RESEARCH INNOVATION DISCOVERY (RID BHARAT)
A place for coder, developer & Researcher
RUN BY TWKSAA WELFARE FOUNDATION & ISO: 9001:2015

Reg.No: 048884
Foundation Day
05-09-2020

Node js / Express JS
Backend Notes

Er. Rajesh Prasad(B.E, M.E)
Founder: TWF & RID Org.

- **RID ORGANIZATION** यानि **Research, Innovation and Discovery** संस्था जिसका मुख्य उदेश्य हैं आने वाले समय में सबसे पहले **NEW (RID, PMS & TLR)** की खोज, प्रकाशन एवं उपयोग भारत की इस पावन धरती से भारतीय संस्कृति, सभ्यता एवं भाषा में ही हो |
- देश, समाज, एवं लोगों की समस्याओं का समाधान **NEW (RID, PMS & TLR)** के माध्यम से किया जाये इसके लिए ही मैं राजेश प्रसाद **इस RID संस्था** की स्थपना किया हूँ|
- Research, Innovation & Discovery में रूचि रखने वाले आप सभी विधार्थियों, शिक्षकों एवं बुधीजिवियो से मैं आवाहनं करता हूँ की आप सभी **इस RID संस्था** से जुड़ें एवं अपने बुध्दि, विवेक एवं प्रतिभा से दुनियां को कुछ नई **(RID, PMS & TLR)** की खोजकर, बनाकर एवं अपनाकर लोगों की समस्याओं का समाधान करें |

त्वक्सा Node JS के इस ई-पुस्तक में आप Node JS से जुड़ी सभी बुनियादी अवधारणाएँ सीखेंगे। मुझे आशा है कि इस ई-पुस्तक को पढ़ने के बाद आपके ज्ञान में वृद्धि होगी और आपको कंप्यूटर विज्ञान के बारे में और अधिक जानने में रुचि होगी

"In this E-Book of TWKSAA Node Js you will learn all the basic concepts related to Node Js. I hope after reading this E-Book your knowledge will be improve and you will get more interest to know more thing about computer Science".

Online & Offline Class:

Python, Web Development, Java, Full Stack Course, Data Science, UI/UX Training, Internship & Research

करने के लिए Message/Call करें. 9202707903 E-Mail_id: ridorg.in@gmail.com

Website: www.ridtech.in

RID हमें क्यों करना चाहिए ?

(Research)

अनुसंधान हमें क्यों करना चाहिए ?

Why should we do research?

1. नई ज्ञान की प्राप्ति (Acquisition of new knowledge)
2. समस्याओं का समाधान(To Solving problems)
3. सामाजिक प्रगति (To Social progress)
4. विकास को बढ़ावा देने(To promote development)
5. तकनीकी और व्यापार में उन्नति(To advances in technology & business)
6. देश विज्ञान और प्रौद्योगिकी के विकास(To develop the country's science & technology)

(Innovation)

नवीनीकरण हमें क्यों करना चाहिए ?

Why should we do Innovation?

1. प्रगति के लिए(To progress)
2. परिवर्तन के लिए(For change)
3. उत्पादन में सुधार(To Improvement in production)
4. समाज को लाभ(To Benefit to society)
5. प्रतिस्पर्धा में अग्रणी (To be ahead of competition)
6. देश विज्ञान और प्रौद्योगिकी के विकास। (To develop the country's science & technology)

(Discovery)

खोज हमें क्यों करना चाहिए?

Why should we do Discovery?

1. नए ज्ञान की प्राप्ति(Acquisition of new knowledge)
2. अविष्कारों की खोज(To Discovery of inventions)
3. समस्याओं का समाधान(To Solving problems)
4. ज्ञान के विकास में योगदान(Contribution to development of knowledge)
5. समाज के उन्नति के लिए (for progress of society)
6. देश विज्ञान और तकनीक के विकास (To develop the country's science & technology)

❖ Research(अनुसंधान):

- अनुसंधान एक प्रणालीकरण कार्य होता है जिसमें विशेष विषय या विषय की नई ज्ञान एवं समझ को प्राप्त करने के लिए सिद्धांतिक जांच और अध्ययन किया जाता है। इसकी प्रक्रिया में डेटा का संग्रह और विश्लेषण, निष्कर्ष निकालना और विशेष क्षेत्र में मौजूदा ज्ञान में योगदान किया जाता है। अनुसंधान के माध्यम से विज्ञान, प्रोधोगिकी, चिकित्सा, सामाजिक विज्ञान, मानविकी, और अन्य क्षेत्रों में विकास किया जाता है। अनुसंधान की प्रक्रिया में अनुसंधान प्रश्न या कल्पनाएँ तैयार की जाती हैं, एक अनुसंधान योजना डिज़ाइन की जाती है, डेटा का संग्रह किया जाता है, विश्लेषण किया जाता है, निष्कर्ष निकाला जाता है और परिणामों को उचित दर्शाने के लिए समाप्ति तक पहुंचाया जाता है।

❖ Innovation(नवीनीकरण): -

- Innovation एक विशेषता या नई विचारधारा की उत्पत्ति या नवीनीकरण है। यह नए और आधुनिक विचारों, तकनीकों, उत्पादों, प्रक्रियाओं, सेवाओं या संगठनात्मक ढंगों का सृजन करने की प्रक्रिया है जिससे समस्याओं का समाधान, प्रतिस्पर्धा में अग्रणी होने, और उपयोगकर्ताओं के अनुकूलता में सुधार किया जा सकता है।

❖ Discovery (आविष्कार):

- Discovery का अर्थ होता है "खोज" या "आविष्कार"। यह एक विशेषता है जो किसी नए ज्ञान, अविष्कार, या तत्व की खोज करने की प्रक्रिया को संदर्भित करता है। खोज विज्ञान, इतिहास, भूगोल, तकनीक, या किसी अन्य क्षेत्र में हो सकती है। इस प्रक्रिया में, व्यक्ति या समूह नए और अज्ञात ज्ञान को खोजकर समझने का प्रयास करते हैं और इससे मानव सभ्यता और विज्ञान-तकनीकी के विकास में योगदान देते हैं।

नोट : अनुसंधान विशेषता या विषय पर नई ज्ञान के प्राप्ति के लिए सिस्टमैटिक अध्ययन है, जबकि आविष्कार नए और अज्ञात ज्ञान की खोज है।

सुविचार:	
1.	**समस्याओं का समाधान करने का उतम मार्ग हैं |** → शिक्षा ,RID, प्रतिभा, सहयोग, एकता एवं समाजिक-कार्य
2.	**एक इंसान के लिए जरूरी हैं |** → रोटी, कपड़ा, मकान, शिक्षा, रोजगार, इज्जत और सम्मान
3.	**एक देश के लिए जरूरी हैं |** - → संस्कृति-सभ्यता, भाषा, एकता, आजादी, संविधान एवं अखंडता
4.	**सफलता पाने के लिए होना चाहिए |** → लक्ष्य, त्याग, इच्छा-शक्ति, प्रतिबद्धता, प्रतिभा, एवं सतता
5.	**मरने के बाद इंसान छोडकर जाता हैं |** → शरीर, अन-धन, घर-परिवार, नाम, कर्म एवं विचार
6.	**मरने के बाद इंसान को इस धरती पर याद किया जाता हैं उनके**
→ नाम, काम, दान, विचार, सेवा-समपर्ण एवं कर्मो से...	

आशीर्वाद (बड़े भैया जी)

Mr. RAMASHANKAR KUMAR

मार्गदर्शन एवं सहयोग

Mr. GAUTAM KUMAR

.....सोच है जिनकी नये कुछ कर दिखाने की, खोज हैं मुझे आप जैसे इंसान की.....

"अगर आप भी **Research, Innovation and Discovery** के क्षेत्र में रूचि रखतें हैं एवं अपनी प्रतिभा से दुनियां को कुछ नया देना चाहतें एवं अपनी समस्या का समाधान **RID** के माध्यम से करना चाहतें हैं तो **RID ORGANIZATION (रीड संस्था)** से जरुर जुड़ें" || धन्यवाद || **Er. Rajesh Prasad (B.E, M.E)**

What is Node.js?

- Node.js is an **open-source**, **cross-platform** JavaScript runtime environment that allows you to run JavaScript code **outside the browser** — typically on the **server side**.
- It uses the **V8 engine** (developed by Google for Chrome) to execute JavaScript code, making it **fast and efficient**. V8 engine + c++=node js
- With Node.js, you can build backend services like APIs, web servers, real-time apps (e.g., chat apps), and more — all using JavaScript.

❖ History of Node.js:

➢ Event Details

- **Created By** :→ Ryan Dahl
- **First Released** :→ 2009
- **Written In** :→ C, C++, JavaScript
- **Current Maintainer** :→ OpenJS Foundation (previously managed by Joyent)
- **Goal** :→ To create scalable, fast, and non-blocking web servers using JavaScript on the backend

❖ Key Features of Node.js

➢ **Feature** **Description**

- **Fast Execution** Uses Google's V8 engine which compiles JS to machine code
- **Event-Driven Architecture** Uses events and callbacks for asynchronous tasks
- **Non-blocking I/O** Handles many requests at once without waiting for one to complete
- **Single Language Stack** JavaScript for both frontend and backend
- **Huge Package Ecosystem** npm (Node Package Manager) offers 1M+ libraries
- **Modular** ArchitectureEasily break code into reusable modules
- **Built-in Modules** Like http, fs, path, etc. to build without external libraries

❖ How Node.js Works

- Node.js uses a **single-threaded event loop** and **non-blocking I/O** to handle multiple concurrent requests.

❖ Event Loop & Non-blocking I/O

- When a request comes in (e.g., reading a file or hitting an API), Node.js **does not wait** for it to finish.
- Instead, it assigns the task to a **worker thread** (background task) and **continues handling other requests**.
- Once the I/O operation is done, the result is passed back via a **callback**.

➢ This makes Node.js highly scalable — it can handle thousands of requests with a single thread.

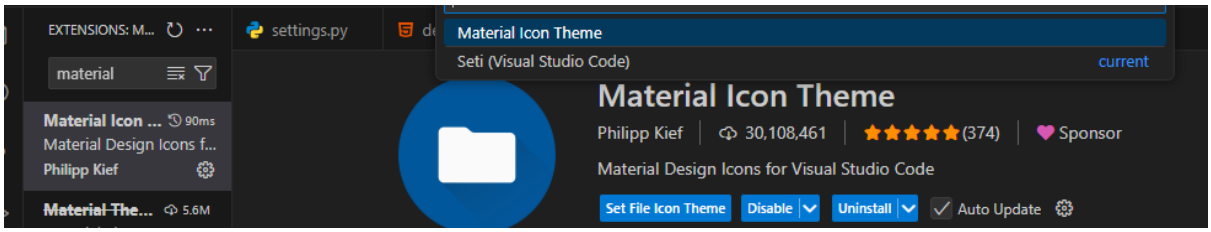
Node.js vs Other Backend Technologies

➢ Feature	Node.js	PHP / Python / Java
• Language	JavaScript	PHP, Python, Java, etc.
• Concurrency	Non-blocking, Event-driven	Multi-threaded, blocking I/O
• Speed	Very fast (V8 engine)	Slower (depends on language)
• Threading	Single-threaded (event loop)	Multi-threaded
• Use Case	Real-time apps, APIs	CMS, ML, large enterprise
• Community	Very active, npm ecosystem	Large (especially for Java, Python)
• Learning Curve	Easier for JavaScript devs	Depends on the language

How to install Node.js & npm

- **What is NPM?**
  - NPM (Node Package Manager) is automatically installed with Node.js. It is used to manage **JavaScript packages/libraries**.
- **What is a Package? - it** is a folder containing a package.json file and code (modules), used to add specific functionality to your Node.js project.
  Example: express, mongoose, nodemailer
- **What is a Library in Node.js?**
  - A library is a collection of reusable code that performs common tasks, which you can import and use in your Node.js app.
    - **Example:** lodash (for utilities), axios (for HTTP requests)
  - **How to Install:**
  - **Windows**
    1. Go to: https://nodejs.org
    2. Download the **LTS (Long Term Support)** version
    3. Run the installer → Accept default settings → Finish
- **Verify Installation:**
  - node -v # Checks Node.js version
  - npm -v # Checks NPM version
- **Running a Simple Node.js Script**
  1. Open any code editor (like VS Code)
  2. Create a file named app.js
  3. Write a simple script:
     console.log("Hello, Node.js!");
  4. Run the script in terminal:
     node app.js
  - **Output will be**: Hello, Node.js!

Note: - Material Icon Theme most popular extension used for file and folder icons in vs code



How to run the first Node js program

1. **Create a JavaScript file**
   Example: app.js
2. **Write code in app.js**
   Example: console.log ("Hello Everyone welcome to rid Bharat")
3. **Open terminal in the file's folder**
   In VS Code: Right-click → "Open in Integrated Terminal"
   Run the file: - node app.js
   Output: - Hello everyone welcomes to rid Bharat

Nodemon, npm start and npm init in Node.js

❖ What is Nodemon?

- **Nodemon** is a tool that **automatically restarts your Node.js app** whenever you make changes to your code.

❖ How to Install Nodemon (via npm)

Option 1: Install Globally(for any project) Check if installed:

- ➢ npm install -g nodemon
- ➢ nodemon -v (it will node.js current version)

Option 2: Install Locally (only in one project) > npm install --save-dev nodemon

❖ What is npm start?

- npm start is a command that runs **start script** defined in your package.json file.

❖ **Why we use it?:-** To **easily start** your Node.js application with a simple command.

❖ What is npm init?

- npm init is a command used to **create a package.json file** in a Node.js project.

Why we use it?

- To define the project's metadata (name, version, description, etc.)
- To manage **dependencies**, **scripts**, and **project configuration**
- It's required for installing and managing packages with npm

Note-1: - It helps organize and track everything your Node.js project needs.

Note-2: package.json file (created by npm init) works like the **overall project settings/configuration file** in a Node.js project.

❖ for open package.json file Run this command in your project folder

- ➢ npm init -y

❖ in package.json, add this script:

```
"scripts": {
   "start": "nodemon app.js"
      }
```

Before adding nodemon

```
{
  "name": "djangopro",
  "version": "1.0.0",
  "main": "demo.js",
  Debug
  "scripts": {
    "test": "echo \"Error: no test
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

After adding nodemon

```
{
  "name": "djangopro",
  "version": "1.0.0",
  "description": "",
  "main": "demo.js",
  Debug
  "scripts": {
    "start": "nodemon demo.js",
    "test": "echo \"Error: no test
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

❖ Why we use it:

1. **Simplifies your command**

Instead of typing this every time:

- ➢ nodemon app.js

You can just type:

- ➢ **npm start**

Note: - You can even change "start" to run any file you want (like index.js, server.js, etc.).

>> npm init

- **package name:** The project's name (default node_js; you wrote server.js which is usually a filename).
- **version:** Project version (default 1.0.0).
- **description:** Short info about the project (new project).
- **git repository:** URL of your project's Git repo (optional, left blank).
- **keywords:** Words to help find your project (optional).
- **license:** License type (default ISC).

These details create your project's package.json file.

```
{
  "name": "djangopro",
  "version": "1.0.0",
  "description": "",
  "main": "demo.js",
  Debug
  "scripts": {
    "start": "nodemon demo.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Note: npm init -y is used to quickly create a default package.json file with all standard settings without asking questions.

Module in Node.js

(Framework → Module → Function → Code → Instruction → Command)

Framework Hierarchy

1. **Framework** – collection of modules. (**Installed in the project**)
2. **Module** – A collection of functions. (**Imported into files**)
3. **Function** – A block of code. (**Needs to be called to run**)
4. **Code** – A set of instructions. (***runs line by line***)
5. **Instruction** – A single command. (**Runs directly**)

(write in any programming language like **JS or python**)

Examples:

1. **Framework:** Express, NestJS (**installed via npm**).
2. **Module:** Core (fs, path) or third-party (**axios, lodash**).
3. **Function:** Callable blocks like fs.readFile(), app.get().
4. **Code:** Logical blocks (e.g., if, loops).
5. **Instruction:** Single statements (console.log ("learn Coding)).

predefined methods/functions generally we called: Technologies & Their Predefined Function Names

1. **React.js** – Hooks
2. **Django** – Modules / Packages
3. **Node.js** – Core Modules
4. **Express.js** – Middleware / Router Methods
5. **Python** – Built-in Functions
6. **JavaScript** – Built-in Methods / Web APIs
7. **Java** – Standard Library Methods
8. **C++** – Standard Library Functions (STL)
9. **Angular** – Services / Lifecycle Hooks
10. **Flask** – Modules / Built-in Functions
11. **Ruby on Rails** – Helpers / Gems
12. **.NET (C#)** – Framework Classes
13. **Spring Boot** – Annotations / Beans

How to Create, Export, and Import Modules in Node.js

1. **Using import (ES Modules) :- Syntax import / export**
2. **Using require (CommonJS) :- require / module.exports**

1. Using import (ES Modules)

Step 1: Create module with any name with .js and write some function like : service.js e.g., add(), multiply()

Step-2: Export functions in botton in service.js file export { function-1, function-2,.... } **Ex**: export { add, multiply }

Step-3: **Update** package.json

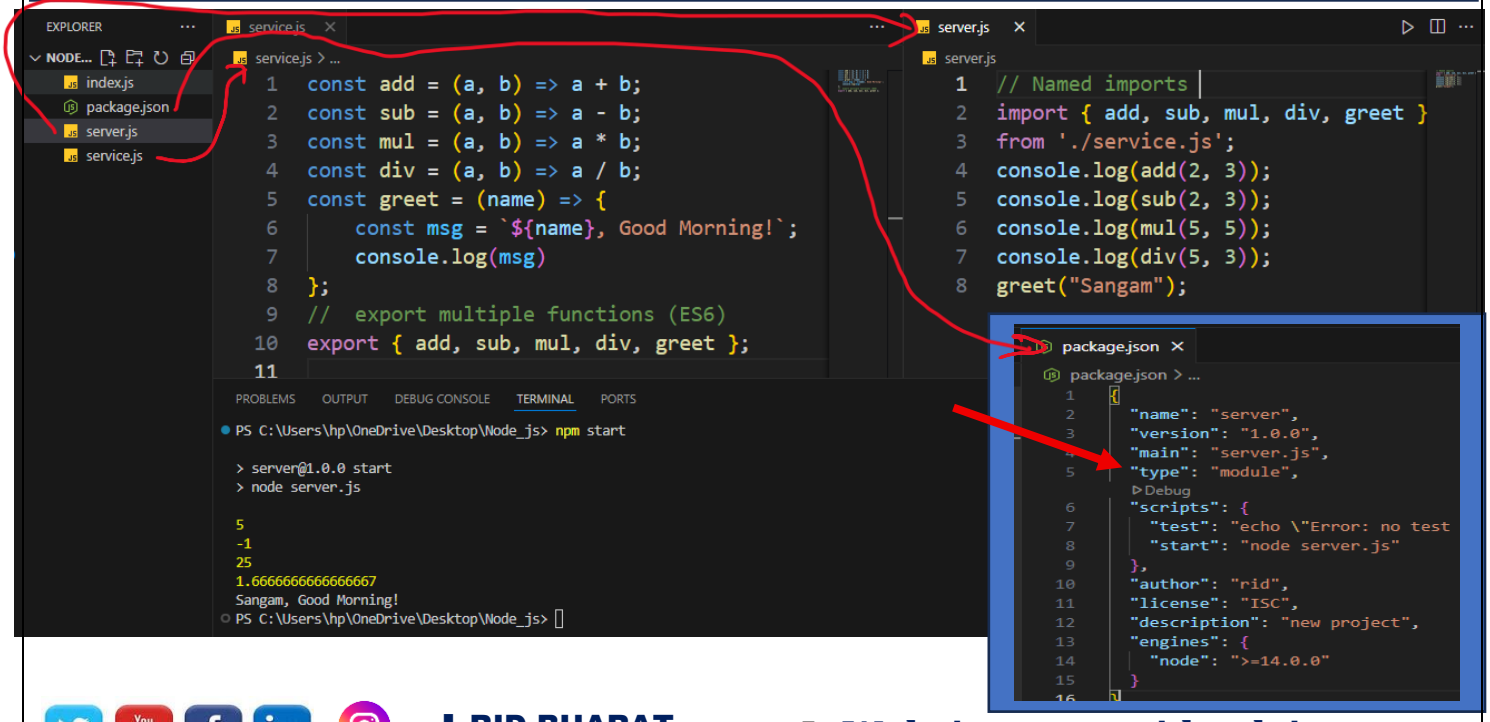
Example:- { "type": "module", **// Enable ES Modules** }

Step-3: **Create server.js** to use the exported functions.

Step-4: **Import functions** in server.js using: import { function-1, function-2, ... } from './service.js'

Example: import { add, multiply } from './service.js'

Step-5: **Run the code** using: node server.js or npm start in terminal.

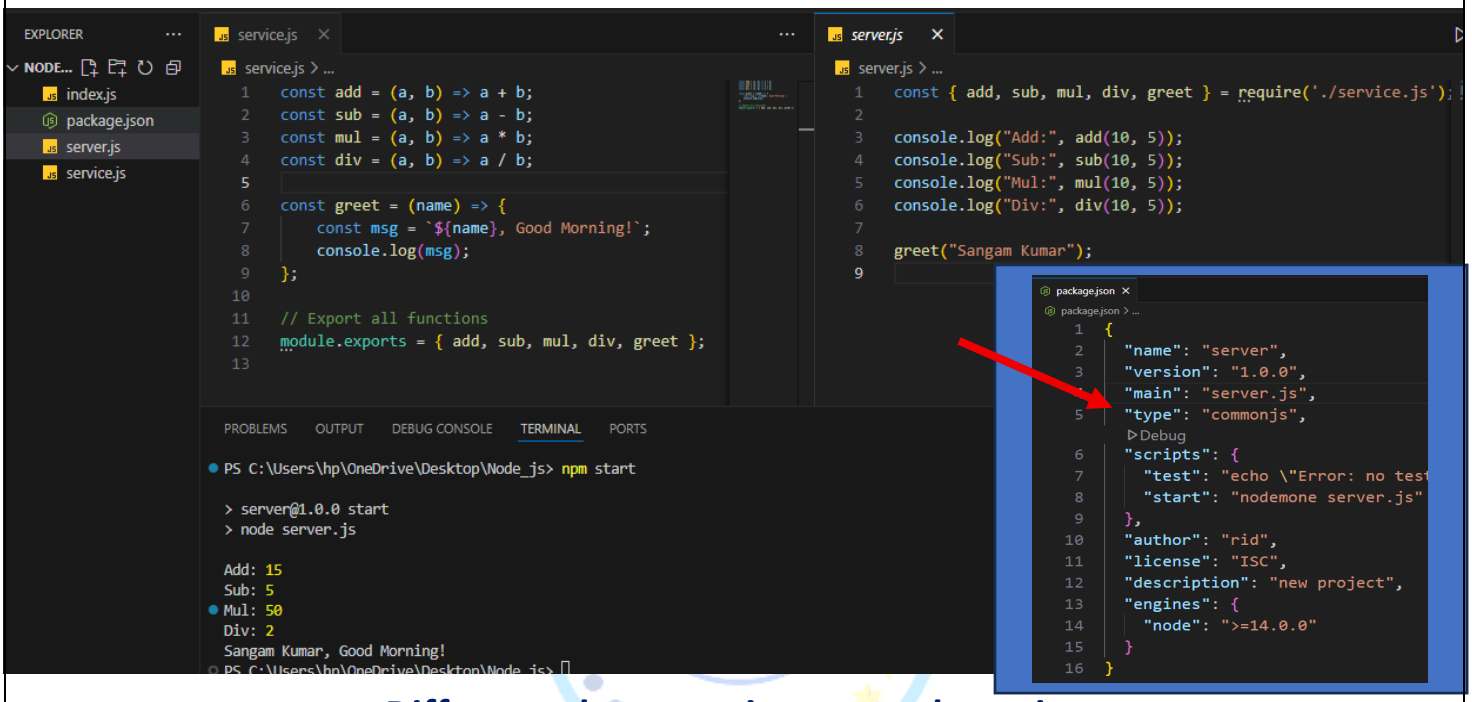


2. Using require (CommonJS)

How to Create, Export, and Import Modules in Node.js using require

1. **Create** service.js and write functions (e.g., add(), multiply()).
2. **Export** functions using: module.exports = { add, multiply };
3. **No need** to update package.json (CommonJS is default).
4. **Create** server.js to use the module.
5. **Import** functions using: const { add, multiply } = require('./service.js');
6. **Run using:** node server.js or npm start

Example:



Difference between import and require

1. Using ES Modules (ESM) — import / export

- **Syntax:** Uses import and export keywords.
- **File type:** Treated as ES modules when "type": "module" is set in package.json or using .mjs extension.
- **Modern** standard supported by browsers and Node.js (from v12+).
- **Supports static analysis** (imports resolved at compile time).
  Example: export function add(a, b) { return a + b; }
- **import** { add } from './service.js';

2. Using CommonJS — require / module.exports

- **Syntax:** Uses require() and module.exports.
- **Default** module system in Node.js before ES modules support.
- **Modules** loaded **dynamically at runtime**.
- **Widely** used in existing Node.js projects and many npm packages.
  Example: function add(a, b) { return a + b; }
- **module**.exports = { add };
- **const { add }** = require('./service.js');

Note: - ES Modules are more useful and preferred in modern development because they align with JavaScript standards and browser compatibility.

Promises in JavaScript / Node.js

❖ What is a Promise?

- A **Promise** is a way to handle things that happen **later** (asynchronous operations) like getting data from a server, reading a file, or waiting for a timer.

It says: I promise to give you result **later**. Either I will **fulfill** promise (give result), or I will **reject** it (if there's an error).

➔ **Promise JavaScript** में एक वादा होता है, जैसे: "मैं तुम्हें भविष्य में रिजल्ट दूँगा - या तो सही (fulfill) या गलत (reject)

- ➢ रियल-लाइफ उदाहरण: मान लो आपने फोन का ऑर्डर दिया:
- ➢ Promise बनता है → दुकानदार कहता है: "3 दिन में डिलीवर कर दूँगा!"

❖ दो ही हालात हो सकते हैं:

1) Fulfilled (पूरा हुआ) → 3 दिन में फोन मिल गया!
2) Rejected (रिजेक्ट) → दुकानदार ने कहा "स्टॉक खत्म हो गया!"

❖ Promise के 3 States:

1) **Pending** → ऑर्डर प्रोसेस हो रहा है (Wait).
2) **Fulfilled** → वादा पूरा हुआ (Success).
3) **Rejected** → वादा टूट गया (Error).

Example:
दुकानदार का वादा (Promise) → "कल तक फोन दे दूँगा!"
.then() → अगर फोन मिला, तो खुशी!
.catch() → अगर नहीं मिला, तो शिकायत!
Promise = भरोसा + इंतज़ार का तरीका!

Syntax:

```
let newdata = new Promise((resolve, reject) => {
  // Async task here
  resolve("Success"); // or reject("Error");
});
newdata  // variable name
  .then(result => console.log(result)) // On success
  .catch(error => console.log(error)) // On failure
  .finally(() => console.log("Done")); // Always runs
```

1. **new Promise()** → Creates a promise object.
2. **new Promise() takes a callback with two arguments:**
   - **resolve** → Marks promise as fulfilled (success).
   - **reject** → Marks promise as rejected (failure).
3. **.then()** → Runs when resolved (success) and handles the resolved value.
4. **.catch()** → Runs when rejected (error) and handles error.
5. **.finally()** → Runs always, after then or catch, regardless of success or failure.

Example-1

new keyword with `Promise` in JavaScript to **create a new Promise object**. **new keyword** with Promise in JavaScript to **create a new Promise object**.

server.js

```
let data = new Promise((resolve, reject) => {
    let haveSkills = true; // Change true to flase
    if (haveSkills) {
        resolve("You got the job! ");
    } else {
        reject("No job – improve your skills first!");
    }
});
data .then((result) => console.log(result))  // On success
    .catch((error) => console.log(error))   // On failure
    .finally(() => console.log("Job application process finished"));
//Note: .then((result) => console.log(result))
// represents the value passed to resolve() – in this case, "You got the job!"
// when the promise is fulfilled.
// If the promise is rejected, result is not used – instead, the .catch(error)
// block handles the value from reject()
```

Example-2

demo.js

Lec-3 > Project folder > demo.js > orderFood

```
// Function to order food without Promises
function orderFood(success, error) {
//   const shopopen = true;
const shopopen = false;

  if (shopopen) {
    success("Pizza will deliver on time ");
  } else {
    error("Shop is closed!");
  }
}
// Call the function
orderFood(
  (msg) => {
    console.log(msg); // If success
  },
  (err) => {
    console.error(err); // If error
  }
);
```

Call back function

demo1.js

Lec-3 > Project folder > demo1.js > orderFood

```
// Function to order food with Promise
function orderFood() {
  const shopopen = false;
  return new Promise((resolve, reject) => {
    if (shopopen) {
      resolve("Pizza will deliver on time");
    } else {
      reject("Shop is closed!");
    }
  });
}
orderFood()
  .then((msg) => {
    console.log(msg); // If success
  })
  .catch((err) => {
    console.error(err); // If error
  });
```

We use new because Promise is a **JavaScript class (constructor function)** — and to create an object from a class, we must use the new keyword.

Example-2

demo.js

Lec-3 > Project folder > demo.js > ...

```
function getData(pra) {
    setTimeout(() => {
        pra("Data received after 2 sec");
    }, 2000);
}

getData(function(result) {
    console.log(result);
});
```

Note: resolve and reject are **not reserved keywords** in JavaScript we can write any things

demo1.js

Lec-3 > Project folder > demo1.js > then() callback

```
function getData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Data received after 2 sec");
            // reject("Something went wrong");
        }, 2000);
    });
}
getData()
    .then(result => {
        console.log(result);
    })
    .catch(error => {
        console.log(error);
    });
```

```
new Promise((yes, no) => {
  yes("Task done");
  no("Something went wrong");
});
```

service.js

service.js > ...

```
function getdata(pra) {
    success = pra
    const a = new Promise((accept, reject) => {
        setTimeout(() => {
            if (success) {accept("You are accepted "); }
            else {reject("You are rejected "); }
        }, 2000);
    });
    return a;
}
getdata(false) // change argumets
    .then((p) => {console.log("Success:", p);})
    .catch((e) => {console.log("Error:", e);})
    .finally(() => {console.log("I will run in any condition");});
```

PROBLEMS DEBUG CONSOLE TERMINAL PORTS Code

```
Error: You are rejected
I will run in any condition
[nodemon] clean exit - waiting for changes before restart
```

Example-3 for promises: (main)

demo6.js

```
let a=10
let b=20
let res=a+b
let resserver=fetch(
    "https://jsonplaceholder.typicode.com/posts"
)
console.log(resserver)
```

```
PROBLEMS  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\hp\OneDrive\Desktop\Javascript> node demo6.js
Promise { <pending> }
PS C:\Users\hp\OneDrive\Desktop\Javascript>
```

Example 1: Fetch users list

```
let a = 10;
let b = 20;
let users = fetch("https://jsonplaceholder.typicode.com/users");
console.log(users);
```

Example 2: Fetch random dog image

```
let x = 5;
let y = 15;
let dogs = fetch("https://dog.ceo/api/breeds/image/random");
console.log(dogs);
```

Example 3: Fetch random joke

```
let p = 50;
let q = 100;
let joke = fetch("https://official-joke-api.appspot.com/random_joke");
console.log(joke);
```

Example 4: Fetch random meal recipe

```
let f1 = 5;
let f2 = 10;
let meal = fetch("https://www.themealdb.com/api/json/v1/1/random.php");
console.log(meal);
```

Example 5: Fetch list of categories of meals

```
let f3 = 15;
let f4 = 20;
let mealCategories = fetch("https://www.themealdb.com/api/json/v1/1/categories.php");
console.log(mealCategories);
```

Synchronous (Sync) and Asynchronous (Async) & await in Node.js

❖ Synchronous (Sync):

- Code runs one step at a time, and each task must finish before the next starts.
- Tasks are executed sequentially, one after another.
- If a task takes time, it blocks the rest of the code from executing.
- **Note:** Synchronous (Sync) is not a variable, keyword, function, or method.
- **It's a concept** in programming that describes how code executes in a sequential, blocking manner.
- **Note:** Blocking manner → When a task stops the execution of the next tasks until it is completed.

Example: real-time example for Synchronous (Sync) Making tea step by step

- **Boil water**. → Must finish before next step
- **Add tea leaves.** → Waits until water is boiled
- **Pour tea into cup.** → Waits until tea is ready
- **Drink tea.** → Waits until tea is poured

Explanation: Each step waits for the previous one to finish.
Nothing else can happen until the current step is completed.
This is exactly how synchronous/blocking code works in programming.

Example: Coding Example in js

Example 1
```
console.log("Step 1");
console.log("Step 2");
console.log("Step 3");
```

Example 2
```
let sum = 5 + 3;
console.log("Sum:", sum);
console.log("Next task");
```

Example 3
```
for(let i=1; i<=3; i++){
  console.log("Loop iteration:", i);
}
```

Note: by default, normal JavaScript code runs synchronously

❖ Asynchronous (Async):

- **Code** can run multiple tasks at the same time without waiting for previous ones to finish.
- **Time-consuming** tasks (like API calls, file reading, or timers) run in the background.
- **Async** code does not block the rest of the program.
- **Uses callbacks**, Promises, or async/await to handle results.
- **Note:** Asynchronous (Async) is not a variable, keyword, function, or method.
- **It's a concept** in programming that describes how code executes in a non-blocking manner.
- **Note:** Non-blocking manner → When a task runs in the background while the rest of the code continues executing.

Example: Real-time example for Asynchronous (Async) Making tea while doing other tasks:

- **Boil water.** → Starts, but you don't wait
- **Wash** cups while water is boiling → Runs at the same time
- **Add tea** leaves when water is ready → Runs after boiling, but other tasks already done
- **Pour** tea into cup → Runs after tea is ready
- **Drink** tea → Final step

Explanation: Tasks like washing cups can run while water is boiling.
Nothing is blocked; multiple things can happen at the same time.
This is exactly how asynchronous/non-blocking code works in programming.

Example: Coding Example in JavaScript

Example 1 - Using setTimeout without promises

```
console.log("Step 1: Start boiling water");
setTimeout(() => {
  console.log("Step 2: Water boiled");
}, 2000); // Runs after 2 seconds
console.log("Step 3: Wash cups");
```

Output:
```
Step 1: Start boiling water
Step 3: Wash cups
Step 2: Water boiled
```

Example-2: Using Promise

```
let data = new Promise((resolve) => {
  setTimeout(() => resolve("Step 2: Promise resolved!"),
1500);
});
data.then(result => console.log(result));
console.log("Step 3: Do other tasks");
```

Output
```
Step 3: Do other tasks
Step 2: Promise resolved!
```

await

await

- **await** is a **JavaScript keyword** with a specific function: it pauses an async function until a Promise is resolved.
- Used **only inside an async function**.
- Makes **asynchronous code look and behave like synchronous code**.
- **Note:** A **concept** would describe an idea like "asynchronous programming" or "blocking vs non-blocking," not a specific keyword in the language.

Example: Waiting for food delivery

1. You **order food** (this is like creating a Promise).
2. You **wait** for the food to arrive before eating (this is like await).
3. Once the food arrives, you **eat it** and then do the next task.
   Order food → Wait for delivery → Eat food → Do other tasks

Explanation:

- await is like **waiting for something important to finish** before moving on.
- Even though other things could be done in parallel, you **pause at that line** until the awaited task completes.

demo6.js

Example-1

```
demo6.js > ...
1  // This is synchronous code (runs immediately in order)
2  let a = 10;
3  let b = 20;
4  let res = a + b;
5  // This is an asynchronous function (runs in the background)
6  async function getdata() {
7      // Wait for the server response from the given API URL
8      let resserver = await fetch("https://jsonplaceholder.typicode.com/posts");
9      // Convert the response to JSON and wait for the parsing to finish
10     console.log(await resserver.json());
11 }
12 // Call the async function (fetch starts here)
13 getdata();
14 // This will run before the async fetch completes
15 console.log(res);
```

```
PROBLEMS  DEBUG CONSOLE  TERMINAL  PORTS                                Code
PS C:\Users\hp\OneDrive\Desktop\Javascript> node demo6.js
30
[
  {
    userId: 1,
    id: 1,
```

Note:

Key points:

- Without await, the code continues immediately (non-blocking).
- With await, the function **waits** for the Promise to finish before moving to the next line.

Example-2

server.js

```
async function getData() {
    let promise = new Promise((resolve) => {
        setTimeout(() => resolve("Data loaded!"), 2000);
    });
    let result = await promise; // Wait here until promise resolves
    console.log(result);
    console.log("Next task");
}

getData();
console.log("Hello Everyone !")
```

PROBLEMS DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\hp\OneDrive\Desktop\Javascript> node server.js
Hello Everyone !
Data loaded!
Next task
```

Example-3

server.js

```
function orderFood() {
    return new Promise((resolve) => {
        setTimeout(() => resolve("Food delivered!"), 3000);
    });// 3 seconds delay
}
// Async function to use await
async function eatFood() {
    console.log("Order placed, waiting for food...");
    let food = await orderFood(); // Wait until food is delivered
    console.log(food);              // Food delivered!
    console.log("Now eating the food");
    console.log("After eating, do other tasks");
}
eatFood();
console.log("Hello ")
```

PROBLEMS DEBUG CONSOLE TERM

```
PS C:\Users\hp\OneDrive\Desktop\Javascript> node server.js
Order placed, waiting for food...
Hello
Food delivered!
Now eating the food
After eating, do other tasks
```

Explanation:

- await orderFood() pauses the function until the food (Promise) is delivered.
- After that, the next steps (eating and other tasks) continue.
- This makes asynchronous code behave **like synchronous code** for readability.

Example-4: Washing clothes

```
function washClothes() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Clothes are clean!"), 1000);
  });}
async function laundryRoutine() {
  console.log("Start washing clothes...");
  let result = await washClothes();
  console.log(result);
  console.log("Now fold the clothes");
}laundryRoutine();
```

Example-5: Sending an email

```
function sendEmail() {
  return new Promise((resolve) => {
    setTimeout(() => resolve(" Email sent!"), 1500);
  });}
async function emailTask() {
  console.log("Composing email...");
  let result = await sendEmail();
  console.log(result);
  console.log("Check inbox for reply");
}emailTask();
```

Example-6: Booking a cab

```
function bookCab() {
  return new Promise((resolve) => {
    setTimeout(() => resolve(" Cab booked!"), 2000);
  });}
async function travelPlan() {
  console.log("Looking for a cab...");
  let result = await bookCab();
  console.log(result);
  console.log("Start traveling to destination");
} travelPlan();
```

Example-7: Downloading a file

```
function downloadFile() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("File downloaded successfully!"), 2500);
  });
}
async function downloadTask() {
  console.log("Downloading file...");
  let result = await downloadFile();
  console.log(result);
  console.log("Open the file now.");
}
downloadTask();
```

File Handling in node JS

❖ What is a File and a Folder?

- **File:** A file is a container in which data is stored. It can be text, image, video, or code.
  Example: data.txt, app.js
- **Folder (Directory):** A folder is a container that holds files and other folders.
  Example: Documents, Projects

❖ What is File Handling in Node.js?

➔ **File Handling** means **creating, reading, updating, and deleting files** using Node.js.

- Node.js provides the built-in **fs module** (File System) for this purpose.
- You can perform both **synchronous** (blocking) and **asynchronous** (non-blocking) operations.

n	Function	
•	Create File	fs.writeFile()
•	Read File	fs.readFile()
•	Update File	fs.appendFile()
•	Delete File	fs.unlink()
•	Create Folder	fs.mkdir()
•	Delete Folder	fs.rmdir()

1. How to Create a File

- **Syntax:** fs.writeFile("filename.txt", "File content here", callback);
- **Note:** If you want to create a file inside another folder
- **Syntax:** fs.writeFile("myFolder/filename.txt", "File content here", callback);

➢ Parameter	Mandatory / Optional	Description
• "filename.txt"	Mandatory	The name (and path) of the file to create or overwrite.
• "File content here"	Mandatory	The content to write into the file.
• Callback	Optional	A function to handle success or error after writing the file.

Note-1: "File content here" you can pass a variable instead of a string directly for the file content in fs.writeFile()

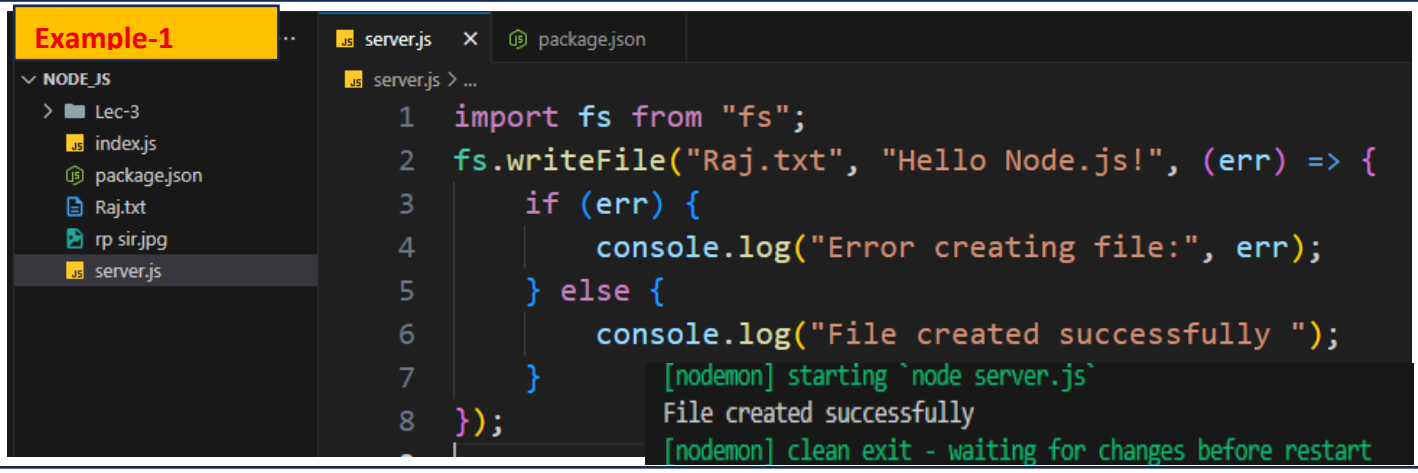
Note-2:

- Node.js does not restrict file types; you can create .txt, .js, .py, .html, .css, .json, etc.
- content inside the file should match the type for it to work correctly (JS code inside .js, HTML inside .html, etc.).

Step: -1: Importing the File System Module

Ex: import fs from "fs" or → const fs = require("fs");

Example-1



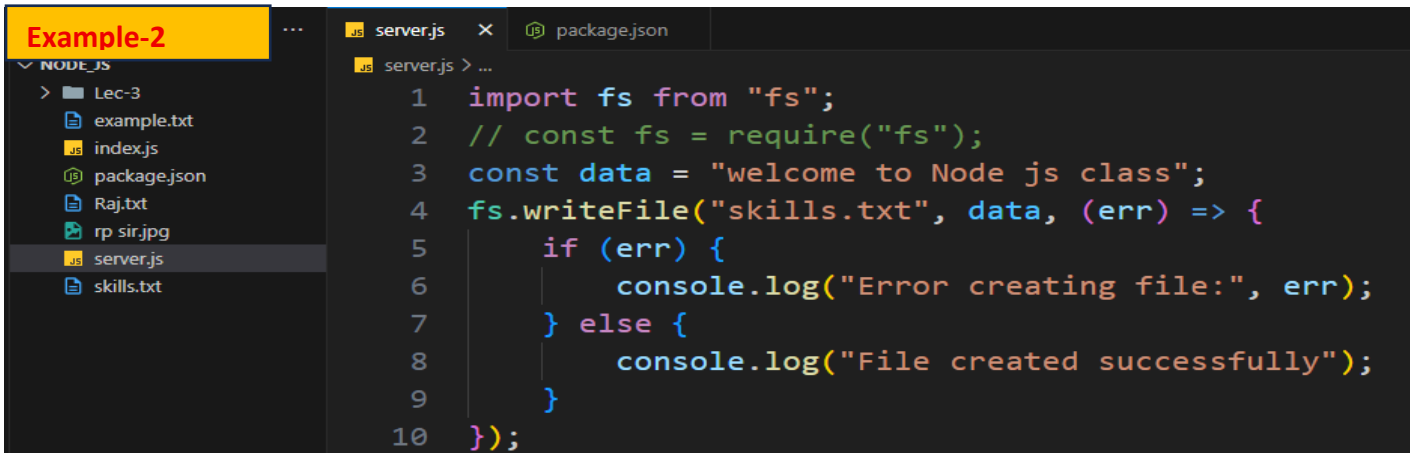
Notes on fs.writeFile (Node.js)

2. Creates a new file with given content.
3. If file exists → overwrites content.
4. If file doesn't exist → creates it.
5. Works asynchronously (non-blocking).
6. Can also create inside folders ("myFolder/file.txt").

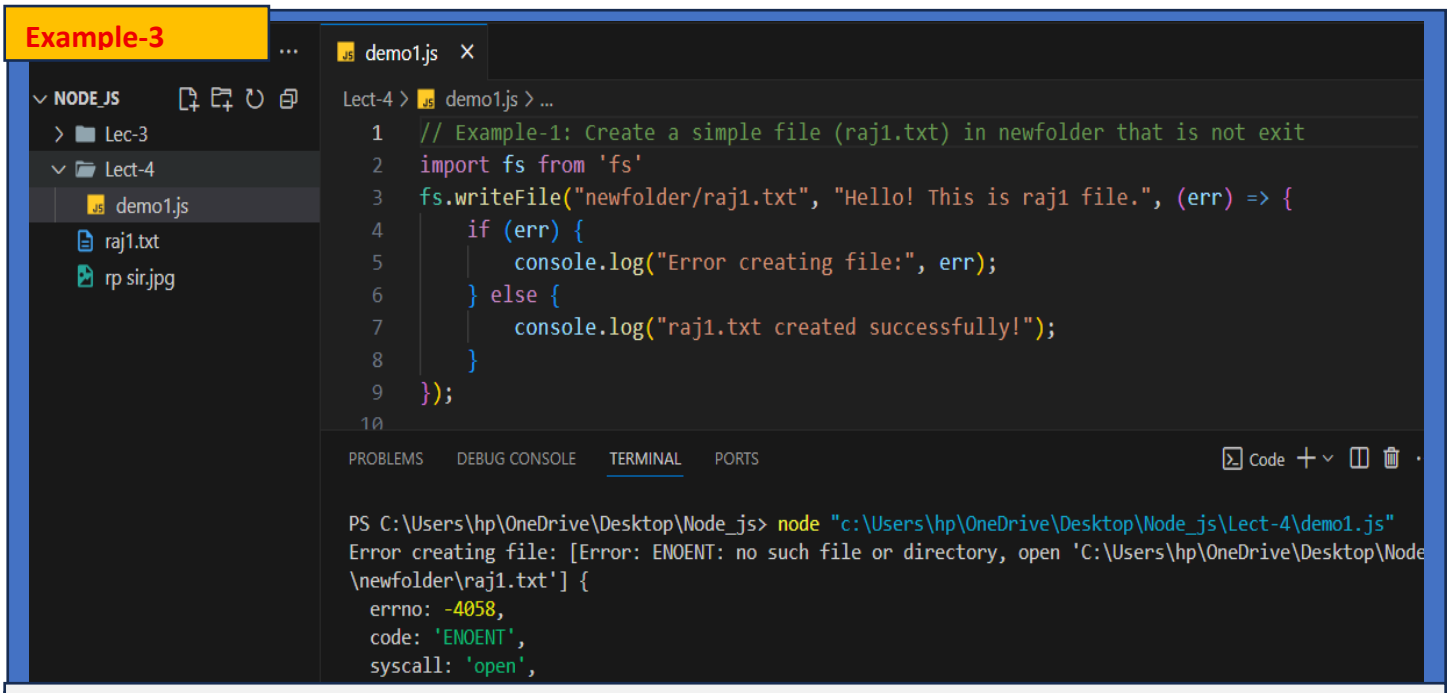
6.Can throw errors if:

- Folder doesn't exist
- Permission denied
- Invalid filename
- Disk full or read-only file system

Example-2

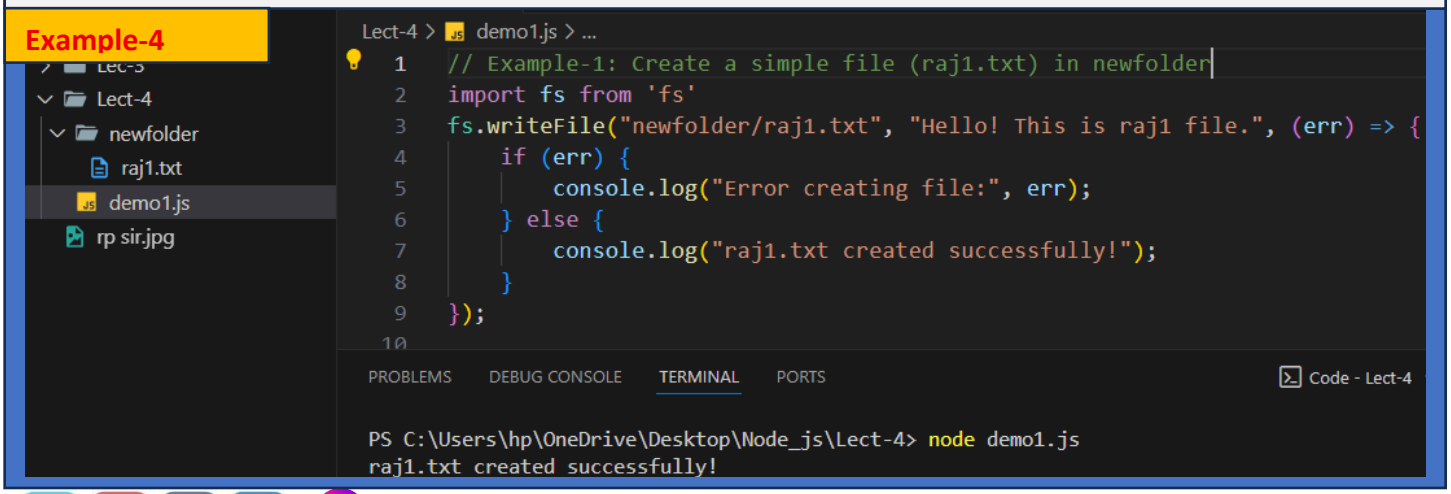


Example-3



Note: if the folder does not exit it will through the error

Example-4



2. How to Read a File

- **Syntax:** fs.readFile("filename.txt", "utf8", callback);
- **Example:**

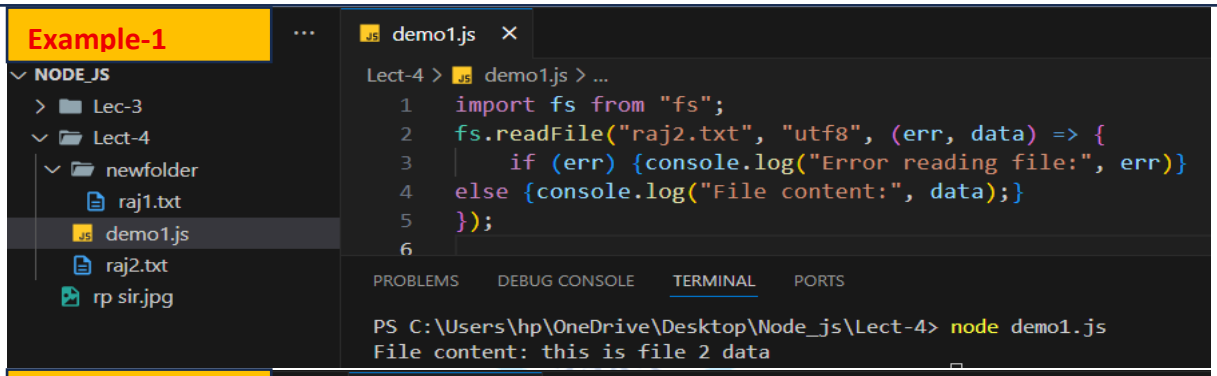
```
import fs from "fs";
fs.readFile("raj.txt", "utf8", (err, data) => {
  if (err) {console.log("Error reading file:", err)}
else {console.log("File content:", data);}
});
```

❖ Parameters:

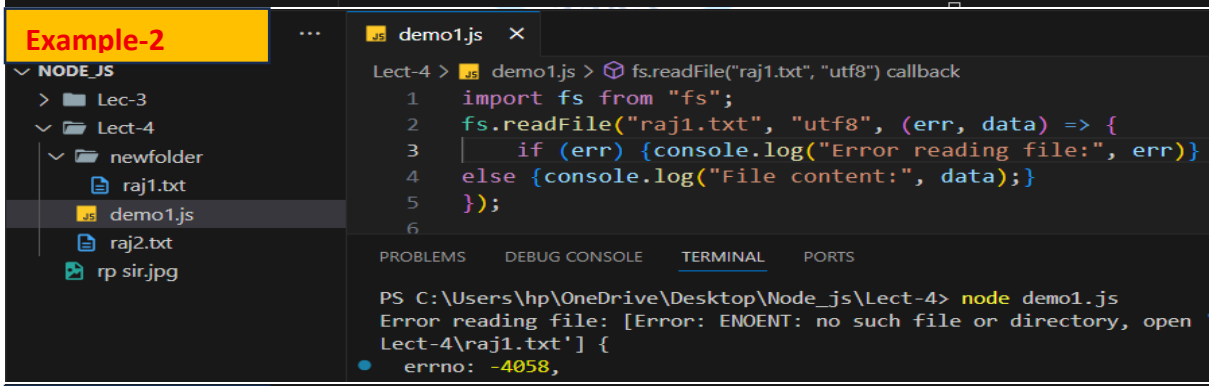
Parameter	Mandatory / Optional	Description
1."filename.txt"	Mandatory	The name (and path) of the file to read
2."utf8"	Optional	Encoding format (to read as text)
3.callback	Mandatory	Function to handle success or error after reading

- **Notes:**
  1. Can read any file type: .txt, .js, .py, .html, .json, etc.
  2. Use "utf8" for text files; omit for binary files.
  3. Errors occur if file doesn't exist, path is wrong, or permission is denied.
  4. Works asynchronously (non-blocking).
- **Step-1:** Import the File System module: **Note:** import fs from "fs"; // or const fs = require("fs");

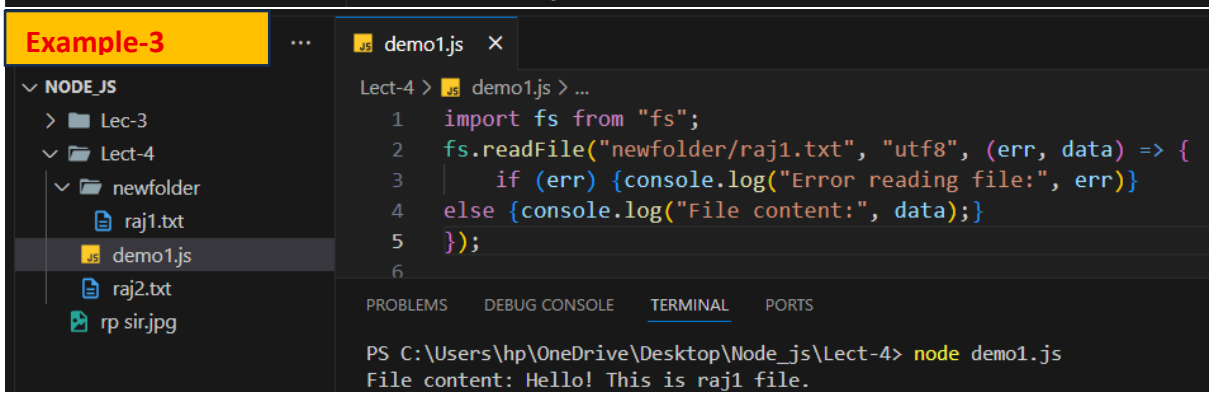
Example-1



Example-2



Example-3



3. How to Update a File

File("filename.txt", "Additional content", callback);

- **Example:**

```
import fs from "fs";
fs.appendFile("raj.txt", "\nThis is new content.", (err) => {
  if (err) console.log("Error updating file:", err);
  else console.log("File updated successfully");
});
```

- **Parameters:**

Parameter	Mandatory / Optional	Description
1. "filename.txt"	Mandatory	Name (and path) of the file to update
2. "Additional content"	Mandatory	Content to append to the file
3. callback	Mandatory	Function to handle success or error

Notes:

1. Can update any file type: .txt, .js, .py, .html, .json, etc.
2. Appends content at the **end of the file**.
3. Errors occur if file doesn't exist, path is wrong, or permission is denied.
4. Works asynchronously (non-blocking).

Note: You want to add content at the beginning or anywhere in a file instead of just appending at the end. Here's a short note on that: **Adding Content at Beginning or Anywhere in a File**

- fs.appendFile() always adds content at the end.
- To add content at the beginning or anywhere, you need to:
  1. Read the file first using fs.readFile().
  2. Modify the content (add text at the start, middle, or specific position).
  3. Write the updated content back using fs.writeFile().
- **Example – Add at the beginning:**

```
import fs from "fs";
fs.readFile("raj.txt", "utf8", (err, data) => {
  if (err) return console.log("Error reading file:", err);
  const updatedContent = "New content at beginning\n" + data;
  fs.writeFile("raj.txt", updatedContent, (err) => {
    if (err) console.log("Error updating file:", err);
    else console.log("Content added at the beginning successfully!");
  });});
```

- **Example – Add anywhere (after line 2):**

```
fs.readFile("raj.txt", "utf8", (err, data) => {
if (err) return console.log("Error reading file:", err);
let lines = data.split("\n");
lines.splice(2, 0, "New content in the middle"); // insert at 3rd line
const updatedContent = lines.join("\n");
fs.writeFile("raj.txt", updatedContent, (err) => {
  if (err) console.log("Error updating file:", err);
  else console.log("Content added in the middle successfully!");
});});
```

Notes:

1. Can update any file type: .txt, .js, .py, .html, .json, etc.
2. Use fs.readFile() → modify → fs.writeFile() to add content anywhere.
3. Works asynchronously (non-blocking).
4. Errors occur if file doesn't exist, path is wrong, or permission denied.

4. How to Create a Folder

- **Syntax:** fs.mkdir("folderName", callback);
- **Example:**

```
import fs from "fs";
fs.mkdir("myFolder", (err) => {
  if (err) console.log("Error creating folder:", err);
  else console.log("Folder created successfully");
});
```

Parameters:

Parameter	Mandatory / Optional	Description
1. "folderName"	Mandatory	Name (and path) of the folder to create
2. callback	Mandatory	Function to handle success or error

Notes:

1. Can create single or nested folders using { recursive: true }.
2. Errors occur if folder already exists (without recursive) or permission is denied.
3. Works asynchronously (non-blocking).

5. How to Delete a File and a Folder

1. Delete a File Syntax: fs.unlink("filename.txt", callback);

Example:

```
import fs from "fs";
fs.unlink("raj.txt", (err) => {
  if (err) console.log("Error deleting file:", err);
  else console.log("File deleted successfully"); });
```

Parameters:

Parameter	Mandatory / Optional	Description
1. "filename.txt"	Mandatory	Name (and path) of the file to delete
2. callback	Mandatory	Function to handle success or error

Notes:

1. Deletes a file permanently.
2. Errors occur if file doesn't exist or permission is denied.
3. Works asynchronously (non-blocking).

2. Delete a Folder

- **Syntax:** fs.rmdir("folderName", callback);
- **Example:**

```
import fs from "fs";
fs.rmdir("myFolder", (err) => {
  if (err) console.log("Error deleting folder:", err);
  else console.log("Folder deleted successfully"); });
```

Parameters:

Parameter	Mandatory / Optional	Description
1. "folderName"	Mandatory	Name (and path) of the folder to delete
2. callback	Mandatory	Function to handle success or error

Notes:

1. Deletes an **empty folder** only.
2. Errors occur if folder is not empty, doesn't exist, or permission is denied.
3. Works asynchronously (non-blocking).

Path Module in Node js

Path Module in Node.js

- The path module is a built-in module in Node.js.
- It is used to work with file and directory paths.
- It helps in joining, resolving, and normalizing paths so your code works across all operating systems.
- Path module makes it easy to handle file & folder paths.

path module methods in Node.js

- **basename()** → file name : Use when you need only the **file name** from a complete path.
- **dirname()** → folder name : Use when you need only the **directory/folder path**.
- **extname()** → file extension : Use when you need the **file extension** (like .js, .html, .css, .pdf.jpg).
- **join()** → combine paths :use when Safely combine multiple folders/files into one full pat
- **resolve()** → absolute path : Converts a relative path into an **absolute path**. Very useful in projects.

Example-1

Lect-5 > demo2.js > ...

```
1  import path from "path";
2  // Join paths
3  const fullPath = path.join("/path", "home.html", "raj.css");
4  console.log("Full Path =", fullPath);
5  // How to find the actual or absolute path
6  const currentPath = path.resolve();
7  console.log("Current Path =", currentPath);
8  // Get file extension
9  const fileExt = path.extname("raj.css");
10 console.log("File Extension =", fileExt);
11 // Get base name (file name)
12 const fileName = path.basename("home.html");
13 console.log("File Name =", fileName);
```

PROBLEMS DEBUG CONSOLE TERMINAL PORTS

```
Full Path = \path\home.html\raj.css
Current Path = C:\Users\hp\OneDrive\Desktop\Node_js
File Extension = .css
File Name = home.html
```

Getting the actual folder of the file (ES Modules)

- In **ES Modules (import)**, __filename and __dirname are not available.
- Use fileURLToPath(import.meta.url) to get the **full file path**.
- Then use path.dirname() to get the **folder containing the file**.

Example-2

NODE_JS
- Lec-3
- Lect-4
- Lect-5
  - demo1.js
  - demo2.js
- rp sir.jpg

demo2.js

Lect-5 > demo2.js > ...

```
1  //How to get the actual folder of the file
2  import path from "path";
3  import { fileURLToPath } from "url";
4  const filename = fileURLToPath(import.meta.url)
5  const foldername = path.dirname(filename);
6  console.log("File Path =", filename);
7  console.log("Folder Path =", foldername);
8
```

PROBLEMS DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\hp\OneDrive\Desktop\Node_js\Lect-5> node demo2.js
File Path = C:\Users\hp\OneDrive\Desktop\Node_js\Lect-5\demo2.js
Folder Path = C:\Users\hp\OneDrive\Desktop\Node_js\Lect-5
```

Example-3

Lect-5 > demo3.js > ...

```
1  import path from "path";
2  const v = "myimage.jpg";
3  // Get the file extension
4  const getFileType = path.extname(v);
5  console.log("File Extension =", getFileType);
6  // Check if the file is a .jpg
7  if (getFileType === '.jpg') {
8      console.log("Your photo uploaded successfully");
9  } else {
10     console.log("Please upload a file with .jpg extension");
11 }
```

PROBLEMS DEBUG CONSOLE TERMINAL PORTS Code - Lect-5

```
PS C:\Users\hp\OneDrive\Desktop\Node_js\Lect-5> node demo3.js
File Extension = .jpg
Your photo uploaded successfully
```

Example-4

Lect-5 > demo3.js > ...

```
1  import path from "path";
2  // 1. extname() → Check file type
3  const uploadedFile = "marksheet.pdf";
4  const fileExt = path.extname(uploadedFile);
5  console.log("File Extension =", fileExt);
6  if (fileExt === ".pdf") {
7  console.log("File uploaded successfully\n");
8  } else {console.log("Please upload a .jpg file\n");}
9  // 2. join() → Combine folder and file paths
10 const folder = "/user";
11 const subFolder = "images";
12 const file = "profile.jpg";
13 const fullPath = path.join(folder, subFolder, file);
14 console.log("Joined Full Path =", fullPath);
15 // 3. resolve() → Get absolute path
16 const absPath = path.resolve(file);
17 console.log("Absolute Path =", absPath);
```

How to Build Server in Node JS

❖ Server:

- server is a **dedicated computer or a computer program** that **accepts requests from clients** (like browsers) and **sends back responses** over a network
- It is a program or computer that **listens for requests** from clients (like web browsers) and **sends back responses**.

❖ Real Life Example of server:

- A **restaurant waiter** acts like a server: the customer (client) places an order (request), and the waiter (server) brings back the food (response).
- Similarly, a **web server** receives requests from your browser and sends back web pages or data.

❖ **Examples of servers:** Web server (sends HTML, CSS, JS files) API server (sends JSON data)

❖ **Note:** It can be **hardware** (a powerful computer) or **software** (a program running on a computer). Its main job is to **listen for requests and respond** with data, files, or services.

❖ Step to create the Server in node JS:

- Node.js makes it easy to build a simple server using its built-in http module.

Step 1: create file with server.js(recommended)

Step 2: run npm init -y (for creating the json file "project setting")

- npm init -y automatically creates a **package.json** with default values.
- This file stores **project information and dependencies**.

Step3: Import the http module

- **Ex:** import **http** from "http";
  - ✓ **http** is built into Node.js, no need to install anything.
  - ✓ It allows us to create a web server.

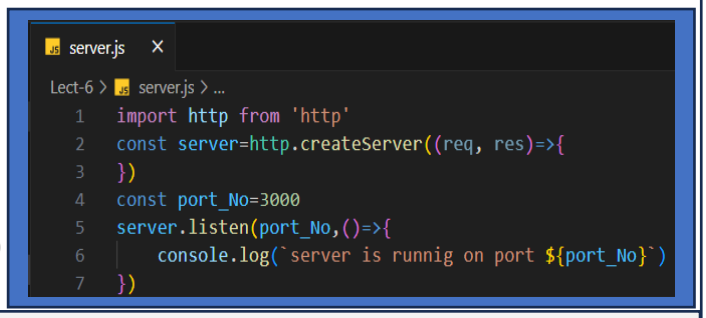
web is a system of interconnected **websites and web pages** that can be accessed over Internet using a browser.

web server: - it is a computer or software that **delivers web pages and content** to clients (browsers) over the Internet.

Step-4: Create a server

Example-1:

```
import http from 'http'
const server=http.createServer((req, res)=>{
})
const port_No=3000
server.listen(port_No,()=>{
   console.log(`server is runnig on port ${port_No}`)})
```



➢ **const server = http.createServer((req, res) => {})**

- Creates a **new server** using http.createServer().
- Takes a **callback function** with two parameters:
  - ✓ req → represents the **client request** (e.g., browser asking for a page).
  - ✓ res → represents the **response** that the server will send back.

➢ **const port_No = 3000**

- ✓ Defines the port number where the server will listen.
- ✓ 3000 is commonly used for local development.

➢ **server.listen(port_No, () => {console.log(`server is runnig on port ${port_No}`) })**

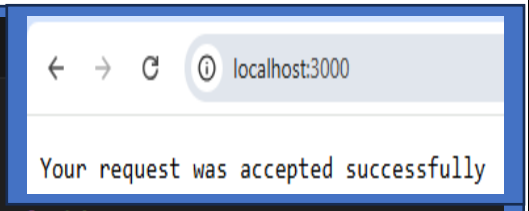
- ✓ Starts the server and makes it listen on the specified port.
- ✓ The callback function runs once the server is successfully running.
- ✓ console.log() prints a message confirming the server is running.

server.js Example-2

Lect-6 > server.js > ...

```
1  // Example for creating the server in Node js
2  import http from 'http';
3  const server = http.createServer((req, res) => { // Create a server
4      // Send response to the client
5      res.writeHead(200, { "Content-Type": "text/plain" });
6      res.end("Your request was accepted successfully");
7  }); // Define the port number
8  const port_No = 3000; // Start the server and listen on the specified port
9  server.listen(port_No, () => {
10     console.log(`Server is running on http://localhost:${port_No}`);
11 });
```

Optional



➢ **res.writeHead(200, { "Content-Type": "text/plain" });**

it is used to set the HTTP response headers before sending the actual content. Let me break it down:

1. res.writeHead(200, ...)

200 → HTTP status code 200 means the request was successful.

Other examples: 404 = Not Found, 500 = Server Error.

2. { "Content-Type": "text/plain" }

This is a header object that tells the browser what kind of content the server is sending.

"text/plain" → the response is plain text.

Other examples: "text/html" → HTML content "application/json" → JSON data

Common HTTP status codes you can use in Node.js:

1. **Successful Responses:**
   200 → OK (request successful),
   201 → Created (resource successfully created)
2. **Client Errors:** **400 →** Bad Request (invalid request from client) ,
   401 → Unauthorized (authentication needed)
   403 → Forbidden (access denied) **404 →** Not Found (resource not found)
3. **Server Errors** **500→** Internal Server Error (something went wrong on server)
   502→ Bad Gateway (server received invalid response)
   503 → Service Unavailable (server temporarily unavailable)

Note: In short: 2xx → Success, 4xx → Client errors, 5xx → Server errors

❖ **What** res.end() **does**

- res.end() **sends the response to the client** and **closes the connection**.
- Whatever you pass inside res.end() is the **actual content** that the browser (or client) will receive.

2. Relation to res.writeHead()

- res.writeHead(200, { "Content-Type": "text/plain" }) → sets the **status code** and **headers** (metadata about the response).
- res.end("...") → sends the **body of the response** (the actual message or data).

res.end() can send:

- **Text** → "Hello World"
- **HTML** → "<h1>Hi</h1>"
- **JSON** → JSON.stringify({...})
- **Binary data** → file buffers like images, PDFs, etc.

res.end() method in Node.js can send different types of data as the response.

1. Plain Text Example:

res.writeHead(200, { "Content-Type": "text/plain" });
res.end("Hello, this is plain text!");

- Content type: text/plain
- Browser displays it as simple text.

```
import http from 'http';
const server = http.createServer((req, res) => {
    res.writeHead(200, { "Content-Type": "text/html" });
    res.end(`Hello this is node js class `);
});
const port_No = 3000;
server.listen(port_No, () => {
    console.log(`Server is running on http://localhost:${port_No}`);
});
```

2. HTML Example:

res.writeHead(200, { "Content-Type": "text/html" });
res.end("<h1>Hello, this is HTML content!</h1>");

- Content type: text/html
- Browser renders it as a formatted webpage.

```
import http from 'http';
const server = http.createServer((req, res) => {
    res.writeHead(200, { "Content-Type": "text/html" });
    res.end(`<h1>Hello this is HTML Content </h1> `);
});
const port_No = 3000;
server.listen(port_No, () => {
    console.log(`Server is running on http://localhost:${port_No}`)
});
```

3. JSON Example:

res.writeHead(200, { "Content-Type": "application/json" });
res.end(JSON.stringify({ name: "Sangam", course: "Node.js" }));

- Content type: application/json
- Browser or client receives it as JSON data.

Optional

```
import http from 'http';
const server = http.createServer((req, res) => {
    res.writeHead(200, { "Content-Type": "text/html" });
    res.end(JSON.stringify({ name: "Sangam", course: "Node.js" }));
    // res.end(`<h1>Name: Sangam</h1><p>Course: Node.js</p>`);
});
const port_No = 3000;
server.listen(port_No, () => {
    console.log(`Server is running on http://localhost:${port_No}`);
});
```

4. Binary / Files Example

```
res.writeHead(200, { "Content-Type": "image/png" });
// Here you would send binary data like image or file buffer
res.end(fileBuffer);
```

- Content type depends on the file (image/png, application/pdf, etc.)
- Used to send **images, PDFs, videos, or any file**.

```
//common types, to use in res.writeHead():
// For Images
res.writeHead(200, { "Content-Type": "image/jpeg" });  // .jpg
res.writeHead(200, { "Content-Type": "image/png" });   // .png
res.writeHead(200, { "Content-Type": "image/gif" });   // .gif
//For PDF
res.writeHead(200, { "Content-Type": "application/pdf" });
// HTML
res.writeHead(200, { "Content-Type": "text/html" });
//For JSON
res.writeHead(200, { "Content-Type": "application/json" });
//For Video
res.writeHead(200, { "Content-Type": "video/mp4" });   // .mp4
res.writeHead(200, { "Content-Type": "video/webm" });  // .webm
res.writeHead(200, { "Content-Type": "video/ogg" });   // .ogg
//For Audio
res.writeHead(200, { "Content-Type": "audio/mpeg" });  // .mp3
res.writeHead(200, { "Content-Type": "audio/wav" });   // .wav
//Note: Always set the correct Content-Type before res.end() when sending the file.
```

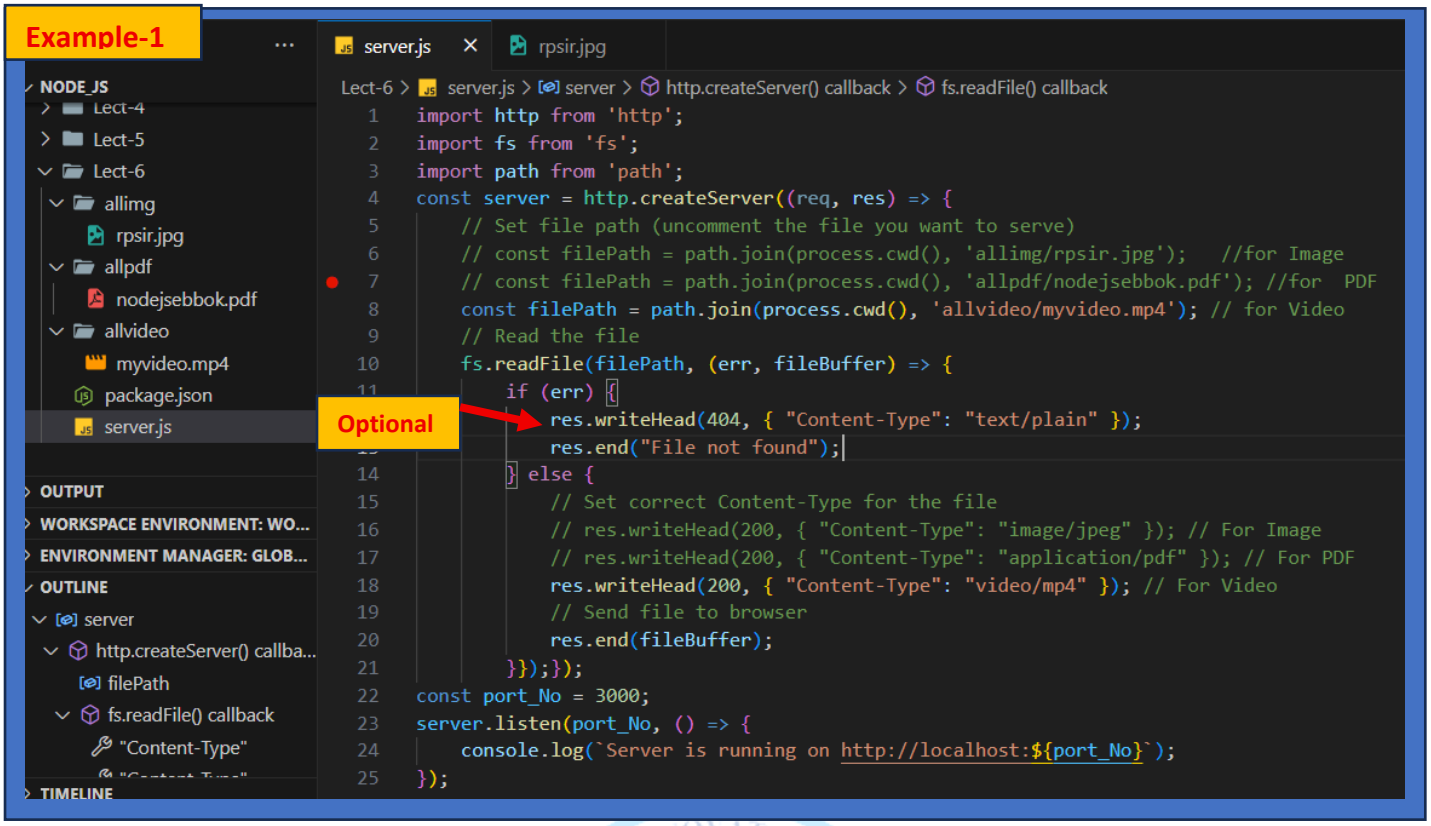
Types for send the data

- For **images** → "image/jpeg" / "image/png"
- For **PDFs** → "application/pdf"
- For **HTML** → "text/html"
- For **JSON** → "application/json"

video content types:

- .mp4 → "video/mp4"
- .webm → "video/webm"
- .ogg → "video/ogg"

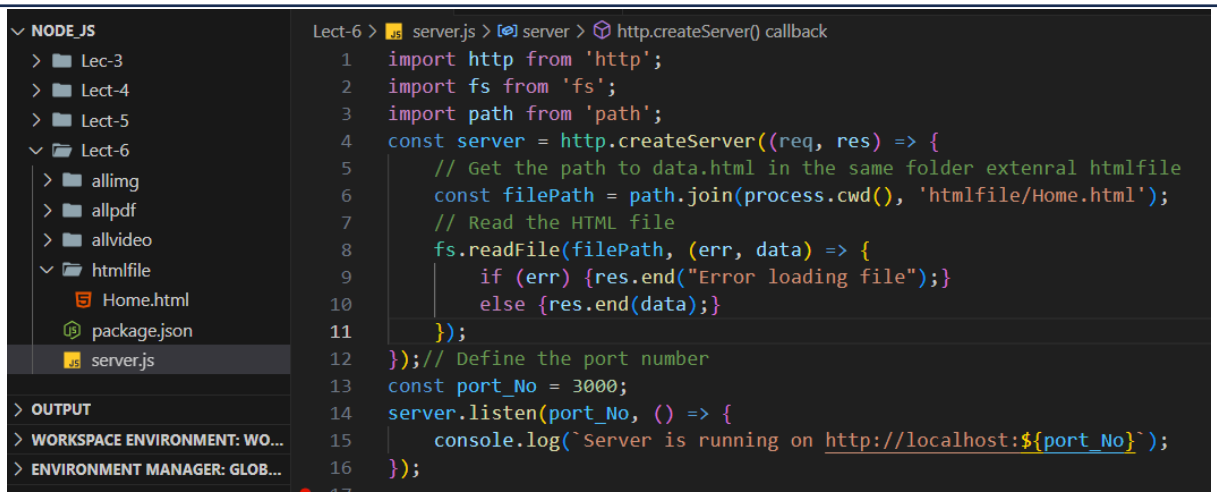
Example-1

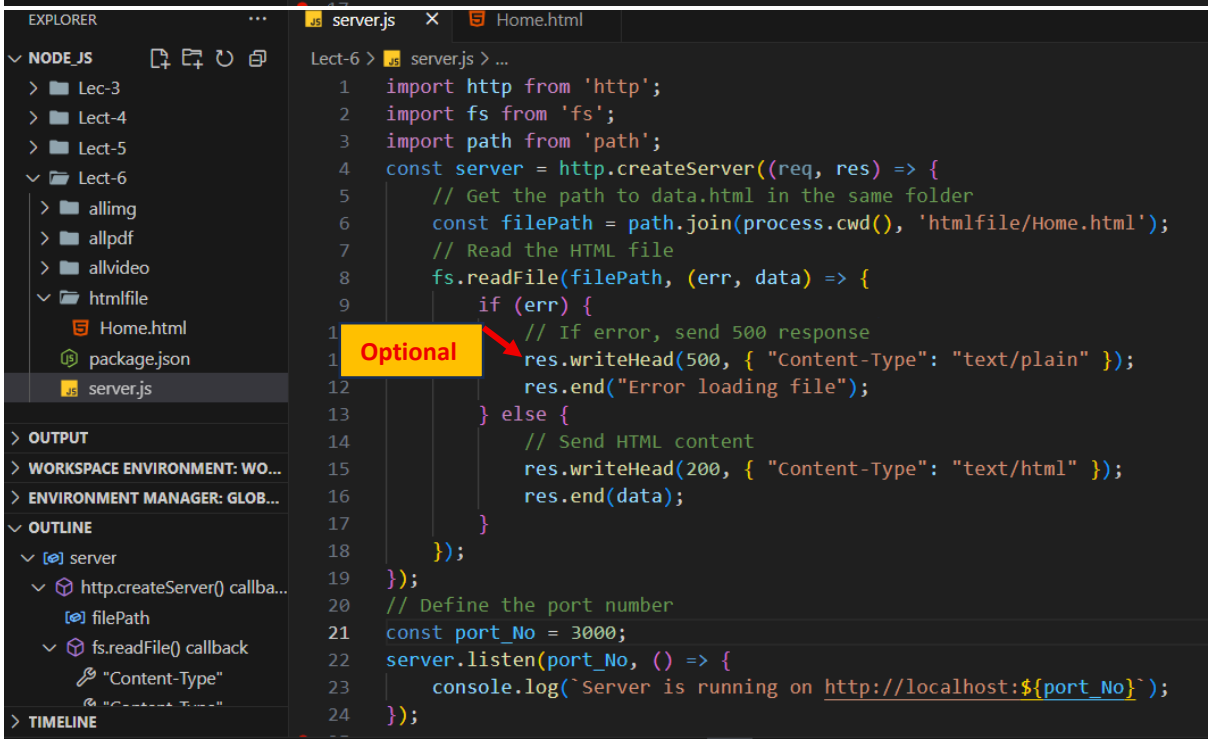


Example-2

```
server.js
Lect-6 > server.js > server > http.createServer() callback
import http from 'http';
const server = http.createServer((req, res) => { // Create a server
    // Set status code and content type to HTML
    res.writeHead(200, { "Content-Type": "text/html" });
    // Send HTML response
    res.end(`
        <html>
            <head>
                <title>My First Node Server</title>
            </head>
            <body>
                <h1>Your request was accepted successfully</h1>
                <p>Welcome to my Node.js server!</p>
            </body>
        </html>
    `);
});
// Define the port number
const port_No = 3000;
// Start the server
server.listen(port_No, () => {
    console.log(`Server is running on http://localhost:${port_No}`);
});
```

How to send external html as response in node using res.end()





Route in Node JS

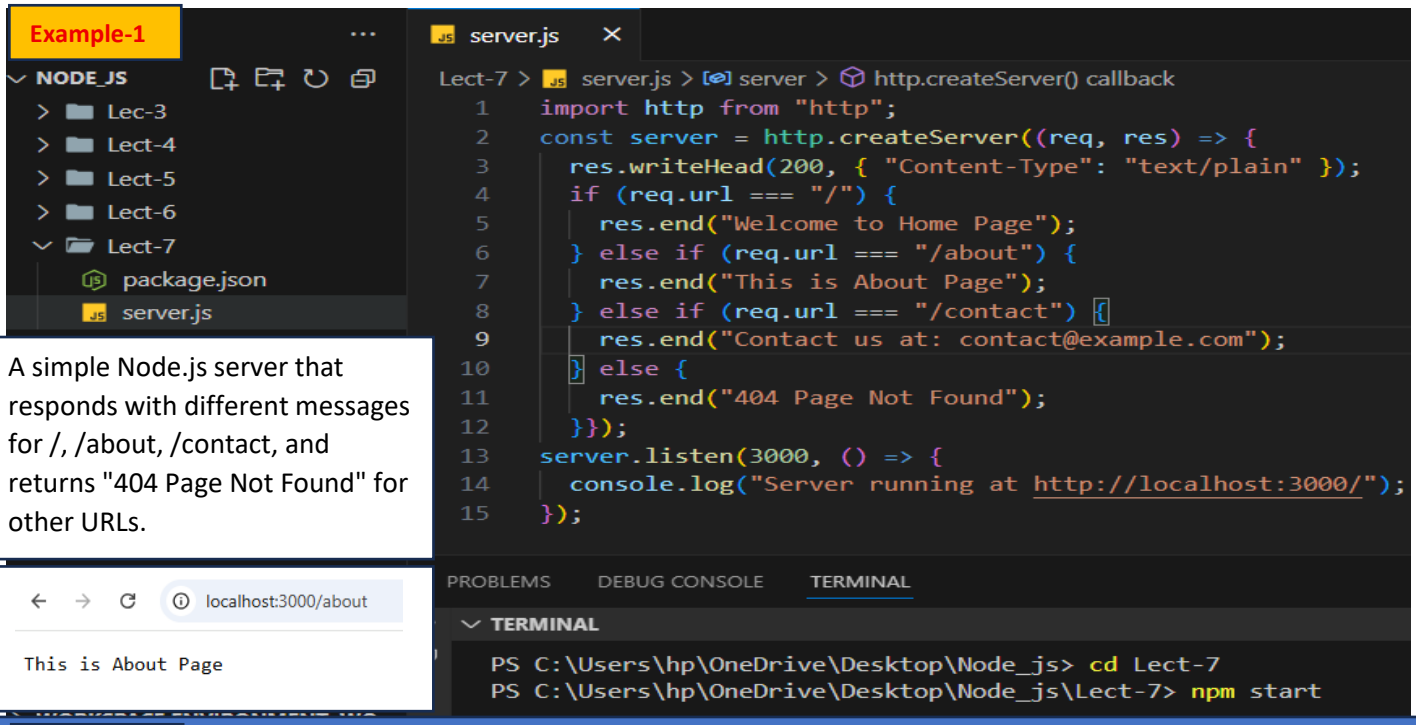
A route is a rule: (HTTP method + URL path) → handler function → response.

A Route is simply a way to tell your server: **"When someone visits a particular URL, show them the correct response.**

1. **HTTP Method →** Action type (like asking, sending, updating, deleting).
2. **URL Path →** The address (like /, /about, /contact).
3. **Handler Function →** The code that runs when that address is visited.
4. **Response →** The answer your server sends back to the client.

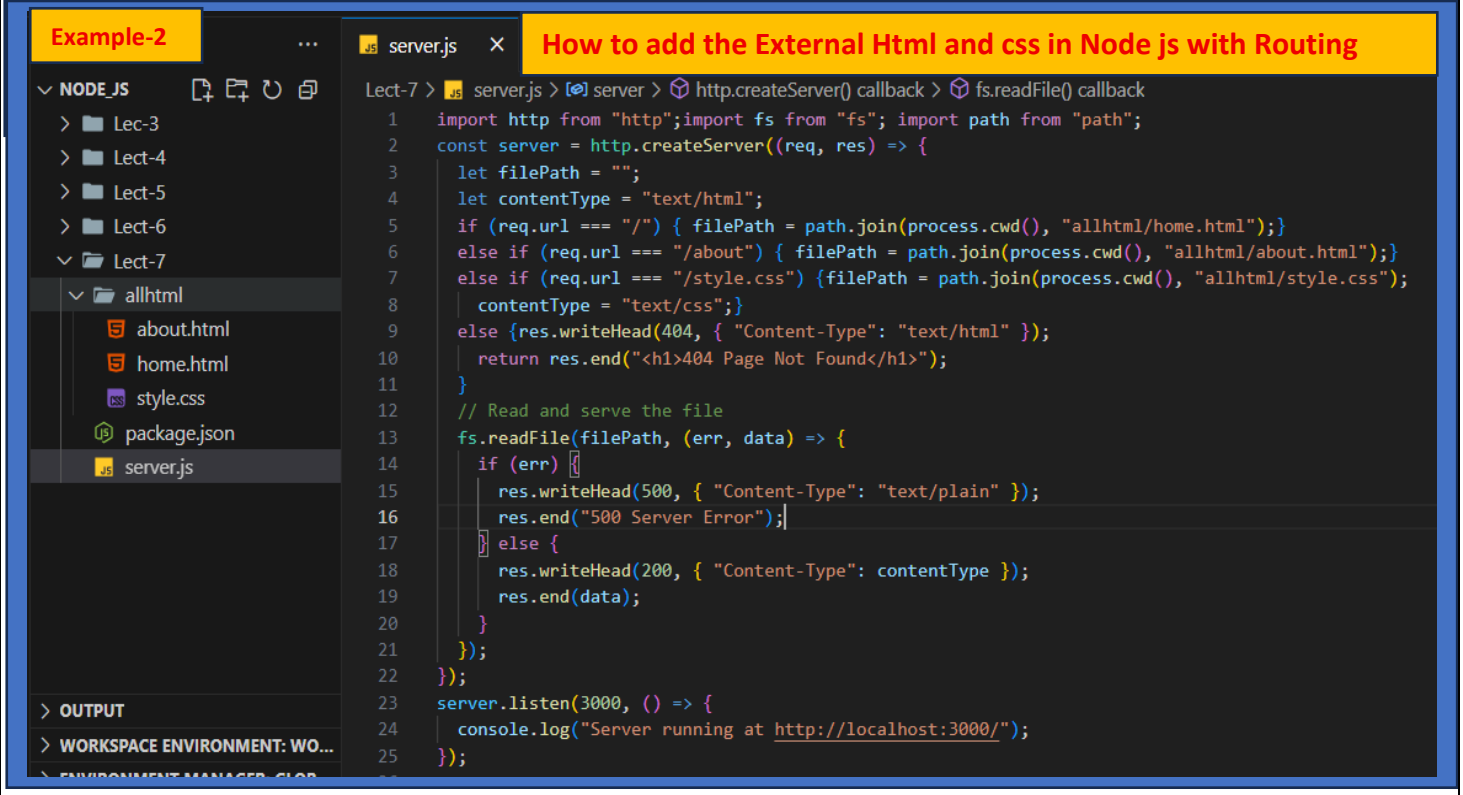
- You **match** the request, **extract** inputs, **run** logic, and **respond**.
- Use **Express** to define routes declaratively; use **http** for learning or very simple servers.

Example-1



A simple Node.js server that responds with different messages for /, /about, /contact, and returns "404 Page Not Found" for other URLs.

Example-2 **How to add the External Html and css in Node js with Routing**



Express JS

What is Express.js?

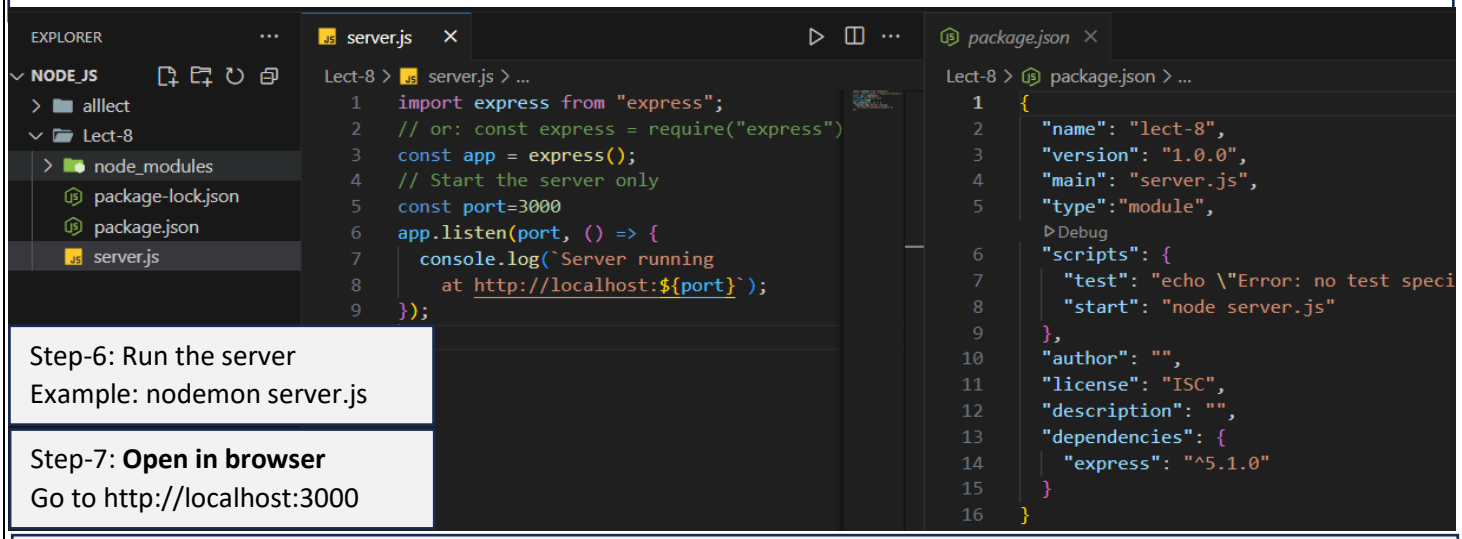
- **Express.js** is a lightweight and flexible **web application framework** for Node.js.
- It simplifies handling **routes, requests, responses, and middleware** compared to writing everything with Node's built-in http module.

Why are we using it?

servers and APIs quickly.
routing system (GET, POST, PUT, DELETE).
lleware (e.g., authentication, logging).
r REST APIs, websites, and backend services.

How to Use Express.js in Node.js

Step-1. Make a new folder — **Example:** mkdir myapp then cd myapp
Step-2: create the file with the .js extension — **Example:** server.js or index.js
Step-3: Initialize project — **Example:** npm init -y
Step-4: Install Express — **Example:** npm install express or npm i express
Step-5: Create the server



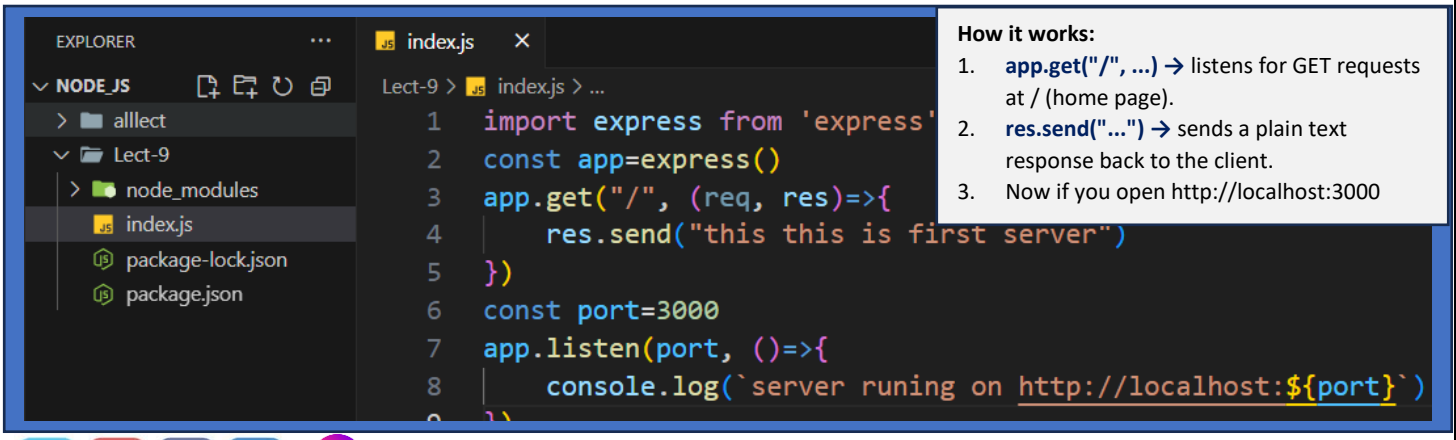
Step-6: Run the server
Example: nodemon server.js

Step-7: **Open in browser**
Go to http://localhost:3000

How to create server and send response on browser in express JS

- **get(path, callback)** → Handles GET requests for given path.
- **req** → Request object (information from client).
- **res** → Response object (used to send data back).
- **res.send()** → Sends a simple text/HTML response.

- **res.send()** automatically sets the correct Content-Type.
- For JSON specifically, you can also use res.json() (better practice for APIs).
- **send()** can handle **strings, HTML, arrays, objects, or buffers**. like files, images, video or any binary content

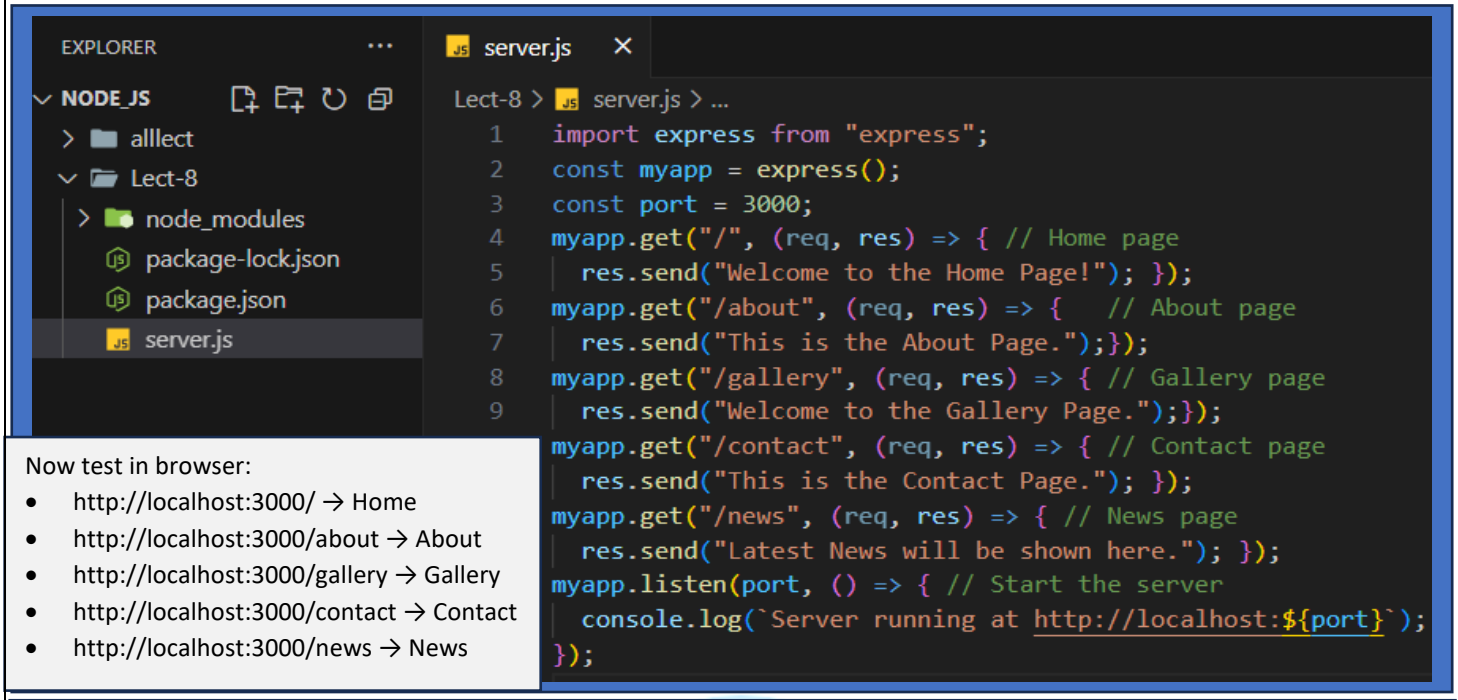


How it works:

1. **app.get("/", ...)** → listens for GET requests at / (home page).
2. **res.send("...")** → sends a plain text response back to the client.
3. Now if you open http://localhost:3000

How to send multiple response on browser

Need to define multiple routes for different pages (/about, /gallery, /contact, /news).



Now test in browser:

- http://localhost:3000/ → Home
- http://localhost:3000/about → About
- http://localhost:3000/gallery → Gallery
- http://localhost:3000/contact → Contact
- http://localhost:3000/news → News

How to send image, video & video response on browser

Folder Structure

```
myapp/                    <-- Project root
|
├── node_modules/         <-- Installed npm packages (auto-created)
├── package.json          <-- NPM config file
|
├── server.js             <-- Main Express server file
|
├── public/               <-- Static files (CSS, JS, images)
|   ├── css/              <-- CSS files
|   |   └── style.css
|   ├── js/               <-- JavaScript files
|   |   └── script.js
|   └── images/           <-- Image files
|       └── rpsir.jpg
|
└── htmlfile/             <-- External HTML files
    └── index.html
```

How it works

1. **server.js** → Creates the Express server and serves routes.
2. **public/** → Use express.static() to serve CSS, JS, and images.
3. **htmlfile/** → Use res.sendFile() to send HTML files to the browser

How to send External multiple response on browser

This all methods we used in Node.js + Express.js for sending external HTML, CSS, images, PDFs, and videos

Step-11. import express from "express"

- Imports the Express framework. Allows you to create an Express server and routes.

Step-2. import path from "path"

- Node.js module to handle file and folder paths. Helps create absolute paths with path.join().

Step 3. import { fileURLToPath } from "url"

- Converts ES Module import.meta.url to a normal file path. Needed because __dirname is not available in ES Modules.

Step-4. const __filename = fileURLToPath(import.meta.url) const any_v_name= fileURLToPath(import.meta.url)

- Gets the absolute path of the current file. **absolute path** means **full path from the root of the file system to a specific file or folder**, showing its exact location.

Step-5. const __dirname = path.dirname(__filename) or const anyvaraible= path.dirname(__filename)

- Gets the directory of the current file. Useful for building paths to HTML, CSS, or other files.

Step-6. express(): This is used to create an Express application (server). You usually store it in a variable.
Example: const app = express();

Step-7. app.use() Middleware to process requests.

Node.js and Express.js, a middleware is a function that runs during the request-response cycle.

- **Purpose: It can process requests, modify request/response, or serve files before sending a response.**
- **Why we use it: To handle tasks like serving static files, parsing data, authentication, logging, etc.**
- **Example:** app.use('/css', express.static(path.join(__dirname, 'public/css')));

Note: express.static() serves static files like CSS, JS, images, PDFs, videos.

Example: app.use('/images', express.static(path.join(__dirname, 'allimg')));

8. path.join() :- Joins multiple path segments into a single platform-independent path.

- **Example:** path.join(__dirname, 'htmlfile', 'index.html')

9. **res.sendFile()** :- Sends an external HTML or other file to the client.

- **Example:** res.sendFile(path.join(__dirname, 'htmlfile', 'index.html'));

10. **res.send():-** Sends a simple text, HTML string, JSON, array, or buffer to the client.

- **Example:** res.send("This is Home Page");

11. app.get(path, callback)

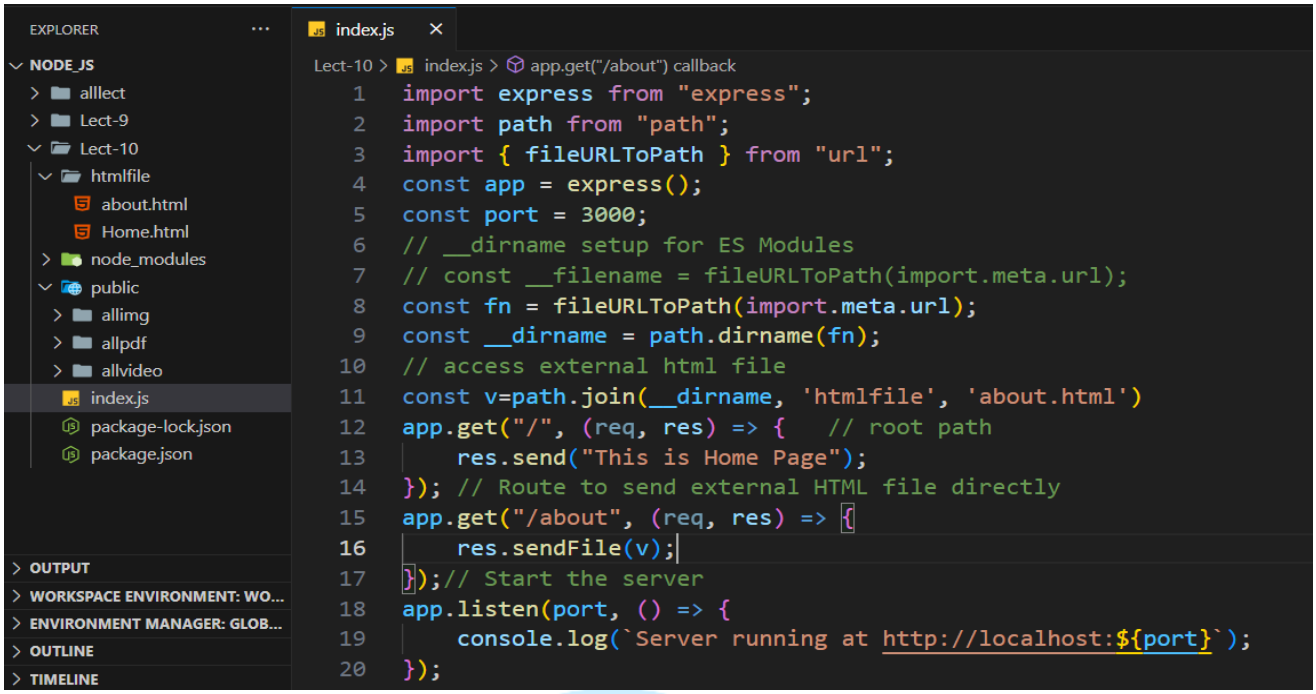
- Creates a GET route.
- path → URL path (e.g., / or /about)
- callback(req, res) → Function executed when route is accessed.

12. app.listen(port, callback)

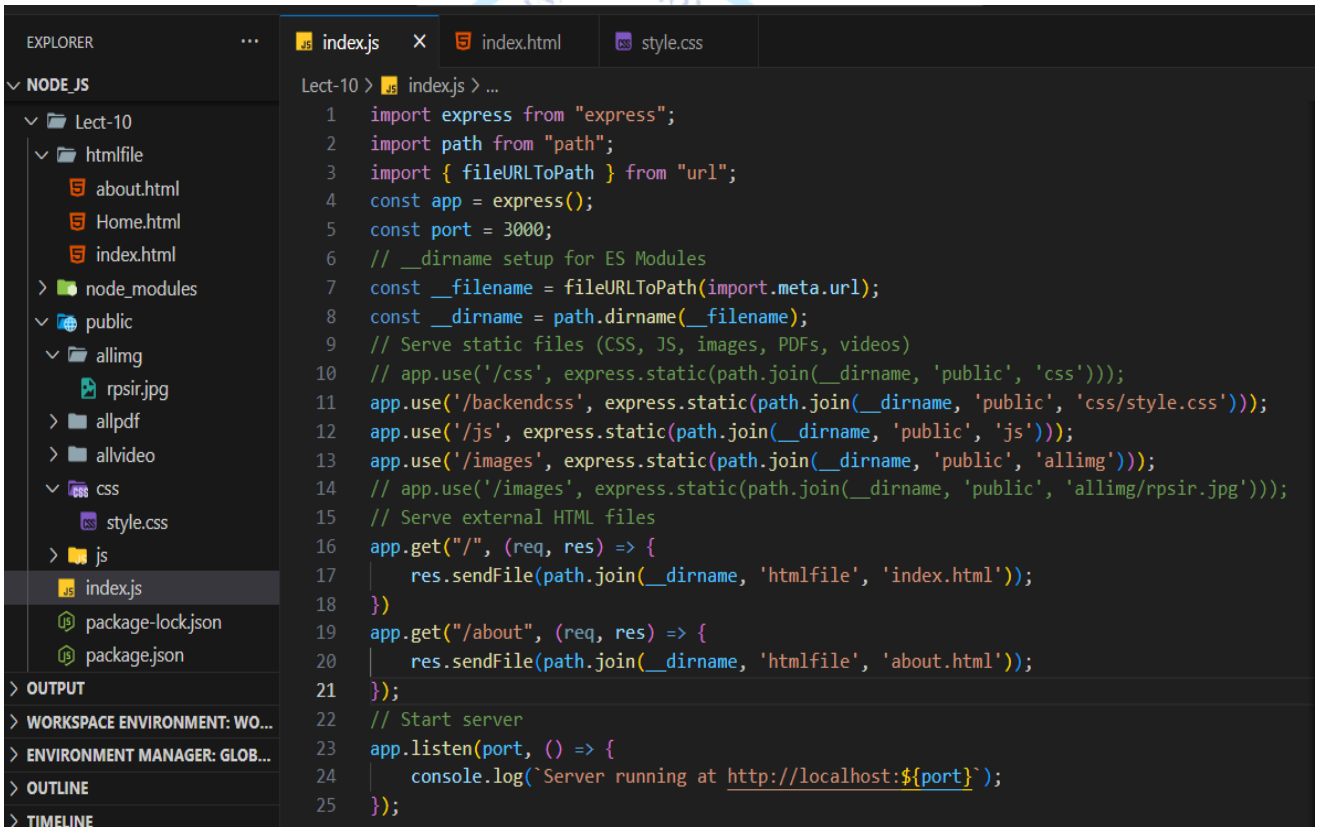
- Starts the Express server and listens on the given port.
- **Example**: app.listen(3000, () => console.log("Server running"));

- **Middleware** = function that processes requests before final response.
- **Absolute path** = full path from root to file/folder.
- **Static files** = CSS, JS, images, PDFs, videos served with express.static().
- Use res.sendFile() for **HTML files**, res.send() for **text/JSON**.

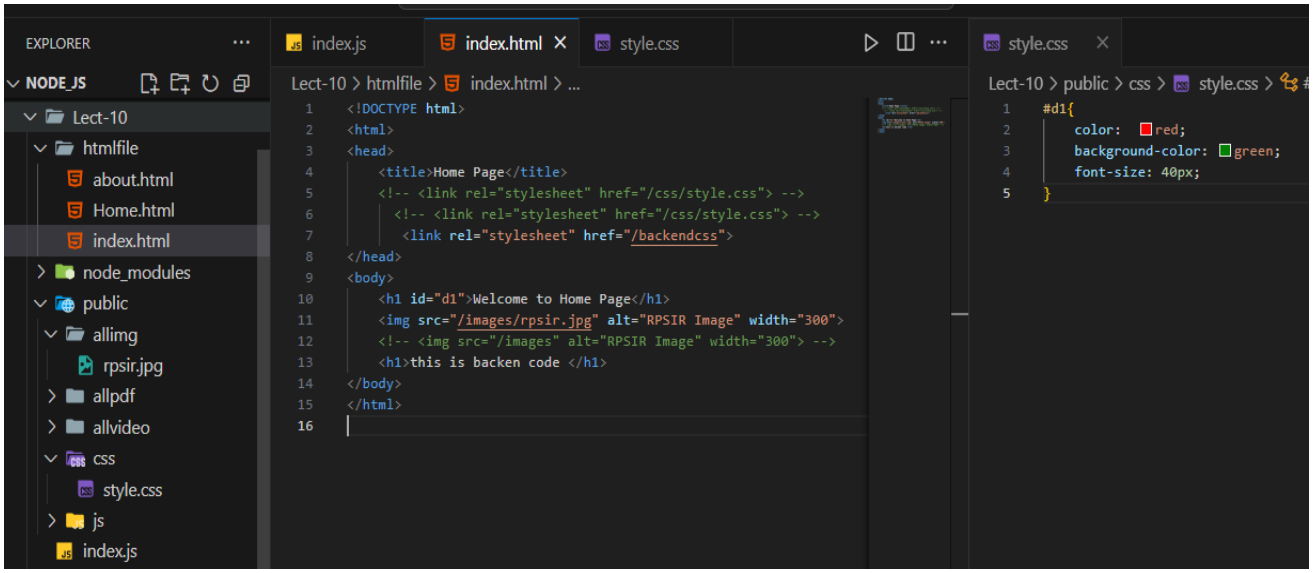
Example: sending external html

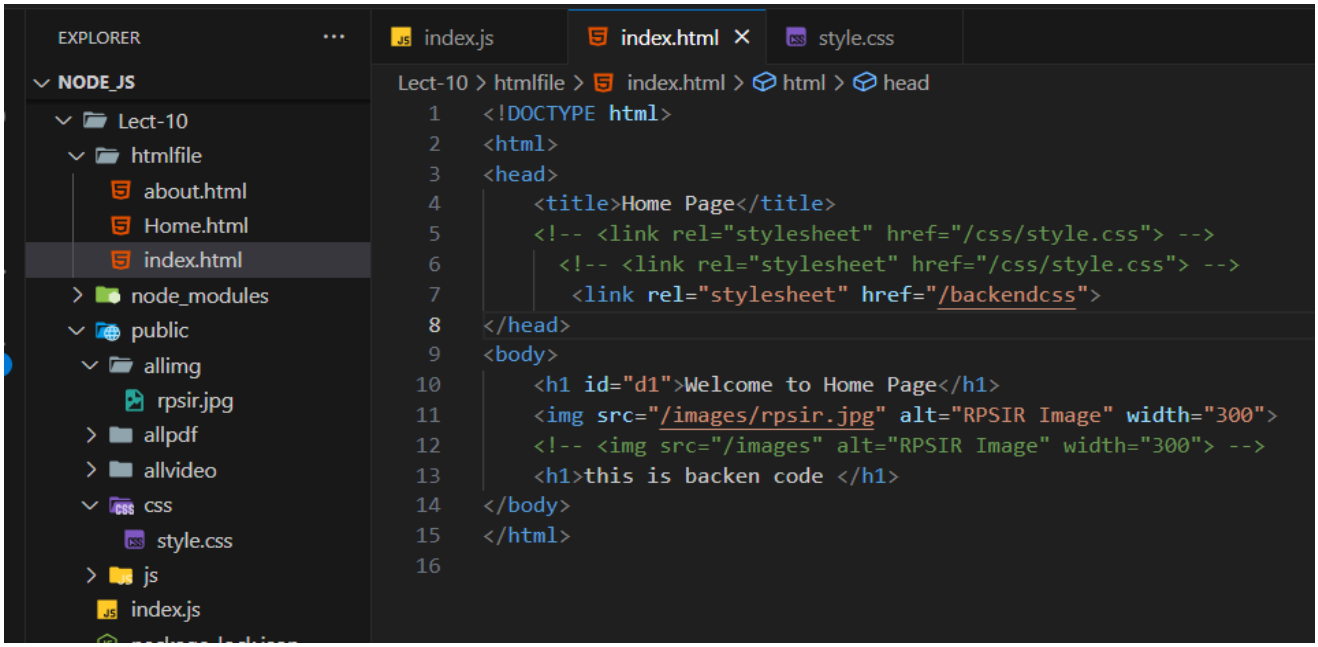


Complete Example

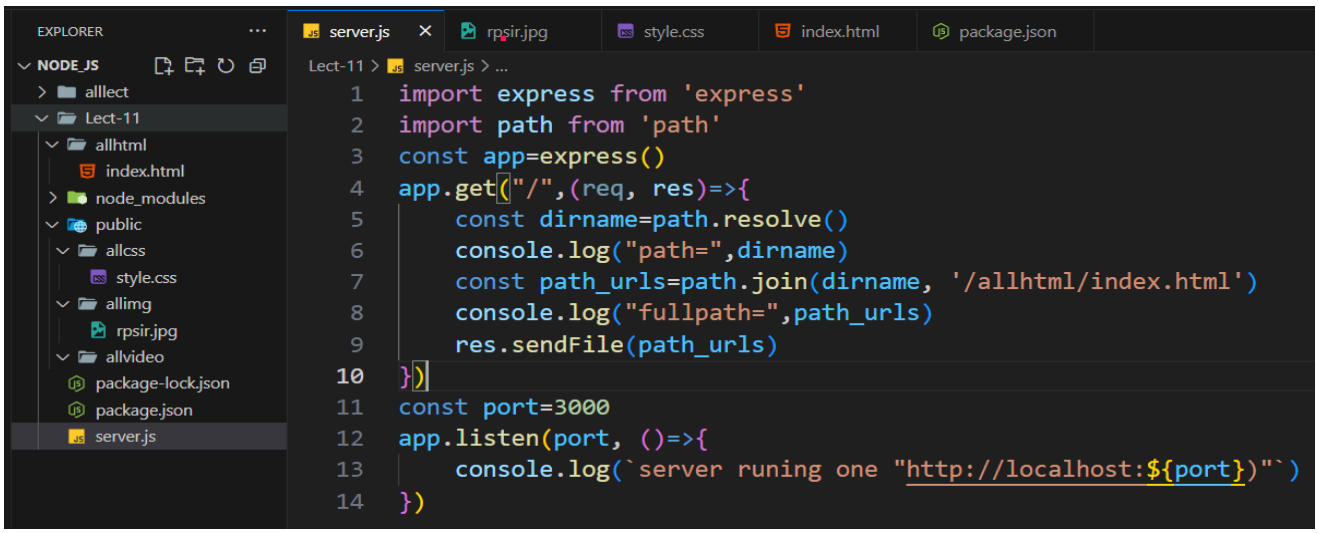


Index.html





Simple example for send external html file as response



Template Engines in Node.js/Express.js (EJS)

❖ **What is a Template Engine?**

- **template engine** is a tool that allows you to build **HTML templates with dynamic content**. Means we can write js variable, condition and looping directly in html file

❖ **Embedded JavaScript (EJS):**

- ejs means **writing JavaScript code directly inside HTML templates** to generate dynamic content on server.

❖ **It enables you to:**

- Inject variables into HTML pages.
- Use programming logic such as **loops** and **conditionals** directly inside templates.
- Generate the final HTML page on the **server side** before sending it to the client's browser.

❖ **Why use one?**

- **Clean separation:** keep your HTML in .ejs files, logic in routes/controllers.
- **Dynamic content**: print variables, loop over lists, use if/else.
- **Reuse**: include headers/footers (partials) and avoid copy-paste.

How to create Templates Engine or (EJS)

Step-1 Create Folder structure

```
my-app/
├── package.json
├── server.js        # Main server file
├── views/           # EJS templates
│   ├── index.ejs
│   ├── media.ejs
│   ├── header.ejs
│   └── footer.ejs
├── public/          #
│   ├── css/
│   │   └── style.css
│   ├── images/
│   ├── pdfs/
│   └── videos/
└── node_modules/
```

Step 2: Initialize the project

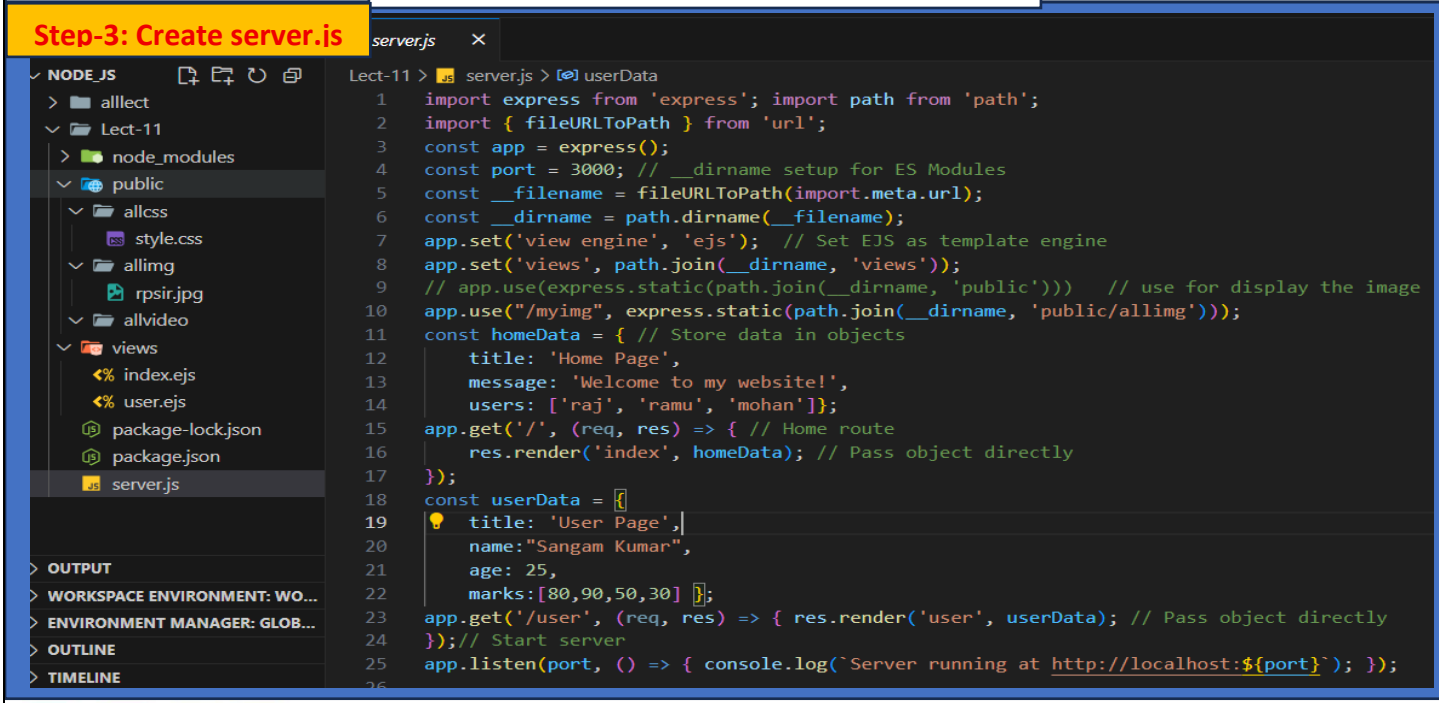
- mkdir my-app
- cd my-app
- npm init -y
- npm install express
- npm install ejs

package.json

```
{
  "name": "lect-11",
  "version": "1.0.0",
  "main": "server.js",
  "type": "module",
  "scripts": {
    "test": "echo \"Error: no
    "start": "node server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "ejs": "^3.1.10",
    "express": "^5.1.0"
  }
}
```

Step-3: Create server.js and set up Express server.
Step-4: Create index.ejs file inside views folder Step-
Step-5: Configure Template Engine (set view engine to ejs and define views directory).
Step-6: Create Routes (/, /user, etc.) and render .ejs templates with data.
Step-7: Start the Server and access templates in the browser (http://localhost:3000).

Step-3: Create server.js



Explanation of the important points in server.js

1. **Imports**

```
import express from 'express';
import path from 'path';
import { fileURLToPath } from 'url';
```

- express → framework for server.
- path → handles file paths.
- fileURLToPath → converts ES module URL to file path.

2. **__dirname setup (needed in ES Modules)**

```
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);
```

- Gets the current file's directory path (since ES modules don't have __dirname by default).

3. **EJS setup**

```
app.set('view engine', 'ejs');
app.set('views', path.join(__dirname, 'views'));
```

- view engine = ejs → Tells Express to render .ejs templates.
- views → Folder where EJS files are stored.

4. **Static files (images, CSS, JS)**

```
// app.use(express.static(path.join(__dirname, 'public')));
app.use("/myimg", express.static(path.join(__dirname, 'public/allimg/rpsir.jpg')));
```

- Normally: app.use(express.static(...)) → makes everything in public/ accessible.
- Your line (/myimg) → only serves **that single image** at http://localhost:3000/myimg.

Note: (Better to keep the first line if you want all images, CSS, JS accessible.)

5. **Data objects**

```
const homeData = { title: 'Home Page', message: 'Welcome to my website!', users: ['raj', 'ramu', 'mohan']};
const userData = { title: 'User Page', age: 25, isAdmin: false };
```

- homeData → passed to **index.ejs**.
- userData → passed to **user.ejs**.

6. **Routes**

```
app.get('/', (req, res) => { res.render('index', homeData); });
app.get('/user', (req, res) => { res.render('user', userData); });
```

- / → loads **index.ejs** with homeData.
- /user → loads **user.ejs** with userData.

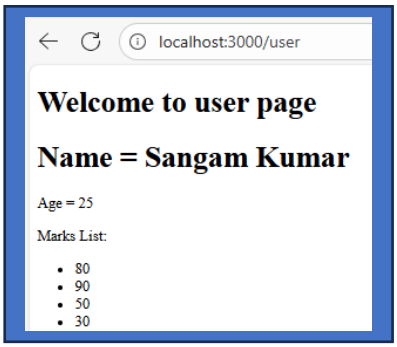
7. **Start server**

```
app.listen(port, () => { console.log(`Server running at http://localhost:${port}`); });
```

- Starts server on port 3000.

Note:

- **Express** server with **EJS templates**.
- **Home (/)** → index.ejs with homeData.
- **User (/user)** → user.ejs with userData.
- **Static files** served (currently only one image via /myimg).



Step-3,4: Create index.ejs and user.ejs

Lect-11 > views > index.ejs > html > body > main > ul > ? > li

```ejs
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title><%= title %></title>
    <link rel="stylesheet" href="/css/style.css">
</head>
<body>
    <header>
        <h1><%= title %></h1>
    </header>
    <main>
        <p><%= message %></p><!-- Correct way: -->
<!-- <img src="/allimg/rpsir.jpg" alt="RPSir"width="300"> -->
        <img src="/myimg/rpsir.jpg" alt="RPSir" width="300">
        <h3>Users:</h3>
        <ul>
            <% users.forEach(user => { %>
                <li><%= user %></li>
            <% }) %>
        </ul>
    </main>
    <footer>
        <p>&copy; <%= new Date().getFullYear() %> My Website</p>
    </footer>
```

user.ejs

Lect-11 > views > user.ejs > html > body > ul > ?

```ejs
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title><%= title %></title>
</head>
<body>

    <h1>Welcome to user page</h1>
    <h1>Name = <%= name %></h1>
    <p>Age = <%= age %></p>

    <p>Marks List:</p>
    <ul>
        <% marks.forEach(function(m) { %>
            <li><%= m %></li>
        <% }) %>
    </ul>

</body>
</html>
```

Expiations

1. **<%= title %> & <%= message %>** → Inserts dynamic values from server (coming via res.render).
2. **<link rel="stylesheet" href="/css/style.css">** → Loads external CSS from public/css/style.css.
3. **Image tag (<img src="/myimg" …>)** → Displays image served from Express static route /myimg.
4. **users.forEach loop** → Iterates over the array of users and displays each one as a list item.
5. **<%= new Date().getFullYear() %>** → Prints the current year automatically in footer.

Note: - This template dynamically shows **title, message, image, list of users, and current year**, all styled with external CSS.

Why we use <%= %> in EJS?

- **<%= %>** → Used to **output dynamic values as HTML (escaped)**.
  Example: <%= title %> → prints the value of title sent from server.
- **<% %>** → Used for **JavaScript logic only** (loops, conditions). It does not print anything directly.
  Example: <% users.forEach(...) %> → runs the loop but doesn't output itself.

- Spaces inside JS (if, for, foreach) are fine → same as normal JS.
- After <%= keep variable name immediately → <%= user %>.
- Don't add random spaces before closing %>.

1. **Inside <% %> (logic only) Spaces don't matter inside <% %>**
   - You can give spaces freely (like normal JS)

   <% if (isAdmin) { %> → ok
   <% if(isAdmin){ %> → ok
   <% for (let i = 0; i < 5; i++) { %> → ok
2. **Inside <%= %> (output)** No space after = before variable name.

   <%= user %> → correct
   <%=user%> → also correct
   <%= user %> → avoid extra spaces at the end
3. **Closing Tags** Always write <% } %> without putting JS code after %>.

   <% } %> → correct
   <% }%> → correct
   <% } %> → avoid extra spaces before `%>`

for loop & conditions in EJS:

1. Loop in EJS

```
<ul>
  <% users.forEach(user => { %>
    <li><%= user %></li>
  <% }) %>
</ul> Iterates over array users and prints each user.
```

2. If Condition

```
<% if (isAdmin) { %>
  <p>Welcome Admin!</p>
<% } else { %>
  <p>Welcome User!</p>
<% } %>
```

Checks isAdmin (boolean) and shows message.

3. For Loop

```
<% for(let i = 1; i <= 5; i++) { %>
  <p>Number: <%= i %></p> <% } %>
```

- **<% %>** = run JavaScript logic (loop/if).
- **<%= %>** = print variable value inside HTML.

Examples of loops & conditions in EJS

1. Check Even/Odd Number

```
<% for(let i = 1; i <= 5; i++) { %>
  <% if(i % 2 === 0) { %>
    <p><%= i %> is Even</p>
  <% } else { %>
    <p><%= i %> is Odd</p>
  <% } %>
<% } %>
```

➔ Prints numbers 1–5 with Even/Odd check.

2. Show Marks with Pass/Fail

```
<% marks.forEach(m => { %>
  <p>
    Marks: <%= m %> -
    <% if(m >= 40) { %>
      Pass
    <% } else { %>
      Fail
    <% } %>
  </p>
<% }) %>
```

→ Loops through marks array, shows **Pass/Fail**.

6. Show First 3 Users Only

```
<% for(let i = 0; i < 3; i++) { %>
  <p>User: <%= users[i] %></p>
<% } %>
```

→ Displays only first 3 users.

7. Conditional Message

```
<% if(users.length === 0) { %>
  <p>No users found!</p>
<% } else { %>
  <p>Total Users: <%= users.length %></p>
<% } %>
```

➔ Checks if array is empty.

8. Nested Loop Example

```
<% classes.forEach((cls, i) => { %>
  <h3>Class <%= i+1 %></h3>
  <ul>
    <% cls.forEach(student => { %>
      <li><%= student %></li>
    <% }) %>
  </ul>
<% }) %>
```

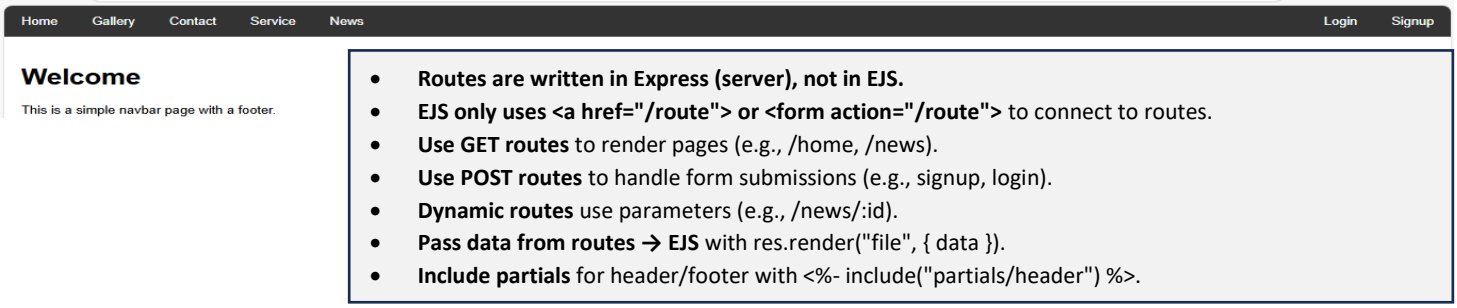
➔ Loops through **array of arrays** (like students in multiple classes).

9. Ternary Operator (inline condition)

```
<p>Status: <%= isAdmin ? "Admin" : "User" %></p>
```

➔ One-line condition using ternary operator.

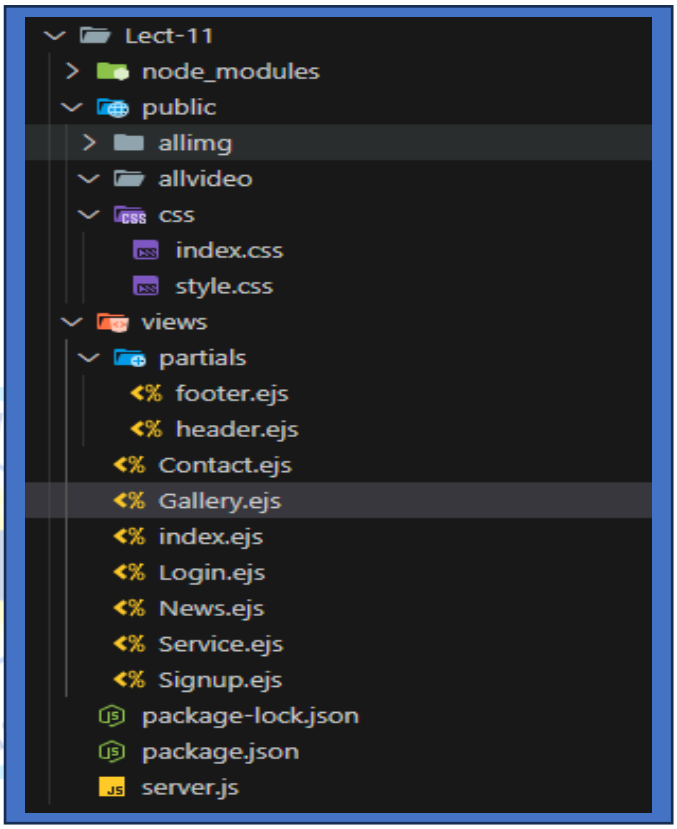
How to add header and footer on all EJS pages and route the route in EJS(HTML file)



- **Routes are written in Express (server), not in EJS.**
- **EJS only uses <a href="/route"> or <form action="/route">** to connect to routes.
- **Use GET routes** to render pages (e.g., /home, /news).
- **Use POST routes** to handle form submissions (e.g., signup, login).
- **Dynamic routes** use parameters (e.g., /news/:id).
- **Pass data from routes → EJS** with res.render("file", { data }).
- **Include partials** for header/footer with <%- include("partials/header") %>.

Step-1: - Create project structure

```
Lect-11/
│
├── app.js (or server.js)
├── package.json
│
├── public/
│   └── css/
│       └── index.css
│       └── nav.css
│
├── views/
│   ├── index.ejs
│   ├── gallery.ejs
│   ├── contact.ejs
│   ├── service.ejs
│   ├── news.ejs
│   ├── login.ejs
│   ├── signup.ejs
│   └── partials/
│       ├── header.ejs
│       └── footer.ejs
│
```



Step-2: -Configure Express (myserver.js)

3) Create the partials views/partials/header.ejs " <%- %> for includes"

4) Use the partials in every page

- **Top of each page: include header and pass the title (already sent from the route). Bottom: include footer.**

Syntax:

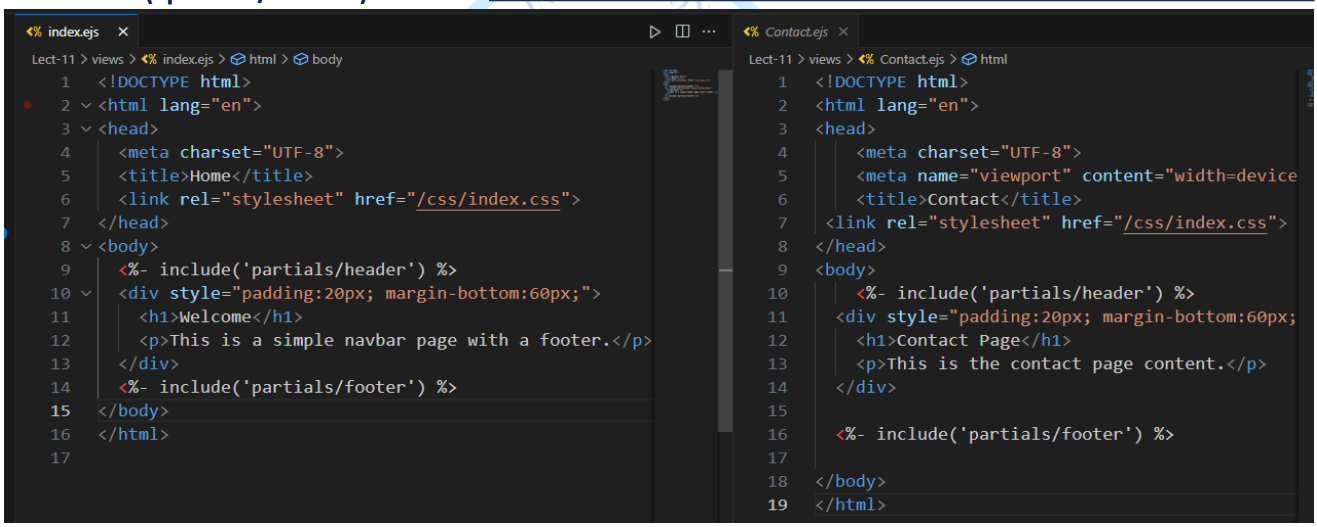
<%- include("partials/header") %>

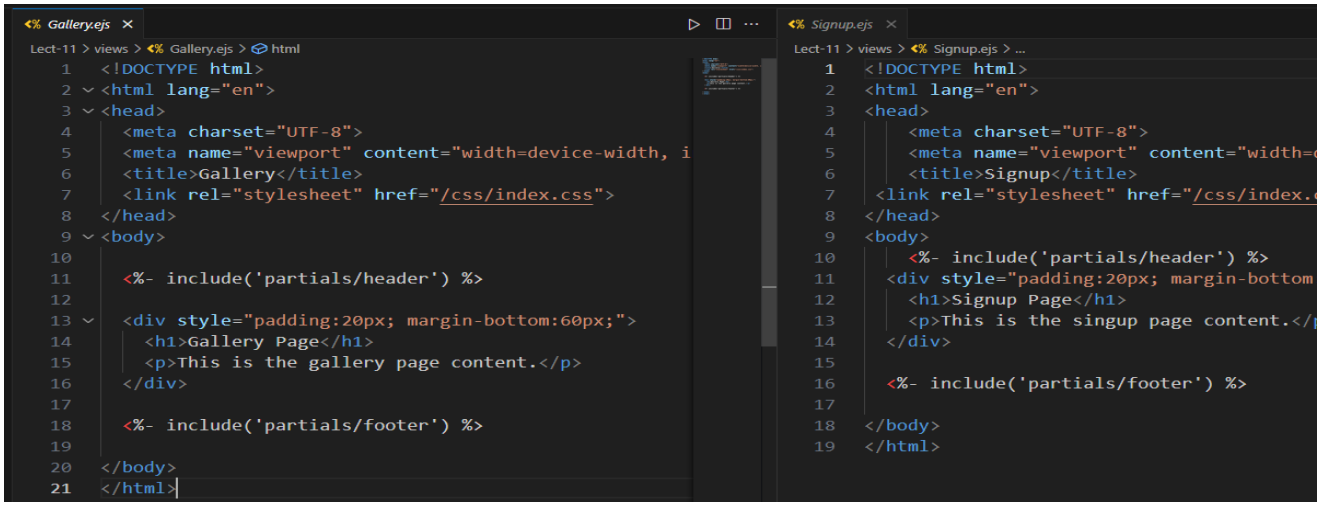
<h2>Welcome to Home</h2>

This is the homepage content.

<%- include("partials/footer") %>

Common mistakes (to avoid that "Unexpected identifier 'partials'" error)
Use <%- include("partials/header") %> (with quotes and parentheses).
Don't write partial/header or partials/header without include().
Keep partial paths relative to views/ (no leading slash).
Use <%- ... %> (not <%= ... %>) for includes.





Form handling (How to get data from the form means front to backend)

Rules for Form Handling

1. Create Form in EJS/HTML

- **Rule:** form tag must have method="post" and action="/your-route".
- **Syntax:**

```
<form action="/route-name" method="post">
  <input type="text" name="fieldName">
  <button type="submit">Submit</button> or
<input type=”submit” value= “submit”>
</form>
```

2. Enable Body Parsing Middleware in Express

- **Rule:** Without this, req.body will be undefined.
- **Syntax:**

```
app.use(express.urlencoded({ extended: true })); // for form data
app.use(express.json()); // for JSON data (API/AJAX)
```

3. Define Backend Route for Form Submission

- **Rule:** Use app.post() (not app.get()) for form submission.
- **Syntax:**

```
app.post("/route-name", (req, res) => {
  const data = req.body;   // access form data
  console.log(data);       // check in console
  res.send("Form submitted!"); or
 res.render(“ejs_file name”)
});
```

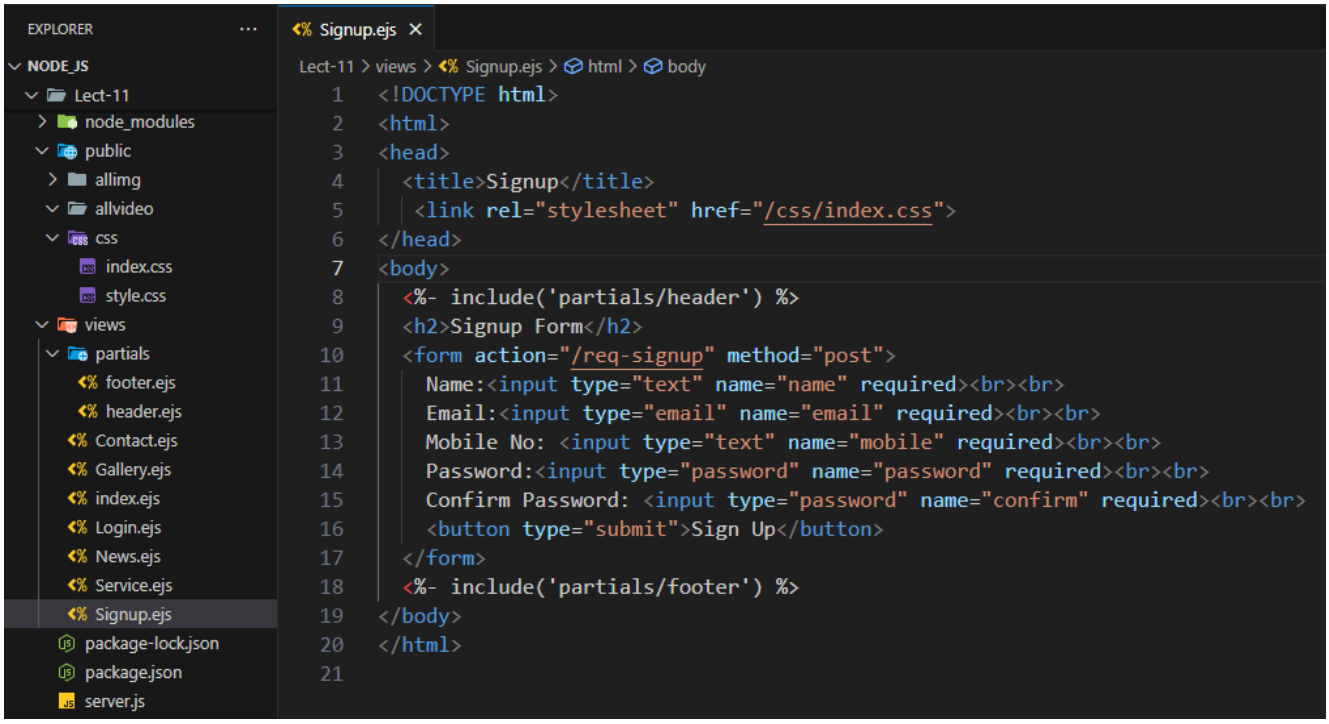
4. Access Data in Backend

- **Rule:** Each input’s name attribute becomes a key in req.body.
- **Syntax:**
  - **req.body**.name; // value from <input name="name">
  - **req.body.email**; // value from <input name="email">
  - **req.body** → text inputs
  - **req.file** → single uploaded file
  - **req.files** → multiple uploaded files
  - **enctype**="multipart/form-data" → required for file uploads

5. Send Response

- **Rule:** Always send a response (render a page or send message).
- **Syntax:**

```
res.send("Form received!");
// or
res.render("success", { user: req.body });
```



EXPLORER

NODE_JS
- Lect-11
  - node_modules
  - public
    - allimg
    - allvideo
    - css
      - index.css
      - style.css
  - views
    - partials
      - footer.ejs
      - header.ejs
      - Contact.ejs
      - Gallery.ejs
      - index.ejs
      - Login.ejs
      - News.ejs
      - Service.ejs
      - Signup.ejs
  - package-lock.json
  - package.json
  - server.js

Signup.ejs

Lect-11 > views > Signup.ejs > html > body

```
<!DOCTYPE html>
<html>
<head>
  <title>Signup</title>
  <link rel="stylesheet" href="/css/index.css">
</head>
<body>
  <%- include('partials/header') %>
  <h2>Signup Form</h2>
  <form action="/req-signup" method="post">
    Name:<input type="text" name="name" required><br><br>
    Email:<input type="email" name="email" required><br><br>
    Mobile No: <input type="text" name="mobile" required><br><br>
    Password:<input type="password" name="password" required><br><br>
    Confirm Password: <input type="password" name="confirm" required><br><br>
    <button type="submit">Sign Up</button>
  </form>
  <%- include('partials/footer') %>
</body>
</html>
```

server.js

Lect-11 > server.js > app.post("/req-signup") callback > user

```
import express from "express";import path from "path";  import { fileURLToPath } from "url
const app = express(); const port = 3000; // Setup __dirname for ES Modules
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename); // Set EJS as template engine
app.set("view engine", "ejs"); app.set("views", path.join(__dirname, "views"));
// Middleware to parse form data from front end to backend
app.use(express.urlencoded({ extended: true })); // Serve static files (CSS, Images, JS)
app.use("/myimg", express.static(path.join(__dirname, "public/allimg")));
app.use(express.static(path.join(__dirname, "public"))); // for CSS, JS
const data = { "Name": "Sangam Kumar", "Age": 18 };
app.get("/", (req, res) => { res.render("index", data); }); // index route
app.get("/Gallery", (req, res) => { res.render("Gallery"); });
app.get("/Contact", (req, res) => { res.render("Contact"); });
app.get("/Service", (req, res) => { res.render("Service"); });
app.get("/News", (req, res) => { res.render("News"); });
app.get("/Signup", (req, res) => { res.render("Signup"); }); // Handle form submission
app.post("/req-signup", (req, res) => {
  const user = {
    name: req.body.name,
    email: req.body.email,
    mobile: req.body.mobile,
    password: req.body.password,
    confirm: req.body.confirm
  };console.log("User data:", user);//   res.send("Signup form submitted successfully!");
 res.render("Login.ejs");});
app.get("/Login", (req, res) => { res.render("Login"); });
app.listen(port, () => {console.log(`Server running at http://localhost:${port}`);});
```

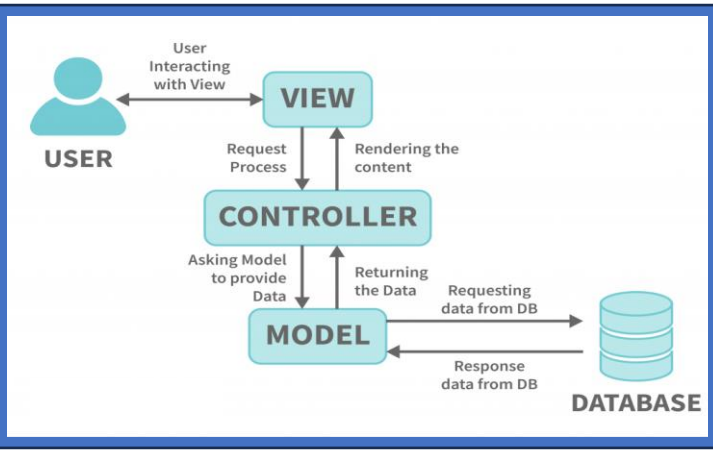
Example **student registration form**

```
<!DOCTYPE html>
<html>
<head>
 <title>Student Registration</title>
</head>
<body>
 <h2>Student Registration Form</h2>
 <form action="/register" method="post" enctype="multipart/form-data">
  <!-- Basic Info -->
  <label>Name:</label>
  <input type="text" name="name" required><br><br>
  <label>Email:</label>
  <input type="email" name="email" required><br><br>
  <label>Mobile No:</label>
  <input type="text" name="mobile" required><br><br>
  <label>Date of Birth:</label>
  <input type="date" name="dob" required><br><br>
  <!-- Gender -->
  <label>Gender:</label>
  <input type="radio" name="gender" value="Male" required> Male
  <input type="radio" name="gender" value="Female" required> Female
  <input type="radio" name="gender" value="Other" required> Other<br><br>
  <!-- File Uploads -->
  <label>Profile Image:</label>
  <input type="file" name="image" accept="image/*" required><br><br>
  <label>Marksheet (PDF):</label>
  <input type="file" name="marksheet" accept="application/pdf" required><br><br>
  <label>Short Video:</label>
  <input type="file" name="video" accept="video/*" required><br><br>
  <!-- Branch -->
  <label>Branch:</label>
  <select name="branch" required>
   <option value="">Select Branch</option>
   <option value="CSE">CSE</option>
   <option value="ECE">ECE</option>
   <option value="ME">ME</option>
   <option value="CE">CE</option>
  </select><br><br>
  <button type="submit">Register</button>
 </form>
</body>
</html>
```

Key Points:

- enctype="multipart/form-data" is **required** for file uploads.
- name attribute for each input is used to access the data in backend (req.body for text, req.file/req.files for files).
- Simple dropdown for branch, radio buttons for gender.

MVC Model in Node JS/ Express JS

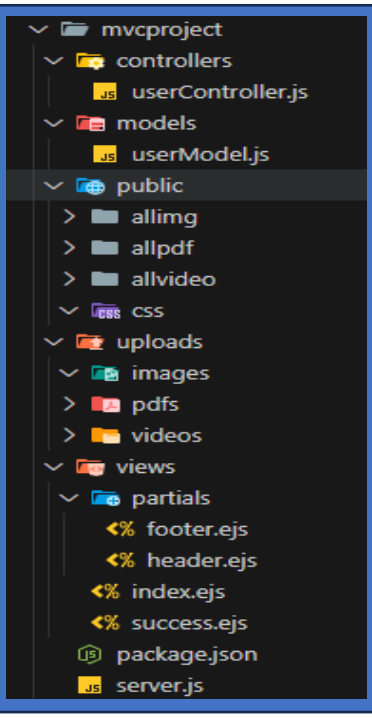


MVC stands for Model-View-Controller, is way to organize code in a clean and scalable way.

1) **Model: -** Handles **data** and **database logic**. Example: Defines what a "User" is and talks to the database.
2) **View: -** Handles the **UI (User Interface)**. What the user sees (HTML, templates like EJS, etc.).
3) **Controller: -** Handles **requests** & **business logic**. Connects Model & View. Gets data from model and sends to view.

Why use MVC? - Keeps code organized. Easy to maintain and scale.

Step-1: Create Folder Structure

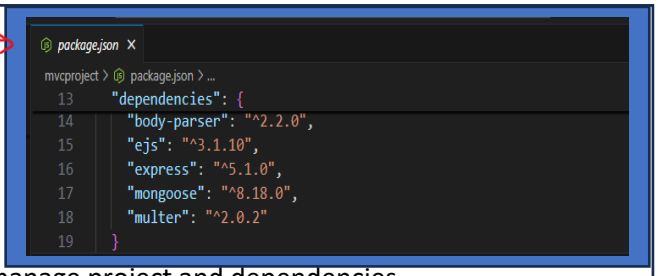



```
project/
│
├── controllers/        → Handle logic (e.g., userController.js)
├── models/             → Define data & interact with DB (e.g., userModel.js)
├── public/             → Static files: CSS, JS, images (e.g., style.css)
├── uploads/            → Store user-uploaded files (images, PDFs, videos)
├── views/              → EJS templates for HTML pages (index.ejs, success.ejs)
├── server.js           → Main server setup and routes
└── package.json        → Project info & dependencies
```

- **controllers/** → Handles **logic** (e.g., processing forms, routes)
- **models/** → Defines **data structure** and interacts with the **database**
- **views/** → Contains **UI templates** (EJS) for dynamic HTML pages
- **public/** → Stores **static files** like CSS, JS, images for the frontend
- **uploads/** → Stores **user-uploaded files** (images, videos, PDFs)
- **server.js** → Main **server setup**: routes, middleware, database connection
- **package.json** → **Project info** & manages **dependencies** (like Express, Mongoose

Step-2: Initialize Project & Install Packages

- ✓ mkdir student-register
- ✓ cd student-register
- ✓ npm init -y
- ✓ **npm** install express mongoose ejs body-parser multer



1) **npm init -y** → Creates package.json with default settings to manage project and dependencies.
2) **npm install express** → Installs Express, a web framework to create servers and handle routes.
3) **npm install mongoose** → Installs Mongoose to define schemas and interact with MongoDB.
4) **npm install ejs** → Installs EJS template engine to render dynamic HTML pages.
5) **npm install body-parser** → Installs middleware to parse incoming form data (JSON or URL-encoded).
6) **npm install multer** → Installs middleware to handle file uploads like images, PDFs, and videos.

Step-3. Connect to MongoDB: - Use MongoDB Atlas or local MongoDB. Replace <your_mongo_uri> in server.js.

How to setting up MongoDB Atlas and MongoDB Compass/Shell:

Step 1: Create MongoDB Atlas Account

1. Go to https://www.mongodb.com/cloud/atlas
2. Click **Sign Up** and create an account (or login if you have one)
3. Verify your email and complete signup

Step 2: Create a Cluster

1. Click **Build a Cluster** → Choose **Free Tier**
2. Select **Cloud Provider & Region** (default is fine)
3. Click **Create Cluster** → Wait a few minutes

Step 3: Create Database User

1. Go to **Database Access** → Click **Add New Database User**
2. Set **Username & Password** → Give **Read and Write** access
3. Save the user

Step 4: Allow IP Access

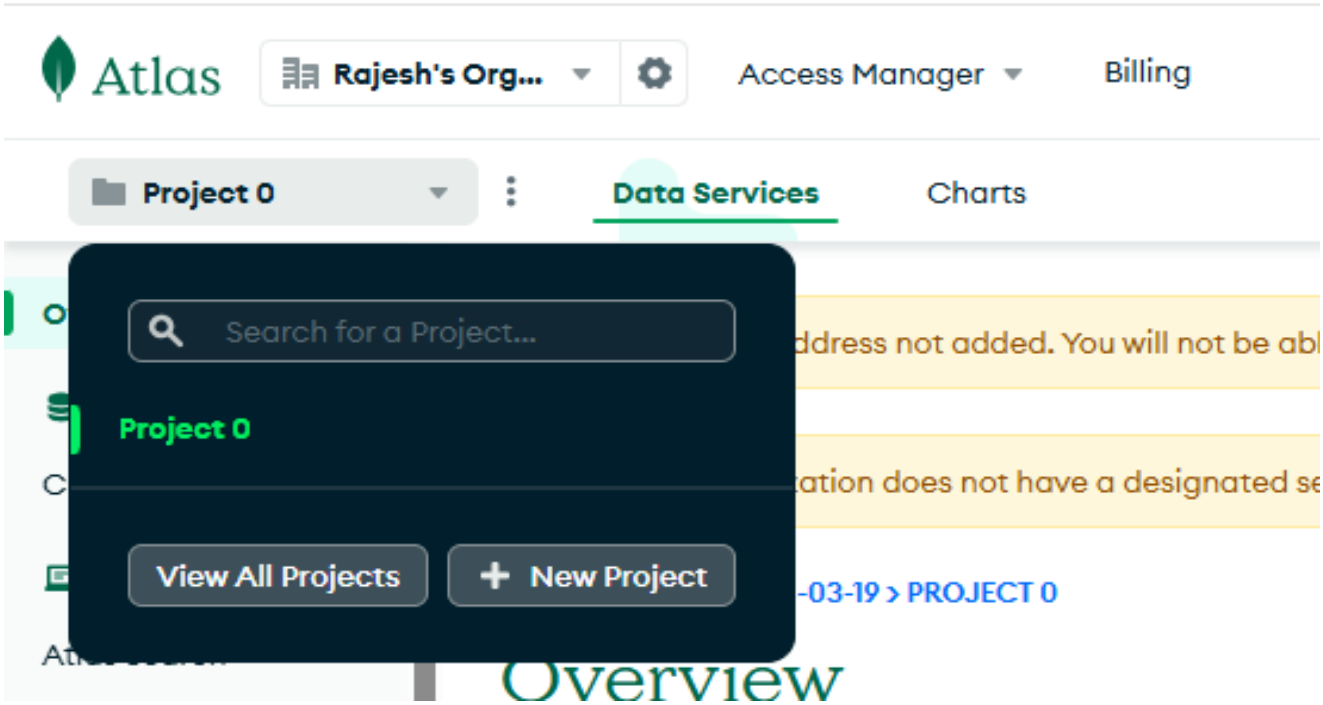
1. Go to **Network Access** → Click **Add IP Address**
2. Choose **Allow Access from Anywhere (0.0.0.0/0)** → Save

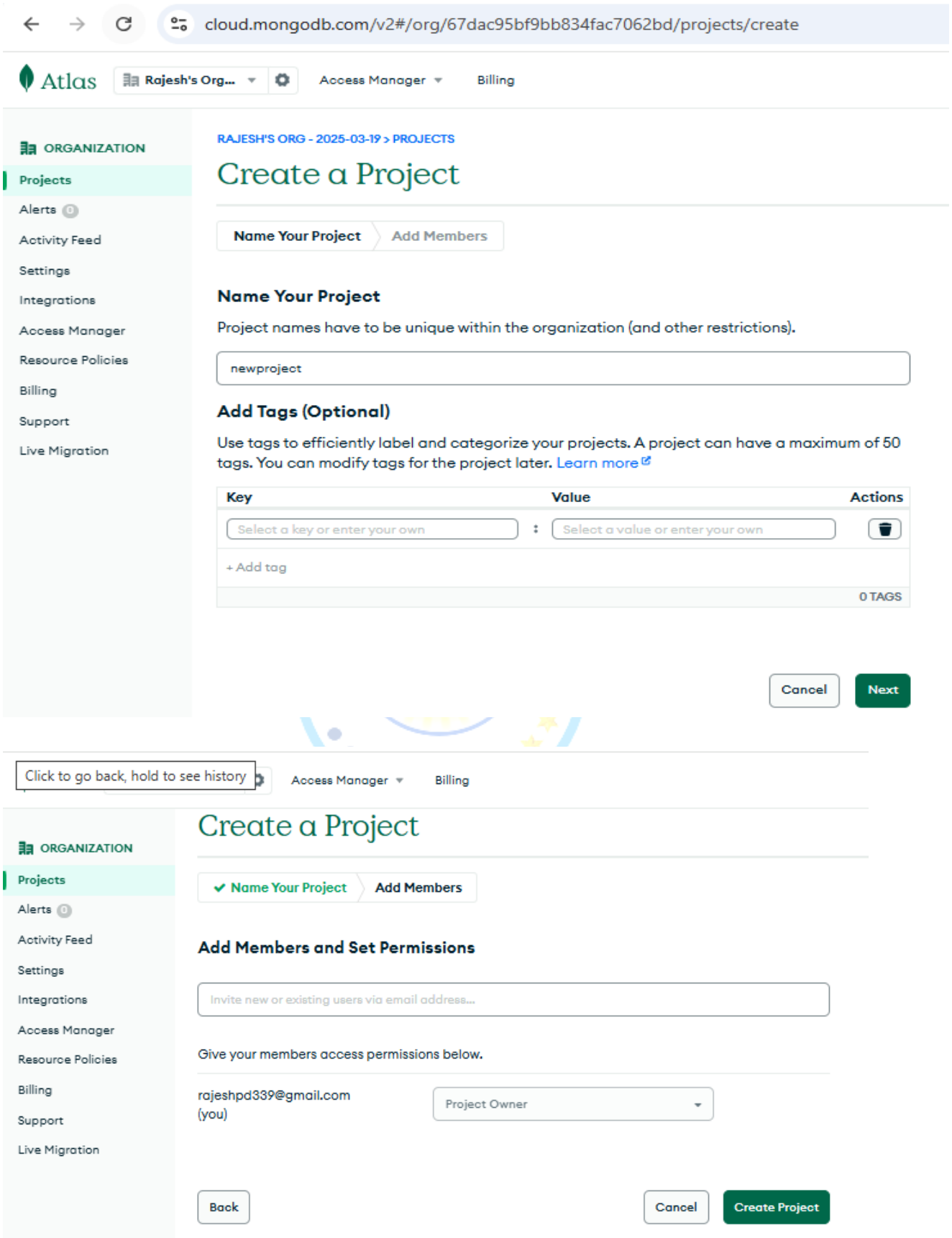
Step 5: Connect Cluster

1. Click **Connect** → Choose **Connect Your Application**
2. Copy **MongoDB URI** → Replace <username> and <password> with your database user credentials

Step 6: Download MongoDB Compass & Shell (Optional)

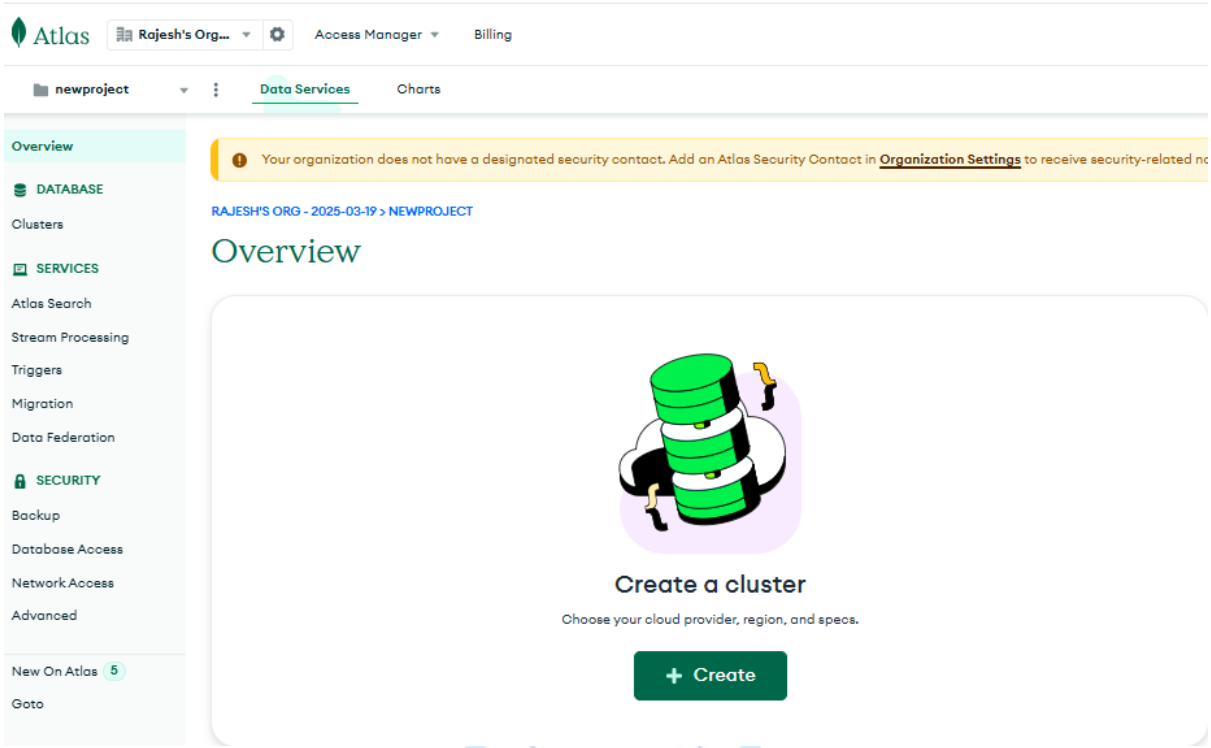
1. Go to https://www.mongodb.com/try/download
2. Download **MongoDB Compass** (GUI to manage DB)
3. Download **Mongo Shell** (CLI to run commands)
4. Install both



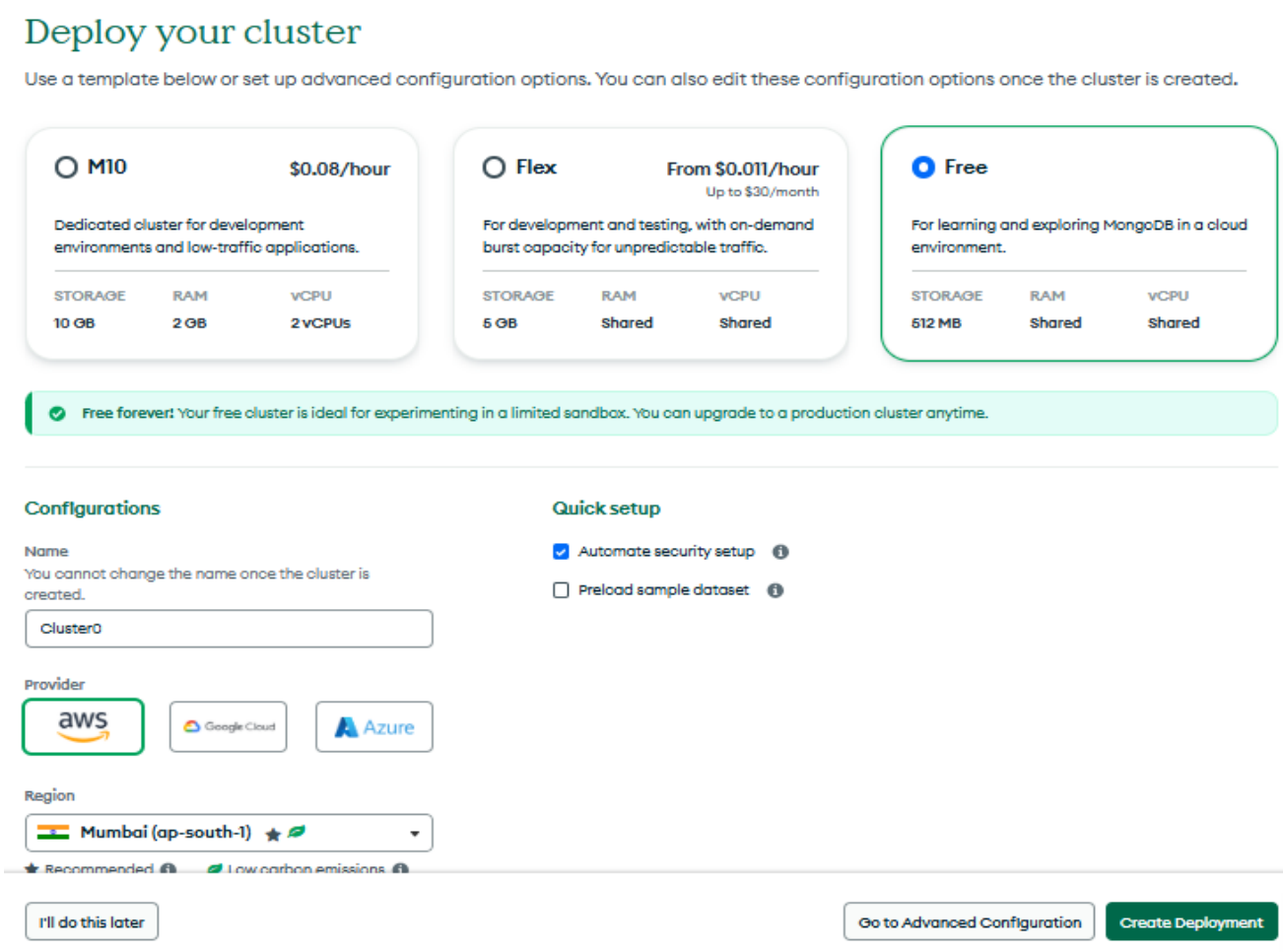

cloud.mongodb.com/v2#/org/67dac95bf9bb834fac7062bd/projects/create
Atlas
Rajesh's Org...
Access Manager
Billing
ORGANIZATION
Projects
Alerts 0
Activity Feed
Settings
Integrations
Access Manager
Resource Policies
Billing
Support
Live Migration
RAJESH'S ORG - 2025-03-19 > PROJECTS
Create a Project
Name Your Project
Add Members
Name Your Project
Project names have to be unique within the organization (and other restrictions).
newproject
Add Tags (Optional)
Use tags to efficiently label and categorize your projects. A project can have a maximum of 50 tags. You can modify tags for the project later. Learn more
Key
Value
Actions
Select a key or enter your own
Select a value or enter your own
+ Add tag
0 TAGS
Cancel
Next

Click to go back, hold to see history
Access Manager
Billing
ORGANIZATION
Projects
Alerts 0
Activity Feed
Settings
Integrations
Access Manager
Resource Policies
Billing
Support
Live Migration
Create a Project
Name Your Project
Add Members
Add Members and Set Permissions
Invite new or existing users via email address...
Give your members access permissions below.
rajeshpd339@gmail.com
(you)
Project Owner
Back
Cancel
Create Project

Create cluster: - cluster means database(table)

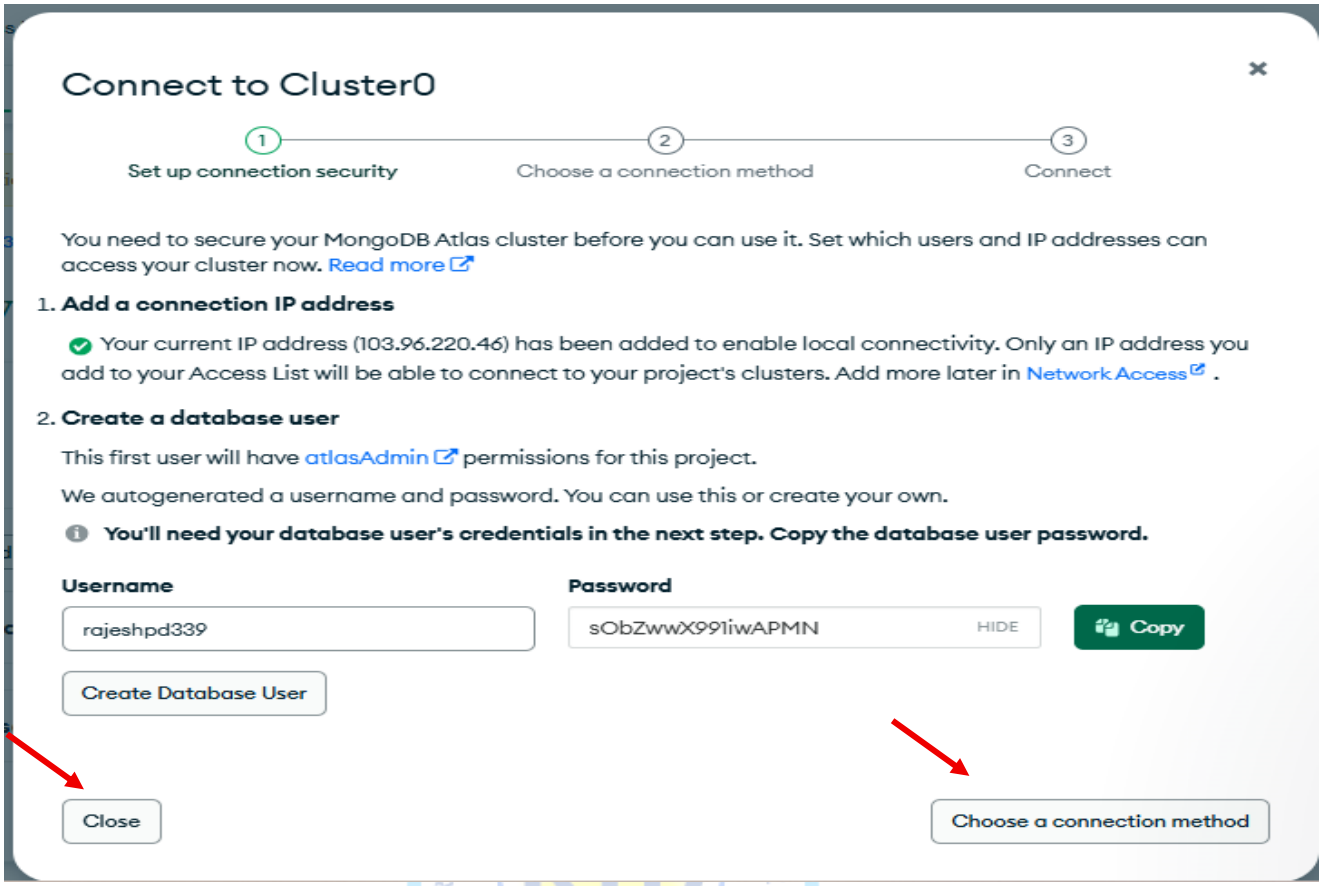


It will take some time then it will redirect to new page



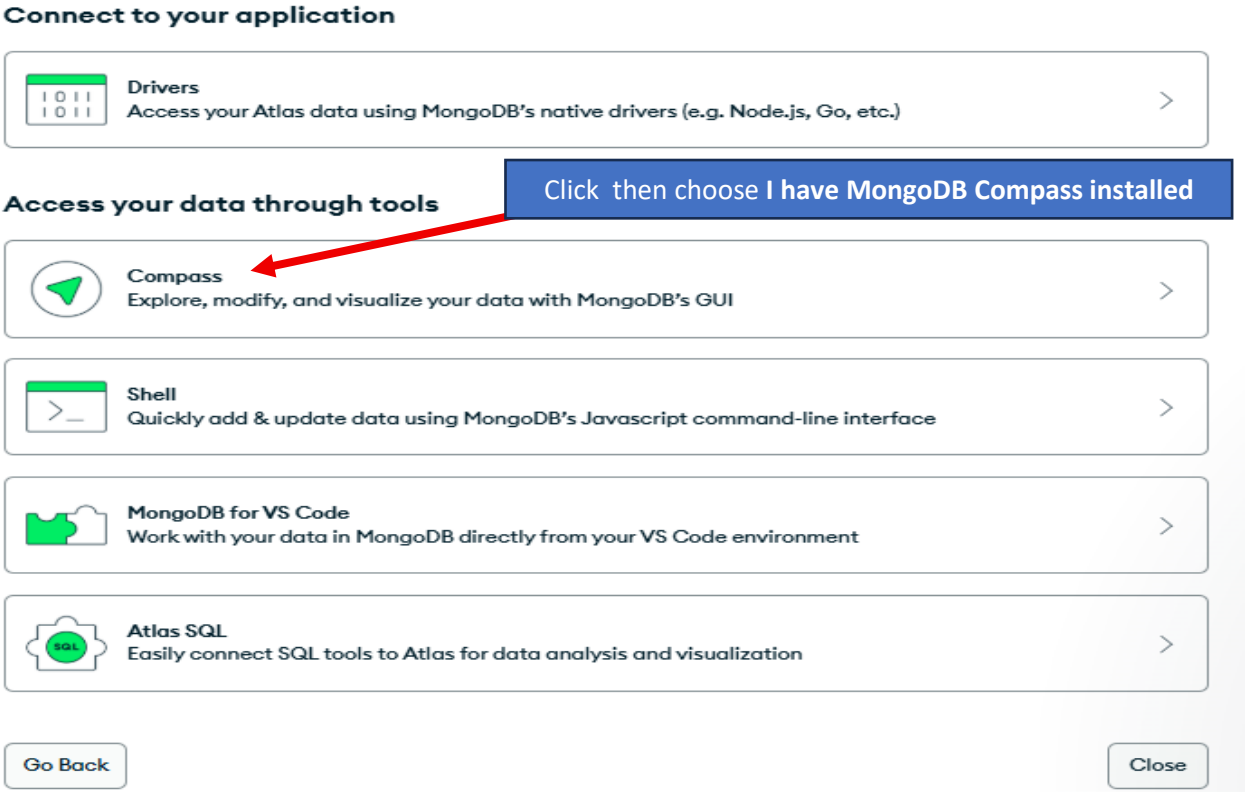
➔ **Select free and aws to deploy your cluster (database to cloud)**

➔ Copy user name and password

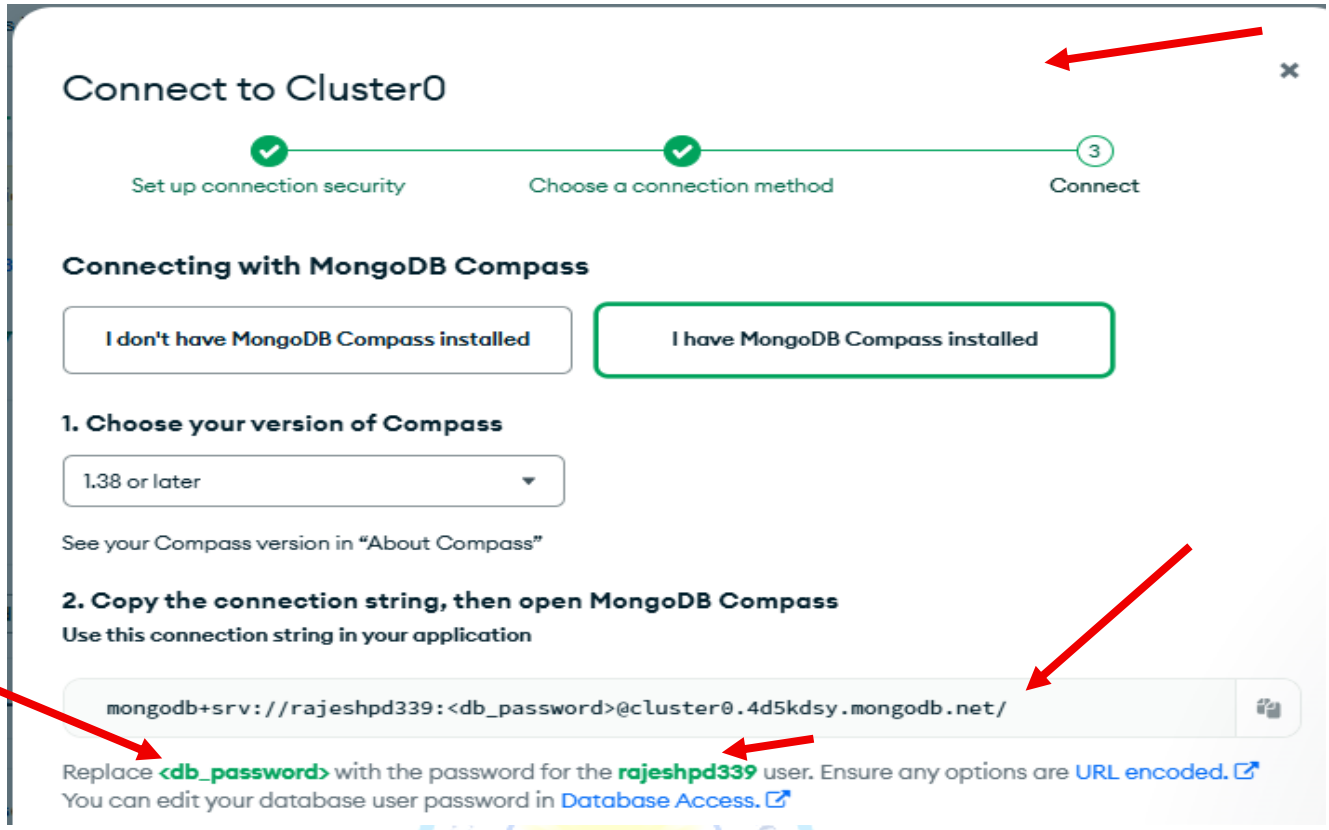


User name: - rajeshpd339 **Password: -** sObZwwX991iwAPMN → sObZwwX991iwAPMN

→ **click create database user and click choose connection method**



➔ **Copy this created urls for connection with mongodb compass to see data**

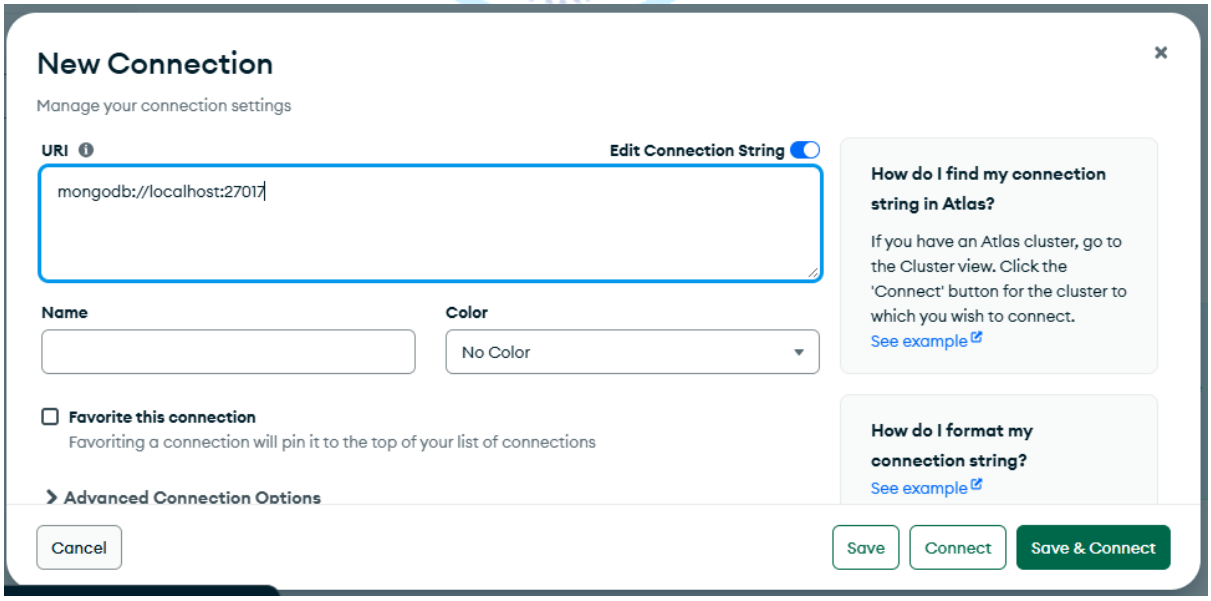


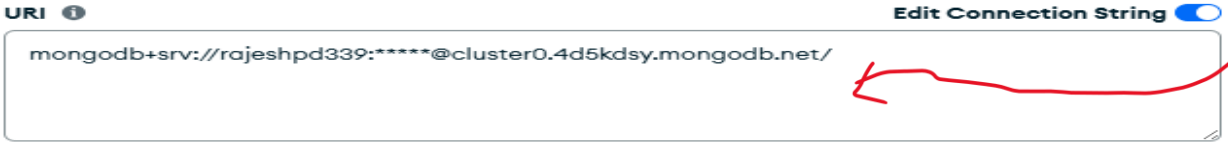
Urls: mongodb+srv://rajeshpd339:<db_password>@cluster0.4d5kdsy.mongodb.net/

➔ Replace <username> and <password> with your DB user details. Keep /test or replace with your database name.

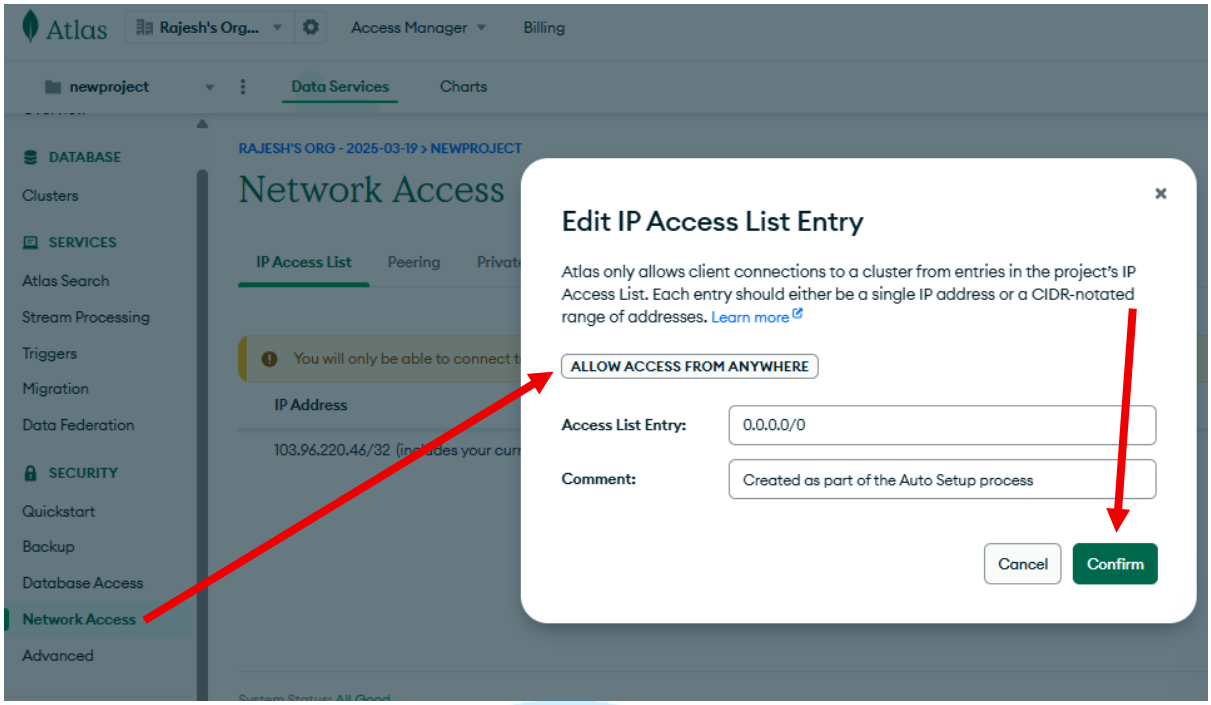
Ursl➔ mongodb+srv://rajeshpd339:sObZwwX991iwAPMN@cluster0.4d5kdsy.mongodb.net/test

➔ Open your mongo db compass in local system and then remove local urls and paste this urls Remove this urls : mongodb://localhost:27017

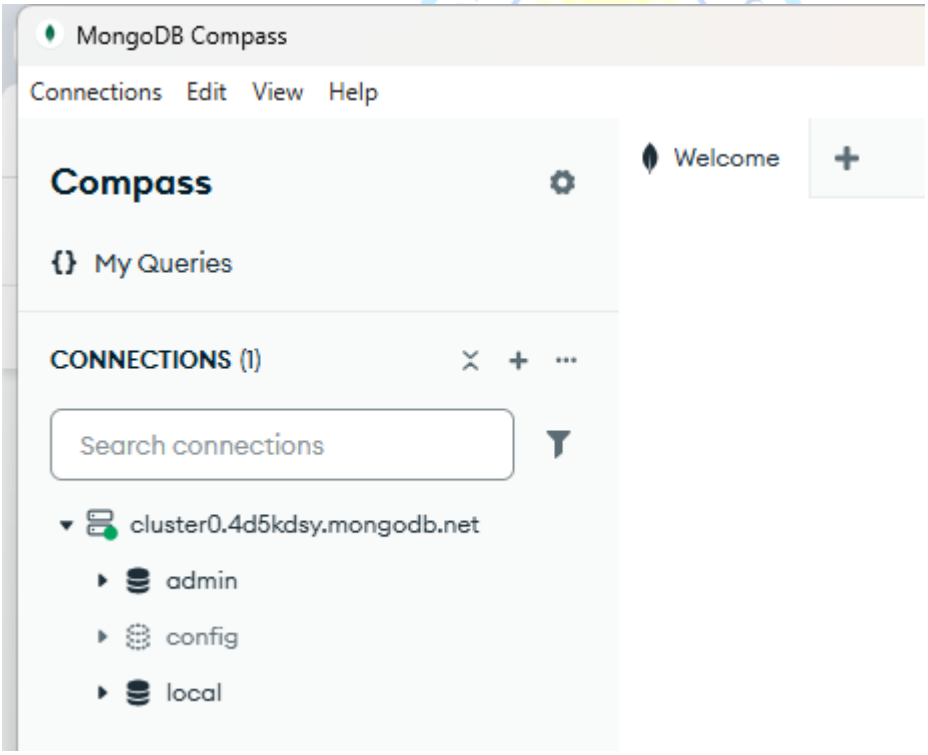




➔ If any error is coming then go to Network access and allow for anywhere



➔ **Now your mongo db. connection will be done now we can send the data from express.**



After connection successful we see admin, config, local now move to next step-4 for model design.

Step-4: - Connect to MongoDB with express js(Backend to Database)

- Install mongosh for connect the **backend to MongoDB**
- **Example:** npm install -g mongosh check install or not:-> mongosh –version
- Connect to MongoDB use this urls
  Mongosh"mongodb+srv://rajeshpd339:sObZwwX991iwAPMN@cluster0.4d5kdsy.mongodb.net/test"

server.js

mvcproject > server.js > ...

```
1  import mongoose from "mongoose";
2  // MongoDB connection URI
3  const mongoURI = "mongodb+srv://<username>:<password>@cluster0.abc123.mongodb.net/<dbname>";
4
5  // Connect to MongoDB
6  mongoose.connect(mongoURI)
7    .then(() => console.log("MongoDB Connected"))
8    .catch(err => console.error("DB Connection Error:", err));
```

- **Replace** <username> and <password> with your MongoDB credentials.
- **Replace** <dbname> with your database name.
- **In Mongoose** 6+ version, useNewUrlParser and useUnifiedTopology are optional

How to give the database name in Express.js with Mongoose

Step 1: Install Required Packages → npm install express mongoose

Step 2: Import Packages→ import mongoose from "mongoose";

Step 3: Create Express App → const app = express();

Step 4: Add Database Name in Connection URI

→ **Database name is "mydatabase"**

const mongoURI = "mongodb+srv://<username>:<password>@cluster0.abc123.mongodb.net/mydatabase";

Note: Replace <username> and <password> with your MongoDB Atlas credentials.

- mydatabase at the end is the **database name**.

Step 5: Connect to MongoDB

```
mongoose.connect(mongoURI)
  .then(() => console.log("MongoDB Connected"))
  .catch(err => console.error("DB Connection Error:", err));
```

Step 6: Start Express Server

```
app.get("/", (req, res) => {
  res.send("Hello, MongoDB is connected!");});
app.listen(port, () => { console.log(`Server is running on ${port}`);});
```

Note: Database name is always in the URI: ...mongodb.net/mydatabase

- No need to give dbName or dnbname in Mongoose options.
- If the database doesn't exist, MongoDB will create it automatically when you add data.

❖ How to check the database is created or not

In Atlas

- Go to **Cluster → Collections**.
- Check if userdatabase exists.
- If not, click **Create Database** → Name: userdatabase, Collection: users.

In Compass Paste connection string:

mongodb+srv://rajeshpd339:sObZwwX991iwAPMN@cluster0.4d5kdsy.mongodb.net/userdatabase

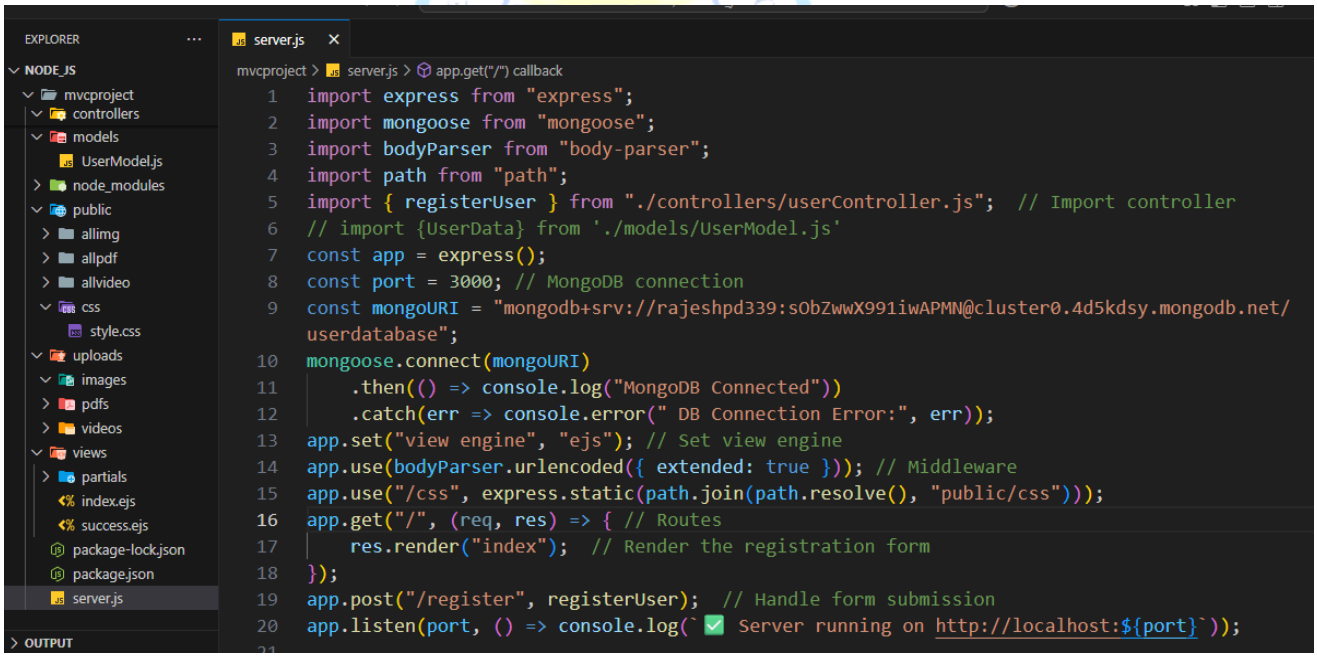
- Click **Connect** → check left sidebar for userdatabase.
- If missing, click **Create Database** → Name: userdatabase, Collection: users.

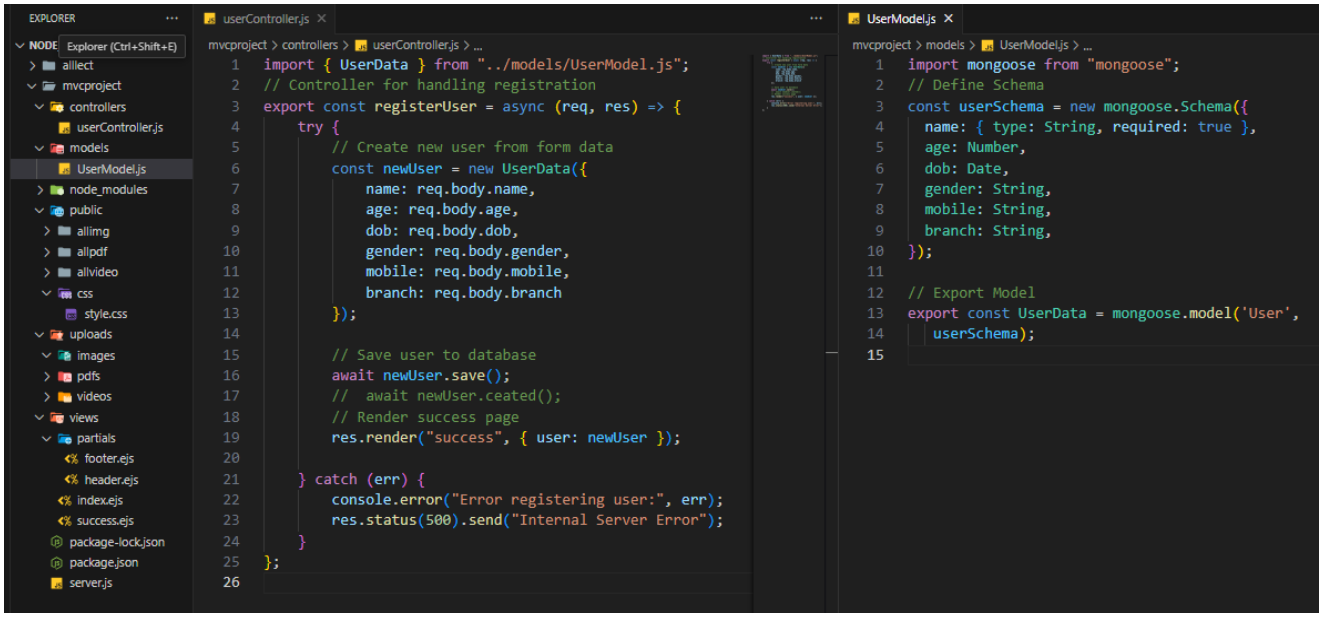
Complete code for connecting the MongoDB database

server.js

```
mvcproject > server.js > ...
1  import express from "express";
2  import mongoose from "mongoose";
3  const myserver = express();
4  const port = 3000;
5  // MongoDB Connection
6  const myurls = "mongodb+srv://rajeshpd339:sObZwwX991iwAPMN@cluster0.4d5kdsy.mongodb.net/
   userdatabase";
7  mongoose.connect(myurls, {
8      useNewUrlParser: true,
9      useUnifiedTopology: true
10     // In Mongoose 6+ version, both are optional
11 })
12 .then(() => console.log("MongoDB Connected"))
13 .catch(err => console.error("DB Connection Error:", err));
14 // Simple route
15 myserver.get("/", (req, res) => {
16   res.send("Hello, MongoDB is connected!");});
17 myserver.listen(port, () => {
18   console.log( `Server is running on ${port}`);
19 });
20
```

Example server.js

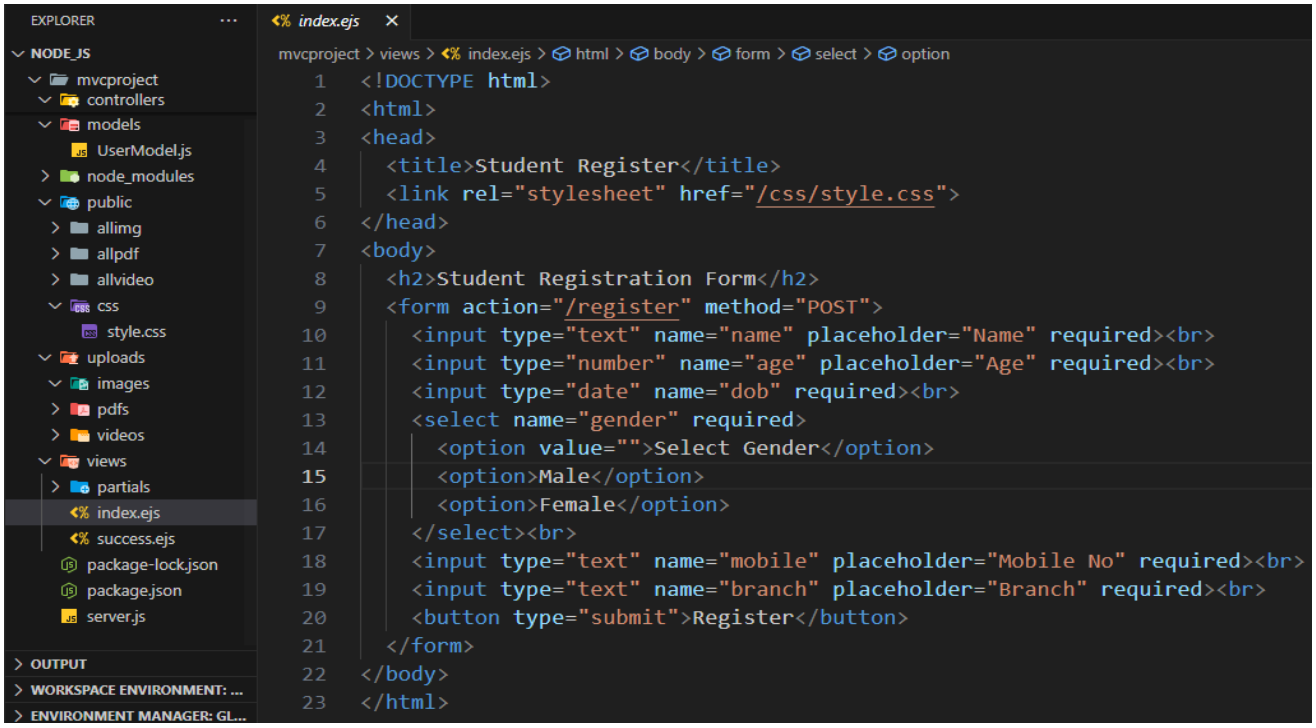




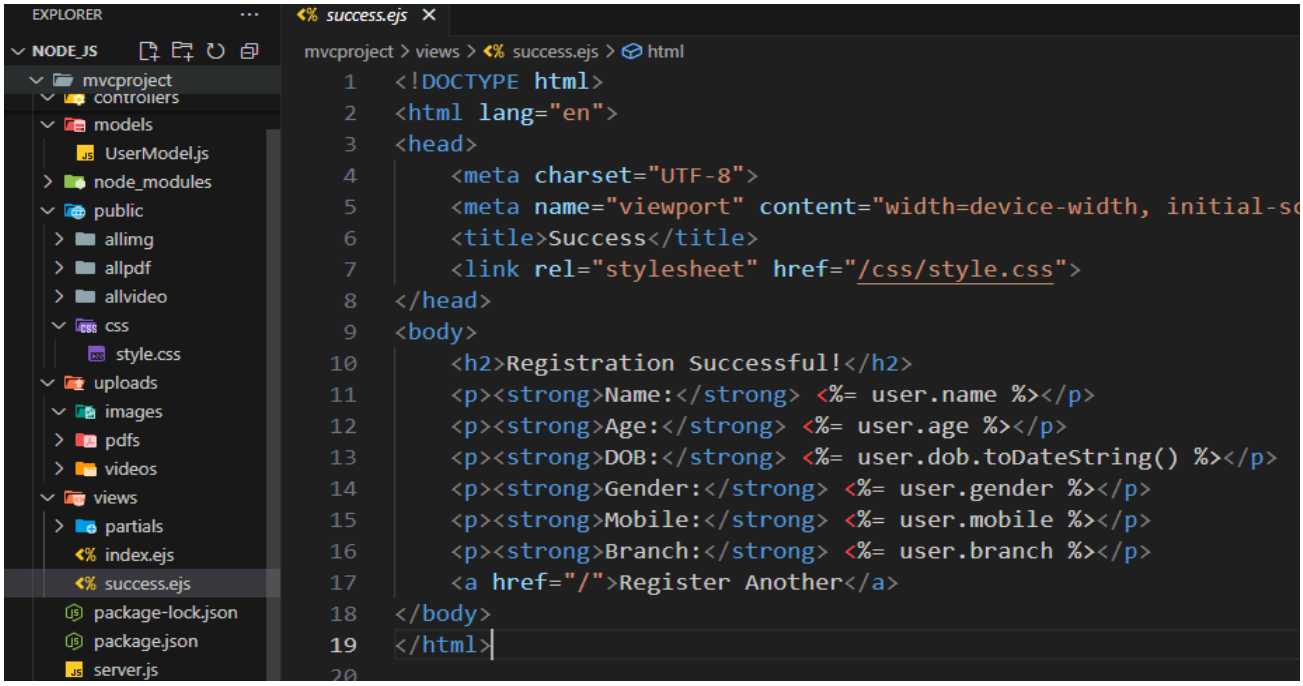
```
// userController.js
import { UserData } from "../models/UserModel.js";
// Controller for handling registration
export const registerUser = async (req, res) => {
    try {
        // Create new user from form data
        const newUser = new UserData({
            name: req.body.name,
            age: req.body.age,
            dob: req.body.dob,
            gender: req.body.gender,
            mobile: req.body.mobile,
            branch: req.body.branch
        });

        // Save user to database
        await newUser.save();
        //  await newUser.ceated();
        // Render success page
        res.render("success", { user: newUser });

    } catch (err) {
        console.error("Error registering user:", err);
        res.status(500).send("Internal Server Error");
    }
};
```

```
// UserModel.js
import mongoose from "mongoose";
// Define Schema
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  age: Number,
  dob: Date,
  gender: String,
  mobile: String,
  branch: String,
});

// Export Model
export const UserData = mongoose.model('User',
    userSchema);
```

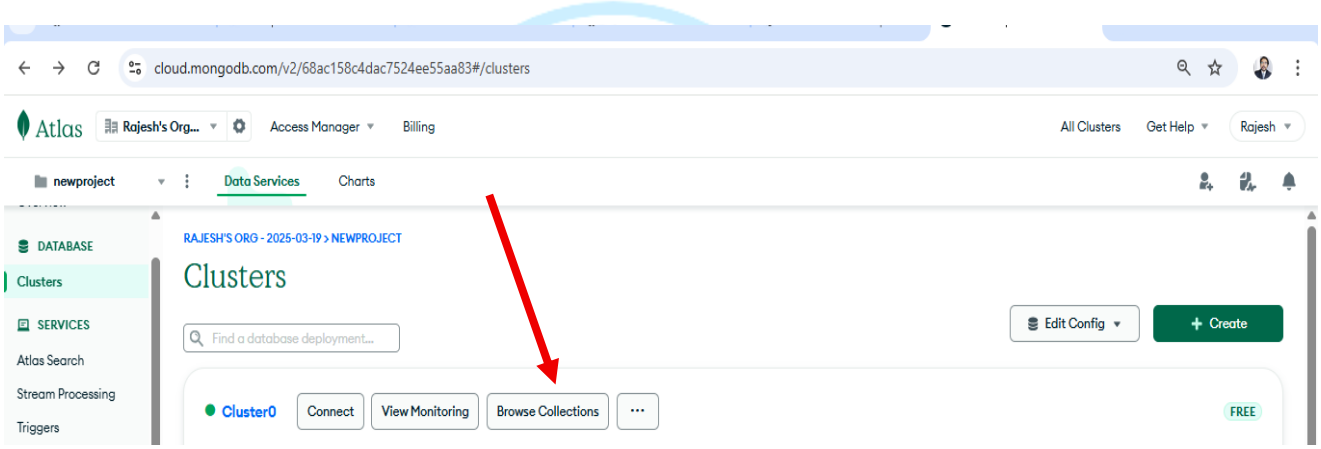
Html file

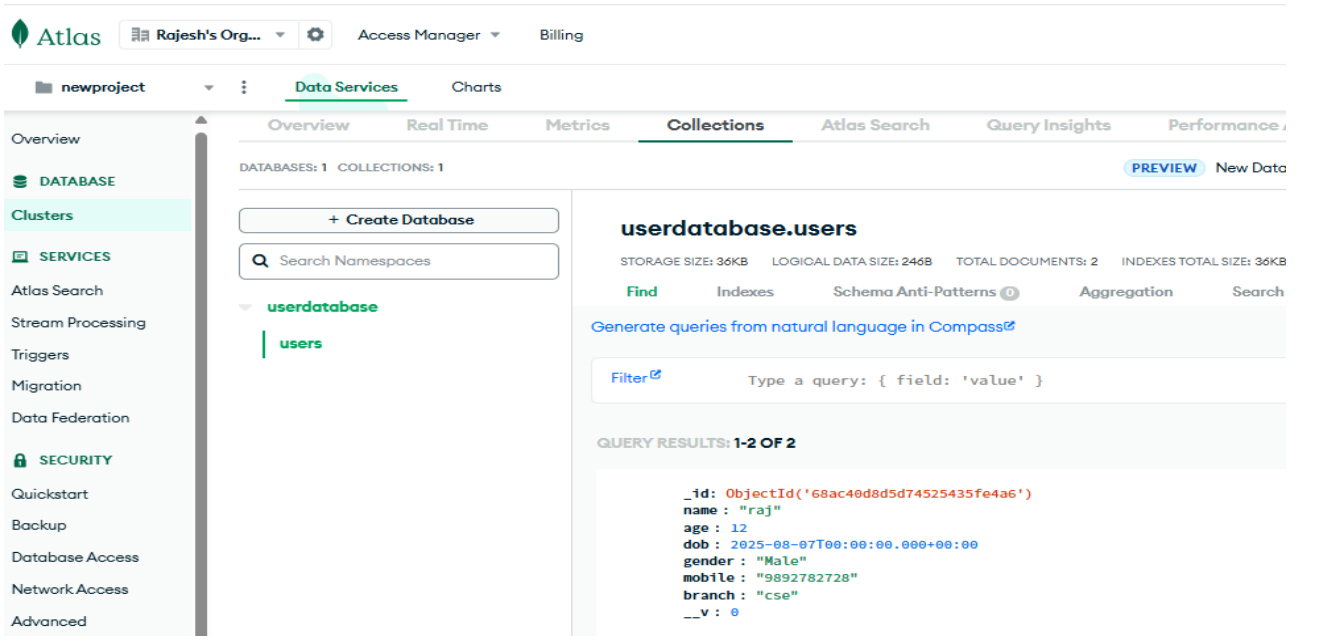


```
<!DOCTYPE html>
<html>
<head>
  <title>Student Register</title>
  <link rel="stylesheet" href="/css/style.css">
</head>
<body>
  <h2>Student Registration Form</h2>
  <form action="/register" method="POST">
    <input type="text" name="name" placeholder="Name" required><br>
    <input type="number" name="age" placeholder="Age" required><br>
    <input type="date" name="dob" required><br>
    <select name="gender" required>
      <option value="">Select Gender</option>
      <option>Male</option>
      <option>Female</option>
    </select><br>
    <input type="text" name="mobile" placeholder="Mobile No" required><br>
    <input type="text" name="branch" placeholder="Branch" required><br>
    <button type="submit">Register</button>
  </form>
</body>
</html>
```



In mongo db click on Browser Collections





How to add plain text, image, video& pdf from Front to Backend to Database

1) Install & Why

npm init -y
npm i express mongoose ejs multer body-parser
optional
npm i -D nodemon

- **express** – web server & routing
- **mongoose** – MongoDB ODM (schemas/models)
- **ejs** – server-side templates (views)
- **multer** – parses multipart/form-data and saves uploaded files
- **body-parser** – parses URL-encoded form fields (text inputs)

2) Folder Structure

```
project/
├─ controllers/
│  └─ userController.js
├─ models/
│  └─ UserModel.js
├─ public/
│  └─ css/
│     └─ style.css
├─ uploads/  ← multer saves files here
├─ views/
│  ├─ index.ejs
│  └─ success.ejs
└─ server.js
```

Add "type": "module" so import works: // package.json

```
{ "type": "module",
  "scripts": { "start": "node server.js", "dev": "nodemon server.js" } }
```

"form → multer → controller → MongoDB → success view"

Create the uploads/ folder (multer needs it).

3) Model (MongoDB) models/UserModel.js

UserModel.js

mvcproject_2 > models > UserModel.js > default

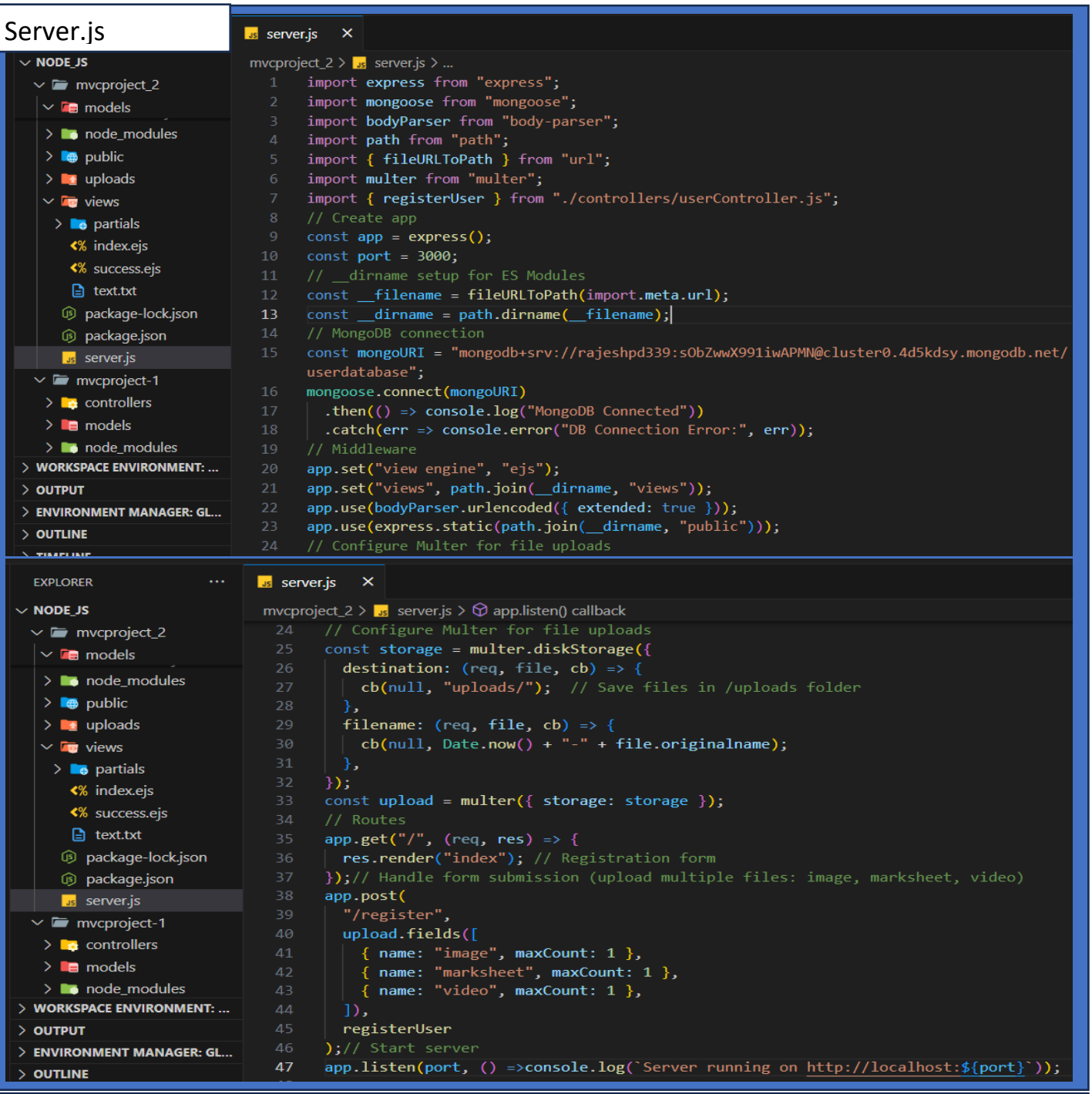
```
1   import mongoose from "mongoose";
2   const userSchema = new mongoose.Schema({
3     name: String,
4     age: Number,
5     dob: Date,
6     gender: String,
7     mobile: String,
8     branch: String,
9     image: String,       // filename for image
10    marksheet: String,   // filename for pdf
11    video: String        // filename for video
12  });
13  const User = mongoose.model("User", userSchema);
14  export default User;
```

- Schema is a **class constructor** in Mongoose.
- new creates a **schema instance** from that class.
- The schema defines the **structure and rules** of documents in a collection.
- Without new, Mongoose still allows it (shortcut), but **best practice** is to use new for clarity and correctness.

```
import User from "../models/Usermodels.js";
export const registerUser=async (req,res)=>{
try{
const user=new User({
    name:req.body.name,
    age:req.body.age_1,
    dob:req.body.dob,
    gender:req.body.gender,
    branch:req.body.branch,
    image:req.files?.image?.[0]?.filename,
    marksheet:req.files?.marksheet?.[0]?.filename ,
    video:req.files?.video?.[0]?.filename
})
await user.save()
res.render("succees")}
catch(error){console.log("form not sumited",error)}}
```

- **import User** → brings in the Mongoose model for users.
- **registerUser function** → handles user registration.
- **req.body** → gets form data (name, age, etc.).
- **req.files** → gets uploaded files (image, marksheet, video).
- **new User({...})** → creates a new user document.
- **user.save()** → saves the document to MongoDB.
- **res.render("success", { user })** → shows success page with user data.
- **try...catch** → handles errors safely.

Server.js



- **express** → framework to build the web server. **mongoose** → connect and interact with MongoDB.
- **body-parser** → parse form data from requests (req.body).**path & fileURLToPath** → handle file paths and set __dirname in ES modules.**multer** → handle file uploads (image, pdf, video).
- **registerUser (controller)** → handles saving user data and files to DB.
- **const app = express()** → create an Express app instance.
- **app.set("view engine", "ejs")** → use EJS templates for rendering views.
- **app.set("views", path.join(__dirname, "views"))** → set the views folder path.
- **app.use(bodyParser.urlencoded({ extended: true }))** → allow form submissions to be parsed.
- **app.use(express.static(path.join(__dirname, "public")))** → serve static files (CSS, JS, images).
- **mongoose.connect(mongoURI)** → connect to MongoDB database.
- **multer.diskStorage({...})** → configure where and how files are saved.
- **upload.fields([...])** → allow uploading multiple files (image, marksheet, video).
- **app.get("/")** → route to display registration form.
- **app.post("/register", ...)** → route to handle form submission + file uploads.
- **app.listen(port, ...)** → start the server at http://localhost:3000.

Views(index.ejs)

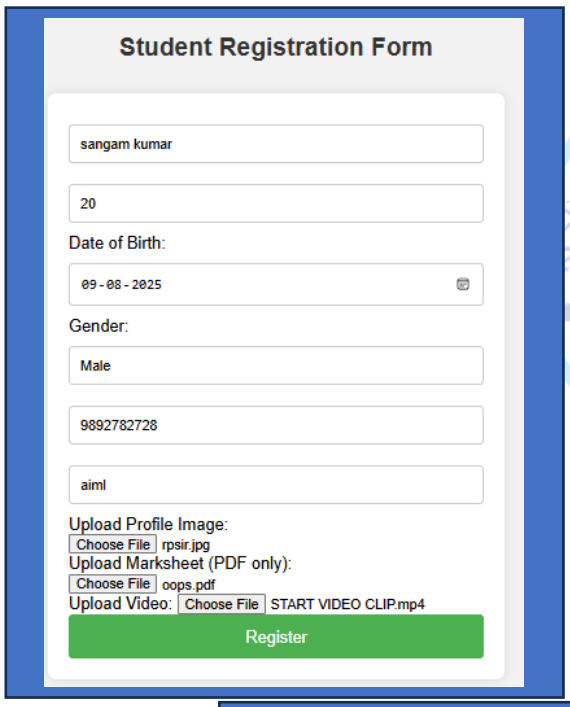
mvcproject-1 > views > index.ejs > html > body

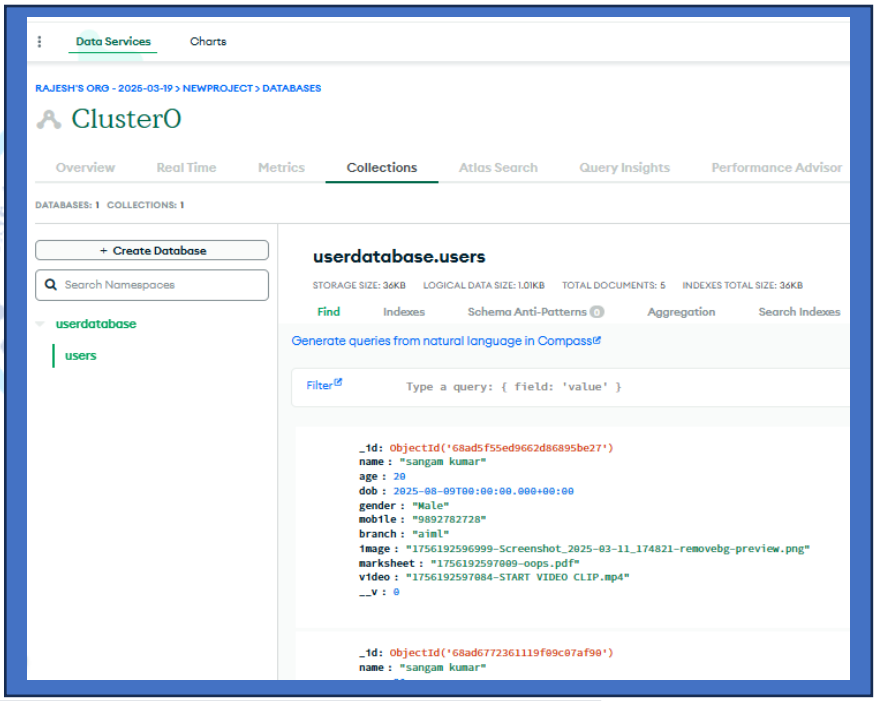
```
<!DOCTYPE html>
<html>
<head>
  <title>Student Register</title>
  <link rel="stylesheet" href="/css/style.css">
</head>
<body>
  <h2>Student Registration Form</h2>
  <form action="/register" method="POST">
    <input type="text" name="name" placeholder="Name" required><br>
    <input type="number" name="age" placeholder="Age" required><br>
    <input type="date" name="dob" required><br>
    <select name="gender" required>
      <option value="">Select Gender</option>
      <option>Male</option>
      <option>Female</option>
    </select><br>
    <input type="text" name="mobile" placeholder="Mobile No" required>
    <input type="text" name="branch" placeholder="Branch" required><br
    <button type="submit">Register</button>
  </form>
</body>
</html>
```

success.ejs

mvcproject-1 > views > success.ejs > html > body

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="w
    <title>Success</title>
    <link rel="stylesheet" href="/cs
</head>
<body>
    <h2>Registration Successful!</h2
</body>
</html>
```





success.ejs

mvcproject_2 > views > success.ejs > html > body

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Registration Success</title>
  <link rel="stylesheet" href="/css/style.css">
</head>
<body>
  <h2>Registration Successful!</h2>

  <p><strong>Name:</strong> <%= user.name %></p>
  <p><strong>Age:</strong> <%= user.age %></p>
  <p><strong>DOB:</strong> <%= user.dob ? new Date(user.dob)
  <p><strong>Gender:</strong> <%= user.gender %></p>
  <p><strong>Mobile:</strong> <%= user.mobile %></p>
  <p><strong>Branch:</strong> <%= user.branch %></p>
```

7. Create View views/index.ejs

```
<!DOCTYPE html>
<html>
<head>
 <title>Student Register</title>
 <link rel="stylesheet" href="/css/style.css">
</head>
<body>
 <h2>Student Registration Form</h2>
 <form action="/register" method="POST" enctype="multipart/form-data">
  <input type="text" name="name" placeholder="Name" required><br>
  <input type="number" name="age" placeholder="Age" required><br>
  <input type="date" name="dob" required><br>
  <select name="gender" required>
   <option value="">Select Gender</option>
   <option>Male</option>
   <option>Female</option>
  </select><br>
  <input type="text" name="mobile" placeholder="Mobile No" required><br>
  <input type="text" name="branch" placeholder="Branch" required><br>
  <label>Upload Profile Image:</label><br>
  <input type="file" name="image" accept="image/*" required><br>
  <label>Upload Marksheet (PDF):</label><br>
  <input type="file" name="marksheet" accept=".pdf" required><br>
  <label>Upload Short Video:</label><br>
  <input type="file" name="video" accept="video/*" required><br>
  <button type="submit">Register</button>
 </form>
</body>
</html>
```

views/success.ejs

```
<h2>Registration Successful!</h2>
<p>Name: <%= user.name %></p>
<p>Branch: <%= user.branch %></p>
<p>Image: <%= user.image %></p>
<p>Marksheet: <%= user.marksheet %></p>
<p>Video: <%= user.video %></p>
<a href="/">Register Another</a>
```

8. Static CSS public/css/style.css

```
body { font-family: sans-serif; padding: 20px;}
input, select {
 display: block;
 margin: 10px 0;
 padding: 8px;
 width: 300px;
}
button {
 padding: 10px 20px;}
```

9. Main Server File server.js

```
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');
const path = require('path');
const multer = require('multer');
const userController = require('./controllers/userController');
const app = express();
// MongoDB Connection
mongoose.connect('your_mongo_uri_here', {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => console.log('MongoDB connected'))
  .catch(err => console.error(err));
// Middleware
app.use(bodyParser.urlencoded({ extended: true }));
app.use(express.static('public'));
app.use('/uploads', express.static('uploads'));
app.set('view engine', 'ejs');
// Multer Config
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    if (file.fieldname === 'image') cb(null, 'uploads/images');
    else if (file.fieldname === 'marksheet') cb(null, 'uploads/pdfs');
    else if (file.fieldname === 'video') cb(null, 'uploads/videos');},
  filename: function (req, file, cb) {
    cb(null, Date.now() + '-' + file.originalname); }});
const upload = multer({ storage: storage });
const cpUpload = upload.fields([
  { name: 'image', maxCount: 1 },
  { name: 'marksheet', maxCount: 1 },
  { name: 'video', maxCount: 1 }]);// Routes
app.get('/', (req, res) => res.render('index'));
app.post('/register', cpUpload, userController.registerUser);// Start Server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => console.log(`Server running on http://localhost:${PORT}`));
```

Contrals.js

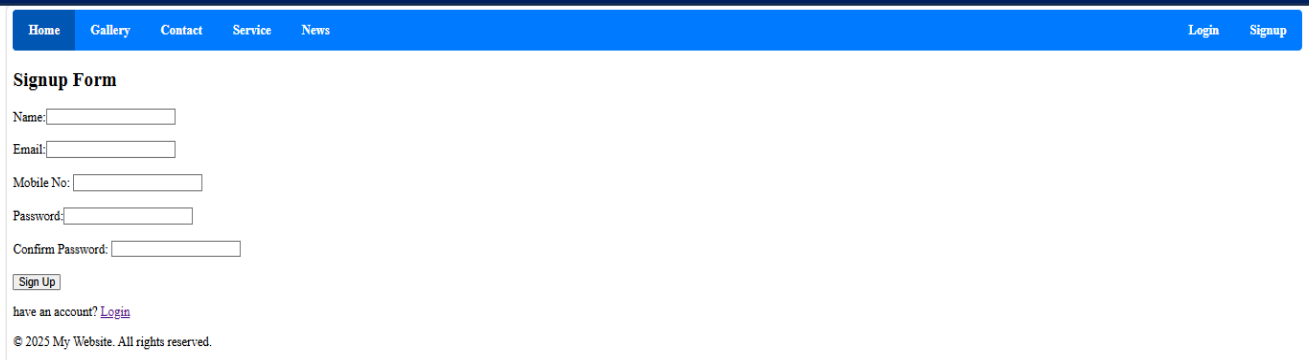
```
import { UserData } from "../models/UserModel.js";
import path from "path";// Controller for handling registration
export const registerUser = async (req, res) => {
   try {
      // Save uploaded files paths
      const imagePath = req.files.image ? `/uploads/images/${req.files.image[0].filename}` : "";
      const marksheetPath = req.files.marksheet ? `/uploads/pdfs/${req.files.marksheet[0].filename}` : "";
     const videoPath = req.files.video ? `/uploads/videos/${req.files.video[0].filename}` : "";
      // Create new user document
      const newUser = new UserData({
         name: req.body.name,
```

```
        age: req.body.age,
        dob: req.body.dob,
        gender: req.body.gender,
        mobile: req.body.mobile,
        branch: req.body.branch,
        image: imagePath,
        marksheet: marksheetPath,
        video: videoPath});
      await newUser.save(); // Render success page
      res.render("success", { user: newUser });
    } catch (err) {
      console.error("Error registering user:", err);
      res.status(500).send("Internal Server Error"); }};
```

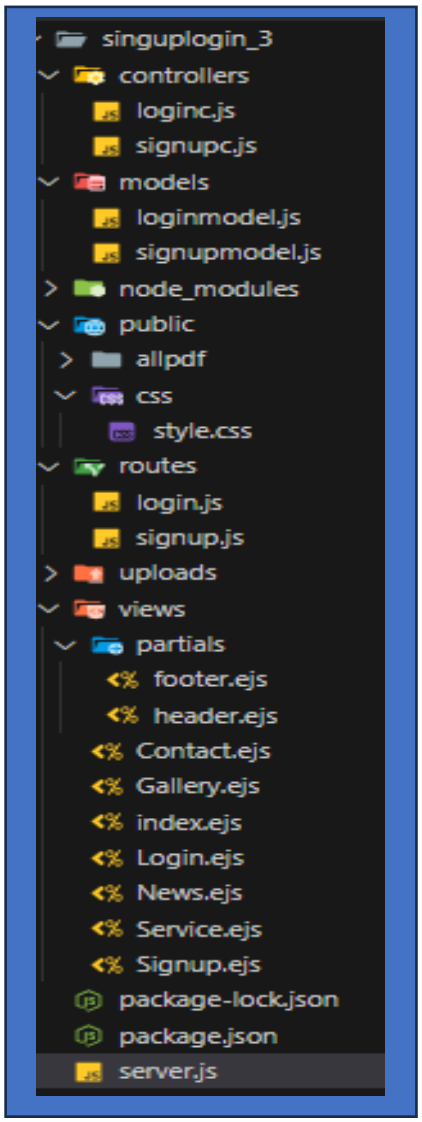
Server.js

```
import express from "express";
import mongoose from "mongoose";
import bodyParser from "body-parser";
import multer from "multer";
import path from "path";
import { registerUser } from "./controllers/userController.js";
const app = express();
const port = 3000;
// MongoDB connection
const mongoURI =
"mongodb+srv://rajeshpd339:sObZwwX991iwAPMN@cluster0.4d5kdsy.mongodb.net/userdatabase";
mongoose.connect(mongoURI)
  .then(() => console.log("MongoDB Connected"))
  .catch(err => console.error("DB Connection Error:", err));
// Set view engineapp.set("view engine", "ejs");// Middleware
app.use(bodyParser.urlencoded({ extended: true }));
app.use("/css", express.static(path.join(path.resolve(), "public/css")));
app.use("/uploads", express.static(path.join(path.resolve(), "uploads")));
// Multer Storage Configuration
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    if(file.fieldname === "image") cb(null, "uploads/images");
    else if(file.fieldname === "marksheet") cb(null, "uploads/pdfs");
    else if(file.fieldname === "video") cb(null, "uploads/videos");},
  filename: (req, file, cb) => {
    cb(null, Date.now() + "-" + file.originalname);}});
const upload = multer({ storage });
// Routesapp.get("/", (req, res) => res.render("index"));
// Form submission route
app.post("/register", upload.fields([
  { name: "image", maxCount: 1 },
  { name: "marksheet", maxCount: 1 },
  { name: "video", maxCount: 1 }
]), registerUser);
app.listen(port, () => console.log(`Server running on ${port}`));
```

Front End +Backend+ Database Connection in Node js/Express



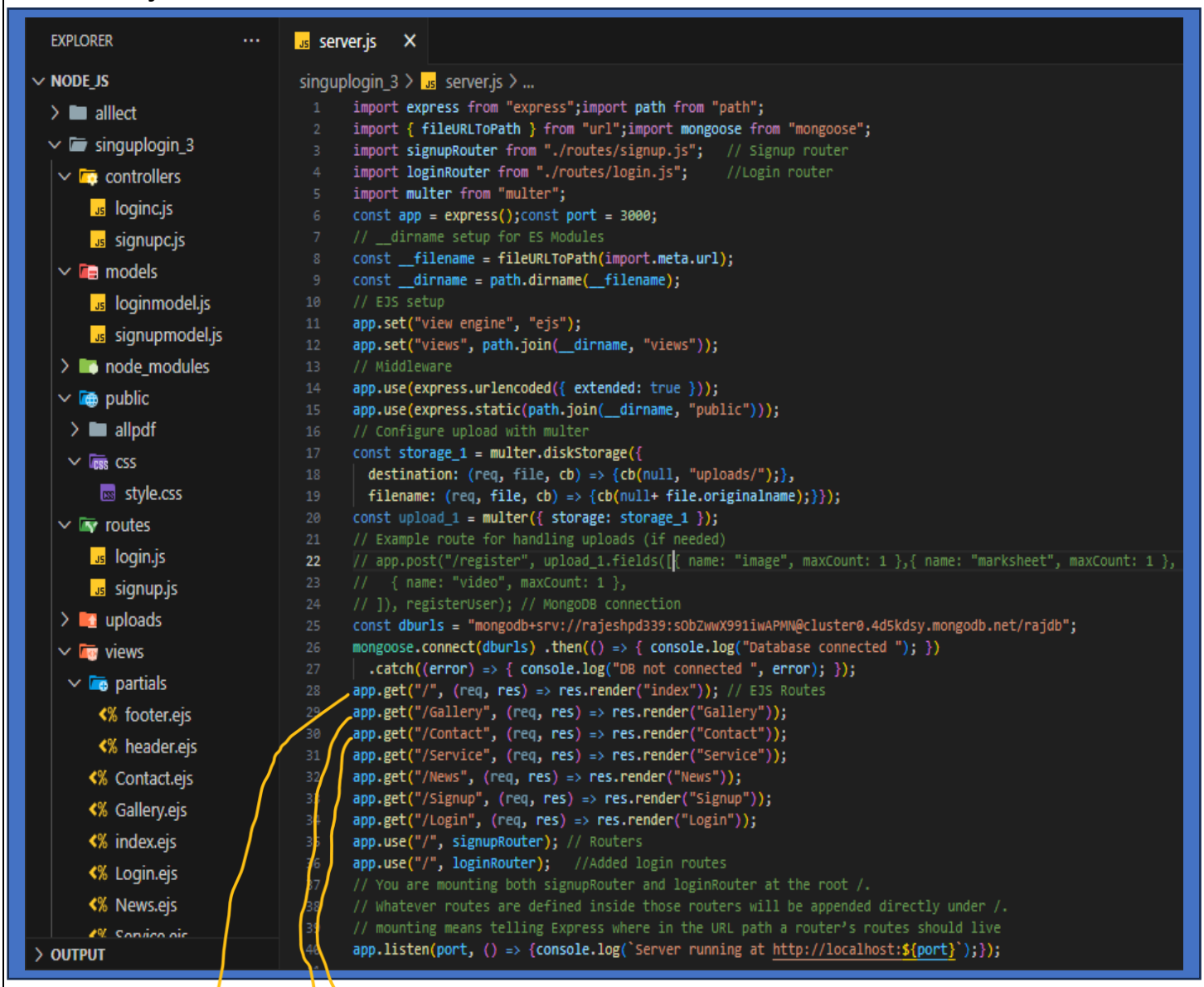

```
project/
├── models/
│   └── signupmodel.js
│
├── controllers/
│   └── signupc.js
│
├── routes/
│   └── signup.js
│
├── views/
│   ├── index.ejs
│   ├── Signup.ejs
│   ├── Login.ejs
│   └── partials/
│       ├── header.ejs
│       └── footer.ejs
│
├── public/
│   └── css/
│       └── style.css
│
└── server.js
```

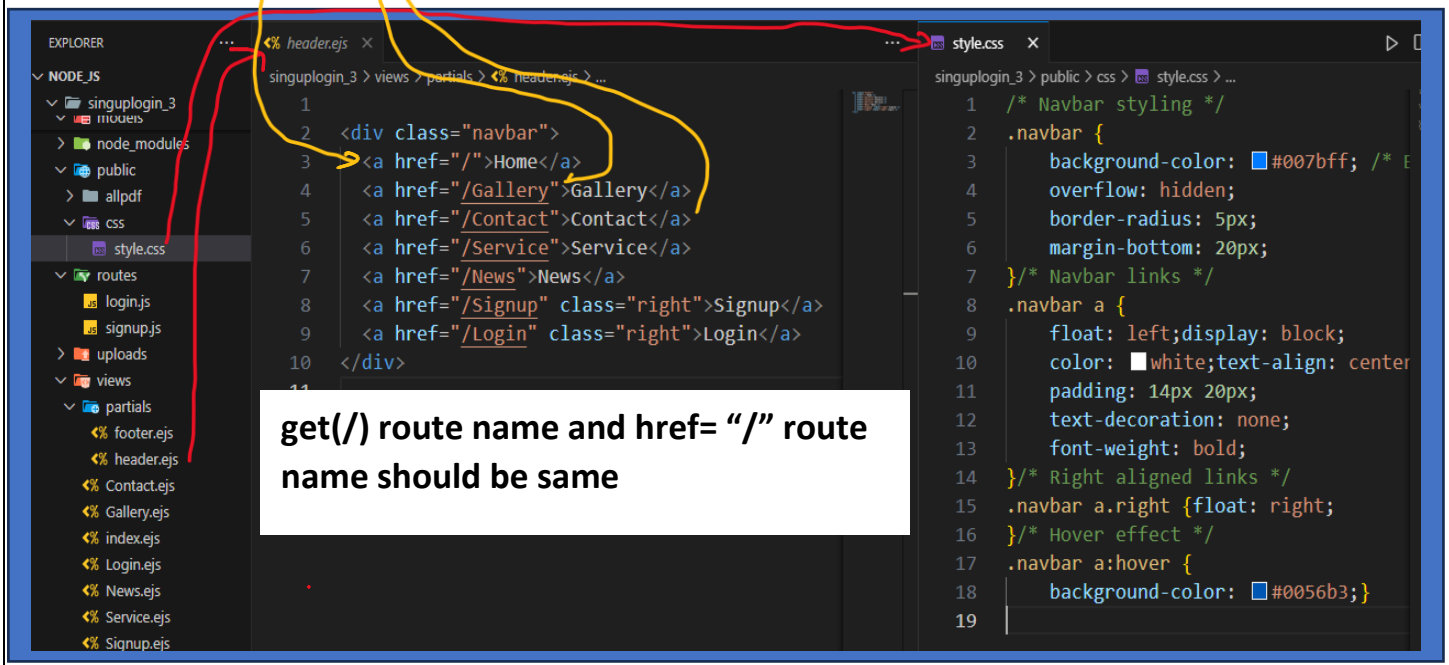


Install→ npm install express ejs mysql2 mongoose multer body-parser path

	Package	Purpose
•	**express**	→ Backend framework for routes and server.
•	**ejs**	→ Template engine for frontend rendering.
•	**mysql2**	→ Connect & query MySQL database.
•	**Mongoose**	→ Connect & manage MongoDB database.
•	**Multer**	→ Handle file uploads (images, PDFs, videos).
•	**body-parser**	→ Parse POST form data (req.body).

Server.js


EXPLORER
server.js
NODE_JS
alllect
singuplogin_3
controllers
loginc.js
signupc.js
models
loginmodel.js
signupmodel.js
node_modules
public
allpdf
css
style.css
routes
login.js
signup.js
uploads
views
partials
footer.ejs
header.ejs
Contact.ejs
Gallery.ejs
index.ejs
Login.ejs
News.ejs
OUTPUT
singuplogin_3 > server.js > ...
import express from "express";import path from "path";
import { fileURLToPath } from "url";import mongoose from "mongoose";
import signupRouter from "./routes/signup.js"; // Signup router
import loginRouter from "./routes/login.js"; //Login router
import multer from "multer";
const app = express();const port = 3000;
// __dirname setup for ES Modules
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);
// EJS setup
app.set("view engine", "ejs");
app.set("views", path.join(__dirname, "views"));
// Middleware
app.use(express.urlencoded({ extended: true }));
app.use(express.static(path.join(__dirname, "public")));
// Configure upload with multer
const storage_1 = multer.diskStorage({
destination: (req, file, cb) => {cb(null, "uploads/");},
filename: (req, file, cb) => {cb(null+ file.originalname);}});
const upload_1 = multer({ storage: storage_1 });
// Example route for handling uploads (if needed)
// app.post("/register", upload_1.fields([{ name: "image", maxCount: 1 },{ name: "marksheet", maxCount: 1 },
// { name: "video", maxCount: 1 },
//]), registerUser); // MongoDB connection
const dburls = "mongodb+srv://rajeshpd339:sObZwwX991iwAPMN@cluster0.4d5kdsy.mongodb.net/rajdb";
mongoose.connect(dburls) .then(() => { console.log("Database connected "); })
.catch((error) => { console.log("DB not connected ", error); });
app.get("/", (req, res) => res.render("index")); // EJS Routes
app.get("/Gallery", (req, res) => res.render("Gallery"));
app.get("/Contact", (req, res) => res.render("Contact"));
app.get("/Service", (req, res) => res.render("Service"));
app.get("/News", (req, res) => res.render("News"));
app.get("/Signup", (req, res) => res.render("Signup"));
app.get("/Login", (req, res) => res.render("Login"));
app.use("/", signupRouter); // Routers
app.use("/", loginRouter); //Added login routes
// You are mounting both signupRouter and loginRouter at the root /.
// Whatever routes are defined inside those routers will be appended directly under /.
// mounting means telling Express where in the URL path a router's routes should live
app.listen(port, () => {console.log(`Server running at http://localhost:${port}`);});


EXPLORER
header.ejs
style.css
NODE_JS
singuplogin_3 > views > partials > header.ejs > ...
singuplogin_3 > public > css > style.css > ...

Home
Gallery
Contact
Service
News
Signup
Login

/* Navbar styling */
.navbar {
background-color: #007bff;
overflow: hidden;
border-radius: 5px;
margin-bottom: 20px;
}/* Navbar links */
.navbar a {
float: left;display: block;
color: white;text-align: center
padding: 14px 20px;
text-decoration: none;
font-weight: bold;
}/* Right aligned links */
.navbar a.right {float: right;
}/* Hover effect */
.navbar a:hover {
background-color: #0056b3;}
get(/) route name and href= "/" route name should be same

views (Login.ejs and Singup.ejs)

singuplogin_3 > views > Login.ejs > html

```ejs
<!-- views/Login.ejs -->
<!DOCTYPE html>
<html>
<head>
  <title>Login</title>
  <link rel="stylesheet" href="css/style.css">
</head>
<body>
  <%- include('partials/header') %>
  <h2 id="dd1">Login Form</h2>
  <form action="/req-login" method="post">
    Email: <input type="email" name="email" required><br><br>
    Password: <input type="password" name="password"><br><br>
    <button type="submit">Login</button>
  </form>
  <p>Don't have an account? <a href="/Signup">Sign up here</a>
  <%- include('partials/footer') %>
</body>
</html>
```

singuplogin_3 > views > Signup.ejs > ...

```ejs
<!-- views/Signup.ejs -->
<!DOCTYPE html>
<html>
<head>
  <title>Signup</title>
    <link rel="stylesheet" href="css/style.css">
</head>
<body>
  <%- include('partials/header') %>
  <h2 id="dd1">Signup Form</h2>
  <form action="/req-signup" method="post">
    Name:<input type="text" name="name" required><br><br>
    Email:<input type="email" name="email" required><br><br>
    Mobile No: <input type="text" name="mobile" required><br><
    Password:<input type="password" name="password" required><
    Confirm Password: <input type="password" name="confirm" re
    <button type="submit">Sign Up</button>
    <p>have an account? <a href="/Login">Login</a></p>
  </form>
  <%- include('partials/footer') %>
</body></html>
```

Signupmodel.js(models) and singupc.js(controllers)

EXPLORER

NODE_JS
- allect
- singuplogin_3
  - controllers
    - logincjs
    - signupcjs
  - models
    - loginmodel.js
    - signupmodel.js
  - node_modules
  - public
    - allpdf
    - css
      - style.css
  - routes
    - login.js
    - signup.js
  - uploads
  - views
    - partials

singuplogin_3 > models > signupmodel.js > ...

```ejs
import mongoose from "mongoose";
// Define schema
const signupSchema = new mongoose.
Schema({
  name: String,
  email: String,
  mobile: String,
  password: String,
  confirm: String,
  createdAt: { type: Date, default:
    Date.now }
});
// Create collection (model)
const SignupUser = mongoose.model
("SignupUser", signupSchema);
export default SignupUser;
```

singuplogin_3 > controllers > signupc.js > ...

```ejs
import SignupUser from "../models/signupmodel.js";  // ✅ spelling co
export const registerUser = async (req, res) => {
  try {
    const { name, email, mobile, password, confirm } = req.body;
    if (password !== confirm) {
      return res.status(400).send("Passwords do not match!");}
    const existingUser = await SignupUser.findOne({ email });
    if (existingUser) {
      return res.status(400).send("Email already registered!");}
    const newUser = new SignupUser({ name, email, mobile, password });
    await newUser.save();
    res.status(201).send("Signup successful! Please login.");
    // res.redirect("/Login");
  } catch (error) {
    console.error("Error during signup:", error);
    res.status(500).send("Internal Server Error");
  }};
```

singuplogin_3 > controllers > loginc.js > loginUser

```ejs
import SignupUser from "../models/signupmodel.js";
export const loginUser = async (req, res) => {
  try {
    const { email, password} = req.body;
    // Step 1: Find user by email
    const user = await SignupUser.findOne({ email });
    if (!user) {return res.send("User not found.");}
    // Step 2: Compare plain text passwords
    if (user.password !==password) {return res.send("Invalid pwd.")}
    // Step 3: If everything matches → redirect
    console.log("Login Successful");
    res.redirect("/");    // Redirect to homepage (index.ejs)
  } catch (error) {
    console.error("Login Error:", error);
    res.send("Something went wrong while logging in.");}};
```

singuplogin_3 > routes > login.js > ...

```ejs
import express from "express";
import { loginUser } from
"../controllers/loginc.js";

const router = express.Router();
// POST route for login form
router.post("/req-login",
    loginUser);

export default router;
```

Route (Login.js and singup.js)

singuplogin_3 > routes > login.js > ...

```ejs
import express from "express";
import { loginUser } from "../controllers/loginc.js";

const router = express.Router();
// POST route for login form
router.post("/req-login", loginUser);

export default router;
```

singuplogin_3 > routes > signup.js > ...

```ejs
import express from "express";
import { registerUser } from "../controllers/signupc.js";  // spelli
const router = express.Router();
// POST request for signup
router.post("/req-signup", registerUser);
export default router;
// import signupRouter from "./routes/signup.js";
// import loginRouter from "./routes/login.js";
// Now signupRouter is just the exported router object from signup.j
// Same with loginRouter.
```

How you can fix your signup and login logic using bcrypt

1. Install bcrypt→ npm install bcrypt 2. Update your signupmodel.js

singuplogin_3 > models > signupmodel.js > signupSchema.pre("save") callback

```
import mongoose from "mongoose";
import bcrypt from "bcrypt";
// Define schema
const signupSchema = new mongoose.Schema({
  name: String,
  email: { type: String, unique: true },
  mobile: String,
  password: String,
  confirm: String,
  createdAt: { type: Date, default: Date.now }
});
// Middleware → Hash password before saving
signupSchema.pre("save", async function (next) {
  if (!this.isModified("password")) return next();
  try {
    const salt = await bcrypt.genSalt(10); // generate salt
    this.password = await bcrypt.hash(this.password, salt); // hash password
    this.confirm = undefined; // remove confirm field (not needed in DB)
    next();
  } catch (err) {
    next(err);
  }});
// Create collection (model)
const SignupUser = mongoose.model("SignupUser", signupSchema);
export default SignupUser;
```

1. Update your loginUser Controller

signupmodel.js | loginc.js

singuplogin_3 > controllers > loginc.js > ...

```
import SignupUser from "../models/signupmodel.js";
import bcrypt from "bcrypt";
export const loginUser = async (req, res) => {
  try {
    const { email, password } = req.body;
    // Step 1: Find user by email
    const user = await SignupUser.findOne({ email });
    if (!user) {
      return res.send("User not found.");
    }
    // Step 2: Compare hashed password
    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) {
      return res.send("Invalid password.");
    }
    // Step 3: If everything matches → login successful
    console.log("Login Successful");
    res.redirect("/"); // redirect to homepage
  } catch (error) {
    console.error("Login Error:", error);
    res.send("Something went wrong while logging in.");
  }
};
```

How you can fix your **signup** and **login** logic using **bcrypt**:

1. Install bcrypt → npm install bcrypt

2. Update your signupmodel.js

Example:

```
import mongoose from "mongoose";
import bcrypt from "bcrypt";
// Define schema
const signupSchema = new mongoose.Schema({
  name: String,
  email: { type: String, unique: true },
  mobile: String,
  password: String,
  confirm: String,
  createdAt: { type: Date, default: Date.now }});
// Middleware → Hash password before saving
signupSchema.pre("save", async function (next) {
  if (!this.isModified("password")) return next();
  try {
    const salt = await bcrypt.genSalt(10); // generate salt
    this.password = await bcrypt.hash(this.password, salt); // hash password
    this.confirm = undefined; // remove confirm field (not needed in DB)
    next();
  } catch (err) {
    next(err);
  }});
// Create collection (model)
const SignupUser = mongoose.model("SignupUser", signupSchema);
export default SignupUser;
```

3. Update your loginUser Controller

```
import SignupUser from "../models/signupmodel.js";
import bcrypt from "bcrypt";
export const loginUser = async (req, res) => {
  try {
    const { email, password } = req.body;
    // Step 1: Find user by email
    const user = await SignupUser.findOne({ email });
    if (!user) { return res.send("User not found."); }
    // Step 2: Compare hashed password
    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) {return res.send("Invalid password.");}
    // Step 3: If everything matches → login successful
    console.log("Login Successful");
    res.redirect("/"); // redirect to homepage
  } catch (error) {
    console.error("Login Error:", error);
    res.send("Something went wrong while logging in.");
  }};
```

How to connect a Node.js/Express app with local MongoDB

Step 1: Install MongoDB locally

- **Windows:** Download from MongoDB Community and install.
- **Mac:** brew install mongodb-community
- **Linux:** sudo apt install mongodb (or use official docs).

Step 2: Start MongoDB server

```
# Windows (from Command Prompt or Services)
mongod
# Mac/Linux → sudo systemctl start mongod
# or
mongod
MongoDB default connection: mongodb://127.0.0.1:27017
```

Step 3: Install required Node.js packages

```
npm install express mongoose
```

- express → web server
- mongoose → MongoDB ODM

Step 4: Import packages and connect

```
import express from "express";
import mongoose from "mongoose";
const app = express();
const port = 3000;
// Local MongoDB URL
const dbURL = "mongodb://127.0.0.1:27017/databaseName";
// Connect to MongoDB
mongoose.connect(dbURL, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log("MongoDB connected"))
  .catch(err => console.error("MongoDB connection error:", err));
// Start Express server
app.listen(port, () => console.log(`Server running at http://localhost:${port}`));
```

Step 5: Optional - Define a Mongoose Schema & Model

```
import mongoose from "mongoose";
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  password: String
});
const User = mongoose.model("User", userSchema);
export default User;
```

Step 6: Use the model in routes

Example: Create new user

```
app.post("/signup", async (req, res) => {
  const user = new User(req.body);
  await user.save();
  res.send("User created");
});
```

Summary

1. Install MongoDB locally.
2. Start MongoDB server.
3. Install mongoose in Node.js.
4. Connect using mongoose.connect("mongodb://127.0.0.1:27017/dbname").
5. Define schema & model.

Use model to create/read/update/delete documents.

MongoDB local connection guide with Node.js

Folder Structure

```
myapp/
├── server.js        # Main Express server
├── models/
│   └── user.js      # Mongoose schema & model
├── routes/
│   └── auth.js      # Signup/Login routes
├── package.json
└── node_modules/
```

Step 1: Install MongoDB locally

- **Windows:** Download from MongoDB Community and install.
- **Mac:** brew install mongodb-community
- **Linux:** sudo apt install mongodb

Step 2: Start MongoDB server

```
# Windows → mongod
# Mac/Linux→ sudo systemctl start mongod
# or mongod → MongoDB default connection: mongodb://127.0.0.1:27017
```

Step 3: Install required Node.js packages

```
npm install express mongoose
```

express → web server **mongoose** → MongoDB ODM Optional dev tool: npm install --save-dev nodemon

Step 4: Create server.js

```
import express from "express";
import mongoose from "mongoose";
import authRoutes from "./routes/auth.js";
const app = express();
const port = 3000;
// Middleware
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
// Local MongoDB URL
const dbURL = "mongodb://127.0.0.1:27017/mydatabase";
// Connect to MongoDB
mongoose.connect(dbURL, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log("MongoDB connected"))
  .catch(err => console.error("MongoDB connection error:", err));
// Routes
app.use("/", authRoutes);
app.listen(port, () => console.log(`Server running at http://localhost:${port}`));
```

Step 5: Create models/user.js

```
import mongoose from "mongoose";
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  password: String
});
const User = mongoose.model("User", userSchema);
export default User;
```

Step 6: Create routes/auth.js

```
import express from "express";
import User from "../models/user.js";
const router = express.Router();
// Signup route
router.post("/signup", async (req, res) => {
 try {
  const user = new User(req.body);
  await user.save();
  res.send("User created");
 } catch (err) {
  res.status(500).send("Error: " + err.message);
 }
});
// Login route (simple example)
router.post("/login", async (req, res) => {
 const { email, password } = req.body;
 const user = await User.findOne({ email });
 if (!user) return res.status(404).send("User not found");
 if (user.password !== password) return res.status(401).send("Invalid password");
 res.send(`Welcome ${user.name}`);
});
export default router;
```

How to run

nodemon server.js

- Signup: POST /signup with JSON { "name": "John", "email": "john@example.com", "password": "1234" }
- Login: POST /login with JSON { "email": "john@example.com", "password": "1234" }

MySQL DB connection with Node JS/Express JS

Node.js + Express + MySQL project setup

Step-1: Install required packages

➔ npm init -y

➔ npm install express ejs body-parser mysql2 bcrypt

Step-2: MySQL table

➔ Login to MySQL and create a database + table:

```
CREATE DATABASE testdb;
USE testdb;
CREATE TABLE users (
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(100),
  email VARCHAR(100) UNIQUE,
  password VARCHAR(255)
);
```

Project structure + working with MySQL

Step-3: Project Structure

```
myapp/
├── server.js
├── package.json
├── views/
│   ├── signup.ejs
│   └── login.ejs
├── node_modules/
└── public/        (optional: CSS, images, JS)
```

server.js

```
import express from "express";
import bodyParser from "body-parser";
import mysql from "mysql2";
import bcrypt from "bcrypt";
import path from "path";
import { fileURLToPath } from "url";
const app = express();
const port = 3000;
// Setup __dirname for ES modules
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);
// Middleware
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(express.static(path.join(__dirname, "public")));
// Set EJS as view engine
app.set("view engine", "ejs");
app.set("views", path.join(__dirname, "views"));

// DB Connection
```

How to Create a Database

- **Syntax:** CREATE DATABASE database_name;
- **Example:** mysql> CREATE DATABASE riddb;

How to Show All Databases

- **Syntax:** SHOW databases;
- **Example:** mysql> SHOW databases

How to Change a Database

- **Syntax:** USE database_name;
- **Example:** mysql> USE riddb;

How to Check Current DB

- **Syntax:** SELECT DATABASE();
- **Example:** mysql> SELECT DATABASE();

How to Delete database

- **Syntax:** DROP DATABASE database_name;
- **Example:** mysql> DROP DATABASE riddb;

How to switch from one database to another,

- **Syntax:** USE new_database_name;
- **Example:** mysql> USE riddb2;

How to Check if a Database Exist

- **Syntax:** SHOW DATABASES LIKE 'database_name';
- **Example:** mysql> SHOW DATABASES LIKE 'rid_db2';

How to Show Database Information (Schema)

- **Syntax:** SHOW TABLE STATUS FROM database_name;
- **Example:** mysql> SHOW TABLE STATUS FROM rid_db;

How to Show Database Users

- **Ex:** mysql> SELECT User, Host FROM mysql.user;

Table query in MySQL

How to create tables

CREATE TABLE table_name(column1 datatype constraints)

How to show all tables :

➢ SHOW TABLES ;

How to describe the table structure

➢ DESCRIBE table_name ;

How to delete the table

➢ DESCRIBE table_name;

How to show the all data from a table

➢ SELECT * FROM table_name;

```
const db = mysql.createConnection({
  host: "localhost",
  user: "root",       // your mysql username
  password: "raj12345", // your mysql password
  database: "testdb"
});
db.connect((err) => {
  if (err) { console.error("DB connection failed:", err);
  } else {  console.log("MySQL Connected!"); } });
// ---------- Routes ----------
// Show signup form
app.get("/signup", (req, res) => {
  res.render("signup");});
// Handle signup form
app.post("/signup", async (req, res) => {
  const { name, email, password } = req.body;
  const hashed = await bcrypt.hash(password, 10);
  const sql = "INSERT INTO users (name, email, password) VALUES (?, ?, ?)";
  db.query(sql, [name, email, hashed], (err) => {
    if (err) return res.status(500).send("Error: " + err.message);
    res.send("Signup successful!");
  });
});
// Show login form
app.get("/login", (req, res) => {
  res.render("login");
});
// Handle login form
app.post("/login", (req, res) => {
  const { email, password } = req.body;
  const sql = "SELECT * FROM users WHERE email = ?";
  db.query(sql, [email], async (err, results) => {
    if (err) return res.status(500).send("Error: " + err.message);
    if (results.length === 0) return res.status(404).send("User not found");
    const user = results[0];
    const match = await bcrypt.compare(password, user.password);

    if (!match) return res.status(401).send(" Invalid password");
    res.send(`Welcome ${user.name}, Login successful!`);
  });});
// Start server app.listen(port, () => { console.log(`Server running at http://localhost:${port}`); });
```

views/signup.ejs

```
<!DOCTYPE html>
<html>
<head>
  <title>Signup</title>
</head>
<body>
```

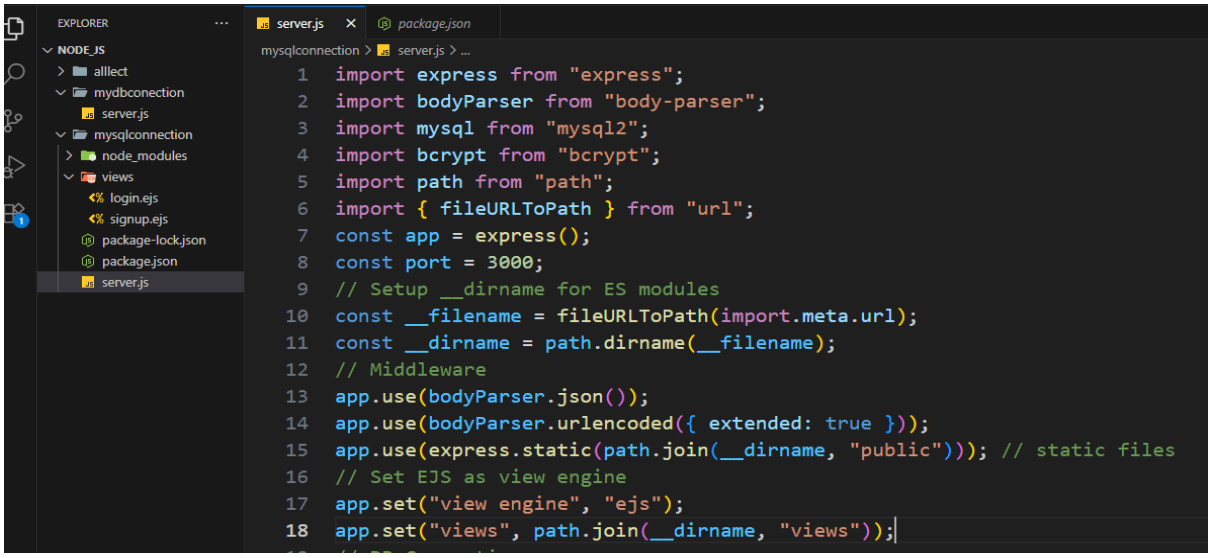
```
 <h1>User Signup</h1>
 <form action="/signup" method="POST">
  <label>Name:</label>
  <input type="text" name="name" required><br><br>
  <label>Email:</label>
  <input type="email" name="email" required><br><br>
  <label>Password:</label>
  <input type="password" name="password" required><br><br>
  <button type="submit">Signup</button>
 </form>
 <p>Already have an account? <a href="/login">Login</a></p>
</body>
</html>
```

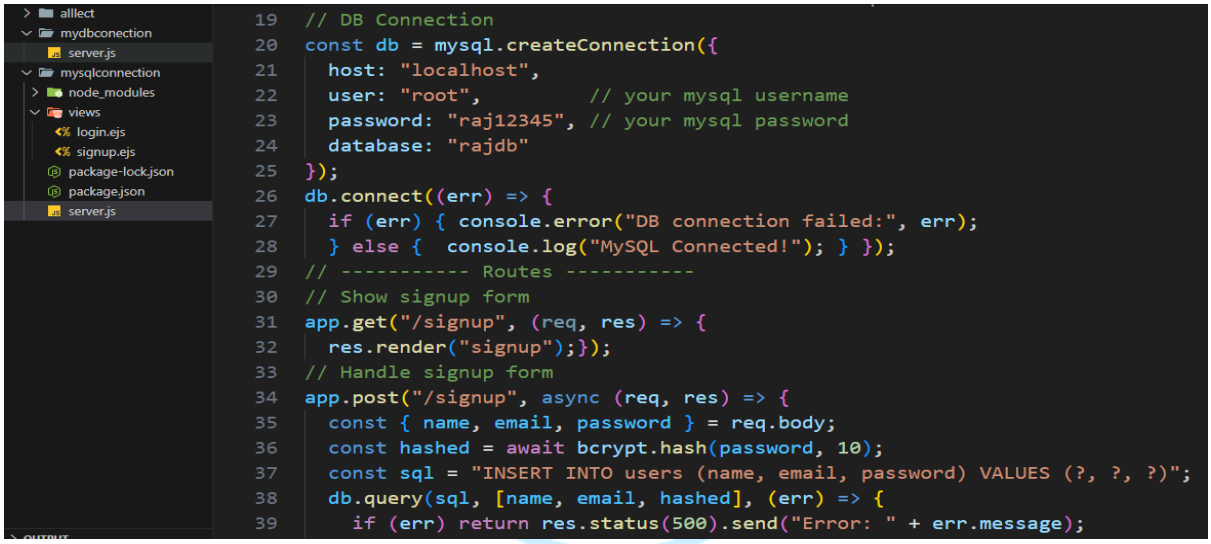
views/login.ejs

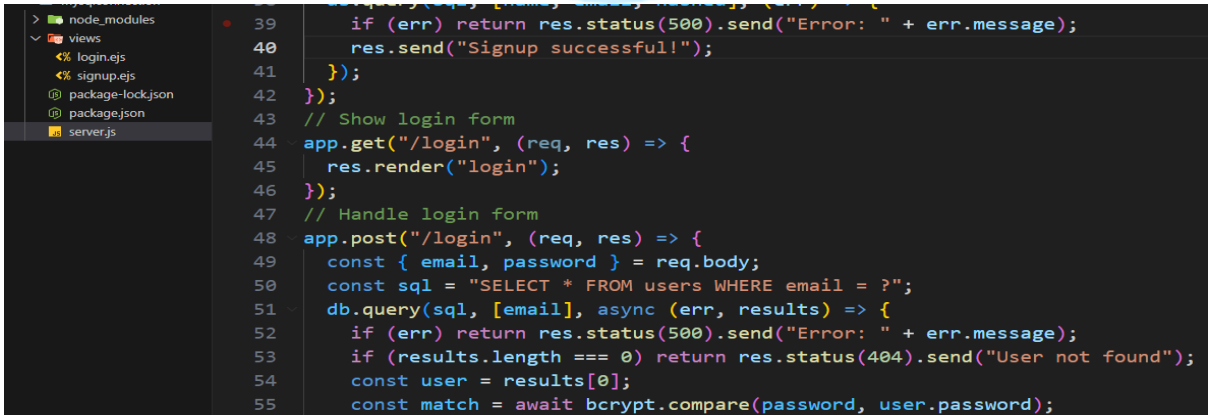
```
<!DOCTYPE html>
<html>
<head>
 <title>Login</title>
</head>
<body>
 <h1>User Login</h1>
 <form action="/login" method="POST">
  <label>Email:</label>
  <input type="email" name="email" required><br><br>
  <label>Password:</label>
  <input type="password" name="password" required><br><br>
  <button type="submit">Login</button>
 </form>
 <p>Don't have an account? <a href="/signup">Signup</a></p>
</body>
</html>
```

Now you can visit:

- http://localhost:3000/signup → register user
- http://localhost:3000/login → login user


EXPLORER
server.js
package.json
NODE_JS
allllect
mydbconection
server.js
mysqlconnection
node_modules
views
login.ejs
signup.ejs
package-lock.json
package.json
server.js
mysqlconnection > server.js > ...
1 import express from "express";
2 import bodyParser from "body-parser";
3 import mysql from "mysql2";
4 import bcrypt from "bcrypt";
5 import path from "path";
6 import { fileURLToPath } from "url";
7 const app = express();
8 const port = 3000;
9 // Setup __dirname for ES modules
10 const __filename = fileURLToPath(import.meta.url);
11 const __dirname = path.dirname(__filename);
12 // Middleware
13 app.use(bodyParser.json());
14 app.use(bodyParser.urlencoded({ extended: true }));
15 app.use(express.static(path.join(__dirname, "public"))); // static files
16 // Set EJS as view engine
17 app.set("view engine", "ejs");
18 app.set("views", path.join(__dirname, "views"));


19 // DB Connection
20 const db = mysql.createConnection({
21 host: "localhost",
22 user: "root", // your mysql username
23 password: "raj12345", // your mysql password
24 database: "rajdb"
25 });
26 db.connect((err) => {
27 if (err) { console.error("DB connection failed:", err);
28 } else { console.log("MySQL Connected!"); } });
29 // ----------- Routes -----------
30 // Show signup form
31 app.get("/signup", (req, res) => {
32 res.render("signup");});
33 // Handle signup form
34 app.post("/signup", async (req, res) => {
35 const { name, email, password } = req.body;
36 const hashed = await bcrypt.hash(password, 10);
37 const sql = "INSERT INTO users (name, email, password) VALUES (?, ?, ?)";
38 db.query(sql, [name, email, hashed], (err) => {
39 if (err) return res.status(500).send("Error: " + err.message);


39 if (err) return res.status(500).send("Error: " + err.message);
40 res.send("Signup successful!");
41 });
42 });
43 // Show login form
44 app.get("/login", (req, res) => {
45 res.render("login");
46 });
47 // Handle login form
48 app.post("/login", (req, res) => {
49 const { email, password } = req.body;
50 const sql = "SELECT * FROM users WHERE email = ?";
51 db.query(sql, [email], async (err, results) => {
52 if (err) return res.status(500).send("Error: " + err.message);
53 if (results.length === 0) return res.status(404).send("User not found");
54 const user = results[0];
55 const match = await bcrypt.compare(password, user.password);


55 const match = await bcrypt.compare(password, user.password);
56 if (!match) return res.status(401).send(" Invalid password");
57 res.send(`Welcome ${user.name}, Login successful!`);
58 });});
59 // Start server
60 app.listen(port, () => { console.log(`Server running at http://localhost:${port}

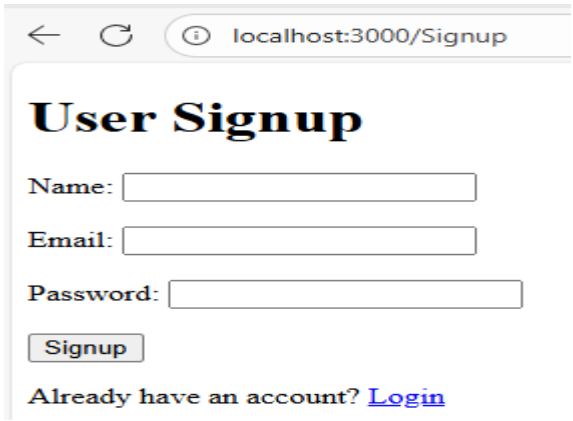
```
mysql> CREATE DATABASE rajdb;
Query OK, 1 row affected (0.12 sec)

mysql> use rajdb;
Database changed
mysql> CREATE TABLE users (
    ->   id INT PRIMARY KEY AUTO_INCREMENT,
    ->   name VARCHAR(100),
    ->   email VARCHAR(100) UNIQUE,
    ->   password VARCHAR(255)
    -> );
Query OK, 0 rows affected (0.29 sec)

mysql> SHOW TABLES;
+----------------+
| Tables_in_rajdb |
+----------------+
| users          |
+----------------+
1 row in set (0.24 sec)

mysql> SELECT * FROM users;
+----+------+--------------+--------------------------------------------------------------+
| id | name | email        | password                                                     |
+----+------+--------------+--------------------------------------------------------------+
|  1 | raj  | raj@gmail.com | $2b$10$USHpWJClKvujkGoY8F6Ol.Vf18VlWEzf/0wPLLYlT.sEY43kq03/G |
+----+------+--------------+--------------------------------------------------------------+
1 row in set (0.00 sec)
```

signup.ejs

```
<!DOCTYPE html>
<html>
<head>
  <title>Signup</title>
</head>
<body>
  <h1>User Signup</h1>
  <form action="/signup" method="POST">
    <label>Name:</label>
    <input type="text" name="name" required><br><br>
    <label>Email:</label>
    <input type="email" name="email" required><br><br>
    <label>Password:</label>
    <input type="password" name="password" required><br><
    <button type="submit">Signup</button>
  </form>
  <p>Already have an account? <a href="/login">Login</a><
</body>
</html>
```

login.ejs

```
<!DOCTYPE html>
<html>
<head>
  <title>Login</title>
</head>
<body>
  <h1>User Login</h1>
  <form action="/login" method="POST">
    <label>Email:</label>
    <input type="email" name="email" required><br><br
    <label>Password:</label>
    <input type="password" name="password" required>
    <button type="submit">Login</button>
  </form>
  <p>Don't have an account? <a href="/signup">Signup
</body>
</html>
```



Here's a **short explanation** of your code:

1. **Imports & Setup**
   - express → Web framework for routing.
   - body-parser → Parses form/JSON data.
   - mysql2 → Connects to MySQL database.
   - bcrypt → Hashes passwords for security.
   - path & fileURLToPath → Handle file paths in ES modules.
2. **Server Setup**
   - app = express() → Initialize Express app.
   - port = 3000 → Server will run on port 3000.
   - __dirname setup → For ES module compatibility.
3. **Middleware**
   - bodyParser.json() & urlencoded() → Parse incoming JSON and form data.
   - express.static() → Serve static files from public folder.
   - app.set("view engine", "ejs") → Use EJS templates.
4. **Database Connection**
   - Connects to MySQL rajdb with root user and password.
   - Logs success or error on connection.
5. **Routes**
   - GET /signup → Shows signup form.
   - POST /signup → Saves user with hashed password in users table.
   - GET /login → Shows login form.
   - POST /login → Checks user email, compares password with hashed password, and responds accordingly.
6. **Start Server**
   - app.listen(port) → Runs server at http://localhost:3000.

💡 **Summary:**

This is a **basic Express + MySQL + EJS app** that allows **user signup and login** with **password hashing**.
If you want, I can also make a **tiny flow diagram** of how signup/login works—it'll be easier to visualize. Do you want me to do that?
Here's a clear explanation of your code:

Code Breakdown

```
app.post("/signup", async (req, res) => {
  const { name, email, password } = req.body; // 1
  const hashed = await bcrypt.hash(password, 10); // 2
  const sql = "INSERT INTO users (name, email, password) VALUES (?, ?, ?)"; // 3
  db.query(sql, [name, email, hashed], (err) => { // 4
    if (err) return res.status(500).send("Error: " + err.message); // 5
    res.send("Signup successful!"); // 6
  });
});
```

Step-by-Step Explanation

1. **Extract form data**

```
const { name, email, password } = req.body;
```

- Takes name, email, password from the signup form sent by the client.

2. **Hash password**

```
const hashed = await bcrypt.hash(password, 10);
```

- Uses bcrypt to securely hash the password before saving it.
- 10 → Number of salt rounds (strength of hashing).

3. **SQL query with ? placeholders**

```
const sql = "INSERT INTO users (name, email, password) VALUES (?, ?, ?)";
```

- ? are **placeholders** for values.
- They prevent **SQL injection attacks** (safer than directly inserting variables into the query).

4. **Execute query**

```
db.query(sql, [name, email, hashed], (err) => { ... });
```

- [name, email, hashed] → These values replace the ? in the SQL query **in order**.
- db.query runs the SQL command in the database.

5. **Error handling**

```
if (err) return res.status(500).send("Error: " + err.message);
```

- Sends an error response if the database insertion fails.

6. **Success response**

```
res.send("Signup successful!");
```

- Sends a confirmation message to the user.

Key point about ? (placeholders)

- **Why use ??**
  - Protects against **SQL injection**.
  - Automatically escapes special characters in user input.
  - Cleaner and safer than concatenating strings manually:

```
// Unsafe way
const sql = "INSERT INTO users (name, email, password) VALUES ('" + name + "','" + email + "','" + hashed + "')";
```

Here's a **line-by-line explanation** of your login code:

```
// Handle login form
app.post("/login", (req, res) => {
```

- Creates a **POST route** for /login.
- This route will handle **form submissions** from the login page.
- req → Request from the client (contains form data).
- res → Response sent back to the client.

```
const { email, password } = req.body;
```

- Extracts email and password submitted by the user from the request body.

```
const sql = "SELECT * FROM users WHERE email = ?";
```

- Prepares a **SQL query** to find a user in the users table with the given email.
- ? is a **placeholder** to safely insert the email (prevents SQL injection).

```
db.query(sql, [email], async (err, results) => {
```

- Executes the SQL query.
- [email] replaces the ? placeholder in the SQL query.
- results contains the rows returned from the database.
- err will contain any database error.

```
if (err) return res.status(500).send("Error: " + err.message);
```

- Checks for **database errors**.
- If an error occurs, sends **HTTP 500** and the error message.

```
if (results.length === 0) return res.status(404).send("User not found");
```

- Checks if **no user** with that email exists.
- Sends **HTTP 404** if user is not found.

const user = results[0];

- If a user is found, take the **first row** (the only user with that email).

const match = await bcrypt.compare(password, user.password);

- Compares the **entered password** with the **hashed password** stored in the database using bcrypt.
- match will be true if passwords match, false otherwise.

if (!match) return res.status(401).send(" Invalid password");

- If password does **not match**, sends **HTTP 401 Unauthorized** with message.

res.send(`Welcome ${user.name}, Login successful!`);

- If password matches, sends a **success message** including the user's name.
-

Mysql connection with Node JS/Express JS

If you want to **switch from MongoDB to MySQL** and connect as user you need to use a MySQL driver for Node.js, such as **mysql2** or **sequelize**.

1. Install MySQL package

- In your project folder, run:
- npm install mysql2

2. Import and connect in server.js

- Replace your MongoDB code with MySQL connection:

```
import mysql from "mysql2";
// Create connection
const db = mysql.createConnection({
  host: "localhost",
  user: "root",
  password: "raj12345",
  database: "your_database_name"  // replace with your DB name
});// Connect
db.connect((err) => {
  if (err) {console.error("MySQL connection failed:", err);
  } else { console.log("MySQL Database connected");
  }});// Export db if needed in routers/controllers
export default db;
```

3. Using MySQL in your login/signup controllers

➔ Instead of Mongoose SignupUser.findOne(), you'll run **SQL queries**:

```
import db from "../server.js"; // import the MySQL connection
export const loginUser = (req, res) => {
  const { email, password } = req.body;
  const sql = "SELECT * FROM users WHERE email = ?";
  db.query(sql, [email], (err, results) => {
    if (err) return res.send("DB Error");
    if (results.length === 0) {  return res.send("User not found");}
    const user = results[0];
    if (user.password !== password) { return res.send("Invalid password");}
    console.log("Login successful");
    res.redirect("/"); // redirect to homepage
  });};
```

➔ users is your MySQL table where you store user data.

Notes MongoDB and Mongoose are **NoSQL**, MySQL is **SQL**, so you need to adjust all database queries accordingly.

- Create a table in MySQL, for example:

```
CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100),
  email VARCHAR(100) UNIQUE,
  mobile VARCHAR(15),
  password VARCHAR(100),
  createdAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Project with MySQL

1. **Install required packages** npm install express ejs mysql2 multer body-parser path
2. **Create MySQL database and table**

```
CREATE DATABASE rajdb;
USE rajdb;
CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  email VARCHAR(100) UNIQUE NOT NULL,
  mobile VARCHAR(15),
  password VARCHAR(100) NOT NULL,
  createdAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP);
```

3. **server.js (with MySQL, file upload, routes)**

```
import express from "express";
import path from "path";
import { fileURLToPath } from "url";
import mysql from "mysql2";
import multer from "multer";
import signupRouter from "./routes/signup.js";
import loginRouter from "./routes/login.js";
const app = express();
const port = 3000;
// __dirname setup
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);
// EJS setup
app.set("view engine", "ejs");
app.set("views", path.join(__dirname, "views"));
// Middleware
app.use(express.urlencoded({ extended: true }));
app.use(express.static(path.join(__dirname, "public")));
// MySQL connection
export const db = mysql.createConnection({
  host: "localhost",
  user: "root",
```

```
  password: "raj12345",
  database: "rajdb"
});
db.connect(err => {
  if (err) console.error("MySQL connection failed:", err);
  else console.log("MySQL Database connected");
});
// Multer setup
const storage_1 = multer.diskStorage({
  destination: (req, file, cb) => cb(null, "uploads/"),
  filename: (req, file, cb) => cb(null, Date.now() + "_" + file.originalname)
});
export const upload_1 = multer({ storage: storage_1 });

// Routes
        app.get("/", (req, res) => res.render("index"));
        app.get("/Gallery", (req, res) => res.render("Gallery"));
        app.get("/Contact", (req, res) => res.render("Contact"));
        app.get("/Service", (req, res) => res.render("Service"));
        app.get("/News", (req, res) => res.render("News"));
        app.get("/Signup", (req, res) => res.render("Signup"));
        app.get("/Login", (req, res) => res.render("Login"));
        app.use("/", signupRouter);
        app.use("/", loginRouter);
        // Start server app.listen(port, () => console.log(`Server running at http://localhost:${port}`));
```

4. controllers/signupc.js (Insert data into MySQL)

```
import { db } from "../server.js";
export const registerUser = (req, res) => {
  const { name, email, mobile, password } = req.body;
  const sql = "INSERT INTO users (name, email, mobile, password) VALUES (?, ?, ?, ?)";
  db.query(sql, [name, email, mobile, password], (err, result) => {
    if (err) { console.error("Signup Error:", err);
      return res.send("Error registering user. Maybe email already exists.");
    }
    console.log("User registered:", result.insertId);
    res.redirect("/Login"); // after signup, redirect to login
  });};
```

5. controllers/logincontroller.js (Check user + password)

```
import { db } from "../server.js";
export const loginUser = (req, res) => {
  const { email, password } = req.body;
  const sql = "SELECT * FROM users WHERE email = ?";
  db.query(sql, [email], (err, results) => {
    if (err) return res.send("DB Error");
    if (results.length === 0) return res.send("User not found");
    const user = results[0];
    if (user.password !== password) return res.send("Invalid password");
    console.log("Login successful");
```

```
    res.redirect("/"); // redirect to homepage
  });};
```

6. **routes/signup.js**

```
import express from "express";
import { registerUser } from "../controllers/signupc.js";
const router = express.Router();
router.post("/req-signup", registerUser);
export default router;
```

7. **routes/login.js**

```
import express from "express";
import { loginUser } from "../controllers/logincontroller.js";
const router = express.Router();
router.post("/req-login", loginUser);
export default router;
```

Note: With this setup, your **signup/login flow works with MySQL**.

- Form data → req.body → saved in MySQL table users
- Login checks email/password from MySQL → redirects on success

What is API in node js/ Express js

❖ **What is an API?**

- **API = Application Programming Interface**
- Works like a **restaurant menu**:
  - Menu = API (tells what you can order).
  - Chef = Server (prepares the data).
  - Waiter = Response (brings the data).
- You don't care *how* it's cooked, you just get the result.

❖ **API in Node.js / Express.js**

- **Express.js** makes it easy to build APIs.
- APIs usually use **JSON** for data exchange.
- Clients (browser, mobile, other servers) talk to the API using **HTTP methods**:
  - GET → Read data
  - POST → Create data
  - PUT → Update data
  - DELETE → Remove data

2) Types of APIs In Node.js, common API types are:

1. **REST API** (Representational State Transfer)
   - Uses HTTP methods: GET, POST, PUT, DELETE.
   - Returns data usually in JSON.
   - Example: /api/books → get list of books.
2. **GraphQL API**
   - Uses queries instead of fixed routes.
   - Example: { books { id title } }.
3. **SOAP API** (less used in Node.js today)
   - XML-based.

Note: In practice, **REST API** is the most popular in Node.js apps.

3) Steps to build a REST API in Node.js + Express

Step 1: Create project

```
mkdir restful-api
cd restful-api
npm init -y
npm install express
npm install --save-dev nodemon
```

Step 2: Create index.js

```
import express from "express";
const app = express();
const port = 3000;
```

Middleware

```
app.use(express.json());
```

Demo "database"

```
let books = [
 { id: 1, title: "Python", author: "Author  RP sir" },
 { id: 2, title: "Node js", author: "Author RP sir" }
];
```

Routes, Root

```
app.get("/", (req, res) => {
```

```
  res.send("Welcome to Books REST API");
});
```

GET all books

```
app.get("/api/books", (req, res) => {
  res.json(books);
});
```

GET one book

```
app.get("/api/books/:id", (req, res) => {
  const id = Number(req.params.id);
  const book = books.find(b => b.id === id);
  if (!book) return res.status(404).json({ error: "Book not found" });
  res.json(book);
});
```

POST create book

```
app.post("/api/books", (req, res) => {
  const { title, author } = req.body;
  if (!title || !author) {
    return res.status(400).json({ error: "Title and author required" });
  }
  const id = books.length ? books[books.length - 1].id + 1 : 1;
  const newBook = { id, title, author };
  books.push(newBook);
  res.status(201).json(newBook);
});
```

PUT update book

```
app.put("/api/books/:id", (req, res) => {
  const id = Number(req.params.id);
  const index = books.findIndex(b => b.id === id);
  if (index === -1) return res.status(404).json({ error: "Book not found" });
  const { title, author } = req.body;
  if (!title || !author) {
    return res.status(400).json({ error: "Title and author required" });
  }
  books[index] = { id, title, author };
  res.json(books[index]);
});
```

DELETE book

```
app.delete("/api/books/:id", (req, res) => {
  const id = Number(req.params.id);
  const index = books.findIndex(b => b.id === id);
  if (index === -1) return res.status(404).json({ error: "Book not found" });

  const removed = books.splice(index, 1)[0];
  res.json({ message: "Deleted", book: removed });
});
```

Error handler

```
app.use((err, req, res, next) => {
  console.error(err.stack);
```

```
  res.status(500).json({ error: "Server error" });
});
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

Step 3: Run the server

npm run dev

Output: Server running at http://localhost:3000

Step 4: Test API

1) GET all books

Browser or Postman:

GET http://localhost:3000/api/books

Output:

```
[
 { "id": 1, "title": "Book One", "author": "Author A" },
 { "id": 2, "title": "Book Two", "author": "Author B" }
]
```

2) GET one book

GET http://localhost:3000/api/books/1

Output:

{ "id": 1, "title": "Book One", "author": "Author A" }

3) POST new book

Request body (JSON):

{ "title": "New Book", "author": "Sangam" }

Output:

{ "id": 3, "title": "New Book", "author": "sangam" }

4) PUT update book

Request:

PUT http://localhost:3000/api/books/2

Body:

{ "title": "Updated Title", "author": "Updated Author" }

Output:

{ "id": 2, "title": "Updated Title", "author": "Updated Author" }

5) DELETE book

DELETE http://localhost:3000/api/books/1

Output:

```
{
 "message": "Deleted",
 "book": { "id": 1, "title": "Book One", "author": "Author A" }
}
```

Note: **API** in Node.js = interface for client apps (frontend, mobile, etc.) to interact with server.

- **Types of APIs**: REST, GraphQL, SOAP (REST is most common).
- **RESTful API** uses HTTP methods:
  - GET → Read
  - POST → Create
  - PUT → Update
  - DELETE → Remove
- **We built a Books API** step by step and tested it.

server.js

allnewproject > server.js > ...

```
1   import express from "express";
2   const app = express();
3   const port = 3000;
4   // Middleware
5   app.use(express.json());
6   // Demo "database"
7   let books = [
8     { id: 1, title: "Book One", author: "Author RP sir" },
9     { id: 2, title: "Book Two", author: "Author Sangam Kumar" }
10  ];// Routes// Root
11  app.get("/", (req, res) => {res.send("Welcome to Books REST API");});
12  // GET all books
13  app.get("/api/books", (req, res) => {res.json(books);});
14  // GET one book
15  app.get("/api/books/:id", (req, res) => {
16    const id = Number(req.params.id);
17    const book = books.find(b => b.id === id);
18    if (!book) return res.status(404).json({ error: "Book not found" });
19    res.json(book);
20  });
21  app.listen(port, () => {
22    console.log(`Server running at http://localhost:${port}`);
23  });
```

always use req.params to access :id from the URL

GET→ http://localhost:3000/api/books

GET→ http://localhost:3000/api/books/1

❖ **Why do we use params (req.params.id)?**

- :id in /api/books/:id is a **route parameter**.
- It means: "Take the id from the URL".

Example: If you go to **http://localhost:3000/api/books/1**

  - req.params.id = "1"
  - Code finds the book with id: 1 → returns **Book One**.

- If you go to **http://localhost:3000/api/books/5**
  - Book doesn't exist → returns **404 error (Book not found)**.

Note: - params let us **capture dynamic values from the URL** (like book ID) instead of making separate routes for each book.

server.js

allnewproject > server.js > ...

```
import express from "express";
const app = express();
const port = 3000;
// Middleware
app.use(express.json());
// Demo "database" (projects)
let projects = [
  { id: 1, title: "AI Chatbot", domain: "Artificial Intelligence", mentor: "Dr. RP. sir" },
  { id: 2, title: "Smart Irrigation System", domain: "IoT", mentor: "Prof. Neha Sharma" },
  { id: 3, title: "Blockchain Voting App", domain: "Blockchain", mentor: "Dr. R.P. Singh" }
];// Routes// Root
app.get("/", (req, res) => { res.send("Welcome to Engineering Projects REST API ");});
// GET all projects
app.get("/api/projects", (req, res) => {res.json(projects);});
// GET one project by id
app.get("/api/projects/:id", (req, res) => {
  const id = Number(req.params.id);
  const project = projects.find(p => p.id === id);
  if (!project) return res.status(404).json({ error: "Project not found" });
  res.json(project);
});
```

```
// Add a new project
app.post("/api/projects", (req, res) => {
  const { title, domain, mentor } = req.body;
  if (!title || !domain || !mentor) {
    return res.status(400).json({ error: "title, domain, and mentor are required" });}
  const id = projects.length ? projects[projects.length - 1].id + 1 : 1;
  const newProject = { id, title, domain, mentor };
  projects.push(newProject);
  res.status(201).json(newProject);});// Update a project
app.put("/api/projects/:id", (req, res) => {
  const id = Number(req.params.id);
  const index = projects.findIndex(p => p.id === id);
  if (index === -1) return res.status(404).json({ error: "Project not found" });
  const { title, domain, mentor } = req.body;
  if (!title || !domain || !mentor) {
    return res.status(400).json({ error: "title, domain, and mentor are required" });}
  projects[index] = { id, title, domain, mentor };
  res.json(projects[index]);});// Delete a project
app.delete("/api/projects/:id", (req, res) => {
  const id = Number(req.params.id);
  const index = projects.findIndex(p => p.id === id);
  if (index === -1) return res.status(404).json({ error: "Project not found" });
  const removed = projects.splice(index, 1)[0];
  res.json({ message: "Project deleted successfully", project: removed });});// Start server
app.listen(port, () => {console.log(`Server running at http://localhost:${port}`);});
```

```
let id;
if (projects.length === 0) {
  id = 1; } else {
  id = projects[projects.length - 1].id + 1;}
const newProject = { id, title, domain, mentor };
projects.push(newProject);
res.status(201).json(newProject);});
```

How to create full flex RESTful API in Node JS

Step 1: Create project

mkdir restful-api
cd restful-api
npm init -y
npm install express mongoose bcryptjs
npm install --save-dev nodemon

Step 2: Create index.js(Server)

import express from "express";
const app = express();
const port = 3000;

Default route app.get("/", (req, res) => {res.send("Hello, Express server is running!");});

Start server app.listen(port, () => { console.log(`Server is listening on http://localhost:${port}`);});

Step-3: Setup the MongoDB connection

index.js

```
api > index.js > ...
1   // index.js
2   import express from "express";
3   import mongoose from "mongoose";
4   // Creating server
5   const app = express();
6   const port = 3000;
7   // Database Connection
8   const dburls = "mongodb+srv://rajeshpd339:sObZwwX991iwAPMN@cluster0.4d5kdsy.mongodb.net/APIDB";
9   mongoose.connect(dburls)
10    .then(() => {console.log("Database connected ");})
11    .catch((error) => {console.log("DB not connected ", error);});
12  // Default route
13  app.get("/", (req, res) => {
14  res.send("Hello, Express server is running!");});
15  // Start server
16  app.listen(port, () => {console.log(`Server runing on http://localhost:${port}`);});
```

```
[nodemon] starting `node index.js`
Server is listening on http://localhost:3000
Database connected
```

Database Connection

const dburls="mongodb+srv://rajeshpd339:sObZwwX991iwAPMN@cluster0.4d5kdsy.mongodb.net/APIDB";

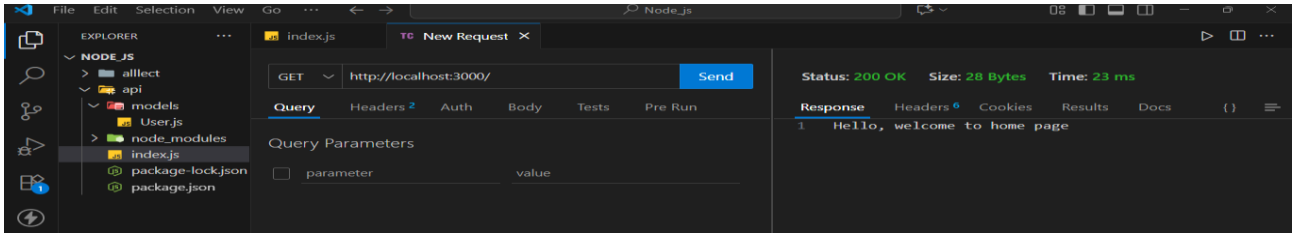
mongoose.connect(dburls)
.then(() => {console.log("Database connected ");})
.catch((error) => {console.log("DB not connected ", error);});

Note:

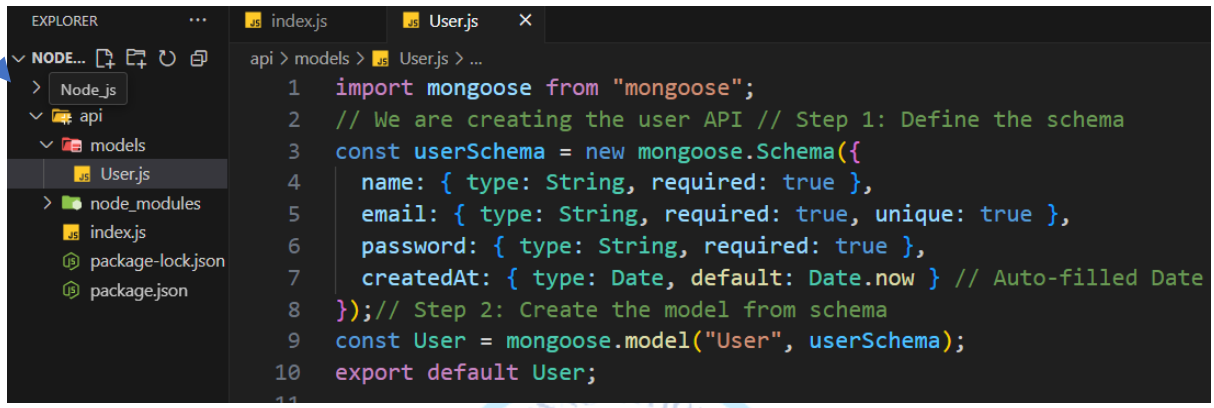
To test APIs without building a full frontend, you can use tools like **Postman or Thunder** Client.

- If you prefer working inside VS Code, install the Thunder Client extension from the Extensions Marketplace. After installation, open Thunder Client and click on "New Request".
- Select the HTTP method (e.g., GET, POST, PUT, DELETE) and enter your API route (e.g., http://localhost:3000/api/users).
- Add any required headers or request body (for POST/PUT).
- Click Send to send the request and view the response directly from your server.

This helps in quickly verifying whether your backend API endpoints are working correctly before connecting them to a frontend application.

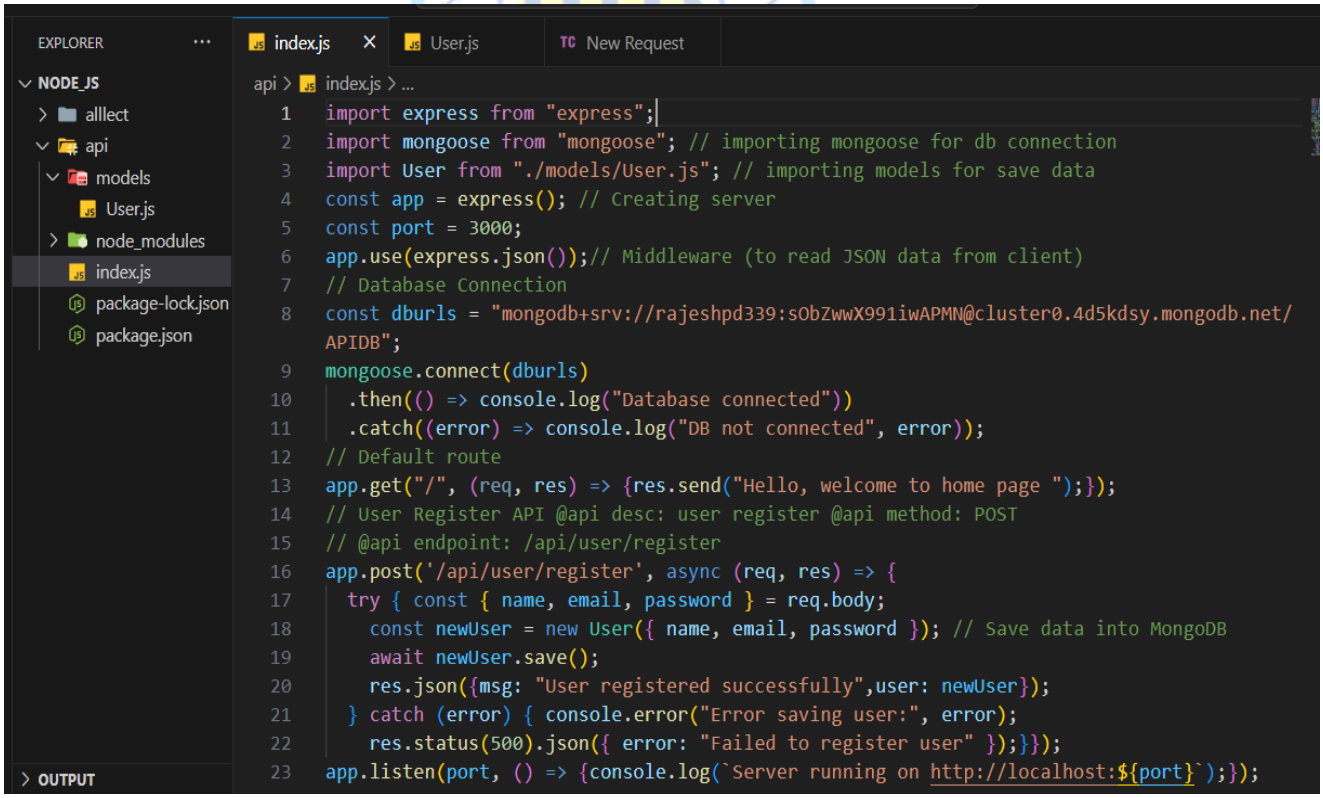


Step-4: Create model (folder) store data

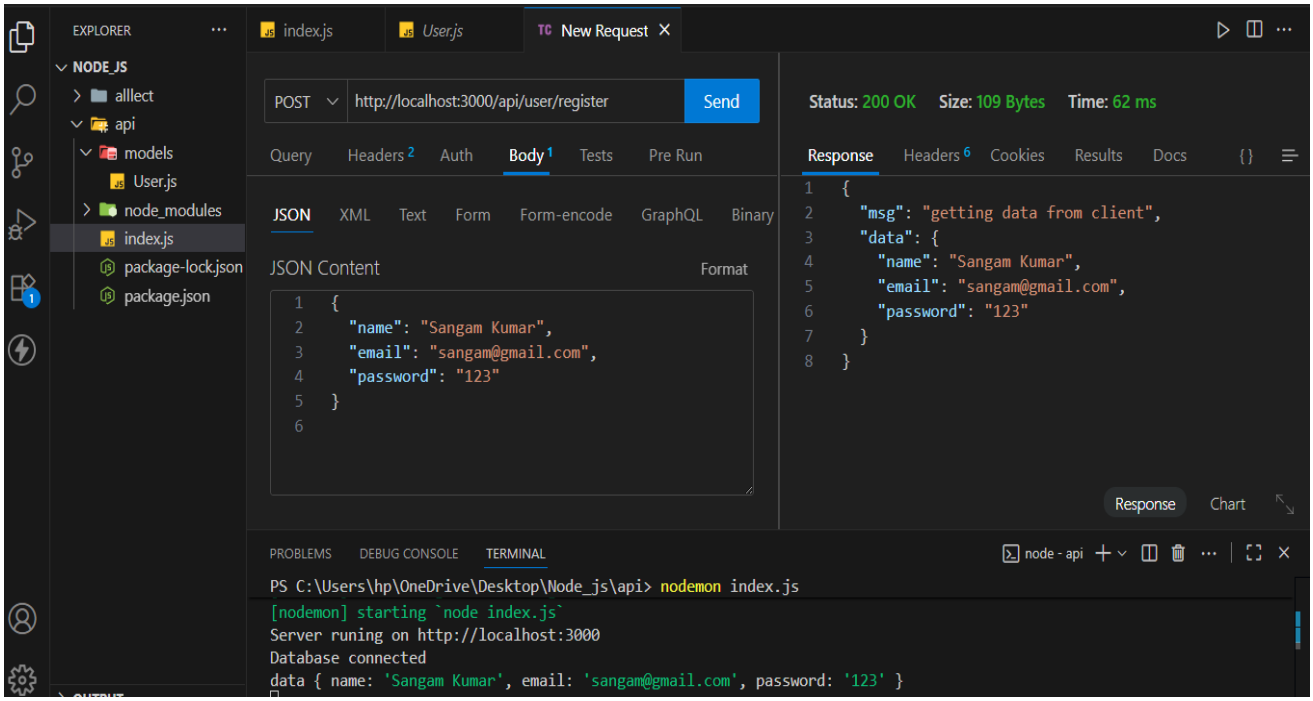


Step-5: Write the route for user models and creating the api

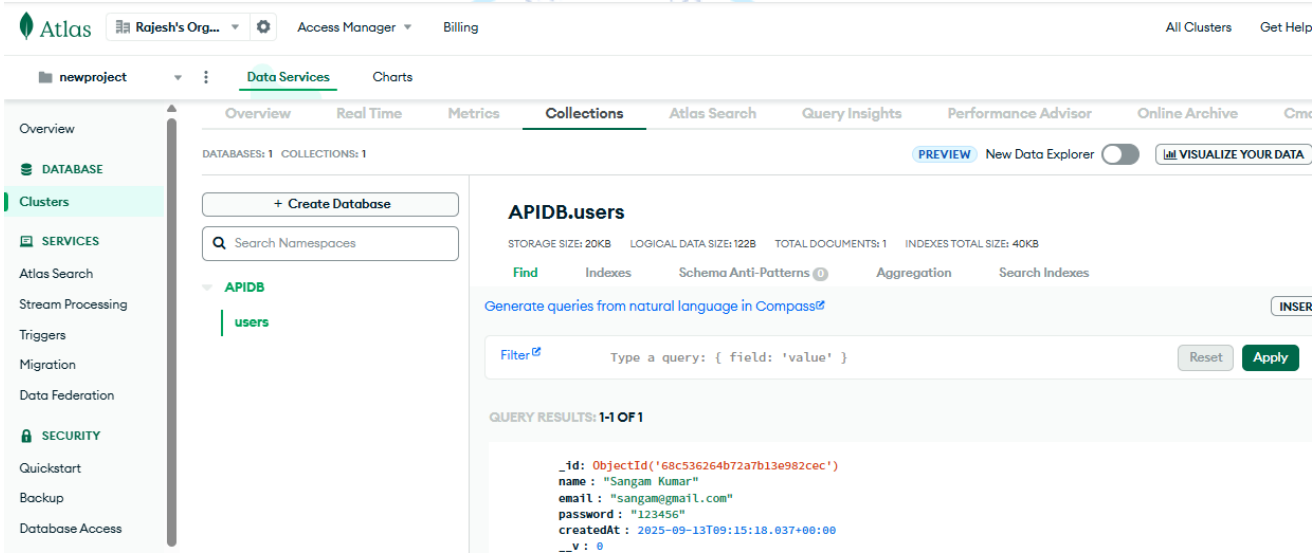
Final Example for testing the api



Test API



Storing data in database



RESTful API in Node.js with MongoDB.

Folder Structure

project-folder/
|— index.js # Main server file
|— package.json
|— models/ # DB Schemas
| └— User.js
|— routes/ # API routes
| └— userRoutes.js
|— controllers/ # Business logic
| └— userController.js

Steps (with requirements)

Step 1: Initialize project

npm init -y

Step 2: Install required packages

npm install express mongoose nodemon

Step 3: Setup package.json

Add "type": "module" and nodemon script:

```
"scripts": {
  "dev": "nodemon index.js"
}
```

Step 4: Create index.js (server + DB connect)

- Import express, mongoose.
- Connect MongoDB (mongoose.connect).
- Add middleware (express.json()).
- Use routes (app.use('/api/users', userRoutes)).
- Start server.

Step 5: Create models/User.js

- Define schema (name, email, password, createdAt).
- Export model.

Step 6: Create controllers/userController.js

- **Functions:**
  - registerUser → POST /register
  - getUsers → GET /
  - getUserById → GET /:id
  - updateUser → PUT /:id
  - deleteUser → DELETE /:id

Step 7: Create routes/userRoutes.js

- Import express.Router().
- Map routes to controller functions.
- Export router.

Step 8: Test API with Postman / Thunder Client

- POST /api/users/register → add new user.
- GET /api/users → list all users.
- PUT /api/users/:id → update user.
- DELETE /api/users/:id → remove user.

Simple api for register the user in MongoDB through the Api

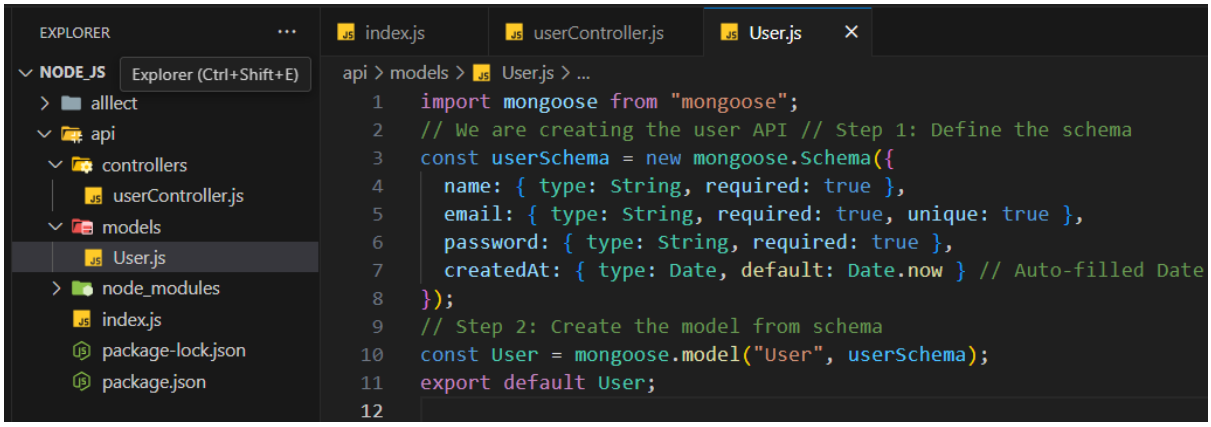
1) Folder structure (what you should have)

```
project-folder/
|— index.js              # updated server file
|— package.json
|— models/
|   └— User.js           # unchanged (your model)
|— controllers/
|   └— userController.js # NEW
└— routes/
    └— userRoutes.js     # NEW
```

2) models/User.js

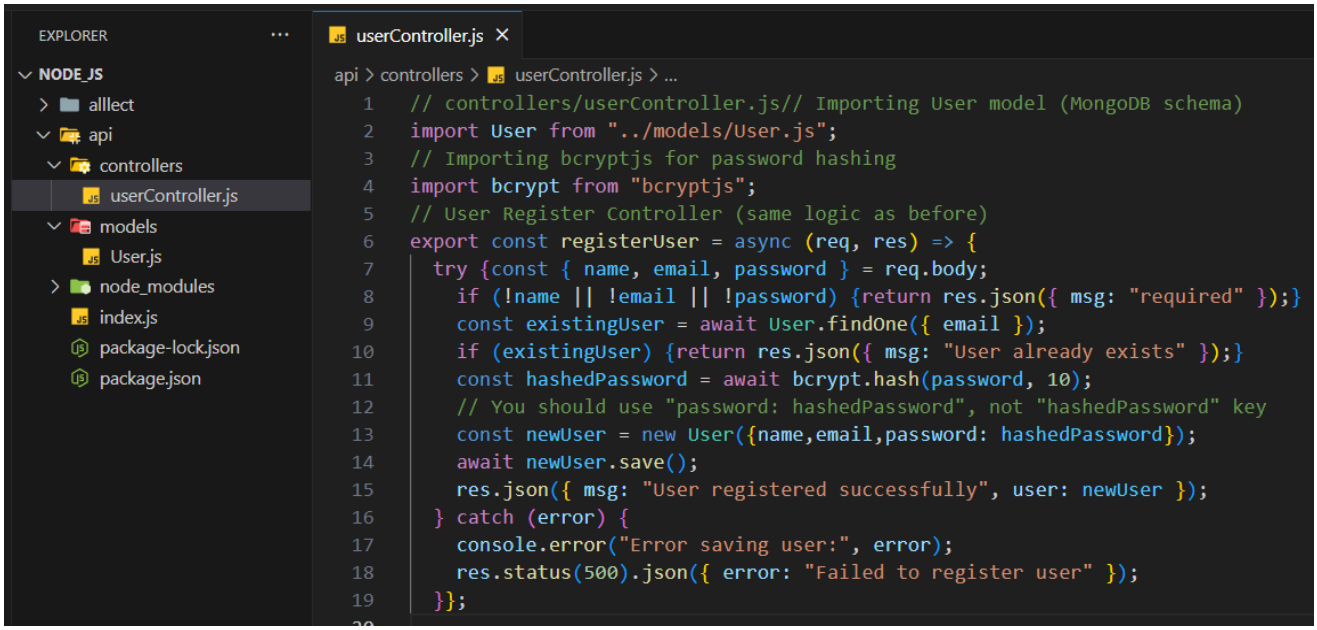
```
import mongoose from "mongoose";  // We are creating the user API // Step 1: Define the schema
const userSchema = new mongoose.Schema({
```

name: { type: String, required: true },
email: { type: String, required: true, unique: true },
password: { type: String, required: true },
createdAt: { type: Date, default: Date.now } **// Auto-filled Date when user is created});**
const User = mongoose.model("User", userSchema); **// Step 2: Create the model from schema**
export default User;



3) Create controllers/userController.js Paste this exact content: //controllers/userController.js
import User from "../models/User.js";
import bcrypt from "bcryptjs"; **// User Register Controller (same logic as before)**
export const registerUser = async (req, res) => {
try {const { name, email, password } = req.body;
if (!name || !email || !password) {return res.json({ msg: "All fields are required" });}
const existingUser = await User.findOne({ email });
if (existingUser) {
return res.json({ msg: "User already exists" }); }
const hashedPassword = await bcrypt.hash(password, 10);
const newUser = new User({name,email,password: hashedPassword});
await newUser.save();
res.json({ msg: "User registered successfully", user: newUser });
} catch (error) {
console.error("Error saving user:", error);
res.status(500).json({ error: "Failed to register user" });}};

Note: This file contains the **same logic and messages** you had in index.js — only moved into a controller function.



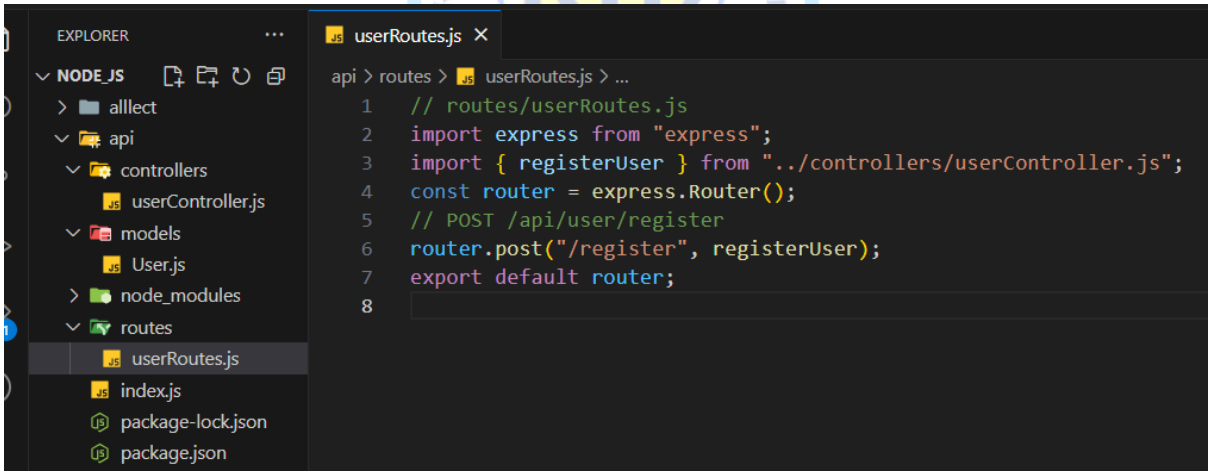
4) Create routes/userRoutes.js → Paste this exact content:

routes/userRoutes.js

```
import express from "express";
import { registerUser } from "../controllers/userController.js"
const router = express.Router();
```

POST /api/user/register

```
router.post("/register", registerUser);
export default router;
```



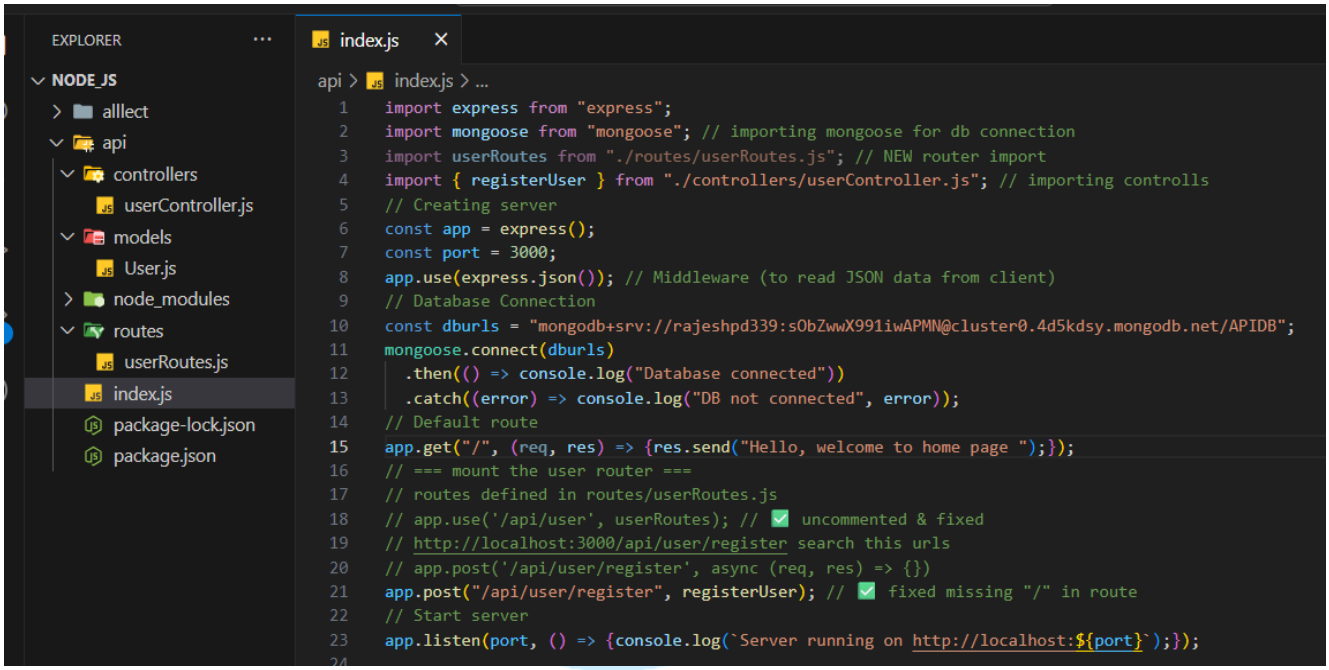
5) index.js (server file)

```
import express from "express";
import mongoose from "mongoose"; // importing mongoose for db connection
import User from "./models/User.js"; // importing models for save data
import bcrypt from "bcryptjs";
import userRoutes from "./routes/userRoutes.js"; // NEW router import
const app = express(); // Creating server
const port = 3000;
app.use(express.json()); // Middleware (to read JSON data from client)
```

// Database Connection

```
const dburls =
"mongodb+srv://rajeshpd339:sObZwwX991iwAPMN@cluster0.4d5kdsy.mongodb.net/APIDB";
mongoose.connect(dburls)
  .then(() => console.log("Database connected"))
  .catch((error) => console.log("DB not connected", error)); // Default route
app.get("/", (req, res) => {res.send("Hello, welcome to home page ");});
// === mount the user router === // routes defined in routes/userRoutes.js
app.use('/api/user', userRoutes);  // Start server
app.listen(port, () => {
  console.log(`Server running on http://localhost:${port}`); });
```

Note: You still have the same database connection and same default route. The register endpoint remains /api/user/register (unchanged behavior) via the mounted router.



6) Quick checklist & commands

1. **Create the folders:**
   mkdir controllers routes
2. **Create the files and paste the code above:**

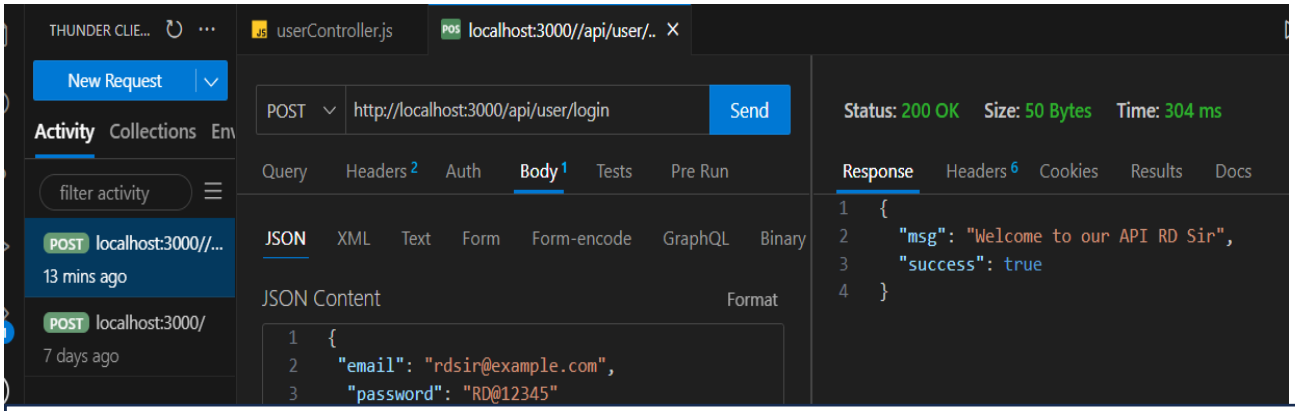
- controllers/userController.js
- routes/userRoutes.js
- update index.js as shown

3. Restart server: npm run dev # or nodemon index.js / node index.js
4. Test with Thunder Client / Postman:

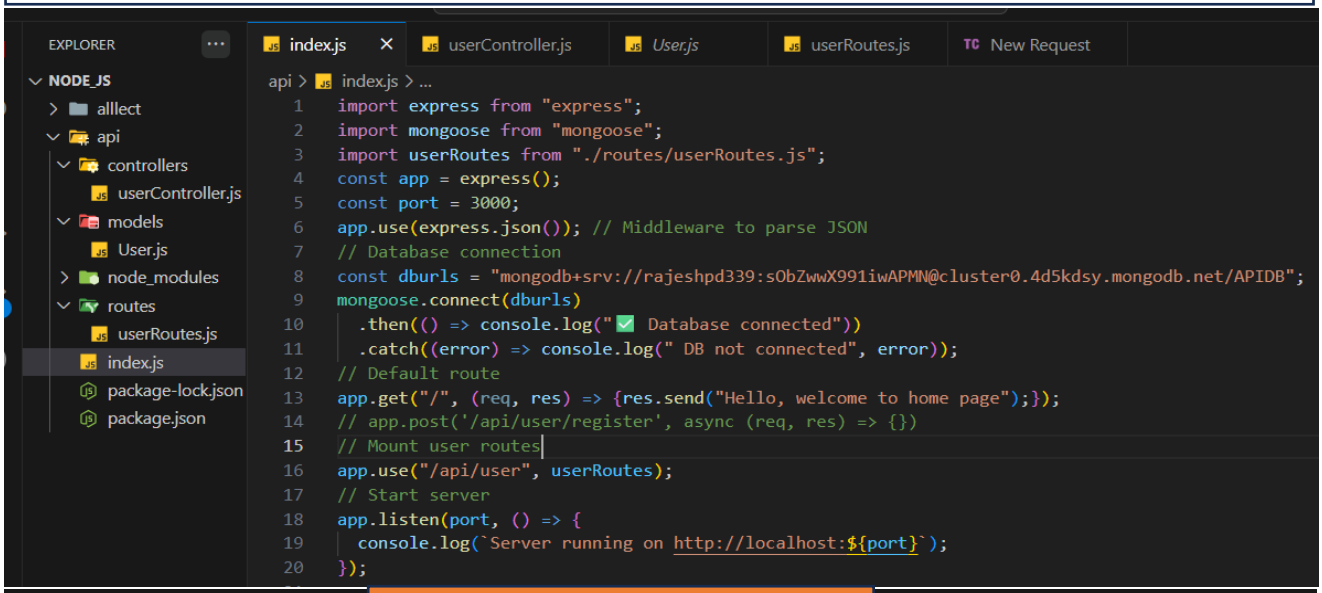
- **POST** http://localhost:3000/api/user/register
- Body (JSON):

{ "name": "Rajesh",
  "email": "rajesh@example.com",
  "password": "123456"
} You should get the same response as earlier:
{"msg": "User registered successfully",
  "user": { ... } }

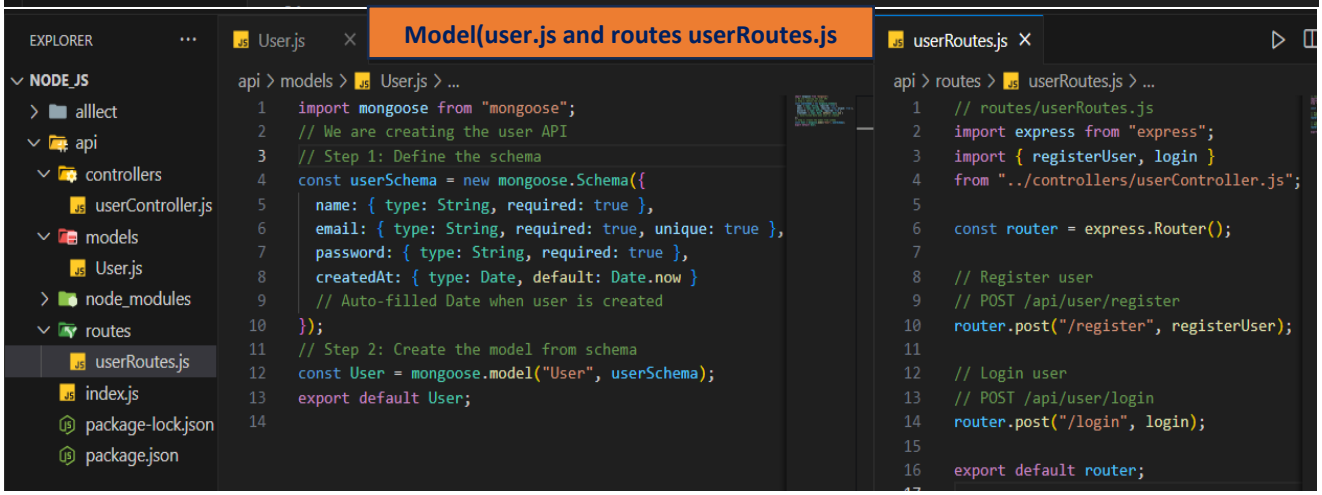
And the user document will be saved in APIDB.users on MongoDB Atlas.



Complete Restful Api with database connection Login Register and Login Step-1: Index.js



Model(user.js and routes userRoutes.js



userController.js

api > controllers > userController.js > login

```
// controllers/userController.js
import User from "../models/User.js";
import bcrypt from "bcryptjs";// Register User
export const registerUser = async (req, res) => {
  try {const { name, email, password } = req.body;
    if (!name || !email || !password) {return res.json({ msg: "All fields are required" });}
    const existingUser = await User.findOne({ email });
    if (existingUser) {return res.json({ msg: "User already exists" });}
    const hashedPassword = await bcrypt.hash(password, 10);
    const newUser = new User({ name, email, password: hashedPassword });
    await newUser.save();
    res.json({ msg: "User registered successfully", user: newUser });
  } catch (error) { console.error("Error saving user:", error);
    res.status(500).json({ error: "Failed to register user" });}};
// Login User
export const login = async (req, res) => {
  try {const { email, password } = req.body;
    if (!email || !password) { return res.json({ msg: "All fields are required" });}
    //  Use User (capital U)
    const user = await User.findOne({ email });
    if (!user) {return res.json({ msg: "User not found", success: false });}
    // password is user definde password and user.password is db password bcrypt and match
    const validPass = await bcrypt.compare(password, user.password);
    if (!validPass) {return res.json({ msg: "Invalid password", success: false });}
    res.json({ msg: `Welcome to our API ${user.name}`, success: true });
  } catch (error) {console.error("Login error:", error);
    res.status(500).json({ error: "Login failed" });}};

```

Cookies Node js

1. What are Cookies in Node.js?

- **Cookies** are small pieces of data stored on the **client's browser**.
- The server sends cookies with the response → the browser saves them → they are automatically sent back with every request.
- **Useful for**: Storing login tokens (like JWT) Remembering sessions Keeping track of user preferences

2. What is jsonwebtoken (JWT)?

- **JWT** = JSON Web Token, a compact way to securely send information between server and client.
- **When user logs in:** Server verifies credentials Creates a JWT (signed with a secret key)
  - Sends it to the client (stored in cookie or localStorage)
- On future requests, the client sends back the token → server verifies it → allows access.

3. Install Required Packages

- Run this in terminal:
- **npm install jsonwebtoken cookie-parser**
  - jsonwebtoken → creates and verifies tokens
  - cookie-parser → helps read/write cookies in Node.js

4. Basic JWT Syntax

```
import jwt from "jsonwebtoken";
// Create token
const token = jwt.sign({ userId: 123 }, "SECRET_KEY", { expiresIn: "1h" });
// Verify token
const decoded = jwt.verify(token, "SECRET_KEY");
```

5.work on controls parts

→ jwt.sign() Method

➔ The **jwt.sign()** method is used to **create (sign) a JSON Web Token (JWT)**. It takes some information (called **payload**) and signs it with a secret key to ensure that the token cannot be tampered with.

Syntax: jwt.sign(payload, secretOrPrivateKey, [options, callback])

- **payload** → The data you want to store inside the token (e.g., user ID, role, email).
- **secretOrPrivateKey** → A secret string or private key used to sign the token (must be kept safe).
- **options** → (Optional) Settings like expiration time, algorithm, etc.
- **callback** → (Optional) If provided, the method works asynchronously.

```
import jwt from "jsonwebtoken";
// Create a token
const token = jwt.sign(
  { userId: 123 },          // payload (data)
  "SECRET_KEY",             // secret key
  { expiresIn: "1h" }       // options ( valid for 1 hour)
);
console.log(token);
```

Valid time formats you can use in **expiresIn**:

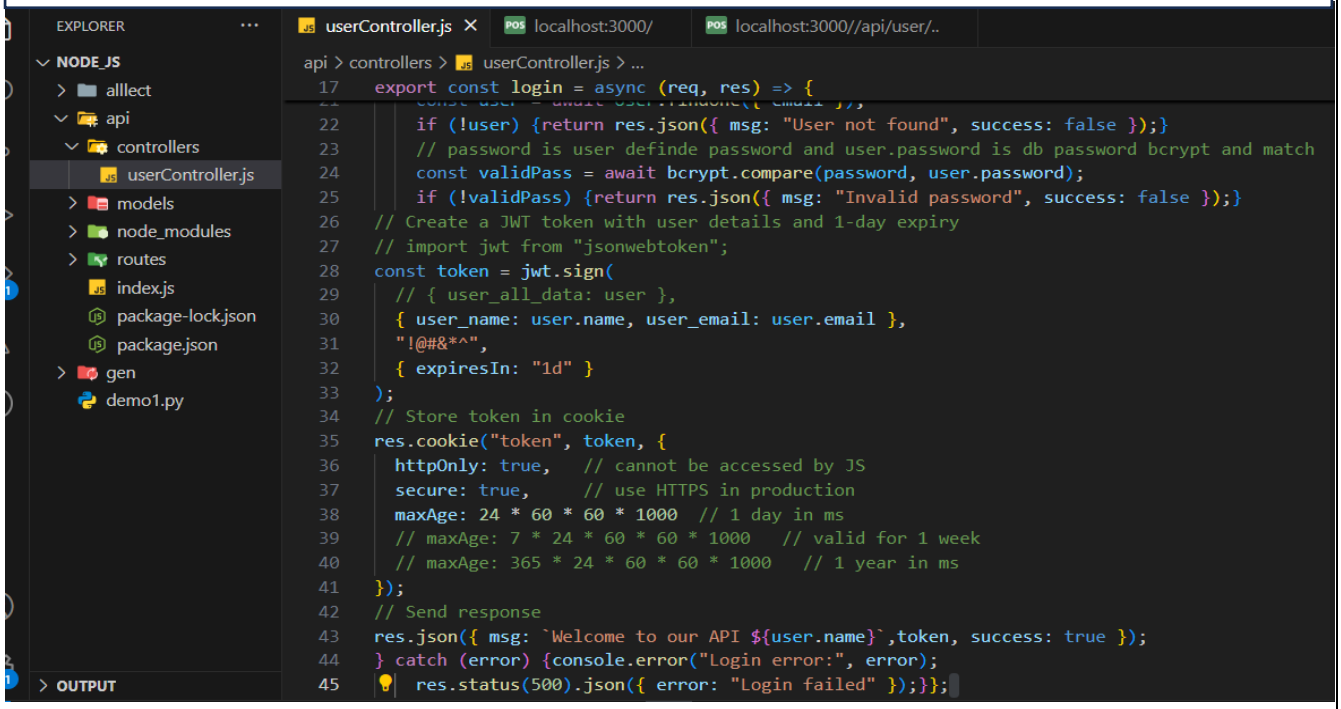
- Seconds → "60"
- Minutes → "10m"
- Hours → "1h"
- Days → "7d", "30d", "365d"

No expiry (not recommended)

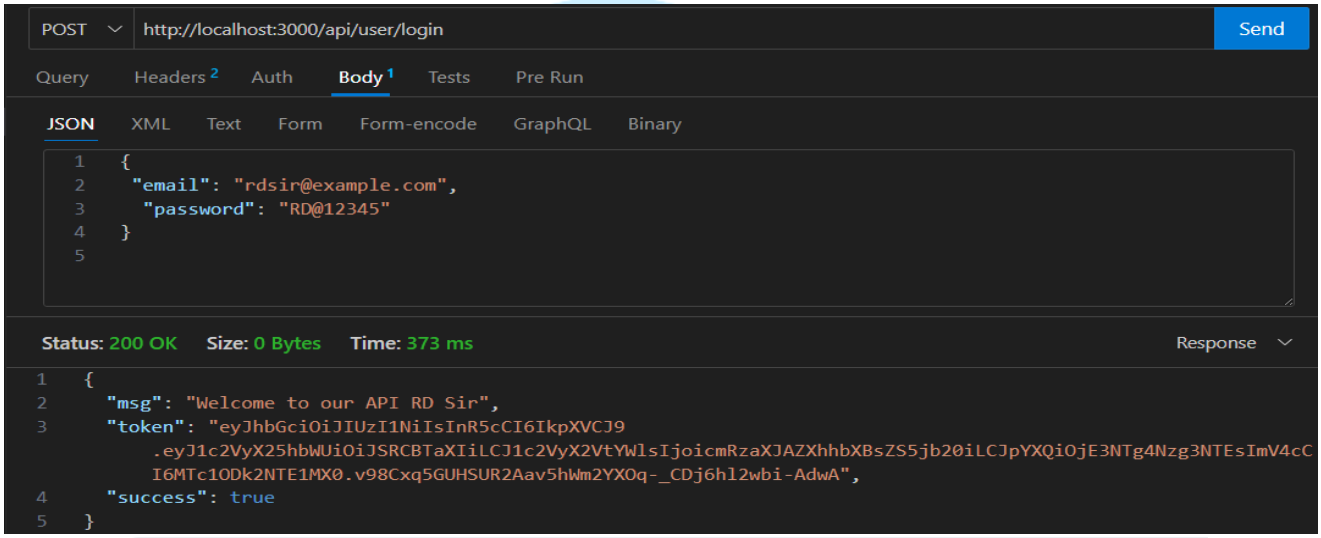
Explanation of Example

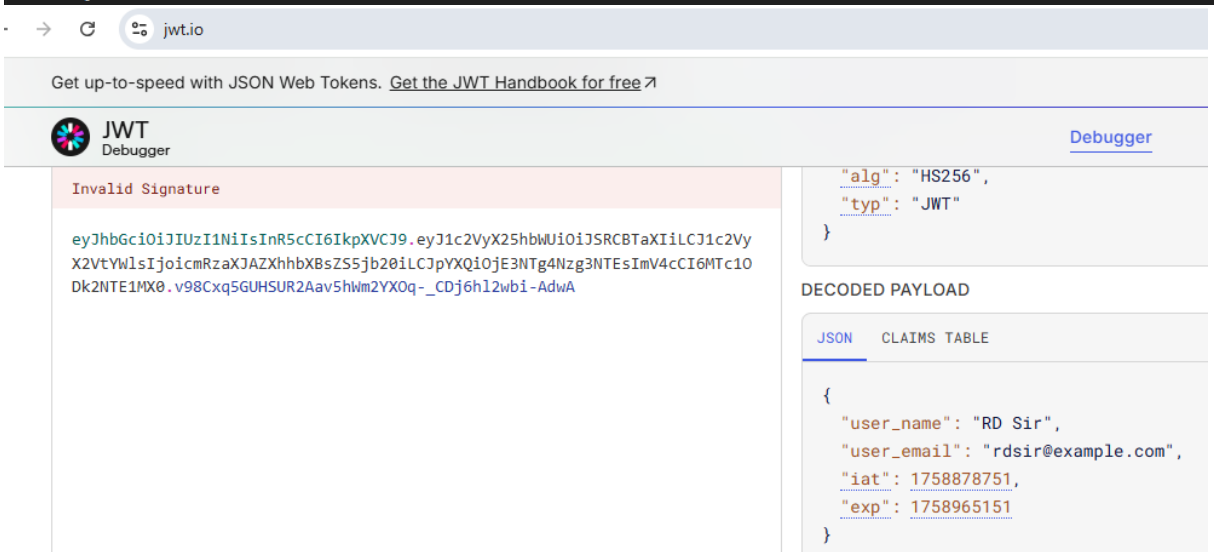
1. **Payload**:
2. { userId: 123 }
   - Stores user ID inside the token.
   - This can be later extracted to identify the user.
3. **Secret Key**:
   "SECRET_KEY"
   - Used to **sign** the token.
   - Only the server knows this key, so it can later verify if the token is valid.
4. **Options (expiresIn)**:
   { expiresIn: "1h" }
   - Token will automatically expire after **1 hour**.
   - Common values: "60s", "10m", "1d", etc.

Write the code for create the cookies in Controls.js



Thunder clients





Example:

```
// controllers/userController.js
import User from "../models/User.js";
import bcrypt from "bcryptjs";
import jwt from "jsonwebtoken";
// Register User (same as before)
export const registerUser = async (req, res) => {
 try {
  const { name, email, password } = req.body;
  if (!name || !email || !password) {return res.json({ msg: "All fields are required" });}
  const existingUser = await User.findOne({ email });
  if (existingUser) {return res.json({ msg: "User already exists" });}
  const hashedPassword = await bcrypt.hash(password, 10);
  const newUser = new User({ name, email, password: hashedPassword });
  await newUser.save();
  res.json({ msg: "User registered successfully", user: newUser });
 } catch (error) {
  console.error("Error saving user:", error);
  res.status(500).json({ error: "Failed to register user" });}};
// Login User with JWT + Cookie
export const login = async (req, res) => {
 try { const { email, password } = req.body;
  if (!email || !password) { return res.json({ msg: "All fields are required" });}
  const user = await User.findOne({ email });
  if (!user) { return res.json({ msg: "User not found", success: false }); }
  const validPass = await bcrypt.compare(password, user.password);
  if (!validPass) {return res.json({ msg: "Invalid password", success: false });}
  // Create JWT token
  const token = jwt.sign(
   { id: user._id, email: user.email }, // payload
   "MY_SECRET_KEY",                  // secret key
   { expiresIn: "1h" }               // validity
  );
  // Save token in cookie
  res.cookie("authToken", token, {
   httpOnly: true, // not accessible via JS (secure)
   secure: false,  // set true if using https
   maxAge: 60 * 60 * 1000 // 1 hour
  });
  res.json({ msg: `Welcome ${user.name}`, success: true });
 } catch (error) {
  console.error("Login error:", error);
  res.status(500).json({ error: "Login failed" });
 }
};
```

6. Middleware to Protect Routes

Later, if you want to protect routes, add:

```
import jwt from "jsonwebtoken";
export const verifyToken = (req, res, next) => {
  const token = req.cookies.authToken;
  if (!token) {return res.status(401).json({ msg: "Access denied, no token provided" }); }
  try {
    const decoded = jwt.verify(token, "MY_SECRET_KEY");
    req.user = decoded; // store decoded user data
    next();
  } catch (err) { res.status(400).json({ msg: "Invalid token" });}};
```

Use in routes:

```
router.get("/dashboard", verifyToken, (req, res) => {
  res.json({ msg: `Hello ${req.user.email}, welcome to dashboard` });
});
```

In short:

1. **Login → create JWT → save in cookie**
2. **Future requests → read cookie → verify JWT → allow/deny**

❖ What is Session Management in Node.js (Express)?

What is Session Management in Node.js (Express)?

- **Session Management** is a way to store user-specific data on the server for use across multiple requests.
- For example, when a user logs in, you can store their user ID or name in a session so that they stay logged in across different pages until they log out.

Why Session Management is Needed

- To **maintain user login state**.
- To **store temporary user data** (like shopping cart items).
- To **identify** which user is making the request.

How Sessions Work in Express

1. When a user logs in → a **session** is created on the server.
2. A unique **Session ID** is sent to the user as a **cookie**.
3. On every next request, this **Session ID** is used to find the user's session data.

```
npm init -y
npm install express express-session
```

```
import express from "express";
import session from "express-session";
const app = express();
app.use(express.urlencoded({ extended: true }));// Configure Session Middleware
app.use(session({
    secret: "mysecretkey",        // Used to sign the session ID cookie
    resave: false,                // Don't save session if unmodified
    saveUninitialized: true,      // Save new sessions
    cookie: { maxAge: 60000 }     // Session expires after 1 minute
}));// Home Route
app.get("/", (req, res) => {
    if (req.session.username) {
        res.send(`<h2>Welcome back, ${req.session.username}!</h2>
                  <a href='/logout'>Logout</a>`);
    } else { res.send(`
            <h2>Login Form</h2>
            <form method="POST" action="/login">
                <input name="username" placeholder="Enter Username" required />
                <button type="submit">Login</button>
            </form>`);}
});// Login Route
app.post("/login", (req, res) => {
    const { username } = req.body;
    req.session.username = username;  // Store username in session
    res.redirect("/");}); // Logout Route
app.get("/logout", (req, res) => {
    req.session.destroy(err => {
        if (err) { return res.send("Error logging out");}
        res.clearCookie("connect.sid"); // Remove session cookie
        res.redirect("/");
    });});// Start Server
app.listen(3000, () => {console.log("Server started on http://localhost:3000");});
```

F

Below is the **cleanest, shortest, teacher-friendly MERN Stack project** redesigned **with RTS (MCQ Exam Page)**.

Everything is:

✓ Simple

✓ Easy to explain

✓ Perfect for teaching

✓ Includes **Home, About, Services, Signup, Login, RTS (MCQ Exam)**

✓ After login → automatically redirect to **RTS Exam Page**

☑ **1. Final MERN Folder Structure (Easy for Students)**

```
mern-class/
|
├── backend/
|   ├── server.js
|   ├── .env
|   ├── package.json
|   ├── config/
|   |   └── db.js
|   ├── models/
|   |   └── User.js
|   ├── controllers/
|   |   └── userController.js
|   └── routes/
|       └── userRoutes.js
|
└── frontend/
    ├── package.json
    └── src/
        ├── App.js
        ├── index.js
        ├── pages/
        |   ├── Home.js
        |   ├── About.js
        |   ├── Services.js
        |   ├── Signup.js
        |   ├── Login.js
        |   └── RTS.js    ← (5 MCQ exam page)
        └── components/
            └── Navbar.js
```

☑ **2. Packages To Install (Explain Easily to Students)**

Backend Modules

Module	Why we install
express	create server

Module	Why we install
mongoose	connect MongoDB
cors	allow frontend–backend connection
dotenv	load .env file

```
cd backend
npm init -y
npm install express mongoose cors dotenv
```

Frontend Modules

Module	Why install
axios	send API requests
react-router-dom	page navigation

```
cd frontend
npx create-react-app .
npm install axios react-router-dom
```

☑ **3. BACKEND CODE (Short & Clean)**

📌 **server.js**

```
const express = require("express");
const cors = require("cors");
const connectDB = require("./config/db");
require("dotenv").config();

const app = express();
app.use(cors());
app.use(express.json());

connectDB();

app.use("/api/users", require("./routes/userRoutes"));

app.listen(5000, () => console.log("Server running on 5000"));
```

📌 **config/db.js**

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

const connectDB = async () => {
  await mongoose.connect(process.env.MONGO_URL);
  console.log("MongoDB Connected");
};

module.exports = connectDB;
```

📌 **.env**

```
MONGO_URL=mongodb://localhost:27017/mernclass
```

📌 **models/User.js**

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

const UserSchema = new mongoose.Schema({
  name: String,
  email: String,
  password: String
});

module.exports = mongoose.model("User", UserSchema);
```

📌 **controllers/userController.js**

```
const User = require("../models/User");

// Signup
exports.signup = async (req, res) => {
  const user = await User.create(req.body);
  res.json({ message: "Signup Success" });
};

// Login
exports.login = async (req, res) => {
  const user = await User.findOne({ email: req.body.email });

  if (!user || user.password !== req.body.password) {
    return res.json({ message: "Invalid Credentials" });
  }

  res.json({ message: "Login Success", user });
};
```

📌 **routes/userRoutes.js**

```
const express = require("express");
const { signup, login } = require("../controllers/userController");

const router = express.Router();

router.post("/signup", signup);
router.post("/login", login);

module.exports = router;
```

☑ **4. FRONTEND (React Code)**

📌 **Navbar.js (Added RTS Page Link)**

```
import { Link } from "react-router-dom";
```

```
function Navbar() {
 return (
  <nav style={{ padding: 10, background: "#eee" }}>
   <Link to="/">Home</Link> |
   <Link to="/about">About</Link> |
   <Link to="/services">Services</Link> |
   <Link to="/rts">RTS Exam</Link> |
   <Link to="/signup">Signup</Link> |
   <Link to="/login">Login</Link>
  </nav>
 );
}
export default Navbar;
```

📌 **App.js (Added RTS Route + Login Redirect)**

```
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Navbar from "./components/Navbar";

import Home from "./pages/Home";
import About from "./pages/About";
import Services from "./pages/Services";
import Signup from "./pages/Signup";
import Login from "./pages/Login";
import RTS from "./pages/RTS";

function App() {
 return (
  <BrowserRouter>
   <Navbar />
   <Routes>
    <Route path="/" element={<Home />} />
    <Route path="/about" element={<About />} />
    <Route path="/services" element={<Services />} />
    <Route path="/signup" element={<Signup />} />
    <Route path="/login" element={<Login />} />
    <Route path="/rts" element={<RTS />} />
   </Routes>
  </BrowserRouter>
 );
}
export default App;
```

📌 **RTS.js (5 Very Simple MCQ Questions)**

```
function RTS() {
 return (
  <div>
   <h1>RTS MCQ Exam</h1>
```

```
    <ol>
     <li>What is 2 + 2?
      <br/>A) 3 B) 4 C) 5
     </li>

     <li>React is a?
      <br/>A) Framework B) Library C) Language
     </li>

     <li>Node.js runs on?
      <br/>A) Browser B) Server C) Phone
     </li>

     <li>MongoDB stores data as?
      <br/>A) Tables B) JSON C) Images
     </li>

     <li>JS stands for?
      <br/>A) JavaStone B) JavaScript C) JetScript
     </li>
    </ol>
   </div>
  );
}
export default RTS;
```

Signup.js

```
import axios from "axios";
import { useState } from "react";

function Signup() {
  const [form, setForm] = useState({ name: "", email: "", password: "" });

  const handleSubmit = async () => {
    await axios.post("http://localhost:5000/api/users/signup", form);
    alert("Signup Success");
  };

  return (
    <div>
      <h1>Signup</h1>
      <input placeholder="Name" onChange={(e)=>setForm({...form, name:e.target.value})}/>
      <input placeholder="Email" onChange={(e)=>setForm({...form, email:e.target.value})}/>
      <input type="password" placeholder="Password" onChange={(e)=>setForm({...form,
password:e.target.value})}/>
      <button onClick={handleSubmit}>Signup</button>
    </div>
```

```
  );
}
export default Signup;
```

📌 **Login.js (Redirect to RTS after Login)**

```
import axios from "axios";
import { useState } from "react";
import { useNavigate } from "react-router-dom";

function Login() {
  const [form, setForm] = useState({ email: "", password: "" });
  const navigate = useNavigate();

  const handleLogin = async () => {
    const res = await axios.post("http://localhost:5000/api/users/login", form);
    alert(res.data.message);

    if (res.data.message === "Login Success") {
      navigate("/rts");  // redirect to MCQ page
    }
  };

  return (
    <div>
      <h1>Login</h1>
      <input placeholder="Email" onChange={(e)=>setForm({...form, email:e.target.value})}/>
      <input type="password" placeholder="Password" onChange={(e)=>setForm({...form,
password:e.target.value})}/>
      <button onClick={handleLogin}>Login</button>
    </div>
  );
}
export default Login;
```

🎉 **Your Teaching-Friendly MERN Project is Now Complete**

✓ **Signup**
✓ **Login**
✓ **RTS MCQ Exam Page**
✓ **Redirect to RTS after Login**
✓ **Clean Folder Structure**
✓ **Simple Code to Explain in Class**
✓ **Can Deploy Anywhere**

👉 **If you want next:**

☑ Add scoring to RTS MCQ
☑ Store exam results in MongoDB

☑ Add dashboard for students
☑ JWT authentication (very simple version)
☑ Deployment guide (Render + Vercel)
Just say: **"Add next feature"** or **"Deploy this project"**.

Here is **exactly where and how** to add your MongoDB URL **inside the MERN project** (for teaching purpose — super simple).

☑ **Where to add the MongoDB URL?**
You must add it in this file:
🗀 **backend/config/db.js**
This file is **only for database connection**.

☑ **What to add?**
Replace previous code and use **your given MongoDB connection code**.

📌 **Final db.js file (with your Mongo URL)**
backend/config/db.js

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

const connectDB = async () => {
  const mongoURI = "mongodb://127.0.0.1:27017/userdatabase";
  // OR: mongodb://localhost:27017/userdatabase

  mongoose.connect(mongoURI)
    .then(() => console.log("Local MongoDB Connected Successfully"))
    .catch(err => console.error("DB Connection Error:", err));
};

module.exports = connectDB;
```

📌 **Also update server.js (no change, just call connectDB)**
backend/server.js

```
const express = require("express");
const cors = require("cors");
const connectDB = require("./config/db");

const app = express();
app.use(cors());
app.use(express.json());

connectDB();  // <-- connection happens here

app.use("/api/users", require("./routes/userRoutes"));

app.listen(5000, () => console.log("Server running on 5000"));
```

⚠ **Do we need .env now?**

❌ **NO — Not required**, because you are using a **direct local URL**.

(But if you want .env later, I can set that too.)

🎯 **CLEAR EXPLANATION FOR STUDENTS**

✓ ***db.js***

→ Only handles database connection

→ Connects Node.js backend to MongoDB

→ Uses mongoose.connect()

→ Prints success or error

✓ ***server.js***

→ Creates API server

→ Calls connectDB()

🎉 **Your backend is now connected properly with:**

📌 MongoDB (local)

📌 Node.js

📌 Express

📌 Mongoose

If you want, I can now:

☑ Add .env version

☑ Add cloud MongoDB (MongoDB Atlas)

☑ Add user authentication (JWT)

☑ Add MCQ result saving in DB

Just tell me: **"Add next feature"**.

Great! Let's add the **next important feature** to your MERN teaching project:

🎯 **NEXT FEATURE: Save RTS MCQ Exam Results to MongoDB**

This will make your project **proper**, **practical**, and perfect for students to understand **full MERN data flow (Frontend → Backend → Database)**.

☑ **What we will add**

✓ A new **Result Model**

✓ A new **Controller** to save exam results

✓ A new **Route**

✓ Update RTS page to submit answers

✓ Store result in MongoDB

✓ Show alert: “Exam Submitted Successfully!”

This will complete your **RTS Exam Module**.

📌 **1. Create Result Model**

📁 backend/models/Result.js

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

const ResultSchema = new mongoose.Schema({
  userEmail: String,
  score: Number,
  answers: Array
});

module.exports = mongoose.model("Result", ResultSchema);
```

📌 **2. Create resultController.js**

📁 backend/controllers/resultController.js

```
const Result = require("../models/Result");

exports.saveResult = async (req, res) => {
  const { userEmail, score, answers } = req.body;

  await Result.create({ userEmail, score, answers });

  res.json({ message: "Exam Submitted Successfully!" });
};
```

📌 **3. Create resultRoutes.js**

📁 backend/routes/resultRoutes.js

```
const express = require("express");
const { saveResult } = require("../controllers/resultController");

const router = express.Router();

router.post("/save", saveResult);

module.exports = router;
```

📌 **4. Add this route to server.js**

📁 backend/server.js

```
app.use("/api/result", require("./routes/resultRoutes"));
```

📌 **5. Update RTS.js (Frontend MCQ exam)**

📁 frontend/src/pages/RTS.js

👉 Very simple scoring + send to backend

```
import { useState } from "react";
import axios from "axios";

function RTS() {
  const [answers, setAnswers] = useState({});
```

```
const correctAns = {
  q1: "4",
  q2: "Library",
  q3: "Server",
  q4: "JSON",
  q5: "JavaScript"
};

const handleSubmit = async () => {
  let score = 0;

  Object.keys(correctAns).forEach(q => {
    if (answers[q] === correctAns[q]) score++;
  });

  // Send result to DB
  await axios.post("http://localhost:5000/api/result/save", {
    userEmail: "student@gmail.com", // later we will get from login
    score,
    answers
  });

  alert("Exam Submitted! Score: " + score);
};

return (
  <div>
    <h1>RTS MCQ Exam</h1>

    <ol>
      <li>
        What is 2 + 2?
        <br />
        <select onChange={(e)=>setAnswers({...answers, q1: e.target.value})}>
          <option></option>
          <option>3</option>
          <option>4</option>
          <option>5</option>
        </select>
      </li>

      <li>
        React is a?
        <br />
        <select onChange={(e)=>setAnswers({...answers, q2: e.target.value})}>
          <option></option>
          <option>Framework</option>
          <option>Library</option>
```

```
          <option>Language</option>
         </select>
        </li>

        <li>
         Node.js runs on?
         <br />
         <select onChange={(e)=>setAnswers({...answers, q3: e.target.value})}>
          <option></option>
          <option>Browser</option>
          <option>Server</option>
          <option>Phone</option>
         </select>
        </li>

        <li>
         MongoDB stores data as?
         <br />
         <select onChange={(e)=>setAnswers({...answers, q4: e.target.value})}>
          <option></option>
          <option>Tables</option>
          <option>JSON</option>
          <option>Images</option>
         </select>
        </li>

        <li>
         JS stands for?
         <br />
         <select onChange={(e)=>setAnswers({...answers, q5: e.target.value})}>
          <option></option>
          <option>JavaStone</option>
          <option>JavaScript</option>
          <option>JetScript</option>
         </select>
        </li>
       </ol>

       <button onClick={handleSubmit}>Submit Exam</button>
      </div>
    );
}

export default RTS;
```

NOW YOUR RTS MODULE IS FULLY FUNCTIONAL

✓ Exam Questions
✓ User Selects Answers
✓ Score Calculation
✓ Result Sent to Backend
✓ Saved in MongoDB
✓ Everything connected
This turns your MERN project into a **real-world exam system**.

🚀 **Want the next feature?**
Here are available options:
1️⃣ Show score history to student
2️⃣ Add timer to RTS exam
3️⃣ Add JWT authentication (simple, teaching version)
4️⃣ Add Admin Panel
5️⃣ Add Result dashboard
6️⃣ Add Logout + Protected Routes
Tell me: "Add Feature 2", "Add Feature 3", etc.

Great! Let's add **Feature 2 → TIMER for RTS MCQ Exam**.
This is a **very simple, clean, teaching-friendly version**.
Perfect to explain to students how **React state + useEffect + countdown timer** works.

🎯 **FEATURE 2: Add Timer (Countdown) to RTS Exam**
✓ **Timer starts automatically**
✓ **Shows countdown (e.g., 60 seconds)**
✓ **Auto-submit exam when time ends**
✓ **Alert "Time Over – Exam Submitted"**

📌 **1. Update RTS.js (Frontend)**
Replace previous RTS.js with this **neat version**.
📁 frontend/src/pages/RTS.js

```
import { useState, useEffect } from "react";
import axios from "axios";

function RTS() {
  const [answers, setAnswers] = useState({});
  const [timeLeft, setTimeLeft] = useState(60); // 60 seconds timer

  const correctAns = {
    q1: "4",
    q2: "Library",
    q3: "Server",
    q4: "JSON",
    q5: "JavaScript"
  };
```

```
// TIMER: counts down every 1 second
useEffect(() => {
  if (timeLeft <= 0) {
    submitExam(true); // auto submit when time ends
    return;
  }

  const timer = setInterval(() => {
    setTimeLeft(timeLeft - 1);
  }, 1000);

  return () => clearInterval(timer);
}, [timeLeft]);

// Exam Submit Function
const submitExam = async (auto = false) => {
  let score = 0;

  Object.keys(correctAns).forEach(q => {
    if (answers[q] === correctAns[q]) score++;
  });

  // Save in DB
  await axios.post("http://localhost:5000/api/result/save", {
    userEmail: "student@gmail.com",
    score,
    answers
  });

  if (auto) {
    alert(" ⏰ Time Over! Exam auto-submitted.\nYour Score: " + score);
  } else {
    alert("Exam Submitted!\nScore: " + score);
  }
};

return (
  <div>
    <h1>RTS MCQ Exam</h1>

    {/* Timer Block */}
    <h2 style={{ color: "red" }}>
      Time Left: {timeLeft} seconds
    </h2>

    <ol>
      <li>
```

```
  What is 2 + 2?
  <select onChange={(e)=>setAnswers({...answers, q1: e.target.value})}>
   <option></option>
   <option>3</option>
   <option>4</option>
   <option>5</option>
  </select>
</li>

<li>
  React is a?
  <select onChange={(e)=>setAnswers({...answers, q2: e.target.value})}>
   <option></option>
   <option>Framework</option>
   <option>Library</option>
   <option>Language</option>
  </select>
</li>

<li>
  Node.js runs on?
  <select onChange={(e)=>setAnswers({...answers, q3: e.target.value})}>
   <option></option>
   <option>Browser</option>
   <option>Server</option>
   <option>Phone</option>
  </select>
</li>

<li>
  MongoDB stores data as?
  <select onChange={(e)=>setAnswers({...answers, q4: e.target.value})}>
   <option></option>
   <option>Tables</option>
   <option>JSON</option>
   <option>Images</option>
  </select>
</li>

<li>
  JS stands for?
  <select onChange={(e)=>setAnswers({...answers, q5: e.target.value})}>
   <option></option>
   <option>JavaStone</option>
   <option>JavaScript</option>
   <option>JetScript</option>
  </select>
</li>
```

```
      </ol>

      <button onClick={() => submitExam(false)}>Submit Exam</button>
    </div>
  );
}

export default RTS;
```

🎉 **Now Your RTS Exam Has a Working Timer**

✓ Timer starts automatically
✓ Counts down from **60 seconds**
✓ Auto submits when time ends
✓ Saves result in MongoDB
✓ Alerts student with score

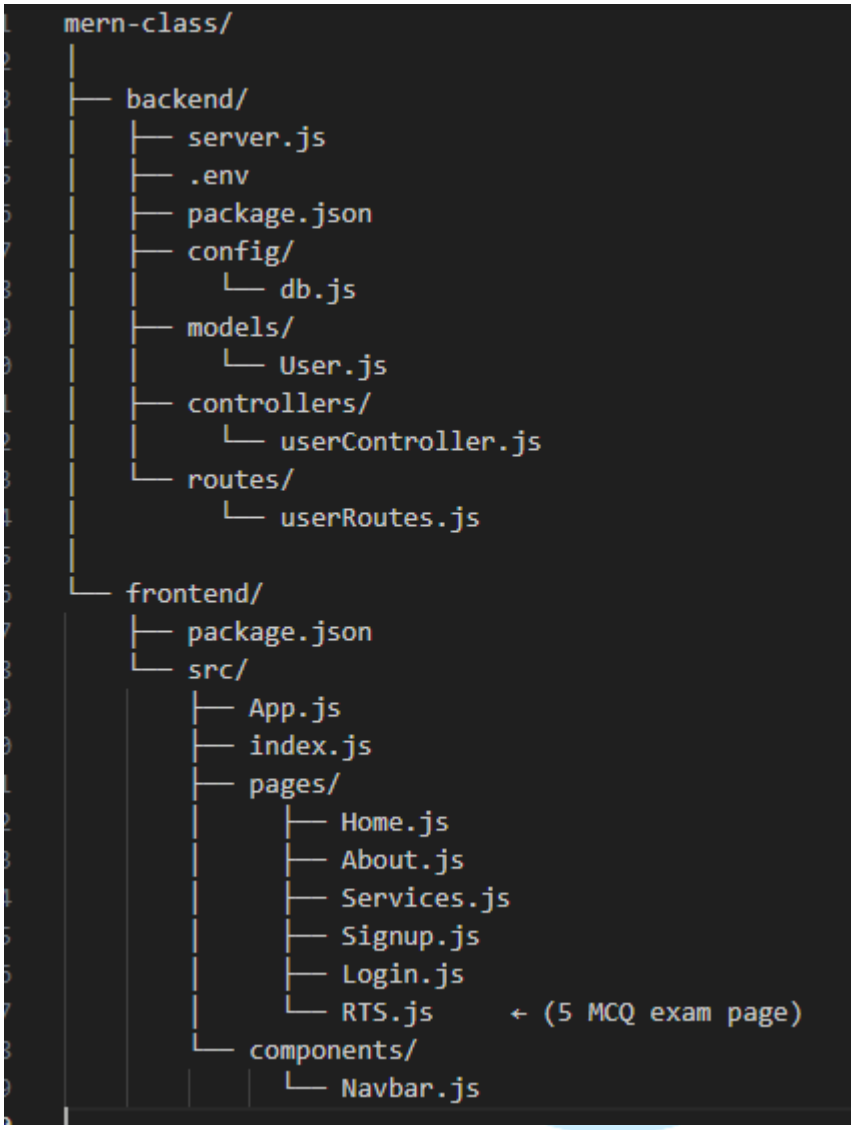
Perfect working **Exam Timer Feature** for MERN classroom teaching.

🚀 **NEXT FEATURES AVAILABLE**

Choose any:

3️⃣ Show Live Dashboard of Student Scores
4️⃣ Add JWT Login (protected exam page)
5️⃣ Add Admin Panel to Add New MCQs
6️⃣ Add Student Result History Page
7️⃣ Add Start Exam Button + Disable Refresh

Just say **"Add Feature 3"**, **"Add Feature 4"**, etc.


mern-class/
backend/
server.js
.env
package.json
config/
db.js
models/
User.js
controllers/
userController.js
routes/
userRoutes.js
frontend/
package.json
src/
App.js
index.js
pages/
Home.js
About.js
Services.js
Signup.js
Login.js
RTS.js ← (5 MCQ exam page)
components/
Navbar.js

RESEARCH INNOVATION DISCOVERY
RID
BHARAT

Import Interview Question and answer

1. What is Node.js?

Node.js is a **JavaScript runtime environment** built on Chrome's **V8 engine** that lets you run JavaScript code on the server side.

2. Why use Node.js for backend development?

Because it is **fast, scalable, non-blocking**, and supports **real-time applications** using asynchronous I/O.

3. What is the difference between JavaScript and Node.js?

- **JavaScript:** Runs in browsers (client-side).
- **Node.js:** Runs outside browsers (server-side).

4. What is the role of the V8 engine in Node.js?

V8 converts JavaScript code into **machine code**, enabling high performance and fast execution.

5. What is npm?

npm (Node Package Manager) manages dependencies (libraries) used in Node.js projects.

6. What is Express.js?

Express.js is a **web application framework for Node.js** used to build APIs and web servers easily.

7. What is middleware in Express.js?

Middleware are functions that execute **between request and response** — for logging, authentication, etc.

Example: app.use((req, res, next) => { console.log('Middleware'); next(); });

8. What are the core modules in Node.js?

Common modules: http, fs, path, url, events, os, crypto.

9. What is event-driven programming in Node.js?

Node.js uses an **event-driven, non-blocking architecture**, meaning actions trigger events handled by callbacks.

10. What is the difference between synchronous and asynchronous code?

- **Synchronous:** Executes one line at a time (blocking).
- **Asynchronous:** Doesn't block — other code runs while waiting.

11. What is a callback function?

A function passed as an argument to another function to be executed later.

12. What are Promises and async/await?

- **Promise:** Handles async tasks and returns success or error.
- **async/await:** Cleaner syntax to handle Promises.

13. How do you create a simple server in Node.js?

```
const http = require('http');
http.createServer((req, res) => res.end("Hello")).listen(3000);
```

14. What is package.json?

It contains project metadata and dependencies for a Node.js app.

15. What is the difference between require and import?

- require: CommonJS module system (Node.js default).
- import: ES6 module system (modern JS).

16. What is Nodemon?

A tool that automatically restarts your Node.js server when file changes are detected.

17. What is REST API?

REST (Representational State Transfer) is an architectural style that uses HTTP methods like GET, POST, PUT, DELETE for CRUD operations.

18. How to define routes in Express.js?

app.get('/users', (req, res) => res.send('User List'));

19. How to handle JSON data in Express.js?

Use built-in middleware:

app.use(express.json());

20. What is the difference between res.send() and res.json()?

- res.send(): Sends any type of response (string, object, etc.)
- res.json(): Sends a JSON response.

21. How to serve static files in Express.js?

app.use(express.static('public'));

Used to serve HTML, CSS, JS, images, etc.

22. How do you handle errors in Express.js?

Use error-handling middleware:

app.use((err, req, res, next) => res.status(500).send(err.message));

23. How to connect MongoDB with Node.js using Mongoose?

const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/mydb');

24. What is Mongoose?

Mongoose is an **ODM (Object Data Modeling)** library for MongoDB and Node.js — helps define schemas and interact with MongoDB easily.

25. How to secure passwords in Node.js?

Use **bcrypt** to hash passwords:

const hash = await bcrypt.hash(password, 10);

26. What is JWT (JSON Web Token)?

JWT is used for **authentication** — it securely transmits user data between client and server.

27. How do you handle CORS in Express.js?

Use CORS middleware:

const cors = require('cors');

app.use(cors());

28. What is process.nextTick()?

It defers the execution of a callback until the next iteration of the event loop.

29. How to handle file uploads in Node.js?

Use **Multer** middleware:

const multer = require('multer');

const upload = multer({ dest: 'uploads/' });

app.post('/upload', upload.single('file'), (req, res) => res.send('Uploaded'));

30. What is MVC architecture in Express.js?

- **Model:** Handles data & database logic
- **View:** UI templates (EJS, Pug)
- **Controller:** Business logic & routing

31. What is the Event Loop in Node.js?

The **event loop** handles all asynchronous callbacks. It allows Node.js to perform non-blocking I/O operations even though JavaScript is single-threaded.

32. What are Streams in Node.js?

Streams are used to **read/write data continuously** (like handling large files).

Types: Readable, Writable, Duplex, Transform.

33. What is the difference between readFile and createReadStream?

- readFile: Loads the entire file into memory.
- createReadStream: Reads file chunk by chunk (memory efficient).

34. How does Node.js handle child processes?
Using the child_process module:
const { exec } = require('child_process');
exec('dir', (err, stdout) => console.log(stdout));
35. What are environment variables in Node.js?
Used to store sensitive information like API keys or DB credentials.
Example:process.env.PORT = 3000;
36. How to manage environment variables securely?
Use **dotenv**:
require('dotenv').config();
const port = process.env.PORT;
37. What is clustering in Node.js?
Clustering allows Node.js to use **multiple CPU cores** for better performance using the cluster module.
38. What is CORS and why is it used?
CORS (Cross-Origin Resource Sharing) allows browsers to request resources from different origins (domains).
Handled via: app.use(require('cors')());
39. Difference between PUT and PATCH in REST API?

- **PUT:** Replaces the entire resource.
- **PATCH:** Updates only specific fields.

40. What is rate limiting and why use it?
To prevent API abuse by limiting requests per user/IP using packages like **express-rate-limit**.
41. How do you handle authentication in Express.js?
Using **JWT** or **Passport.js** for session or token-based authentication.
42. How to implement session management?
Using **express-session**:
const session = require('express-session');
app.use(session({ secret: 'secretKey', resave: false, saveUninitialized: true }));
43. What are cookies and how to set them in Express.js?
Cookies store small data on the client side.
res.cookie('username', 'Tanmay', { maxAge: 60000 });
44. What are WebSockets in Node.js?
Used for **real-time, two-way communication** (e.g., chat apps) using the ws or socket.io library.
45. What are CommonJS and ES Modules?

- **CommonJS:** Uses require() and module.exports.
- **ES Modules:** Uses import and export.

46. How to handle 404 errors in Express.js?
app.use((req, res) => res.status(404).send('Page Not Found'));
47. What is the difference between res.sendFile() and res.render()?

- **res.sendFile():** Sends static files.
- **res.render():** Renders templates (EJS, Pug, etc.) with dynamic data.

48. How to validate user input in Node.js?
Use validation libraries like **Joi** or **express-validator**:
const { body } = require('express-validator');
app.post('/user', body('email').isEmail(), (req,res)=>{...});
49. What is the purpose of app.use() in Express.js?
It mounts middleware or routes at the application level — runs for every request that matches the path.

What is RID Organization (RID संस्था क्या है)?

- **RID Organization** यानि **Research, Innovation and Discovery Organization एक** संस्था हैं जो TWF (TWKSAA WELFARE FOUNDATION) **NGO** द्वारा RUN किया जाता है | जिसका मुख्य उदेश्य हैं आने वाले समय में सबसे पहले **NEW (RID, PMS & TLR)** की खोज, प्रकाशन एवं उपयोग भारत की इस पावन धरती से भारतीय संस्कृति, सभ्यता एवं भाषा में ही हो |
- देश, समाज, एवं लोगों की समस्याओं का समाधान **NEW (RID, PMS & TLR)** के माध्यम से किया जाये इसके लिए ही **इस RID Organization** की स्थपना 30.09.2023 किया गया है | जो TWF द्वारा संचालित किया जाता है |
- TWF (TWKSAA WELFARE FOUNDATION) NGO की स्थपना 26-10-2020 में बिहार की पावन धरती सासाराम में Er. RAJESH PRASAD एवं Er. SUNIL KUMAR द्वारा किया गया था जो की भारत सरकार द्वारा मान्यता प्राप्त संस्था हैं |
- Research, Innovation & Discovery में रूचि रखने वाले आप सभी विधार्थियों, शिक्षकों एवं बुधीजिवियो से मैं आवाहनं करता हूँ की आप सभी **इस RID संस्था** से जुड़ें एवं अपने बुध्दि, विवेक एवं प्रतिभा से दुनियां को कुछ नई **(RID, PMS & TLR)** की खोजकर, बनाकर एवं अपनाकर लोगों की समस्याओं का समाधान करें |

MISSION, VISSION & MOTIVE OF "RID ORGANIZATION"	
मिशन	हर एक ONE भारत के संग
विजन	TALENT WORLD KA SHRESHTM AB AAYEGA भारत में और भारत का TALENT भारत में
मकसद	NEW (RID, PMS, TLR)

MOTIVE OF RID ORGANIZATION NEW (RID, PMS, TLR)		
NEW (RID)		
R	I	D
Research	Innovation	Discovery
NEW (TLR)		
T	L	R
Technology, Theory, Technique	Law	Rule
NEW (PMS)		
P	M	S
Product, Project, Production	Machine	Service

RID रीड संस्था की मिशन, विजन एवं मकसद को सार्थक हमें बनाना हैं |
भारत के वर्चस्व को हर कोने में फैलना हैं |
कर के नया कार्य एक बदलाव समाज में लाना हैं |
रीड संस्था की कार्य-सिध्दांतों से ही, हमें अपनी पहचान बनाना हैं |

Er. Rajesh Prasad (B.E, M.E)
Founder:
TWF & RID Organization