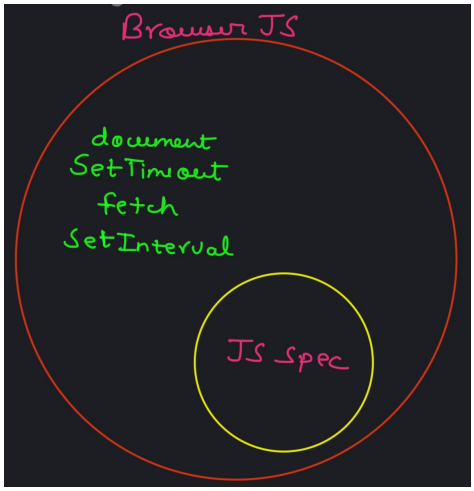
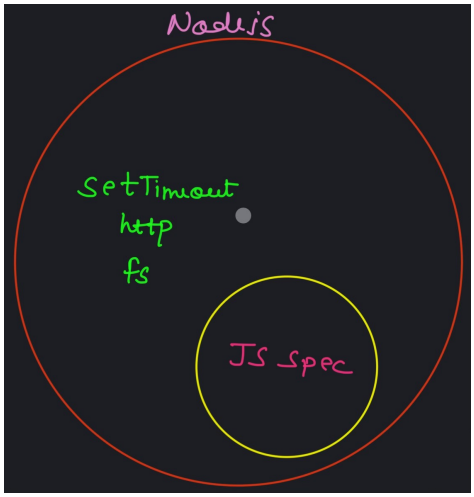
WEEK - 4

INTRO TO DOM

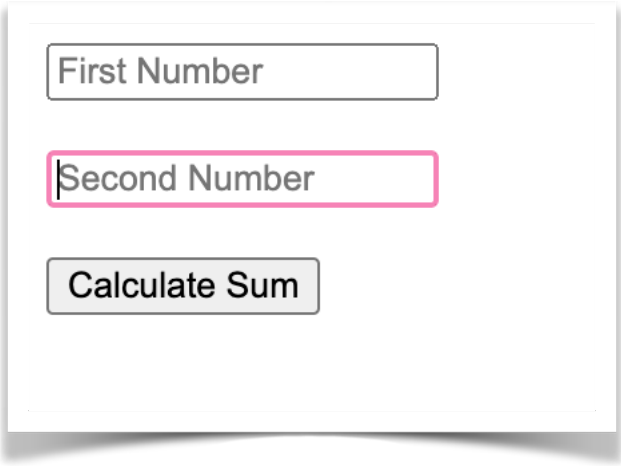


As we know that the browser provides some functionality on top of the javascript spec(which was initially not the part of the ECMA script spec).



Similarly for the node js process.backend also needs some functionality on top of the javascript such as reading file from the file system .which is an async call.

Creating a simple Frontend application such as




```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <input type="text" placeholder="First Number"></input><br></br>
    <input type="text" placeholder="Second Number"></input><br></br>
    <button>Calculate Sum</button><br></br>
</body>
</body>
</html>
```

So what do you mean by dynamic Website?

1.Changing the elements on the website once the website is loaded .
2.Actually calculating the sum based on the inputs and rendering it on the screen.

So that's what DOM facilitates for you to do and in the longer term that's what react makes it very easy for me .

So now lets create another simple frontend for example this -

CODE -

```
<body>

    <div style="background-color: red;margin: 20px;padding: 20px;">Hi there</div>
    <div style="background-color: red;margin: 20px;padding: 20px;">hi there 2</div>

</body>
```

As we see code we are violating the dry principle .so that's why here we use classes . It does the same task with out violating the dry principle .

```
<html>
    <style>
        .container{
            background-color: red;
            margin: 20px;
            padding: 20px;
            color: aqua;
        }
    </style>
    <body>
        <div class="container">Hi there </div>
        <div class="container">Hi there 2</div>
    </body>
</html>
```

What are IDs in HTML?

Ids are supposed to be unique . They are used to uniquely identify a div/span/input/button.

IDs in HTML are unique identifiers assigned to elements within a document to enable targeted selection and manipulation via CSS and JavaScript. Here are the key points about IDs in HTML:

- **Definition**: The ID attribute in HTML provides a unique identifier for an element within a document. It allows specific elements to be targeted and manipulated using CSS and JavaScript for customized styling and functionality.
- **Uniqueness**: IDs must be unique within the entire document, ensuring that each element is uniquely identifiable for styling and scripting purposes.
- **Valid Values**: Valid values for IDs in HTML consist of letters, numbers, hyphens, underscores, and colons. IDs must start with a letter or underscore and should be unique to each element.
- **Selection:** In CSS, IDs are used with the # symbol followed by the ID name to target specific elements for styling. In JavaScript, elements can be selected by their IDs using methods like getElementById().

- **Styling Example**: IDs allow for specific styling of elements. For instance, in CSS, an element with a specific ID can be styled uniquely to stand out from other content, enhancing visual presentation and user experience.

```html
<html>
    <style>
        .container{
            background-color: red;
            margin: 20px;
            padding: 20px;
            color: aqua;
        }
    </style>
    <body>
        <div class="container" id="container1">Hi there </div>
        <div class="container" id="container2">Hi there 2</div>
    </body>
</html>
```

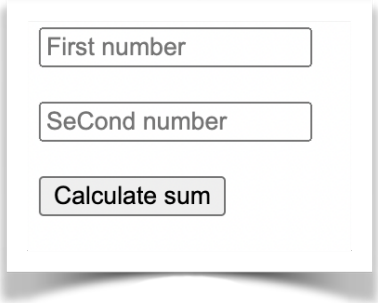
In summary, IDs in HTML serve as unique identifiers for elements, enabling targeted selection and manipulation through CSS and JavaScript. They play a crucial role in customizing the styling and functionality of individual elements within a web document, ensuring precise control over design and behavior.

NOTE - to make website Dynamic We need JAVASCRIPT.

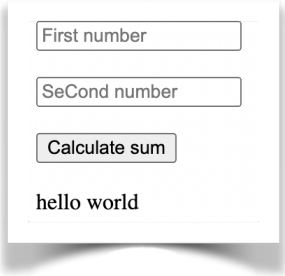
As we go back to our first assignment where we created a frontend to add two numbers . Basically what we need is when the user add two numbers and click on the button calculate the sum .

It shows us a text . (this part is done by the javascript.)

BEFORE THE BUTTON IS CLICKED



AFTER THE BUTTON IS CLICKED-



How it is done ?
This is what DOM lets you do.its help make your website become dynamic.its lets you manipulate your HTML page using JAVAscript.

```html
<html>
<script>
    function populateDiv(){
        const element = document.getElementById("FinalSum")
        element.innerHTML="hello world";
    }
</script>
<body>
    <input type="text" placeholder="First number"></input><br></br>
    <input type="text" placeholder="SeCond number"></input><br></br>
    <button onclick="populateDiv()">Calculate sum </button><br></br>
    <div id="FinalSum"></div>
</body>
</html>
```

DOM

Imagine a web page is like a big tree made of different parts, such as branches, leaves, and fruits. The Document Object Model (DOM) is like a way to organize and understand this tree.

- **Making Web Pages Interactive**: The DOM helps make web pages interactive. Just like you can move branches or pick fruits from a tree, you can use the DOM to change parts of a

web page. For example, you can add new text, change colors, or move things around when a user clicks a button.

- **Working Everywhere**: Different web browsers understand the DOM in the same way. It's like a language that all browsers speak. So, when developers write code using the DOM, it works the same way on Chrome, Firefox, Safari, and others. This makes sure that web pages look and behave consistently no matter what browser people are using.

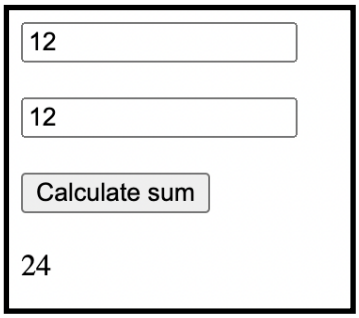
- **Easy Access to Web Page Elements**: Imagine you want to find a specific leaf on a tree. The DOM helps you find specific parts of a web page, like headings, paragraphs, or images. This makes it easy for developers to write code that interacts with these elements.

- **Listening and Reacting**: Just like you can listen for the wind blowing through the leaves of a tree, you can use the DOM to listen for actions on a web page, like clicks or keystrokes. This allows developers to create interactive experiences that respond to what users do.

- **Keeping Things Organized**: The DOM helps keep web page content separate from how it looks. It's like having the structure of the tree separate from its colors and decorations. This separation makes it easier for developers to manage and update web pages without messing up the content.

```html
<html>
<script>
    function populateDiv(){
        const a =document.getElementById("FirstNumber").value
        const b=document.getElementById("SecondNumber").value
        const element = document.getElementById("FinalSum")
        element.innerHTML=parseInt(a)+parseInt(b);

    }
</script>
<body>
    <input id="FirstNumber" type="text" placeholder="First number"></input><br></br>
    <input id="SecondNumber" type="text" placeholder="SeCond number"></input><br></br>
    <button onclick="populateDiv()">Calculate sum </button><br></br>
    <div id="FinalSum"></div>
</body>
</html>
```

And this is how with the help of IDs we complete our assignment .



Q. What is DOM API ?

The DOM API (Application Programming Interface) is a set of rules and tools provided by web browsers to allow developers to interact with the Document Object Model (DOM) of web pages using programming languages like JavaScript.

Here's what the DOM API allows developers to do:

- **Access Elements**: You can use the DOM API to find and access elements within a web page. This includes finding elements by their ID, class, tag name, or even their relationship to other elements in the DOM.

- **Manipulate Elements**: With the DOM API, you can change the content, style, and structure of elements on a web page. For example, you can add or remove elements, change their text or HTML content, modify their CSS styles, or even move them around within the document.

- **Respond to Events**: The DOM API allows you to listen for and respond to user actions and events, such as clicks, keypresses, mouse movements, form submissions, and more. You can attach event listeners to DOM elements and specify the actions to take when these events occur.

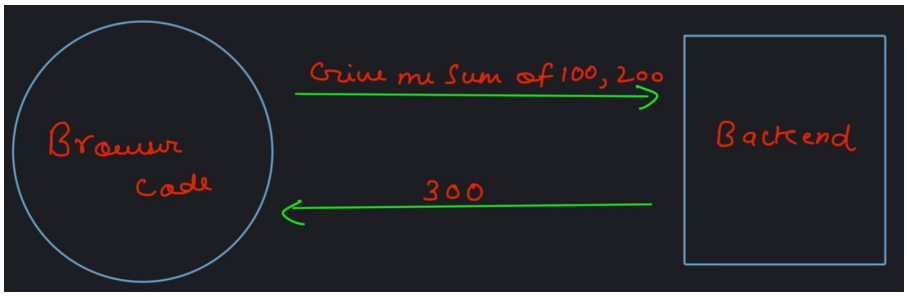
- **Create and Modify Elements**: Using the DOM API, developers can dynamically create new elements and add them to the DOM, as well as remove existing elements or modify their attributes, properties, and contents.

- **Traverse the DOM Tree**: The DOM API provides methods for navigating and traversing the DOM tree. This allows developers to move between different elements, access their parent, child, and sibling nodes, and perform actions based on their relationships within the document.

As we saw previous example of adding two number . We can also do it without clicking on the button. By using onInput on the input tag.

```
<body>
    <input oninput="populateDiv()" id="FirstNumber" type="text" placeholder="First number"></input><br></br>
    <input oninput="populateDiv()" id="SecondNumber" type="text" placeholder="SeCond number"></input><br></br>
    <button onclick="populateDiv()">Calculate sum </button><br></br>
    <div id="FinalSum"></div>
</body>
```

Now let's say you don't have access to the calculation logic on the frontend . Lets assume its a hard problem that someone has exposed on a backend server and you need to hit the backend server and get back the value .



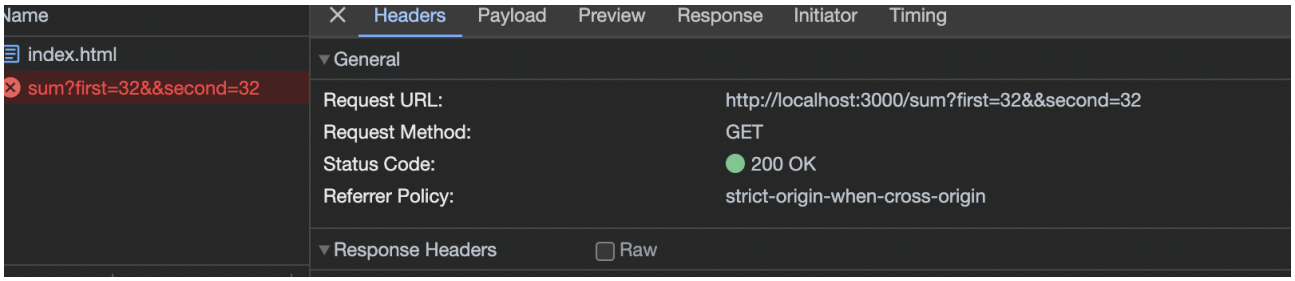
So for that I have created the backend which has the logic sum on it .

```
const express=require("express")
const app=express()



app.get("/sum",function(req,res){
    const first=parseInt(req.query.first)
    const second=parseInt(req.query.second)
    const result=first+second;
    res.send(result.toString())
})
app.listen(3000)
```

To send request to the backend that we have created we need to use fetch api on the front end part-

```
const request= await fetch("http://localhost:3000/sum?first="+a+"&&second="+b)
```

And then we click on the button calculate the sum you will see the request will go out to the server -



Your request will not be send due to cors . It will say “blocked by CORS policy”.

Solution -

```
const express=require("express")
const cors=require("cors")
const app=express()



app.use(cors())
```

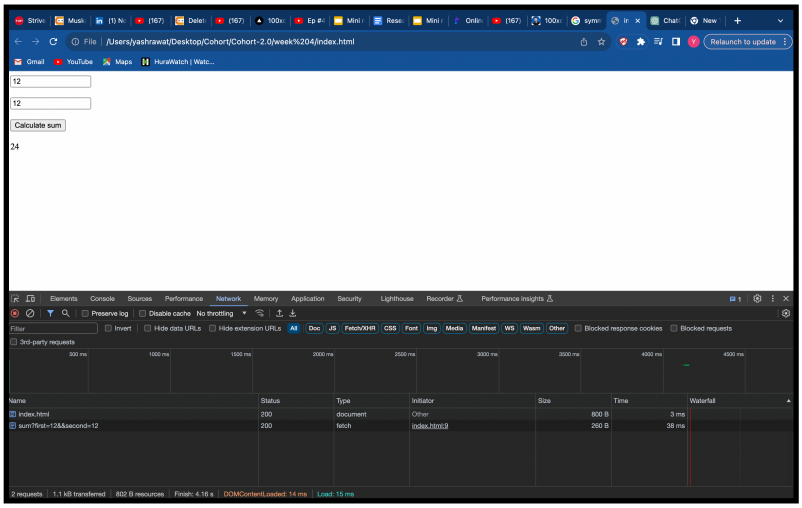
Q What is Cors ?

CORS stands for Cross-Origin Resource Sharing. It's a security feature implemented by web browsers to prevent malicious websites from making unauthorized requests to other websites. When a web page hosted on one domain (origin) makes a request to a resource on a different domain, the browser typically blocks the request for security reasons. CORS provides a way for servers to explicitly allow such cross-origin requests.

A CORS middleware, in the context of web development, is a piece of software that sits between the client (usually a web browser) and the server. It intercepts requests from the client and adds the necessary CORS headers to allow or restrict cross-origin requests based on the server's configuration.

Now this will solve your issue.

```
<html>
<script>
    async function populateDiv(){
        const a =document.getElementById("FirstNumber").value
        const b=document.getElementById("SecondNumber").value
       // const element = document.getElementById("FinalSum")
       // element.innerHTML=parseInt(a)+parseInt(b);

      const request= await fetch("http://localhost:3000/sum?first="+a+"&&second="+b)
      const finalans=await request.text()
      document.getElementById("FinalSum").innerHTML=finalans


    }
</script>
<body>
    <input  id="FirstNumber" type="text" placeholder="First number"></input><br></br>
    <input  id="SecondNumber" type="text" placeholder="SeCond number"></input><br></br>
    <button onclick="populateDiv()">Calculate sum </button><br></br>
    <div id="FinalSum"></div>
</body>
</html>
```



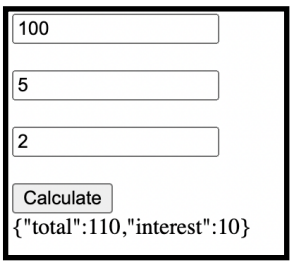
Now building the same assignment again but this time we will find out principal interest -

Frontend

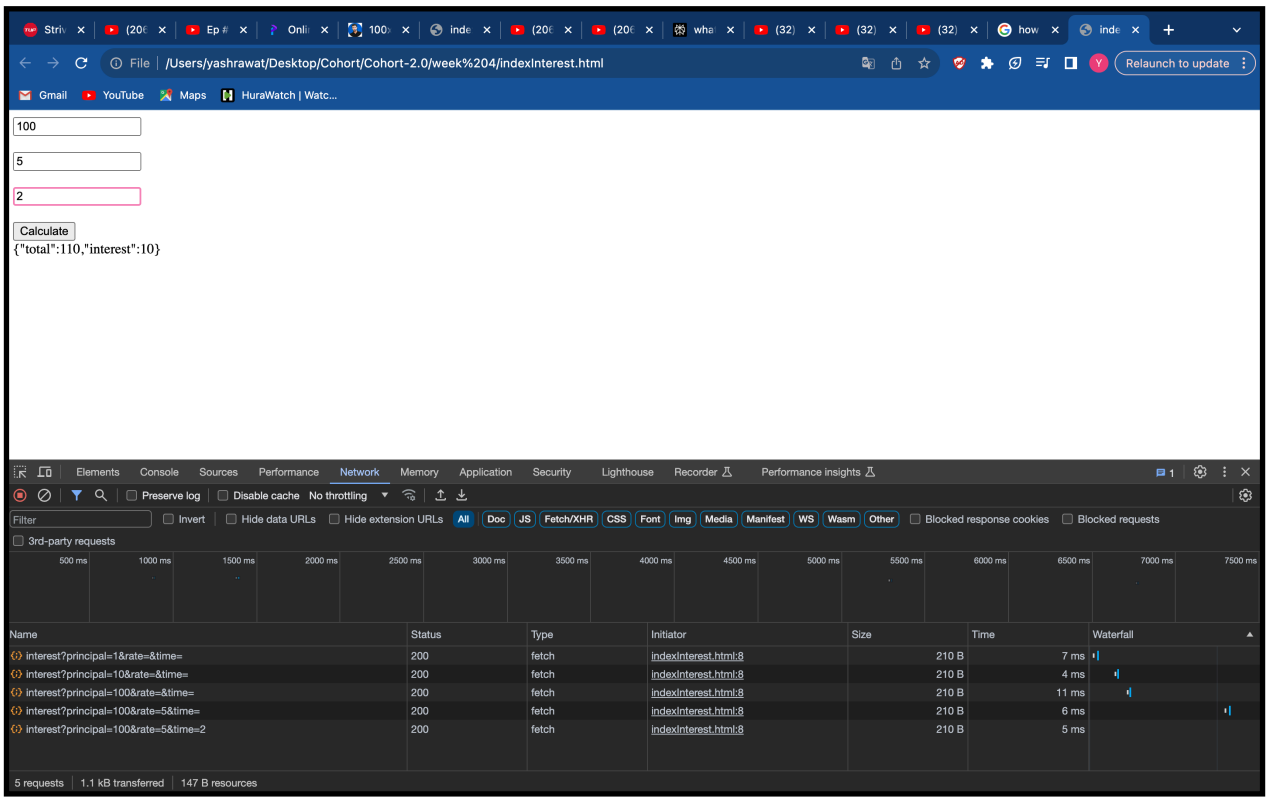
```
<html>
<script>
    async function cal(){
    const p=document.getElementById("Principal").value
    const r=document.getElementById("Rate").value
    const t=document.getElementById("time").value

    const response=await fetch("http://localhost:3000/interest?principal="+p+"&rate="+r+"&time="+t)
    const ans=await response.text()
    document.getElementById("FinalAns").innerHTML=ans;
    }
</script>

<body>
    <input type="text" placeholder="Principal Amount" id="Principal"><br></br>
    <input type="text" placeholder="Rate of Interest" id="Rate"><br></br>
    <input type="text" placeholder="Time" id="time"><br></br>
    <button onclick="cal()">Calculate</button>
    <div id="FinalAns"></div>
</body>
</html>
```

Backend

```
const express=require("express")
const app=express()
const cors=require("cors")

app.use(cors())

app.get("/interest",function(req,res){
    const principal=parseFloat(req.query.principal)
    const rate=parseFloat(req.query.rate)
    const time=parseFloat(req.query.time)
    const interestamount=(principal*rate*time)/100
    const totalamount=principal+interestamount

    return res.json({total:totalamount,interest:interestamount})
})
app.listen(3000)
```



Now I want to do it with out clicking on a button.
As we studied earlier we can do it with input tag on input function .



When I did with without involving button the request that hit backend is 5 . But it can also be done in 3 .so that's why here comes the concept of debouncing .

DEBOUNCING

Debouncing is a programming technique used in various scenarios, particularly in user interface (UI) programming and event handling. It's employed to optimize performance by delaying the execution of a function until after a certain period of time has elapsed without the function being called again.

The primary purpose of debouncing is to prevent a function from being called multiple times in rapid succession, especially when triggered by an event that may fire repeatedly over a short period, such as scrolling, resizing, typing, or button clicks.

Here's how debouncing typically works:

- **Initial Trigger**: When an event (such as a button click or keypress) occurs, the associated function is initially triggered.
- **Delay**: Instead of immediately executing the function, the debounce function starts a timer.
- **Debounce Timer**: If the event occurs again before the timer expires, the previous timer is canceled and a new timer is started.
- **Execution**: After the specified delay (the debounce interval), if no further events have occurred, the function is finally executed.

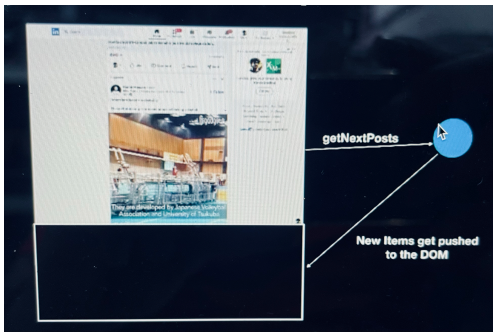
Debouncing helps in scenarios where you want to perform an action in response to an event, but you're only interested in the final state of that event after it has settled down. This is useful for scenarios like auto-saving in text editors (where you want to save changes only after the user has

stopped typing for a moment), or in handling expensive operations like network requests or DOM updates, where you want to avoid unnecessary computations or rendering.

Starting with the week 4.2 class(Intro to react)

The internet has static websites as well as dynamic websites .

Dynamic Websites - the initial website or html page you got from the server is slowly getting being updated .this what we called Dome manipulation it is the heavy concept when the react came into the picture .



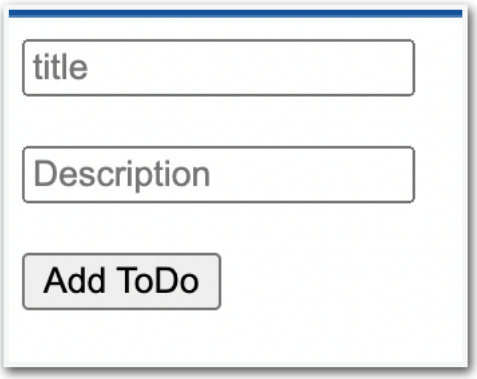
The reason react came into the picture because it was very hard to do DOM manipulation . React makes it slightly easier .

But at high level to create a dynamic website you want to allow people to have some functionality for example as they scroll they get more and more posts.There needs to be a request that goes out to a backend to get more posts and then finally append them to the last slowly and steadily .

DOM manipulation is very hard to write as a developer .making dynamic website with the primitives(such as getelementByid) that Dom provides you is very hard .

Now lets Understand The Dom using the ToDo application

We created this simple front end now what I want is when the user click on the button we will append the title(what the user has entered) along with its Description (again the user has entered) and a button of what that's says mark as done .



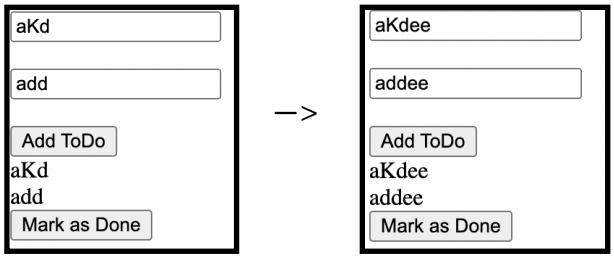
The ugly way of doing it .

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <script>
        function todo(){

            const title =document.getElementById("title").value
            const description=document.getElementById("description").value
            document.getElementById("Container").innerHTML= `
            <div>${title}</div>
            <div>${description}</div>
            <button>Mark as Done</button>
            `
        }
    </script>
</head>
<body>
    <div>
        <input id="title" type="text" placeholder="title"><br>
        <br>
        <input id="description" type="text" placeholder="Description"><br>
        <br>
        <button onclick="todo()">Add ToDo</button>
        <div id="Container">
        </div>


    </div>
</body>
</html>
```

—>

It almost work as the way we want it to . But the only problem is that it override the previous todo Title and description with the new one instead of appending it .

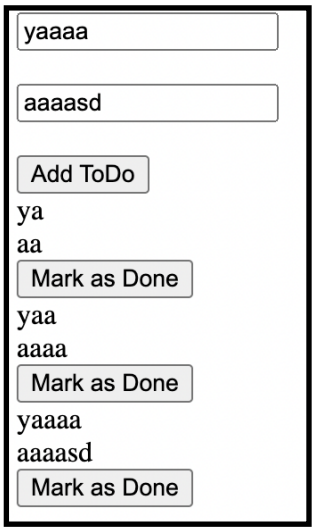
The problem - whenever I want to add another to do it override the previous one .



The solution for it will be -

```
const original=document.getElementById("Container").innerHTML
document.getElementById("Container").innerHTML=original+ `
<div>${title}</div>
<div>${description}</div>
<button>Mark as Done</button>
`
```

—->



This is the ugliest way of doing it .

The Best way of doing it is by the CreateElement.

Lets understand

CreateElement

the `document.createElement()` method is used to dynamically create HTML elements within the Document Object Model (DOM) using JavaScript.

When you use `document.createElement()` in JavaScript, it creates a new HTML element, but it exists only in memory until you append it to the DOM. This means that the element is created and stored in memory, but it's not visible on the webpage until you attach it to an existing element in the DOM using methods like `appendChild()` or `insertBefore()`

```
const child=document.createElement("div")
```

It will create a div element .but it will not be visible on the webpage until you append it .

Now the question is how do you append it .

The answer will be using appendChild.

appendChild

The `appendChild()` method in JavaScript is used to append a child node to the end of the list of children of a specified parent node.

```
const firstgrandChild=document.createElement("div")
firstgrandChild.innerHTML=title
child.appendChild(firstgrandChild)
```

In this first we created first grandchild using createElement then we given title to it using innerhtml .and after that I have appended to the child . Which is the parent node to first grandchild.

```
function createchild(title,description,id){
    const child=document.createElement("div")
    const firstgrandChild=document.createElement("div")
    firstgrandChild.innerHTML=title
    child.appendChild(firstgrandChild)
    const secondGrandchild=document.createElement("div")
    secondGrandchild.innerHTML=description
    child.appendChild(secondGrandchild)
    const thirdGrandChild=document.createElement("button")
    thirdGrandChild.innerHTML="Mark As Done"
    thirdGrandChild.setAttribute("onclick",`markAsDone(${id})`)
    child.appendChild(thirdGrandChild)
    child.setAttribute("id",id)
    return child;
}
```

Created a function and then created a child and appended three of its children .

Now you might be wondering that what setAttribute lets you do .

setAttribute

`setAttribute()` is a method in JavaScript that allows you to set the value of an attribute on a specified element. This method takes two parameters: the name of the attribute and the value to set.

```
thirdGrandChild.setAttribute("onclick",`markAsDone(${id})`)
```

```
function markAsDone(todoId){
    const parent=document.getElementById(todoId)
    parent.children[2].innerHTML="Done!"

}
```

Its first lets you create a o'clock parameter and then value for it .

Now you might be wondering what does children property lets you do.

Children

The `children` property in JavaScript provides a way to access a collection of child elements of a specified element node.

```
parent.children[2].innerHTML="Done!"
```

So this is how you create the TODO .

javaScript Code -

```
<script>
    let globalId=1;
    function todo(){
        const title =document.getElementById("title").value
        const description=document.getElementById("description").value
        const parent=document.getElementById("Container")
        parent.appendChild(createchild(title,description,globalId++))
    }
    function markAsDone(todoId){
        const parent=document.getElementById(todoId)
        parent.children[2].innerHTML="Done!"

    }
    function createchild(title,description,id){
        const child=document.createElement("div")
        const firstgrandChild=document.createElement("div")
        firstgrandChild.innerHTML=title
        child.appendChild(firstgrandChild)
        const secondGrandchild=document.createElement("div")
        secondGrandchild.innerHTML=description
        child.appendChild(secondGrandchild)
        const thirdGrandChild=document.createElement("button")
        thirdGrandChild.innerHTML="Mark As Done"
        thirdGrandChild.setAttribute("onclick",`markAsDone(${id})`)
        child.appendChild(thirdGrandChild)
        child.setAttribute("id",id)
        return child;
    }
</script>
```

Html code -

```html
<body>
    <div>
        <input id="title" type="text" placeholder="title"><br>
        <br>
        <input id="description" type="text" placeholder="Description"><br>
        <br>
        <button onclick="todo()">Add ToDo</button>
        <div id="Container">
        </div>


    </div>
</body>
```

So we have created a TODO application using DOM manipulation .

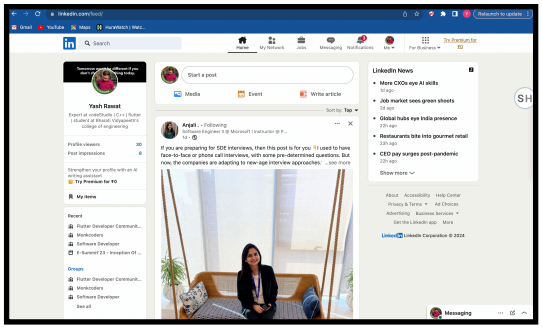
Problem with this approach - very hard to add and remove the elements No central State.

Now what if there was a server where these todos are put what if you update a TODO from your mobile app. You will get back the new array of TODOs on the frontend .

How will you update the DOM then ?
you only have a addtodo function you don't have an updateTodo or revmoveToDo function yet.\

WHAT IS STATE?

In programming, "state" typically refers to the collection of data that represents the condition of a program, system, or component at a specific point in time. It encompasses all the variables, properties, and data structures that describe the current status or configuration of the software.
State is important because it helps to determine how a program behaves and how it responds to input or events. The behavior of a program can change based on changes in its state. For example, in a web application, the state might include information such as the current user's authentication status, the contents of a shopping cart, or the visibility of certain UI elements.

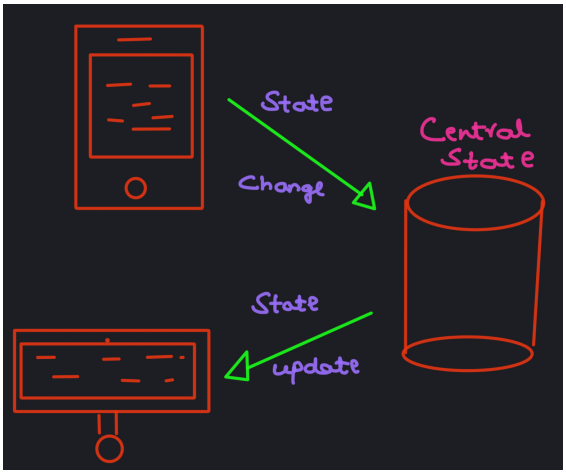


For example state of the linked in page might be -

The state of a LinkedIn page can encompass various aspects depending on the context in which you're considering it. Here are some potential elements that could be described as part of the state of a LinkedIn page:

- **User Authentication Status**: Whether the user is logged in or logged out.
- **User Profile Information**: Information about the logged-in user, such as their name, profile picture, job title, and connections.
- **Network Activity**: Updates on the user's network activity, such as new connection requests, messages, notifications, and posts.
- **Content Feed**: The content displayed in the user's feed, including articles, posts, and updates from connections and followed companies.
- **Search Queries**: Any search queries performed by the user and the corresponding search results.
- **Job Listings**: The job listings displayed on the page, including any saved jobs or recommendations.
- **Messaging**: Any ongoing conversations or messages with connections.
- **Settings and Preferences**: User preferences, such as language settings, notification preferences, and privacy settings.
- **Advertisements**: Any advertisements or sponsored content displayed on the page.

- **Page Layout and UI Components**: The arrangement and appearance of various UI components, such as menus, buttons, and navigation elements.



These elements collectively define the state of a LinkedIn page at any given moment. Managing and updating this state dynamically allows LinkedIn to provide users with a personalized and interactive experience tailored to their preferences and activities on the platform.

Now what we have to do is we are given an array(basically a state on the basis of which we have to render the output)

```
// state will always be an array .
// every element of state would have a title,description and id .
function updateDomAcctoState(state){

}
```

State is something that will come from the server.

we just have to render the updated list based on the state.

NOTE - the script element needs to below the div in which we wanna render it .for example we wanna render it in the container div .otherwise it will throw us null innerHTML error.

```
<br>
<button onclick="todo()">Add ToDo</button>
<div id="Container">
</div>
<script>
    let globalId=1;
  //  function addtodo(){
  //      const title =document.getElementById("title").valu
```

Now the function will look like this -

```
// state will always be an array .
// every element of state would have a title,description and id .
function updateDomAcctoState(state){
    const parent=document.getElementById("Container")
    parent.innerHTML=""
    for(let i=0;i<state.length;i++){
        const child=createchild(state[i].title,state[i].description,state[i].id)
        parent.appendChild(child)
    }
}
updateDomAcctoState([{title:"gym",description:"at 5-6",id:2},{title:"gym 33",description:"at 53-6",id:2}])
```

Now you will question if we wanna just reendow the state element why don't we just manually write it in the container element . Why we using all this javascript logic to render it ?

Actually we don't know how many element does the state array holds . It changes dynamically .and server returns to us the updated list of an arrays .for example think of it as an application which updates by multiple users and every user must have the updated array for that. And needs to be shown in the application when you are using it(means the desktop needs to be refreshed every seconds) .

So that's why for your understanding we created this data .

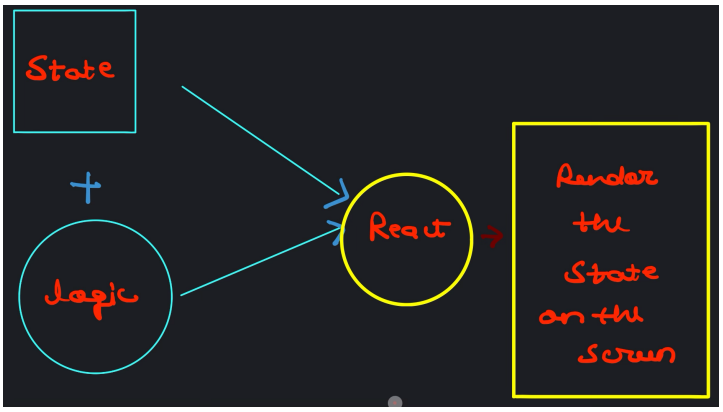
So you simply just have to call the backend after specified time to make sure you have the updated list of array . All the logic of updating the array , adding in the array and removing the element of an array is written on the backend .

You just simply have to call the backend .

```
window.setInterval(async function(){
  const res=await fetch("https://sum-server.100xdevs.com/todos")
  const json=await res.json()
  updateDomAcctoState(json.todos)

},5000)
```

It will automatically call the backend every 5 sec .

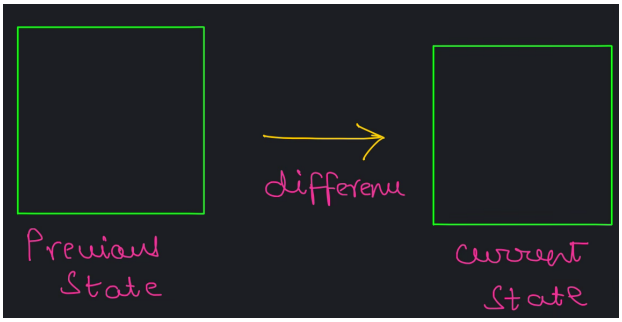
This all what react lets you do in the easy way.



Now if you look at the function we are first clearing the parent element and then adding the element . And if you see the screen you would see in the fraction of seconds that first the list is getting empty and then you see the element on the screen .

Much better approach

Instead of Deleting the list and then showing it on the on the screen . Why don't we calculate the difference between the two state .



Q . What is the easiest way to create a dynamic Frontend Website ?

Ans -
1. Update a state Variable
2. Delegate the task of figuring out difference between the state to a hefty function .
3. Tell the hefty function how to add, update and remove the element .

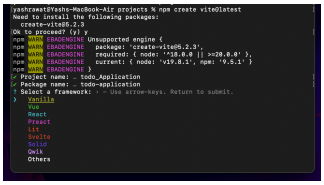
React is just a heffty function which calculate the difference between the state .

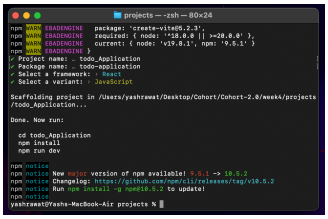
For website - React Dom tells react that is how you update, delete or remove the element from the list.

For application - React Native tells react this is how u put something in the application and this is how you remove something from the application .

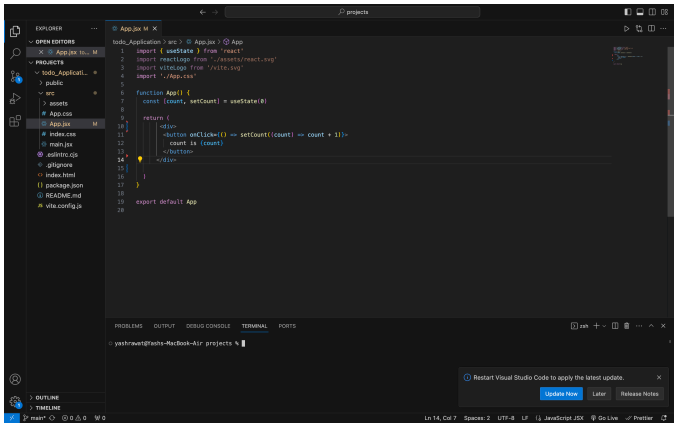
HOW TO CREATE A REACT PROJECT ?

1. to create the react project first create a folder .
2. And then through terminal navigate to that folder .
3. And run command npm create vite@latest.
4. And then write your project name .
5. And then choose react —> javascript .
6. And then your project is created .
7. Open it on the vs code.



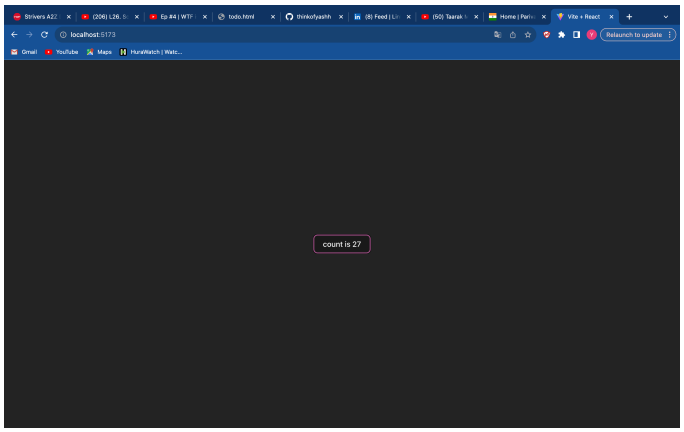


After deleting all the html code from the app.jsx file . The code will look like this .



And then in the terminal
1. Run npm install.
2. Run npm run dev

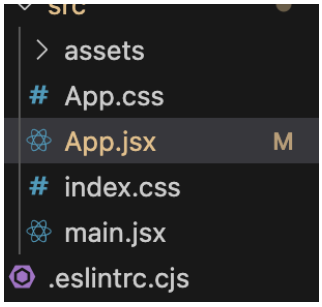
NOTE - Run this command in the same file you created while creating a react project .



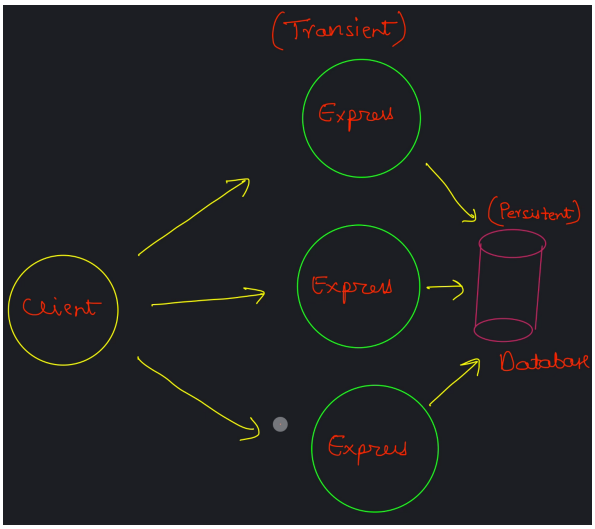
Jsx - file can support html as well as javascript .

Now you might be thinking how nicely it is rendered . Why don't it not just look like a simple html button ?

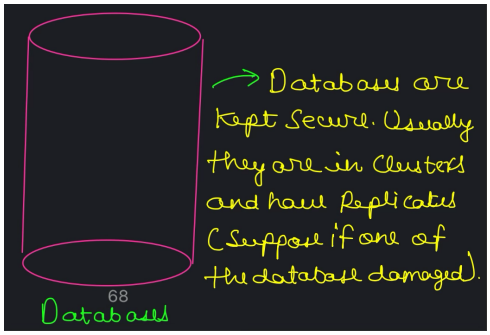
It is due to this css file. Which makes it nicely render the screen .



STARTING WITH 4.3 (Deep Diving Into Mongo DB)



Transient - not permanent ,not lasting (Servers)



Persistent - permanent ,long lasting (Databases)

Examples of data stored in dataBases -

For linked in -
1. User data
2. Users posts
3. Users connection relationships
4. Messages

Q. Why don't we let the user hit the database directly ? What extra does the http server provide exactly ?

1. Databases were created using protocols that browsers don't understand .
2. Databases don't have gradual access as a first class citizen . Very hard to do user specific access in them .
3. There are some Databases(firebase) that let you get rid of the http server and try their best to provide granola access .

For better understanding -
Okay, imagine you have a toy box (the database) with all your toys inside. Now, if you want to play with a toy, you don't just grab it directly from the box because that might be messy and some toys might get broken. Instead, you ask your parent (the HTTP server) to get the toy for you. They make sure it's safe, clean, and that you're allowed to play with it. They also make sure you only get the toys you're supposed to have. The HTTP server helps keep things organized, safe, and fair, just like your parent does with your toys.

Q. What is databases ?

Databases usually allow access to four primitives -
1. Create data
2. Read data
3. Update data
4. Delete data

Popularly know as CRUD.

CRUD stands for Create, Read, Update, and Delete. These are the basic operations that can be performed on any data storage system. Let's break down how these operations might be applied in the context of Facebook:

- **Create**: This operation involves adding new data to the system. In Facebook, this could be creating a new user account, posting a status update, uploading a photo, or creating a new page or group.

- **Read**: This operation involves retrieving data from the system. In Facebook, reading could involve viewing someone's profile, reading a status update, looking at a photo or video, or viewing comments on a post.

- **Update**: This operation involves modifying existing data in the system. In Facebook, updating could involve editing your profile information, changing your privacy settings, editing a post or comment, or updating the details of a page or group you manage.

- **Delete**: This operation involves removing data from the system. In Facebook, deleting could involve deleting your account, removing a post or comment, deleting a photo or video, or deleting a page or group.

Let's go through a specific example:

Scenario: You want to post a status update on Facebook.

- **Create**: You compose your status update and click the "Post" button. This action creates a new post in the Facebook database associated with your account.

- **Read**: Your friends and followers can now read your status update when they visit your profile or when it appears in their news feed.

- **Update**: After posting, you realize there's a typo in your status update. You click the "Edit" button on your post and make the necessary changes. This action updates the content of your post in the database.

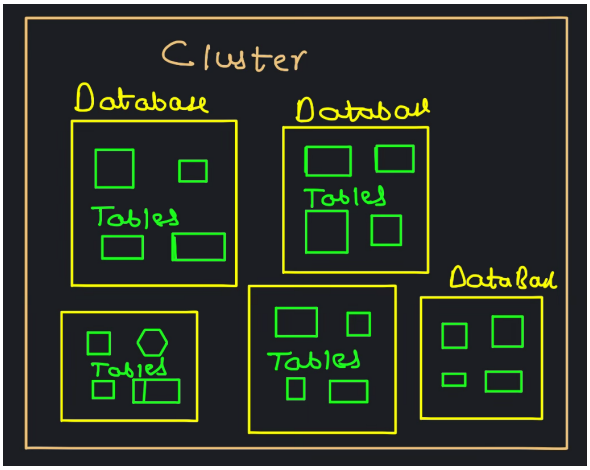
- **Delete**: Later on, you decide you no longer want the post to be visible. You click the "Delete" button on your post, and it is removed from your profile and your friends' news feeds. This action deletes the post from the database.

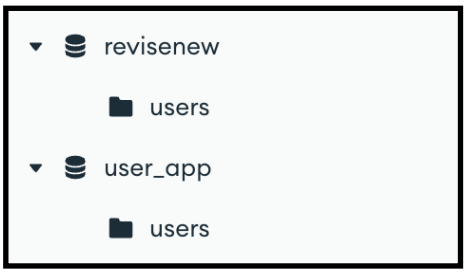
So, these CRUD operations are fundamental to how Facebook (and many other applications) manage and interact with data.

Now to talk to database you see mongoose library.Eventually , we will be using prisma(which is the industry standard way of doing this).

3 jargons you need to know about the databases -

Clusters
Databases
Tables



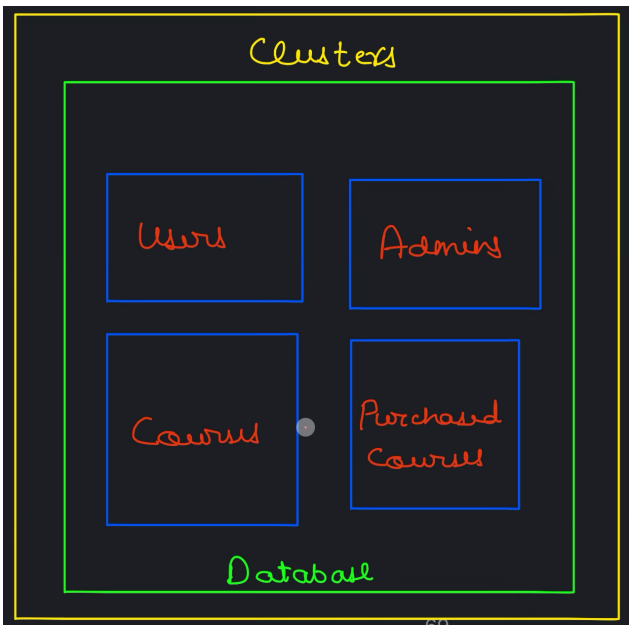


Databases - revise name
Tables - users

Now suppose we are building an application where users can sell the courses ,buy the courses and see the courses . And we have admin too where admin can make there account , sell the courses .

Your data base will look like this -

It has a table for users (where all the users will be there for website),Admins (where all the admins will be there),courses(where all the courses will be there on the website),purchased(where all the purchased courses will be there).

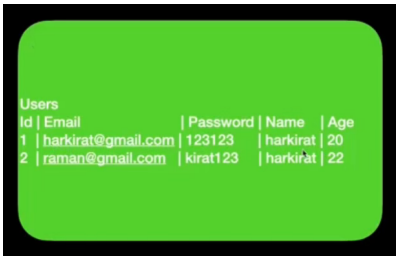


Alternative way-

We can also club the purchased courses table into user schema through 1:many relationship schema).

Q1 What will user scheme will look like ?

It will have id ,email , password , name , age.



Users

Id	Email	Password	Name	Age
1	harkirat@gmail.com	123123	harkirat	20
2	raman@gmail.com	kirat123	harkirat	22

Q2 what will admin schema will look like?

It will have id, email , password , name .

Admins

Id	Email	Password	Name
1	admin@gmail.com	123123	Raman
2	admin2@gmail.com	123123	Kirat

Q3 . What will course Schema will look like ?

It will have id , title , description , price etc.

Courses

id	Title	Description	Price
1.	Full stack	Learn Full stack	5000
2.	Web3	Learn Web3.	3999

Q4 what will purchases schema will look like ?

It will have user id , course id , timestamp , payment _ref.

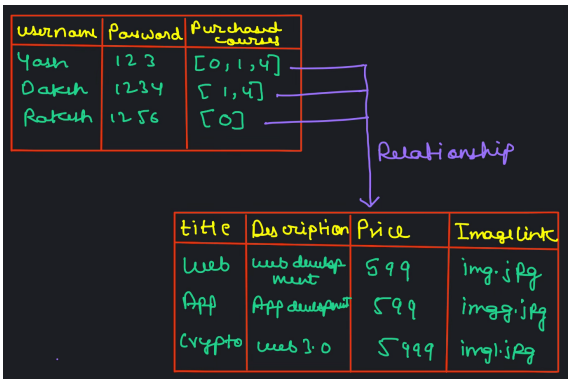
In mongoose you have to define the schema first . This sounds counter intuitive since monogdb is schema less ?
That is true , but mongoose makes you define schema for things like autocompletions/validating data before it goes in the DB to make sure you're doing things right .
Schemeless Dbs can be very dangerous ,using schemas in mongo makes it less dangerous .

So the first thing you have to define the schema .

Step 1 -
We firstly define the user schema this how it looks like . It takes username , password ,and an array which set up relationship with courses.(basically this array stores the id of the courses bought by the user .)

It is the way to establish relationship with other tables.you can only do it in no SQL databases.

```
const UserSchema = new mongoose.Schema({
    // Schema definition here
    username:String,
    password:String,
    purchasedCourses:[{
        type:mongoose.Schema.Types.ObjectId,
        ref:"Course"}]
});
```

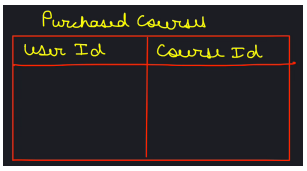



```
const CourseSchema = new mongoose.Schema({
    // Schema definition here
    title:String,
    description:String,
    price:Number,
    ImageLink:String
});
```

It can only happen in no sql databases .where you can store a complex object in the entry(like an array). It can happen in sql databases but it is a ugly way of doing it .

Alternative way of doing the same thing is creating a third table.

This is more sql way of doing things .



But we can do it the no sql too.

So what's happening here is whenever any user does the payment for courses . You just simply put the userid and the course id the user bought in these table .

This is one way of creating a course selling application .

The other way is making use of No sql databases .which can store complex object to establish relationship between the table .

So we will do it No SQL way .

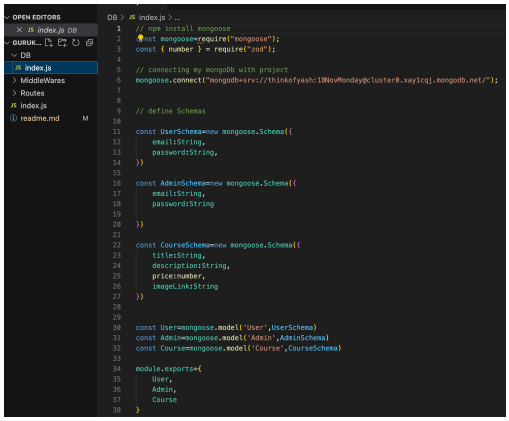
GOOD PRACTICE-

Q. How to structure your code file to make it more readable ?

Step 1- Create Different files for different work.

For example -

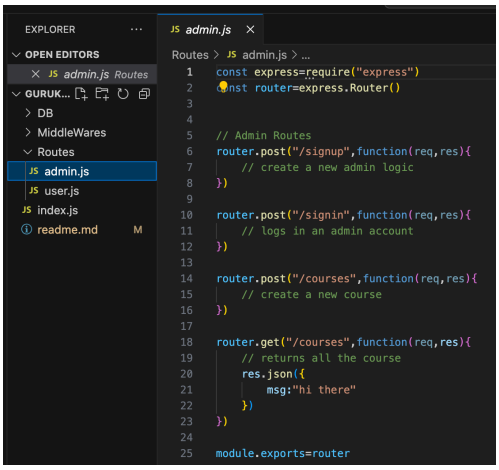
1. DataBases directory -
In which you have to define the database schema.



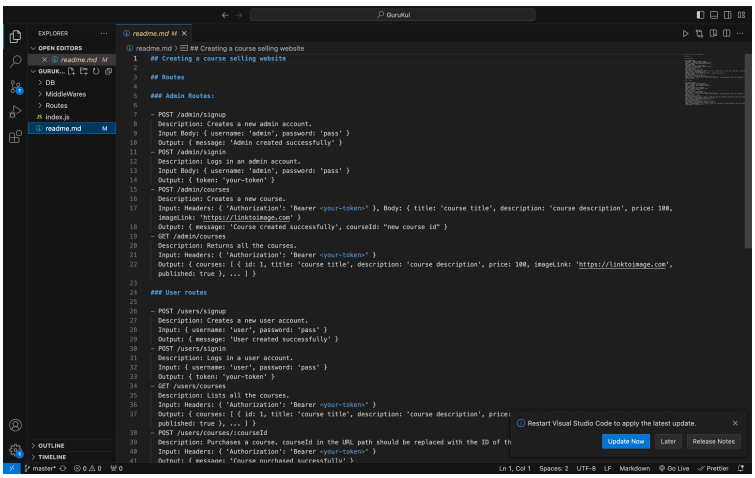
2. MiddleWares directory-
You have to define all the MiddleWares Logic and then export it .

3. Routes directory -
You have to define the different route your backend would take to handle multiple user request.

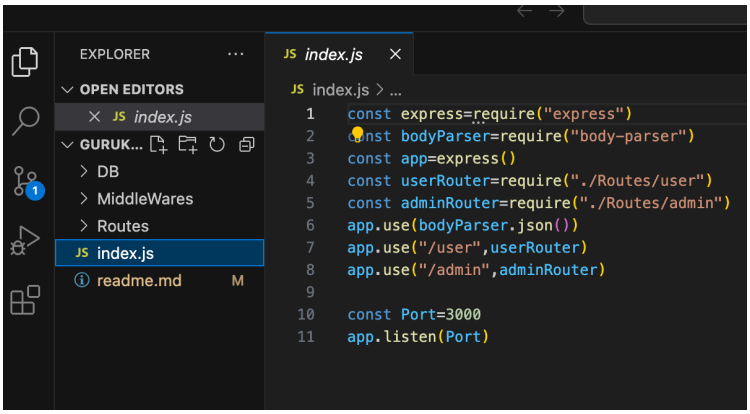
You need to handle direct routes .similarly for user too.



4. Readme file -
Which tells us about the project .(Its routes etc etc.)



5. The main file where all the things are connected.



Q. Why we are using

```
const router=express.Router()
```

?

routers in Express provide a clean and efficient way to structure and manage your application's routes, making your codebase more maintainable and scalable.
As your application grows, you'll likely have many routes handling different endpoints. A router allows you to group related routes together, making your codebase more organized and easier to manage.

For example in our course selling website we have two routes one for user and the other for admin .And within these routes we have sub routes such as signup, sign in etc.

```
const userRouter=require("./Routes/user")
const adminRouter=require("./Routes/admin")
app.use("/user",userRouter)
app.use("/admin",adminRouter)
```

First we imported two files for example user, admin.

And then using app.use we grouped them in two parts one for user and other for admin .

```
const router=express.Router()
```

```
// Admin Routes
router.post("/signup",function(req,res){
    // create a new admin logic
})
```

Now request here will come for “/admin/signup” and not for only “signup”

Similar for user.

RECAP OF JWT

Note - whenever we use jwt Token we first have to import the library .

Name - npm install jsonwebtoken

STEP - 1 -> Standard token that comes is in the way - Bearer <token> .

So if we want real token we have to extract it .

```
const token= req.headers.authorization
const words=token.split(" ")
const jwtToken=words[1];
```

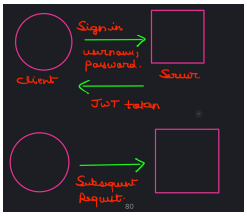
So as we know when we need to verify the JWT token or when we need to sign the token we need to have JWT_Secret.

So we have to create it to.

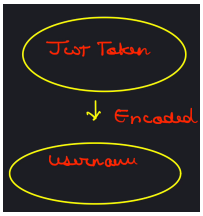
So we will write it in our main file and then export it from that .

```
JS index.js > <unknown>
const express=require("express")
const bodyParser=require("body-parse
const JWT_SECRET="yash_server";
const app=express()
const userRouter=require("./Routes/u
const adminRouter=require("./Routes/
app.use(bodyParser.json())
app.use("/user",userRouter)
app.use("/admin",adminRouter)
const Port=3000
app.listen(Port)
module.exports=JWT_SECRET
```

Import it too to the file we need it . Such as middleWares to (authenticate user)



JWT is included.



We never include password in the jwt token .

One Benfit of JWT - Its save your DataBase Call.

```
MiddleWares > JS admin.js > adminMiddleware
const express=require("express");
const { Admin } = require("../DB");
const jwt=require("jsonwebtoken")
const secret=require("../index")
async function  adminMiddleware(req,res,next){
    const token= req.headers.authorization
    const words=token.split(" ")
    const jwtToken=words[1];
    const decodedvalue=jwt.verify(jwtToken,secret)
    if(decodedvalue.username){
        next();
    }else{
        res.status(403).json({
            msg:"You are not authenticated."
        })
    }

}

module.exports=adminMiddleware
```

Error While Sending Http request on admin/signin route after successfully creating the admin.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

yashrawat@Yashs-MacBook-Air GuruKul % node index.js
(node:36553) Warning: Accessing non-existent property 'Symbol(kKeyType)' of module exports inside circular dependency
(Use `node --trace-warnings ...` to show where the warning was created)
(node:36553) Warning: Accessing non-existent property 'Symbol(kKeyObject)' of module exports inside circular dependency
(node:36553) Warning: Accessing non-existent property 'key' of module exports inside circular dependency
(node:36553) Warning: Accessing non-existent property 'encoding' of module exports inside circular dependency
(node:36553) Warning: Accessing non-existent property 'format' of module exports inside circular dependency
/Users/yashrawat/Desktop/node_modules/jsonwebtoken/sign.js:121
        return failure(new Error('secretOrPrivateKey is not valid key material'));
                    ^

Error: secretOrPrivateKey is not valid key material
    at module.exports [as sign] (/Users/yashrawat/Desktop/node_modules/jsonwebtoken/sign.js:121:24)
    at /Users/yashrawat/Desktop/GuruKul/Routes/admin.js:43:25
    at process.processTicksAndRejections (node:internal/process/task_queues:95:5)

Node.js v19.8.1
yashrawat@Yashs-MacBook-Air GuruKul %
```

It is because of JWT_SECRET key which is associated in the main file. To resolve this error we have to export it from separate file (we have to create the new one)

A new file is created name - config.js and then JWT_SECRET is imported in such a way that JWT_SECRET is a field(new way).

And it is not directly exported from the main file .(old way)

NEW WAY

```
JS config.js > <unknown> > JWT_SECRET
module.exports={
    JWT_SECRET:"yash_server"
}
```

OLD WAY

```
module.exports=JWT_SECRET
```

And then change whenever we have used the previous way(old ways).

Q. If we are sending the JWT token in the Subsequent request so what we will do when the user want to buy/purchase the course .beacuse we will no longer send the user name in the headers we will just send the JWT token ?

```
router.post("/courses/:courseId",userMiddleWare,async function(req,res){
    // buy courses
    const courseId=req.params.courseId
   // const username=req.headers.username (this will be no longer allowed since we are only sending the JWT token in the headers)

    try{

       await  User.updateOne({username:username},{
            '$push':{purchasedCourses:courseId}
        })

        return res.json({msg:"SuccessFully Purchased the Course."})

    }catch(err){
        return res.status(500).json({msg:"Something Went Wrong"})
    }


})
```

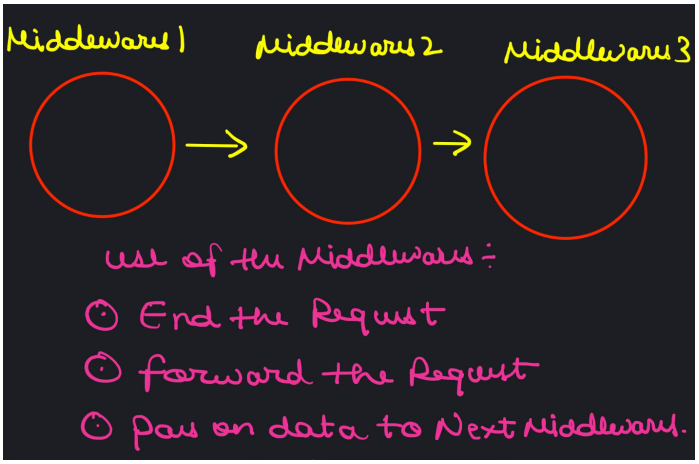
Dumb way - we will send the username along with JWT token . In the headers .
The problem is what if the user sends the JWT token of the username1 and the username as username2.

So technically the username1 has bought the course but we have given access to username2.

Better way - as we able to decode the username in the middleware from the jwt token .

```
try{
    const decodedvalue=jwt.verify(jwtToken,JWT_SECRET)
if(decodedvalue.username){
    next();
}else{
    res.status(403).json({
```

What is the use of the middleWares ?



So we will use the third property of the middleWares .
We will pass on data to the next middleWare for example -

```
if(decodedvalue.username){
    req.username=decodedvalue.username
    next();
```

```
router.post("/courses/:courseId",userMiddleWare,async function(req,res){
    // buy courses
    const courseId=req.params.courseId
    const username=req.username
   // const username=req.headers.username (this will be no longer allowed since we are only sending the JWT token in the headers)

    try{

       await  User.updateOne({username:username},{
            '$push':{purchasedCourses:courseId}
        })

        return res.json({msg:"SuccessFully Purchased the Course."})

    }catch(err){
        return res.status(500).json({msg:"Something Went Wrong"})
    }
})
```