatypes
11) Destructuring in JavaScript
12) Adding and Deleting key from Object
13) Why typeof array is object
14) Browser Object Model / BOM / Window Object
15) Document Object Model / DOM / Document Object
16) HTML Collection v/s NodeList
17) Difference in BOM and DOM
18) Using 'this' keyword in EventListener
19) Changing CSS with JavaScript of the element
20) getComputedStyle()
21) EventObject
22) Date Object
23) Generate Unique IDs using JS's crypto.randomUUID()
24) Delete Parent node/element
25) To get Sibling node
26) Toggle class
27) Get height of window/viewport
28) Get position of any html element according to screen
29) Higher Order Functions
30) Difference between method and function
31) Anonymous Function
32) JavaScript Program to remove duplicate values from array (using array's includes() method
33) Asynchronous nature of JavaScript
34) Promise
35) JSON (JavaScript Object Notation)
36) Fetch() method || Asynchronous Data Fetcher : Converting JSON into JS Object
37) Event Loop
38) Using 3rd Part Libraires/Utilities/Frameworks

39) Difference in Framework and Library
40) BootStrap
41) Bootstrap v/s Tailwind
42) Tailwind darkMode using tailwind variables
43) NodeJS
44) LTS (Long Term Support)
45) Built-in Modules
46) Include Modules
47) Create your own Module
48) Difference between Module and Package
49) Module v/s Package v/s Library
50) Running JavaScript using node in cmd/terminal
51) NPM (Node Package Manager)
52) Setup/Create a Project using NodeJS
53) CommonJS v/s ECMAscript Modules [require v/s import/export]
54) Type of Packages in NPM
55) Types of Dependencies
56) Package.json v/s Package-lock.json
57) React
58) Real DOM v/s Virtual DOM/React DOM
59) Optimization: Update Local State Instead of Re-fetching After a Successful Update API Call
60) Hydration in React
61) Creating a React App
62) Importing Images in React
63) Exporting Multiple modules from one file
64) Blank Fragment
65) Using JS code inside JSX code
66) Using inline CSS in React
67) Routing in React
68) Dynamic Routing
69) useParams of react-router-dom
70) Nested Routing
71) Nested Routing in NextJS (using AppRouting)
72) AppRouting v/s PageRouting in NextJS
73) Passing Data between React Components (props, contextapi, redux etc)
74) PROPS (properties)
75) Destructuring Props
76) ContextAPI
77) Redux
78) State Management (Dynamic Changes)
79) State Management

80) State Immutability
81) Issues with Direct Mutation
82) Brief on ‘const’ states
83) Babel Compiler & Transpiling
84) Hooks
85) Create React App using Vite
86) Component Life Cycle
87) Clean up Function / Unmounting
88) Data Transfer from Child Components to Parent Component using props
89) Axios (node package to fetch APIs)
90) Difference between Axios and fetch()
91) useReducer
92) useMemo
93) useCallback
94) useActionState
95) useContext
96) React APIs
    i) memo
    ii) cache
97) useDeferredValue
98) useLayoutEffect
99) useRef
100) useTransition
101) key attribute to trigger Mount and Unmount of React Component
102) <Profiler>
103) <Suspense>

JavaScript

JavaScript only used to work on the browser(on client side). We needed to learn another language to handle the server side database tasks (ex. php) But node.js changed it. Now because of node.js, we can use JavaScript on server side too.
Browsers used V8 Engine to run JavaScript code. Node.js adopted that V8 Engine on server side. Node.js is runtime software which we installed on server side.
Now we can use JavaScript on both client side and server side.

JS implementation:

1. Inline (should avoid. bad practice)
2. Internal
3. External

1) Inline JS : We use event as an attribute of the tag. Some inline attributes events are : onclick, onfocus, ondblclick, onload & onchange.

```
<head>
  <script>
    document.write("Welcome to Javascript World");
  </script>-
</head>
<body onload="alert('Hello')">
  <button onclick="alert('hello world')"></button>
</body>
```

JavaScript Type Conversions

To convert JavaScript variables to another datatype

- By the use of JavaScript functions
- Automatically by JavaScript itself

Some Type Conversion functions are :

1. Number()
2. parseInt()
3. parseFloat()

4. Boolean()
5. String()

Brief : https://www.w3schools.com/js/js_type_conversion.asp

Tip : 'clg' for shortcut for 'console.log()' in VsCode

```
1. var a = Number(promt('Enter a Number'));
```

```
1. var a = true;              Boolean datatypes (true = 1, false = 0)
2. var b = true;              [Plus operator will try to add every possible datatype that can be added.
4. console.log(a+b);           Otherwise it will concat them ]
5.
6. output : 2
```

```
1. var a = "5";               [JavaScript try to convert datatype to a reasonable datatype so it can perform operations,
2. var b = true;               If the datatype of variable are not conversable and can't be added together, then it just
3. console.log(a+b);           concatenate them ]
4.
5. output : 5true
```

```
1. var a = 'hello';
2. var b = 2;
3. console.log(a-b);
4.
5. output : NaN [Not a Number]
```

```
1. var a = 5;
2. var b = 10;
3. var c = a == b;
4. console.log(c);
6.
7. output : false
```

```
1. var a = "5";
2. var b = 5;
3. var c = a == b;
4. var d = a === b; // will check datatype too
5. console.log(c);
6. console.log(d);
7.
8. output: true
9.         false
```

Accessing JavaScript variables in HTML (without using DOM)
https://github.com/69JINX/FrontEnd/blob/main/Javascript/Marksheet-Table%20-%20Inserting%20script%20between%20HTML%20tags.html

---

Creating HTML tags by JavaScript(without using DOM) :

```
1. document.write("<h1>Hello</h1>");
2. document.write("Hello" + "</br>");
```

```
1. let str = "Hello";
2. document.write(str[2]);  // this will print 'e'
```

Template Literals (String Interpolations)

Used to print text and variables together without using the '+' operator. ${} is used to print variables. TL are used without BackTicks (not back slashes);
Without using TL :

```
1. document.write(“Hello” + name + “GM”);
```

With TL :

```
1. document.write(`Hello ${name} GM`);
```

Code written after return keyword in a function never get executed.

Arrow Functions

Arrow Functions were introduced in ES6. Arrow Functions allow us to write shorter function syntax.

```
1. const fun1 = (a,b) => a*b;
```

It gets shorter if the function has only one line of statement and the statement returns a value, you can remove the brackets and the return keyword.

Note : Arrow functions are variables, just like regular variables, it shouldn’t be called before defining it. Not like regular function that can be define anywhere in the code and be called anywhere.

Arrow Function :

```
1. fun1();
2. let fun1 = () => {
3.   document.write(“Hello”);
4. }
5.
6. output :
7. Reference Error : Cannot access ‘fun1’
8. before initialization
```

Normal Function :

```
1. fun1();
2. function fun1(){
3.   document.write(“Hello”);
4. }
5.
6. output : Hello
```

```
1. let temp1  = {}; //object
2. let temp2 = []; // array
```

We can change key:value of a constant object/array

In a constant object(const), even though it is const, we can change its key/value pairs and add/delete key/value Because we are not changing the whole object when we are altering the properties/methods of an object, we are not re-assigning or re-declaring the contact, it’s already declared and assigned. We’re just adding to the list of elements or properties to which the constant points

So this works fine:

```
1. const obj = {};
2. obj.foo = ‘bar’;
3. console.log(obj); // {foo:’bar’}
4. obj.foo = ‘bar1’;
5. console.log(obj); // {foo:’bar1’}
```

and this works fine too:

```
1. const arr = [];
2. arrr.push(‘foo’);
3. console.log(arr); // [‘foo’]
4. arr.unshift(‘foo2’);
5. console.log(arr); // [‘foo2’,’foo’]
6. arr.pop();
7. console.log(arr); // [‘foo2’]
```

but these will through error:

```
1. const obj = {};
2. obj = {foo:’bar’}; // error:reassigning
3.
4. const arr = [];
5. arr = [‘bar’];   // error:reassigning
```

So the properties of elements of a constant object/array can be changed but whole object/array can’t

---

Entry Control Loop

for and while are entry control loop are entry control loop cause they check the condition before entering inside the loop.

Exit Control Loop

do-while is a exit control loop cause it run the loop then check the condition means even if condition is false, it will at least run one time.

---

3 types of function :

1. Without argument (simple function)
2. With argument
3. Functions the 'return' something

Mutable v/s Immutable

There are 2 types of datatype in JavaScript :

1. Mutable / Non-Primitive / Reference Type : Something that can not be changed or added to.
   Eg : Object, Arrays, Functions
2. Immutable / Primitive : Something that can be changed or a new property be added to
   Eg : Number, Boolean, String, Null, Undefined, Symbol

Immutable, once created can't change their value but can be reassigned to a new one.

Immutable:

```
1. let a = 'john';
2. let b = a;
3. b = 'steve';
4. console.log(a);
5. console.log(b);
6.
7. output : john
8.          steve
```

In memory:

both variables point at different memory location of their own

```
a -> 0xf094c649 [john]
b -> 0x7de53064 [steve]
```

Mutable:

```
1. let a = [1,2,3,4]
2. let b = a;
3. b[1] = 100;
4. console.log('a = ',a);
5. console.log('b = ',b);
6.
7. output :
8. a = [1, 100, 3, 4]
9. b = [1, 100, 3, 4]
```

In memory:

both variables point at same memory location because we have assigned memory address of b to point at memory address of a by doing 'let b = a' (which is the same address)

```
a
   \
    > 0x6a99aa2f [1, 100, 3, 4]
   /
b
```

By default, all Reference types (array, object, function) are mutable in JavaScript like if we normally assign a Reference datatypes of another data type, they both will point at same value at single memory address unlike C/CPP language where we have to use a pointer to do that.

```
1. int a = 10;
2. int*b = &a; // pointer in C language
```

If you don't want address of a Reference type to be copied to another variable when we assign them but only value to be copied, you can use the spread operator.

```
1. let str = 'Hello';
2. str[1] = a;        // <- this won't work because strings are immutable
3. console.log(str); // Hello
4. consolelog(str[1]); // e
```

Spread Operator

Spread Operator will copy the value of Reference datatype(or any other) of another data variable without copying the address. They both will point at different memory location.
Syntax : b = [...a]

```
Without Spread Operator
let a = [1, 2, 3, 4]
let b = a;                [ value/s of a = value/s of b] (value/s will be copied)
b[1] = 100;               [ address of a = address of b] (address will also be copied)
console.log(a);           [ they will point at same address where a single array is stored ]
console.log(b);

output:
[1, 100, 3, 4]
[1, 100, 3, 4]

With Spread Operator
let a = [1, 2, 3, 4]
let b = [...a];           [ value/s of a = value/s of b] (value/s will be copied)
b[1] = 100;               [ address of a ≠ address of b] (address will not be copied)
console.log(a);           [ they will have their own separate memory addresses ]
console.log(b);

output:
[1, 2, 3, 4]
[1, 100, 3, 4]
```

Checking Mutability with Object

```
const obj = {
      name : ‘john’,
      gender : ‘male’,
      age : 25
}
const obj1 = obj;
obj.name = ‘steve’;
console.log(obj);
console.log(obj1);

output :
{name:‘steve’,gender:‘male’,age:25}
{name:‘steve’,gender:‘male’,age:25}
```

Mutability is commonly associated with non-primitive data types (e.g., arrays, objects) in most programming languages, including C, C++, Java, and JavaScript. However, JavaScript explicitly labels these as "mutable data types." While the concept is similar across languages, non-primitive types aren't always mutable, as some languages or implementations offer immutable versions.

Comparing Mutable Datatypes

Why comparing Mutable datatypes with same values give false ?
Lets look at an example:

```
let a = ‘john’; // immutable datatype
let b = ‘john; // immutable datatype

let x = {}; // mutable datatype
let y = {}; // mutable datatype

let obj = {    // mutable datatype
```

```
 8.       name : 'steve',
 9.       gender : 'male',
10.       age: 25
11. };
12. let obj1 = {    // mutable datatype
13.       name : 'steve',
14.       gender : 'male',
15.       age: 25
16. };
17.
18. let arr = [1, 2, 3];   // mutable datatype
19. let arr1 = [1, 2, 3];   // mutable datatype
20.
21.
22. console.log(a == b);
23. console.log(obj == obj1);
24. console.log(x == y);
25. console.log(arr == arr1);
26.
27. output:
28. true
29. false
30. false
31. false
32.
```

When we compare two mutable datatypes, it returns false because JavaScript deals with similar object/array properties as a different one. That is in JavaScript, two object/array are considered equal only if they are the same object/array, not if they have the same properties and values, and in JavaScript object/array are assigned and compared by the reference not by the value.

In the above program, when we compared object obj with object obj1, we are comparing their reference address not their value. Because they have different address reference, the comparison will be false.

So can't we get true ?

When we have the same properties and value, you we can!

In order to get true, the object must point at same memory address reference.

```
1. let obj = {name:'steve', age:22};
2. let refobj = obj;  // create a reference to the object
3. console.log(obj == refobj);
4. console.log(obj === refobj);
5.
6. output : true
7.          true
```

Destructuring in JavaScript

Destructuring assignment is a JavaScript expression that allows you to extract data from array, objects and maps and set them into new, distinct variables. Destructuring allows us to extract multiple properties or items from an array at a time.

Here are some examples of Destructuring:

Basic Object Destructuring:

```
 1. const user = {
 2.       name : 'Alex',
 3.       gender: 'Male',
 4.       age:25
 5.    }
 6. const { name } = user;  // variable 'name' should
 7. console.log(name);      // be same as key from the
 8.                         // object
 9. output : Alex
```

Nested Object Destructuring:

```
1. const user = {
2.        name : 'Alex',
3.        address :{
4.             country:'USA',
5.             city:'New York'
6.            }
7.    }
8. const { address:{city} } = user;
9. console.log(city);
10.
11. output : New York
```

Array Destructuring:

```
1. const numbers = [1, 2, 3, 4, 5]
2. const [first, second, third] = numbers;
3. console.log(first, second, third);
4.
5. output : 1 2 3
```

Default Values:

```
1. const  user = {
2. name : 'Alex',
3.        age:43
4. }
5. const {name , age = 25} = user;
6. console.log(name, age);
7.
8. output : Alex 43
```

Rest Parameter:

```
1. const numbers = [1, 2, 3, 4, 5];
2. const [first, second, ...rest] = numbers;
3. console.log(first, reset);
4.
5. output : 1 2 [3, 4, 5]
```

Adding and Deleting key from Object :

Deleting Key from Object:

```
1. const obj = {
2.        name: 'Alex',
3.        age : 25
4. }
5. delete obj.name;
6. console.log(obj);
7.
8. output : {age:25}
```

Adding Key from Object:

```
1. const obj = {
2.        name: 'Alex',
3.        age : 25
4. }
5. obj.gender = 'male';
6. console.log(obj);
7.
8. output : {name:'Alex', age:25, gender:'male'}
```

Why typeof array is object :

In JavaScript, arrays are objects because JavaScript is a prototype-based language This means that there are only primitive types and objects. Arrays are a special case of objects inside the JavaScript engine they have :

- Special handling of indices : Array indices are represented as strings, that contain numbers.
- A length property : The length property of an array indicates the number of elements in the array.
- Methods : Array have a number of methods such as push(), pop(), shift() and unshift() that can be used to add, remove, and manipulate elements in the array.

The type of operator in JavaScript return "object" for arrays. This is because arrays are object in JavaScript even though they have some special properties and methods.

```
1. var arr = [1, 2, 3];
2. console.log(typeof arr);
3.
4. output : Object
```

If you need to check if a variable is an Array, you can use the Array.isArray() method. This method returns true if the variable is an array otherwise false.

```
1. var arr = [1, 2, 3];
2. console.log(Array.isArray(arr));
3.
4. output : true
```

To check the size of array, we use array.length

```
1. let arr = [1, 2, 3]
2. console.log(arr.length); // 13
```

Its is an property of array object, not a method. That's why we didn't use parenthesis at the end of arr.length

---

Browser Object Model / BOM / Window Object

The BOM is used to interact with the browser. The default object of browser is 'window' means you can call all the functions of window by specifying window or directly. For example

```
1. window.alert('hello');  // all three are same
2. alert('hello');
3. this.alert('hello');   // this represents the parent object, in this case the top upper object of alert is window
```

You can use a lot of properties (other objects) defined underneath the window object like document, history, screen, navigate, location, innerHeight, innerWidth etc.

If we print 'this', the window object will be printed.

```
1. console.log(this);
2. console.log(window); // both will print window object.
```

this inside an object

If we try to print 'this' inside an object made by programmer, then this will print the object which is its parent object

```
1. const obj = {
2.        fun : function(){
3.                  console.log(this);
4.         }
5.    }
6. obj.fun();
7.
8. output : {fun:f}
```

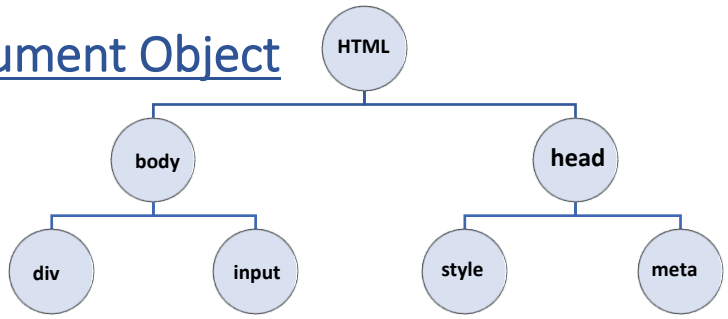
this inside a arrow function

but if we use an arrow function inside an object and try to print 'this' then we might think that it will print the object itself but when using 'this' inside an arrow function. It always print the root parent object which is the window object itself.

```
1. const obj = {
2.        fun : () => {
3.                  console.log(this);
4.         }
5.    }
6. obj.fun();
7.
8. output : window {window:Window,document...}
```

```
1. const obj = {
2.    fun : () =>{
3.       this.console.log(this); // will run because this inside an arrow function represents the window object.
4.                               // which in this cause is correct
5.    },
6.    fun1 : function(){
7.       this.console.log(this); // will throw error because this inside a normal function represents the parent object
8.                               // which is obj and obj doesn't have any console name of function
9.    }
10. }
11.
```

Document Object Model / DOM / Document Object



DOM creates a tree like structure of whole HTML page.

Every node have only three type relationship between other nodes :

1) Parent
2) Child
3) Sibling

In DOM, every element node is an objects

Whenever a new node is added to the tree, whole DOM tree is destroyed and recreated again, this slows down the webpage. That's why we use React which provide Virtual DOM.

Dynamic changes using DOM :

1) Text change
2) Attribute
3) CSS properties
4) HTML Structure

DOM Selectors :

1) **Class** : document.getElementByClassName('class');
2) **Id** : document.getElementById('id');
3) **TagName** : document.getElementByTagName('tag');

→ return HTML Collection

4) document.querySelector('.class'/ '#id' / 'tag'); ----------> return single HTML element that comes first in tree, if there are more, the ignores rest all
5) document.querySelectorAll('.class'/ '#id' / 'tag'); -----> return NodeList

HTML Collection v/s NodeList

HTML Collection	**NodeList**
It only contains tags	It can contain any node like text, attribute, comment, new line etc.
Cannot iterate over its elements using forEach()	It is possible to iterate over it with forEach()
HTML Collection is the HTML DOM is live. It is automatically updated when the underlying document is changed.	It is live & static both in different conditions querySelectorAll() returns a static NodeList

Difference in BOM and DOM

DOM : The Document Object Model (DOM) and the Browser Object Model (BOM) are two important concepts of JavaScript. The DOM is programming interface for HTML and XML documents. It represents the page so that programs can change the document structure, style and context. The DOM provides a standard object model for accessing, manipulating and navigating HTML and XML documents.
BOM : The BOM is collection of objects that represents the browser window and its contents. It allows JavaScript to interact with browser, such as opening and closing windows, navigating history and manipulating cookies.

Change text written between tag :

```
1. element.innerText = 'new_Text';
```

Attribute Selector :
To select the attribute, we can directly write the name of the Attribute.

```
1. imgElement.src = "img1.jpg";
```

Using 'this' keyword in EventListener

Arrow functions always give window object when using 'this' but a normal function give that specific object (parent object) when printing 'this'.

```
X Wrong
1. img1.addEventListener('click', () => {
2.         console.log(this);    // this will print window object
3.         this.src = 'img2.jpeg';  // this won't work, will give error
4. }

✓ Right
1. img1.addEventListener('click', function() {
2.         console.log(this);    // this will print image tag
3.         this.src = 'img2.jpeg';  // this will work
4. }
```

Key events :

- keypress
- keyup
- keydown

Changing CSS with JavaScript of the element :

Syntax : elementobj.style.css-property = "css-propery-value"
eg:

```
1. img1.style.src = “img2.jpeg”;
2. div1.style.backgroundColor = “blue”;
```

Note : when specifying css properties in JS, if there is a hyphen in the name, then it should be removed and make the first letter capital after hyphen.

CSS	JavaScript
border-radius	borderRadius

getComputedStyle()

We can set CSS property of any elementobj with the above syntax but we can’t access or get css property of any elementobj with this method. That’s why we need to use the getComputedStyle() method where we pass the element as argument.
getComputedStyle() method gets the computed CSS properties and values of an HTML element in an object form (in key:value pairs).
getComputedStyle() only take one element as argument. It only work when there is only one element, it doesn’t work the ‘the DOM selector’ that give an html collection or nodelist.
eg:

```
1. getComputedStyle(document.getElementByClassName(‘para’)); // error : HTML collection
2. getComputedStyle(document.getElementByClassName(‘para’)[0]); // run successfully without any error
3. getComputedStyle(document.querySelector(‘p’)); // run successfully without any error
```

```
1. const element = document.getElementById(‘test’);
2. const obj = getComputedStyle(element);
3. let bgColor = obj.getPropertyValue(‘background-color’);
4. console.log(bgColor);
5.
6. output : rgb(173, 216, 230)
```

EventObject (4-july-24)

If we have to make multiple event listeners for multiple element, it would increase code lines and make it lengthy. That’s why e.target is used which is a property of event object ‘e’. e.target gets the whole element where user slick or hover or performed any action.

```
1. <div id=“frame”>
2.         <img src=“imgmain.jpeg”/>
3. </div>
4. <div class=“boxes”>
5.         <img src=“img1.jpeg”/>
6.         <img src=“img2.jpeg”/>
7.         <img src=“img3.jpeg”/>
8. </div>
```

Without EventObject (e.target)

```
1. img1.addEventListener(‘click’, ()=> {
2.         mainimg.src=img1.src;
3. });
4. Img2.addEventListener(‘click’, ()=> {
5.         mainimg.src=img1.src;
6. });
7. Img3.addEventListener(‘click’, ()=> {
8.         mainimg.src=img1.src;
9. });
```

With EventObject (e.target)

```
1. boxes.addEventListener(‘click’, (e)=> {
2.         if(e.target.tagName == ‘IMG’)
3.         {
4.             mainimg.src = e.target.src;
5.         }
6. }
```

We didn’t had to make event listeners for every element. Just added an event listener on parent div and set e.target If target click give IMG tag then pass that img src to mainimg.

---

innerText = only give text that is inside the tag
innerHTML = give entire html element

JS program to show html element on window screen scroll :

```
1. window.addEventListener(“scroll”, () => {
2.         if(window.scrollY >= 100){
3.             para.style.opacity = 1;
```

```
4.         }
5. });
```

---

when targeting section with anchor tag to jump on a section the page jump directly without any smoothness in scroll so to scroll smooth, apply this css property to html tag :

CSS

```
1. html{
2.     scroll-behaviour:smooth;
3. }
```

Date Object

Date() is already inbuild class in JavaScript. We created a date object to access methods/properties of Date() object

```
1. let date = new Date(); // date constructor
2. date.getMonth();
3. date.getHours(); // there are many functions of date, see on web for more
4. // above getHours method give 24 format, to get in 12 format :-
5. let finalHours = (hours>12)?hours-12:hours;
```

Generate Unique IDs using JS's crypto.randomUUID()

The **randomUUID()** method of the Crypto interface is used to generate a v4 UUID using a cryptographically secure random number generator.

When you use crypto.randomUUID() , it generates a new unique number every time because it leverages a cryptographically secure random number generator to create a 128-bit Universally Unique Identifier (UUID), which is essentially a random string of characters with a specific format, making it highly unlikely that the same UUID will be generated twice, even across different systems and times

```
1. let ID = crypto.randomUUID();
2. console.log(ID);
3.  // output : 28e12b51-e53f-4fab-9e40-7f7377e0cd59
```

Delete Parent node/element :

```
1. element.parentNode.remove();
```

to get sibling node :

```
1. element.nextElementSibling;
```

Toggle class :

```
1. element.classList.toggle('class_name');
```

Get height of window/viewport:

```
1. window.innerHeight;
```

get position of any html element according to screen :

```
1. element.getBoundingClientRect();
```

Don't use window.scrollY when doing some changes on scroll in any element because screen size might be different in other devices and this won't work according to planned, that's why window.innerHeight and getBoundingRect() is used together.s

---

Higher Order Functions

In JavaScript, a higher order function is a function that can take another function as argument or return functions are their result.

Some Higher Order Functions are :

- map()
- filter()
- reduce()
- forEach()
- sort()
- reduce
- setTimeout()

---

floor() ---> 2
2.7
ceil() ----> 3

Difference between method and function

>> A function is a block of code written to perform some specific set of tasks. We can define a function using the function keyword by name and optional parameters.

>> A JavaScript Method is a property of an object that contains a function definition. Methods are functions stored as object properties. Object methods can made by following syntax.

```
1. const object = {
2.         methodName : function(){
3.                   // method content
4.         }
5. }
6. object.methodName(); // accessing method of object, just like arr.push() or any other predefined method
```

##############################

Anonymous Function

It is a function that does not have any name associated with it. Normally we use the function keyword before the function name to define a function in JavaScript, however in anonymous function of JavaScript we use only function keyword without the function name.

An Anonymous function is not accessible after its initial creation, it can only be accessed by a variable it is stored in as a function as a value.

We can also declare an Anonymous function using the arrow function.

Anonymous Function:

syntax:

```
1.  function() {
2. // function body
3. }
```

Eg:

```
1. let fun1 = function(a){
2.       console.log(a);
```

Anonymous Arrow Function:

syntax:

```
1. () => {
2. // function body
3. }
```

```
3. }
4. fun1(a);
```

Anonymous function as IIFE:

```
1. (function(){
2.     console.log('Hello');
3. })();
```

Anonymous functions are mostly used in Higher order function or as a IIFE(Immediately Invoked Function Express)

JavaScript Program to remove duplicate values from array (using array's includes() method)

```
1. const arr = [2, 8, 7, 6, 2, 4, 8, 7, 5, 6, 1, 2, 6]
2. const arr1 = [];
3. arr.forEach(
4.       (v) => {
5.             If(!arr1.includes(v)){
6.                   arr1.push(v);
7.                }
8.              }
9.       );
10. console.log("Array After Duplicate Values removed : " + arr1);
```

Practice questions :

- program to find max
- program to find min
- program to find 2nd max
- program to find 2nd min

Asynchronous nature of JavaScript

Nature of JavaScript is Asynchronous like that it will not stop other next lines of code if one line of code is taking too much time, it will run all codes that can run and other codes that take time will print in the end. It is useful but not always, imagine we are fetching some data from somewhere else and when that data get fetch, we will print the data. Due to asynchronous nature of JS, it won't wait for the data to fetch and execute next line where we are printing the fetch, Now that might cause problems because we haven't wait for the data and directly printed it, this will cause garbage data/undefined to appear to user.

That's why Synchronous nature of JS was introduced.

To run the written code is synchronous way, we use Promises.

Asynchronous way:

```
1. let data;
2. setTimeout(()=>{  // we have created a setTimeout function to mimic how a fetched data/connection would take time
3.       data = 'some data';  // to fetch data
4. },5000);
5. console.log(data); // now here the data will be printed 'undefined' because it will take 5 second to fetch data
```

Synchronous Way :

As the name suggest, synchronous means to be in a sequence. i.e. every statement of the code gets executed one by one. Here tasks are performed sequentially with the help of a call stack. Each individual task must complete before the next one can begin. This results in delays in execution if any task takes a significant amount of time to execute.

Promise object is used.

Promise

Promise has 3 States :

1) Pending
2) Fulfill
3) Reject

Promises are :

1) Created and
2) Handled

Mostly, we handle already created promises in APIs when we fetch an API, it return a promise with one of the state from 3 states.

We can't directly access an promises, we need to handle it by two methods :

1. Then and Catch method
2. Async & Await

Back in the days, there wasn't any technology in JS for synchronous data flow, we had to rely on other 3rd party applications like 'Q and BlueBird'

Syntax:

```
1. const promise_name = new Promise((resolve, reject)=>{
2.        resolve({object}/[array]/value); // sending data with resolved promise
3.        reject(throw "error"); // in case need to reject promise
4. });
```

Program to show the use of Promise Creations and Promise Handling in Synchronous way :

https://github.com/69JINX/FrontEnd/blob/main/Javascript/Pizza%20making%20process%20using%20syncnronous%20Promise/index.js

JSON (JavaScript Object Notation) 14-07-24

JSON is a lightweight format for storing and transporting data. JSON is often used when data is sent from a server to a webpage. JSON is 'self-describing' and easy to understand.

This example defines a employees object : an array of 3 employee records (objects):

```
1. {
2.     "employees":[
3.           {"first_name": "John", "last_name": "Doe"},
4.           {"first_name": "Anna", "last_name": "Smith"},
5.           {"first_name": "Peter", "last_name": "Jones"}
6.       ]
7. }
```

JSON syntax rules :

- Data is in name:value pairs
- Data is separated by commas(like object)
- Curly braces hold objects
- Square brackets hold arrays

The JSON format is syntactically identical to the code for creating JavaScript Object.

The JSON syntax is derived from JavaScript object notation syntax, but the JSON format is text only code for reading and generating JSON data can be written in any programming language.

JSON data's datatypes is 'string' because string data is globally used in every language.

XML

From early on, the format that data was transferred between servers and computers, was XML. The best format were open standards that anyone could use and contribute to. XML gained early popularity , as it looked like HTML the foundation of the web. But it was clunky and confusing.

That's where JSON come sin while the format was first developed in the early 2000s, the first standards were published in 2006

JSON in better than XML : https://www.w3schools.com/js/js_json_xml.asp

If we try to build previous employee JSON data in XML, it will look like this :

```
<employees>
        <employee>
            <firstName>John</firstName><lastName>John</lastName>
        </employee>
        <employee>
            <firstName>Anna</firstName><lastName>Smith</lastName>
        </employee>
        <employee>
            <firstName>Peter</firstName><lastName>Jones</lastName>
        </employee>
</employees>
```

When using JSON in our webpage, we need to convert it so it's data can be read on webpage.
We use .json() method to convert it.
It convert JSON to JavaScript Object which can be consumed on webpage.
The json() method of the response interface takes a response stream and reads it to completion. It returns a promise which resolves with the result of parsing the body text as JSON.
Note that despite the method being named json(), the result is not JSON but is instead the result of taking JSON as input and parsing to produce a JS Object .

Fetch() method || Asynchronous Data Fetcher: Converting JSON into JavaScript Objects

Eg:

```
const url = "link_to_JSON_file      ";
async function fetchData(){
       const response = await fetch(url);  // now a Promise is stored in 'response' variable
       return await response.json();       // converts 'response' Promise to a JS readable Object using json()
}
```

The fetch method starts the process of fetching a resource from a server. It returns a Promise that resolves to a Response Object.

```
const x = await fetch(file);
const y = await x.text();
console.log();
```

Event Loop

Cohort 2.0 (Harkirat) => Week 2.1 – Revision of Async (Timestamp : 00:41:27) [The best explanation]

Using 3rd party Libraries/Utilities/Frameworks

JavaScript Core concept :

- HTML
- CSS
- Media Query
- JavaScript

When we try to build any web app/structure site, it take too much time building from scratch with there upper described core concepts, that's where we need to save time to build these web structures fast, so we use code written by other 3rd parties which from them some are open source and some are paid.
Using code from 3rd parties are accessed by coders from Framework & Libraries.
a Framework is like a structure that provides a base for the application development process, with the help of a Framework you can avoid writing anything from scratch. Frameworks provide a set of tools and elements that help in speedy development process. It acts like a template that can be used and even modified to meet project requirements. Some popular frameworks that are most used are
Django, Flutter, Angular, Vue, React Native, Apache Spark, Ionic etc.

Framework provide premade components like button, navbar, Cards, Animation classes, pop-up windows etc.
If you choose JavaScript, then you need to research the JavaScript frameworks such as Angular, Next.js, Vue.js, express.js Bootstrap etc.

React JS is categorized as a library because it provides a collection of tools, including components and functions for use without forcing strict rules or structure for building the entire application .It is the most commonly used front end library for web development .Don't get confused this with React Native. React Native is an open source UI software framework created by Facebook Inc. It is used to develop application for Android, AndroidTV, iOS, macOS, tvOS, Web Windows and UWP. React and React Native both are product of Facebook. React utilizes HTML, CSS and JavaScript to create interactive user interfaces, whereas React Native utilizes API and native UI components to build mobile applications.

- GSAP (Green Sock Animation Platform) JavaScript Animation Libray
- Locomotive (a scroll library)

Library

A library is a collection of pre written code that programmers can use to optimize tasks. Some JS libraries are React, jQuery, FontAwesome etc.

Difference in Framework and Library :

- Libraries tend to be simpler than frameworks and offer a narrow scope of functionality. If you pass an input to a method and receive an output, you probably use the library. We tell our program where we want to call it. This is much like going to a physical library and pulling certain books of the self as we want them. A framework controls the program, we have to fulfill in the blanks of the code, while rest of code written in the syntax of a framework.
- A library is similar to building a house from ground up. You may design your house as you want with just about any architecture you want, and you can arrange your room however you want. Framework on the other hand is like buying a new house. You don't have to deal with construction issues, but you can't pick how to arrange your areas because the house is already completed.
- Libraries are single oriented task, Frameworks are multitasking.

Bootstrap

- Bootstrap is a free front end framework for faster and easier web development.
- Bootstrap include HTML and CSS based design templates for typography, forms, button, tables, navigation models, images, carousels and many other as well as optional JavaScript plugins.
- Anybody with basic knowledge of HTML and CSS can start using Bootstrap.
- Bootstrap responsive CSS adjust to phones, tablets and desktops.
- Bootstrap is compatible with all modern browsers, Chrome, Firefox, Internet Explorer, Edge, Safari and Opera.

Bootstep was developed by Mark Otto and Jacob Thornton at Twitter and released as an open source product in August 2011 on GitHub.
You can

- download bootstrap from getbootstrap.com and host it on your server.
- Include bootstrap from a CDN [3 files : 1 CSS and 2 JS]

https://www.w3schools.com/bootstrap/bootstrap_get_started.asp (read for more details about Bootstrap)

Bootstrap v/s Tailwind

- Bootstrap offers ready to use components for fast development but with limited customization.
- Tailwind CSS provides utility classes for highly customizable designs requiring a bit more effort up front.
- Tailwind CSS might make your site run smoother by removing unused styles.
- Bootstrap is generally easier for beginner, while Tailwind CSS offer more control for those willing to learn.
- Bootstrap allows for quicker prototyping, whereas Tailwind CSS is better for tailored design.

- Bootstrap supports older browser well, Tailwind CSS focuses on the newer ones.

Aspect	Bootstrap	Tailwind CSS
Philosophy	Read made Components	Utility-first Customization
Customization	Limited	Extensive
Development Speed	Faster Initial Development	Slower, but more precise
Performance	Potentially heavier	Light Weight
Learning Curve	Easier	Steeper/hard
Browser support	Broad	Modern

Full Explanation : https://daily.dev/blog/bootstrap-vs-tailwindcss

- We shouldn't use bootstrap and tailwind CSS together because they might interfere with CSS of each other cause they might have same name classes which won't give the result we want.
- Bootstrap is specialised for responsive design.
- Most companies minimise the use of pure CSS and focus only on Bootstrap to save time and to focus on more complex tasks.
- Bootstrap doesn't have uniqueness, but Tailwind have.
- Bootstrap give 3 files in its CDN, 1 CSS file and 2 JavaScript files.
- Read the documentation of Bootstrap to use classes as your requirement. You don't have to remember every class, just read it from docs.

Bootstrap Responsive Sizes :

Bootstrap includes 6 default break points, sometimes referred to as grid tiers for building responsibly.
These breakpoints can be customised if we are using bootstrap source SASS files
https://getbootstrap.com/docs/5.0/layout/breakpoints/#:~:text=Extra%20extra%20large

Tailwind darkMode using Tailwind variables

Process of using tailwind variables :-

1) define variables in global.css in :root{} || --primary: #aedcfa;
2) use those variables in tailwind.config.mjs to define new classes so those can be used with tailwind classes || primary: "var(--primary)",
3) use those classes in your code || bg-primary

Process of using tailwind darkMode :-

1) define darkMode in tailwind.config.mjs || darkMode: "class"
2) use className="dark" in your code to enable dark mode
3) use dark:bg-black to give black color in backgound if className="dark" dark mode

https://github.com/69JINX/FrontEnd/tree/main/React/nextjs/tailwinddarkmodeusingvariables
darkMode : https://www.youtube.com/watch?v=NxIBnvb8B7Y

NodeJS

MERN Stack is called a technology stack. When multiple technologies are grouped together, then it is called a technology stag. Some are.

- MERN => MongoDB / ExpressJS / React / NodeJS
- MEAN => MongoDB / ExpressJS / Angular / NodeJS
- LAMP => Linux / Apache / MySQL / PHP

Our website is stored in database and a server provides the environment for it to code and manipulate etc. MERN stack use JavaScript in everything. NodeJS is used to code server side. Node JS used V8 engine to run JavaScript code. Node JS is just a way to run JavaScript outside the browser. It can be used to run desktop app, server or anything else that we want to do with JavaScript. We can create a web server with node JS.

https://kinsta.com/knowledgebase/what-is-node-js/ (Full Detail of NodeJS)

---

https://www.geeksforgeeks.org/react-jsx-in-depth/ (React)

ReactJs course for Beginners – freecodeCamp (07:10:27)
https://www.youtube.com/watch?v=nTeuhbP7wdE
React surface level understanding
https://www.skillreactor.io/learn/react

LTS (Long Term Support)

Node JS can be used to interact with operating system on which it is installed using OS module. Before NodeJs we had to use other languages for server side coding light like PHP, Java, python, etc. but with node JS we can use JavaScript on server side too . When installing NodeJS on your system, It is recommended that we use LTS (long term support) version instead of latest version because latest version/beta version might have some bugs because it is still new and still in testing state and can cause our code to behave differently than expected. LTS version is bug free and security updates being regularly released. LTS version is recommended for applications that prioritise stability, compatibility and minimises exposure to breaking changes.

Node JS have **Packages/Modules** which is considered to be the same as JavaScript libraries. A set of functions you want to include in your applications.

Built-in modules

Node JS has a set of built in modules which you can use without any further installation.
Look at https://www.w3schools.com/nodejs/ref_modules.asp for a complete list of modules.

Include modules

To include a module, use the "require()" function with the name of the module.

```
1. const http = require('http');
```

Now your application has access to HTTP module and is able to create a server.

```
1. http.createServer((req,res)=>{
2.     res.writeHead(200 , {'content-type':'text/html'});
3.     res.end('Hello World')
4. }).listen(8080);
```

Create your own module

You can create your own module and easily include them in your application. The following example create a module that returns a date and time object.

```
1. exports.myDateTime = () => {
2.      return Date();
3. }
```

Use the exports keyword to make properties and methods available outside the module file. Save the code above in a file called 'showDate.js' docs.
Use the module in index.js file :

```
1. let x = require('./showDate');
2. console.log(x.MyDateTime());
```

now run this file in cmd using the node :

```
1. node index.js // 'node index' will also work | This will output the currentDate
```

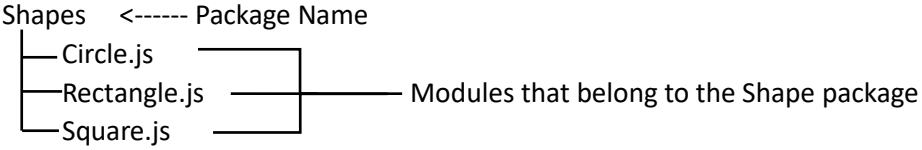
Notice that we use './' to locate the module. That means that the module (showDate.js) is located in the same folder as the index JS file. If you tried to write the code of the index.js file in the HTML file inside the script tag because of you thinking that

it is a JS code and JS code will run on the browser. But you are wrong because the browser DOM will show error on require function 'Uncaught reference error. Require is not defined at index JS'. This is common when you try to use the required in client side JavaScript which runs in a web browser. Web browsers don't support the required function natively because it is part of nodeJS Module system (commons) not part of standard JavaScript.

Difference between module and package.

- Module are libraries for node JS. Examples of modules circle.js, rectangle.js, square.js
- A package is one or more modules, libraries grouped or packaged together. These are commonly used by other packages or a project of your own.

Node JS uses a **package manager** (npm) where you can find and install thousands of packages. Example of package :



in the node JS module system. Each file is treated as a separate module.Package dot Jason file has metadata about all the packages used in a project and about the project.

Module v/s package v/s library

- A module is just a file containing lines of JS code. Uses import or export to exchange information between modules.
- A library uses one or many modules to provide a set of features.
- A package is a downloadable version library. Think of someone putting it in a box and shipping it to you. You can import it and use it in combination with your own code.

Running JavaScript using node in cmd/terminal

We can write and execute the JS code inside our system in cmd using node environment, then it just works like console in the browser. NodeJs uses commonJs

- To do that open the terminal in VBS code or just open cmd directly from window.
- Write 'node' and hit enter. Now you have entered in the node environment where you can write and execute JS code.
- We usually create a JS file and execute that with this syntax : 'node index.js'
- This will give the output of the index.js file in the terminal/cmd.
- To exit the node environment, press ctrl+c 2 times or press ctrl+D typed '.exit'.

NPM (Node Package Manager)

NPM is the world's largest software library (registry). NPM is also a software package manager and installer. The registry contains over 800,000 code packages. Open-source Developers use npm to share software. Many organizations also use npm to manage private development. NPM is free to use. You can download all npm public software packages without any registration or log-on. NPM includes a CLI command line interface that can be used to download and install software.

Install a package from npm:

```
1. > npm install package_name
```

installing npm:

NPM is installed with node JS. This means that you have to install nodejs to get npm installed on your computer.

NPM v/s NPX

NPM is a package management that is used to install, uninstall and update JavaScript packages on your workstation, whereas NPX is a package executor that is used to directly execute JavaScript packages without installing them. The downside of using NPM is that it may be outdated like as we might have installed some package previously on our system and there is a newer version available with bug fixed, new features and etc, but we are still using the older version that we have installed previously with NPM and we don't even know that there might be a newer version of that package published. But NPX always executes the latest version from the library because it directly check the latest version on library and execute it without installing it.

that's why it is recommended to create React project using NPX .We could create React app using NPM by installing it globally once in our system and using that installed version to create react app as many as we want, but we can left behind using the latest version.

Setup/Create a Project using Node.js

We should always create a new folder for every project we create.
2 commands to create a node project

- npm init
- npm init -y

whichever you use, both will create a package.json file automatically which will contain all the information about the project like project name, project dependencies, other packages used by project etc.
It is just that that the npm init command will ask every detail about the project from developer and feed it into package.json file but npm init -y won't ask anything from developer and will create all details with default value eg. choosing package name as of the folder name currently we are in it.
In the package.json file, the main key is specified as index.js because it is the root and main file where we are going to write our code in. But it hasn't been created by npm automatically. Only package..json is created. So we have to create the index.js manually. So now we have got 2 files (index.js and package.json) in our project folder. If we edit the value of test key in package.json to "node index.js" and we type "npm test" in cmd. It will run that script inside package.json file which is executing the index.js file. So finally it will execute the index.js

CommonJS v/s ECMAscript Modules [require v/s import/export]

https://www.youtube.com/watch?v=bU69doALJGU
It is important to learn the module side of node JS because React uses modules to import component but nodejs uses common JS by default. So to utilise the module version of node JS, we have to add a "type" key to the package Jason file and specify its value to "module". If there is no "type" key in package.json, that means the type is commonJS.
Change it to module like that:
"type": "module
Node JS uses commonJS by default in the backend coding.

CommonJS (by default)	**Module**
"type": "commonjs"	"type": "module"
require()	Import/export

Type of Packages in NPM

1) Predefine packages
2) 3rd party packages
3) user define packages (build & release npm packages)

Package Installation :

1) Globally
2) Locally

- If the package is installed globally, then that package can't have different versions for different applications/projects.
- By installing the package locally, we can ensure that each program can have its local package of the desired version. When we install a package globally, we just need Internet for that installation. In future we can create or inherit package in our project from that previously installed package. But installing locally required Internet connection every time we install it because it is getting that package's latest version from the Internet.
- Global packages have advantages too, they can directly run in cmd using node without first opening the project and some packages only run globally and won't execute locally that does not depend on a project.

Types of Dependencies

1) Project dependencies (mostly installed locally. Package.json records it)
2) Developer dependencies (mostly installed globally, eg . nodemon)

Others :

3) Peer dependencies
4) Optional dependencies
5) Bundled dependencies.

When project is finally build and ready to get live, all project dependencies are bundled together in the project, but developer dependencies are not, because those will just cause project to become more heavier if any dependency/package recorded in package.json, it will be bundled together in the final build of the project, so it is necessary to always install all developer dependencies globally

```
Local Installation :
> npm install package_name
Global Installation :
> npm install package_name -g
Unistall package:
> npm uninstall package_name
```

sometimes -f is used to forcefully install a package.

When a package is installed locally, its name & version is recorded in the package.json file.
But when a package is installed globally. It doesn't get recorded in the package Jason, the global install package can be found at

```
> C:\Users\your_pc_name\Appdata\Roaming\npm
```

Locally installed packages (aka project dependencies) are found in the node_modules directory within your project.

Package.json v/s Package-lock.json

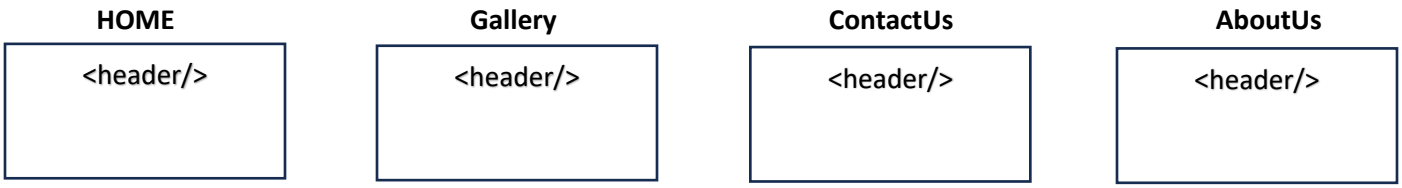
Package.json : This file is primarily used for managing and documenting metadata about the project. Including its name, version, author, dependencies, scripts and other configuration details, it acts as a manifest for the project.
Package-lock.json : This file is generated and updated automatically by npm when installing or updating packages. It is used to lock the exact version of dependencies installed in the project, ensuring reproducibility and consistent installation across different environments.

React

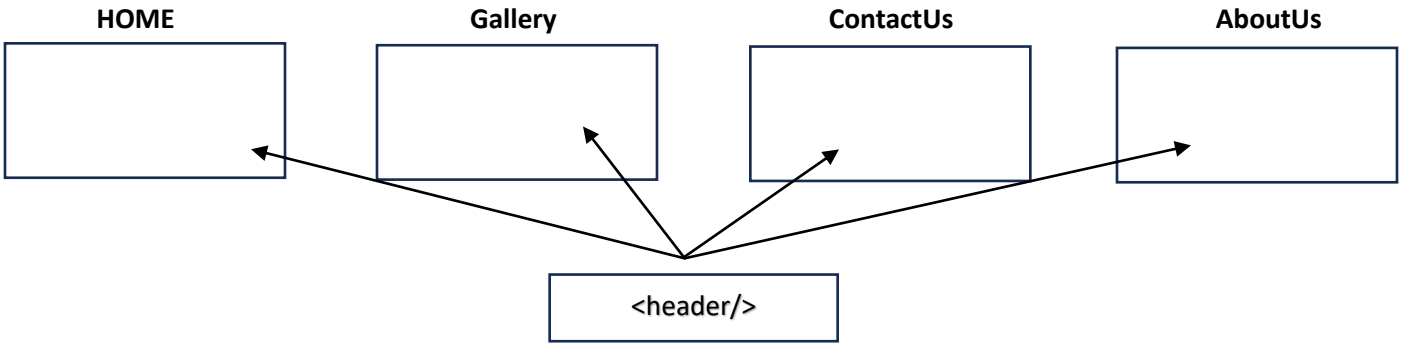
React Basics : https://www.youtube.com/@CodeSketched/videos
React is a free and open source front end JavaScript library for building user interfaces based on components by Facebook Inc. It is maintained by Meta and a community of individual developers and companies. React can be used to develop single page, mobile or server rendered applications with frameworks like Next.js.
React was created to solve the “Ghost message” problem in Facebook. In early days of Facebook, when someone gets any notification or message on Facebook web page, even after reading the message, the notification icon shows number of unread messages until you refresh the page. After refresh, the notification of unread messages gets updated. To outcome this problem react introduced with component based programming. So whenever we interact with any component, only that specific component gets updated without needing to update the whole page. React has fast response than regular website because it doesn't refresh the page on every click. React is based on SPA Logic (Single Page Application).
eg : https://www.niftytrader.in/ is made by React

Because React is component based, we don't have to make same HTML element for different page every single time.
We can just create a component of that and use import that component anywhere on the page.

Without React :



With React :



(made the header only one time and used/imported in all pages. Any HTML element can be made once and use many times in page)

Real DOM v/s Virtual DOM/React DOM

Whenever there some changes occur in the any single node of node tree, in Real DOM, the whole tree is deleted and created from start with the new changes but with Virtual DOM, it only re-render the changing node where changes has occurred without recreating the whole tree.

https://www.geeksforgeeks.org/difference-between-virtual-dom-and-real-dom/

Update Local State Instead of Refetching After a Successful Update API Call

Update Local State Instead of Refetching After a Successful Update API Call

When running update api to updating some data in backend, instead of fetching new data to show latest data after update api, just change the old react state if data has been successfully updated. This will save us from calling fetch api after every update api.

❌ Inefficient Approach (Refetching Data)

```
const handleUpdate = async (updatedItem) => {
  const res = await updateItemAPI(updatedItem);
  if (res.success) {
    fetchData(); // Re-fetches all data again – not optimal
  }
};
```

✅ Recommended Approach (Update Local State)

```
const handleUpdate = async (updatedItem) => {
  const res = await updateItemAPI(updatedItem);
  if (res.success) {
    setData((prevData) =>
      prevData.map((item) =>
        item.id === updatedItem.id ? { ...item, ...updatedItem } : item
      )
    );
  }
};
```

✅ Benefits

Reduces API load by avoiding unnecessary data fetching

Improves performance, especially in large data sets

Provides instant feedback to the user

Keeps state management within React predictable and efficient

Hydration in React

Frameworks like React build the HTML from the JavaScript that you write. But, if you don't run that JavaScript on the server first, the user will simply see a white screen until all your content loads and renders. To fix this, we run the JavaScript first on the server to get the HTML.
But, since it's not interactive because we can't send the Dom nodes with the event listeners, we need to run the exact same JavaScript AGAIN on the client so that things like click listeners actually work. React renders all of your HTML again, but this time it's just trying to add all the event listeners to make your page interactive.
This process is called hydration or sometimes rehydration because the same JS runs twice.
https://www.reddit.com/r/reactjs/comments/18fky3q/what_is_hydration_in_react/
https://github.com/69JINX/FrontEnd/blob/main/React/react/Notes/why-react-uses-hydration.png

Creating a React App12-Aug-24

```
1. > npx create-react-app
2. > npm star // starting the react app server on the localhost in the browser
```

.gitignore : File/Folder name written in the .gitignore file will be not pushes on git eg. node_modules
Every page/component in react is made with .jsx extension (JavaScript + XML) not HTML and when connecting any external css, js, jpg file in the jpsx page, we can't use it's direct url. We have to first import it from its url as a variable/component then use it in our jsx page.

Importing Images in React :

App.jsx

```
1. Import img1 from ‘./images/1.jpeg’
2. function App(){
3.         return(
4.             <img src={img1}/>
5.         )
6. }
7. export default App;
```

When exporting multiple components/functions/images, don't use default. Default is used when only one file is to be exported by-default. When importing App.jsx in other page. Default can only be used one time in a jsx page.

exporting multiple :

exporting multiple :

```
1. export {App, img1};
```

import CSS:

```
1. Import ‘./style.css’ // there is not need to export CSS file
```

Previously in React, all component were made by inheriting classes but now components are made with functions.
Component name's first letter should be capitalized : App, Navbar, Header. By this React automatically identifies that it's a component otherwise it will give error.

React Component :

```
1.      //JS Code write here
2. function App(){
3.      return(
4.        // JSX code write here (a way to use HTML inside JavaScript)
5.     )
6.  }
7.    // JS code write here
```

Inside the return of the function, JSX code will be written, which will finally be pass to the parent element when importing the App Component in parent component.

Blank Fragment :

Whenever returning some JSX code from a component, their should always be an singe parent tag to wrap all JSX tag inside it.

X Wrong

```
return(
    <h1>Hello</h1>
   <span>World</span>
)
// this will through error
```

✓ Right

```
return(
     <div>
       <h1>Hello</h1>
       <span>World</span>
     </div>
  )
```

✓ Right

```
return(
     <>  // blank fragment
       <h1>Hello</h1>
       <span>World</span>
     </>
  )
```

Here in the third example, we have used a Blank Fragment, which will act as a single master parent.

Every tag in the JSX code should be closed. If some tags are only single(img, input), then we should use a backslash at the end to tag to close it. <input type= “text” />

Importing a Component

```
import App from ‘./App.jsx’
<App/>   //using App Component as a tag (self closing tag)
```

Using JS code inside JSX code

Whenever we have to use JS code inside JSX, we use curly braces {}, and write the JS code inside those curly braces.

Using inline CSS

Inline CSS will take an CSS object

```
<img src={img1}
  style = {{
       width: ‘100px’,
       height: ‘100px’
}} />
```

```
const obj = {
    height: ‘100px’,
    width : ‘100px’
  }
<img src={img1} style={obj} />
```

Routing in React

We can’t use anchor tag to go between pages in React. We have to use Routing method of React. We have to install a package for Routing.

```
npm I react-router-dom
```

After installing, we need to create Browser routes =>

Routes will be created in main.js/index.js file, which is the main root file of the project so we can wrap everything in that.

Main.js

```
import {createBrowserRouter, RouterProvider} from ‘react-router-dom’
const router = createBrowserRouter([
      {
         path : ‘/’,              // all routes are identified as object inside an array
         element : <Home/>
      },
      {
         path : ‘/about’,
         element : <About/>
      },
      {
         path : ‘/gallery’,
         element : <Gallery/>
      },
      {
         path : ‘*’,
         element : <h1>404 Not Found</h1>
      }
])
const root = ReactDOM.createRoot(document.getElementById(‘root’));
root.render(
      <React.StrictMode>
```

```
            <RouterProvider router={router} />        // Providing routes
        </React.StrictMode>
);
```

In any page you want to add anchor Link to go on another page/route :

```
Linking(anchor):
1. Import Link from 'react-router-dom'
2. <Link to={'/about'}> About </Link>
```

JSX does not natively support traditional conditional statements like if/else for directly rendering content. However, it supports conditional rendering using the ternary operator and allows the use of array methods like map() for rendering lists. While if/else statements can be used outside the JSX to perform operations or determine conditions, they cannot directly output elements within JSX. To conditionally render elements or content within JSX, the ternary operator or logical operators (e.g., &&) are commonly used.

Components : Components are independent and reusable bits of code. They sere the same purpose as JavaScript functions, but work in isolation and return HTML.

Dynamic Routing

Dynamic routing refers to the ability to create routes that can change based on certain parameters or data. Instead of defining static routes explicitly, dynamic routing allow you to generate routes dynamically, making your application more flexible and scalable.
Because of dynamic routing we don't have to create design for every single product. (Imagine Amazon have so many products they don't design web page for every single product. They have used dynamic routing)
How it works?
Route parameters. You define place holders in your route paths(inside the createBrowserRouter) using the colon notation(:)
These placeholders represent dynamic values that will be extracted from the URL.

```
1. {
2.     path: '/singleproduct/:id',
3.     element: <Product/> // This single component will be used to show all products.
4. }
```

The data for every product will be shown based on the ID or object received in URL. Then that specific product detail will be fetched from API according to the ID receive in URL.
In the link tag we can send the ID or whole product detail separating each detail with backslash on the product component.

```
1. Products.map((product)=>(
2.     <Link to={`/singleproduct/${product._id}`}> {product.title} </Link>
3. ))
```

for sending multiple details in url, we can use below in path :

```
1. {
2.     path: '/singleproduct/:id/:title',
3.     element: <Product/> // This single component will be used to show all products.
4. }
```

The url will look like this of product page :
https://Domain_name.com/singleproduct/36G2d346hN

useParams of react-router-dom

How to access parameters from URL in the single product details page?
The react-router-dom library provides components like Route and useParams to handle dynamic routing. The useParams hook extract the dynamic values from the URL based on the route parameter defined.
In the <Product /> page we will use useParams.

```
1. const {id} = useParams();
2. console.log(id); // This will have the ID of the product from the link which will be used to fetch data from API based on the product ID
```

Benefits

- Flexibility : enables you to create reusable components that can adapt to different data.
- Scalability : Make it easier to manage complex applications with a large number of routes.
- Improved user experience : Provides a more Intuitive and user friendly navigation experience.

Nested Routing :

Nested routing in React allows you to create a hierarchy of routes, where a parent route can have child routes, enabling you to build more complex than organised navigation structures within your application.
We had to import navbar in every page. Nested routing saves us from it. Using <outlet/> by importing Navbar in main route file

[WSB-113 24-Aug-24]

Here's the breakdown of how it works:

- Parent and child roads : You define a parent route, which represents a section of your application, and then next child routes within it, representing sub-sections or specific views.
- Outlet component : The outlet component from react-router-dom is used within the parent route to render the content of the matching child route.
- URL structure : The URL structure reflects the hierarchy of routes. For example, a parent route '/dashboard' might have child routes like '/dashboard/profile' and '/dashboard/settings'.

eg:

main.js

```
1. const router = createBrowserRouter(
2.     createRoutesFromElements(
3.        <Route path="/" element={<Common/>} >
4.              <Route path="" element={<Home/>}>
5.              <Route path="about" element={<About/>}>
6.        </Route>
7. ));
```

Common.js

```
1. import {outlet} from 'react-router-dom'
2. export default function Common(){
3.         return(
4.           <>
5.             <Header/>
6.             <Outlet/> // will be render according to path
7.             <Footer/>
8.           </>
9. )}
```

<outlet/> represents < Home /> and <About/> page based on the path you are on. Because of <Common/> Component, we won't have to include <Header/> and <Footer/> in every single page (Home & About) anymore.

Nested Routing in NextJs (using AppRouting)

https://github.com/69JINX/FrontEnd/blob/main/React/nextjs/nestedrouting/Nested%20routing%20with%20NextJS.png

AppRouting vs PageRouting in NextJS

https://github.com/69JINX/FrontEnd/blob/main/React/nextjs/AppRouting%20VS%20PageRouting.png

---

For ICONS : React Icons

```
1. > npm i react-icons
```

There are 2 types of Bootstrap we can include in our React App.
Bootstrap
Reactive Bootstrap
Installing :

Boostrap:

```
> npm i bootstrap
```

React Bootstrap :

```
> npm i react-bootstrap
```

import 2 files in the main.js page after installing bootstrap using npm. 1 for CSS & other one for JS of bootstrap

```
1. Import 'bootstrap/dist/css/bootstrap.min.css';
2. Import 'bootstrap/dist/js/bootstrap.bundle.min';
```

Passing Date between React Components

In React, there are several ways to send data between components. Here are some of the most common methods.

1) Props (pass data from parent component to a child component)
2) state and callbacks (child component to parent component)
3) Context API (share data between components without passing props down manually)
4) Redux (global state by connecting components to a centralised store)
5) Mobx (manage reactive state by observing changes to observables)
6) Event Emitter (Emit events from child components to notify parent components).
7) React Portals (send data between components rendered in different parts of the DOM)
8) refs
9) Custom HOOKS

PROPS

Imagine a scenario where we want to create an image component and want different image for every time we use that component.

Without Props :

Home.jsx

```
1. import Image from './Image.jsx';
2. <Image/>
3. <Image/>
4. <Image/>
/* in the Home.jsx, we have used the image component
but all images will look same, we want the source of
want the source of image src to change dynamically from
the component For that, we have used Props(properties) */
```

Image.jsx

```
1. import img1 from './images/img1.jpg';
2. const Image = () => {
3.       return(
4.            <div>
5.                 <img src={img1} />
6.            </div>
7.      )
8. }
```

With Props:

Home.jsx

```
1. import image from './Image.jsx';
2. import img1 from './images/img1';
3. import img2 from './images/img2';
4. import img3 from './images/img3';
5.
6. <Image myimage={img1} />
7. <Image myimage={img2} />
8. <Image myimage={img3} />
```

Image.jsx

```
1. const Image = (props) => {  // default parameter props
2.     return(
3.          <div>
4.             <img src = {props.myimage} />
5.         </div>
6.     )
7. }
```

without props, we had to make 3 different components for 3 images because they have different image. But with the use of props, we didn't have to make multiple components, we just created a single Image component and used props to change the src of image

Destructuring Props

main.jsx

```
1. <Image myimage={im1} text="This is a Dog Image" />
/* name should be same from the prop when
   destructuring
   const {myimage, text} = props
   above structure can also be used */
```

Image.jsx

```
1. const Image = ({myimage, text}) => {
2.      return (
          3.            <img src={myimage} alt={text} />
4.       )
5. }
```

Props are arguments passed into React components. Props are passed to components via HTML attributes. React props are like function argument in JS and attributes in HTML. To send props into a component, use the same syntax as HTML attributes. The component receives the argument as a props object(when not destructuring)

ContextAPI

Context API is a feature (or built-in tool) in React that allows you to share state (data) across the entire component tree without passing props manually at every level. A way to create global state that can be accessed by any component, no matter how deeply nested.

So, Context API is:

1. A React concept
2. Implemented through a set of tools (like createContext, Provider, useContext)
3. A mechanism to solve the "prop drilling" problem

useContext

useContext is a React Hook that lets you consume (read/use) a context that was created using the Context API.
You use it inside a component to access the context value. It replaces the older pattern of <MyContext.Consumer> components.
Context API is the whole system (create, provide, consume context). useContext is a part of the system — specifically the hook that reads from the context.

Context.jsx

```jsx
import { createContext, useState } from 'react'
export const counterContext = createContext(0); // default value '0' used when there's no matching provider above in the tree.

export function Context({ children }) {
  const [count, setCount] = useState(0);

  return (
    <counterContext.Provider value={{ count, setCount }}> //Wraps part of your component tree and provides a value to all
      {children}                                          //components inside that tree.
    </counterContext.Provider>
  );
}
```

App.jsx

```jsx
<Context> // wrap the whole app in the created context so all child components can access the context state
  <App />
</Context>
```

Note : In case of app routing, in the layout.js, wrap the {children} prop in <Context>

Accessing value from context in child components with useContext(MyContext) :

```jsx
import { useContext } from 'react';
const { count, setCount } = useContext(counterContext);
```

Redux

Redux is a state management library for JavaScript applications (commonly used with React), especially when you have:

1. A lot of components sharing the same data
2. Complex or deeply nested state
3. A need for predictable, centralized state management

📦 Redux Toolkit (Simplified Redux)

Redux Toolkit is now the standard way to use Redux, because it Reduces boilerplate, provides a better developer experience and includes createSlice, configureStore, etc.
Redux Toolkit Quick Start : https://redux.js.org/tutorials/quick-start
https://github.com/69JINX/FrontEnd/blob/main/Notes/Notes_Pic/Redux1.png
https://github.com/69JINX/FrontEnd/blob/main/Notes/Notes_Pic/Redux2.png
program : https://github.com/69JINX/FrontEnd/tree/main/React/nextjs/Practice/redux

Slice

a slice represents a self-contained piece of the application's state that is managed by a specific reducer. Slices help break down the large, global Redux state object into smaller, more manageable pieces. Each slice typically corresponds to a specific feature or domain of your application.

Store

The store is where we include all redux slices
It is the central container that holds the entire application state(slices). It's a JavaScript object with a few special functions that make it different from a plain global object. The store's main purpose is to manage the state and allow actions to be dispatched to modify it.

Redux Setup :

```
>> npm install @reduxjs/toolkit react-redux
```

counterSlice.js

```
import { createSlice } from "@reduxjs/toolkit";

const initialValue = {
    value: 0,
    loading: false,
    error: null
}

export const counterSlice = createSlice({
    name: 'products', // name of slice
    initialState: initialValue,
    reducers: {            // Functions will be defined in this reducer object that update the state(initialValue) in response to actions
      increment: (state, action) => { state.value += 1 }, // action is the parameter which will be send in the dispatch function
      decrement: (state, action) => { state.value -= 1 }, // state represents the initialValue (updating the value with function)
        incrementByAmount: (state, action) => {
            state.value += action.payload;
        }
    }
})
export const { increment, decrement, incrementByAmount } = counterSlice.actions; // export the reducers
export default counterSlice.reducer;
```

In Redux, the only legal way to update the state is through reducers
You can’t do this:

```
initialValue.value = 10; // ❌ will NOT update Redux state
```

But you can do this:

```
dispatch({ type: 'incrementByAmount', payload: 10 }); // ☑ Triggers reducer, Classic Redux
```

or

```
dispatch(incrementByAmount(10));
```

Note : the reducer function does not support asynchronous tasks (eg. async-await) so we can't fetch some data from api and update the slice state according to that api data. That's where we need to use Redux Thunk. It support asynchronous tasks like api fetching.

store.js

```
import { configureStore } from '@reduxjs/toolkit';
import counterSlice from './slices/counterSlice';

export const store = configureStore({
    reducer: {
        products: counterSlice,
    },
});

Wrap the children in layout.js in Redux Provider :
 import { Provider } from 'react-redux'
 <Provider store={store}>
   {children}
 </Provider>
```

layout.js (Wrap the children in layout.js with Redux Provider) :

```
 import { Provider } from 'react-redux'
 <Provider store={store}>
   {children}
 </Provider>)
```

Preserving SSR with extra steps : [this step can be skipped if you don't want SSR support]
the above code (in layout.js) will force you to use 'use client' and remove metadata from layout.js (which is a server component).
The Redux <Provider> is a Client Component, and when you wrap your RootLayout with it, the whole file becomes a Client Component — which disables: export const metadata & the benefits of server components (streaming, SSR performance, etc.)

To fix this issue & avoid breaking SSR + metadata, you should not wrap the entire RootLayout (layout.js) in Redux <Provider>. Instead leave layout.js as server component & Wrap your main app/page shell (like AppProvider.js) in Redux provider as a Client Component.
FileStructure :

```
.
└── app/
    ├── layout.js          ← server component (no 'use client')
    ├── page.js            ← still server-rendered
    └── providers/
        └── redux-provider.js   ← client component
```

app/layout.js (Server Component — SSR + metadata safe) [this step can be skipped you if don't want SSR support]

```
import ReduxProvider from "./providers/redux-provider";
 <ReduxProvider>
    {children}
 </ReduxProvider>
```

providers/redux-provider.js (Client Component) [this step can be skipped if you don't want SSR support]

```
'use client';
import { Provider } from 'react-redux';
import { store } from '@/store';

export default function ReduxProvider({ children }) {
  return <Provider store={store}>{children}</Provider>;
}
```

App.jsx (use Redux state in any Component)

```
'use client'
import { decrement, increment, incrementByAmount } from '@/slices/counterSlice';
import { useDispatch, useSelector } from 'react-redux';


function App() {
  const count = useSelector(state => state.products.value);
```

```
  const dispatch = useDispatch();

  return (
    <div>
      <h1>Redux Counter</h1>
      <h2>{count}</h2>
      <button onClick={() => dispatch(increment())}>Plus</button>
      <button onClick={() => dispatch(decrement())}>Minus</button>
      <button onClick={() => dispatch(incrementByAmount(5))}>Plus by 5</button>
    </div>
  );
}
export default App
```

Thunk

Redux Thunk is a Redux middleware used to handle asynchronous actions, making it a powerful tool for performing operations like fetching data from an API or any other asynchronous tasks. It allows action creators to return functions that can then be dispatched, enabling asynchronous logic within your Redux application.

When working with Thunk, we need to create extraReducers in createSlice. You would use extraReducers when you are dealing with an action that you have already defined somewhere else.

reducers vs extraReducers : https://stackoverflow.com/questions/66425645/what-is-difference-between-reducers-and-extrareducers-in-redux-toolkit

counterSlice.js (with Thunk):

```
import { createAsyncThunk, createSlice } from "@reduxjs/toolkit";
import axios from "axios";

export const generateRandomNumber = createAsyncThunk(
    'products/generateRandomNumber',
    async (data, thunkApi) => { // data is the parameter which will be send in the generateRandomNumber function as an argument
                                // when calling it, eg : generateRandomNumber(10);
        try {
            const response = await axios.get('/api/random-number');
            return response.data; // returning data so builders can access it,
                                  // this line "state.value = action.payload" won't work if data is not returned
        }
        catch (error) {
            console.log(error);
            return thunkApi.rejectWithValue(error.message);
        }
    }
)

const initialValue = {
    value: 0,
    loading: false,
    error: null
}

export const counterSlice = createSlice({
    name: 'products',
    initialState: initialValue,
    reducers: {
        increment: (state, action) => { state.value += 1 },
        decrement: (state, action) => { state.value -= 1 },
        incrementByAmount: (state, action) => {
            state.value += action.payload;
        }
    },
    extraReducers: (builder) => {   // extraReducers for Thunk
        builder
            .addCase(generateRandomNumber.pending, (state, action) => {     // when api is pending
                state.loading = true;
            })
            .addCase(generateRandomNumber.fulfilled, (state, action) => {   // when api is fulfilled
                state.loading = false;
                state.value = action.payload;
            })
            .addCase(generateRandomNumber.rejected, (state, action) => {    // when api return error (500,400 etc)
                state.loading = false;
                state.error = action.payload
```

```
46.             })
47.     }
48. })
49. export const { increment, decrement, incrementByAmount } = counterSlice.actions;
50. export default counterSlice.reducer;
```

App.jsx :

```
1. <button onClick={() => dispatch(generateRandomNumber())}>Generate Random Number</button>
```

State Management (Dynamic Changes)

In React, dynamic changes refer to updating the user interface(UI) in response to various factors such as :

1) Data changes : When the underlying data that a component relies on changes, React automatically re-renders the component to reflect the updated data. This is achieved through React's State Management and useState and useReducer hooks. Dynamic changes in text/attributes/CSS property/HTML.
2) User interactions: when a user clicks a button, fill out a form or interacts with the page in anyway, React can dynamically update the UI to reflect those changes by using state by re-rendering the component.
3) Conditional rendering : based on certain conditions. You can dynamically render different components or elements. For example you might show loading spinner while data is being fetched or display a different message depending on the users login status.

Key mechanisms enabling dynamic change in React :

1) State : State is a JS object that holds data that can change based on user interaction (click, hover etc.) or on certain condition. When the state changes, React automatically re-renders the components that depend on that state.
2) Props : Changes in props can also trigger re-rendering.
3) Life cycle methods : React components have life cycle methods that allow you to perform actions at specific points in their existence, such as when they are mounted, updated, or unmounted. These methods can be used to update the UI or perform other action in response to changes.

Examples of dynamic changes in React

1) Updating a counter : When a user clicks a button you can increment a counter value in the components state which will cause the UI to display a updated value.
2) Fetching data from API : You can fetch data from an API and store it in the components state. When the data is received, React will update the UI to display the retrieved data.
3) Toggling visibility : You can use a boolean state variable to control the visibility of an element. When the user performs an action you can toggle the state which will cause the element to appear or disappear by re-rendering.

State Management

State management in React refers to the process of managing the state of an application, which includes storing, updating and retrieving data that changes overtime. In React, state is used to store data that affects the rendering of components.

Type of state :

1) local state : stored within individual components using the useState hook.
2) Global state said across multiple components managed using state management libraries or build-in APIs like context API.

Popular State Management Libraries :

- Redux
- Mobx
- Recoil
- Jotai
- Zustand

These libraries terminate the use of probe drilling and avoid passing props deeply through many layers of components.

Reasons for state management

Dynamic user interface: React components need to reflect changes in data which require managing state
Component communication : components need to share data which require centralized state management system
Performance optimization : proper state management minimizes unnecessary renders

Challenges without state management :

1) Data consistency : Data inconsistencies arise when multiple components manage their own state.
2) Debugging complexity : Debugging becomes challenging due to scattered state
3) Performance issues : Unoptimized state updates leads to performance problems.

Benefits of State Management :

1) Centralised data storage: Single source of truth for data.
2) Easy data sharing : Components can access shared data.
3) Efficient updates : Optimised state update, reduce unnecessary renders.
4) Improved debugging : Easier debugging with a centralised state.
5) Better scalability : Efficient state management enables scalable applications.

Best Practices :

1) Keep state minimal : only store necessary data.
2) Use immutable state : prevent unintended side effects. State updates, Create new copies.
3) Use single source of truth : avoid duplicated state.
4) Optimise state update : minimise unnecessary re-renders.

State immutability

[https://www.youtube.com/watch?v=-hi-QQHWlHg]

In React, Immutable state means that you never directly modify the existing state object. Instead when you want to update the state, you create a new copy of the state object with the desired changes. In other words, we don't mutate state if we want to change it. Instead we make a copy of it and replace the old state with the new copy. That's immutability. This approach is important because Reacts reconciliation algorithm relies on immutability to quickly determine what has changed. When you replace the state with a new object. React can efficiently compare the new and old states to update the UI only where necessary. More benefits of immutability are predictable state management and simplified debugging.

In practice, you often manage immutable state in React by creating a new copy of the state object with the updated values rather than modifying the existing state directly.

For Example :

```
1. const [state, setState] = useState({items:[]});

2. X Wrong (Mutating the state directly)
3. state.items.push('NewItem'); // Directly mutating the state array
4. setState({items:state.items}); // Destructive update

5. ✓ Right (Creating a new state object)
6. setState(prevState => ({items:[…prevState.items, 'NewItem']})); // Non-Destructive Update
```

In the correct example, prevstate.item is copied using the spread operator and a new array with added item is created. This maintains the immutability of the state. Changing an immutable state in React typically results in a new reference address for the state. This is a key aspect of immutability. When you update state immutability, you create a new instance of the state object/array rather than modifying the existing one. This means that the reference address of the state will change with each update.

Never changed the value of state by directly assigning in the new value. Always use setState function(updater function/ setter function) because it help maintain immutability which makes state management more predictable and efficient.

Directly changing the state value is called state mutation.

Issues with direct mutation.

1) Inconsistent UI update : React might not detect the changes if the reference of state, object or array has not changed.
2) Performance problems : React's reconciliation algorithm relies on immutability to quickly determine what has changed. Direct mutations can lead to inefficient rendering.
3) Referred approach : Use immutable updates to create new state objects or array rather than modifying existing ones.

Tip :

- That is why state variables are defined as constant (const) to avoid direct accidentally mutations.
- React does not render the component if the new state is same as old state.

```
1. const [arr, setArr] = useState([1, 2, 3, 4]);
2. arr[1] = 8; // Mutable update (wrong)
3. setArr([1, 8, 3, 4]); // Immutable update (Right)
4. // This Mutable Update does not update the React state, it just changes the local variable arr which won't re-render
the component
```

Brief on 'const' states :

State variables are typically declared with const to ensure that the reference to the state variable does not change, while the value that const holds cannot be reassigned. The contents of state like an object or array can be updated through Reacts state updater function. This prevents accidental reassignment of the state variable which could lead to confusing bugs on or unintended behavior. React uses setter function (eg. setState or setNumber from use state) to update the state. These functions handle creating new state value and triggering renders. The const state variable itself is not meant to be reassigned. Instead you use the updated function to request state changes, which React manages internally.

Babel Compiler & Transpiling

https://www.youtube.com/watch?v=4QIfqpP1QKw

Babel is mainly used to convert ECMAscript 2015+ code into backwards compatible JS code that can be run by older JS engines. It allows web developers to take advantages of the newest features of the languages. Babel transforms JSX code into regular JS code which typically uses React's React.createElement() function. This process allows JSX syntax to be interpreted and rendered by JS engine such as those used in web browsers.

For example the following JSX code :

```
For example the following JSX code :
1. const element = <h1>Hello, World</h1>
Gets transpiled by Babel into :
2. const element = React.createElement('h1', null, 'Hello, World');
```

Hooks :

In React Hooks are functions that allow you to use state and other React features like life cycle methods. In functional components, which previously were only available in class components, hooks make it easier to share logic across components without the complexity of higher-order components or render props.

Here are some React hooks.

- useState
- useEffect
- useContext
- useReucer
- useRef

etc.

They all have different functionality.

Create React App using vite

Vite offers a faster and more efficient development experience with modern tooling, while CRA(create-react-app) is more traditional and has some limitations in term of speed and flexibility. Vite is generally a better choice for larger, more performance critical projects. To create a React app using Vite, run this command in cmd :

```
Create react app using vite :
> npm create vite@latest
Run the vite react app :
> npm run dev
```

Vite doesn't bundle or load all the node modules during development because it uses a more efficient approach of serving only the necessary modules when they are requested. Optimising for speed and performance. The full optimization and bundling happen when you run the build for production.

In CRA, the main file was index.js where we edit things but in vite, the main js file is in src folder with the name main.jsx(src/main.jsx)

<StrictMode> (Renders the component 2 times for safety purpose. So user always gets the newly updated value ensuring to not show old data in accidental situations.)

Component Life Cycle

There are 2 types of components in React:

1) Class components
2) Functional components.

Class components are outdated and not used now a days. Instead we use functional components. The React components life cycle refer to the series of phases that React component goes through from its creation and rendering to updated and eventual removal from the DOM while the traditional life cycle methods are associated with the class component. The introduction of hooks has provided a more versatile way to manage component behaviour.
In functional component there are 3 phases in the
React components Life cycle :

1) Mounting phase
2) Updating phase
3) Unmounting phase

Class components life cycle methods

1) constructor()
2) getDerivedStateFromProps()
3) render()
4) componentDidMount()
5) componentWillMount() etc and many more.

Functional component life cycle methods:

1) Hooks (all hooked methods)

We will only be discussing about functional component's life cycle methods and phases.

https://manikandan-b.medium.com/react-functional-component-lifecycle-e8525f8fadea

1) Mounting phase :

- During the mounting phase, a functional component is being created and added to the DOM. In this phase, you typically initialise state and perform any setup that's needed when the component is first rendered.
- useState : The useState hook allows you to add state to your functional components. It replaces the need for a constructor and this.state in class components. You can initialise state and retrieve the current value and a function to update it.
- useEffect : The useEffect hook with an empty dependency array simulates the componentDidMount lifecycle method. It runs the provided function after the component is first rendered. This is a good place to perform data fetching or initial setup.

2) Updating Phase :

- In the updating phase, the functional component is re-rendered due to changes in its props or state. You can use the useEffect hook without an empty dependency array to achieve behaviour similar to componentDidUpdate.
- useEffect : By using the useEffect hook without a dependency array, you can simulate the behaviour of componentDidUpdate. The provided function will run on every render.

3) Unmouting Phase :

- In the unmounting phase, the functional component is being removed from the DOM. The cleanup function in the useEffect hook simulates the behaviour of componentWillUnmount.

- useEffect : By returning a function from the useEffect hook, you can specify cleanup operations to be performed when the component is unmounted.

Note : useEffect() is used to stop the infinity loop of a component when updating any state.

Clean up function / Unmounting.

Unmounting is performed by using return keyword inside the useEffect. The return statement inside the useEffect hook is used to define a cleanup function in React. This cleanup function is executed when the component is unmounted or when the effect is re-run due to changes in the dependencies. Its primary purpose is to clean up side effects like:

- unsubscribing from subscription(eg. websocket , event listeners)
- cancelling network requests
- Clearing timers or intervals
- Any other cleanup logic that needs to run before the component is removed from the DOM or the effect is re executed.

Data Transfer from Child Component to Parent Component using props (24-Aug)

Parent.jsx

```jsx
import child from './Child.jsx'
function Parent() {
    let print = (name) =>{
        console.log(name);
    }
    return(
       <Child fn={print) />
    )
}
export default Parent
```

Child.jsx

```jsx
function Child({fn}) {
     let name = 'John';
     fn(name);
    return();
}
export default Child
```

Axios (node package to fetch APIs) (27-Aug)

Axios is a popular JS library used for making HTTP requests from a web browser or Node..JS. It simplifies the process of sending asynchronous HTTP request to a server and also handles the response.

Difference between Axios and fetch()

- Axios will automatically transforms the server's response data, while with fetch you need to call the response..json method to pass the data into a JavaScript object.
- Axios delivers the data response within the data object, whereas fetch allows the final data to be stored in any variable.

```
> npm i axios
```

Then-catch

```jsx
const axios = require('axios');
useEffect(()=>{
   axios.get('url_of_api')
   .then((response) => {
    console.log(response.data);
    })
   .catch((error)=>{
    console.log(error);
    })
},[])
```

Async-await

```jsx
useEffect(async()=>{
     const response = await axios.get('url');
     console.log(response);
},[])
```

How to use axios in Vanilla JS ?

In Case of Post, we will add a 2nd parameter which will be the date to post. axios.post('url', data_to_post);

useReducer :

https://www.youtube.com/watch?v=CvrBYYh5osw

useMemo :

https://www.youtube.com/watch?v=rRiBpNhFgoM

useCallback :

https://www.youtube.com/watch?v=_AyFP5s69N4

useMemo and useCallback are used to improve the performance of the webapp by utilizing cache in browser and by stopping unwanted re-renders

useActionState : (used in Form Submission)

https://www.youtube.com/watch?v=-aBKrvK5Vn8
https://github.com/69JINX/FrontEnd/tree/main/React/nextjs/Practice/Hooks/useactionstate
(read the comments of the code, you will understand quicker)

memo

Re-render the component only if the props changes, not when the parent component re-renders (normally in react application when a parent component re-renders, all child component of that parent component re-renders even if there are no changes happening in the child components)
memo is a higher-order component (HOC) that helps optimize the performance of your application by preventing unnecessary re-renders of functional components.
https://github.com/69JINX/FrontEnd/blob/main/React/react/Notes/memo.png
a small program to show the use of memo : https://github.com/69JINX/FrontEnd/tree/main/React/nextjs/memo
cons of memo (when to not use it) : https://github.com/69JINX/FrontEnd/blob/main/React/react/Notes/cons-of-memo.png

cache

lets you cache the result of a data fetch or heavy computation. cache is only for use with React Server Components.
https://react.dev/reference/react/cache
When multiple components make the same data fetch, only one request is made and the data returned is cached and shared across components. All components refer to the same snapshot of data across the server render.

```
syntax: const cachedFn = cache(fn); // fn can be any api fetch or heavy computation
```

api.js

```
1. import {cache} from 'react';
2. export default const getTemperature = cache(async (city) => {
3.         return await fetchTemperature(city);
4. });
5.
```

App.jsx

```
1. import { getTemperature } from './api.js';
2. async function AnimatedWeatherCard({city}) {
3.  const temperature = await getTemperature (city);
4.         // ...
5. }
```

If multiple components called getTemperature method with different city, then fetchTemperature will be called that multiple times and each call site will receive different data. The city acts as a cache key.
Pitfall :

Calling memoized function outside of component will not memorize. React only provides cache access to the memoized function in a component. When calling getUser outside of a component, it will still evaluate the function but not read or update the cache. This is because cache access is provided through a context which is only accessible from a component.

```
import {cache} from 'react';

const getUser = cache(async (userId) => {
  return await db.user.query(userId);
});

getUser('demo-id'); // 🚩 Wrong: Calling memoized function outside of component will not memoize.

async function DemoProfile() {
  const user = await getUser('demo-id'); // ☑ Good: `getUser` will memoize.
  return <Profile user={user} />;
}
```

```
function MapMarker(props) {
  // 🚩 Wrong: props is an object that changes every render. So caching won't work and api will call on every render
  const length = calculateNorm(props);
  // ...
}

function MapMarker(props) {
  // ☑ Good: Pass primitives to memoized function
  const length = calculateNorm(props.x, props.y, props.z);
  // ...
} full problem : https://react.dev/reference/react/cache#troubleshooting
```

When should I use cache, memo, or useMemo?
https://react.dev/reference/react/cache#cache-memo-usememo

useDeferredValue

https://github.com/69JINX/FrontEnd/blob/main/React/react/Notes/useDeferredValue.png

useLayoutEffect

useLayoutEffect in React is a hook that allows you to perform side effects that directly interact with the DOM, and it runs synchronously after a component renders but before the browser paints the screen.
useLayoutEffect runs synchronously, while useEffect runs asynchronously after the paint.
If you need to make changes to the DOM that could affect the layout of your component, useLayoutEffect ensures that these changes are made before the user sees the screen update. This can help prevent visual glitches like flickering or jumping elements.
Cons : useLayoutEffect in React essentially blocks the browser from painting the screen until the code inside the effect has finished running, meaning it executes synchronously after DOM mutations but before the browser updates the screen, potentially causing a slight delay in rendering if not used carefully.
useEffect : Asynchronous and non-blocking. Executes after the component has rendered and the browser has painted the updates to the screen.
useLayoutEffect : Synchronous and blocking. Executes after React has made all DOM mutations but before the browser paints the updates
https://www.youtube.com/watch?v=wU57kvYOxT4
https://github.com/69JINX/FrontEnd/blob/main/React/react/Notes/useLayoutEffect.png

useRef

useRef is used to

1. store values that stays persistence across re-renders

2. reference DOM elements

https://github.com/69JINX/FrontEnd/blob/main/React/react/Notes/useRef.png
eg. program : https://github.com/69JINX/FrontEnd/tree/main/React/nextjs/Practice/Hooks/useref
changes in useRef doesn't trigger re-render .

reference DOM elements

```
1. const divRef = useRef(null);
2. console.log(divRef.current.clientWidth);
3. // will print width of div
4. <div ref={divRef}>
5.       Some text;
6.  </div>
```

Store Values that stays persistence across re-renders

```
1. const count = useRef(0);
2. <div>
3. Count is {count.current}
4. </div>
// look at the above eg. program to see use case of useRef
// to store values
```

useTransition

useTransition is a React Hook that allows to update the state without blocking the UI. It returns an array containing two elements: isPending and startTransition. isPending is a boolean indicating if a transition is in progress, and startTransition is a function to wrap around state updates that should be handled as transitions.
Transitions are useful when performing potentially slow state updates, such as filtering a large list or fetching data. By wrapping these updates in startTransition, React can defer them and allow other interactions, like typing in an input field, to remain responsive. This prevents the UI from feeling sluggish or frozen during long-running tasks.
https://www.youtube.com/watch?v=N5R6NL3UE7I
https://github.com/69JINX/FrontEnd/tree/main/React/nextjs/Practice/Hooks/usetransition

key attribute to trigger Mount and Unmount of react component

https://react.dev/reference/react/useState#resetting-state-with-a-key
explanation : https://github.com/69JINX/FrontEnd/blob/main/React/react/Notes/key%20attribute.png

<Profiler>

<Profiler> lets you measure rendering performance of a React tree programmatically.
https://react.dev/reference/react/Profiler

```
1. <Profiler id="App" onRender={onRender}>
2.   <App />
3. </Profiler>
```

```
1. function onRender(id, phase, actualDuration, baseDuration, startTime, commitTime) {
2.   // Aggregate or log render timings...
3. }
```

<Suspense>

```
1. <Suspense fallback={<h2>Loading...</h2>}>
2.   <SomeComponent />
3. </Suspense>
```

lets you display a fallback until its children have finished loading.

🔵 **How Suspense decides when to show fallback:**

✅ In React, **if any child inside <Suspense> "suspends" (i.e., throws a Promise)** →

➡️ React **immediately** shows the fallback loading UI.

✅ Then, **when that Promise resolves** (meaning: the thing the component was waiting for is ready) →

➡️ React **re-renders** and **shows the real UI** (not fallback anymore).

🧠 **In simple words:**

Condition	What React does
Child component is **waiting** (throws a Promise)	Show the fallback
Child component is **ready** (Promise resolved)	Show the real component

```
import React, { Suspense, useState } from 'react'

const Comp1 = async () => {
    // const [state, setState] = useState(null); // Never declare hooks inside async component, this will cause
                                                    infinite re-renderers
    const promise = await fetch('https://official-joke-api.appspot.com/random_joke')
    const data = await promise.json();
    console.log('A Joke', data); // we could also return jsx instead
}

function App() {
    return (
        <Suspense fallback={<div>...Loading</div>}>
            <Comp1 />
        </Suspense>
    )
}
export default App
```

When React sees that a component is an **async function**, it knows:

- "Hey! This function might not give me JSX immediately."
- "It will *suspend* (wait) before returning."

🔵 **In technical terms:**

- If a React function component is an **async function**, it **always returns a Promise** which causes the <Suspense/> to trigger and run the fallback UI
- Not immediately JSX.

Note : never use **Hooks** inside a async component, because :

1. You **pause** (await) before the **hook calls complete**.
2. React gets **very confused** — hooks must run **synchronously** and **immediately**, but execution is paused for network request.
3. Plus, useState causes **state change** → **re-render** → since component is still async → **again throws a Promise** → **again re-renders**.
4. 🎯 **This loop never ends → Infinite re-render!**

Result: 💥 **Infinite rendering, crash, or freeze.**

If you are using async function components with Suspense, you CANNOT use normal hooks like useState, useEffect inside them.

If you need to use hooks → keep the component synchronous, and manage fetch inside useEffect.

If you really want to use hooks **and** data fetching nicely inside a Suspense boundary, you can use **React Query**, **SWR**, or **server components** (in Next.js).

They are built exactly for this!

Error :

hook.js:608 <Comp1> is an async Client Component. Only Server Components can be async at the moment. This error is often caused by accidentally adding `'use client'` to a module that was originally written for the server.

https://react.dev/reference/react/Suspense

More complex example with react.lazy :

https://github.com/69JINX/FrontEnd/blob/main/React/nextjs/Practice/Components/suspense/src/app/Components/App.jsx