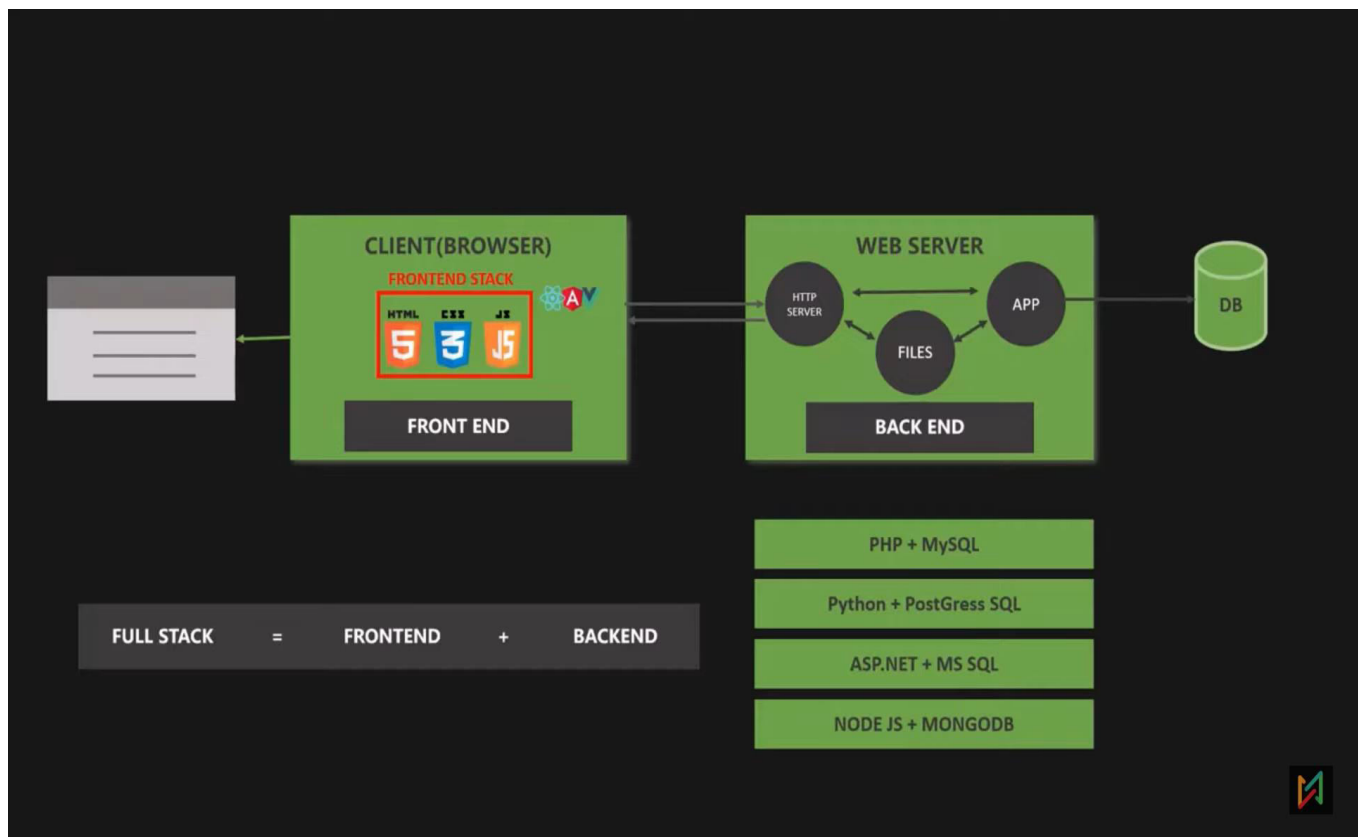


The complete guide for MERN stack



FRONTEND

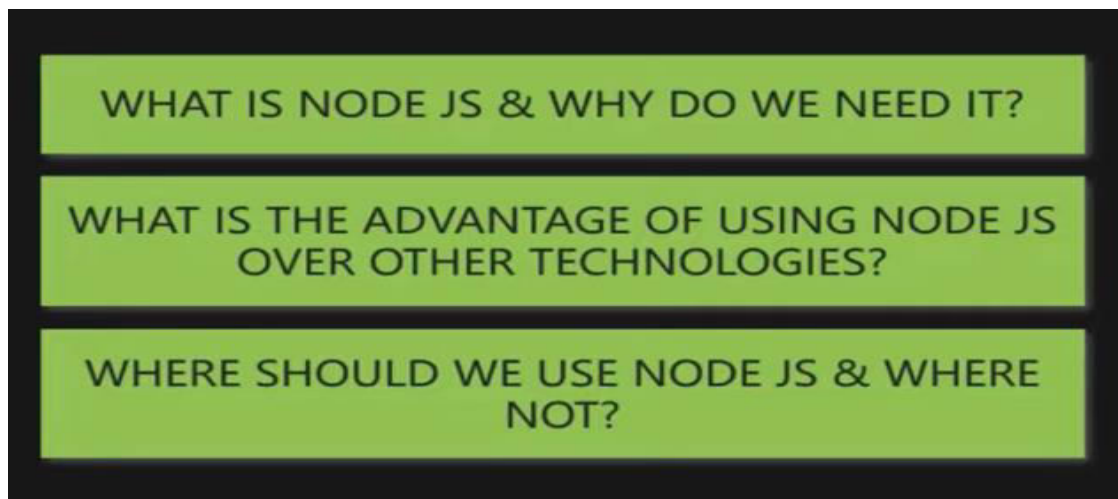
It is just html+css+js or react and nothing else;

BACKEND

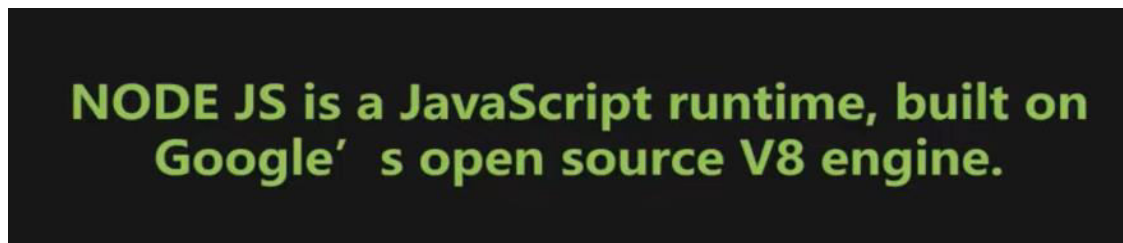
Whenever we create a mern stack project, we follow certain steps:-

1) NODE.JS:-

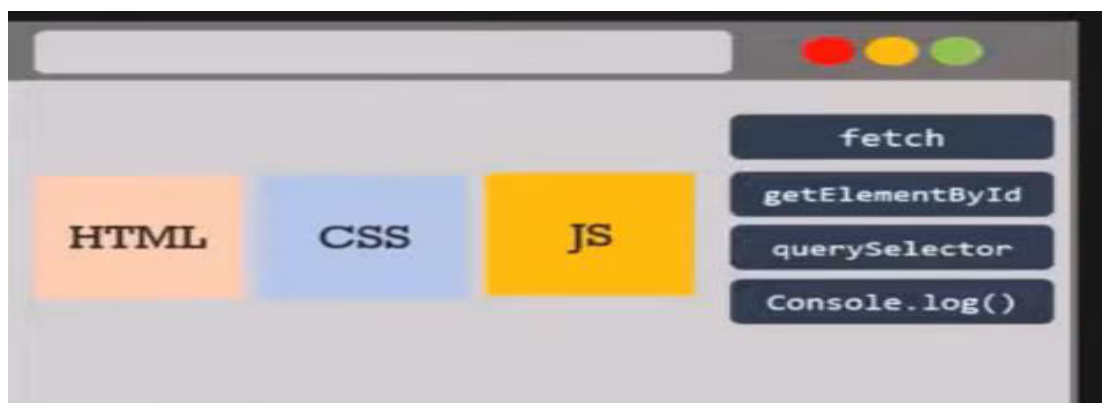
First of all, we have to answer some of the following questions:-



So node.js is :-



NOTE:- when we run our javascript on the browser(means whenever we create a website of html+css+js then we do inspect in the browser and see on the console for error or any message that we have written in vs code), then the browser provides some of the api's and functions :-



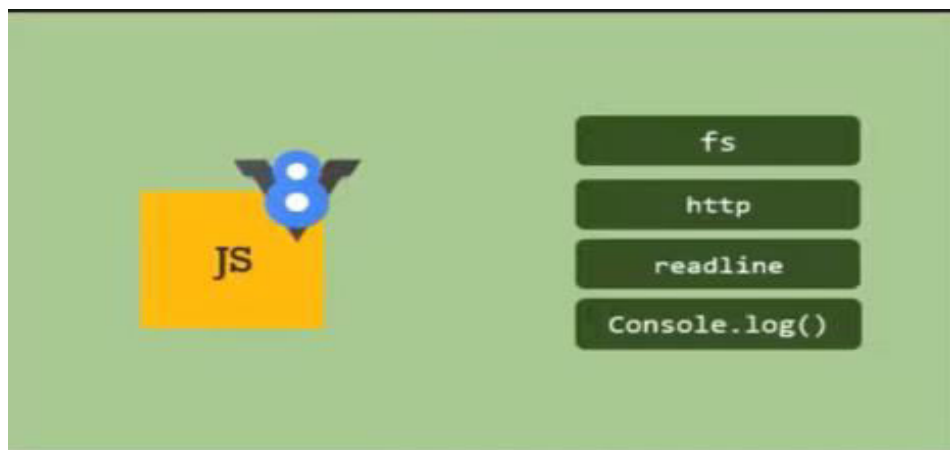
These fetch and getElementById are the ex of API and functions provided by the browser when js is running in the browser.

Every browser has its javascript engine, to run the javascript code.

When the V8 engine is embedded in C++ language to run javascript code outside the browser then this is termed node.js.

So, node.js is a runtime environment for executing javascript code and it also contains its javascript engine(V8 engine) to run the code.

Node.js also provides some of the functions and APIs:-

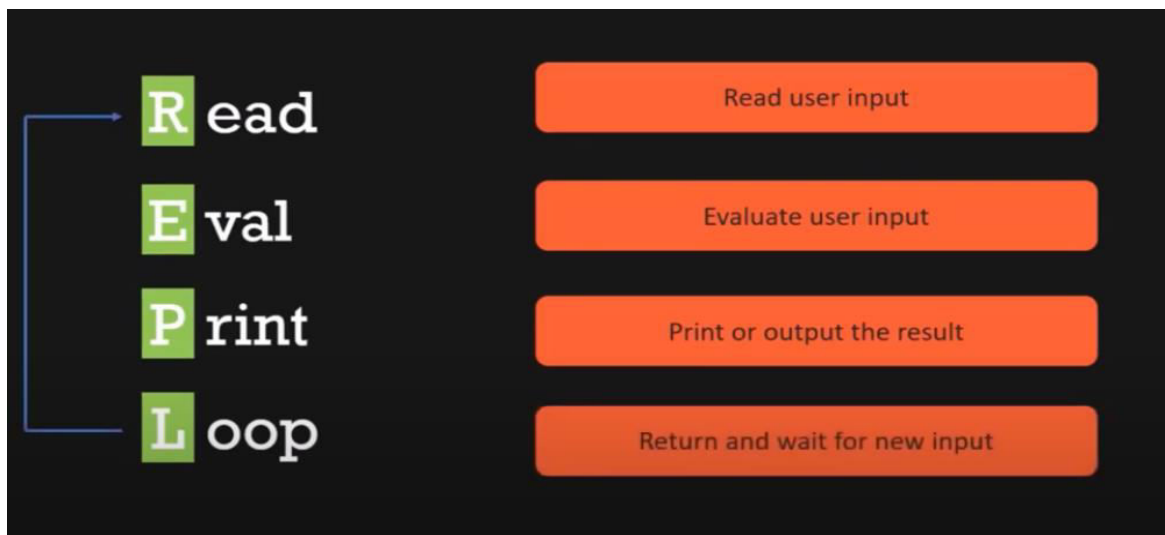


The advantages of using node.js:-

NODE JS IS SINGLE THREADED, EVENT DRIVEN & NON-BLOCKING	SAME TECHNOLOGY ACROSS ENTIRE STACK
PERFECT FOR BUILDING FAST & SCALABLE, DATA INTENSIVE APPLICATION	HUGE LIBRARY OF OPEN SOURCE PACKAGES ARE AVAILABLE
MANY TOP COMPANIES LIKE NETFLIX, UBER, PAYPAL, EBAY USES NODE JS	THERE IS A HUGE ACTIVE COMMUNITY OF NODE JS DEVELOPERS

REPL in NODE.JS:-

We can run node.js in the command prompt using REPL.



REPL is an environment where we can run our JavaScript code.

INPUT AND OUTPUT IN NODE.JS TERMINAL:-

The syntax for taking and printing input and output respectively in node.js.

```
pp.js > node (close) callback
const readline = require('readline');
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question("Please enter your name: ", (name) => {
  console.log("You entered: " + name);
  rl.close();
})

rl.on('close', () => {
  console.log("Interface closed");
  process.exit(0);
})
```

READING AND WRITING FILES USING NODE.JS (SYNCHRONOUSLY):-

To read and write files, first, we need to import the file system module.

```
const fs = require('fs');  
  
let textIn = fs.readFileSync('./Files/input.txt', 'utf-8');  
console.log(textIn)
```

Here, this `readFileSync` function is used to read files synchronously and it has two arguments that need to be provided, 1st is the path of the file that needs to be studied. And in 2nd argument we provide the encoding.

```
28 let content = `Data read from input.txt: ${textIn}. \nDate created ${new Date()}`  
29 fs.writeFileSync('./Files/output.txt', content);
```

Here, `writeFileSync` is used to write content inside the file, It has 2 arguments, in 1st argument we have to provide the path of the file in which we are going to provide the content, and in the 2nd argument we have to provide the content that we have to attach.

```
Files > output.txt  
1 Data read from input.txt: This is a sample text file which we are going to read using NODE JS.  
2 Date created Sat Sep 03 2022 12:41:34 GMT+0530 (India Standard Time)
```

NOTE:- if the provided path does not have any file then, the writeFileSync function will create a file and then add content.

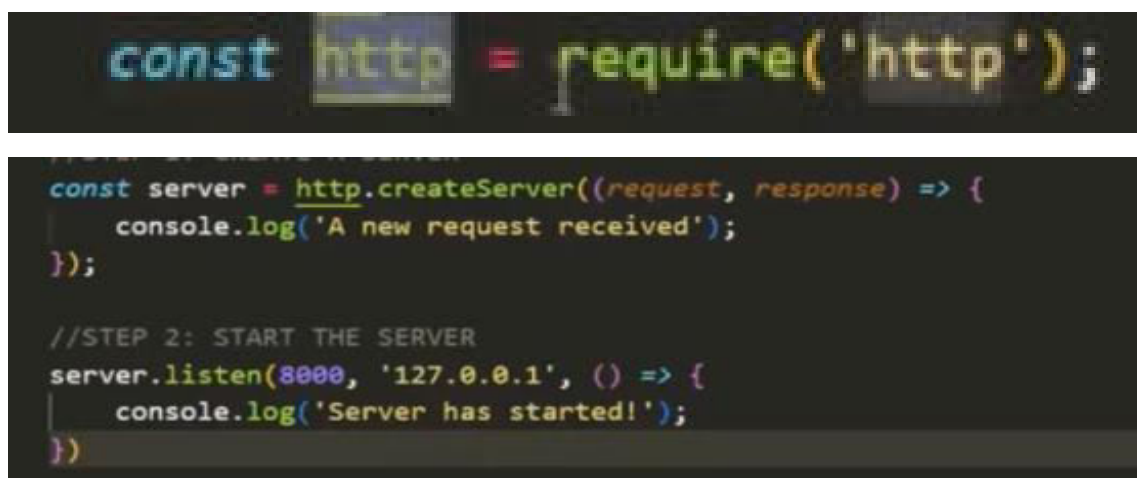
READING AND WRITING FILES USING NODE.JS (SYNCHRONOUSLY):-



Here, the readFile function works asynchronously (means that another line can work, line 1 code does not block the code of line 2), and here in the picture, another method of printing data is mentioned.

CREATING A WEB SERVER:-

To create a server we need to import the http module.



Here createServer function is responsible for creating a new server, and It has a callback function that has two arguments request and response.

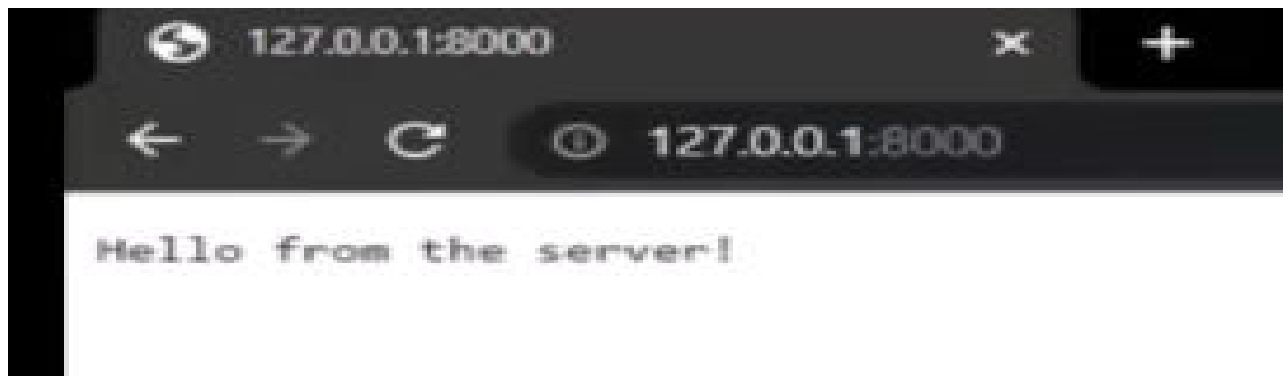
The request contains every detail about the server.

The response also has some methods and details stored.

Response.end() is used to provide data of anything to the user when he/she comes to their server:-

```
const server = http.createServer((request, response) => {  
  response.end('Hello from the server!');  
  console.log('A new request received');  
  //console.log(response);  
});  
  
//STEP 2: START THE SERVER  
server.listen(8000, '127.0.0.1', () => {  
  console.log('Server has started!');  
});
```

Se we get to see this:-



Now, server.listen functions take 3 arguments, 1st takes the PORT no. where we want to start our server, the next argument takes the hostname, by default it is the hostname, and the third argument is the callback function.

DEMO CODE FOR CREATING A WEB SERVER:-

```
const http = require('http');
const server = http.createServer((request, response) => {
  // Handle the incoming HTTP request

  // The 'request' object represents the incoming request,
  containing information about the client's request, such as
  headers, method, URL, etc.

  // The 'response' object is used to send the HTTP
  response back to the client. You can set headers, write the
  response body, and end the response using this object.

  response.writeHead(200, { 'Content-Type': 'text/plain'
});
  response.write('Hello, World!');
  response.end();
});

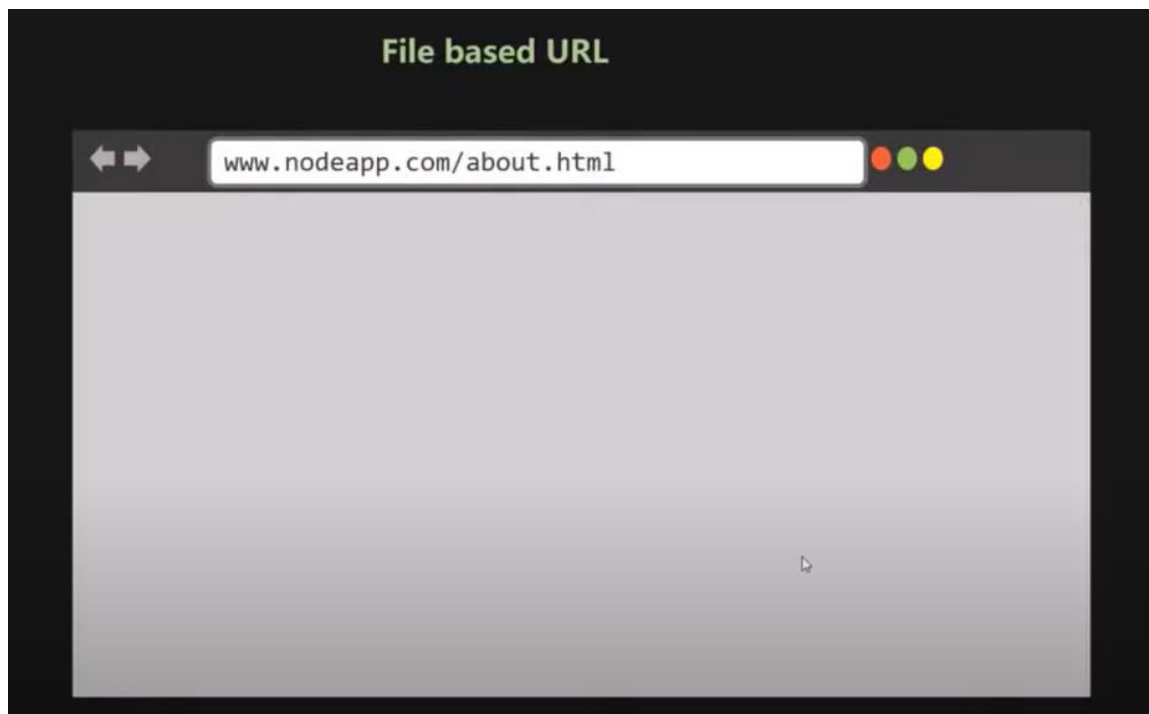
const PORT = 3000;
server.listen(PORT, () => {
  console.log(`Server listening on port ${PORT}`);
});
```

ROUTING:-

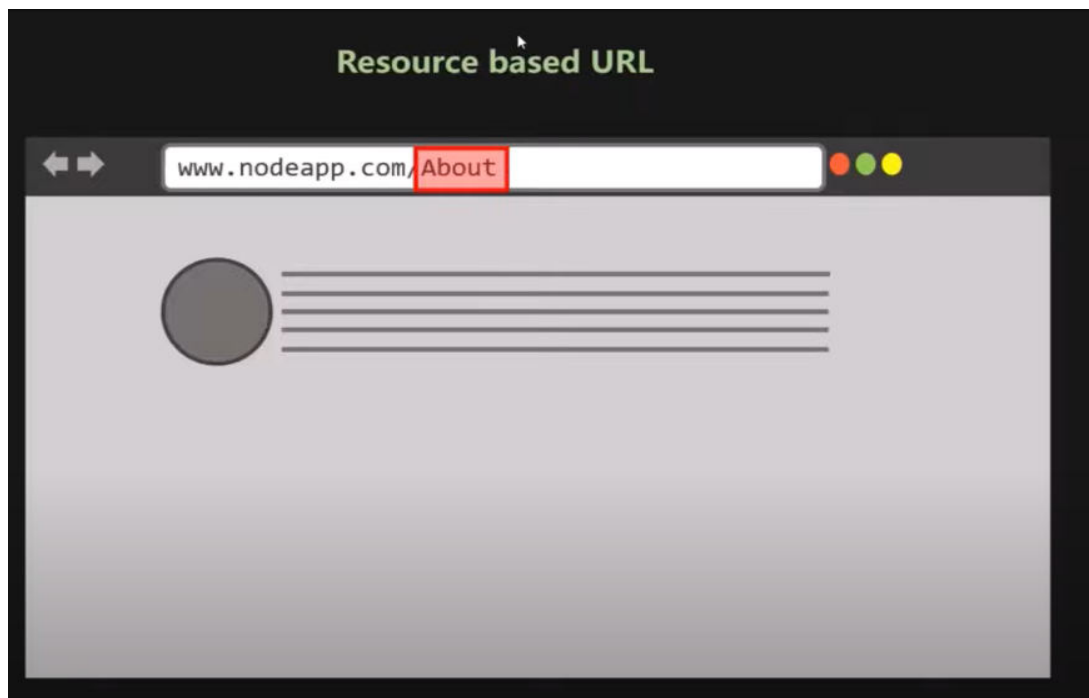
Routing defines the way in which the client requests are handled by the application endpoints

There are two types of [URL:-](#)

1) File-based [URL:-](#) search on the basis of the name of the file and render the page to the browser.



2) Resource-based [URL:-](#) Search on the server is based on the direct name and renders the page on the browser.

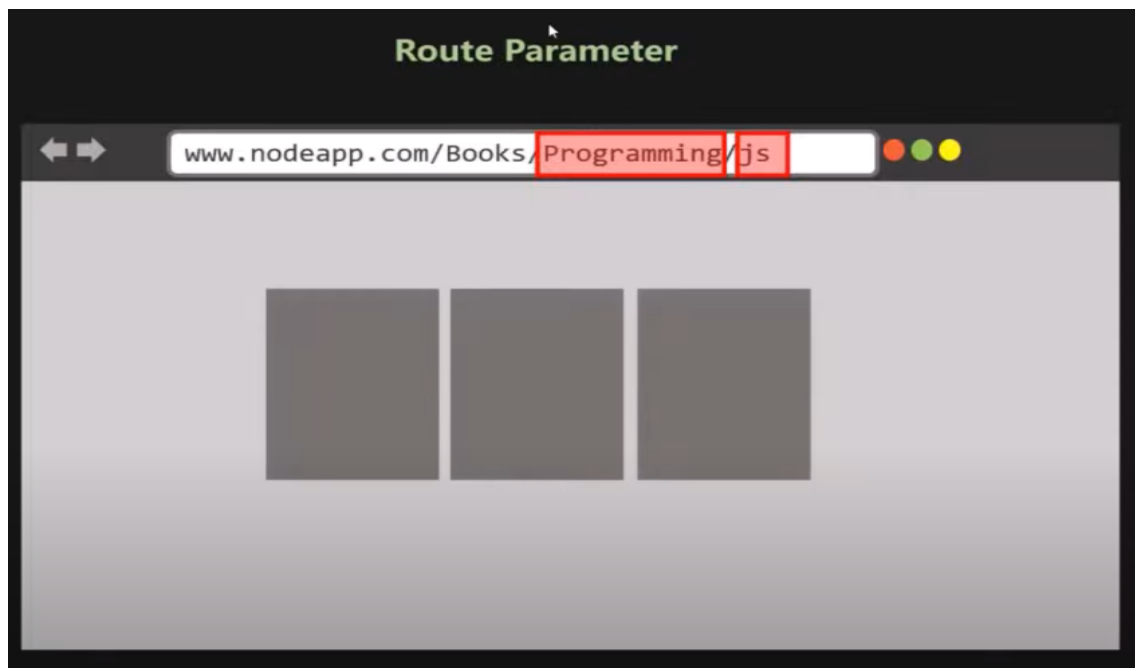


We can make our application to respond to different URLs with different responses using Routing.

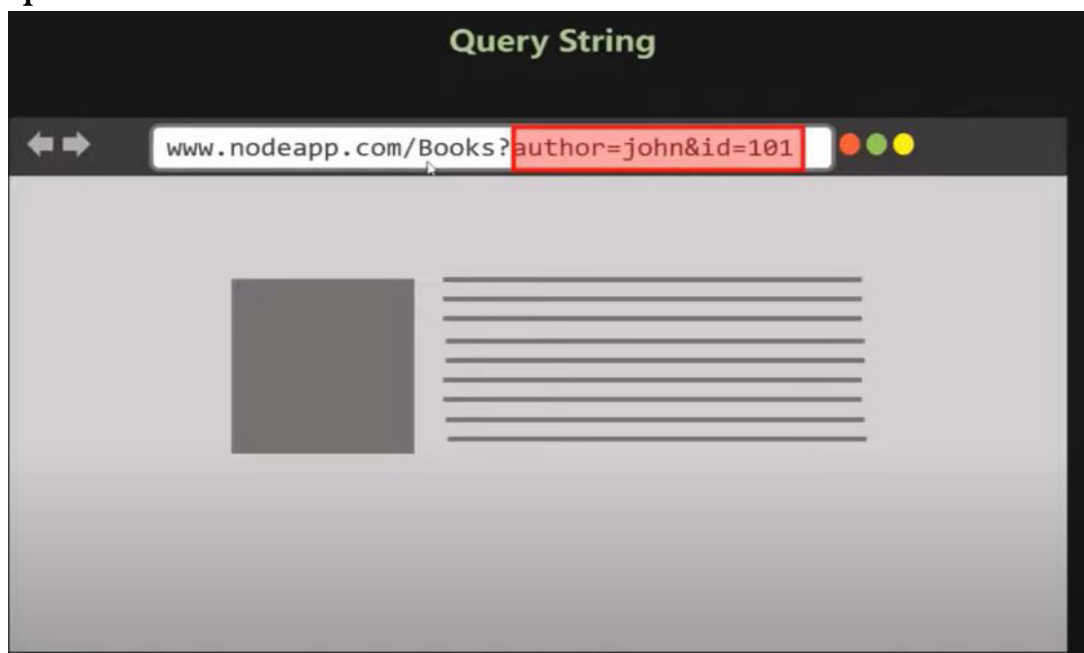
Routing basically means implementing different actions for different URLs.

These actions can be implemented in different ways, for example, by creating a function

Route can also take parameter:-



Route also has query string:- It starts after the question mark.



CREATING A ROUTE :- As we know, a request contains many objects and one of the objects is URL.

```

56 const html = fs.readFileSync('./Template/index.html', 'utf-8')
57 //STEP 1: CREATE A SERVER
58 const server = http.createServer((request, response) => {
59   let path = request.url;
60
61   if(path === '/' || path.toLocaleLowerCase() === '/home'){
62     response.end('You are in home page');
63   } else if(path.toLocaleLowerCase() === '/about'){
64     response.end('You are in about page');
65   } else if(path.toLocaleLowerCase() === '/contact'){
66     response.end('You are in contact page');
67   } else {
68     response.end('Error 404: Page');
69   }
70 });
71
72 //STEP 2: START THE SERVER
73 server.listen(8000, '127.0.0.1', () => {
74   console.log('Server has started!');
75 })

```

we have created routing.

Sending HTML Response:- In this, we have some different HTML pages and we are passing it to response.end() fn so that is, render that HTML page.

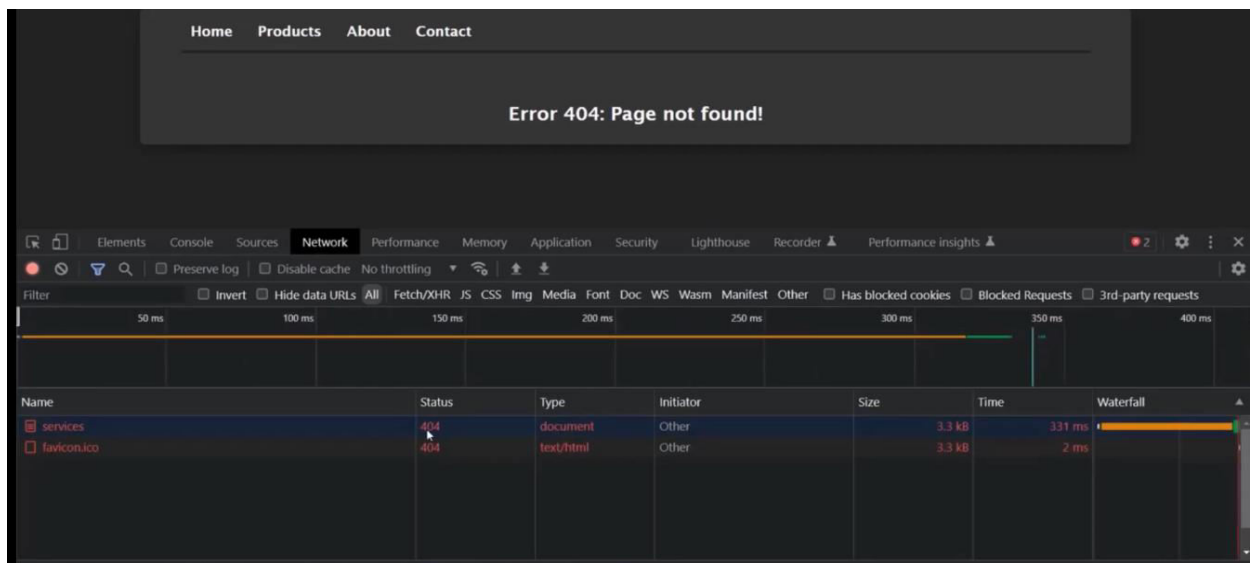
Setting Headers For Response:-

As we know 200 status code is the default status code unless it is not altered by the backend developer.

```
const html = fs.readFileSync('./Template/index.html', 'utf-8')
//STEP 1: CREATE A SERVER
const server = http.createServer((request, response) => {
  let path = request.url;

  if(path === '/' || path.toLocaleLowerCase() === '/home'){
    response.writeHead(200);
    response.end(html.replace('{{%CONTENT%}}', 'You are in Home page'));
  } else if(path.toLocaleLowerCase() === '/about'){
    response.writeHead(200);
    response.end(html.replace('{{%CONTENT%}}', 'You are in About page'));
  } else if(path.toLocaleLowerCase() === '/contact'){
    response.writeHead(200);
    response.end(html.replace('{{%CONTENT%}}', 'You are in Contact page'));
  } else {
    response.writeHead(404);
    response.end(html.replace('{{%CONTENT%}}', 'Error 404: Page not found!'));
  }
});
```

As you can see we have mentioned the status code for every particular route by using writeHead fn() and it should always be written before response.end() function.



But, to define the headers of a response, we need a 2nd argument in a writehead fn(), I guess 1st argument is for The status code and 2nd argument are for headers.

```
response.writeHead(200, {
  'Content-Type': 'text/html',
  'my-header': 'Hellow, world'
});
```

Here 200 is the status code and the next argument represents headers, where the content type tells the client that the response he/she is getting is a text/HTML file or page and my-header is a custom header created by us.

Working with JSON data:-

JSON is a javascript object notation.

We have created a page products.json where we have stored the json data.

```
products.json > (1) 0 > 4+ id
[
  {
    "id": 0,
    "name": "APPLE iPhone SE",
    "color": "Black",
    "ROM": 128,
    "price": 990,
    "modelName": "iPhone SE",
    "modelNumber": "MHGT3HN/A",
    "size": "11.94 cm (4.7 inch) Retina HD Display",
    "camera": "12MP Rear Camera | 7MP Front Camera",
    "Description": "Widescreen HD LCD Retina Multi-touch IPS Display (1400:1 Contrast Ratio (Typical), True Tone Display",
    "productImage": "http://atlas-content-cdn.pixelsquid.com/stock-images/iphone-x-smartphone-xwVXQLD-600.jpg"
  },
  {
    "id": 1,
    "name": "APPLE iPhone XR",
    "color": "White",
    "ROM": 64,
    "price": 790,
    "modelName": "iPhone XR",
    "modelNumber": "MH6N3HN/A",
    "size": "15.49 cm (6.1 inch) Display",
    "camera": "12MP Rear Camera | 7MP Front Camera",
    "Description": "1400:1 Contrast Ratio (Typical), True Tone Display (Six-channel Light Sensor), Wide Colour Display (F
```

And we have coded to get the data .

```

    } else if(path.toLocaleLowerCase() === '/products'){
        response.writeHead(200, {
            'Content-Type': 'application/json'
        });
        fs.readFile('./Data/products.json', 'utf-8', (error, data) => {
            response.end(data);
        })
    } else {

```

And now It looks like this:-

```

[
  {
    "id": 0,
    "name": "APPLE iPhone SE",
    "color": "black",
    "ROM": 128,
    "price": 990,
    "modelName": "iPhone SE",
    "modelNumber": "MHGT3HN/A",
    "size": "11.94 cm (4.7 inch) Retina HD Display",
    "camera": "12MP Rear Camera | 7MP Front Camera",
    "Description": "Widescreen HD LCD Retina Multi-touch IPS Display (1400:1 Contrast Ratio (Typical), True Tone Display, Wide Color Display (P3), Haptic Touch, 625 nits Max Brightness (Typical), Fingerprint-resistant Oleophobic Coating, Display Zoom, Reachability)",
    "productImage": "http://atlas-content-cdn.pixelsquid.com/stock-images/iphone-x-smartphone-xwVXQLD-600.jpg"
  },
  {
    "id": 1,
    "name": "APPLE iPhone XR",
    "color": "White",
    "ROM": 64,
    "price": 790,
    "modelName": "iPhone XR",
    "modelNumber": "MH6N3HN/A",
    "size": "15.49 cm (6.1 inch) Display",
    "camera": "12MP Rear Camera | 7MP Front Camera",
    "Description": "1400:1 Contrast Ratio (Typical), True Tone Display (Six-channel Light Sensor), Wide Colour Display (P3), 625 nits Maximum Brightness (Typical), Fingerprint-resistant Oleophobic Coating, Support for Display of Multiple Languages and Characters Simultaneously, Liquid Retina HD Display, Tap to Wake, Wide Colour Gamut",
    "productImage": "https://c8.alamy.com/zooms/6/e98284ded5444c08949d7fd9f2bae166/2cd68c5.jpg"
  },
  {
    "id": 2,
    "name": "APPLE iPhone 11",
    "color": "White"
  }
]

```

Now, we are converting this JSON data into a javascript object, and for that `JSON.parse()` method is used, where we provide data in the parse function that we want to convert.

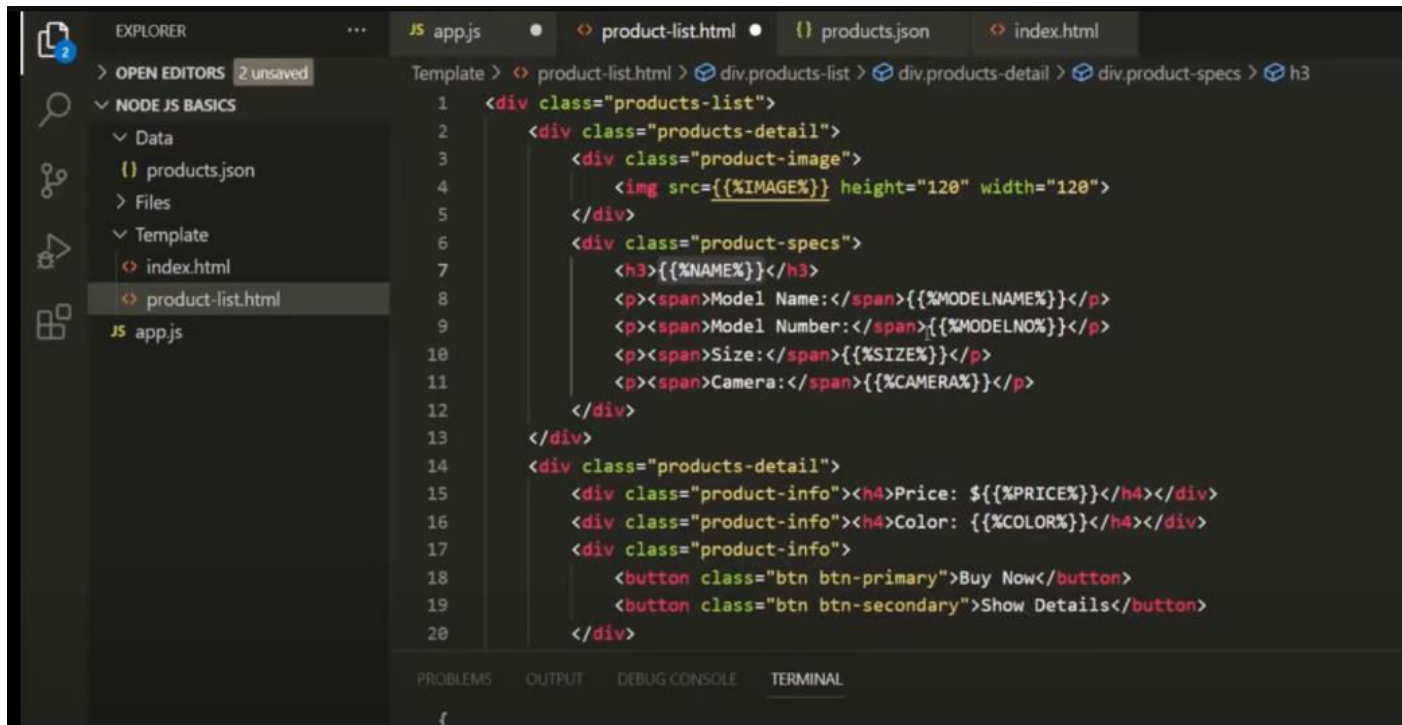
```

    fs.readFile('./Data/products.json', 'utf-8', (error, data) => {
        let products = JSON.parse(data);
        response.end(data);
    })
} else {

```

Transforming JSON data into HTML:-

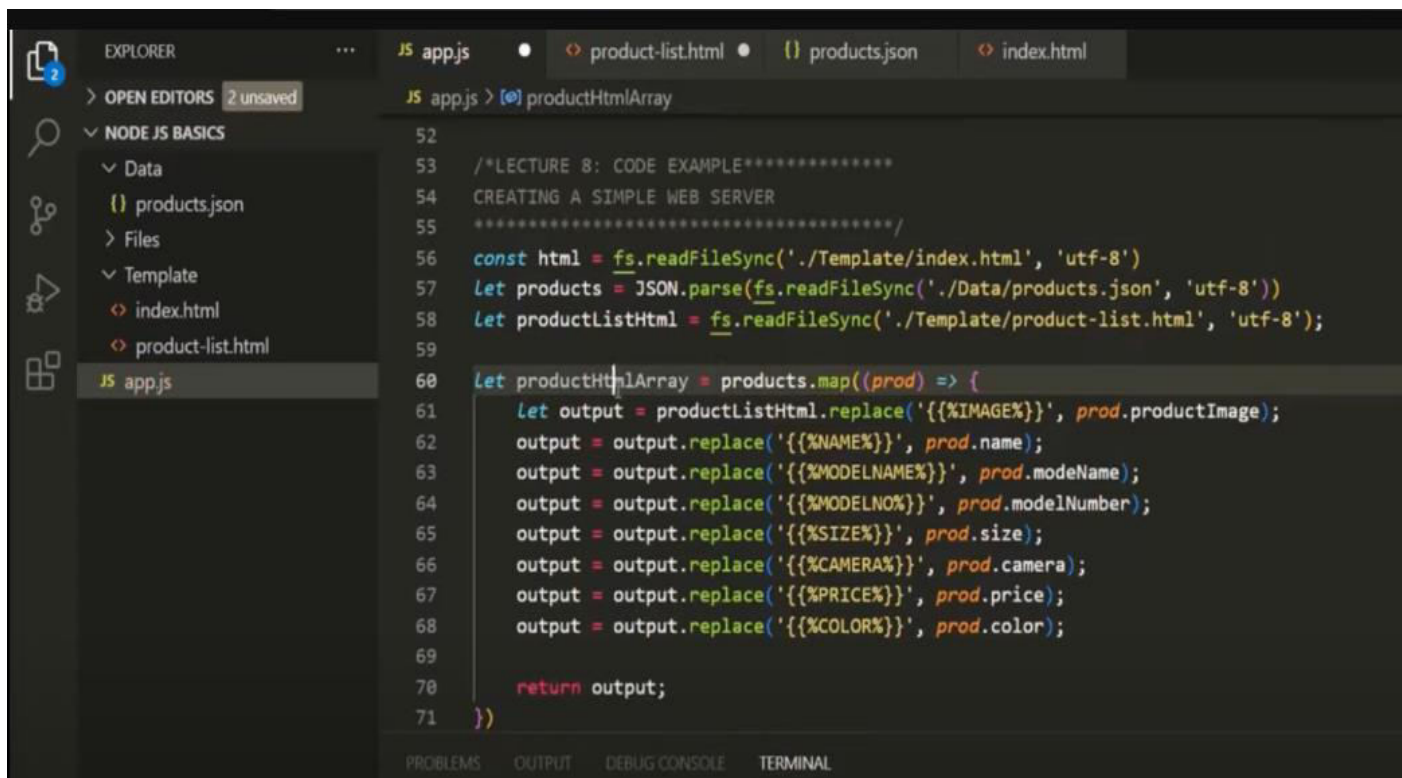
First, we have our HTML page:-



The screenshot shows the VS Code editor with the file explorer on the left and the editor window on the right. The file explorer shows the project structure: NODE JS BASICS, Data, products.json, Files, Template, index.html, product-list.html, and JS app.js. The editor window shows the HTML template for product-list.html, which is a template for a product list. The template includes a header, a list of products, and a detail view for each product. The detail view includes a product image, product name, model name, model number, size, camera, price, and color. The template uses placeholders like {{NAME}}, {{MODELNAME}}, {{MODELNO}}, {{SIZE}}, {{CAMERAX}}, {{PRICE}}, and {{COLOR}} for dynamic content.

```
1 <div class="products-list">
2   <div class="products-detail">
3     <div class="product-image">
4       
5     </div>
6     <div class="product-specs">
7       <h3>{{NAME%}}</h3>
8       <p><span>Model Name:</span>{{MODELNAME%}}</p>
9       <p><span>Model Number:</span>{{MODELNO%}}</p>
10      <p><span>Size:</span>{{SIZE%}}</p>
11      <p><span>Camera:</span>{{CAMERAX%}}</p>
12    </div>
13  </div>
14  <div class="products-detail">
15    <div class="product-info"><h4>Price: ${{PRICE%}}</h4></div>
16    <div class="product-info"><h4>Color: {{COLOR%}}</h4></div>
17    <div class="product-info">
18      <button class="btn btn-primary">Buy Now</button>
19      <button class="btn btn-secondary">Show Details</button>
20    </div>
```

Now we have to assign the values according to the JSON data.



The screenshot shows the VS Code editor with the file explorer on the left and the editor window on the right. The file explorer shows the project structure: NODE JS BASICS, Data, products.json, Files, Template, index.html, product-list.html, and JS app.js. The editor window shows the JavaScript code in app.js, which reads the JSON data from products.json and assigns values to the HTML template. The code uses fs.readFileSync to read the files, JSON.parse to parse the JSON data, and productHtmlArray.map to iterate over the products and assign values to the HTML template. The code uses replace to replace the placeholders in the HTML template with the actual values from the JSON data.

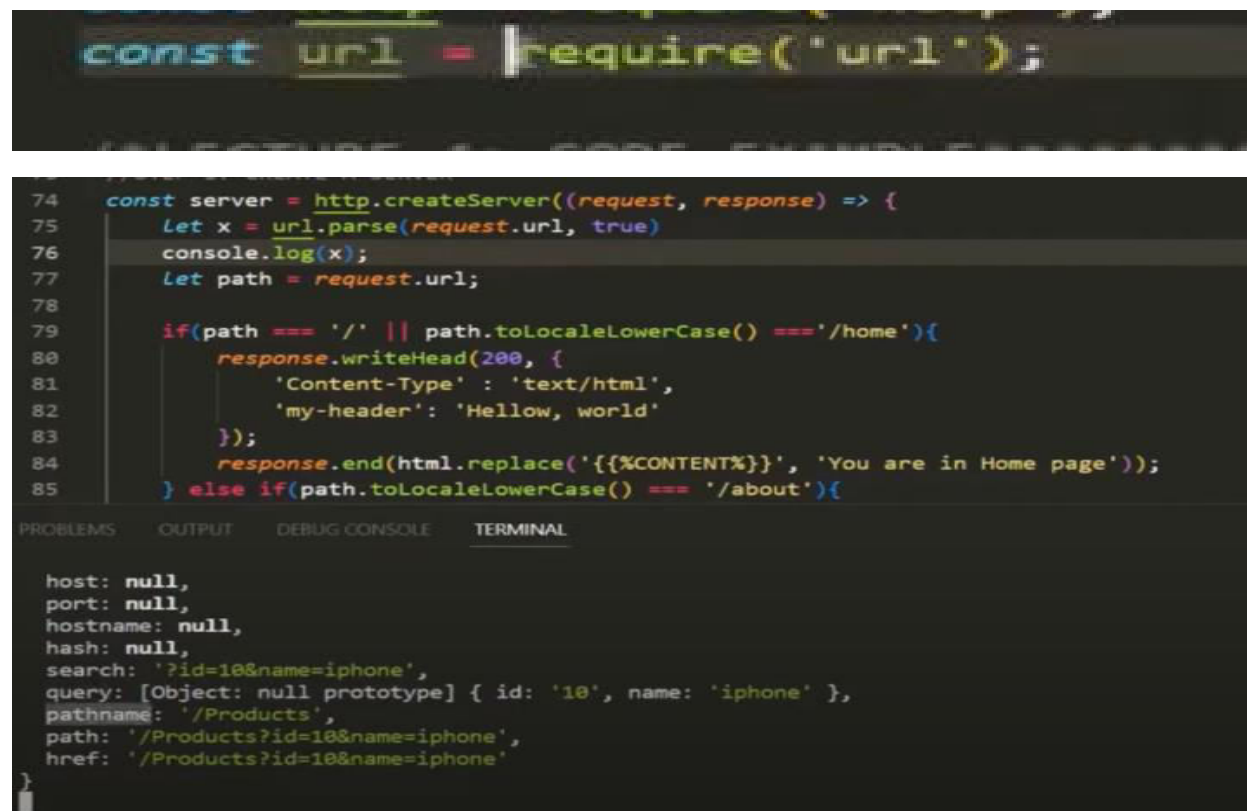
```
52
53 /*LECTURE 8: CODE EXAMPLE*****
54 CREATING A SIMPLE WEB SERVER
55 *****/
56 const html = fs.readFileSync('./Template/index.html', 'utf-8')
57 let products = JSON.parse(fs.readFileSync('./Data/products.json', 'utf-8'))
58 let productListHtml = fs.readFileSync('./Template/product-list.html', 'utf-8');
59
60 let productHtmlArray = products.map((prod) => {
61   let output = productListHtml.replace('{{IMAGE%}}', prod.productImage);
62   output = output.replace('{{NAME%}}', prod.name);
63   output = output.replace('{{MODELNAME%}}', prod.modelName);
64   output = output.replace('{{MODELNO%}}', prod.modelNumber);
65   output = output.replace('{{SIZE%}}', prod.size);
66   output = output.replace('{{CAMERAX%}}', prod.camera);
67   output = output.replace('{{PRICE%}}', prod.price);
68   output = output.replace('{{COLOR%}}', prod.color);
69
70   return output;
71 })
```


This `{{%%name%%}}` is used as a placeholder, to replace the content inside it.

And now this new `producthtmlarray` is used to show the output.

Parsing query string from URL:-

First, we need to import url module/package.



```
const url = require('url');

74 const server = http.createServer((request, response) => {
75   let x = url.parse(request.url, true)
76   console.log(x);
77   let path = request.url;
78
79   if(path === '/' || path.toLocaleLowerCase() === '/home'){
80     response.writeHead(200, {
81       'Content-Type': 'text/html',
82       'my-header': 'Hellow, world'
83     });
84     response.end(html.replace('{{%CONTENT%}}', 'You are in Home page'));
85   } else if(path.toLocaleLowerCase() === '/about'){
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
host: null,
port: null,
hostname: null,
hash: null,
search: '?id=10&name=iphone',
query: [Object: null prototype] { id: '10', name: 'iphone' },
pathname: '/Products',
path: '/Products?id=10&name=iphone',
href: '/Products?id=10&name=iphone'
}
```

Here in `url.parse fn()` 1st argument will be the URL and 2nd argument will tell is Boolean and it determines whether the parse function will work or not.

As you can see in this picture, in terminal, there is a mention of query and pathname.

If we have to take these two particular values then we can do this:-

```

4  const server = http.createServer((request, response) => {
5      let {query, pathname} = url.parse(request.url, true)
6      console.log(x);
7      ⚡ let path = request.url;
8

```

And this:-

```

//STEP 1: CREATE A SERVER
const server = http.createServer((request, response) => {
  ⚡ let {query, pathname: path} = url.parse(request.url, true)
  console.log(x);
  //let path = request.url;

```

Here the path will store the resource (like HOME, About) name

.

Creating a Custom Module:-

Each js file in node.js is a module.

There are different types of modules:-

1. CORE Modules:-

```

//CORE MODULES
const readline = require('readline');
const fs = require('fs');
const http = require('http');
const url = require('url');

```

2. User-defined/Custom Modules:- we create our own js file which is termed as a module:-

```

1 module.exports = function(template, product){
2   let output = template.replace('{{%IMAGE%}}', product.productImage);
3   output = output.replace('{{%NAME%}}', product.name);
4   output = output.replace('{{%MODELNAME%}}', product.modelName);
5   output = output.replace('{{%MODELNO%}}', product.modelNumber);
6   output = output.replace('{{%SIZE%}}', product.size);
7   output = output.replace('{{%CAMERA%}}', product.camera);
8   output = output.replace('{{%PRICE%}}', product.price);
9   output = output.replace('{{%COLOR%}}', product.color);
10  output = output.replace('{{%ID%}}', product.id);
11  output = output.replace('{{%ROM%}}', product.ROM);
12  output = output.replace('{{%DESC%}}', product.Description);
13
14  return output;
15 }

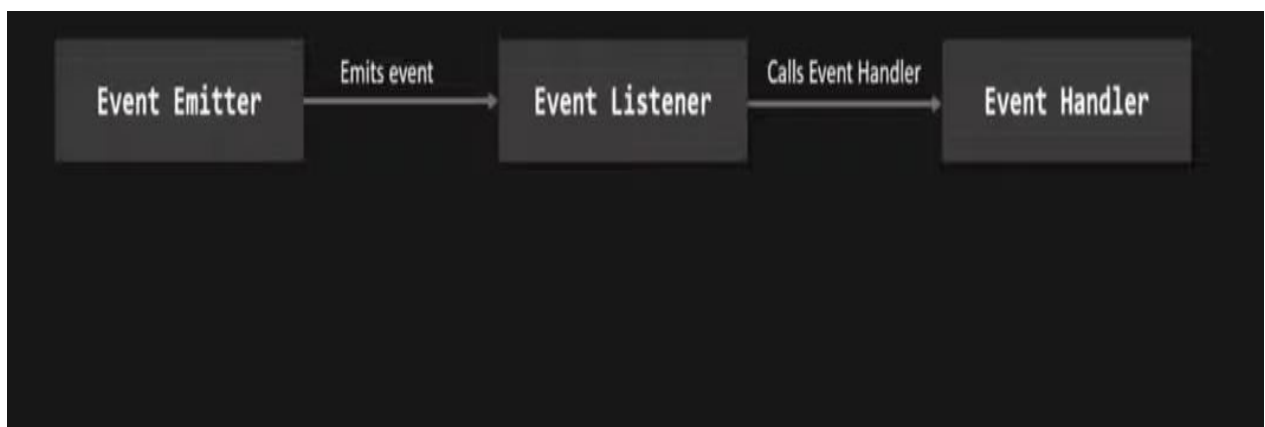
```

Created our module, and this module needs to be imported, where is it to be used.

EVENT driven Architecture:-

It has 3 main players:-

1. Event Emitter:- emits the event.
2. Event Listener:- receives or listens to whatever event is emitted and fires the callback function which was attached to the event Listener when that event happens.
3. Event Handler:- The call back function is termed Event Handler. It reacts to the event.



Let's say we have created a server:-

```
const server = http.createServer();

server.listen(8000, '127.0.0.1', () => {
  console.log('listening to requests...')
})

server.on('request', (req, res) => {
  res.end('Hello from the server!');
})
```

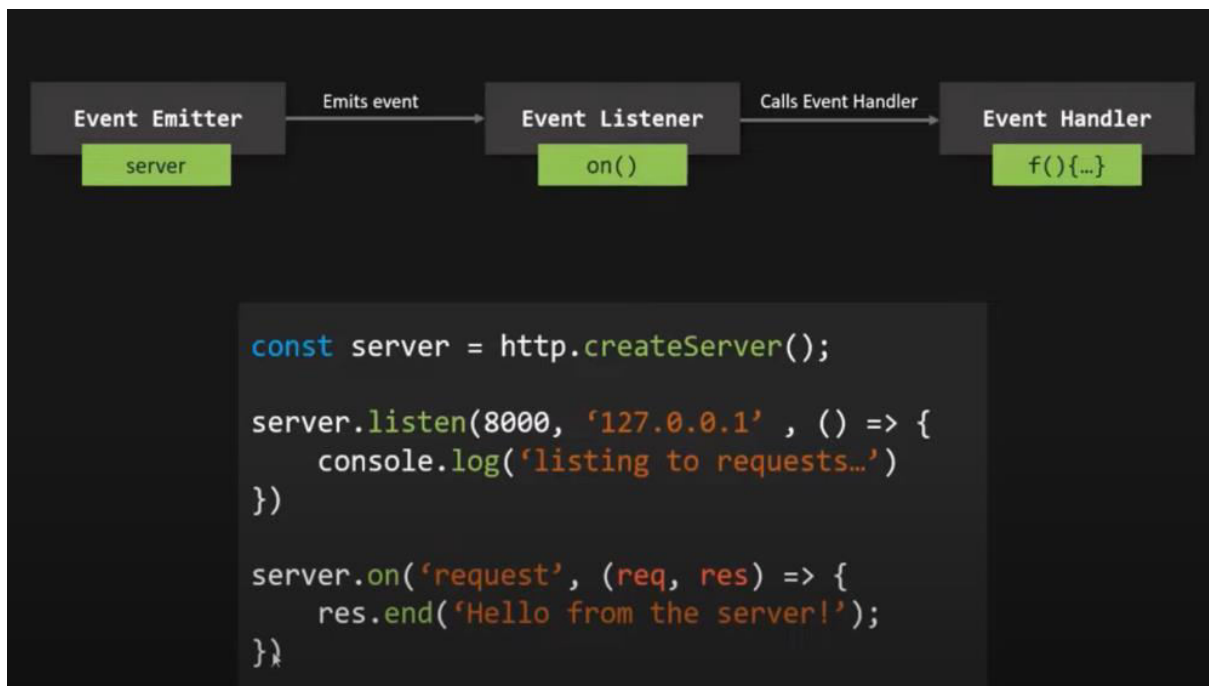
Whenever a request hits the server then the server will emit the named event called REQUEST.

So REQUEST is the named event and the server is the event emitter.

So when this event is emitted then is listened to by (.on())method.

1st parameter in .on() method is the event that we want to listen.

And 2nd parameter is the callback function.



Notice one thing this method is different from we used earlier to create a server.

Here we have not passed any arguments to the `http.createServer()` function.

There is one syntax:-

```
Const server=http.createServer((request,response)=>{
// write your code.
});
```

And with same functionalities with another syntax;

```
Const server=http.server();
Server.on('request/event name',(request,response)=>{
});
```

And `server.listen()` function will be written in both the syntax.

Emitting and Handling Custom Events:-

1st we have to import the events module.

```
6 const events = require('events');  
  
// .....  
Let myEmitter = new events.EventEmitter();
```

myEmitter will store the instance of EventEmitter class, and by using this we can raise custom events.

```
myEmitter.emit('userCreate');
```

The above line of code will emit an event named userCreate.

By using .on() method, we can listen to our custom event.

```
Let myEmitter = new events.EventEmitter();  
// .....  
myEmitter.on('userCreated', () => {  
  console.log('A new user is created!')  
})  
  
myEmitter.emit('userCreated');
```

We can send some parameters also:-

```
Let myEmitter = new events.EventEmitter();  
  
myEmitter.on('userCreated', (id, name) => {  
  console.log(`A new user ${name} with ID ${id} is created!`)  
})  
  
myEmitter.on('userCreated', (id, name) => {  
  console.log(`A new user ${name} with ID ${id} is added to database!`)  
})  
  
myEmitter.emit('userCreated', 101, 'John');
```


Streams in Node.Js:-

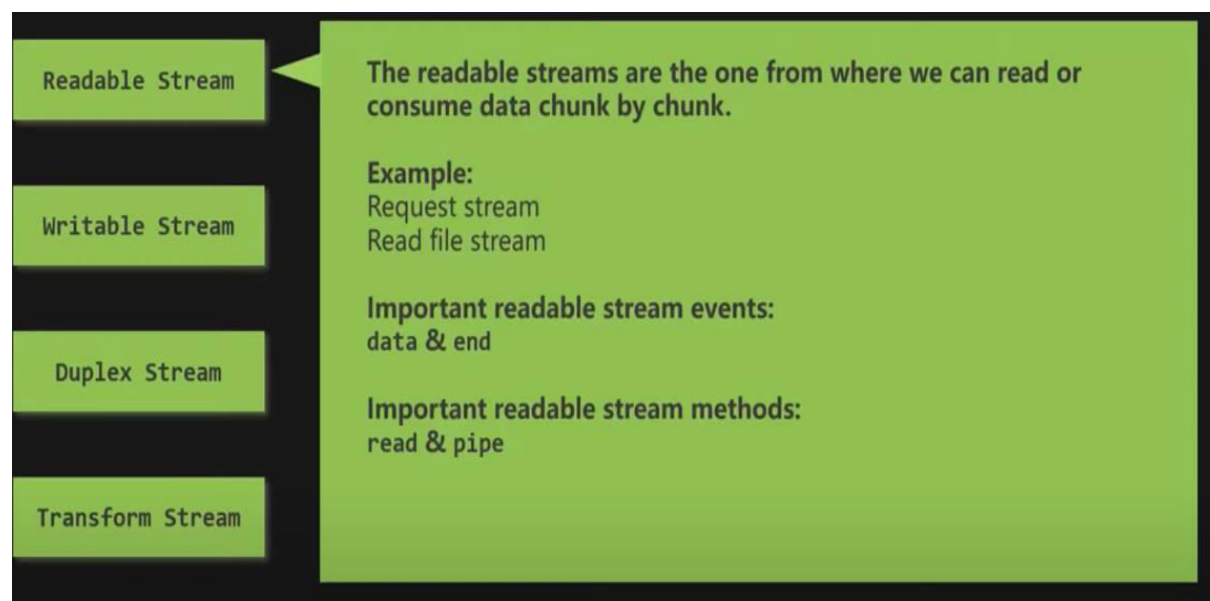
With Streams, we can process data piece by piece instead of reading or writing the whole data at once.

Advantage

Streaming makes the data processing more efficient in terms of memory. Because there is no need to keep all the data in the memory.

In terms of performance & time also, streaming has its advantage because we can start processing the data as soon as the first chunk of data arrives.

There are 4 types of streams in node.js:-



This diagram illustrates the relationship between different types of streams. On the left, a vertical stack of four light blue boxes contains the labels: 'Readable Stream', 'Writable Stream', 'Duplex Stream', and 'Transform Stream'. A yellow arrow points from the 'Writable Stream' box to a large light blue box on the right. This box contains the following text:

The writable streams are the one to which we can write data chunk by chunk. It's the opposite of readable stream

Example:
Response stream
Write file stream

Important readable stream events:
drain & finish

Important readable stream methods:
write & end

This diagram focuses on Duplex Streams. On the left, a vertical stack of four light blue boxes contains the labels: 'Readable Stream', 'Writable Stream', 'Duplex Stream', and 'Transform Stream'. A yellow arrow points from the 'Duplex Stream' box to a large light blue box on the right. This box contains the following text:

Duplex stream is simply a stream that is both readable & writable at the same time.

Example:
Web Sockets

This diagram focuses on Transform Streams. On the left, a vertical stack of four light blue boxes contains the labels: 'Readable Stream', 'Writable Stream', 'Duplex Stream', and 'Transform Stream'. A yellow arrow points from the 'Transform Stream' box to a large light blue box on the right. This box contains the following text:

Transform streams are duplex streams which can also modify or transform data as it is read or written.

Example:
zlib

NPM [Node Package Manager]:-

It is a command line interface for managing packages and also a repository from where we install packages.

It automatically installs when we install the node.js.

To use the command line interface of NPM we can use our command prompt or vs code terminal.

Whenever we write npm init in our terminal, a package.json file is created which stores all information about the project that we are making (like dependencies and all);

Types of packages/dependencies:-

Types of dependencies	
Regular Dependencies	A package is called as a simple or regular dependency if the working of our application or the code which we are writing, depends on that package.
Development Dependencies	A package is called as development dependency, if that package is only required for the development purpose and on which, the working of our application does not depend.

Ex for regular dependencies is express.

Ex for development dependencies is nodemon.

Skipped Node.Js architecture

2) EXPRESS.JS

Express JS is a free and open-source web application framework for NODE JS.

Express is used to shorten the length of node.js code.

EXPRESS IS COMPLETELY BUILD ON NODE JS

IT IS ONE OF THE MOST POPULAR
FRAMEWORK FOR NODE JS

EXPRESS CONTAINS VERY ROBUST AND
USEFUL SET OF FEATURES

EXPRESS ALLOWS TO WRITE NODE JS
APPLICATION FASTER & SIMPLER

WITH EXPRESS WE CAN ORGANIZE NODE JS
CODE IN MVC ARCHITECTURE

```
//IMPORT PACKAGE  
const express = require('express');
```

This require('express') will return a function that will be stored in express;

```
Let app = express();
```

Here we have called express fn and, all the objects are stored in a variable named app.

Creating a server:-

```
const express = require('express');
let app = express();

//ROUTE = HTTP METHOD + URL
app.get('/', (req, res) => {
  res.status(200).send('<h4>Hello from express server</h4>');
})

//CREATE A SERVER
const port = 3000;
app.listen(port, () => {
  console.log('server has started...');
})
```

The app.listen() method here is used to start the server, and the parameters are the same as those of node.js.

Here, if want to tackle a get request at the '/' URL then we have the app.get() method, and here res.send() signifies that the content type that we are sending is of text/HTML type.

But to send json response we can do this:-

```
app.get('/', (req, res) => {
  res.status(200).json({message: 'Hello, world', status: 200});
})
```

Web API:-

Static Website:-

A static website is a type of website that is comprised of fixed, unchanging files. These files are delivered to the user's web browser exactly as they are stored, without any server-side processing. In other words, the content of a static website remains the same for all users and doesn't change in response to user interactions or data from a database.

Key characteristics of static websites include:

1. **HTML, CSS, and JavaScript:** Static websites typically consist of HTML for structure, CSS for styling, and JavaScript for client-side interactivity. These files are pre-written and do not change based on user input or other external factors.
2. **Fast Loading:** Since there is no need for server-side processing or database queries, static websites can load quickly. Each page is a standalone file that is served directly to the user's browser.
3. **Hosting:** Static websites are often hosted on simple web servers or content delivery networks (CDNs). They can be hosted on platforms like GitHub Pages, Netlify, or Amazon S3.
4. **Security:** Due to their simplicity and lack of server-side processing, static websites are generally considered to be more secure than dynamic websites. There are fewer attack vectors, and the attack surface is reduced.
5. **Scalability:** Static websites are highly scalable because they can be easily distributed and cached on a global content delivery network (CDN), reducing the load on the server.
6. **Examples:** Many types of websites can be static, including personal blogs, portfolios, company landing pages, documentation sites, and more. These are websites where the content doesn't change frequently and doesn't rely on user-generated or real-time data.

While static websites have their advantages, they are not suitable for every type of project. They work well for projects with content that doesn't change often and doesn't require dynamic features. If a website needs to handle user accounts, real-time data, or complex interactions, a dynamic (server-side rendered or client-side rendered) approach may be more appropriate. The choice between a static and dynamic approach depends on the specific requirements and goals of the project.

Dynamic Website:-

A dynamic website is a type of website that generates content on the server side in response to user requests. Unlike static websites, dynamic websites can change content and functionality based on user input, interactions, and data from databases. They typically involve server-side scripting, and database operations, and may use a combination of server-side and client-side technologies.

Key characteristics of dynamic websites include:

1. **Server-Side Scripting:** Dynamic websites use server-side scripting languages (such as PHP, Python, Ruby, Node.js, etc.) to generate content dynamically on the server in response to user requests. The server processes the script,

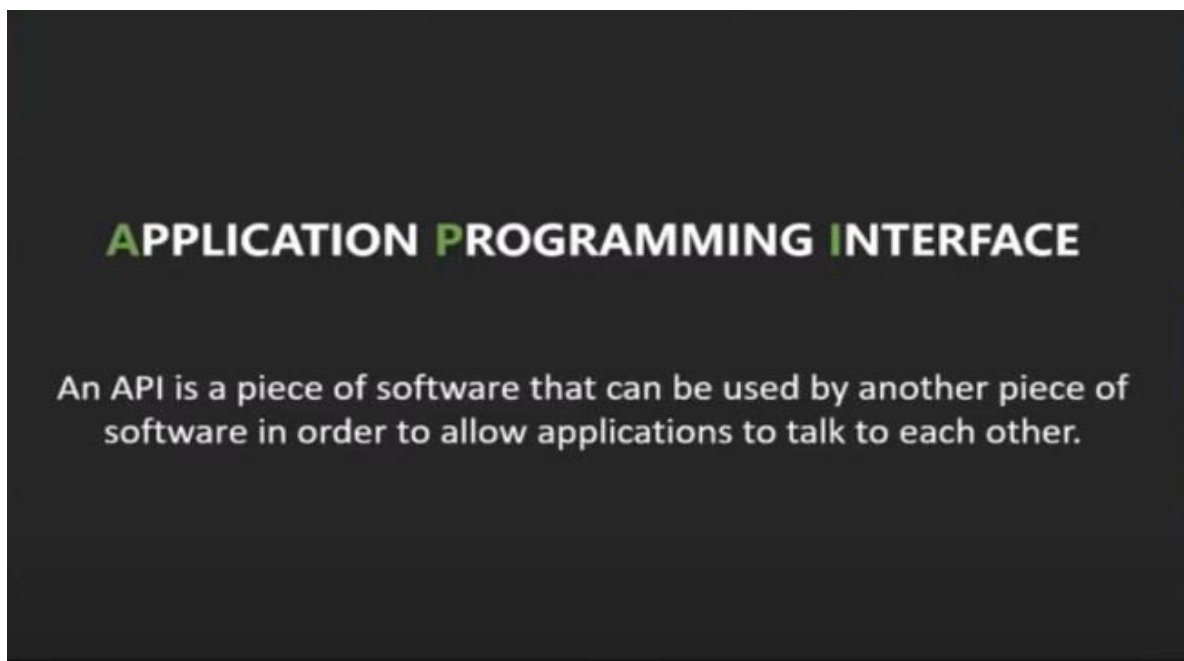
interacts with databases, and generates the HTML that is sent to the user's browser.

2. **Database Integration:** Dynamic websites often interact with databases to store, retrieve, and update data. This allows for the management of user accounts, content, and other dynamic elements.
3. **User Authentication:** Dynamic websites can implement user authentication and authorization systems. Users can log in, access personalized content, and perform actions that are specific to their account.
4. **Real-Time Interaction:** Dynamic websites can provide real-time interactions and updates without requiring the user to refresh the entire page. Technologies such as AJAX (Asynchronous JavaScript and XML) or more modern techniques like WebSockets are commonly used for this purpose.
5. **Content Management Systems (CMS):** Many dynamic websites use content management systems like WordPress, Drupal, or Joomla. These systems allow users to easily manage and update website content without extensive technical knowledge.
6. **Customization and Personalization:** Dynamic websites can personalize content based on user preferences, history, or behavior. This can result in a more engaging and tailored user experience.
7. **E-commerce:** Dynamic websites are commonly used for e-commerce platforms where users can browse products, add items to their cart, and complete transactions. Shopping carts, inventory management, and order processing are typical dynamic features.
8. **Web Frameworks:** Developers often use web frameworks (such as Django, Ruby on Rails, Laravel, Express.js, etc.) to streamline the development of dynamic websites. These frameworks provide tools and structures for building robust and maintainable web applications.

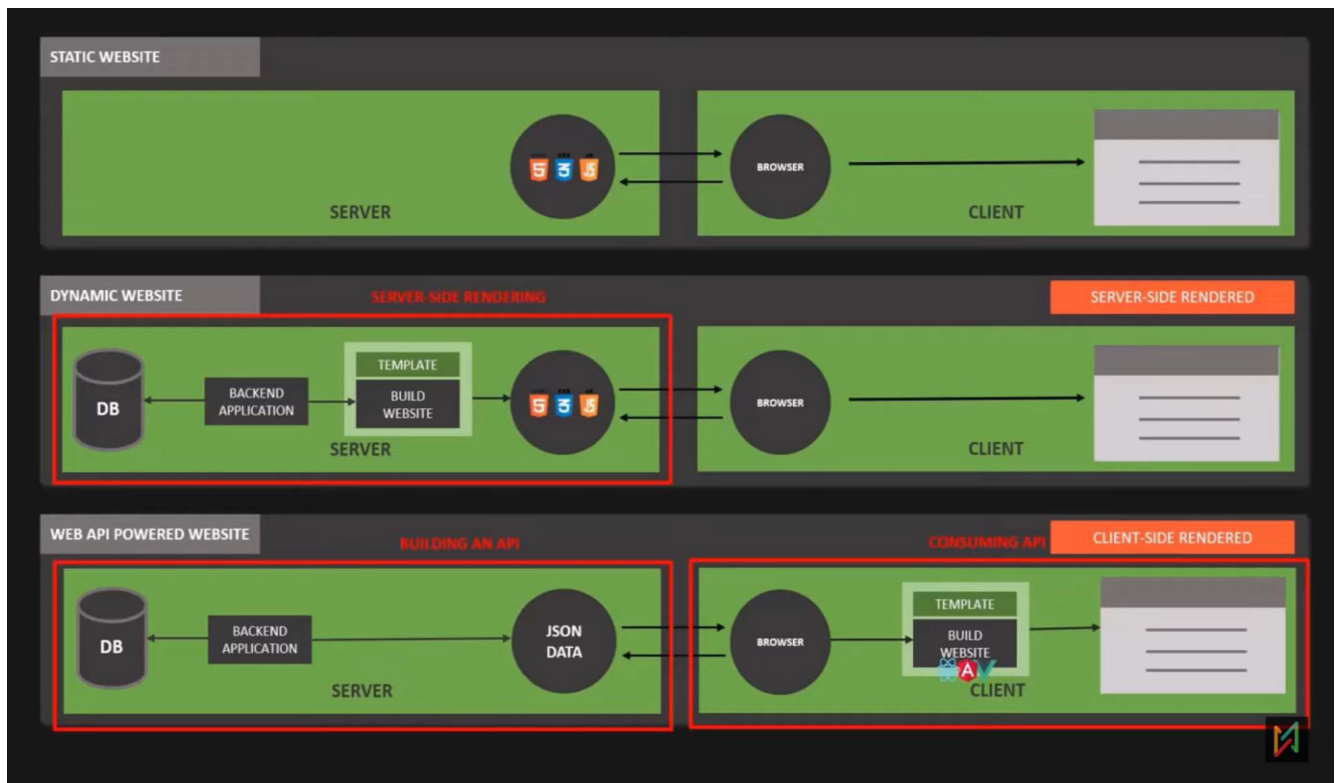
While dynamic websites offer more interactivity and flexibility, they may require more server resources, have higher development complexity, and could potentially be slower to load compared to static websites. The choice between a dynamic and static approach depends on the specific requirements of the project and the desired functionality.



API:-



API sends the JSON data to the client instead of the HTML, CSS, and JS files.



Client-side rendering (CSR) and server-side rendering (SSR) are two approaches to rendering web pages, and they involve where the rendering process takes place — on the client side (in the user's browser) or on the server side (on the web server).

Client-Side Rendering (CSR):

1. **Rendering Location:** The rendering of the web page occurs in the user's browser after the initial HTML, CSS, and JavaScript files are downloaded.
2. **JavaScript Heavy:** Typically, a substantial amount of rendering logic is implemented in JavaScript. The client's browser executes JavaScript to build the final page structure.
3. **Single Page Applications (SPAs):** Commonly associated with Single Page Applications where the initial page load is minimal, and subsequent content changes are handled by JavaScript (often using a frontend framework like React, Angular, or Vue.js).
4. **Advantages:**
 - Fast initial page load (only necessary assets are loaded).
 - Smooth transitions between pages as JavaScript handles page changes dynamically.
5. **Disadvantages:**

- Initial page load may be slower for users with slow network connections or less powerful devices.
- Limited support for SEO, as search engines may not effectively index content rendered through JavaScript.

Server-Side Rendering (SSR):

1. **Rendering Location:** The rendering of the web page occurs on the server before sending the HTML to the client's browser.
2. **HTML from Server:** The server generates HTML content based on the requested URL and sends a fully rendered page to the client.
3. **Traditional Multi-Page Applications (MPAs):** Often associated with traditional multi-page applications where each page is a separate HTML document.
4. **Advantages:**
 - Better initial page load performance, especially for users with slower network connections or less powerful devices.
 - Improved SEO, as search engines can index content directly from the server-rendered HTML.
5. **Disadvantages:**
 - Slower transitions between pages compared to SPAs because the entire page needs to be reloaded.
 - Higher server load, as the server has to render pages for each user request.

Hybrid Approaches:

In some cases, developers use hybrid approaches, combining elements of CSR and SSR to leverage the benefits of both. This is often referred to as "universal" or "isomorphic" rendering. In these approaches, certain parts of the page may be initially rendered on the server, while others are handled by client-side JavaScript.

The choice between CSR and SSR depends on factors such as the type of application, user experience goals, SEO requirements, and performance considerations. Modern frameworks and libraries often provide tools for implementing both CSR and SSR, allowing developers to choose the approach that best suits their needs.

REST Architecture:-

It is used to make web API's.

REpresentational State Transfer

REST is an architectural style for providing standards between computer systems on the web, making it easier for systems to communicate with each other.

Principles to follow for creating REST APIs:-

1

Separate APIs into logical resources

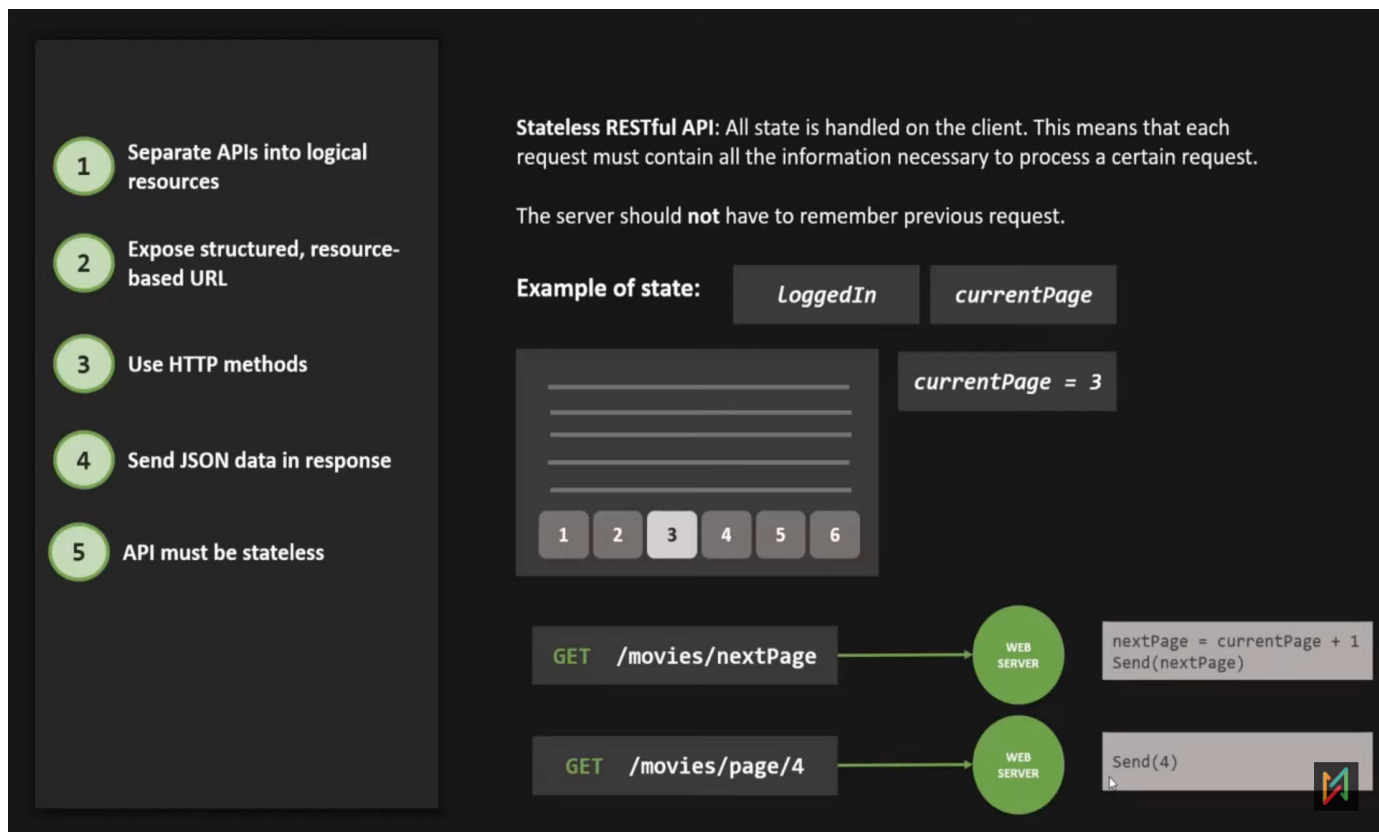
Resource: A resource is an object or representation of something which has data associated to it. Any information that can be **named** can be a resource.

movies

users

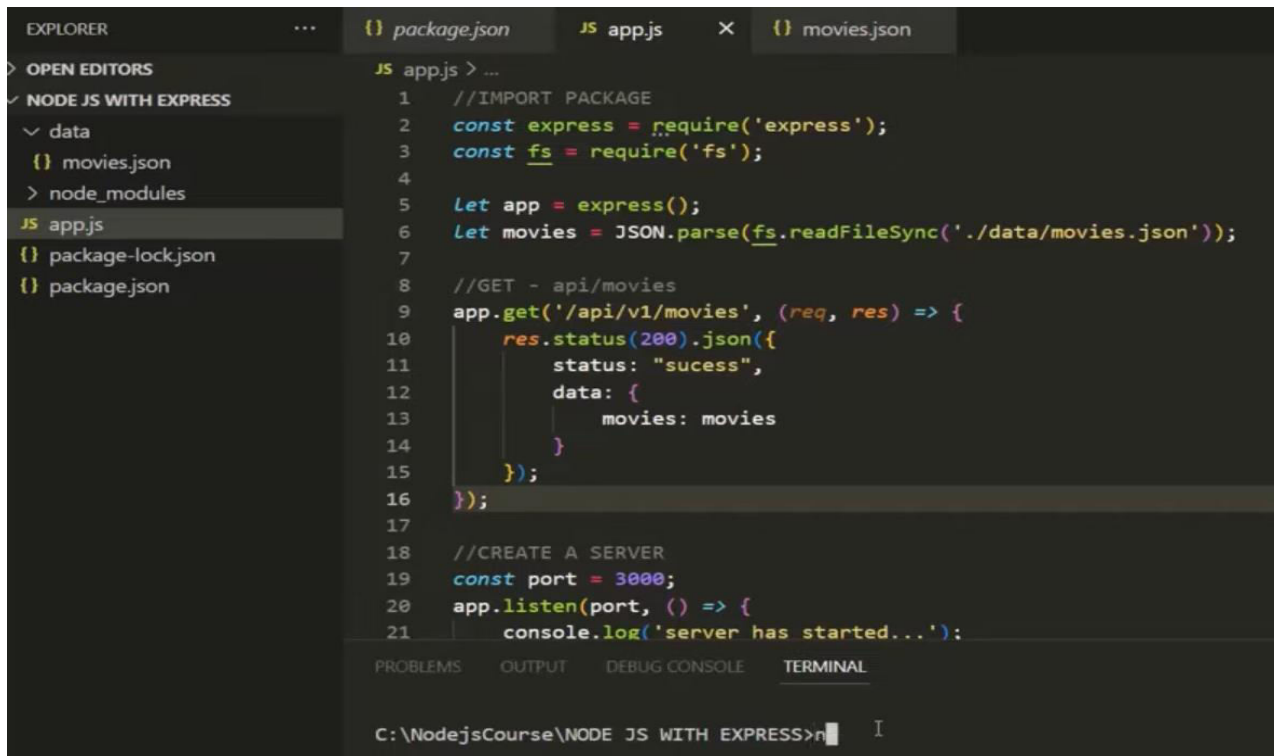
reviews

Resource should only be a noun.



Handling GET Request:-

Here, we are creating an API using REST architecture.



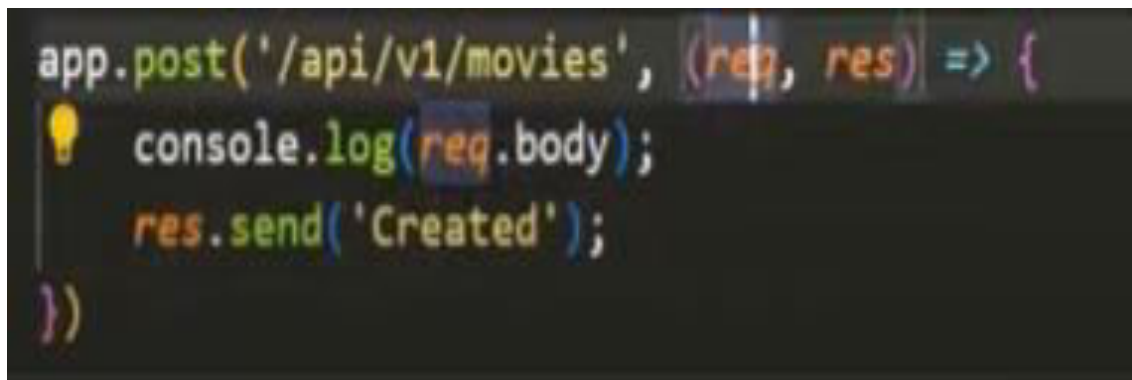
The screenshot shows the VS Code editor with the following components:

- EXPLORER:** Shows the project structure with files like `package.json`, `app.js`, `movies.json`, `package-lock.json`, and `package.json`.
- EDITOR:** Displays the `app.js` file with the following code:

```
1 //IMPORT PACKAGE
2 const express = require('express');
3 const fs = require('fs');
4
5 let app = express();
6 let movies = JSON.parse(fs.readFileSync('./data/movies.json'));
7
8 //GET - api/movies
9 app.get('/api/v1/movies', (req, res) => {
10   res.status(200).json({
11     status: "sucess",
12     data: {
13       movies: movies
14     }
15   });
16 });
17
18 //CREATE A SERVER
19 const port = 3000;
20 app.listen(port, () => {
21   console.log('server has started...');
```
- TERMINAL:** Shows the command prompt at `C:\NodejsCourse\NODE JS WITH EXPRESS>`.

That's how we can create GET API by REST Architecture.

Handling POST Request:-



The screenshot shows the VS Code editor with the following code snippet for a POST endpoint:

```
app.post('/api/v1/movies', (req, res) => {
  console.log(req.body);
  res.send('Created');
})
```

We set data to the given URL (that's how we send data using POSTMAN).

But when we sent the data , so req.body should contain the data , but in vs code terminal it is showing undefined:-

```
JS app.js > app.post('/api/v1/movies') callback
5   let app = express();
6   let movies = JSON.parse(fs.readFileSync('./data/movies.json'));
7
8   //GET - api/v1/movies
9   app.get('/api/v1/movies', (req, res) => {
10     res.status(200).json({
11       status: "sucess",
12       count: movies.length,
13       data: {
14         movies: movies
15       }
16     });
17   });
18
19   //POST - api/v1/movies
20   app.post('/api/v1/movies', (req, res) => {
21     console.log(req.body);
22     res.send('Created');
23   })
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2
```

Route parameters are named URL segments that are used to capture the values specified at their position in the URL

127.0.0.1:3000/api/v1/movies/:id

127.0.0.1:3000/api/v1/movies/4

id = 4

127.0.0.1:3000/api/v1/movies/23

id = 23

127.0.0.1:3000/api/v1/movies/217

id = 217

This (:id) is the route parameter which can be replaced by any integer value.

That's how the code will look like:-

```
GET - api/v1/movies/id
app.get('/api/v1/movies/:id', (req, res) => {
  console.log(req.params);

  res.send('Test movie');
})
```

Console.log(req.params) will print the parameters and in this case the id's will be printed.

For multiple parameters we can do this:-

```
app.get('/api/v1/movies/:id/:name/:x', (req, res) => {  
  console.log(req.params);  
  
  res.send('Test movie');  
})
```

The no. of URL parameters which is mentioned in the `app.get()` method should be the same as the no. of request URL parameters which is done when you are making the request in POSTMAN or anywhere else, otherwise it will show an error.

If we do not want any error, so we can make the route parameter optional by putting the (?) question mark in front of the parameter.

It looks like this:-

```
GET - api/v1/movies/id  
app.get('/api/v1/movies/:id/:name?/:x?', (req, res) => {  
  console.log(req.params);  
  
  res.send('Test movie');  
})
```

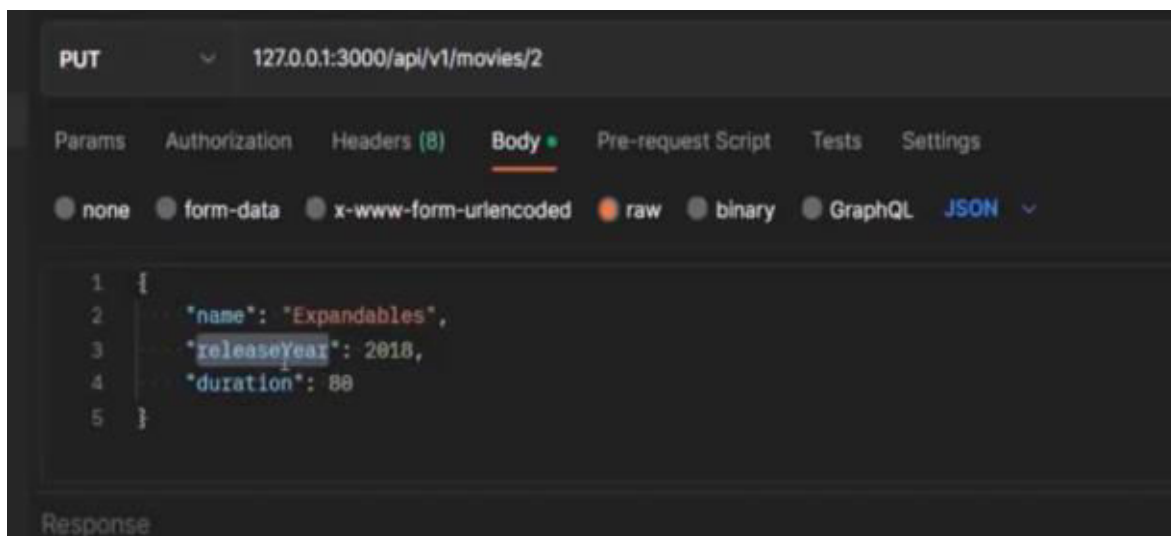
Now even if we do not provide any name or route parameter then it won't be an issue.

Handling PATCH API:-

PUT vs PATCH

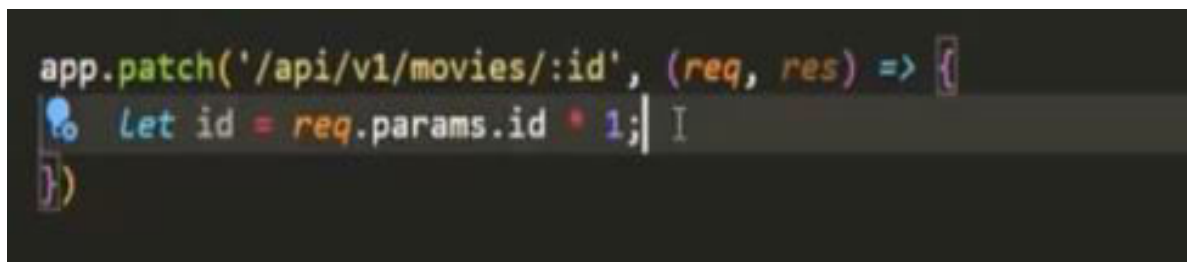
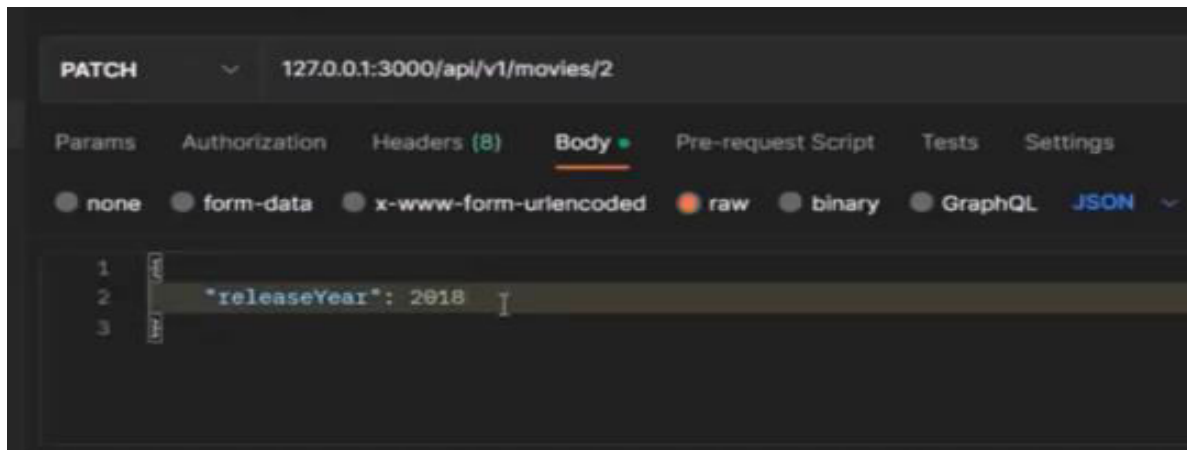
- **PUT** is a method of modifying resource where the client sends data that updates the entire resource .
- **PATCH** is a method of modifying resources where the client sends partial data that is to be updated without modifying the entire data.

In PUT method , if we have to change the existing json data of a particular variable, then also we have to provide entire JSON data in order to update the desired variable.



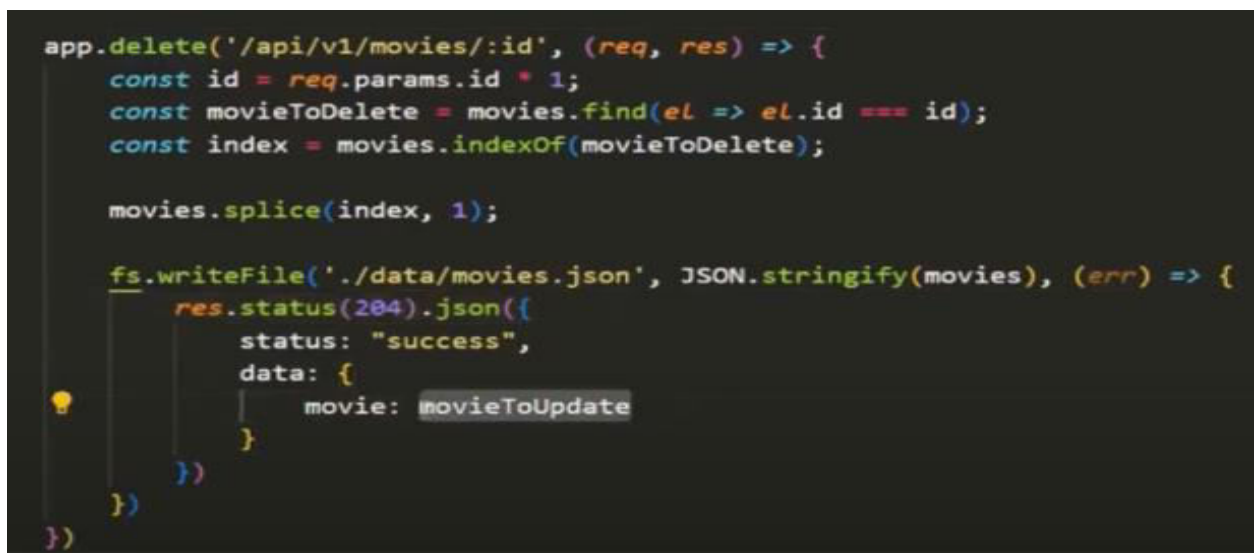
Here, only releaseYear is changing but still, we are providing entire JSON data.

But if use PATCH method, then I will only do this:-



Handling DELETE API:-

This is the basic code , the internal code is the logic of deletion but syntax is this only.



The handlers functions are also the middleware functions.

MIDDLEWARE:-

In backend development, middleware refers to software components or functions that are executed between the receiving of a request and the generation of a response in a web application or server. Middleware operates on the server side and plays a crucial role in processing, augmenting, or controlling the flow of data between the client and the server. It provides a way to modularize and handle specific tasks or features in a web application.

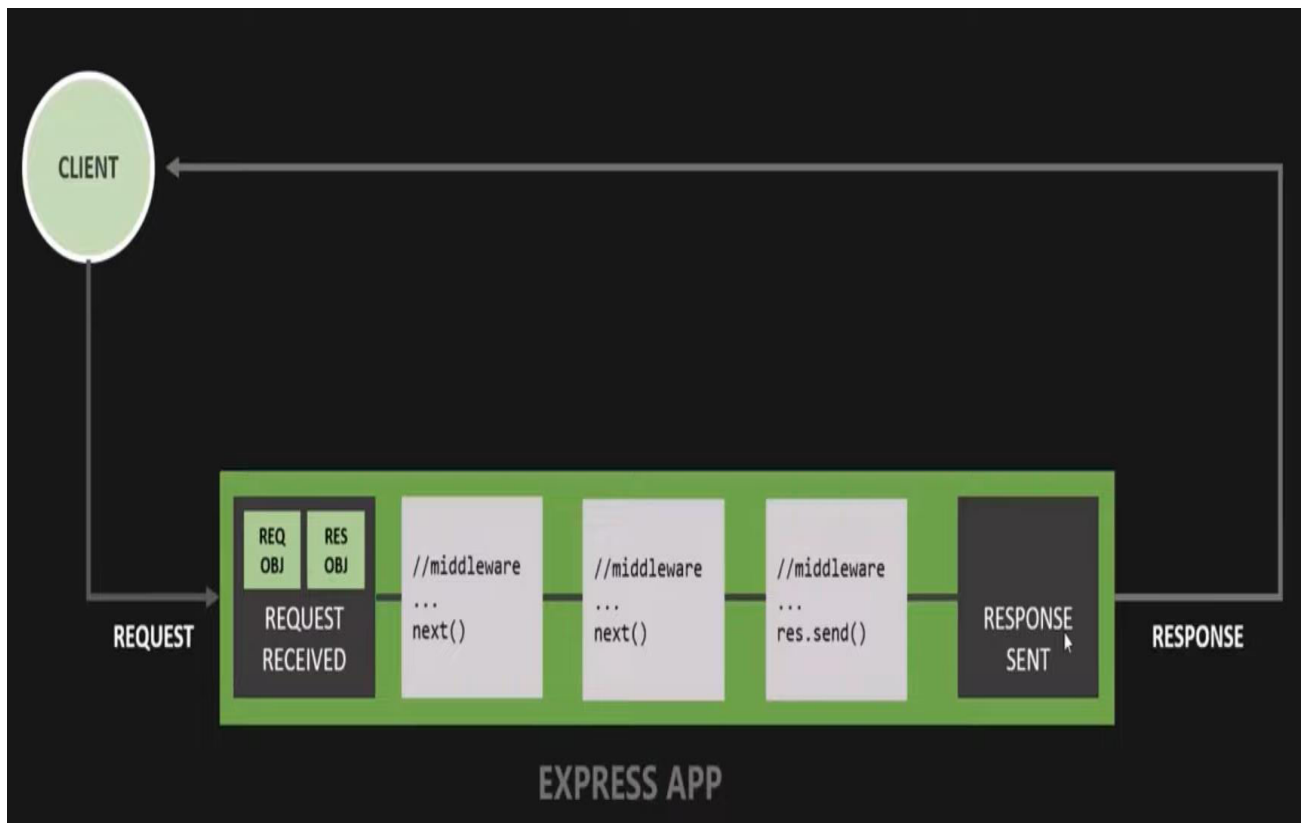
Here are some common use cases for middleware in backend development:

1. **Request Processing:** Middleware can intercept incoming HTTP requests and perform actions such as parsing request data, validating input, or extracting information.
2. **Authentication and Authorization:** Middleware is often used for user authentication and authorization. It can check user credentials, verify access rights, and grant or deny access to specific resources.
3. **Logging and Monitoring:** Middleware can log information about incoming requests, responses, errors, and other events. This logging is valuable for debugging, performance monitoring, and security analysis.
4. **Error Handling:** Middleware can catch errors that occur during request processing and handle them in a centralized manner. It helps improve the robustness of the application and provides a consistent way to deal with errors.
5. **Caching:** Middleware can implement caching mechanisms to store and retrieve data, reducing the need to recompute or fetch the same information repeatedly.
6. **Compression and Encryption:** Middleware can perform tasks such as compressing responses to reduce bandwidth usage or encrypting communication for security.
7. **Routing:** Middleware can handle URL routing, directing incoming requests to the appropriate endpoints or controllers based on predefined rules.
8. **Response Transformation:** Middleware can modify or transform the server's response before it is sent to the client. This can include modifying headers, formatting data, or adding additional information.
9. **Request Filtering:** Middleware can filter or preprocess incoming requests based on certain criteria, such as blocking requests from specific IP addresses or applying security checks.
10. **Rate Limiting:** Middleware can enforce rate limits on incoming requests to prevent abuse or ensure fair usage of resources.

In various web frameworks and server-side platforms, middleware is often a fundamental concept. For example:

- In Express.js (Node.js framework), developers can use existing middleware or create custom middleware functions to enhance the functionality of their routes.
- In Django (Python web framework), middleware classes can be used to process requests and responses globally across the entire application.
- In Ruby on Rails, middleware is used to handle tasks like session management, security, and more.

Middleware provides a way to extend and modularize the functionality of backend systems, making the codebase more modular, maintainable, and scalable. It allows developers to separate concerns and address specific aspects of request-response processing in a structured manner.



If there are multiple middleware functions then each middleware function will run according to the position where they are written in the code, the upper will run 1st and the bottom will run in the last.

Creating Custom Middleware:-

To create any middleware we use the `(use())` method.

A middleware function always receives 3 arguments which are:- 1) request object 2) response object 3) next method.

```
const logger = function(req, res, next){  
  console.log('Custom middleware called');  
  next();  
}  
  
app.use(express.json());  
app.use(logger);
```

Here, we have created the logger function, which is a custom middleware function and to make it work we use the .use() method.

After console.log() next() is written so that, we pass into another middleware function, else we will stuck in that particular middleware.

Route handlers are also middleware's.

NOTE:- instead of doing this –

```
GET - api/v1/movies/id
app.get('/api/v1/movies/:id', (req, res) => {
  console.log(req.params);

  res.send('Test movie');
})
```

We can also do this:-

```
app.route('/api/v1/movies/:id')
  .get(getMovie)
  .patch(updateMovie)
  .delete(deleteMovie)
```

Provided route by .route() method and attached what type of requests need to be present with this route.

We can define middleware function inside the.use() method:-

```
app.use((req, res, next) => {
  req.requestedAt = new Date().toISOString();
  next();
})
```

3rd Party Middleware:-

It needs to be installed externally for example Morgan.

Morgan is a login middleware and it allows us to see request data by writing in a console.

NOTE-> Whenever we want to install dependencies as regular dependencies we simply write (npm install {name of the package}) but if we want to install dependencies as dev dependencies we have to write (npm install {name of the package} (--save-dev)).

After installing this we need to require the package.

```
const morgan = require('morgan');
```

And to use this:-

```
app.use(morgan());
```

Here morgan() is going to return the middleware function.

But in morgan() function we have to pass two parameters the 1st one is format and the 2nd one is optional.

In 1st formal we can send these 5 things:-

```
app.use(morgan(' '))
app.use(logger);
app.use((req, res) => {
  req.requested
  next();
})
```

	combined
common	
dev	
short	
tiny	

Let's say we select the 'dev' one.

Whatever you select it will only return the info about the request , just their way of representation will change , nothing else.

This morgan middleware will be applied to every request.

Let's say we made one request:-



So the output will look like this if we use the morgan function:-



1st thing it will tell is the type of request which is GET in this case and 2nd thing it will tell the route where logged, and 3rd one is the status code, and 4th one is the time taken by the server to send the response and the 5th one is the memory in bytes.

Mounting Router:-

When we do app.use() to run a middleware, the middleware will applied on all the routes but we can apply middleware to perticular routes.

```
app.route('/api/v1/movies')
  .get(getAllMovies)
  .post(createMovie)

app.route('/api/v1/movies/:id')
  .get(getMovie)
  .patch(updateMovie)
  .delete(deleteMovie)
```

Here our routes are in the same router and the router is this app object.

But if we have to separate routes, for different resources, then we create a router like this:-

```
const moviesRouter = express.Router();
```

Here, we have created a router by using `express.Router()`;

The router we have created will separately work for a particular resource like in this case movies.

Here now `express.router()` returns a middleware so `moviesRouter` will also be a middleware here.

But we want to make this middleware work for only chosen routes so we will do this:-

```
app.use('/api/v1/movies', moviesRouter)
```

Here this `moviesRouter` will work only for those requests whose URL/path is this `/api/v1/movies`.


```
const moviesRouter = express.Router();
moviesRouter.route('/')
  .get(getAllMovies)
  .post(createMovie)

moviesRouter.route('/:id')
  .get(getMovie)
  .patch(updateMovie)
  .delete(deleteMovie)
app.use('/api/v1/movies', moviesRouter)
//CREATE A SERVER
```

Here, in `moviesRouter.route('/:id')`, whenever we make request, which consists of this URL `/api/v1/movies` and has a particular id then the which we are adding will append to the url `/api/v1/movies` and will make `app.use('/api/v1/movies',moviesRouter)` to `app.use('/api/v1/movies/:id',moviesRouter)`.

Creating Routes Module:-

We can implement this :-

```

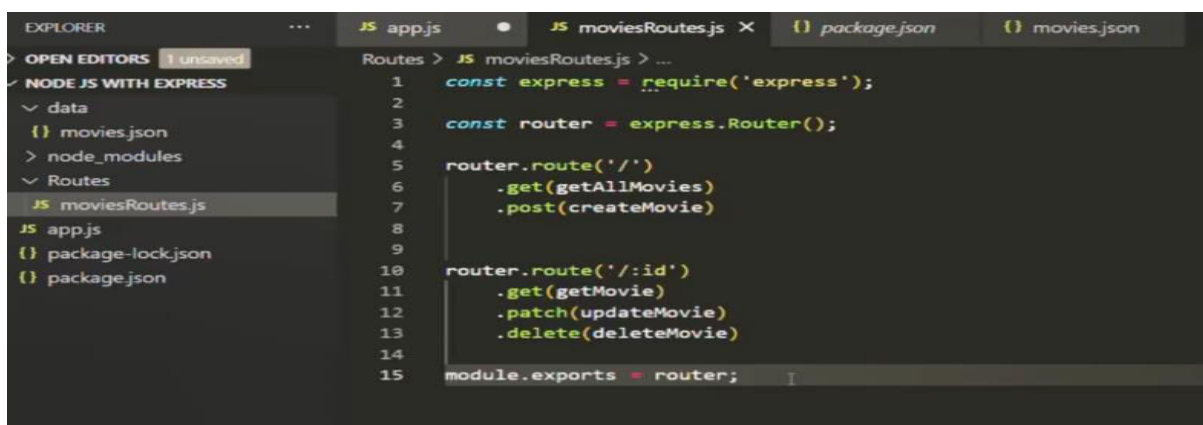
const moviesRouter = express.Router();
moviesRouter.route('/')
  .get(getAllMovies)
  .post(createMovie)

moviesRouter.route('/:id')
  .get(getMovie)
  .patch(updateMovie)
  .delete(deleteMovie)

app.use('/api/v1/movies', moviesRouter)
//CREATE A SERVER

```

By creating separate module like this:-



```

1  const express = require('express');
2
3  const router = express.Router();
4
5  router.route('/')
6    .get(getAllMovies)
7    .post(createMovie)
8
9  router.route('/:id')
10   .get(getMovie)
11   .patch(updateMovie)
12   .delete(deleteMovie)
13
14
15  module.exports = router;

```

Here, we have created another module and here we have to add all the functions that we are using for getallmovies and all types of other resources but generally, we create another folder named controllers(the name controllers is depicted because of MVC architecture) to store all route handler functions. This module also needs to be imported inside the app.js file.

Param Middleware:-

It is a special Middleware which runs for a certain Route Parameters.

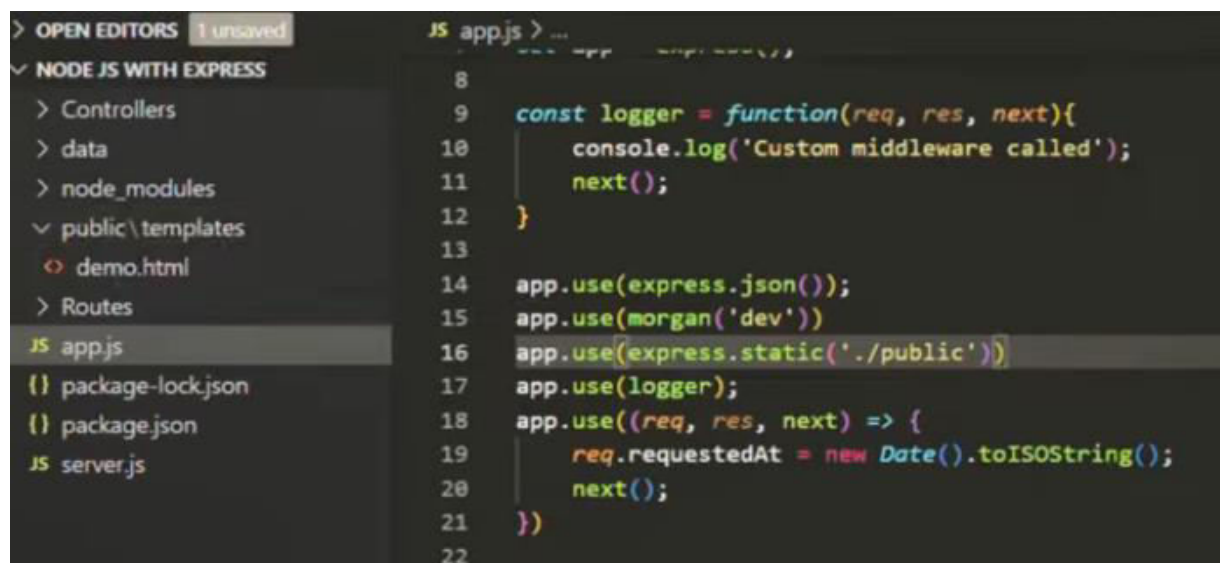
```
router.param('id', (req, res, next, value) => {  
  })
```

Here, the 1st argument is the name of the parameter and the next parameter is the callback function which itself has 4 arguments, request object, response object, next function, and the value of the id.

Serving Static Files:-

static files refer to files that are served directly to the client without any processing by the server. These files typically include things like stylesheets (CSS files), client-side JavaScript files, images, fonts, and other assets that do not require server-side logic to generate or process.

Express provides a built-in middleware function called `express.static` to serve static files. This middleware takes a directory path as an argument and serves the files from that directory.



```
> OPEN EDITORS 1 unsaved  
✓ NODE JS WITH EXPRESS  
  > Controllers  
  > data  
  > node_modules  
  ✓ public\templates  
    demo.html  
  > Routes  
  JS app.js  
  {} package-lock.json  
  {} package.json  
  JS server.js  
JS app.js > ...  
8  
9 const logger = function(req, res, next){  
10   console.log('Custom middleware called');  
11   next();  
12 }  
13  
14 app.use(express.json());  
15 app.use(morgan('dev'))  
16 app.use(express.static('./public'))  
17 app.use(logger);  
18 app.use((req, res, next) => {  
19   req.requestedAt = new Date().toISOString();  
20   next();  
21 })  
22
```

Inside `express.static()` we have to provide the path of the folder where our static files are present.

Environment Variables:-

Environment variables in Express.js are configuration settings that are external to your application code and are set in the environment where your Node.js application is running. These variables are accessible through the `process.env` object in Node.js.

`Process.env` has already many variables where the process is the core Module of the node.js. The process Module we do not need to require it, is automatically present everywhere. Express.js applications commonly use environment variables to configure various aspects of the application, such as database connection strings, API keys, server ports, and other settings.

Here's how you can work with environment variables in an Express.js application:

1. **Setting Environment Variables:**

- You can set environment variables in various ways, depending on your deployment environment.
- For local development, you might use a `.env` file to store environment variables.
- For production, you might set environment variables directly on your server or use a configuration management tool.

2. **Accessing Environment Variables in Express.js:**

- Express.js applications can access environment variables using `process.env`.
- For example, if you have an environment variable named `DATABASE_URL`, you can access it in your Express.js application as `process.env.DATABASE_URL`.

3. **Using a Package like dotenv (for local development):**

- For local development, you might use a package like `dotenv` to load environment variables from a `.env` file.
- Install `dotenv` using `npm install dotenv` and then include the following line at the top of your entry file (e.g., `app.js`):
- Ensure that your `.env` file contains key-value pairs (e.g., `PORT=3000`).

By using environment variables, you can separate configuration from your code, making your application more flexible and secure. It also allows you to configure different settings for development, testing, and production environments. Always be mindful of not exposing sensitive information in your code or configuration files.

Environment variables are the global variables that are used to define the environments in which the node.js app is running.

After we create our config.env file to store our created variables we have to do this:-

```
1  const dotenv = require('dotenv');  
2  dotenv.config({path: './config.env'});  
3
```

First requires dotenv module, which enables to use of the env variables locally.

By doing dotenv.config() whatever variables are stored in config.env file will also be stored in node.js environment variables.

3) MONGODB:-

1st download MongoDB compass.

Creating a Hosted Database Using Atlas:-

Follow this series:-

What is your goal today?
Your answer will help us guide you to successfully getting started with MongoDB Atlas.

- ☒ Learn MongoDB
- ☐ Explore what I can build
- ☐ Build a new application
- ☐ Migrate an existing application


What type of application are you building?

Select...

- Real-Time Messaging
- Mobile App
- Microservices or APIs
- I'm just exploring**
- Other

What is your preferred language?
We'll use this to customize code samples and content we share with you. You can always change this later.

JS JavaScript

 **MongoDB.**

Deploy your database

Use a template below or set up [advanced configuration options](#). You can also edit these configuration options once the cluster is created.

M10 **\$0.08/hour**

For production applications with sophisticated workload requirements.

STORAGE	RAM	vCPU
10 GB	2 GB	2 vCPUs

SERVERLESS **\$0.10/1M reads**

For application development and testing, or workloads with variable traffic.




STORAGE	RAM	vCPU
Up to 1 TB	Auto-scale	Auto-scale

M0 **FREE**


For learning and exploring MongoDB in a cloud environment.

STORAGE	RAM	vCPU
512 MB	Shared	Shared

Provider



Region ★ Recommended region ⓘ

 N. Virginia (us-east-1) ★

Name

You cannot change the name once the cluster is created.

Cluster0

Name the Cluster whatever you want.

Now, click on Create.

Atlas Manoj's Org ... Access Manager Billing

Project 0 Data Services App Services Charts

DEPLOYMENT

Database

Data Lake PREVIEW

SERVICES

Triggers

Data API

Data Federation

Search

SECURITY

Quickstart

Database Access

Network Access

Advanced

New On Atlas 2

MO Cluster Provisioning...

We autogenerated a username and password for your first database user in this project using your MongoDB Cloud registration information. ✕

Create a database user using a username and password. Users will be given the *read and write to any database privilege* by default. You can update these permissions and/or create additional users later. Ensure these credentials are different to your MongoDB Cloud username and password.

Username

admin

Password ⓘ

n9Hb3bOrVoF4pGKw ⓘ

Autogenerate Secure Password

Copy

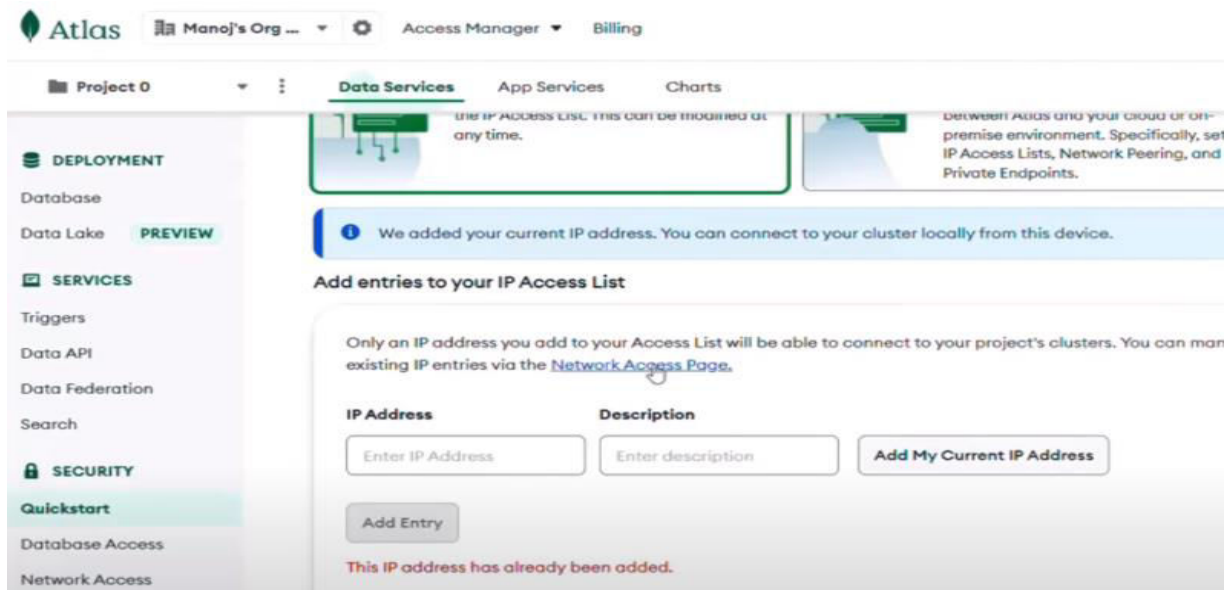
Create User

to connect from?

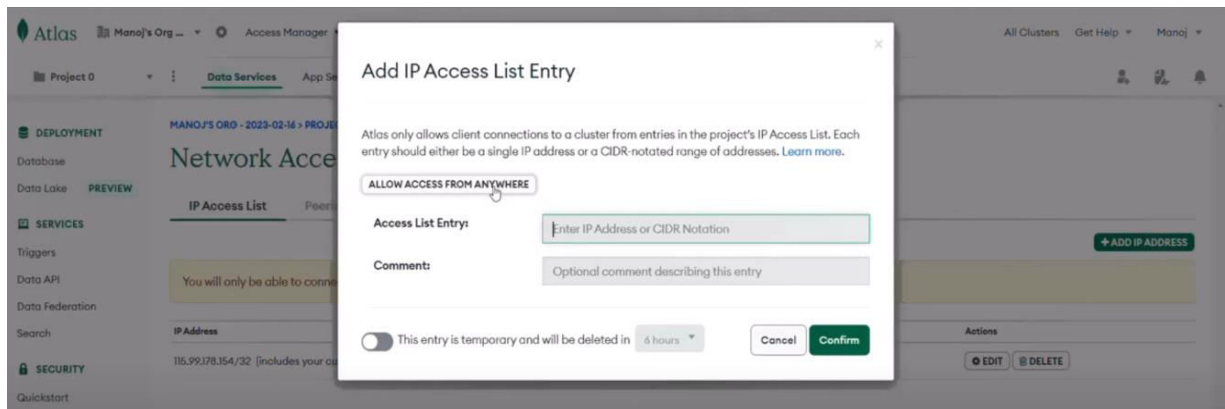
Now, set username and Password and click on Create User.

And store the username and password in config.env file.


```
NODE_ENV=development
PORT=3000
DB_USER=admin
DB_PASSWORD=n9Hb3b0rVoF4pGKw
```

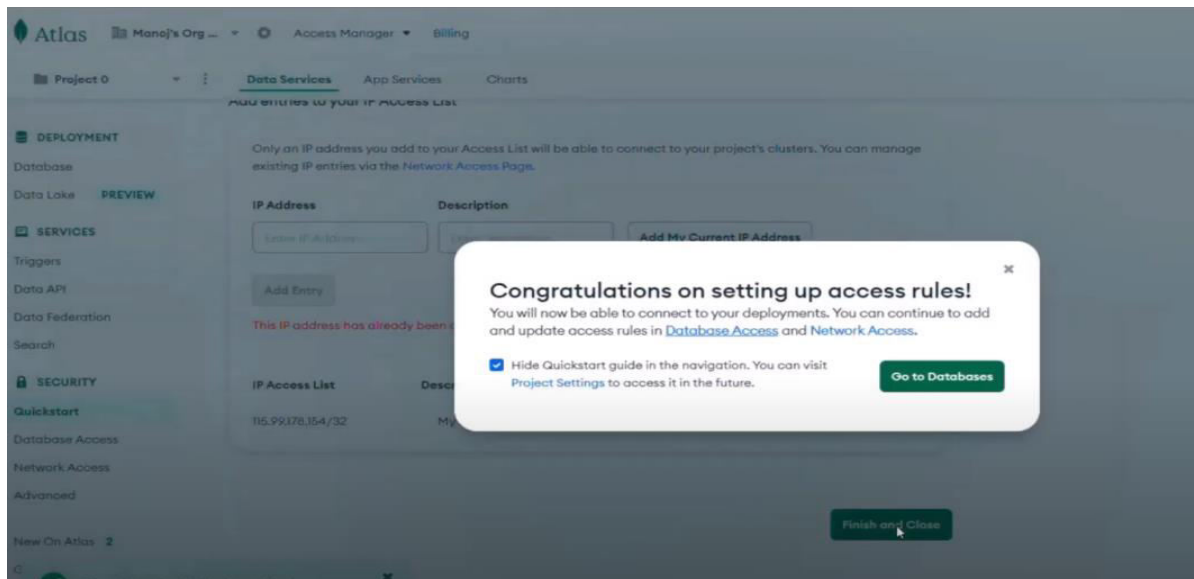


Now click on the Network Access Page.

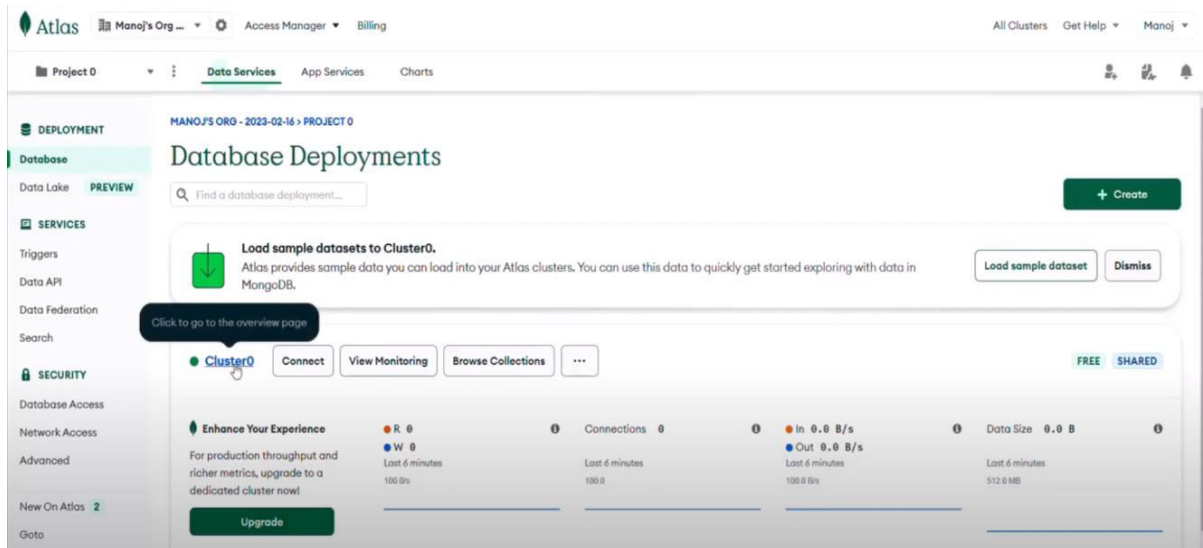


Click on ALLOW ACCESS FROM ANYWHERE and then in confirm.

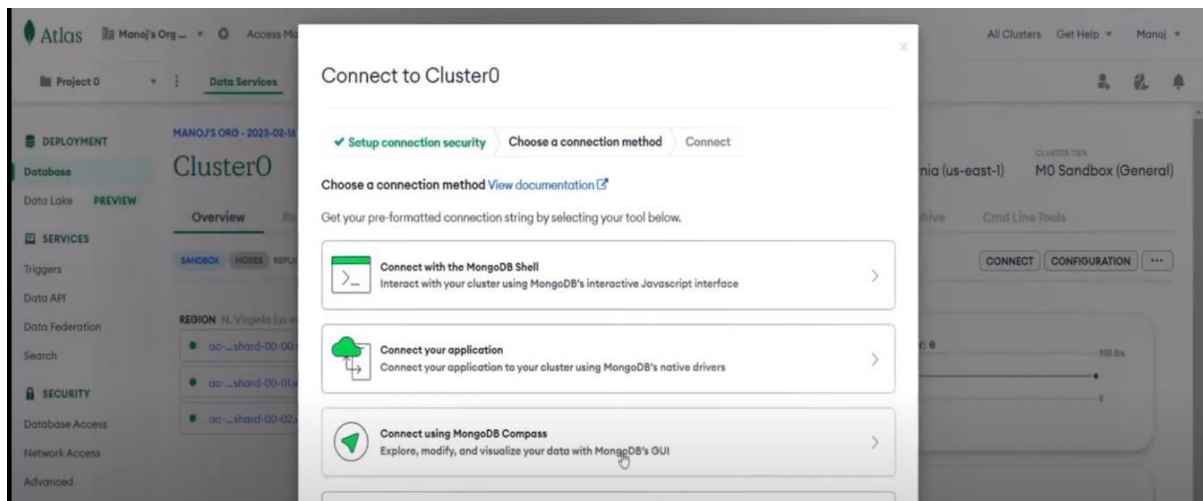
Then click on Atlas logo, the click on finish and Close Button.



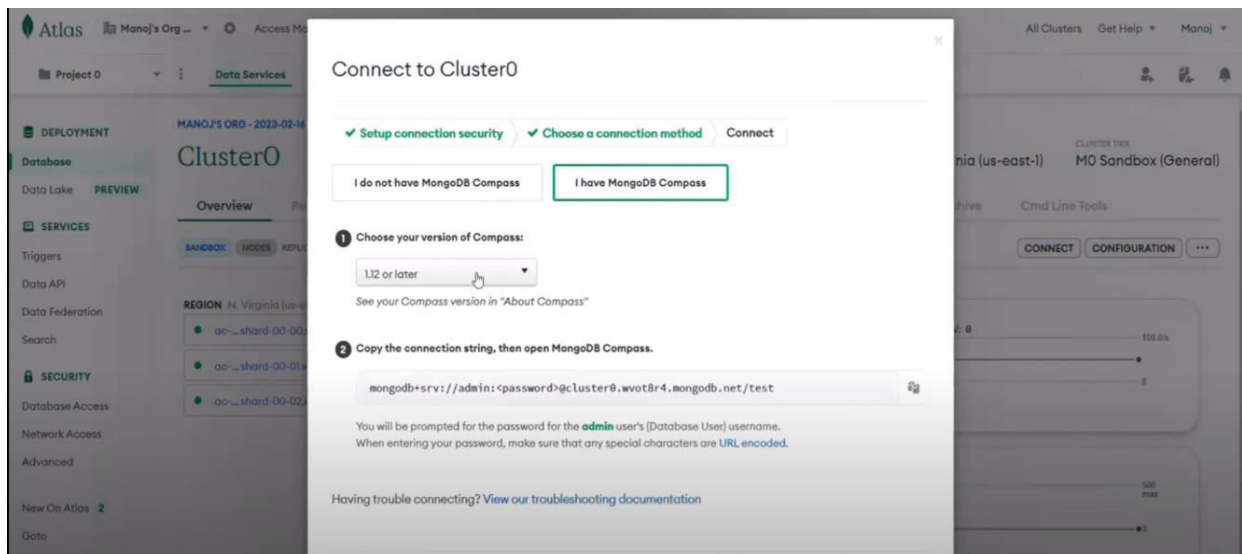
Then click on go to Databases.



Then click on connect.

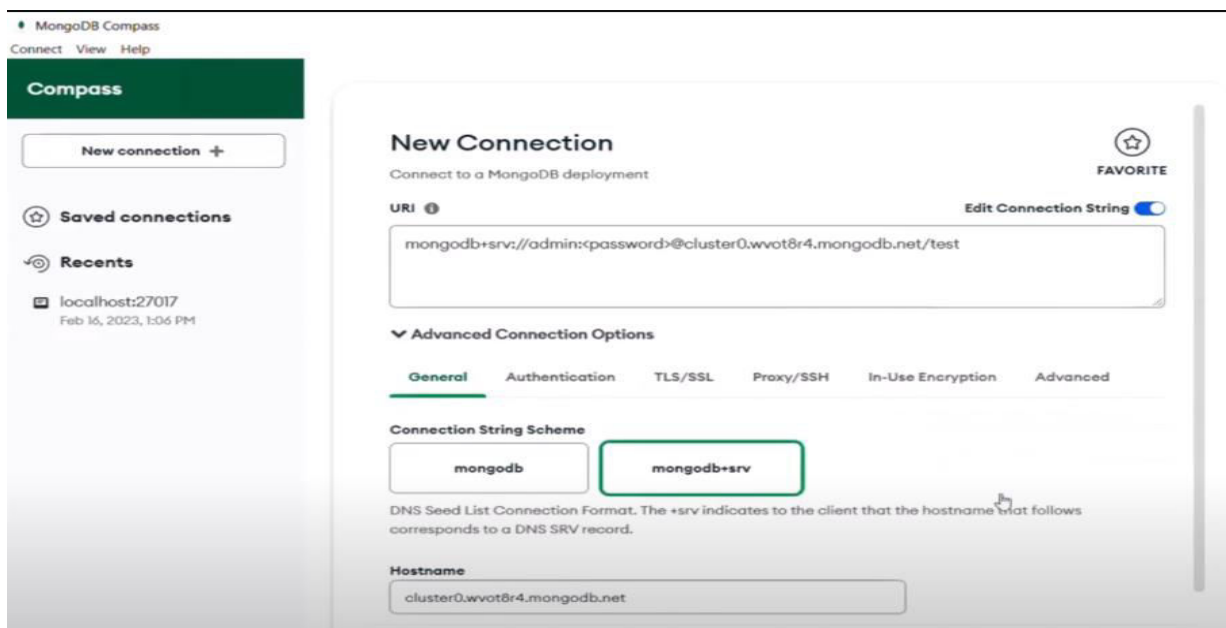


Then connect using MongoDB Compass.



Then choose the version of the compass.

Copy the link which is written under copy the connection string and then go to MongoDB compass.



Past the string inside the URL box and then open advanced connection options.

This screenshot shows the 'Authentication Method' configuration in MongoDB Atlas. The 'Username/Password' method is selected. The 'Username' field contains 'admin' and the 'Password' field is masked with dots. The 'Authentication Database' field is empty and marked as optional. Under 'Authentication Mechanism', 'Default' is selected. At the bottom, there are 'Save', 'Save & Connect', and 'Connect' buttons.

Authentication Method

None Username/Password X.509 Kerberos LDAP AWS IAM

Username

admin

Password

.....

Authentication Database ⓘ

Optional

Authentication Mechanism

Default SCRAM-SHA-1 SCRAM-SHA-256

Save Save & Connect Connect

Then click on Authentication, then choose the authentication method as username/password. Those fill username and password the same as you have created in MongoDB Atlas. This one:-

This screenshot shows the 'Create User' form in the MongoDB Atlas 'Data Services' section. A notification at the top states that a username and password were autogenerated. The form includes fields for 'Username' (filled with 'admin') and 'Password' (filled with 'n9Hb3bOrVoF4pGKw'). There are buttons for 'Autogenerate Secure Password' and 'Copy'. A 'Create User' button is at the bottom. The left sidebar shows navigation options like 'DEPLOYMENT', 'SERVICES', and 'SECURITY'.

Atlas Manoj's Org ... Access Manager Billing

Project 0 Data Services App Services Charts

DEPLOYMENT

Database

Data Lake PREVIEW

SERVICES

Triggers

Data API

Data Federation

Search

SECURITY

Quickstart

Database Access

Network Access

Advanced

New On Atlas 2

M0 Cluster Provisioning...

We autogenerated a username and password for your first database user in this project using your MongoDB Cloud registration information.

Create a database user using a username and password. Users will be given the *read and write to any database privilege* by default. You can update these permissions and/or create additional users later. Ensure these credentials are different to your MongoDB Cloud username and password.

Username

admin

Password ⓘ

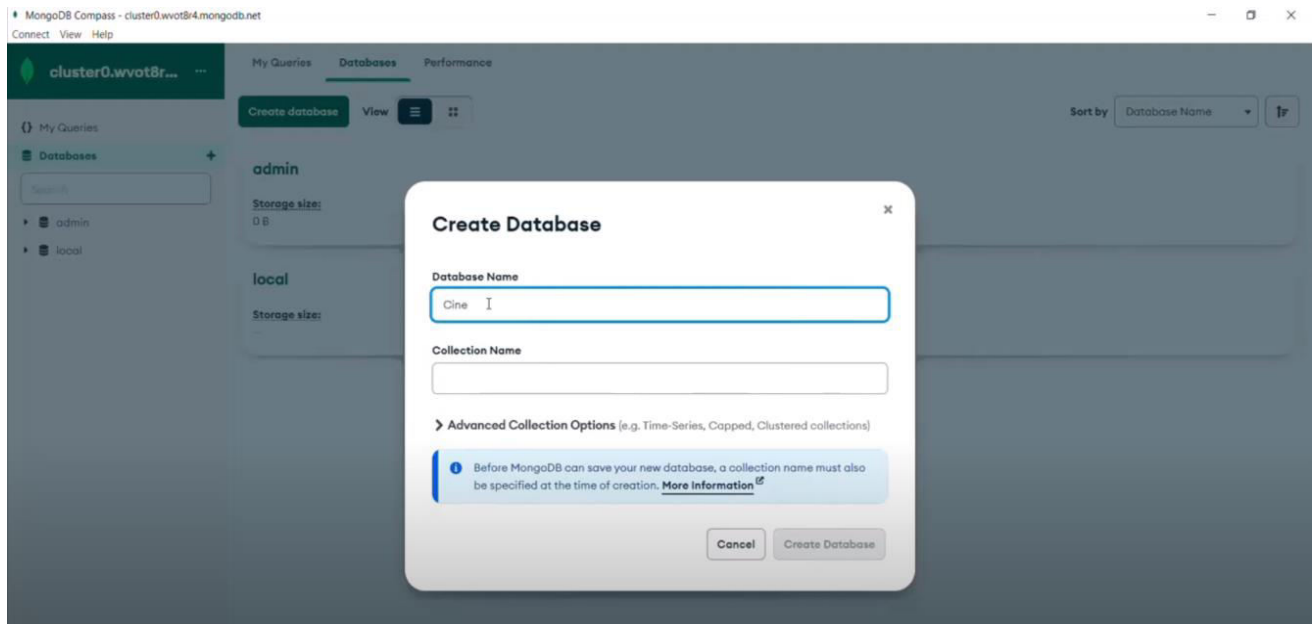
n9Hb3bOrVoF4pGKw

Autogenerate Secure Password Copy

Create User

to connect from?

Then click on create and your cluster has been connected. Now, create a Database and name it:-



Connecting Remote Database From Express:-

Store the connection string in config.env file.

```
config.env
config.env
1  NODE_ENV=development
2  PORT=3000
3  CONN_STR=mongodb+srv://admin:<password>@cluster0.wvot8r4.mongodb.net/?retryWrites=true&w=majority
4  DB_USER=admin
5  DB_PASSWORD=n9Hb3b0rV3F4pGKw
```

In CONN_STR replace <password> with the actual password (DB_PASSWORD).

To connect MongoDB database to our node.js we should have to install mongoose by npm install mongoose.

And in server.js file or your main file , require mongoose and connect the string like this:-

```
config.env JS server.js X
JS server.js > then() callback
1  const mongoose = require('mongoose');
2  const dotenv = require('dotenv');
3  dotenv.config({path: './config.env'});
4
5  const app = require('./app');
6
7  //console.log(app.get('env'));
8  console.log(process.env);
9
10 mongoose.connect(process.env.CONN_STR, {
11   useNewUrlParser: true
12 }).then((conn) => {
13   console.log(conn);
14   console.log('DB Connection Successful');
15 })
16
17 const port = process.env.PORT || 3000;
18
19 app.listen(port, () => {
20   console.log('server has started...');
21 })
```

Mongoose:-

What is Mongoose and its features:

- Mongoose is an object data modelling (ODM) library for MongoDB & NODE JS, providing higher level of abstraction
- **Features:** Schema to model our data and relationships, easy data validation, a simple query API, middleware etc.
- In mongoose, a schema is where we model our data. Using schema, we can describe the structure of our data, default values & validations.
- We use this schema to create model out of it.

Creating a Schema and Model:-

"schema" is often associated with the structure or definition of data, especially when working with databases. It typically refers to the way data is organized and the rules that define the valid structure of that data. The concept of a schema becomes particularly relevant when dealing with databases, object-relational mapping (ORM) libraries, and data validation.

Syntax:-

```
const movieSchema = new mongoose.Schema({  
  // ...  
});
```

Here, in the 1st argument, we have to determine the properties and the 2nd argument is optional.

For ex:-

```
const movieSchema = new mongoose.Schema({  
  name: {  
    type: String,  
    required: [true, 'Name is required field!']  
  },  
  description: String,  
  duration: {  
    type: Number,  
    required: [true, 'Duration is required field!']  
  },  
  ratings: Number,  
});
```

In this schema every property has its specifications for example in name , we have assigned the datatype as string and mentioned whether this field is required or not and the required is a array in which 1st element tells whether this field should be filled compulsorily or not and 2nd element will be printed out of we do not fill anything and 1st element is true.

And based on the Schema we have created a model.

```
const Movie = mongoose.model('Movie', movieSchema);
```

Model name must start with the capital letter.

Here, in the 1st argument will assign the name of the model, and in the 2nd argument, we have to tell on what schema we are making this model.

Creating a Document from Model:-

```
const Movie = mongoose.model('Movie', movieSchema);

const testMovie = new Movie({
  name: "Die hard",
  description: "Action packed movie starring bruce willis in this trilling adventure.",
  duration: 139,
  ratings: 4.5
});

testMovie.save()
  .then(doc => {
    console.log(doc);
  })
  .catch(err => {
    console.log("Error occured: " + err);
  });
```

Here, testMovies is a document created from a model that follows Schema. Now testMovies.save() will save this document in the database collection and this (.save()) will return a promise.

But there is another way of storing documents in a database collection like this:-

```

exports.createMovie = async (req, res) => {
  try{
    const movie = await Movie.create(req.body);

    res.status(201).json({
      status: 'success',
      data: {
        movie
      }
    })
  }catch(err){
    res.status(400).json({
      status: 'fail',
      message: err.message
    })
  }
}

```

```

const express = require('express');
const moviesController = require('../Controllers/moviesController');

const router = express.Router();

//router.param('id', moviesController.checkId)

router.route('/')
  .get(moviesController.getAllMovies)
  .post(moviesController.createMovie)

router.route('/:id')
  .get(moviesController.getMovie)
  .patch(moviesController.updateMovie)
  .delete(moviesController.deleteMovie)

module.exports = router;

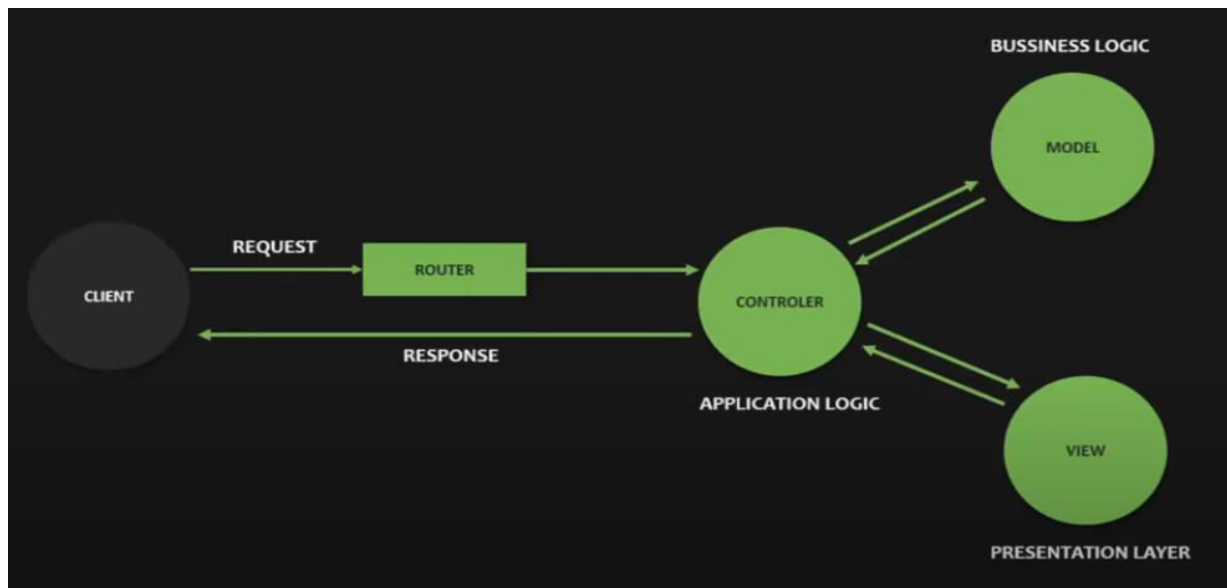
```

Here, we already mentioned the handlers request type in Routes section for example createMovie has post type Request method.

Here this createMovie handler is a post request , here we have created this as a async function because inside we are using await. Here inside try{} (Movie.create() [this .create() will also return promise] will stores the document inside the document collection and as we know, whatever post we do it

all will store in request body , (req.body) is the document that we wanna store in a database collection.

MVC Architecture:-



The MVC (Model-View-Controller) architecture is a software design pattern that separates an application into three interconnected components to achieve modularity and maintainability. It is commonly used in backend development, as well as in frontend development, to organize code and promote a clear separation of concerns. Each component has a specific responsibility, and changes to one component do not directly affect the others.

Here's a brief overview of each component in the MVC architecture:

1. **Model:**

- The Model represents the application's data and business logic. It is responsible for managing the application's state, responding to queries from the View, and updating the data when instructed by the Controller.
- In a backend context, the Model often interacts with the database, performs data validation, and encapsulates the application's core functionality.

2. **View:**

- The View is responsible for presenting the data to the user. It displays information and interacts with the user interface. Views receive data from the Model and present it to the user in a human-readable format.
- In a backend context, the View might involve rendering templates, generating responses, or formatting data for presentation.

3. **Controller:**

- The Controller acts as an intermediary between the Model and the View. It receives user input from the View, processes it, updates the Model accordingly, and determines what should be presented to the user in the View.
- In a backend context, the Controller often handles HTTP requests, invokes methods on the Model to perform operations, and selects the appropriate View for rendering the response.

How MVC Works in a Backend Framework (e.g., Express.js):

In a backend web application built with a framework like Express.js, the MVC architecture is often implemented as follows:

- **Model:** Represents data models and interacts with the database. It might include database schemas, data access logic, and business logic.
- **View:** Represents the presentation layer. In a web application, this could be HTML templates, JSON responses, or other formats that are sent to the client.
- **Controller:** Handles incoming requests, processes the data, interacts with the Model, and selects the appropriate View. In Express.js, route handlers often act as Controllers.

Here's a simplified example using Express.js:

```
// Model
```

```
const userModel = require('./models/user');
```

```
// Controller
```

```
const userController = {  
  getUser: async (req, res) => {  
    const userId = req.params.id;
```

```
try {  
    const user = await userModel.getUserById(userId);  
    res.render('userProfile', { user });  
} catch (error) {  
    res.status(500).json({ error: 'Internal Server Error' });  
}  
},  
};
```

// View (not explicitly shown in this example, but it could involve rendering templates or formatting JSON responses)

- The Model (`userModel`) might have methods to retrieve user data from a database.
- The Controller (`userController`) handles the HTTP request, interacts with the Model, and decides which View to render.
- The View could be a template engine that renders the user profile page with the retrieved user data.

By following the MVC pattern, the code is organized, modular, and easier to maintain, making it a widely adopted architectural pattern in backend development.

Query Documents From The Database:-

```
const Movie = require('../Models/movieModel');

exports.getAllMovies = async (req, res) => {
  try{
    const movies = await Movie.find();

    res.status(200).json({
      status: 'success',
      length: movies.length,
      data: {
        movies
      }
    });
  }catch(err){
  }
}
```

Whenever we have to query the document from the database collection, we have to (.find()) method.

This async and await is the part of the syntax.

If we do not provide anything inside (.find()) then (.find()) will return the whole document.

UPDATE document from Express:-

If we have to update the document on the basis of their id:-

```

exports.updateMovie = async (req, res) => {
  try{
    const updatedMovie = await Movie.findByIdAndUpdate(req.params.id, req.body, {new: true, runValidators: true});

    res.status(200).json({
      status: "success",
      data: {
        movie: updatedMovie
      }
    });
  }catch(err){
    res.status(404).json({
      status: "fail",
      message: err.message
    });
  }
}

```

Here, we have used (.findByIdAndUpdate()) function which returns a promise, in this function(.findByIdAndUpdate()) there are 3 arguments present, the 1st argument is the document id that we have to update and 2nd argument will tell the updated document, and 3rd argument is optional.

DELETE document from Express:-

```

exports.deleteMovie = async (req, res) => {
  try{
    await Movie.findByIdAndDelete(req.params.id);

    res.status(204).json({
      status: 'success',
      data: null
    });
  }catch(err){
    res.status(404).json({
      status: "fail",
      message: err.message
    });
  }
}

```


If we have to delete a document from the database collection then we would use `findByIdAndDelete()` and this we only have to pass one argument and that is id which we want to delete.

Filtering:-

We can filter the documents according to some parameters.

```
exports.getAllMovies = async (req, res) => {
  try{
    console.log(req.query);
    const movies = await Movie.find(req.query);

    // const movies = await Movie.find()
    //   .where('duration')
    //   .equals(req.query.duration)
    //   .where('ratings')
    //   .equals(req.query.ratings);

    res.status(200).json({
      status: 'success',
      length: movies.length,
      data: {
        movies
      }
    });
  }
}
```

Here, the commented code can also be implemented to query the document.

Sorting:-

```
//SORTING LOGIC
if(req.query.sort){
  const sortBy = req.query.sort.split(',').join(' ');
  query = query.sort(sortBy);
}else{
  query = query.sort('-createdAt');
}

const movies = await query;
// const movies = await Movie.find()
//   .where('duration')
//   .gte(req.query.duration)
//   .where('ratings')
//   .gte(req.query.ratings)
//   .where('price')
//   .lte(req.query.price)

res.status(200).json({
  status: 'success',
```

Limiting fields:-

Specifying the query and which field we are going to have in our document.

Let's say we have given this:-



So limiting can be applied like this:-

```
//LIMITING FIELDS
if(req.query.fields){
  //query.select('name duration price ratings')
  const fields = req.query.fields.split(',').join(' ');
  console.log(fields);
  query = query.select(fields);
}else{
  query = query.select('-_id');
}
}
```

Pagination:-

Specifying how much content does the single page will contain.

```
//PAGINATION
const page = req.query.page*1 || 1;
const limit = req.query.limit*1 || 10;
//PAGE 1: 1 - 10; PAGE 2: 11 - 20; PAGE 3: 21 - 30
const skip = (page -1) * limit;
query = query.skip(skip).limit(limit);

if(req.query.page){
  const moviesCount = Movie.countDocuments();
  if(skip >= moviesCount){
    throw new Error("This page is not found!");
  }
}
```

Aggregation Pipeline:-

the aggregation pipeline is a powerful framework for data processing and transformation. It allows you to perform a sequence of data processing operations on a collection of documents. Aggregation pipelines are used to filter, transform, and analyze data before returning the results.

The aggregation pipeline consists of a series of stages, where each stage performs a specific operation on the input documents and passes the results to the next stage. Each stage in the pipeline is represented by an object, and the order of the stages determines the order of execution.

Here's a high-level overview of some common stages used in the MongoDB aggregation pipeline:

\$match:

- Filters the documents based on specified criteria.
- It is similar to the `find` method but operates within the aggregation pipeline.

Code:-

```
db.collection.aggregate([  
  { $match: { status: "A" } }  
]);
```

here, db.collection is just the name of collection from where we are fetching the document.

\$group:

- Groups documents based on specified key(s) and performs aggregation operations on the grouped data (e.g., sum, average, count).

Code:-

```
db.collection.aggregate([  
  { $group: { _id: "$category", total: { $sum: "$quantity" } } }  
]);
```

\$project:

- Reshapes documents by specifying the inclusion or exclusion of fields, renaming fields, or creating computed fields.

Code:-

```
db.collection.aggregate([  
  { $project: { name: 1, price: 1, discountedPrice: { $subtract:  
    ["$price", "$discount"] } } }  
]);
```

\$sort:

- Sorts the documents based on specified criteria.

code:-

```
db.collection.aggregate([  
  { $sort: { price: 1 } }  
]);
```

\$unwind:

- Deconstructs an array field, creating a new document for each array element.

Code:-

```
db.collection.aggregate([  
  { $unwind: "$tags" }  
]);
```

\$lookup:

- Performs a left outer join between two collections to retrieve documents from the joined collection.

Code:-

```
db.orders.aggregate([  
  {  
    $lookup: {
```

```
    from: "products",
    localField: "productId",
    foreignField: "_id",
    as: "product"
  }
}
]);
```

These stages, along with others, can be combined to create complex data processing pipelines that suit the requirements of your application. Aggregation pipelines are especially useful for tasks like reporting, analytics, and transforming data before presenting it to users.

For ex:-

```
db.sales.aggregate([
  { $match: { date: { $gte: ISODate("2023-01-01"), $lt:
ISODate("2023-02-01") } } },
  { $group: { _id: "$product", totalSales: { $sum: "$quantity" }
} },
  { $sort: { totalSales: -1 } },
  { $limit: 5 }
]);
```

This pipeline:

1. Matches documents with a specific date range.
2. Groups the documents by product.
3. Calculates the total sales for each product.

4. Sorts the results in descending order based on total sales.
5. Limits the output to the top 5 products.

Virtual Properties:-

virtual properties are properties that you can define on a document's schema but are not persisted to the database. They are computed properties that are derived from the existing data in the document or from other sources. Virtual properties can be useful for presenting data in a certain way, performing calculations, or transforming data before it is returned to the application.

To implement virtual properties in MongoDB, you can use the Mongoose library, which is an ODM (Object-Document Mapper) for MongoDB and Node.js. Mongoose allows you to define virtuals in your schema.

Here's an example of using virtual properties in a Mongoose schema:

```
const mongoose = require('mongoose');

// Define a Mongoose schema
const userSchema = new mongoose.Schema({
  firstName: { type: String, required: true },
  lastName: { type: String, required: true },
});

// Define a virtual property for the full name
userSchema.virtual('fullName').get(function () {
  return `${this.firstName} ${this.lastName}`;
});

// Create a Mongoose model using the schema
const User = mongoose.model('User', userSchema);

// Example usage
const user = new User({ firstName: 'John', lastName: 'Doe' });
console.log(user.fullName); // Output: John Doe
```

In this example:

1. We define a Mongoose schema with two fields: `firstName` and `lastName`.

2. We create a virtual property called `fullName` using the `virtual` method. The `get` function is used to define how the virtual property is computed.
3. When we create a new user instance and access the `fullName` property, Mongoose invokes the `get` function to compute the value based on the `firstName` and `lastName` fields.

Note that virtual properties are not stored in the database; they are computed on the fly when you access them. This makes them useful for scenarios where you want to present data in a specific format without modifying the underlying stored data.

Virtual properties can also be used for other purposes, such as formatting dates, performing calculations, or creating composite values. They provide a way to encapsulate certain logic related to the presentation of data without affecting the actual data stored in the database.

Mongoose Middleware:-

Just like express middleware's mongoose also has some middleware's;

Document Middleware:-

document middleware (also known as "pre" and "post" hooks) allows you to attach functions that execute before or after certain operations on a Mongoose document. These middleware functions are useful for performing additional logic or modifications to the document before or after it is saved, updated, or removed from the database.

Here are some key points about document middleware in Mongoose:

1. **Pre and Post Hooks:**

- Mongoose supports two types of document middleware: "pre" (before) and "post" (after) hooks.
- "pre" hooks are executed before a specified operation (e.g., `save`, `update`, `remove`), while "post" hooks are executed after the operation is completed.

2. **Operations with Middleware:**

- Document middleware can be applied to various operations, including `init`, `validate`, `save`, `remove`, and custom methods.

3. **Access to this:**

- Inside a document middleware function, the `this` keyword refers to the document being processed. This allows you to access and modify the document's properties.

4. Error Handling:

- You can use the `next` function to proceed with the operation or, in the case of "pre" middleware, to skip the operation and move on to the next middleware in the stack.
- If an error occurs, you can pass an error to `next`, and Mongoose will handle it accordingly.

Here's a simple example of using "pre" and "post" hooks in Mongoose:

Code:-

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

// Define a Mongoose schema
const userSchema = new Schema({
  username: String,
  email: String,
  password: String,
});

// "pre" middleware before saving a user
userSchema.pre('save', function (next) {
  // Do something before saving (e.g., hash the password)
  // Access document properties using `this`
  console.log(`Saving user: ${this.username}`);
  // Assuming there is a method to hash the password
  hashPassword(this.password)
    .then((hashedPassword) => {
      this.password = hashedPassword;
      next();
    })
    .catch((error) => next(error));
});
```

```
// "post" middleware after saving a user
userSchema.post('save', function (doc, next) {
  // Do something after saving (e.g., log the saved user)
  console.log(`User saved: ${doc.username}`);
  next();
});

// Create a Mongoose model using the schema
const User = mongoose.model('User', userSchema);

// Example usage
const user = new User({ username: 'john_doe', email: 'john@example.com' });
user.save()
  .then(() => {
    // Document has been saved
  })
  .catch((error) => {
    // Handle save error
  });
```

In this example, the "pre" middleware is used before saving a user to hash the password, and the "post" middleware is used after saving to log information about the saved user. Keep in mind that the hashing function (`hashPassword`) is just an example, and you would typically use a secure hashing library for password hashing in a real-world scenario.

Query Middleware:-

query middleware allows you to attach functions that execute before or after a certain query is executed on a Mongoose model. This type of middleware is helpful for performing additional operations or modifications to the query before it is sent to the database or after the database responds.

Query middleware includes "pre" hooks (executed before a query) and "post" hooks (executed after a query). These hooks can be used to modify the query conditions, add additional filters, or perform other tasks related to the query execution.

Here are some key points about query middleware in Mongoose:

1. **Pre and Post Hooks:**

- "Pre" hooks are executed before a query is executed, and "post" hooks are executed after the query has been executed.

2. Query Methods:

- Query middleware can be applied to various query methods such as `find`, `findOne`, `update`, `deleteOne`, `deleteMany`, and custom methods.

3. Access to this:

- Inside a query middleware function, the `this` keyword refers to the query being processed. This allows you to access and modify the query conditions.

4. Error Handling:

- As with document middleware, you can use the `next` function to proceed with the query or, in the case of "pre" middleware, to skip the query and move on to the next middleware in the stack.
- If an error occurs, you can pass an error to `next`, and Mongoose will handle it accordingly.

Here's a simple example of using "pre" and "post" hooks in Mongoose for the `find` method:


code:-

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

// Define a Mongoose schema
const userSchema = new Schema({
  username: String,
  email: String,
  password: String,
});

// "pre" middleware before executing a 'find' query
userSchema.pre('find', function (next) {
  // Modify the conditions of the query
  this.where({ active: true });
  next();
});

// "post" middleware after executing a 'find' query
userSchema.post('find', function (docs, next) {
  // Do something after the query (e.g., log the found documents)
  console.log(`Found ${docs.length} users`);
  next();
});
```



```
// Create a Mongoose model using the schema
const User = mongoose.model('User', userSchema);

// Example usage
User.find({ username: 'john_doe' })
  .then((users) => {
    // Do something with the found users
  })
  .catch((error) => {
    // Handle query error
  });
```

In this example, the "pre" middleware is used to add a condition to the `find` query to only retrieve active users. The "post" middleware logs information about the found documents.

Aggregation Middleware:-

aggregation middleware allows you to attach functions that execute before or after a certain aggregation operation is executed on a Mongoose model. Aggregation middleware is useful for performing additional operations or modifications to the aggregation pipeline before it is sent to the MongoDB server or after the aggregation results are received.

Aggregation middleware includes "pre" hooks (executed before an aggregation operation) and "post" hooks (executed after an aggregation operation). These hooks can be used to modify the aggregation pipeline, add additional stages, or perform other tasks related to the aggregation.

Here are some key points about aggregation middleware in Mongoose:

1. **Pre and Post Hooks:**

- "Pre" hooks are executed before an aggregation operation is executed, and "post" hooks are executed after the aggregation operation has been executed.

2. **Aggregation Operations:**

- Aggregation middleware can be applied to various aggregation operations such as `aggregate`, `findOne`, and custom methods.

3. **Access to this:**

- Inside an aggregation middleware function, the `this` keyword refers to the aggregation object being processed. This allows you to access and modify the aggregation pipeline.

4. **Error Handling:**

- As with document and query middleware, you can use the `next` function to proceed with the aggregation or, in the case of "pre" middleware, to skip the aggregation and move on to the next middleware in the stack.
- If an error occurs, you can pass an error to `next`, and Mongoose will handle it accordingly.

Here's a simple example of using "pre" and "post" hooks in Mongoose for the `aggregate` method:

Code:-

```
const mongoose = require('mongoose');
```

```
const Schema = mongoose.Schema;
```

```
// Define a Mongoose schema
```

```
const userSchema = new Schema({
```

```
  username: String,
```

```
  email: String,
```

```
  password: String,
```

```
  age: Number,
```

```
});
```

```
// "pre" middleware before executing an 'aggregate'  
operation
```

```
userSchema.pre('aggregate', function (next) {
```

```
// Modify the aggregation pipeline
this.pipeline().unshift({ $match: { age: { $gte: 18 } } });
next();
});

// "post" middleware after executing an 'aggregate' operation
userSchema.post('aggregate', function (result, next) {
  // Do something after the aggregation (e.g., log the results)
  console.log(`Aggregation result: ${JSON.stringify(result)}`);
  next();
});

// Create a Mongoose model using the schema
const User = mongoose.model('User', userSchema);

// Example usage
User.aggregate([
  { $group: { _id: '$age', count: { $sum: 1 } } }
])
.then((result) => {
  // Do something with the aggregation result
})
```



```
.catch((error) => {  
  
  // Handle aggregation error  
  
});
```

the "pre" middleware is used to add a `$match` stage to the aggregation pipeline to filter users with an age greater than or equal to 18. The "post" middleware logs information about the aggregation results.

Data Validators:-

In an Express.js application using a framework like Mongoose for MongoDB integration, you can use built-in and custom validators in the schema definition to validate the data before it's stored in the database. Here's an example that demonstrates both built-in and custom validators in a Mongoose schema:

Code:-

```
const mongoose = require('mongoose');  
  
const Schema = mongoose.Schema;  
  
const validator= require('validator');  
  
// Define a Mongoose schema with built-in and custom  
validators  
  
const userSchema = new Schema({  
  username: {  
    type: String,  
    required: true,  
    minlength: [5, 'Username must be at least 5 characters  
long'],  
    maxlength: [20, 'Username cannot exceed 20 characters'],
```

```
validate: {
  validator: (value) => /^[a-zA-Z0-9]+$/.test(value),
  message: 'Username can only contain letters and
numbers',
},
},
email: {
  type: String,
  required: true,
  unique: true,
  validate: {
    validator: (value) => /\S+@\S+\.\S+/.test(value),
    message: 'Invalid email address',
  },
},
password: {
  type: String,
  required: true,
  validate: {
    validator: (value) => value.length >= 8,
    message: 'Password must be at least 8 characters long',
  },
}
```

```
    },  
  });  
  
  // Create a Mongoose model using the schema  
  const User = mongoose.model('User', userSchema);  
  
  // Example usage  
  const newUser = new User({  
    username: 'john_doe123',  
    email: 'john.doe@example.com',  
    password: 'password123',  
  });  
  
  newUser.save()  
    .then(() => {  
      console.log('User saved successfully');  
    })  
    .catch((error) => {  
      console.error('Error saving user:', error.message);  
    });
```

In this example:

- **Built-in Validators:**

- **required**: Ensures that the field is present.
- **minlength** and **maxlength**: Define the minimum and maximum length of a string field.
- **unique**: Ensures that the value is unique within the collection.
- **Custom Validators**:
 - **validate**: Allows you to define a custom validation function. It takes an object with a **validator** function and a **message** property for the error message.

The custom validators use regular expressions to validate the **username** and **email** fields. You can customize these validators based on your specific requirements.

Remember that the validation occurs at the schema level, and when you attempt to save a document with invalid data, Mongoose will reject the save operation, and you can handle the validation errors accordingly.

Feel free to adapt the validators and error messages based on your application's validation needs.

ERROR HANDLING:-

Operation Errors

Operational errors are the problems that we can predict that will happen at some point in future. We need to handle them in advance.

- User trying to access an invalid route.
- Inputting invalid data.
- Application failed to connect to server.
- Request timeout etc.

Programming Errors

Programming errors are simply bugs that we programmers, by mistake, introduces them in our code.

- Trying to read property of an undefined variable.
- Using await without async.
- Accidentally using req.query instead of req.body
- Passing a number where an object is expected etc.

In express we generally deals with operational errors.

So we global error handling error to catch all the errors and handles them accordingly.

Global Error Handling Middleware :-

```
exports.getMovie = async (req, res) => {
  try{
    //const movie = await Movie.findOne({_id: req.params.id});
    const movie = await Movie.findById(req.params.id);

    res.status(200).json({
      status: 'success',
      data: {
        movie
      }
    });
  }catch(err){
    res.status(404).json({
      status: 'fail',
      message: err.message
    });
  }
}
```

here we are using catch to find the errors , but when we use global error handling middleware then we do not have to implement catch errors individually.

Syntax:-

```
app.use((error, req, res, next) => {
  //
})
```

Here we have passed the middleware function inside app.use() and in the function, we have passed 4 arguments in which 3 are of the middleware arguments and the 1st one is the extra one for error.

```

app.use((error, req, res, next) => {
  error.statusCode = error.statusCode || 500;
  error.status = error.status || 'error';
  res.status(error.statusCode).json({
    status: error.statusCode,
    message: error.message
  });
});

```

Created an global error handling middleware.

Now, we have to use it(middleware), 1st of all we have to use this (below picture) if we to use the middleware of error handling.

```

const err = new Error("");

```

Here ERROR is the built in function provided by javascript.

Now,

```

app.use('/api/v1/movies', moviesRouter);
app.all('*', (req, res, next) => {
  // res.status(404).json({
  //   status: 'fail',
  //   message: `Can't find ${req.originalUrl} on the server!`
  // });
  const err = new Error(`Can't find ${req.originalUrl} on the server!`);
  err.status = 'fail';
  err.statusCode = 404;

  next(err);
});

app.use((error, req, res, next) => {
  error.statusCode = error.statusCode || 500;
  error.status = error.status || 'error';
  res.status(error.statusCode).json({
    status: error.statusCode,
    message: error.message
  });
});

```

Here we have created an error object, and by using next we are passing an error, so if we pass anything inside next then , express will skip all the middleware and reached to global handling middleware.

Creating a Custom Error Handling class:-

Instead of using this :-

```
const err = new Error("");
```

We create own Error function:-

```
class CustomError extends Error{
  constructor(message, statusCode){
    super()
  }
}
```

Here this **super** is calling constructor of the **Error** class.

```
class CustomError extends Error{
  constructor(message, statusCode){
    super(message);
    this.statusCode = statusCode;
    this.status = statusCode >= 400 && statusCode < 500 ? 'fail' : 'error';

    this.isOperational = true;

    Error.captureStackTrace(this, this.constructor);
  }
}

module.exports = CustomError;
```


captureStackTrace() will tell exactly where the error exactly occurred. Here, we have created our own Error function.

And we can use it by importing , and then write this:-

```
const err = new CustomError('Can't find ${req.originalUrl} on the server!', 404);
```

Error Handling in Async Function:-

```
const asyncErrorHandler = (func) => {  
  return (req, res, next) => {  
    func(req, res, next).catch(err => next(err));  
  }  
}  
  
exports.createMovie = asyncErrorHandler(async (req, res) => {  
  const movie = await Movie.create(req.body);  
  res.status(201).json({  
    status: 'success',  
    data: {  
      movie  
    }  
  })  
});
```

Instead of writing the catch method, we have created a global async error handling function in order to catch all the errors which going to be written inside it.

AUTHORIZATION AND AUTHENTICATION

First of all create a userModel.js script file separately.

```
const validator = require('validator');

//name, email, password, confirmPassword, photo
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, 'Please enter your name.']
  },
  email: {
    type: String,
    required: [true, 'Please enter an email.'],
    unique: true,
    lowercase: true,
    validate: [validator.isEmail, 'Please enter a valid email.']
  },
  photo: String,
  password: {
    type: String,
    required: [true, 'Please enter a password.'],
    minlength: 8
  },
  confirmPassword: {
    type: String,
    required: [true, 'Please confi']
  }
})
```

```
    type: String,
    required: [true, 'Please enter a password.'],
    minlength: 8
  },
  confirmPassword: {
    type: String,
    required: [true, 'Please confirm your password.']
  }
})

const User = mongoose.model('User', userSchema);

module.exports = User;
```

Now, create New Route for the User.

```
app.use('/api/v1/users', authRouter);
```

Create a new file named authRouter.js and require this page where you are using the authRouter route.

Also, create a file like authcontroller.js for authRouter handler function[all handler function is a middleware].

AuthController.js:-

```
const User = require('../Models/userModel');
const asyncErrorHandler = require('../Utils/asyncErrorHandler');

exports.signup = asyncErrorHandler(async (req, res, next) => {
  const newUser = await User.create(req.body);

  res.status(201).json({
    status: 'success',
    data: {
      user: newUser
    }
  });
});
```

The asyncErrorHandler function is the global error handling middleware.

AuthRoutr.js:-

```
const express = require('express');
const authController = require('../Controllers/authController')

const router = express.Router();

router.route('/signup').post(authController.signup);

module.exports = router;
```

Now, we will Encrypt Password.

We will add one more property to our userModel.

```
confirmPassword: {  
  type: String,  
  required: [true, 'Please confirm your password.'],  
  validate: {  
    validator: function(val){  
      return val == this.password;  
    },  
    message: 'Password & Confirm Password does not match!'  
  }  
}
```

To encrypt the password we are going to use bcryptjs file so first we will install and require it.

```
const bcrypt = require('bcryptjs');
```

Now further we will use mongoose middle to encrypt the password.

```
userSchema.pre('save', async function(next) {  
  if(!this.isModified('password')) return next();  
  
  //encrypt the password before saving it  
  this.password = await bcrypt.hash(this.password, 12);  
  
  this.confirmPassword = undefined;  
  next();  
})
```

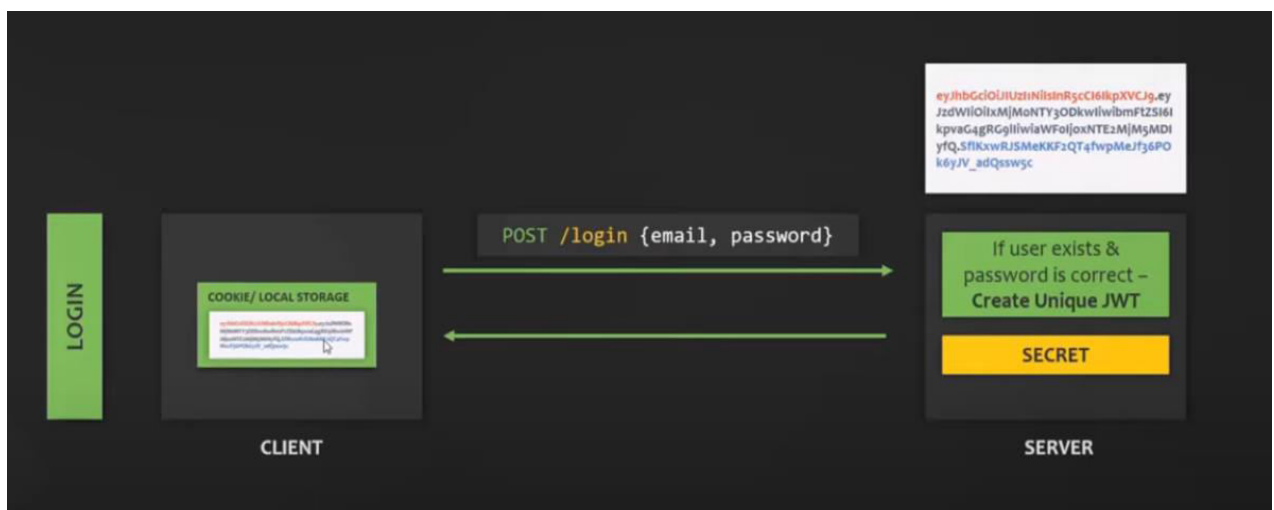
Here, bcrypt.hash() is used to encrypt the data, and in the 1st argument we provide which data we want to

encrypt, and in the 2nd argument we provide salting no. (No. of random strings attached to the password so that no two passwords have the same encryption).

Now, we will separate login and non-login users to interact with certain routes.

There many ways to achive this but we will use , JWT Authentication.

At the time of login:-



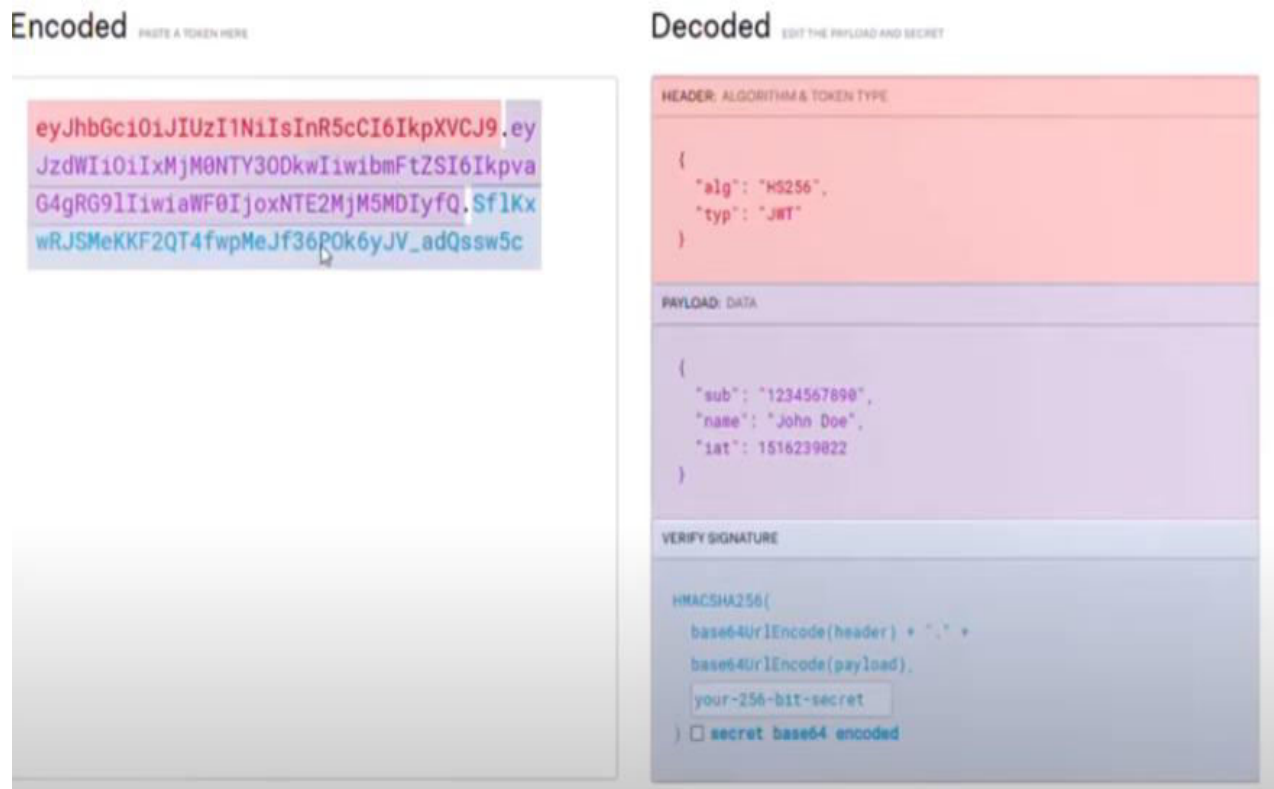
here, when we made a post request for login so if email and password is correct then a JWT token is created [white box] and a secret key is created then this JWT token is sendd back to client which is stored under cookie/local storage.

Now, once the user is logged in , the user tries to use the protected routes.



Whenever a user tries to access the protected routes, it sends the get request to access, along with the request, the JWT token will also be sent as proof of logged-in user, so at the server side JWT token is being verified and if verified , the sent back the requested data to the client.

JWT token is made up of 3 parts:-



1) Header 2) Payload 3) verify signature.

JWT, or JSON Web Token, is a compact and self-contained way to represent information between two parties. It is commonly used for authentication and authorization in web development. A JWT token consists of three parts separated by dots (.), and each part serves a specific purpose:

1. **Header:** The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA. It is Base64Url encoded to form the first part of the JWT.

Example Header:

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```


After Base64Url encoding, it becomes something like:

eyJhbGciOiAiSFMyNTYiLCJhdHlwIjogIkpXVCJ9

Payload: The second part of the JWT is the payload. The payload contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: registered, public, and private claims.

Example Payload:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

After Base64Url encoding, it becomes something like:

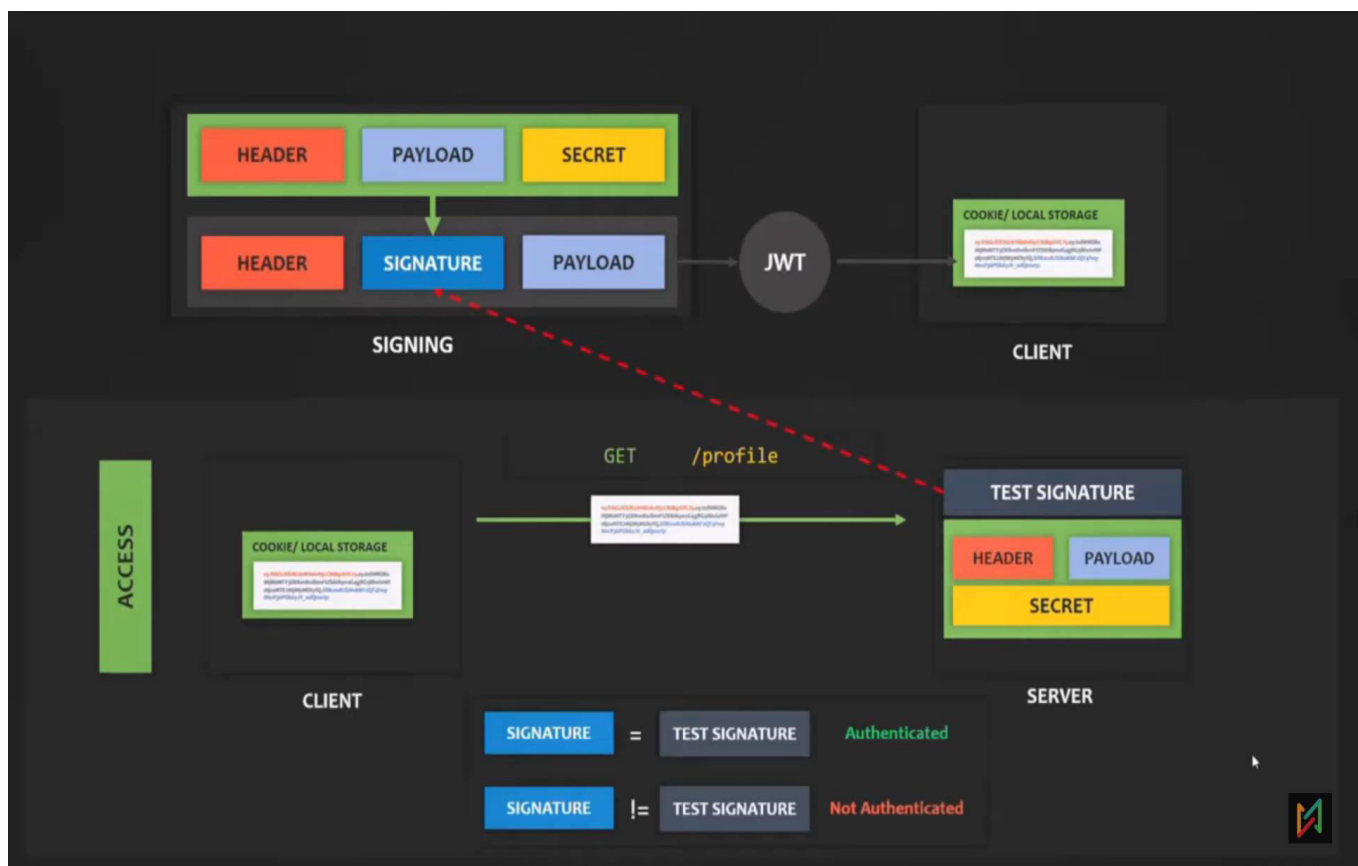
eyJzdWIiOiAiMTIzNDU2Nzg5MCIsICJuYW1lIjogIkpvaG4gRG9lIiwgaWQiOiJhdCI6IDE1MTYyMzkwMjJ9

Signature: To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that. The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.

Example Signature (using the HMAC SHA256 algorithm):

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
)
```

The final JWT is the concatenation of these three encoded parts, separated by dots (.). When a JWT is sent over the network or stored, it can be decoded and the information in the payload can be retrieved. The signature is used to verify that the token has not been tampered with.



Now, we will implement JWT token.

`jsonwebtoken` to implement JSON Web Tokens (JWT). Below is a simple example of how you can create and verify JWTs in an Express.js application.

First, you'll need to install the `jsonwebtoken` library:

```
npm install jsonwebtoken
```

Now, let's create a basic Express.js server with JWT implementation:

```
const express = require('express');

const jwt = require('jsonwebtoken');


const app = express();

const secretKey = 'your-secret-key'; // Replace with a secure secret key in a real
application


// Middleware to parse JSON requests

app.use(express.json());


// Route to generate a JWT

app.post('/login', (req, res) => {

  // Assuming you have a user authentication process

  const { username, password } = req.body;


  // Verify user credentials (This is a simplified example; in a real app, you would
  check against a database)

  if (username === 'demo' && password === 'password') {

    // Create a JWT token with the user information

    const token = jwt.sign({ username }, secretKey, { expiresIn: '1h' });


    // Send the token in the response

    res.json({ token });
```

```
} else {  
  res.status(401).json({ message: 'Invalid credentials' });  
}  
});  
  
// Route to access a protected resource using JWT  
app.get('/protected', authenticateToken, (req, res) => {  
  res.json({ message: 'You have access to this protected resource!' });  
});  
  
// Middleware to authenticate JWT token  
function authenticateToken(req, res, next) {  
  const token = req.header('Authorization');  
  
  if (!token) {  
    return res.status(401).json({ message: 'Unauthorized: Token missing' });  
  }  
  
  jwt.verify(token, secretKey, (err, user) => {  
    if (err) {  
      return res.status(403).json({ message: 'Forbidden: Invalid token' });  
    }  
  })  
}
```

```

// Attach the user information to the request for further processing

req.user = user;

// Continue to the next middleware or route handler

next();

});

}

// Start the Express server

const port = 3000;

app.listen(port, () => {

  console.log(`Server is running on http://localhost:${port}`);

});

```

1. **Creating a JWT (Login Route):**

- When a user logs in (`/login` route), you can create a JWT using `jwt.sign()`. The token includes the user information (in this case, just the username).
- The token is then sent as a JSON response.

2. **Protecting a Route with JWT (Protected Route):**

- The `/protected` route is protected using the `authenticateToken` middleware.
- The middleware checks if a valid JWT is present in the `Authorization` header.
- If the token is valid, the user information is attached to the request (`req.user`), and the request is allowed to proceed to the protected route.

3. **JWT Verification Middleware (`authenticateToken`):**

- This middleware extracts the JWT from the `Authorization` header.
- It then verifies the token using `jwt.verify()` and the secret key.
- If the verification is successful, the user information is attached to the request, allowing the request to proceed.

- If the verification fails, the middleware sends an error response.

This is a basic example, and in a real-world application, you would likely store more information in the JWT, handle token expiration, and use a secure way to store the secret key. Additionally, user authentication would involve more robust mechanisms, such as checking against a database.