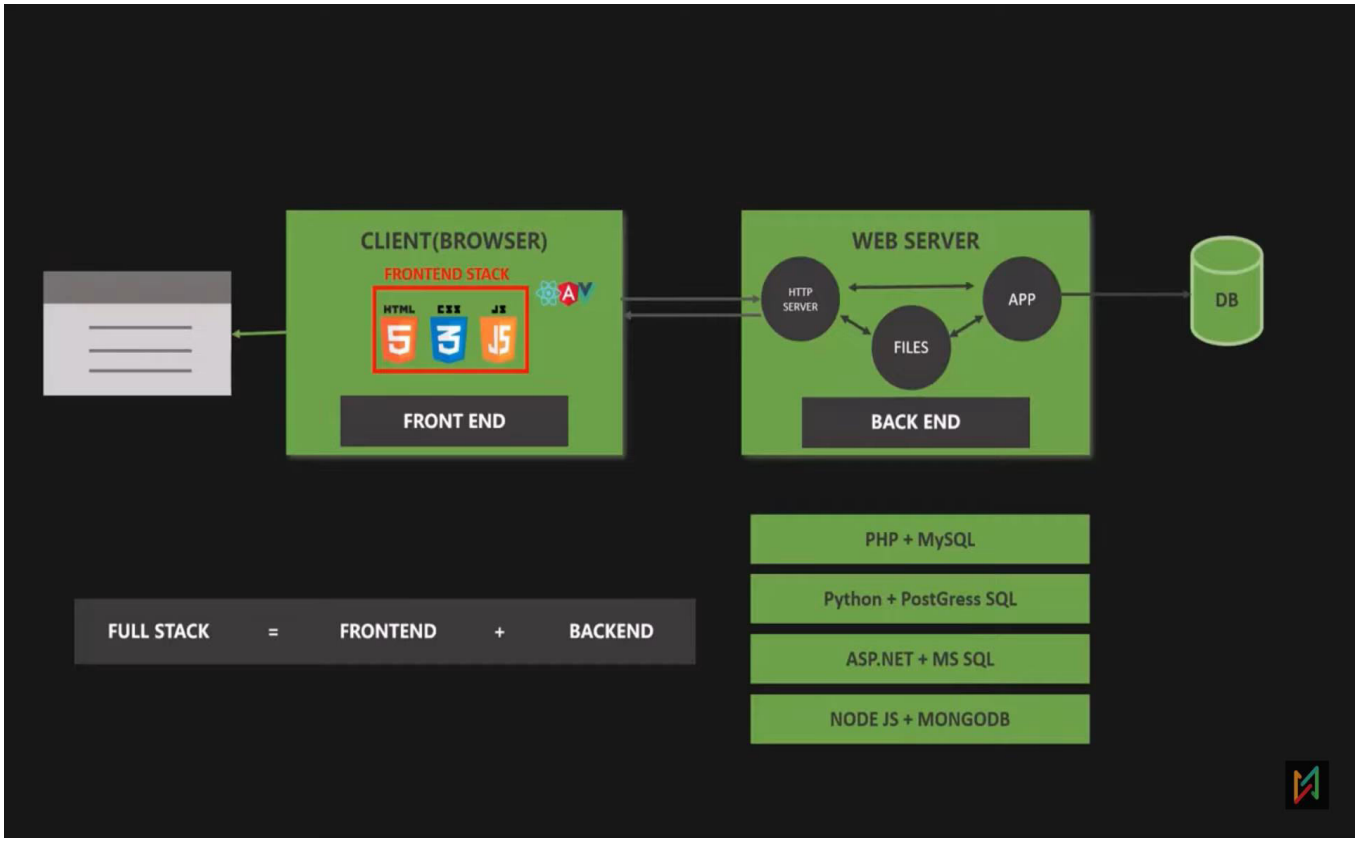
The complete guide for MERN stack



FRONTEND

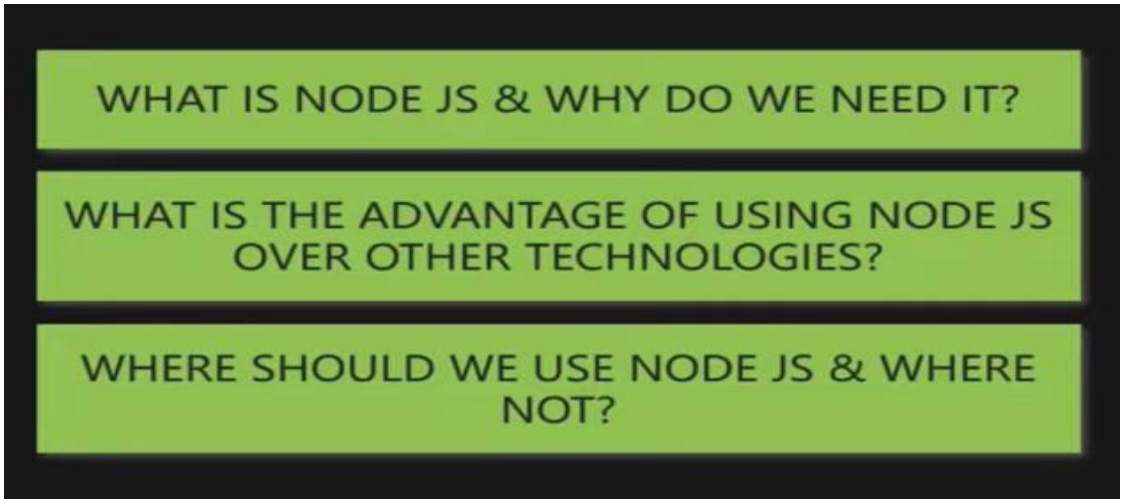
It is just html+css+js or react and nothing else;

BACKEND

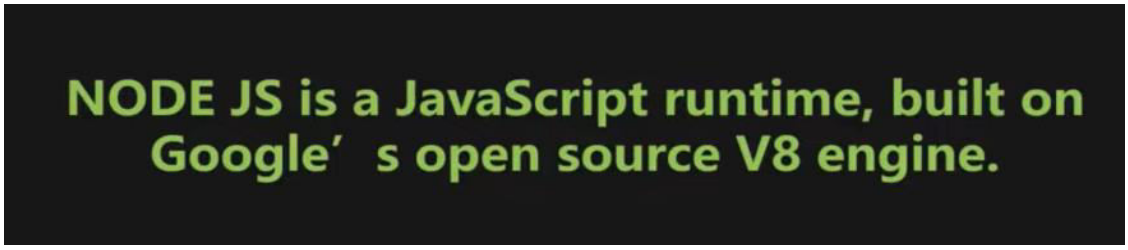
Whenever we create a mern stack project, we follow certain steps:-

1) **NODE.JS**:-

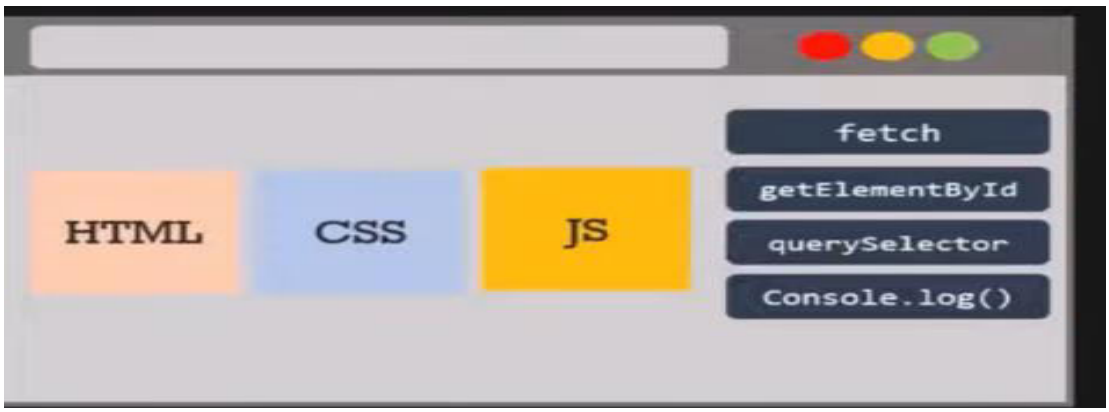
First of all, we have to answer some of the following questions:-



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 :-



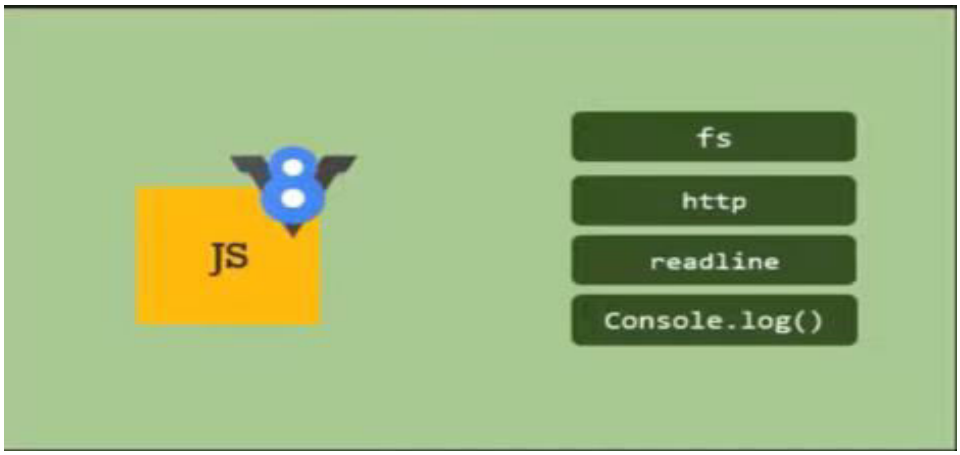
These fetch and getElementById are the ex of API and functions provided by the browser when js is running in the browser.

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

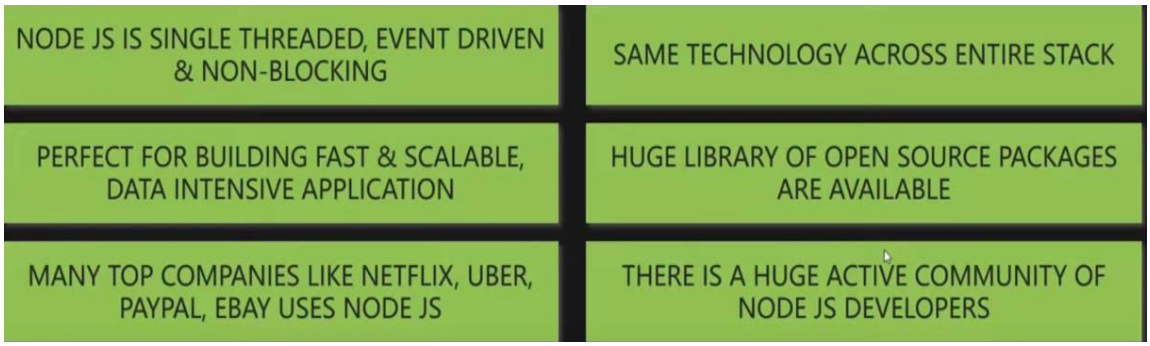
When the V8 engine is embedded in C++ language to run javascript code outside the browser then this is termed node.js.

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:-

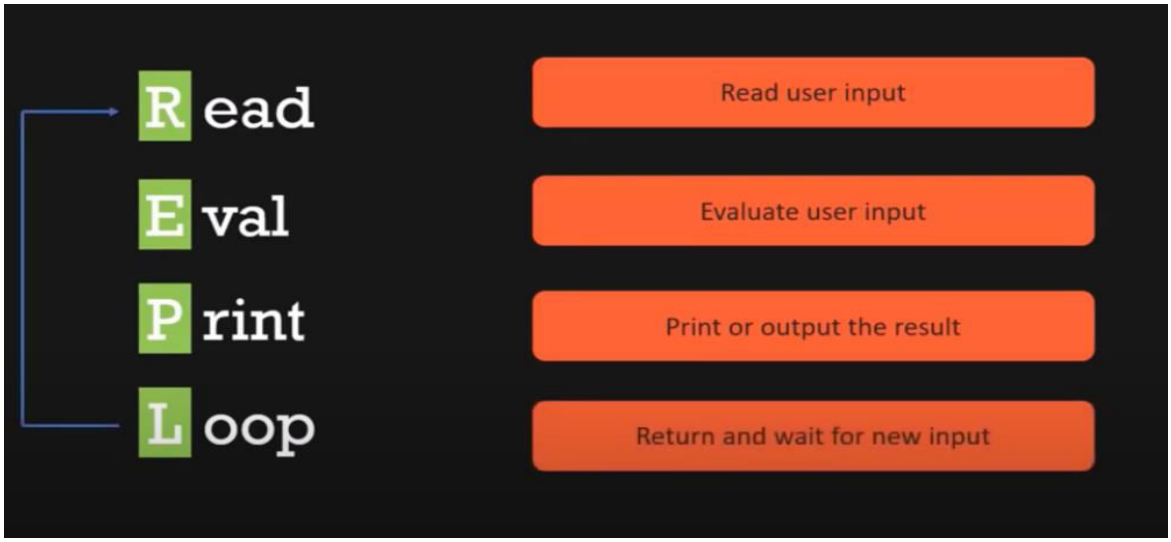


The advantages of using node.js:-



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.

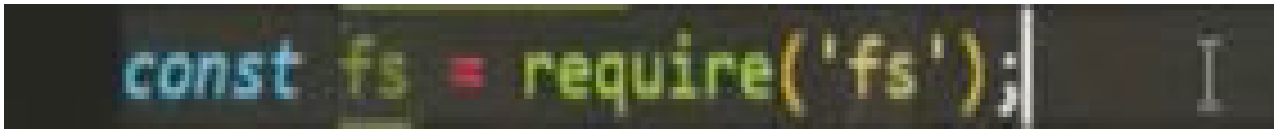
INPUT AND OUTPUT IN NODE.JS TERMINAL:-

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

```
const readline = require('readline');
const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

rl.question("Please enter your naame: ", (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.




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

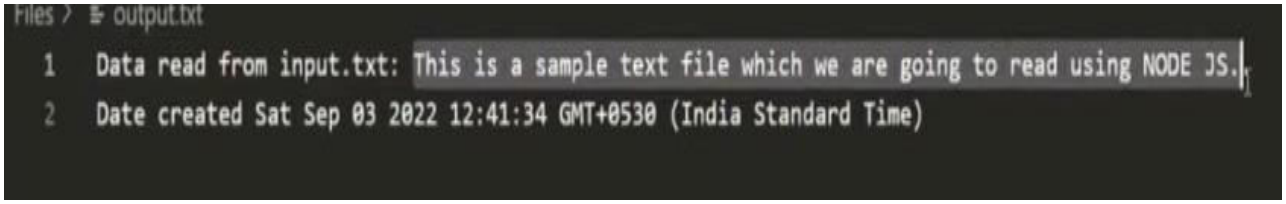
Here, this readFileSync function is used to read files synchronously and it has two arguments that need to be provided, 1st is the path of the file that needs to be studied.

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);
```

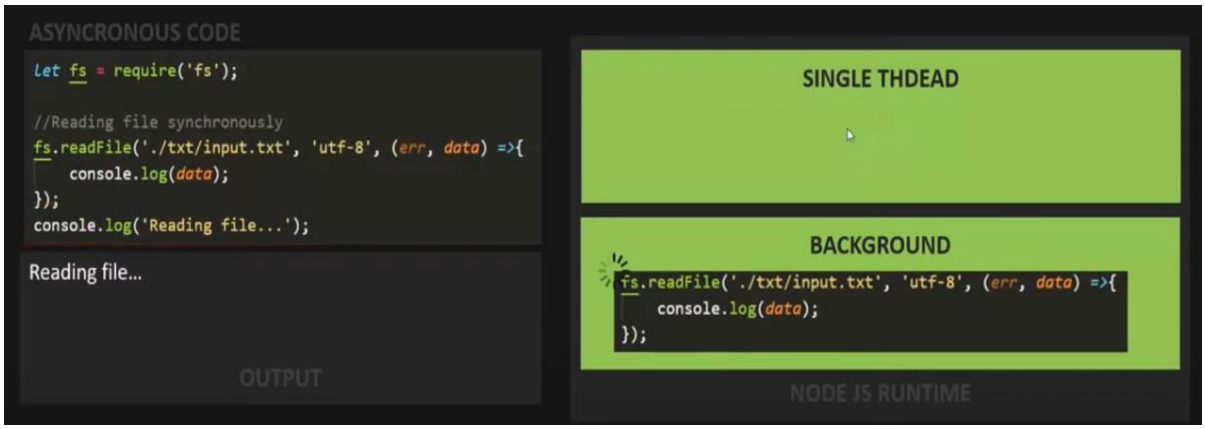
Here, writeFileSync is used to write content inside the file,

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.



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.

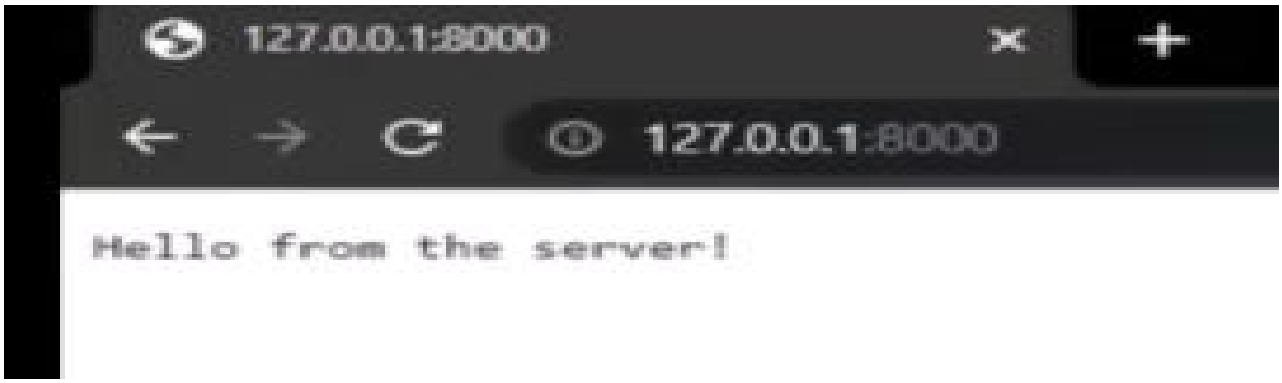
The request contains every detail about the server.

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:-

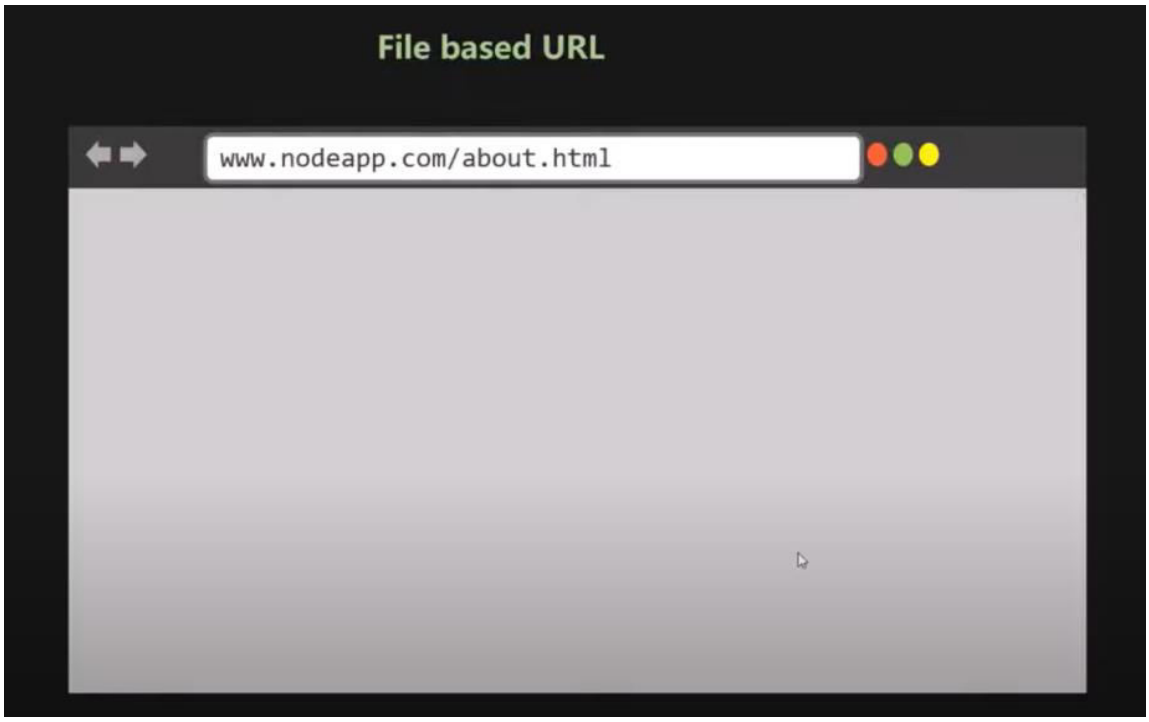
```
const http = require('http');
const server = http.createServer((request, response) => {
  // Handle the incoming HTTP request

  // 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:-

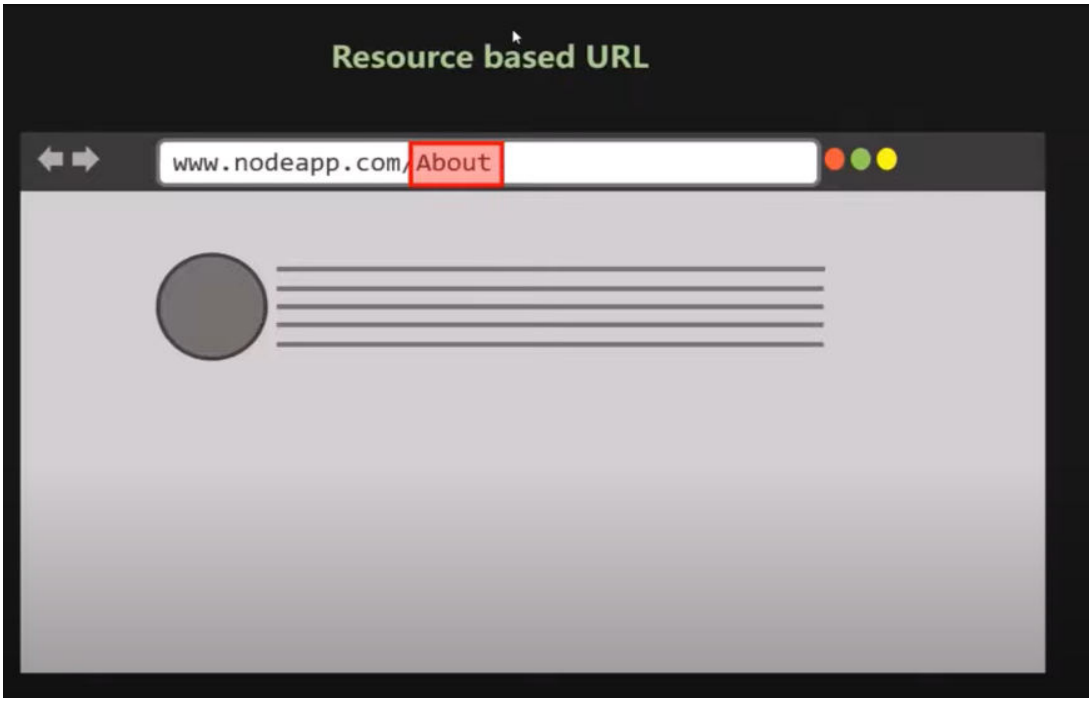
Routing defines the way in which the client requests are handled by the application endpoints

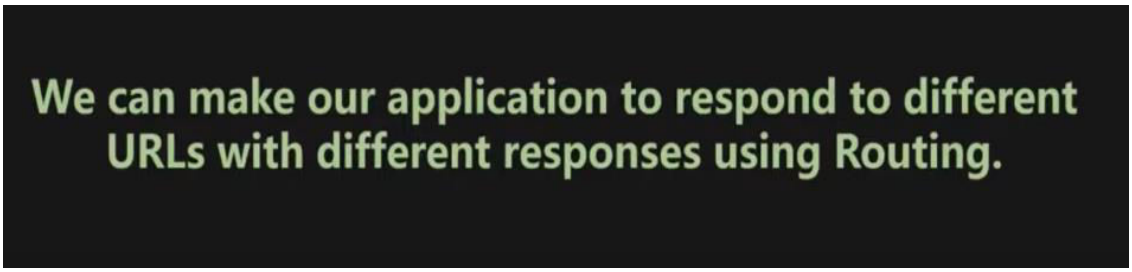
There are two types of URL:-

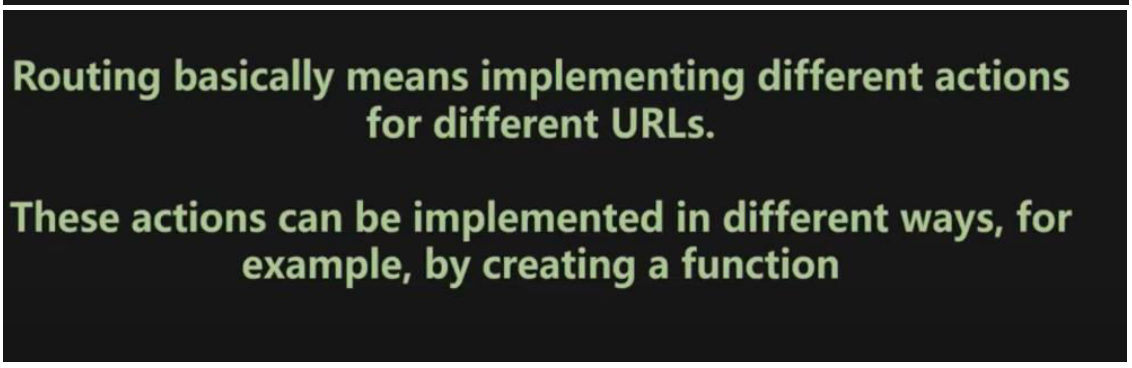
1) File-based URL:- search on the basis of the name of the file and render the page to the browser.



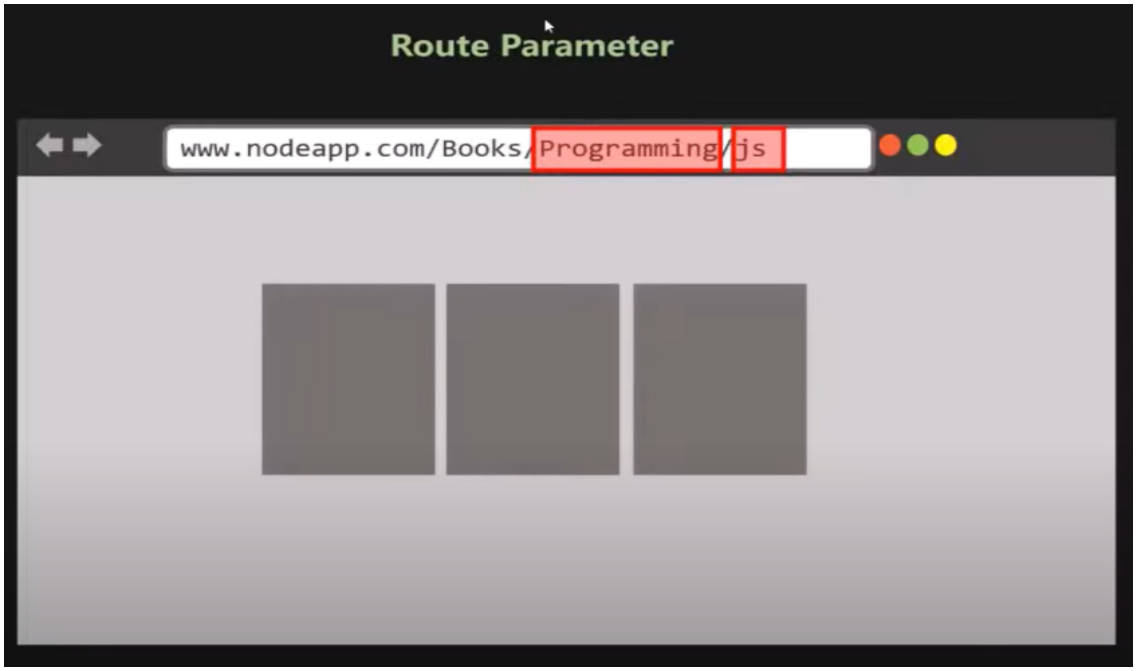
2) Resource-based URL:- Search on the server is based on the direct name and renders the page on the browser.



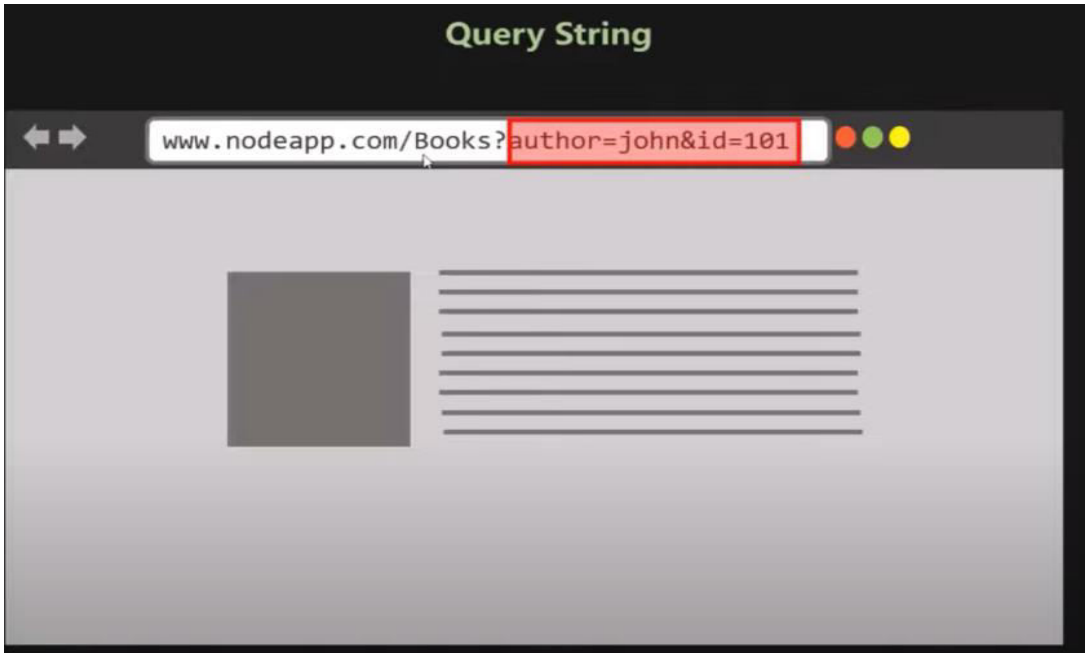




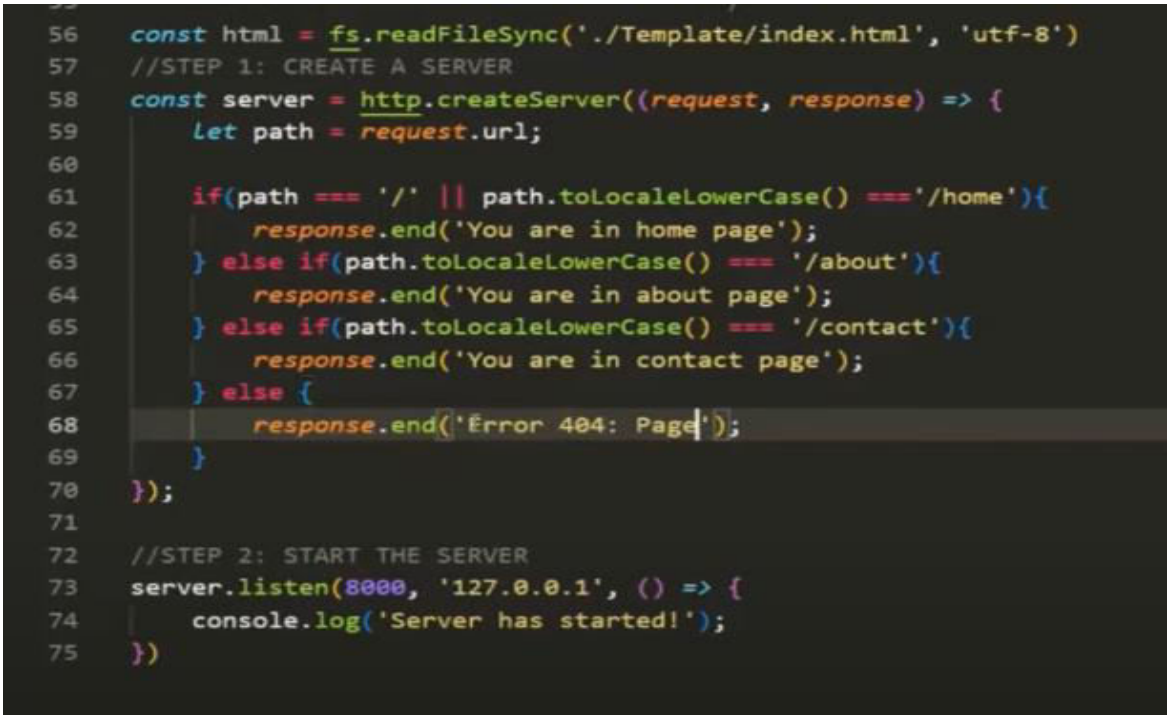
Route can also take parameter:-



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



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



```
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.end('You are in home page');
    } else if(path.toLocaleLowerCase() === '/about'){
        response.end('You are in about page');
    } else if(path.toLocaleLowerCase() === '/contact'){
        response.end('You are in contact page');
    } else {
        response.end('Error 404: Page');
    }
});

//STEP 2: START THE SERVER
server.listen(8000, '127.0.0.1', () => {
    console.log('Server has started!');
})
```

we have created routing.

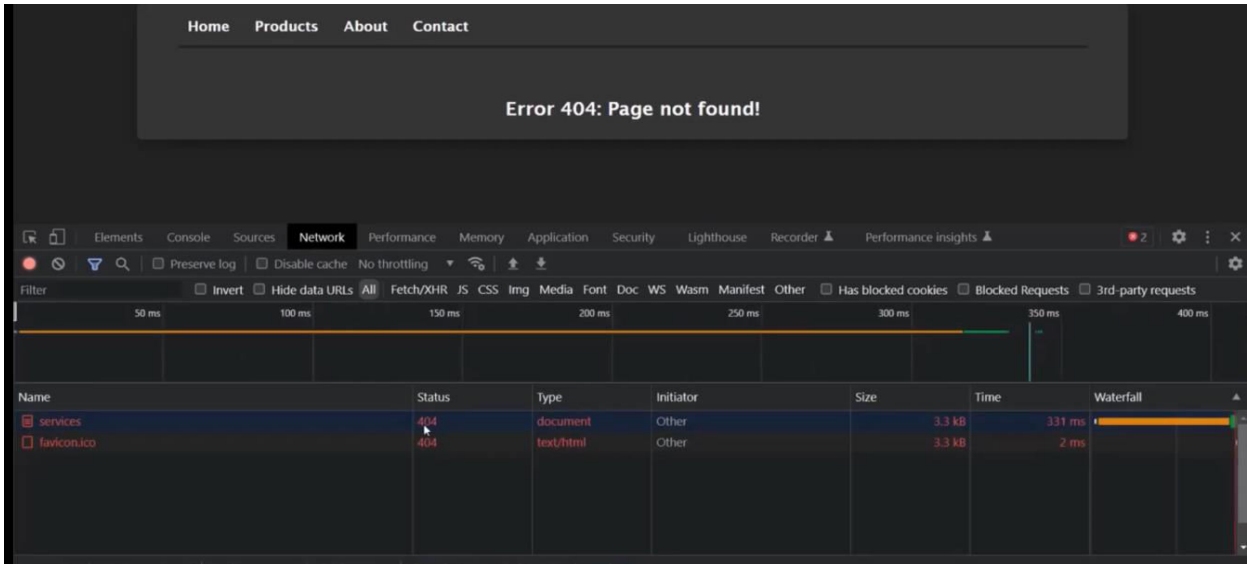
Sending HTML Response:- In this, we have some different HTML pages and we are passing it to response.end() fn so that is, render that HTML page.

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.

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:-

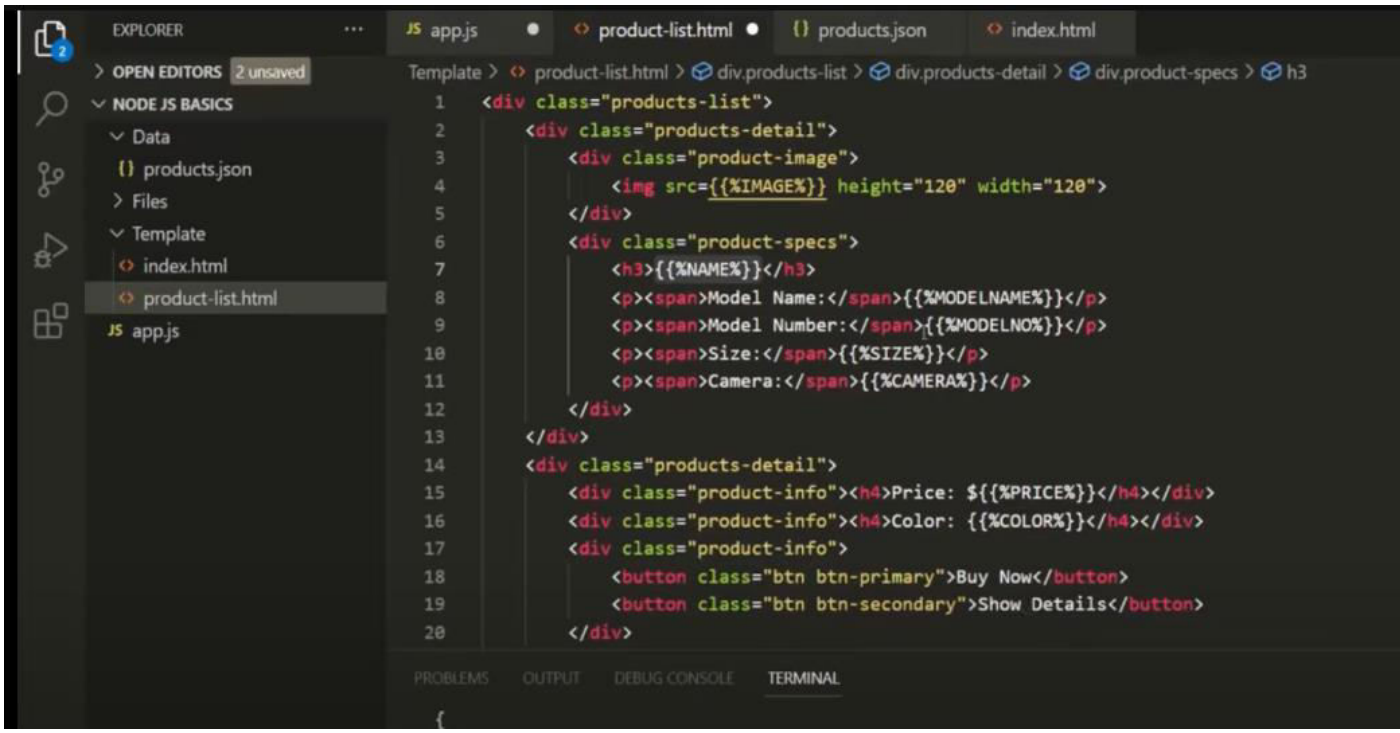

```
[
    {
      "id": 0,
      "name": "APPLE iPhone SE",
      "color":"Black",
      "ROM": 128,
      "price": 990,
      "modeName": "iPhone SE",
      "modelNumber": "MHGT3HN/A",
      "size": "11.94 cm (4.7 inch) Retina HD Display",
      "camera": "12MP Rear Camera | 7MP Front Camera",
      "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,
      "modeName": "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",
```


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.

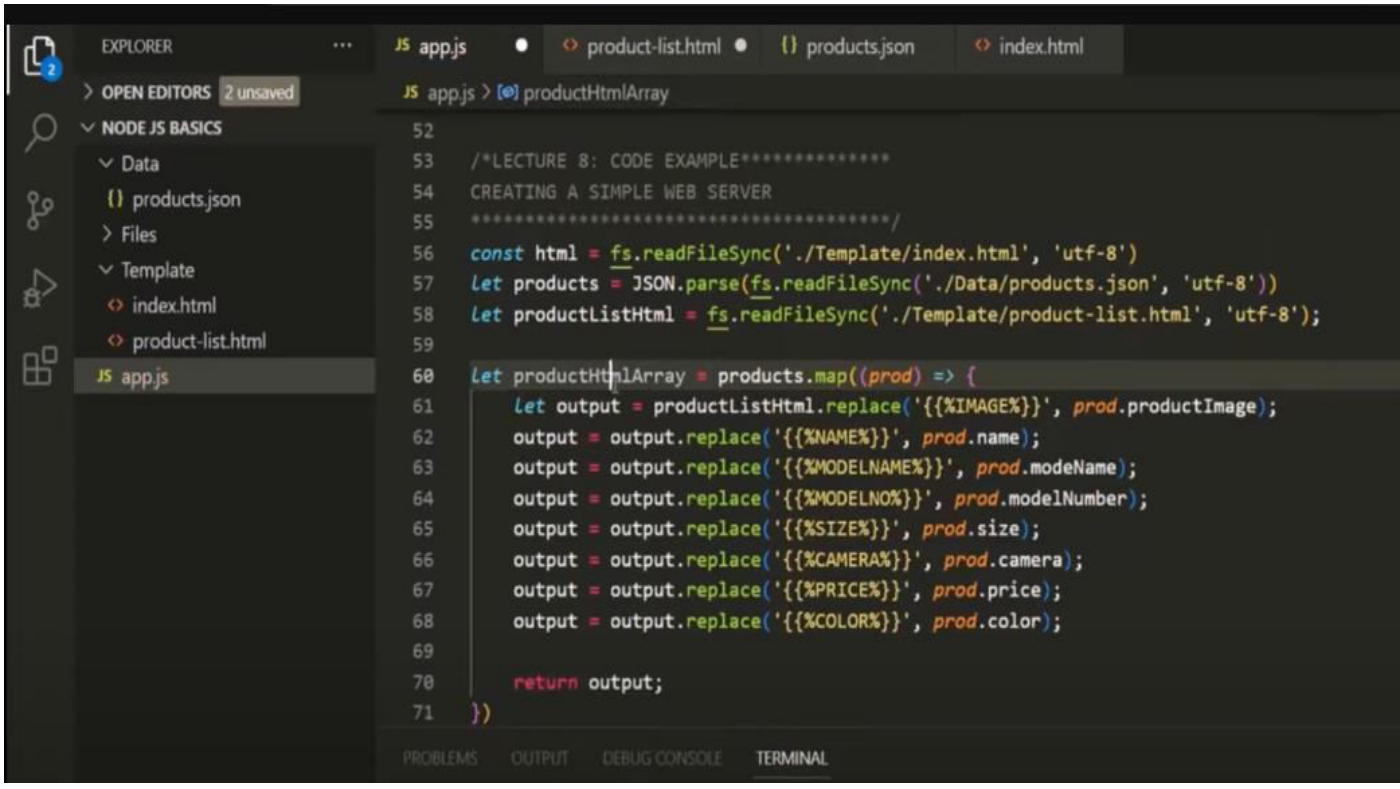
```
    fs.readFile('./Data/products.json', 'utf-8', (error, data) => {
        let products = JSON.parse(data)
        response.end(data);
    })
} else {
```

Transforming JSON data into HTML:-

First, we have our HTML page:-



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

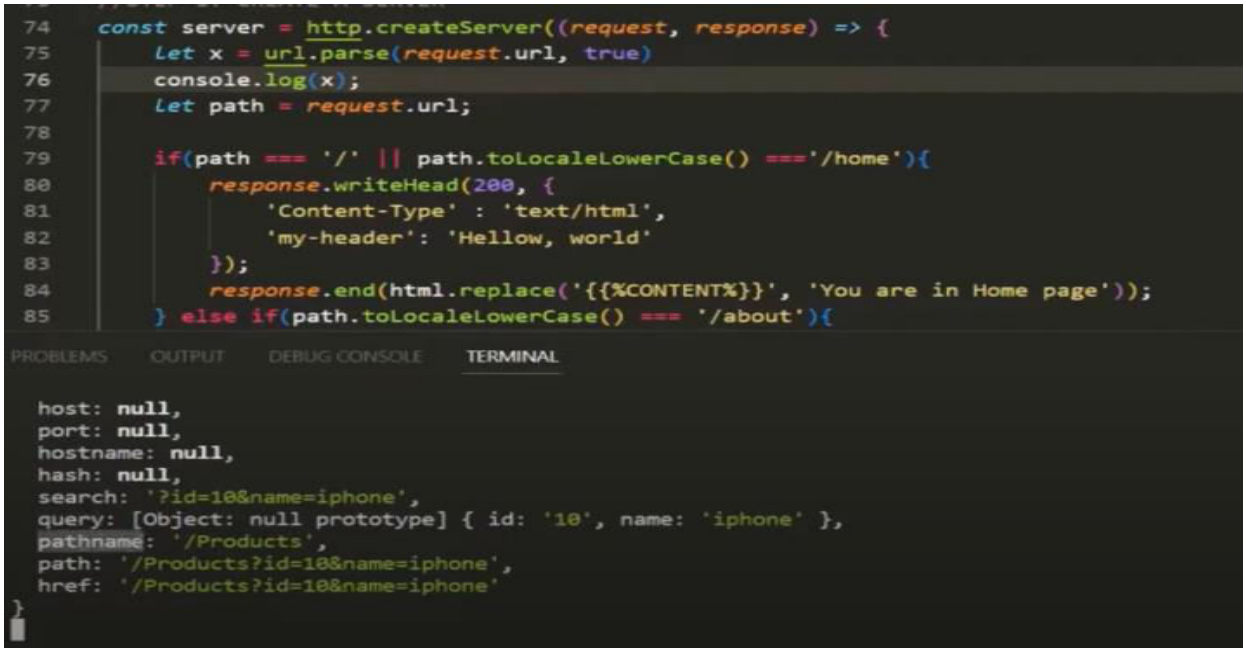


This {{%%name%%}} is used as a placeholder, to replace the content inside it.

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

Parsing query string from URL:-

First, we need to import url module/package.



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:-

```
const server = http.createServer((request, response) => {
    let {query, pathname} = url.parse(request.url, true)
    console.log(x);
    let path = request.url;

```

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:-

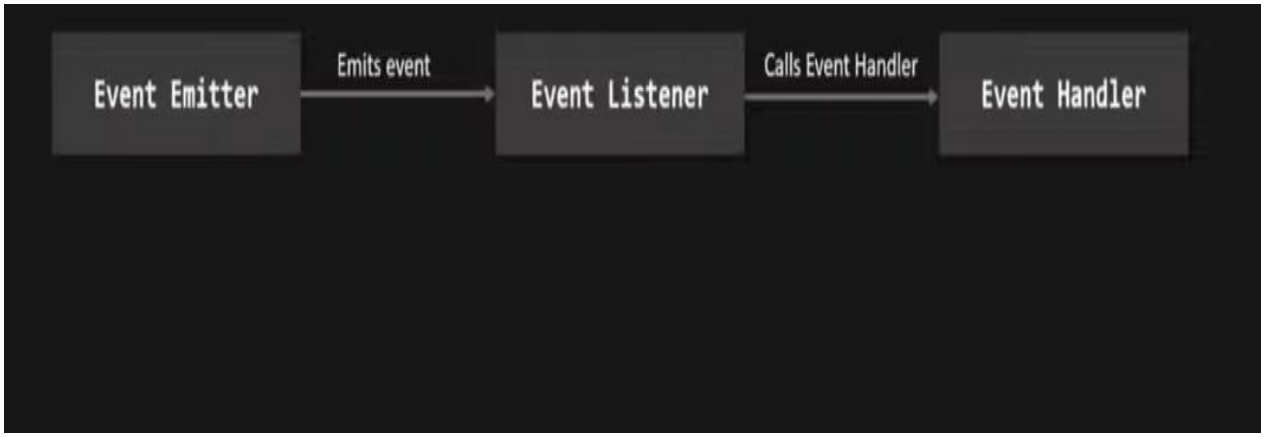
```
module.exports = function(template, product){
    let output = template.replace('{{%IMAGE%}}', product.productImage);
    output = output.replace('{{%NAME%}}', product.name);
    output = output.replace('{{%MODELNAME%}}', product.modeName);
    output = output.replace('{{%MODELNO%}}', product.modelNumber);
    output = output.replace('{{%SIZE%}}', product.size);
    output = output.replace('{{%CAMERA%}}', product.camera);
    output = output.replace('{{%PRICE%}}', product.price);
    output = output.replace('{{%COLOR%}}', product.color);
    output = output.replace('{{%ID%}}', product.id);
    output = output.replace('{{%ROM%}}', product.ROM);
    output = output.replace('{{%DESC%}}', product.Description);

    return output;
}
```

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

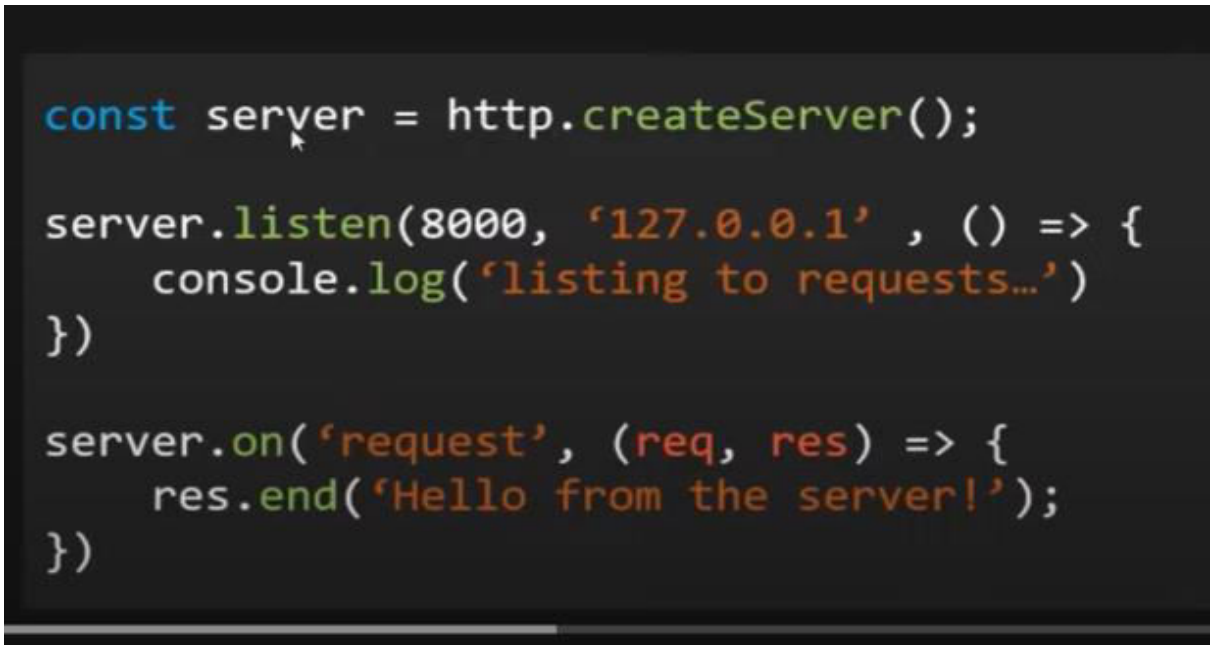
EVENT driven Architecture:-

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 even 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:-



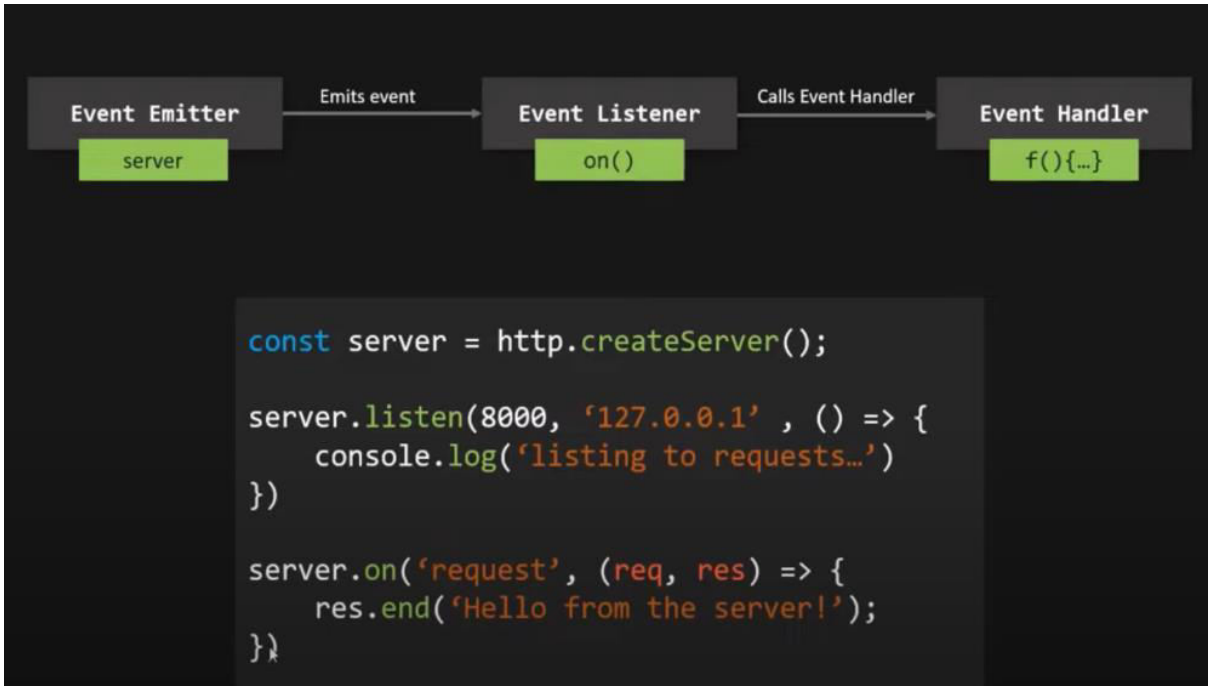
Whenever a request hits the server then the server will emit the named event called REQUEST.

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 ame 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.

```
nst events = require('events');
```

```
let myEmitter = new events.EventEmitter();
```

myEmitter will store the instance of EvenEmitter 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:-

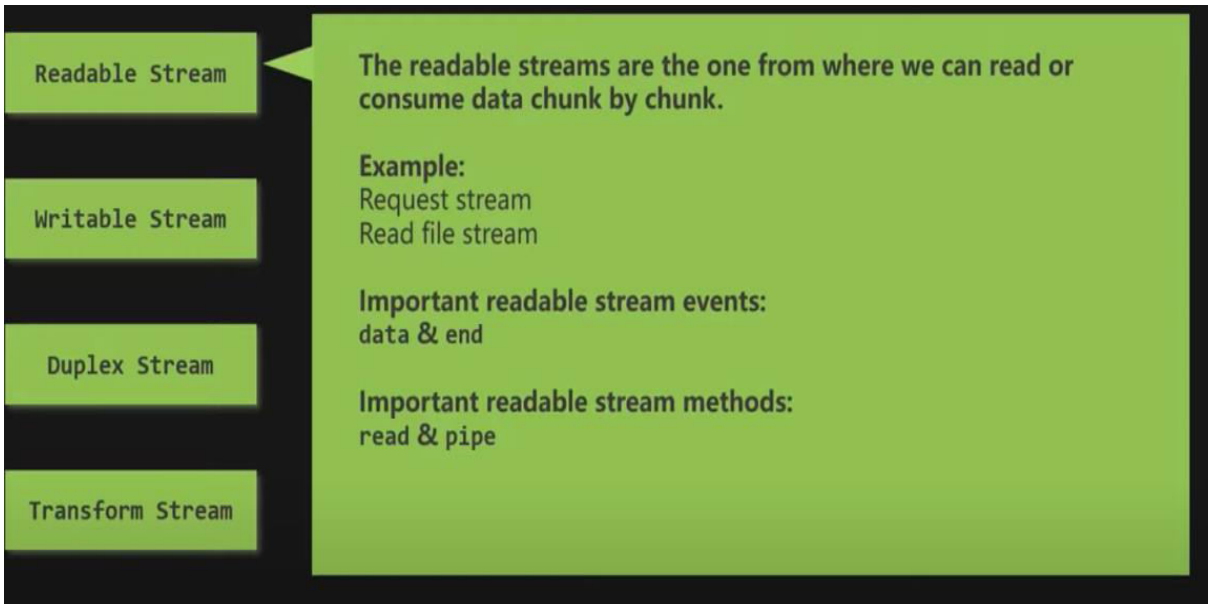
With Streams, we can process data piece by piece instead of reading or writing the whole data at once.

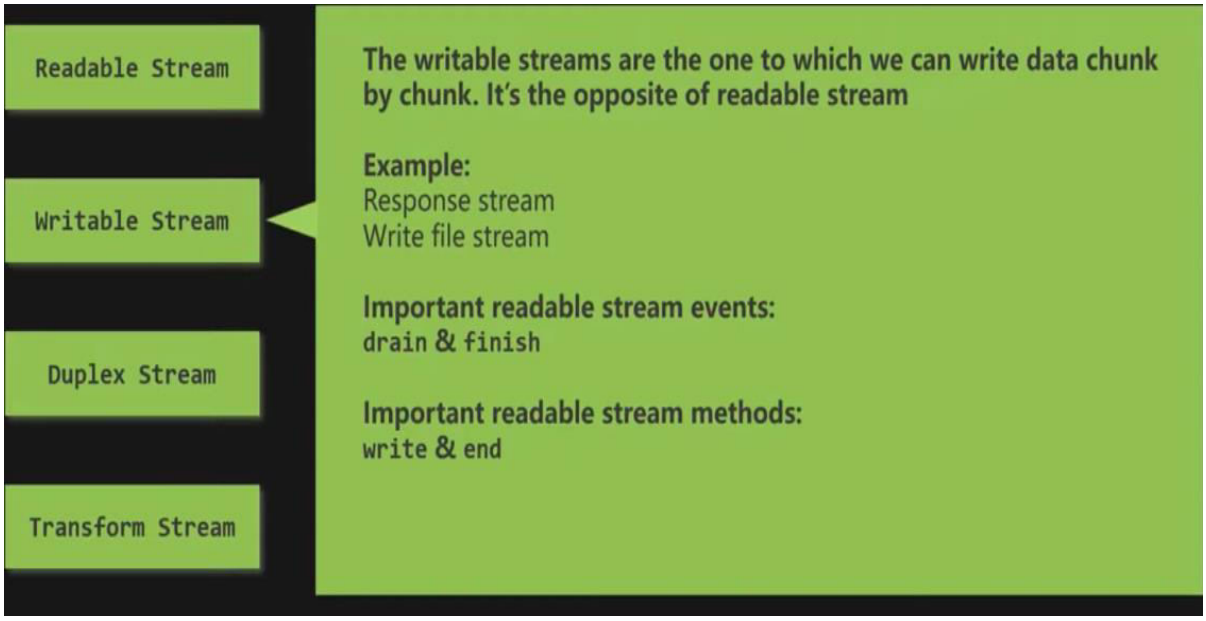
Advantage

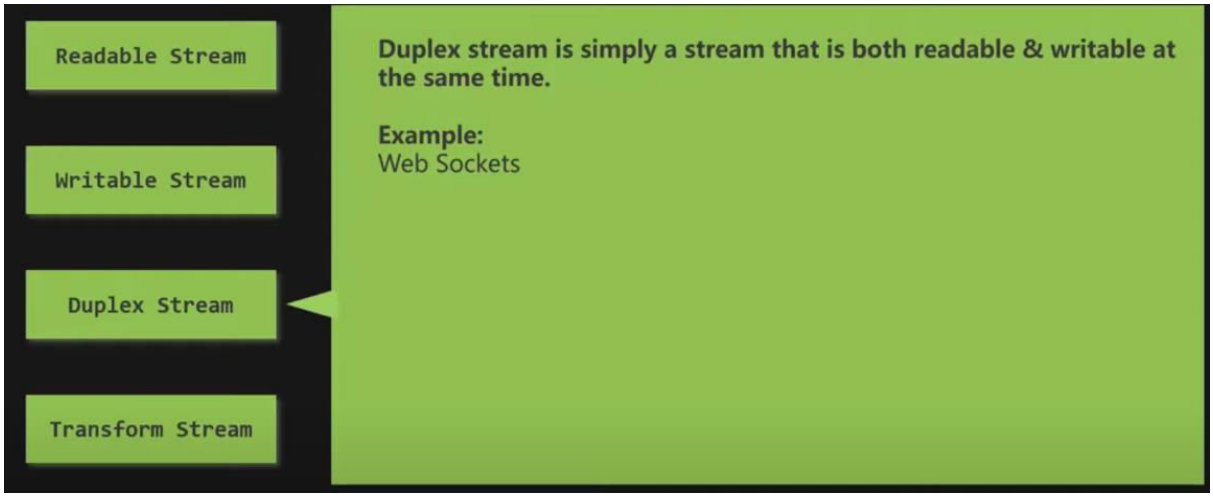
Streaming makes the data processing more efficient in terms of memory. Because there is no need to keep all the data in the memory.

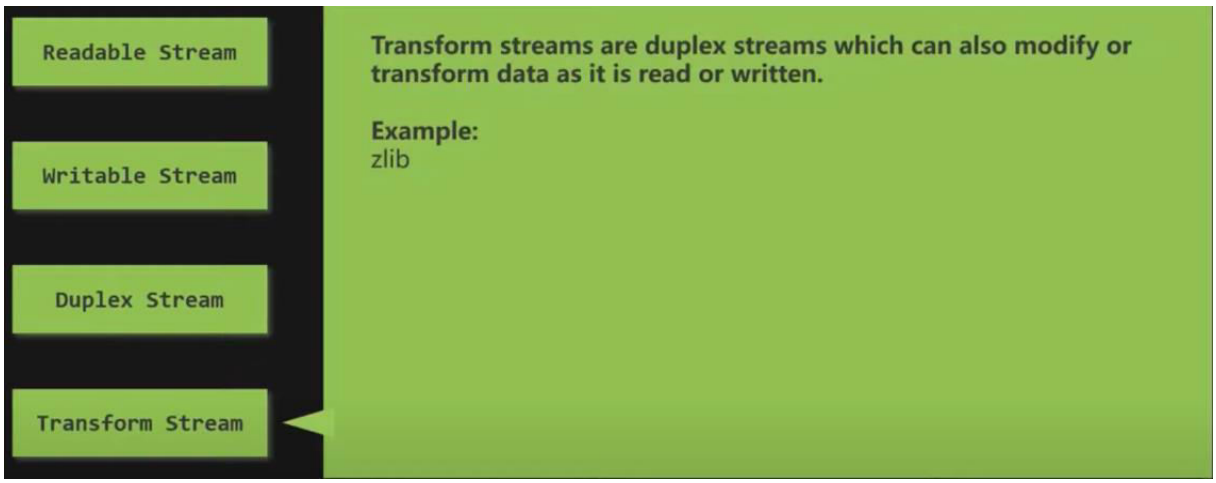
In terms of performance & time also, streaming has its advantage because we can start processing the data as soon as the first chunk of data arrives.

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




Readable Stream
Writable Stream
Duplex Stream
Transform Stream
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


Readable Stream
Writable Stream
Duplex Stream
Transform Stream
Duplex stream is simply a stream that is both readable & writable at the same time.
Example:
Web Sockets


Readable Stream
Writable Stream
Duplex Stream
Transform Stream
Transform streams are duplex streams which can also modify or transform data as it is read or written.
Example:
zlib

NPM [Node Package Manager]:-

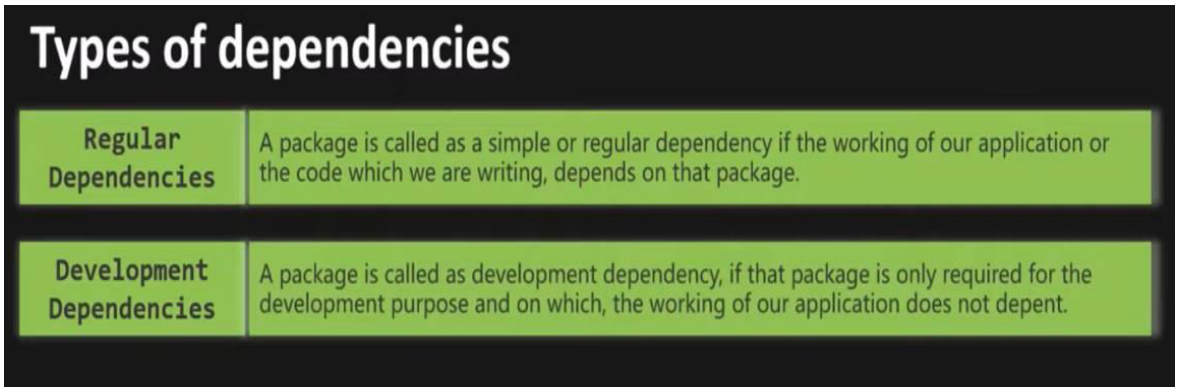
It is a command line interface for managing packages and also a repository from where we install packages.

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:-



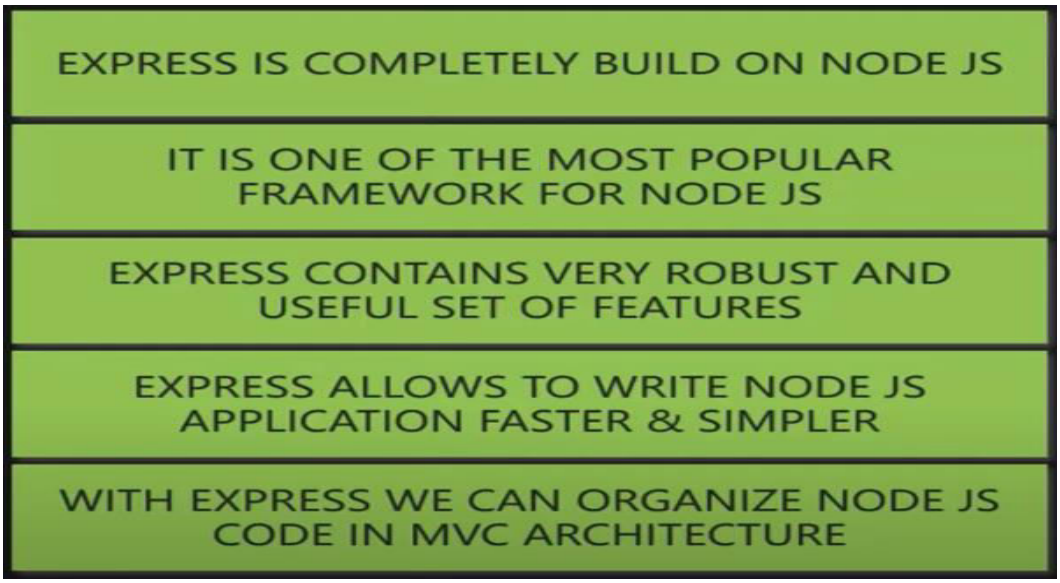
Ex for regular dependencies is express.

Ex for development dependencies is nodemon.

Skipped Node.Js architecture

2) ***EXPRESS.JS***

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



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

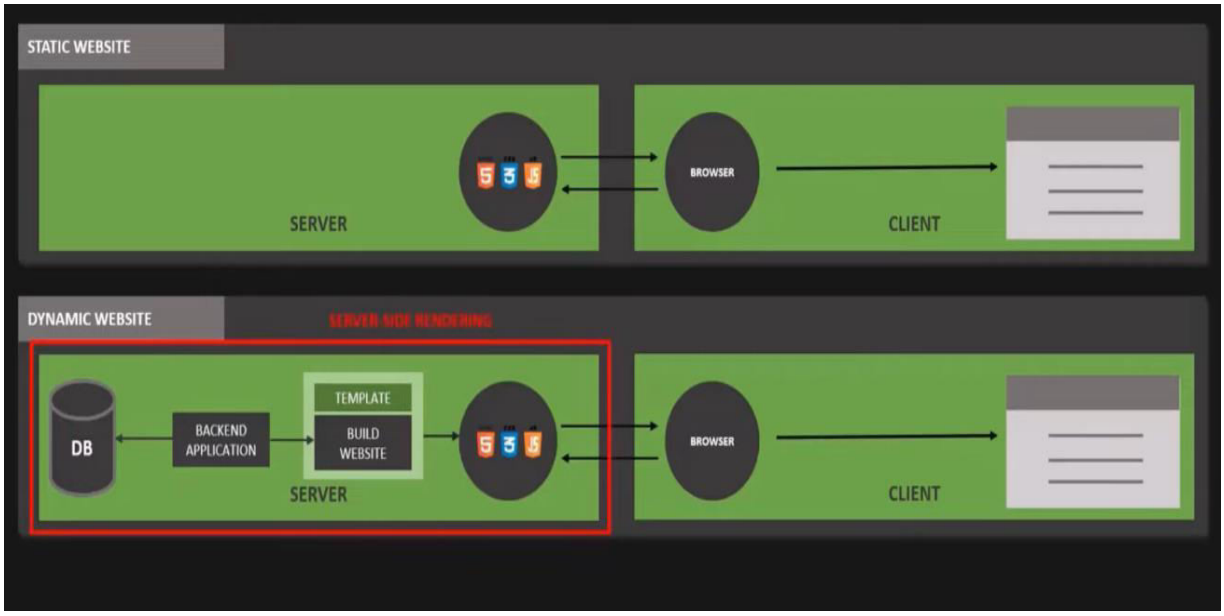
A dynamic website is a type of website that generates content on the server side in response to user requests. Unlike static websites, dynamic websites can change content and functionality based on user input, interactions, and data from databases. They typically involve server-side scripting, and database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

Key characteristics of dynamic websites include:

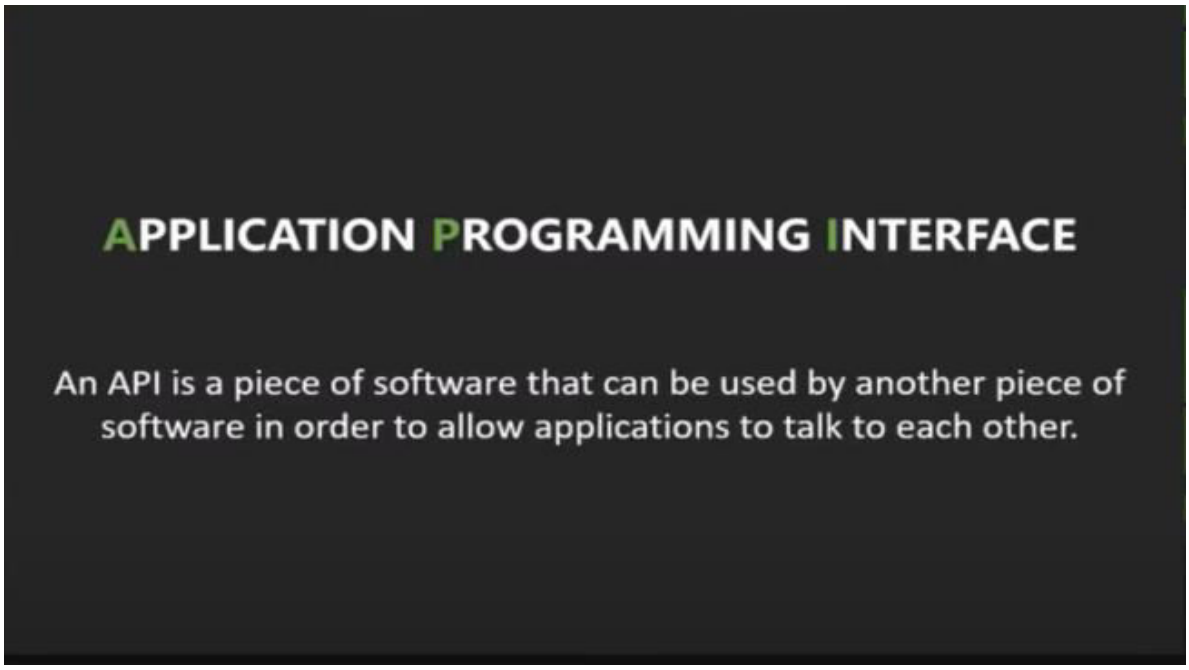
1. **Server-Side Scripting:** Dynamic websites use server-side scripting languages (such as PHP, Python, Ruby, Node.js, etc.) to generate content dynamically on the server in response to user requests. The server processes the script,

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.

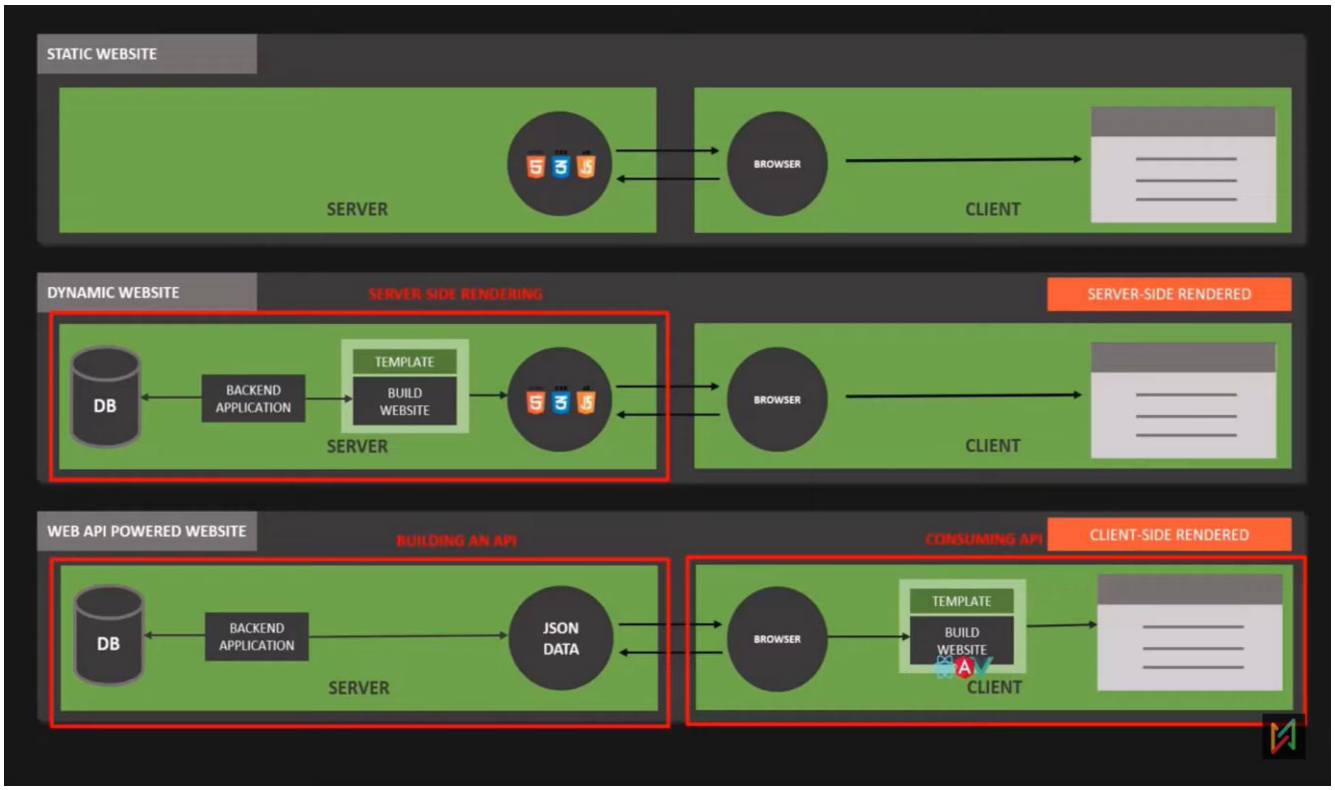
While dynamic websites offer more interactivity and flexibility, they may require more server resources, have higher development complexity, and could potentially be slower to load compared to static websites. The choice between a dynamic and static approach depends on the specific requirements of the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

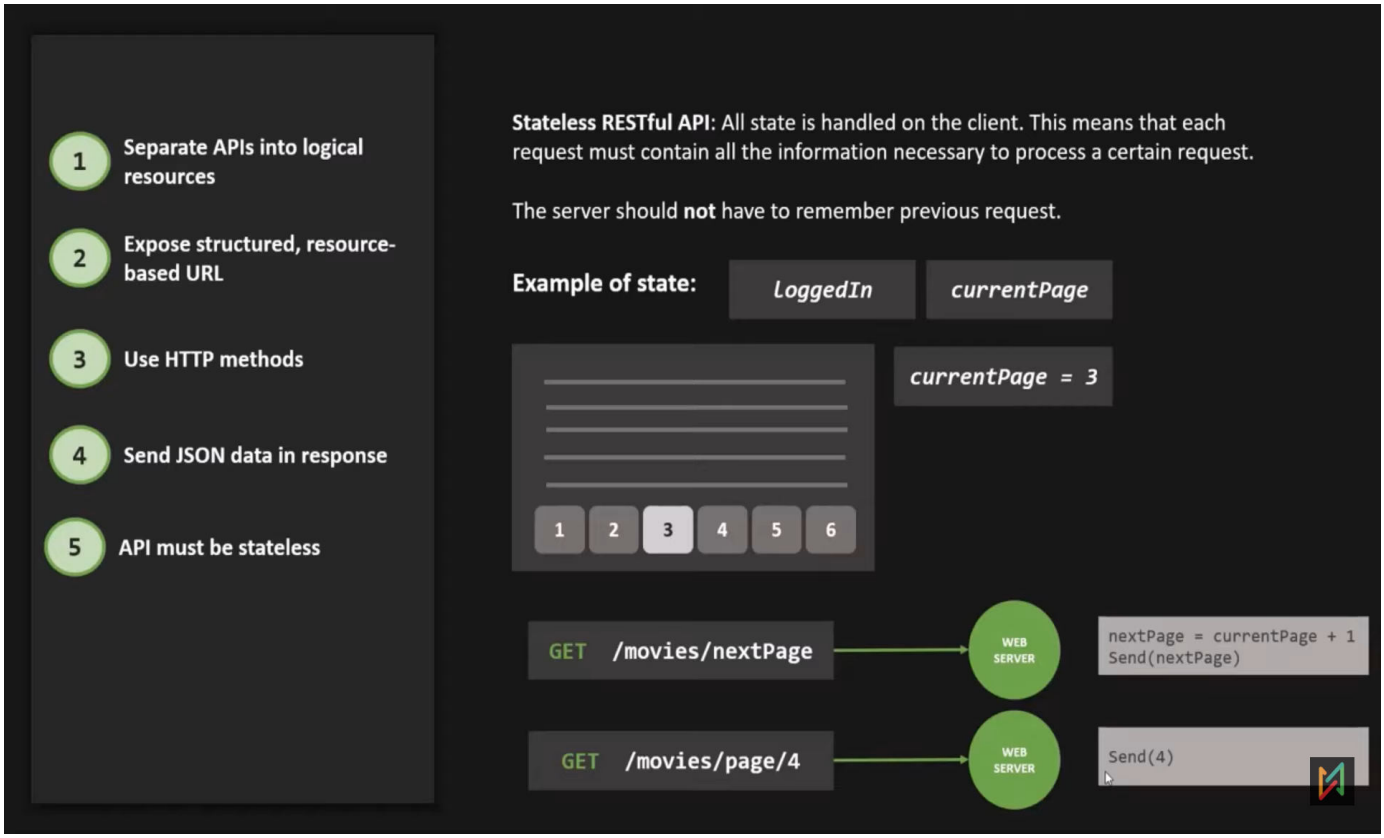
The choice between CSR and SSR depends on factors such as the type of application, user experience goals, SEO requirements, and performance considerations. Modern frameworks and libraries often provide tools for implementing both CSR and SSR, allowing developers to choose the approach that best suits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

REST Architecture:-

It is used to make web API’s.

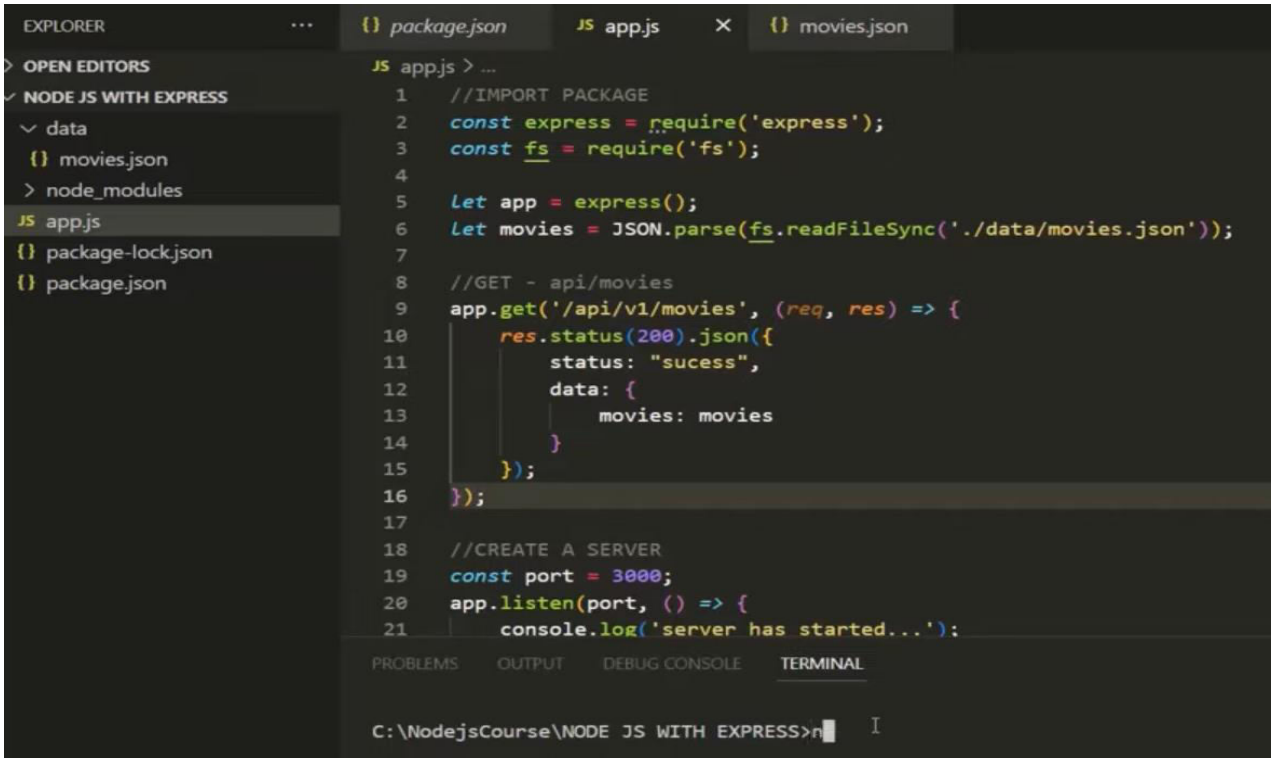
Principles to follow for creating REST APIs:-

Resource should only be a noun.



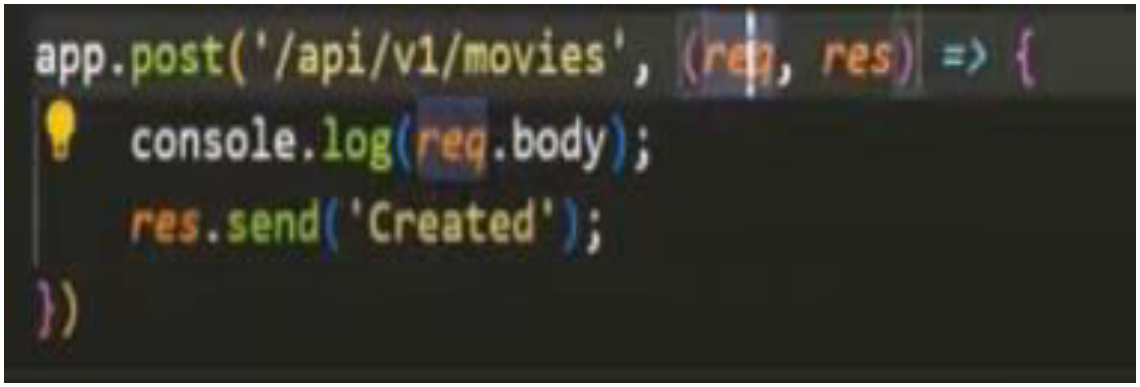
Handling GET Request:-

Here, we are creating an API using REST architecture.



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

Handling POST Request:-



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:-

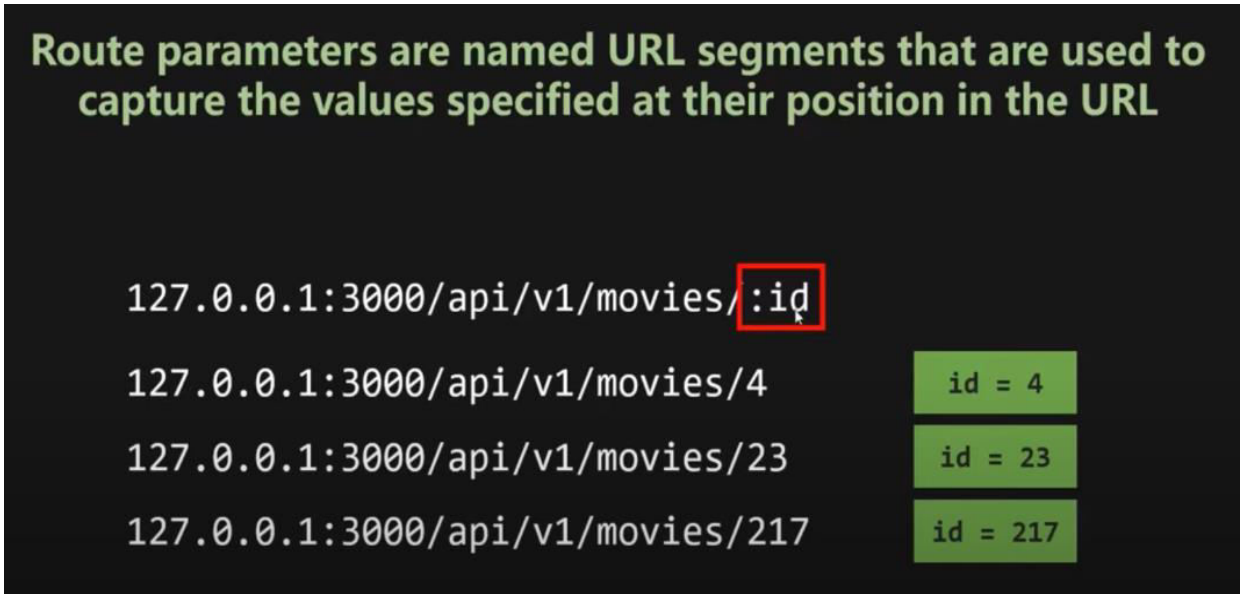
```
JS app.js > app.post('/api/v1/movies') callback
  5   let app = express();
  6   let movies = JSON.parse(fs.readFileSync('./data/movies.json'));
  7
  8   //GET - api/v1/movies
  9   app.get('/api/v1/movies', (req, res) => {
 10       res.status(200).json({
 11           status: "sucess",
 12           count: movies.length,
 13           data: {
 14               movies: movies
 15           }
 16       });
 17   });
 18
 19
 20   //POST - api/v1/movies
 21   app.post('/api/v1/movies', (req, res) => {
 22       console.log(req.body);
 23       res.send('Created');
 24   })

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

C:\NodejsCourse\NODE JS WITH EXPRESS>node app.js
server has started...
undefined
```

So to clarify this issue we need to use middleware.

Here, we are using express.json() middleware, which adds the request body to the request object, so when we console req.body it won't be printed as undefined in the terminal.

It is termed as middleware because it stands between request and response.

Route Parameters:-



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');
})
```

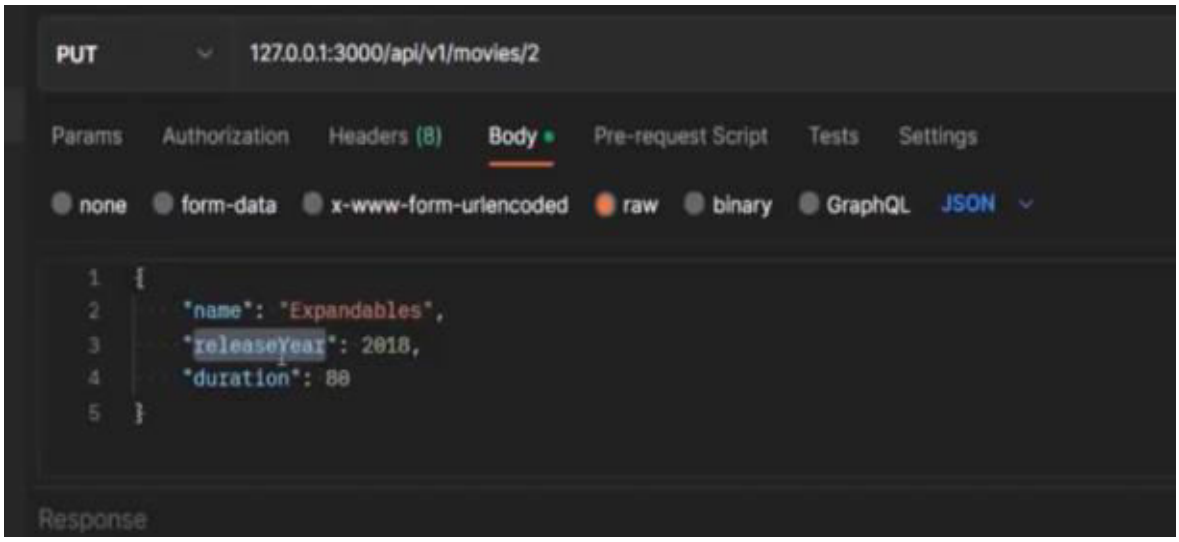
Now even if we do not provide any name or route parameter then it won't be an issue.

Handling PATCH API:-

PUT vs PATCH

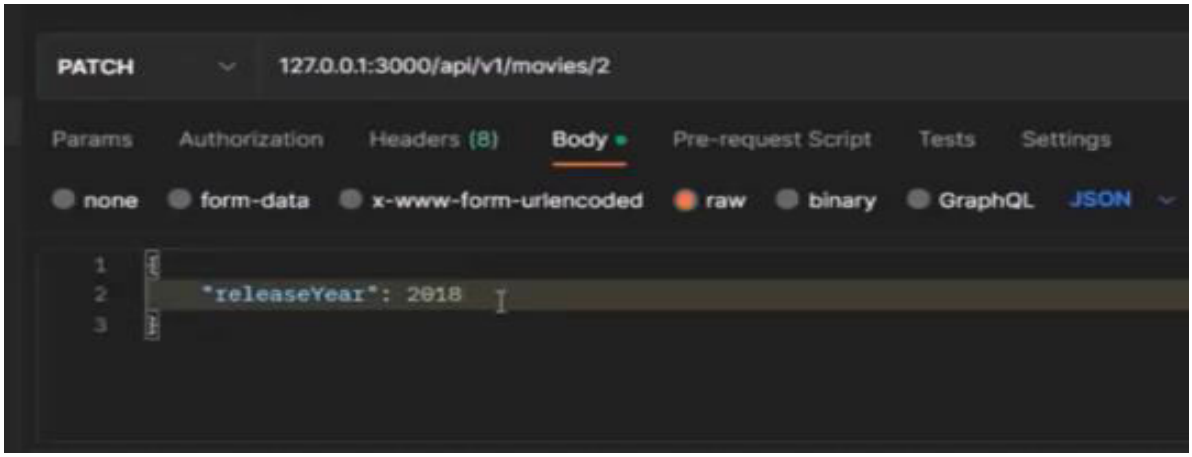
- PUT is a method of modifying resource where the client sends data that updates the entire resource .
- PATCH is a method of modifying resources where the client sends partial data that is to be updated without modifying the entire data.

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:-




```
app.patch('/api/v1/movies/:id', (req, res) => {
    let id = req.params.id * 1;
})
```

Handling DELETE API:-

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

```
app.delete('/api/v1/movies/:id', (req, res) => {
    const id = req.params.id * 1;
    const movieToDelete = movies.find(el => el.id === id);
    const index = movies.indexOf(movieToDelete);

    movies.splice(index, 1);

    fs.writeFile('./data/movies.json', JSON.stringify(movies), (err) => {
        res.status(204).json({
            status: "success",
            data: {
                movie: movieToUpdate
            }
        })
    })
})
```

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.

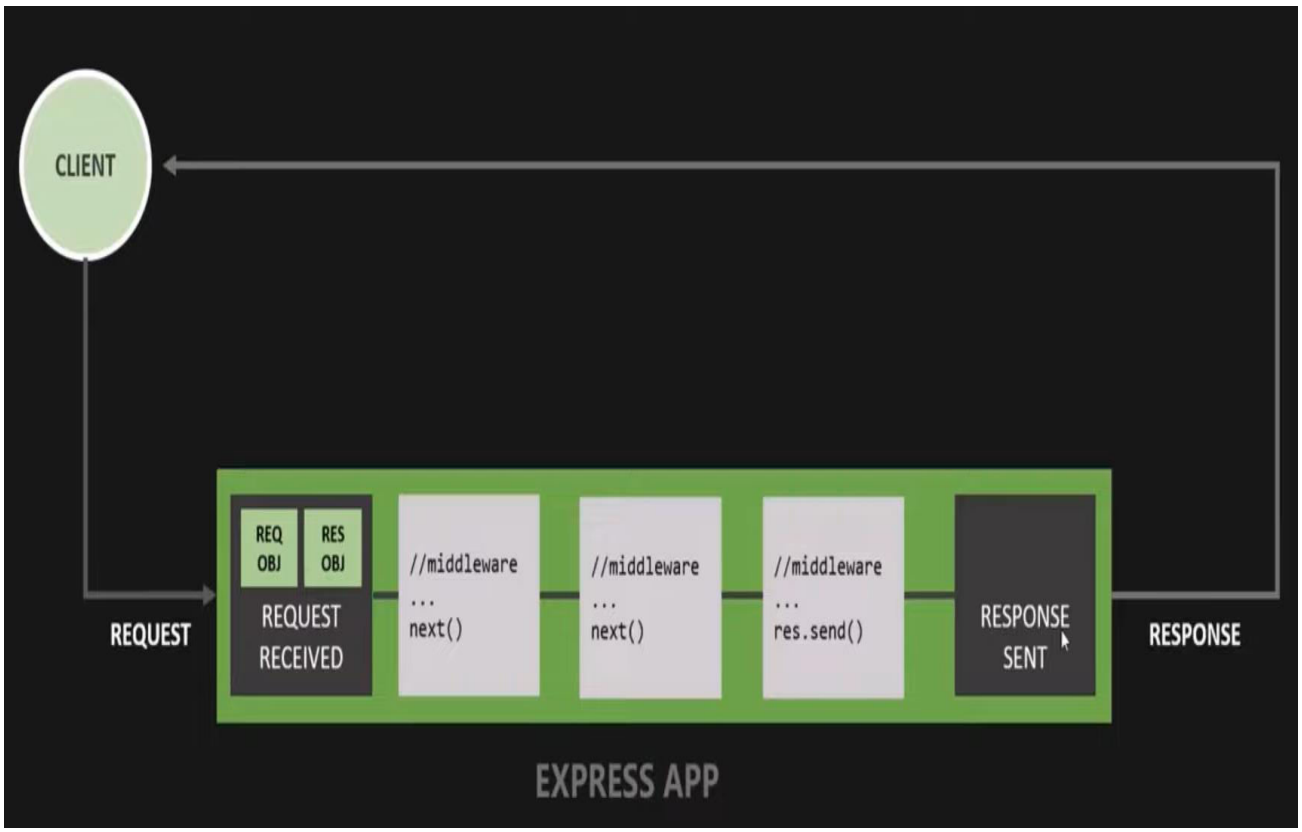
Here are some common use cases for middleware in backend development:

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.

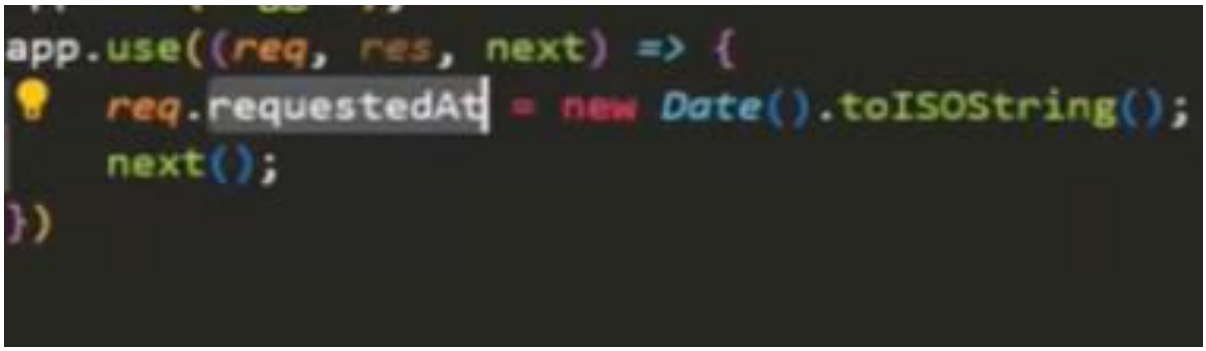
Route handlers are also middleware's.

NOTE:- instead of doing this –

We can also do this:-

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:-



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)).

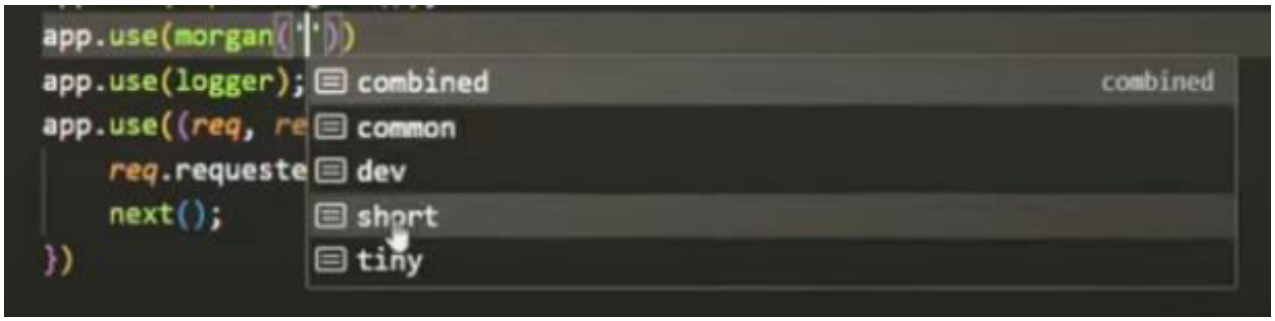
After installing this we need to require the package.

And to use this:-

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:-



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();

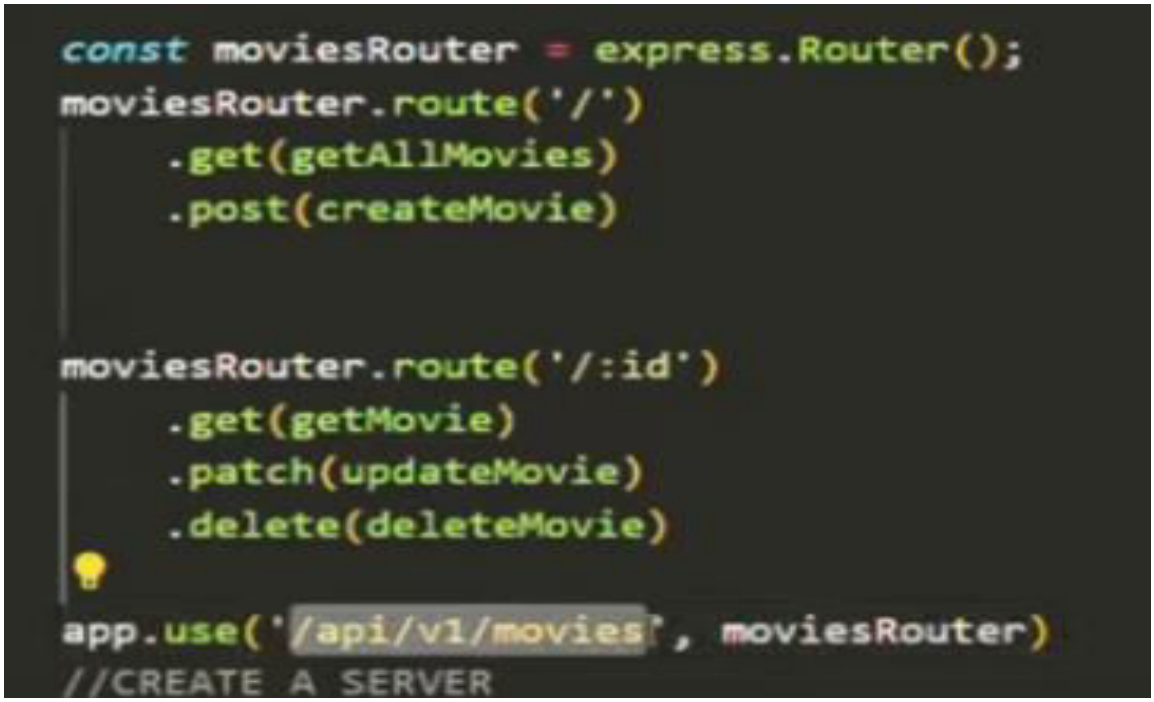
The router we have created will separately work for a particular resource like in this case movies.

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.



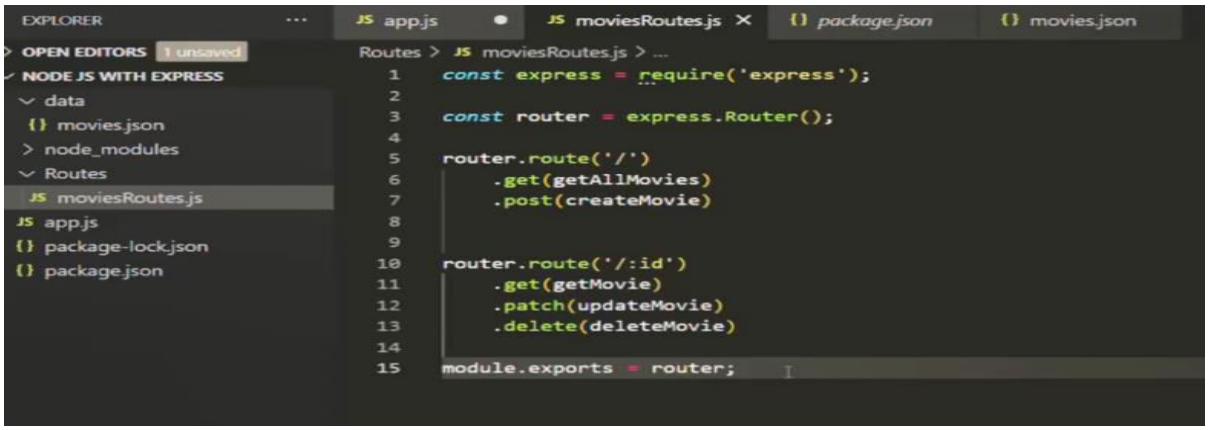
Here, in moviesRouter.route('/: id'), whenever we make request , which consists of this URL/api/v1/movies and has a particular id then the which we are adding will append to the url /api/v1/movies and will make app.use('/api/v1/movies',moviesRouter) to app.use('/api/v1/movies/:id',moviesRouter).

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:-



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.

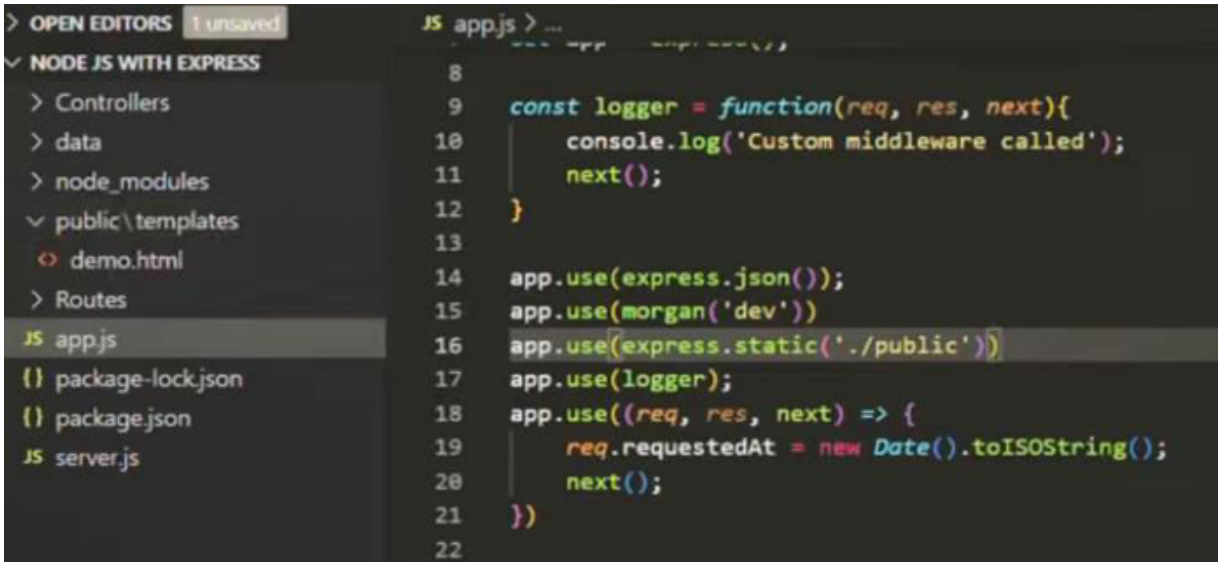
```
router.param('id', (req, res, next, value) => {

})
```

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.



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 the 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.

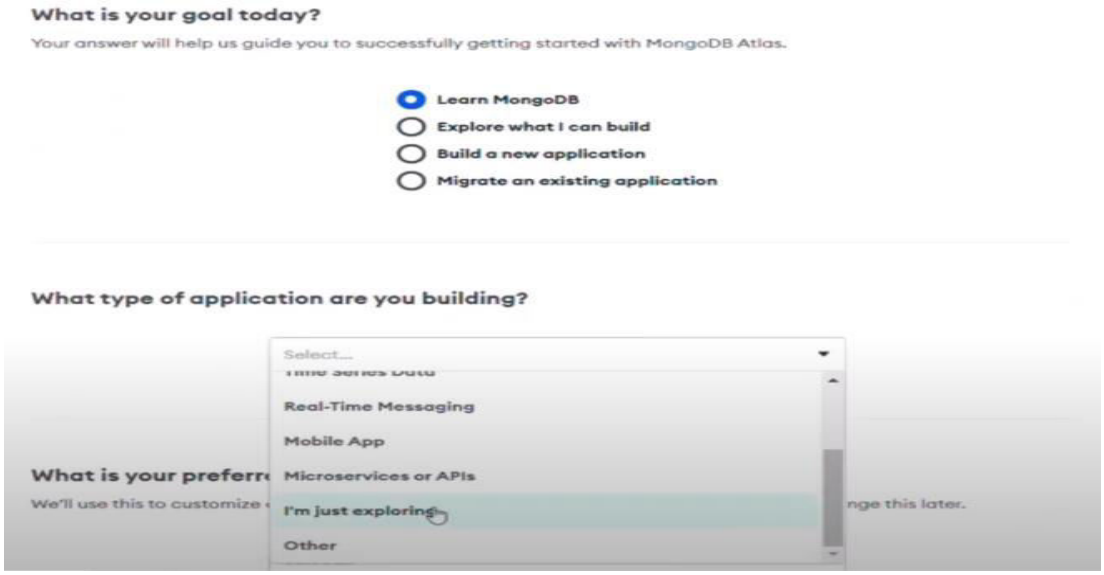
By doing dotenv.config() whatever variables are stored in config.env file will also stored in node.js environment variables.

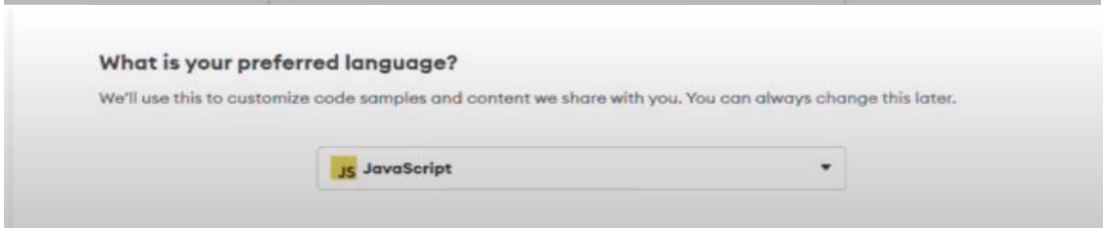
3) MONGODB:-

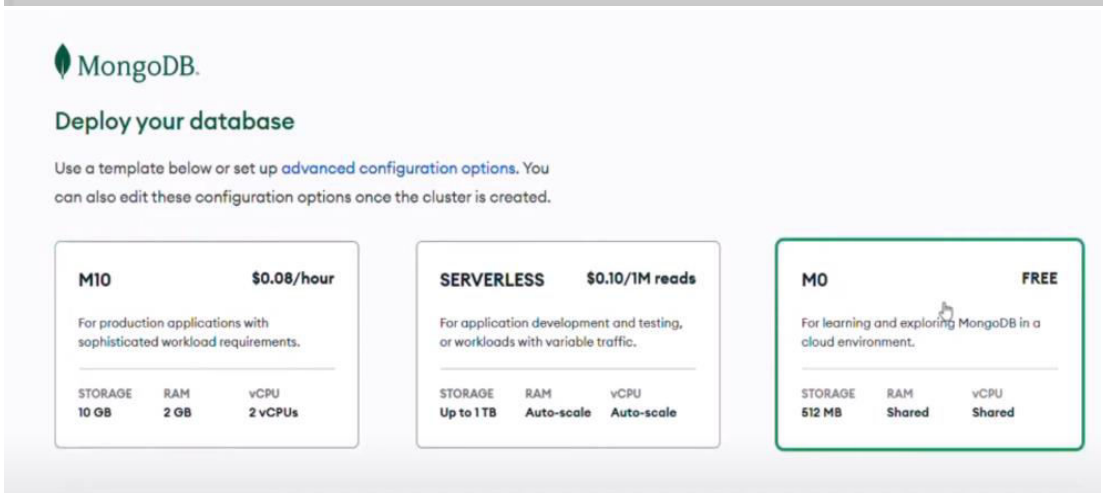
1st download MongoDb compass.

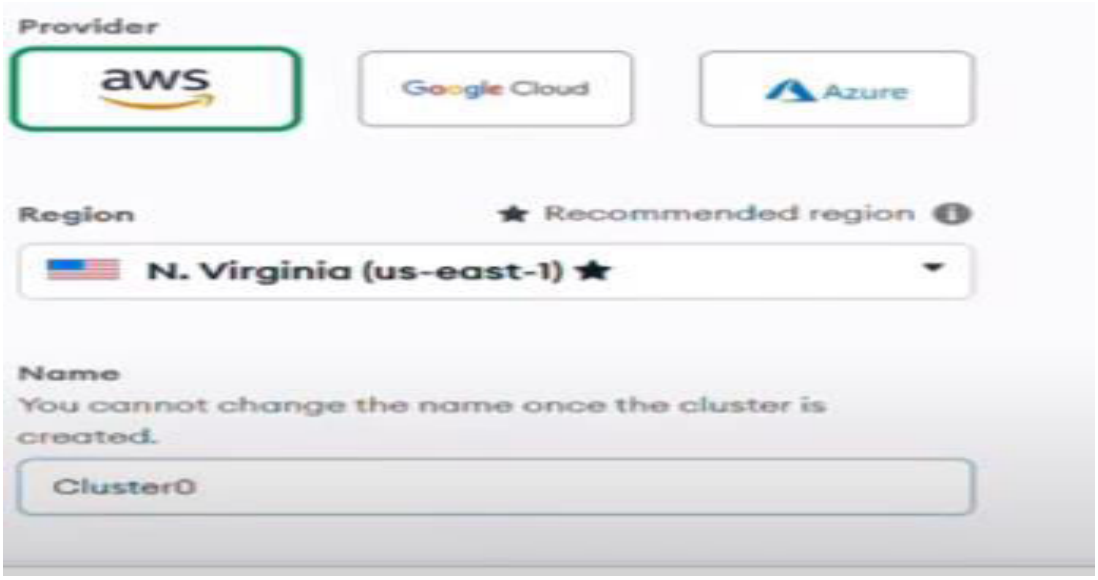
Creating a Hosted Database Using Atlas:-

Follow this series:-



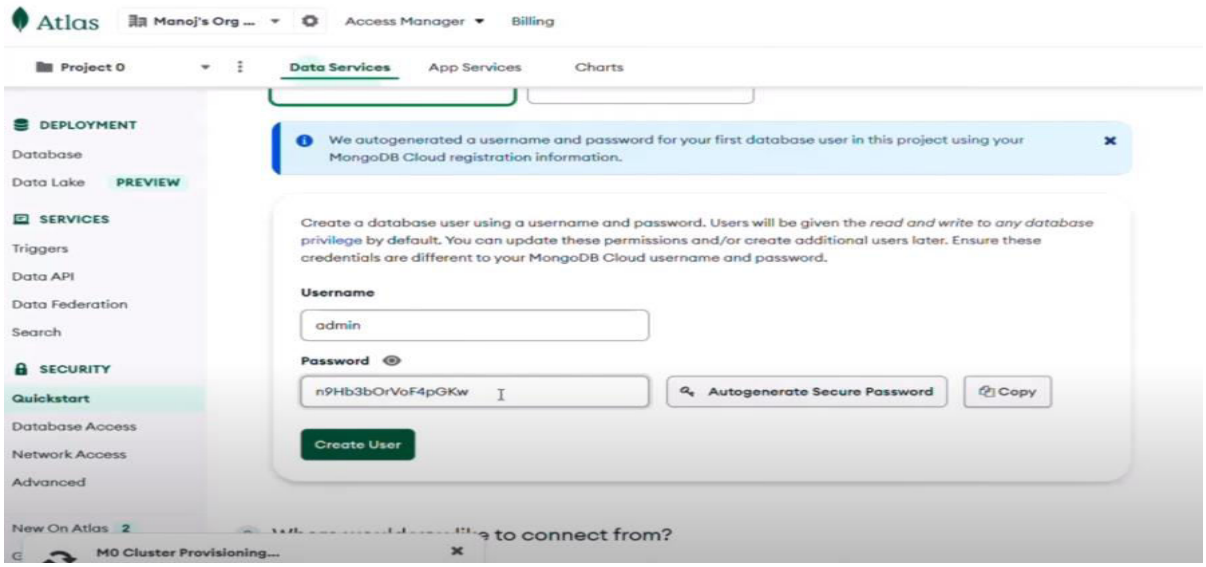






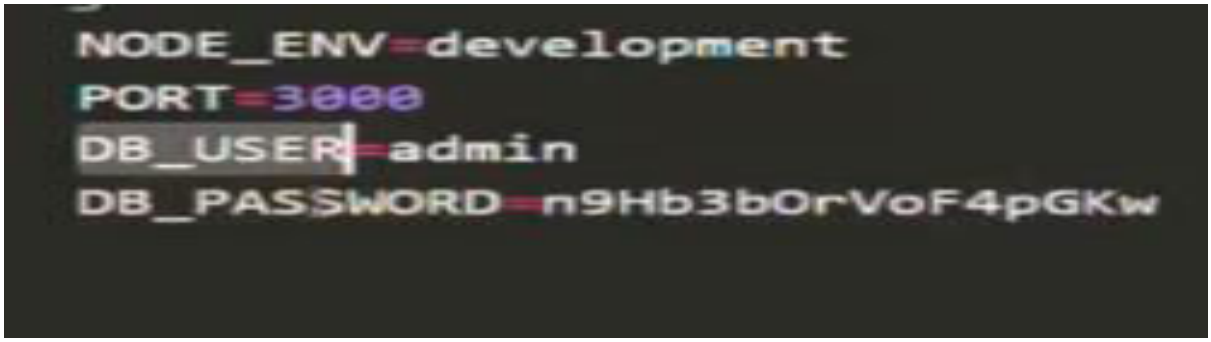
Name the Cluster whatever you want.

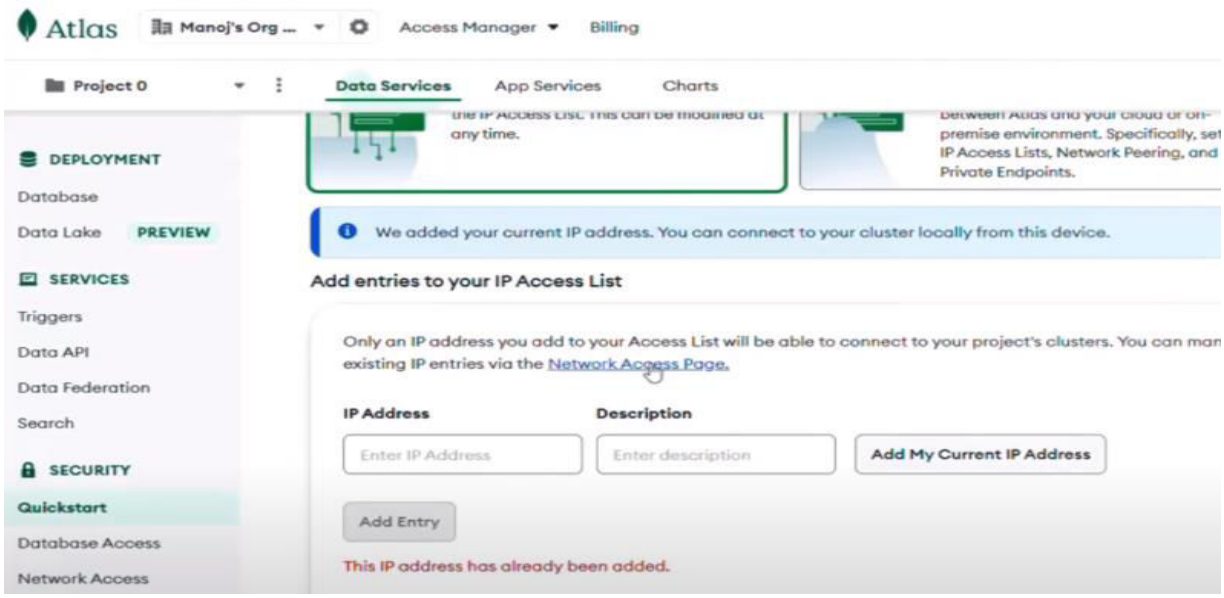
Now, click on Create.



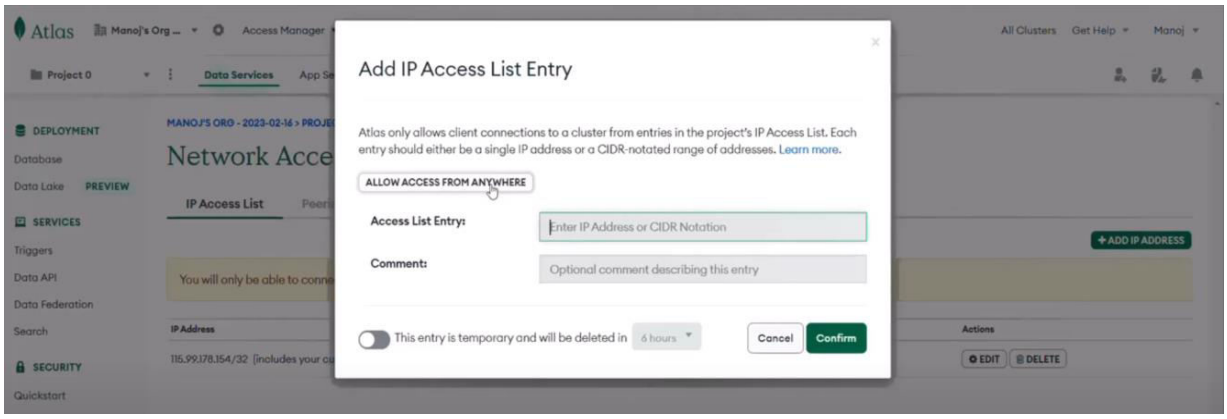
Now, set username and Password and click on Create User.

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



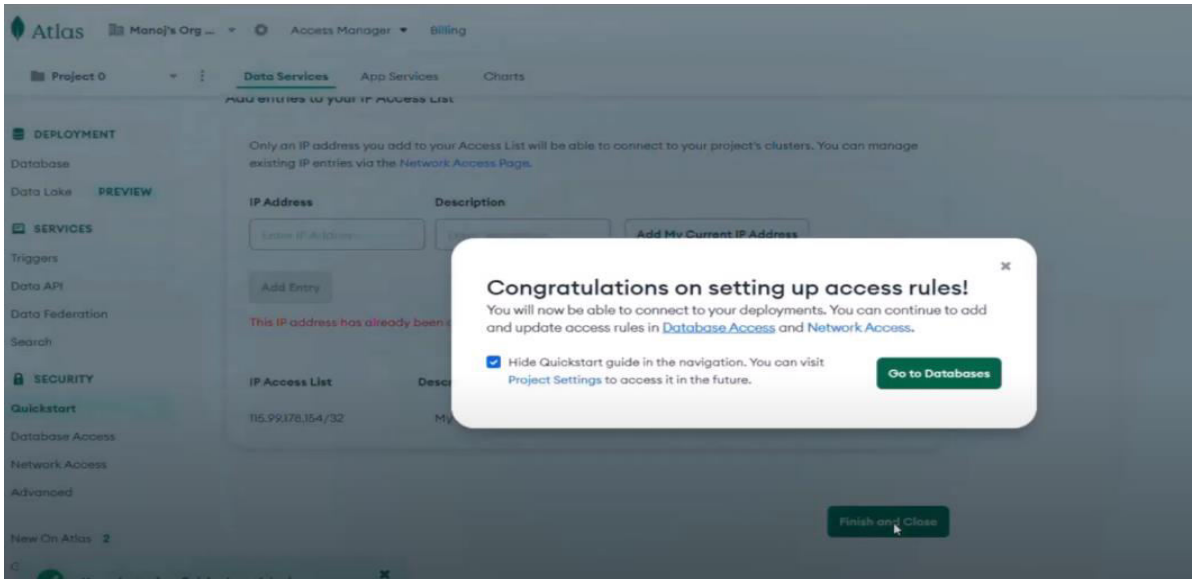


Now click on the Network Access Page.

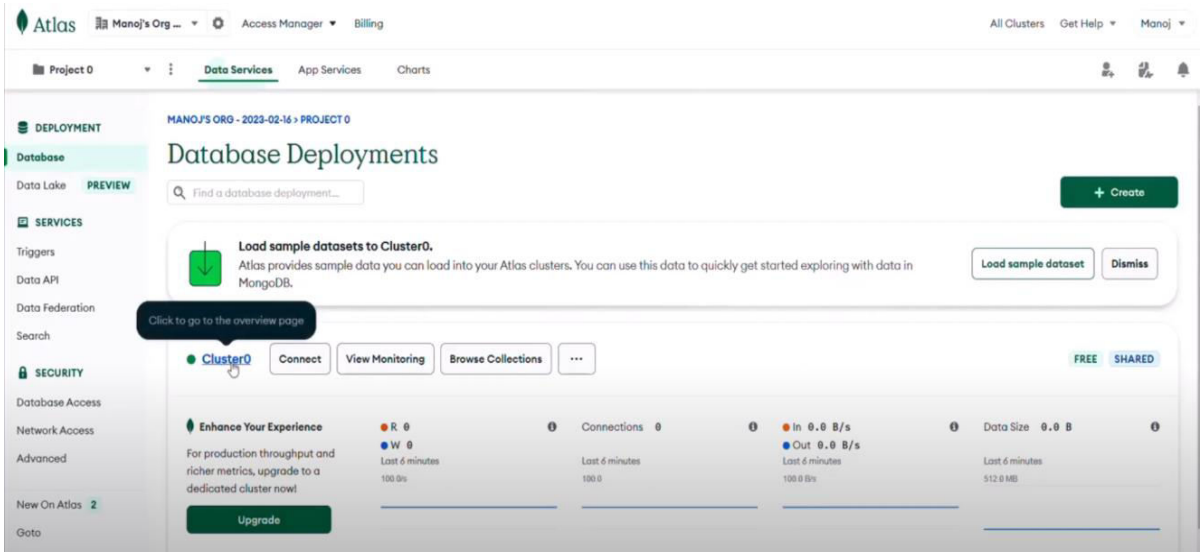


Click on ALLOW ACCESS FROM ANYWHERE and then in confirm.

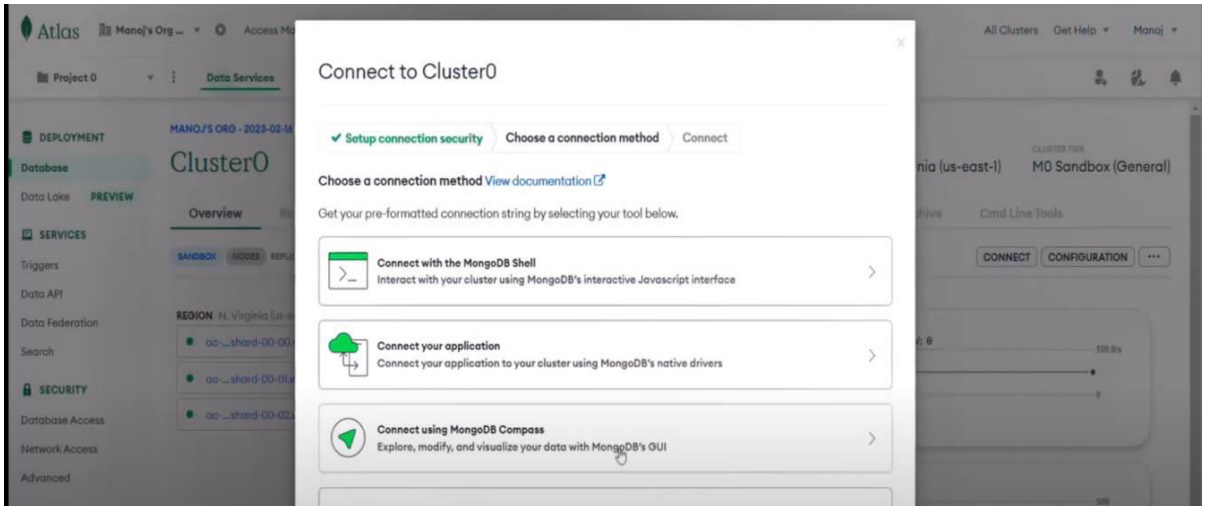
Then click on Atlas logo, the click on finish and Close Button.



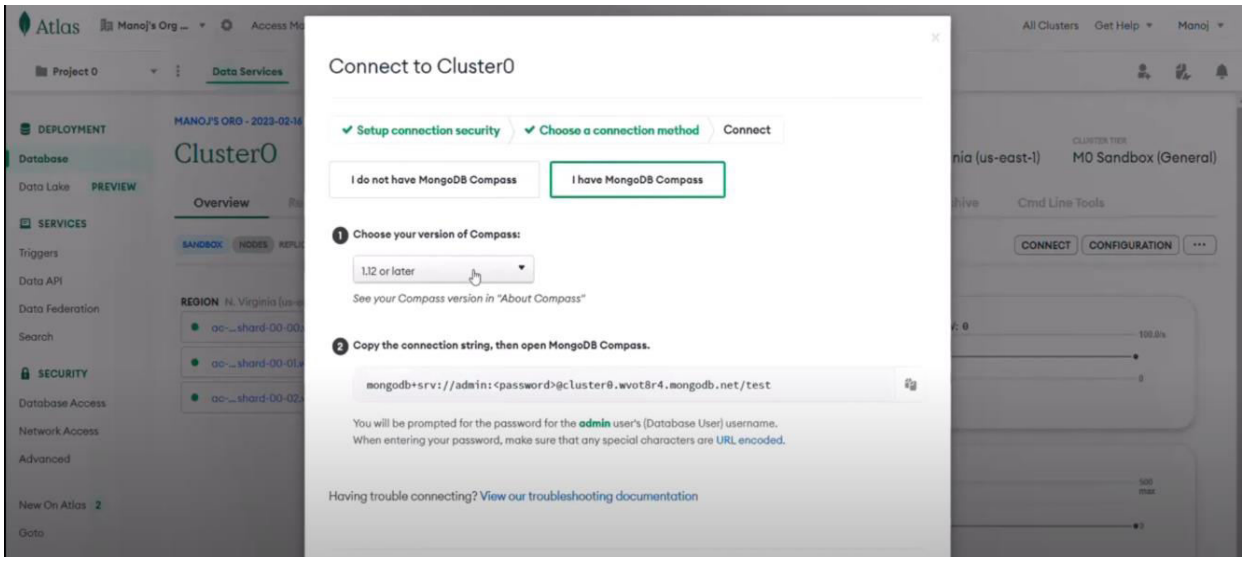
Then click on go to Databases.



Then click on connect.

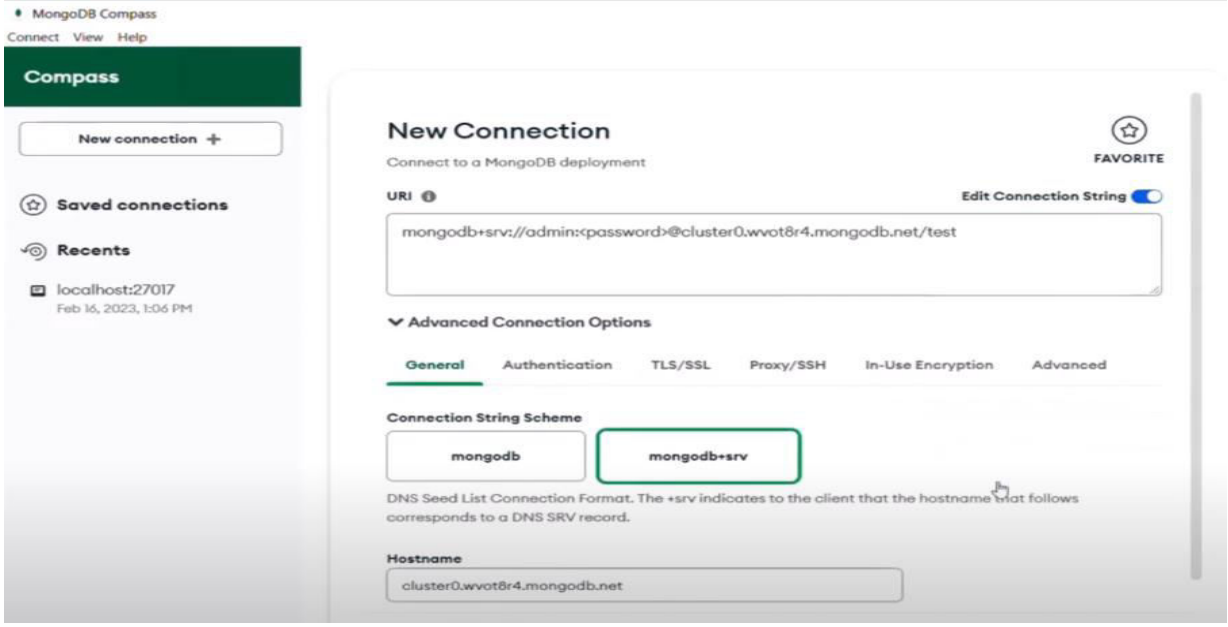


Then connect using MongoDB Compass.

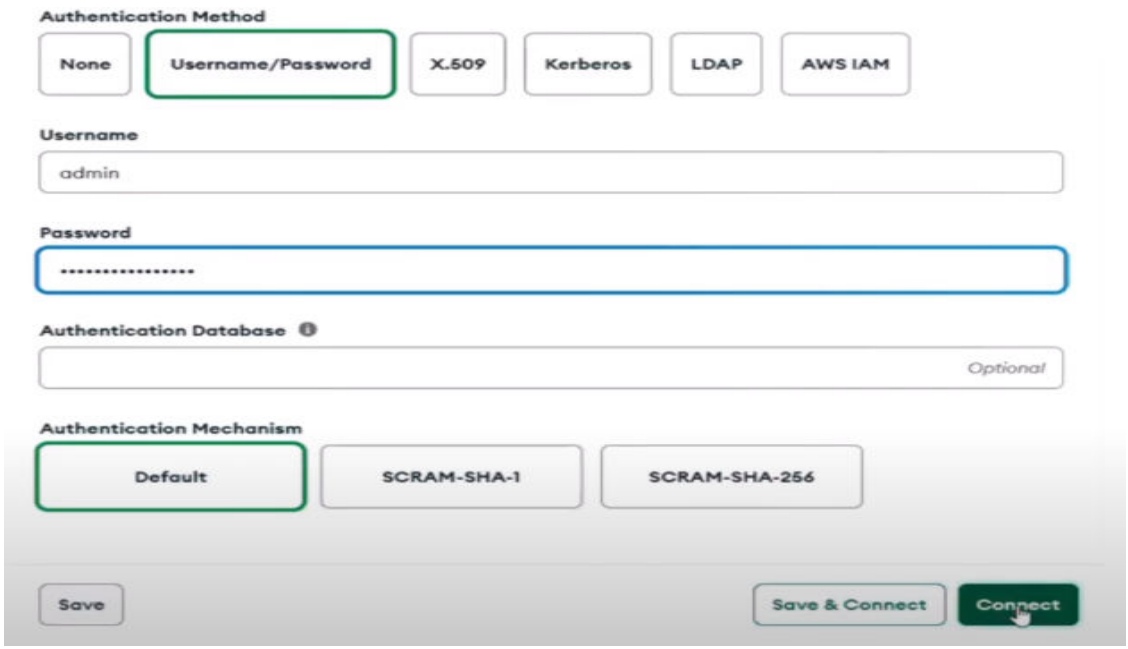


Then choose the version of the compass.

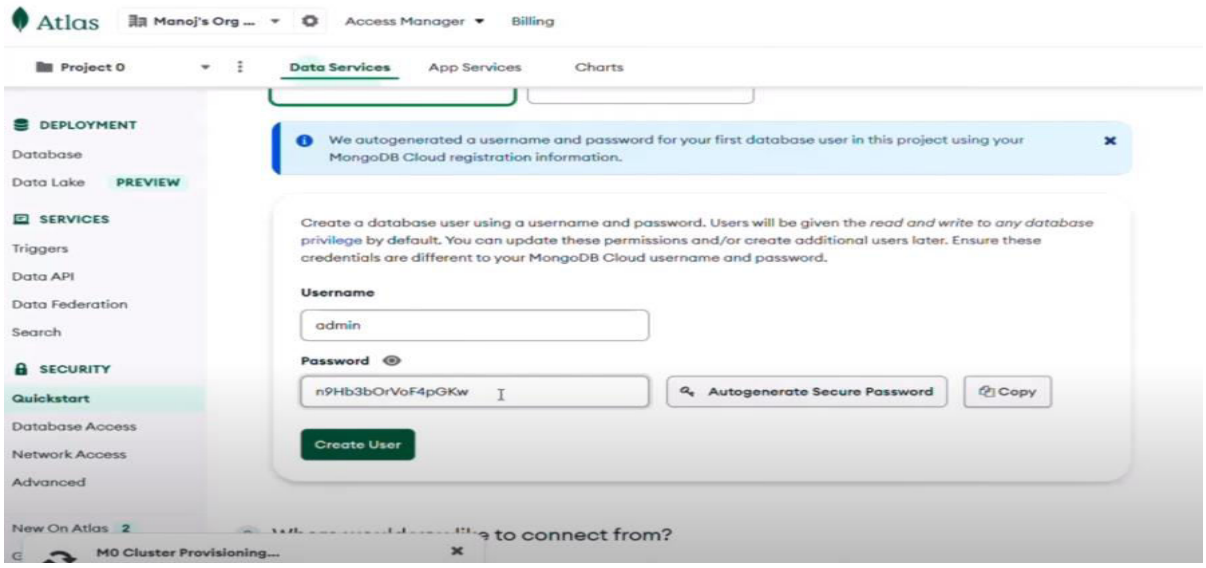
Copy the link which is written under copy the connection string and then go to MongoDB compass.



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

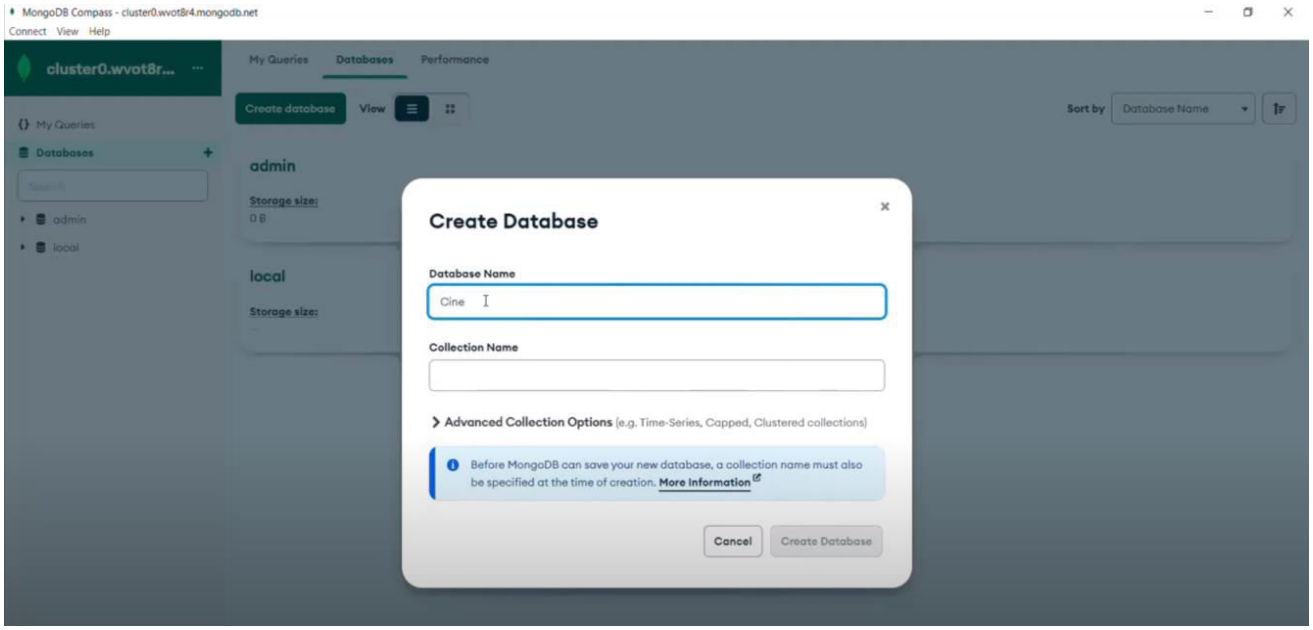


Then click on Authentication, then choose the authentication method as username/password. Those fill username and password the same as you have created in MongoDB Atlas. This one:-



Then click on create and your cluster has been connected.

Now, create a Database and name it:-



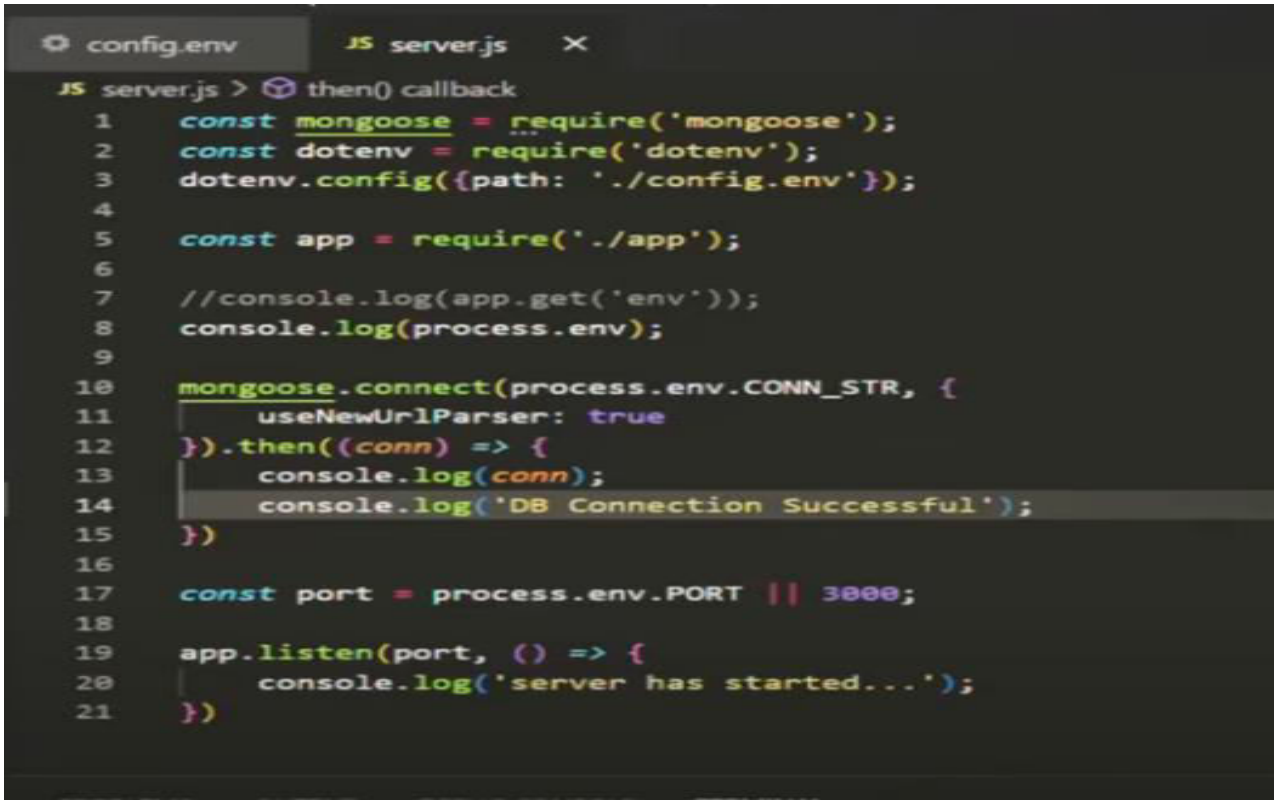
Connecting Remote Database From Express:-

Store the connection string in config.env file.

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:-



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.

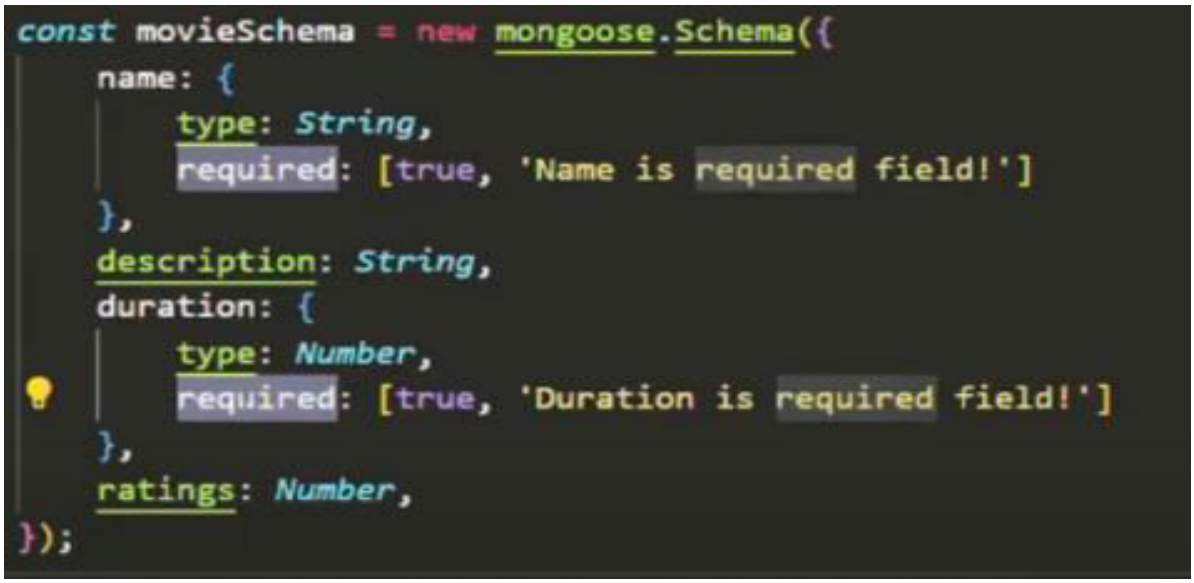
Creating a Schema and Model:-

"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:-

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

For ex:-



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 staring 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
        })
    }

}
```

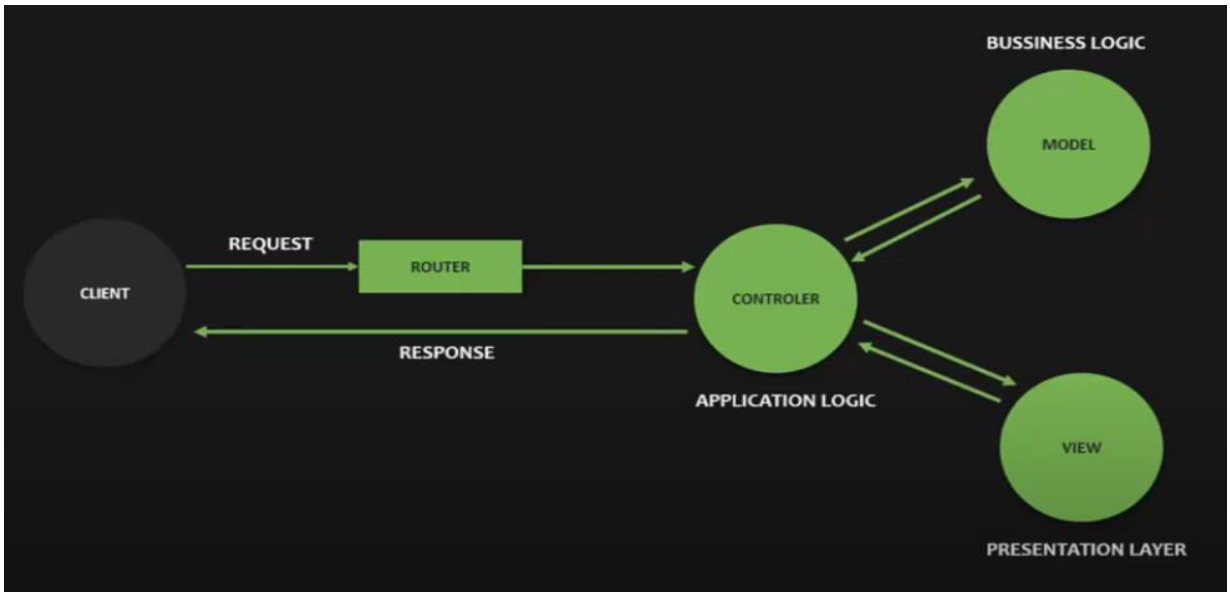
```
const express = require('express');
const moviesController = require('./../Controllers/moviesController');

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:

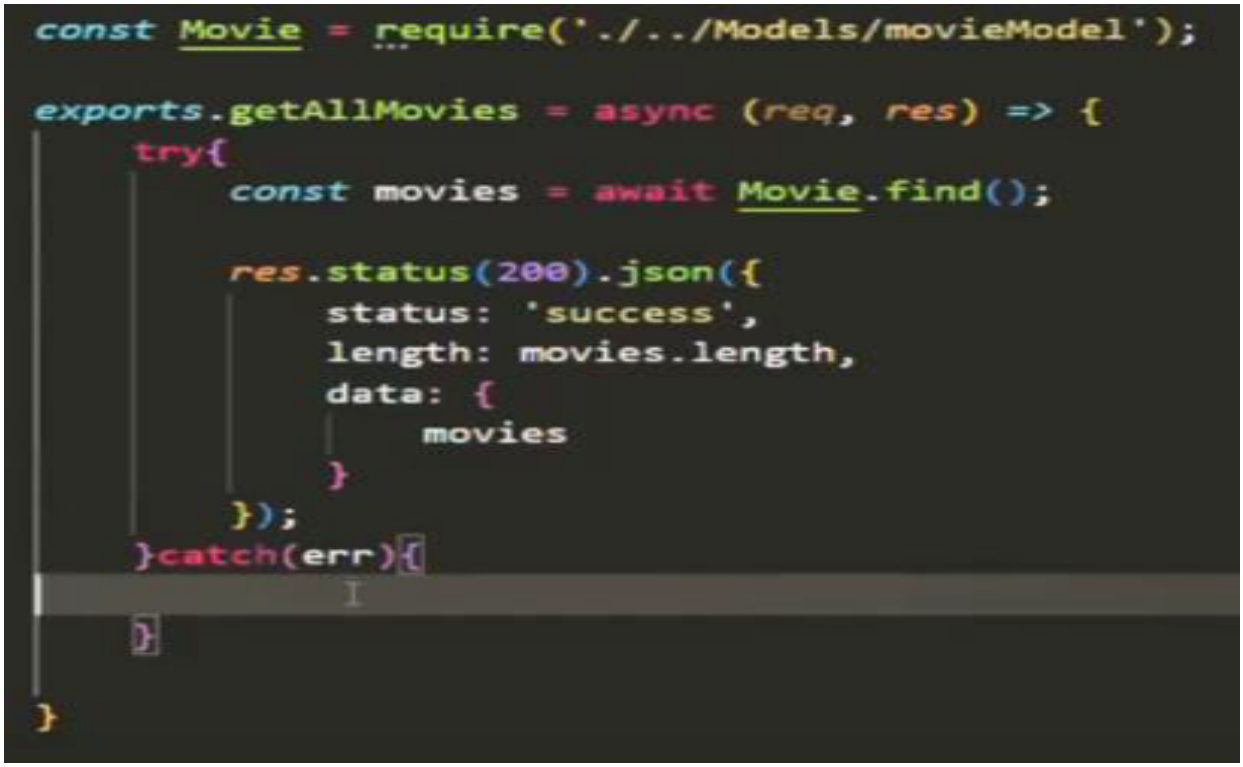
```
// Model
const userModel = require('./models/user');


// 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:-



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('-__v');
}
```

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 feching 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`:**

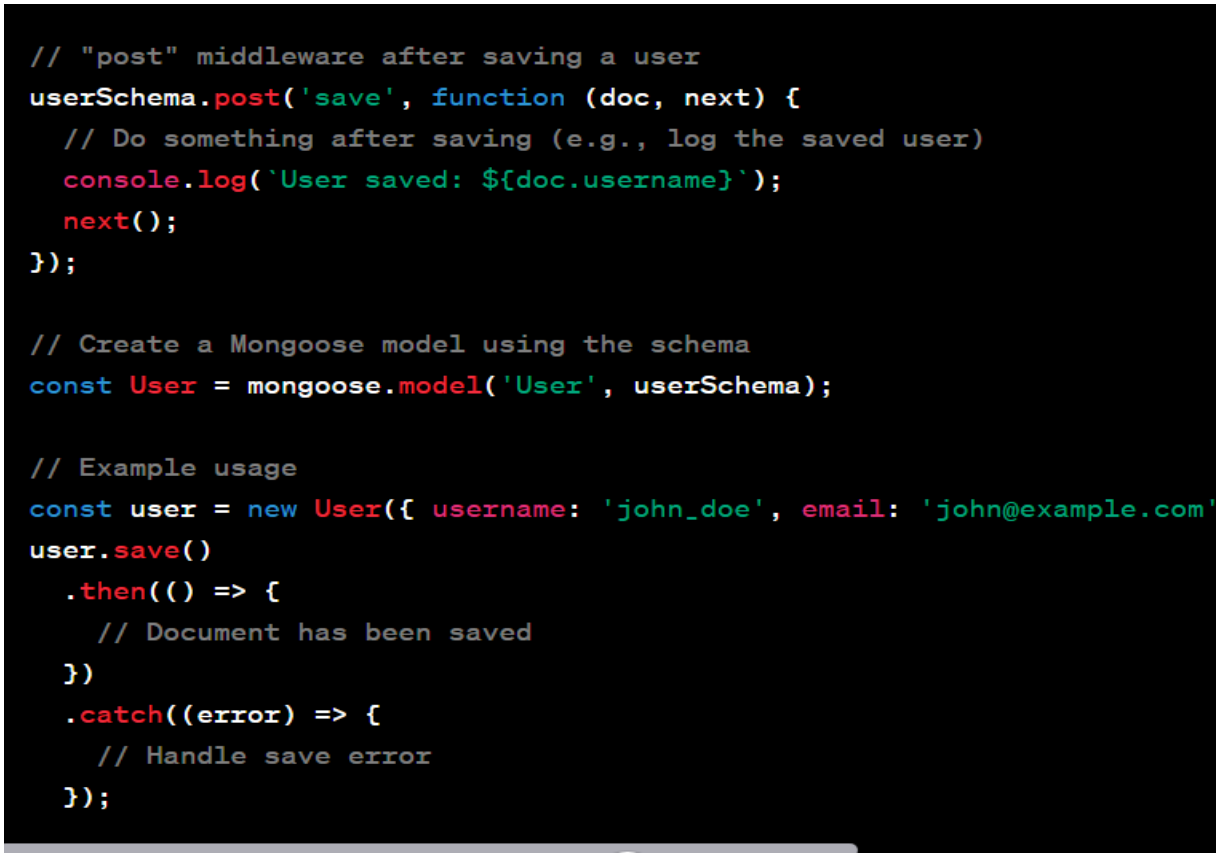
- Inside a document middleware function, the `this` keyword refers to the document being processed. This allows you to access and modify the document's properties.

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.

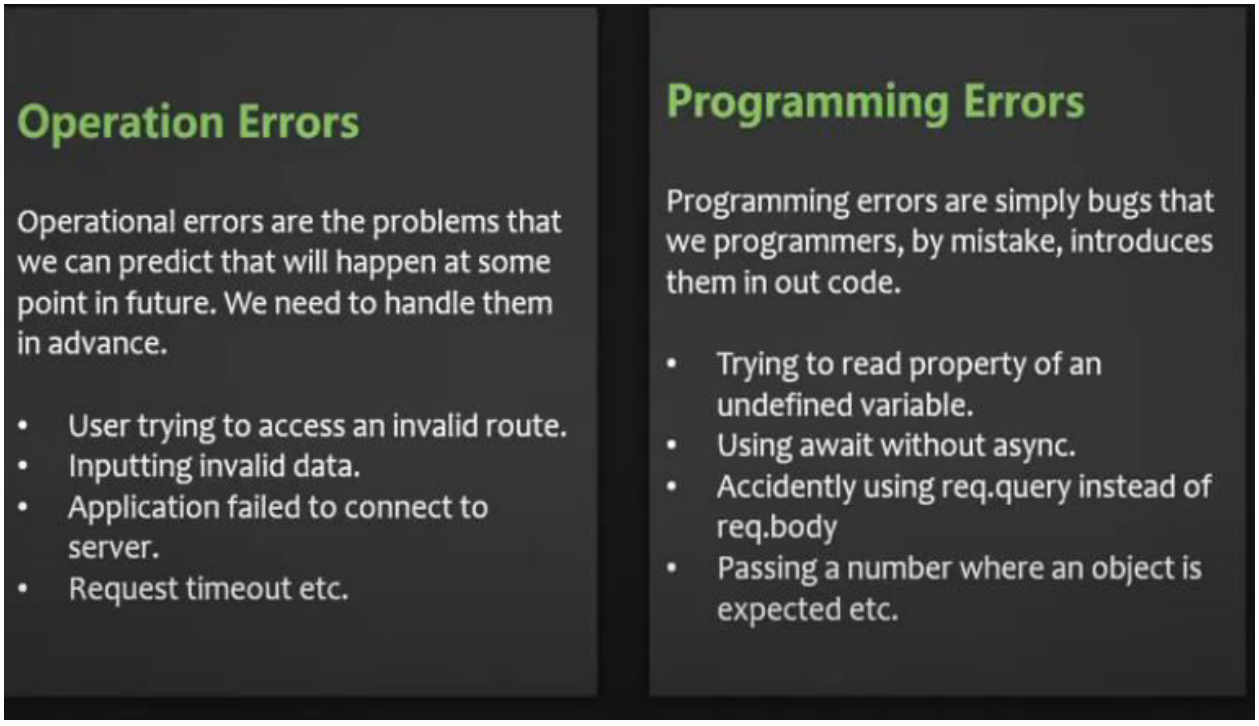
- **Custom Validators:**
  - `validate`: Allows you to define a custom validation function. It takes an object with a `validator` function and a `message` property for the error message.

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.

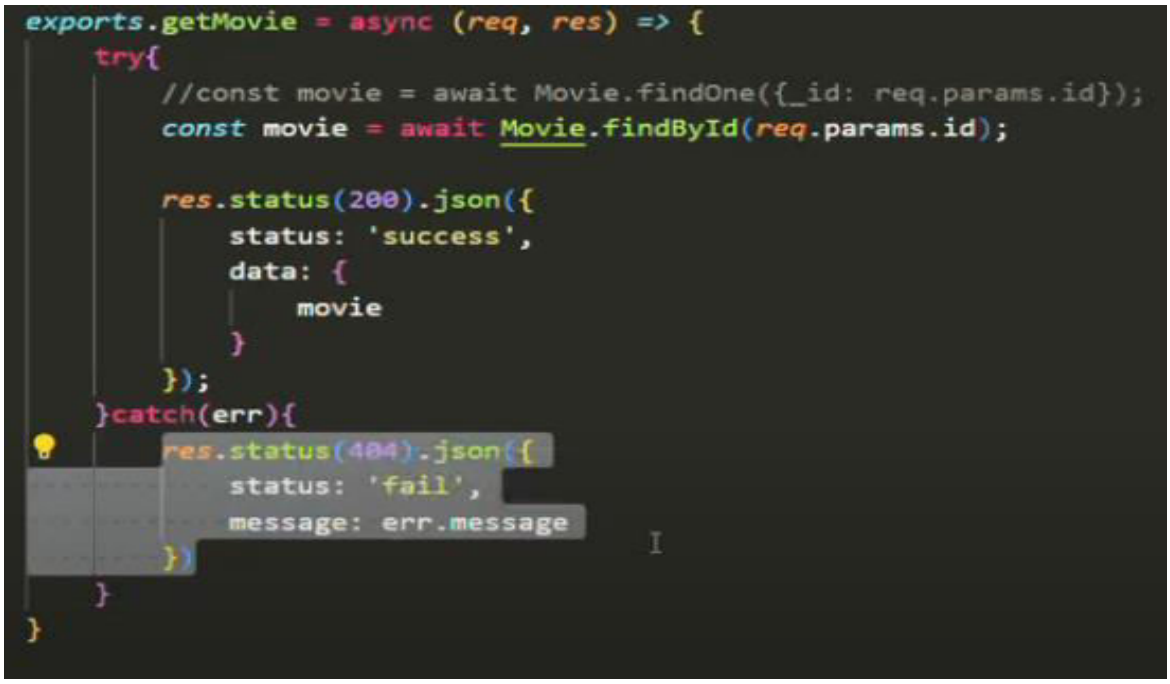
ERROR HANDLING:-



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 :-



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 individiually.

Syntax:-

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,

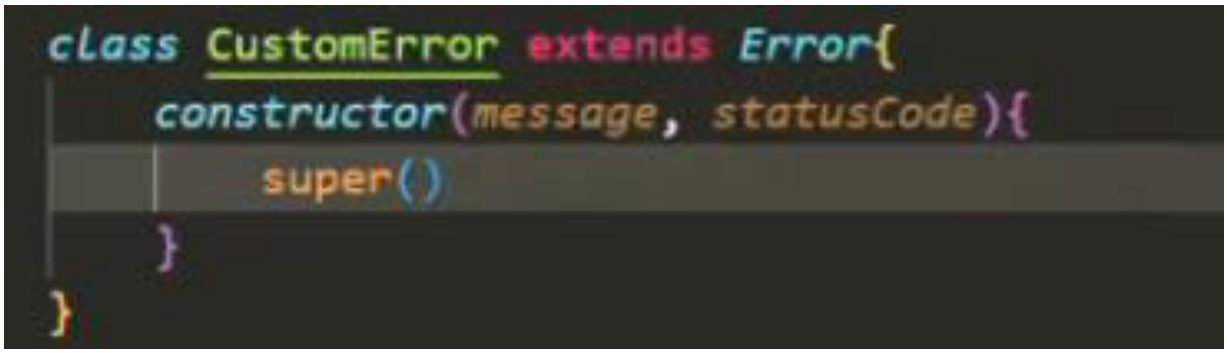
```
app.use('/api/v1/movies', moviesRouter);
app.all('*', (req, res, next) => {
    // res.status(404).json({
    //     status: 'fail',
    //     message: `Can't find ${req.originalUrl} on the server!`
    // });
    const err = new Error(`Can't find ${req.originalUrl} on the server!`);
    err.status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

We create own Error function:-



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);
```

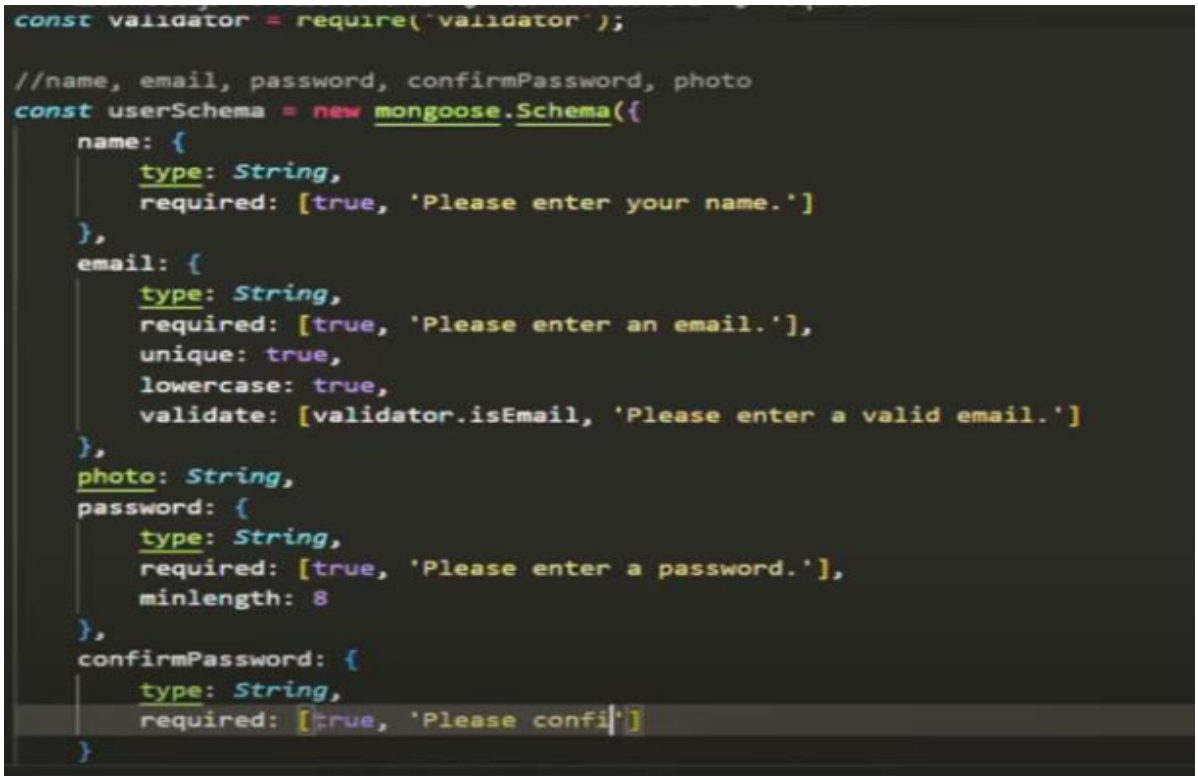
Error Handling in Async Function:-

```
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
        }
    })
});
```

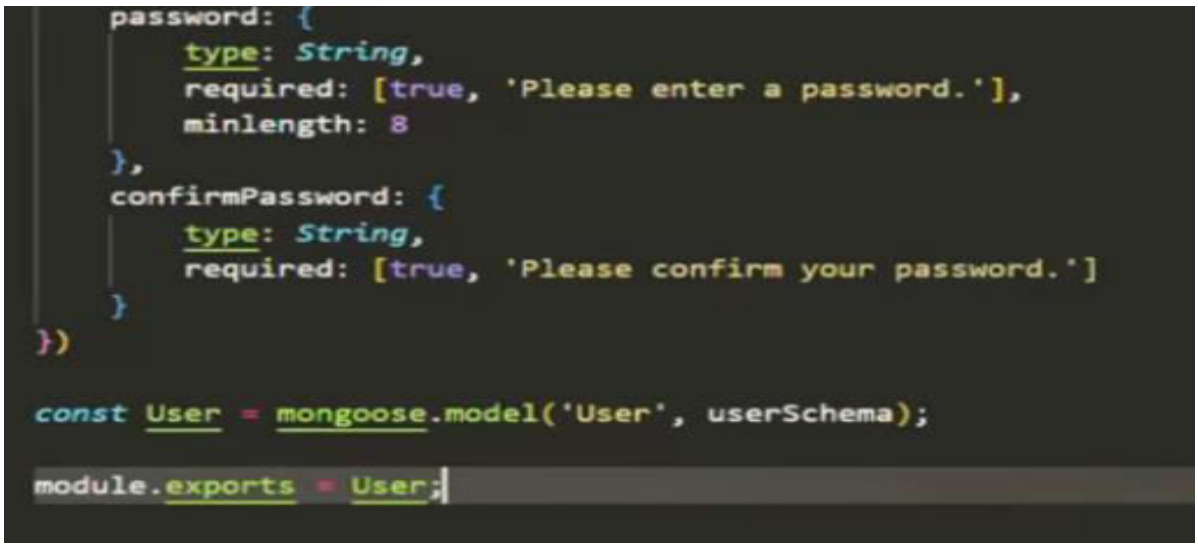
Instead of writing the catch method, we have created a global async error handling function in order to catch all the errors which going to be written inside it.

AUTHORIZATION AND AUTHENTICATION

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

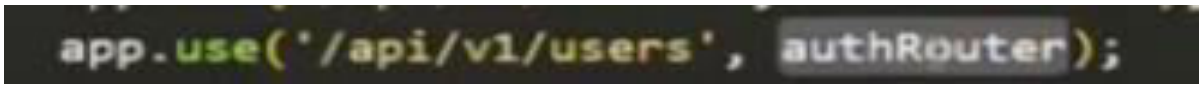


```
const validator = require('validator');

//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']
    }
```



```
    password: {
        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 mode 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();
})
```

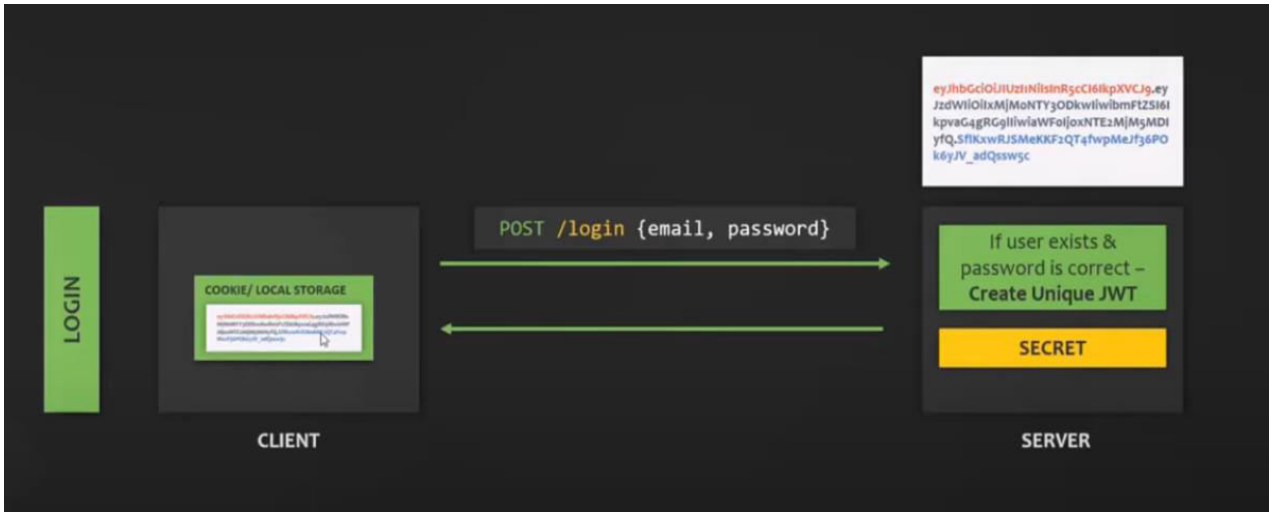
Here, bycrpt.hash() is used to encrpth the data , and in the 1st argument we to provide which data we want to

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:-

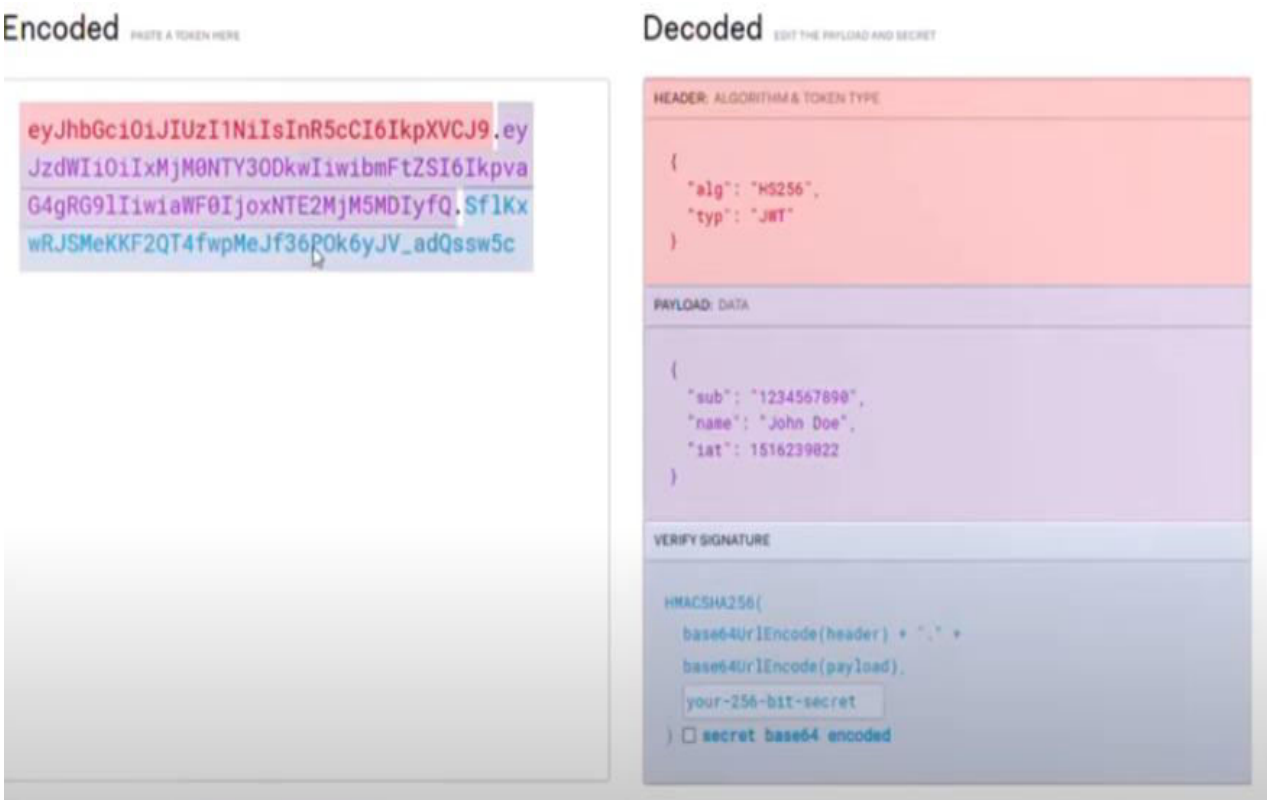


here, when we made a post request for login so if email and password is correct then a JWT token is created [white box] and a secret key is created then this JWT token is sended back to client which is stored under cookie/local storage.

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:
`eyJhbGciOiAiSFMyNTYiLCAidHlwIjogIkpXVCJ9`

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:

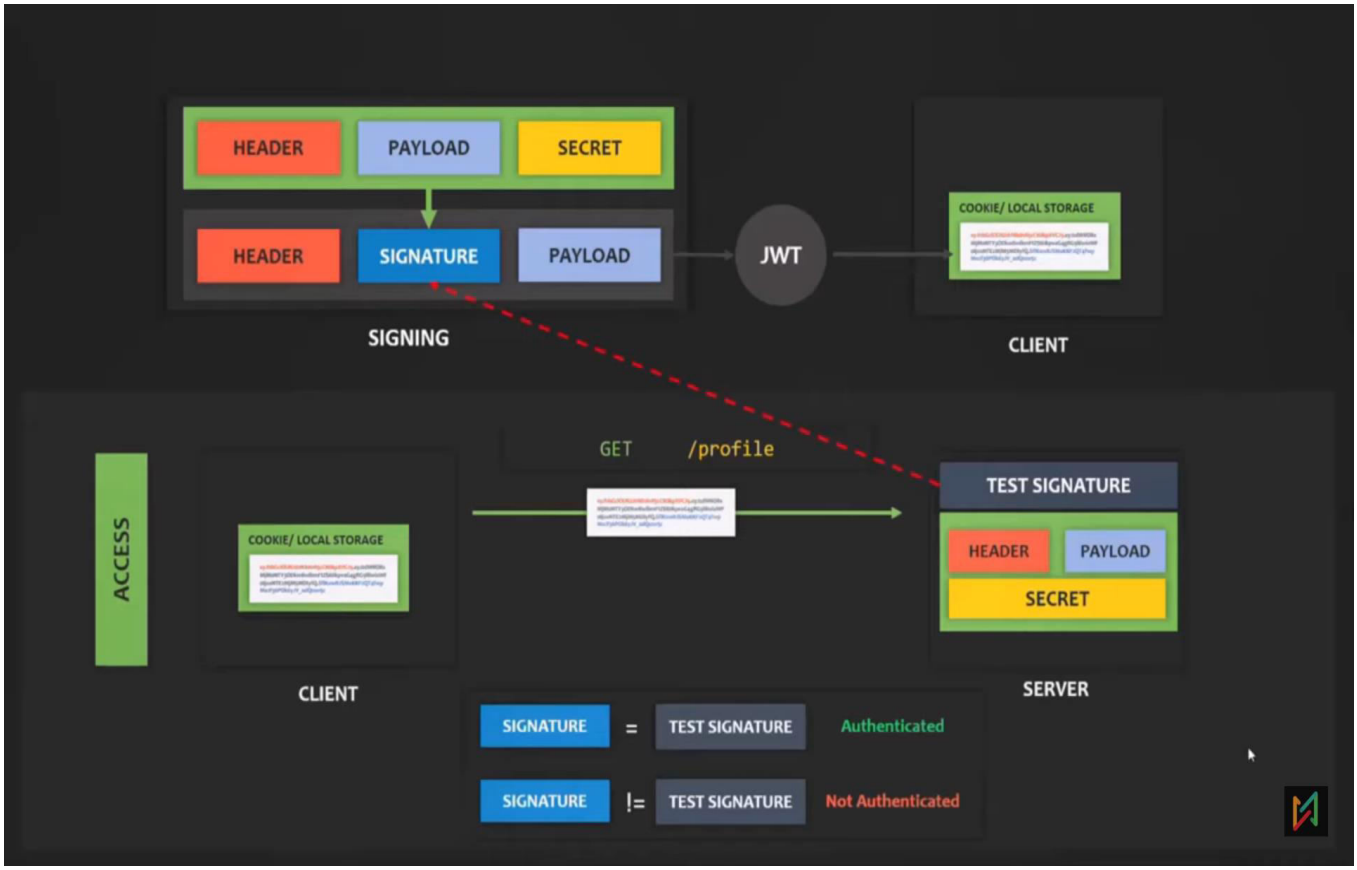
```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

After Base64Url encoding, it becomes something like:
`eyJzdWIiOiAiMTIzNDU2Nzg5MCIsICJuYW1lIjogIkpvaG4gRG9lIiwgImlhdCI6IDE1MTYyMzkwMjJ9`

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.