Introduction to Node.js and Express.js

1. What is Node.js?

Definition

Node.js is an **open-source, server-side JavaScript runtime** that allows developers to run JavaScript outside the browser. It is built on Chrome's **V8 JavaScript engine** and is used for building scalable, high-performance applications.

Key Features of Node.js

- **Asynchronous & Non-blocking I/O**: Handles multiple requests efficiently.
- **Single-threaded Event Loop**: Uses an event-driven architecture for scalability.
- **Fast Execution**: Powered by Google's V8 engine.
- **Cross-platform**: Works on Windows, macOS, and Linux.
- **Rich Package Ecosystem**: Uses npm (Node Package Manager) with over a million libraries.

Installing Node.js

To install Node.js, download it from the official site: https://nodejs.org/

Verify the installation:

```
node -v   # Check Node.js version
npm -v    # Check npm version
```

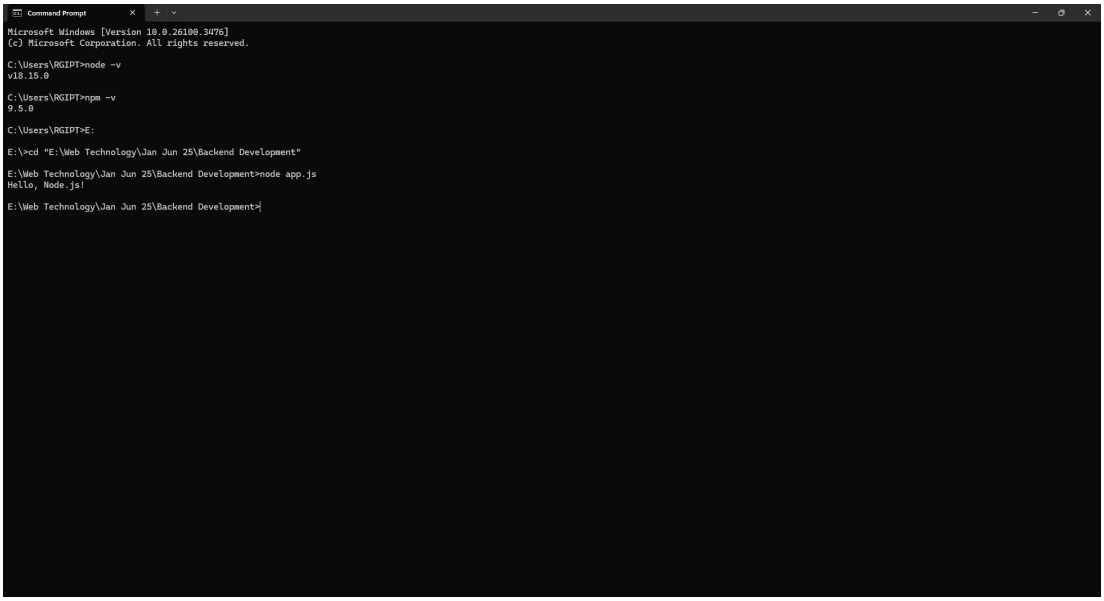
Running a Simple Node.js Script

Create a file `app.js`:

```
console.log("Hello, Node.js!");
```

Run the script:

```
node app.js
```



2. What is Express.js?

Definition

Express.js is a **minimal and flexible Node.js web framework** that simplifies building web applications and APIs. It provides:

- **Routing**: Handling different URLs.
- **Middleware support**: Enhancing request/response handling.
- **Template engines**: Rendering dynamic HTML pages.

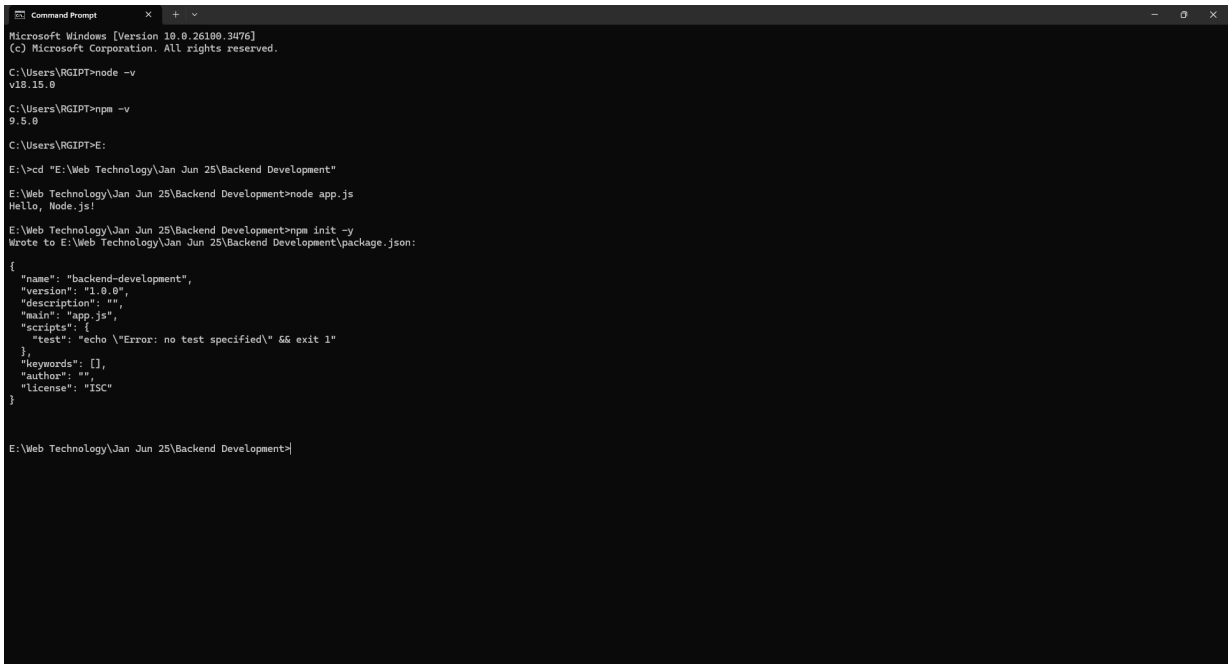
Why Use Express.js?

- **Simplifies Node.js development**
- **Handles HTTP requests easily**
- **Supports middleware for modular code**
- **Integrates with databases like MongoDB and PostgreSQL**

Installing Express.js

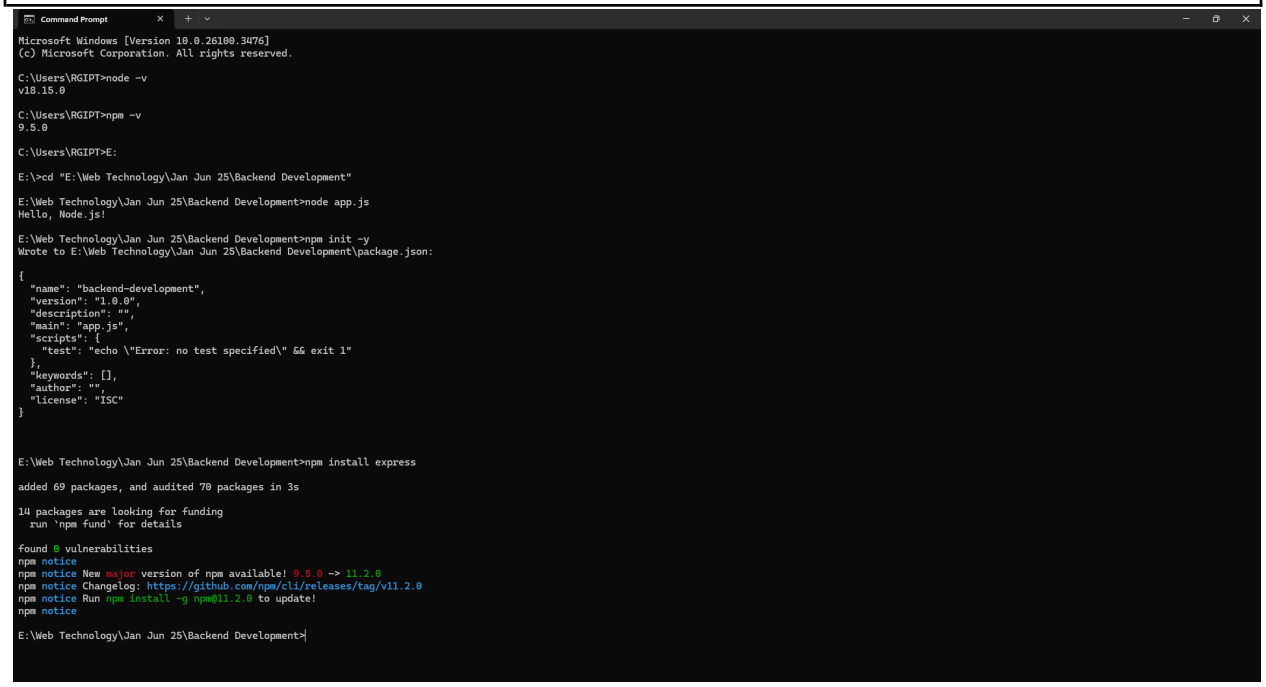
First, initialize a Node.js project:

```
npm init -y   # Creates a package.json file
```



Then, install Express:

```
npm install express
```



---

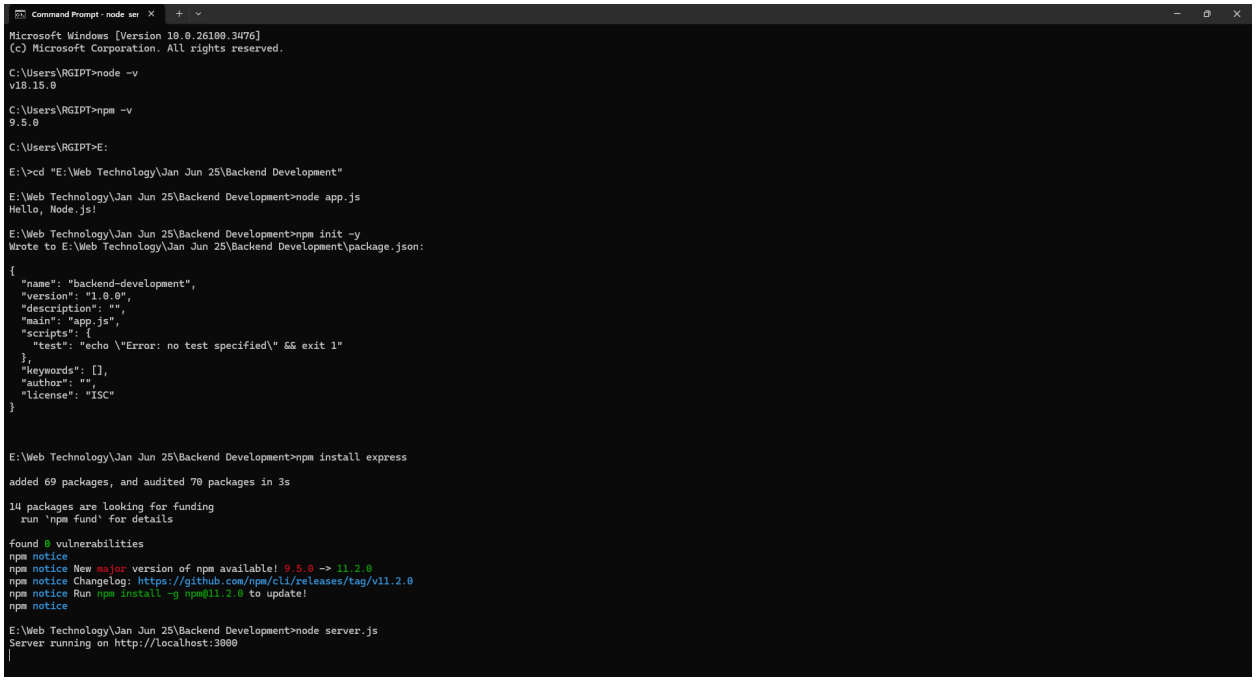
3. Creating a Basic Express Server

Create a file `server.js`:

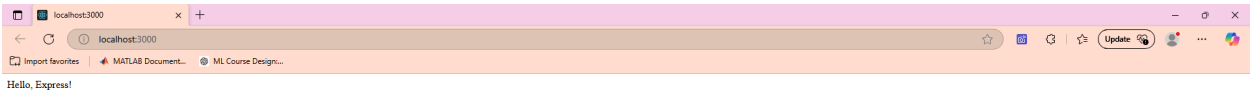
```
const express = require("express");
const app = express();

app.get("/", (req, res) => {
   res.send("Hello, Express!");
});

app.listen(3000, () => {
   console.log("Server running on http://localhost:3000");
});
```

Run the server:

```
node server.js
```



Go to `http://localhost:3000/` in your browser to see the response.



---

4. Express.js Routing

Routing allows different URLs to trigger specific responses.

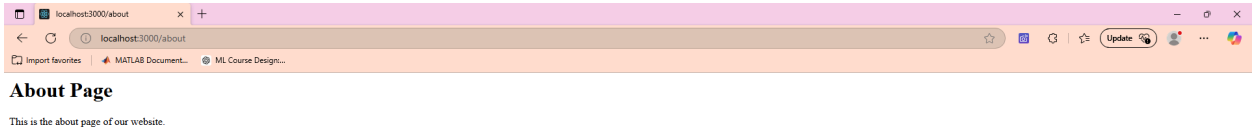
Example of Express Routes:

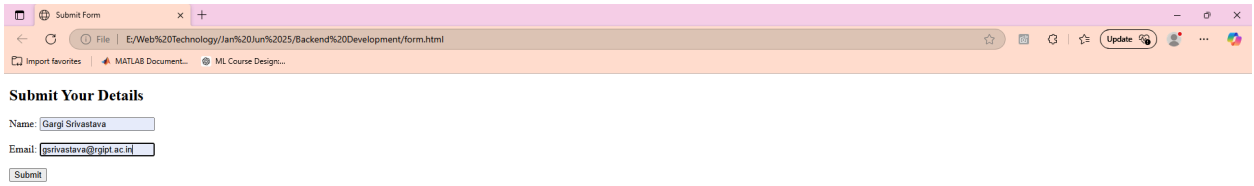
```
// server.js
const express = require("express");
const app = express();

// Middleware to parse form data
app.use(express.urlencoded({ extended: true }));

// Route for About Page (GET request)
app.get("/about", (req, res) => {
   res.send("<h1>About Page</h1><p>This is the about page of our website.</p>");
});

// Route to handle form submission (POST request)
app.post("/submit", (req, res) => {
```

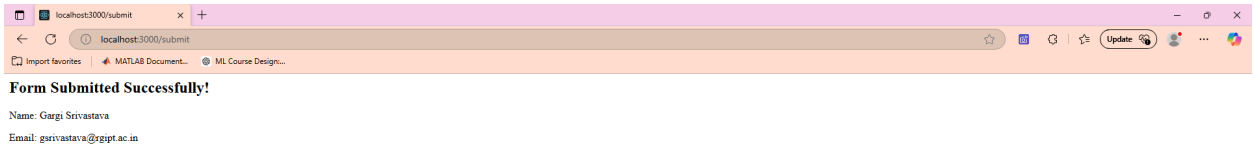
```
  console.log("Form Data Received:", req.body);
  res.send(`<h2>Form Submitted Successfully!</h2><p>Name: ${req.body.name}</p><p>Email:
${req.body.email}</p>`);
});

// Start the server
app.listen(3000, () => {
  console.log("Server running on http://localhost:3000");
});
```

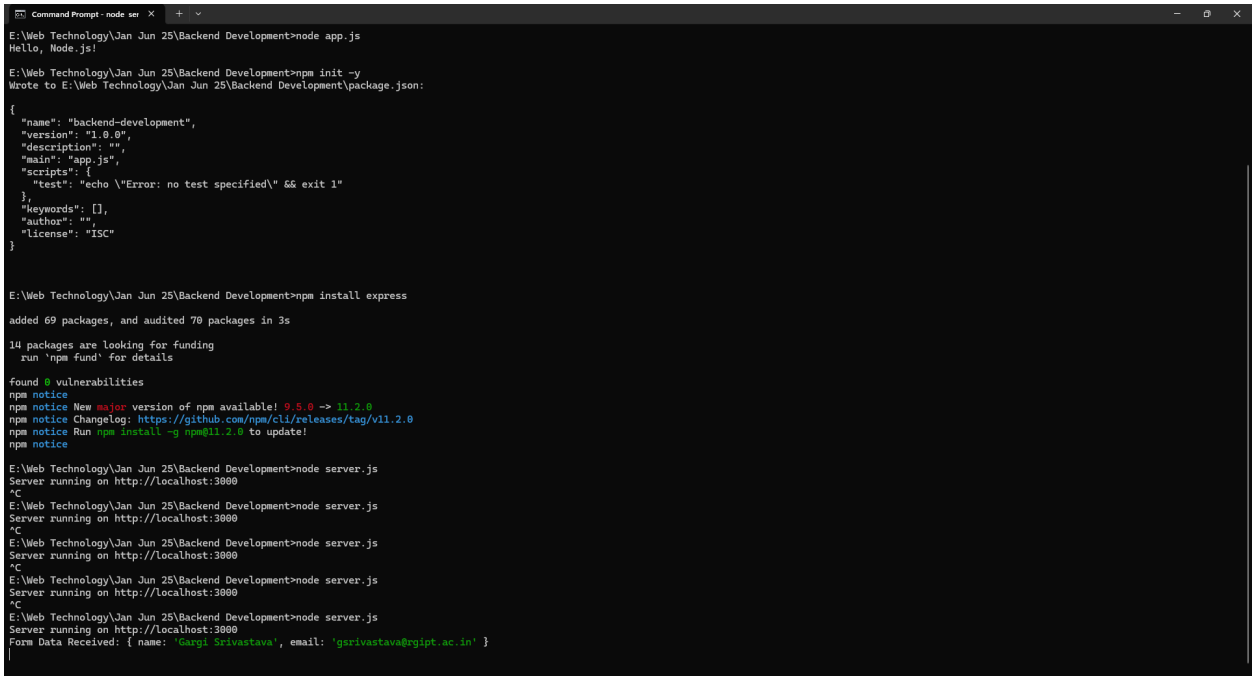
form.html

```
<h2>Submit Your Details</h2>

<form action="http://localhost:3000/submit" method="post">

    <label>Name: <input type="text" name="name"></label><br><br>

    <label>Email: <input type="email" name="email"></label><br><br>

    <button type="submit">Submit</button>

</form>
```




Submit Form
File | E:/Web%20Technology//an%20Jun%2025/Backend%20Development/form.html
Update
Import favorites
MATLAB Document...
ML Course Design:...
Submit Your Details
Name: Gargi Srivastava
Email: gsrivastava@rgipt.ac.in
Submit


localhost:3000/submit
localhost:3000/submit
Update
Import favorites
MATLAB Document...
ML Course Design:...
Form Submitted Successfully!
Name: Gargi Srivastava
Email: gsrivastava@rgipt.ac.in



📌 **GET for retrieving data, POST for sending data.**

---

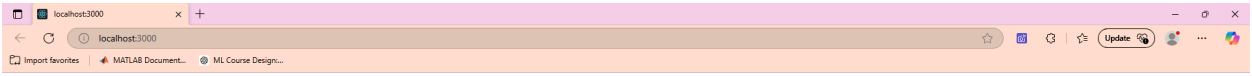
5. Middleware in Express.js

Middleware functions modify requests before they reach the final route handler.

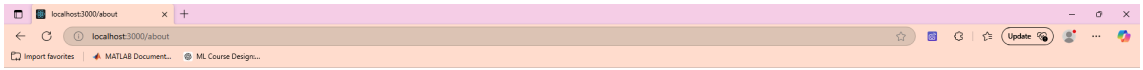
Example: Logging Middleware

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

// Middleware to log incoming requests
app.use((req, res, next) => {
  console.log(`${req.method} request to ${req.url}`);
  next(); // Pass control to the next middleware or route handler
});

// Sample GET route
app.get("/", (req, res) => {
  res.send("<h1>Welcome to the Home Page</h1>");
```

```
});

// About route
app.get("/about", (req, res) => {
  res.send("<h1>About Page</h1><p>This is the about section.</p>");
});

// Contact route
app.get("/contact", (req, res) => {
  res.send("<h1>Contact Page</h1><p>Email us at contact@example.com</p>");
});

// Start the Express server
app.listen(3000, () => {
  console.log("Server running on http://localhost:3000");
});
```

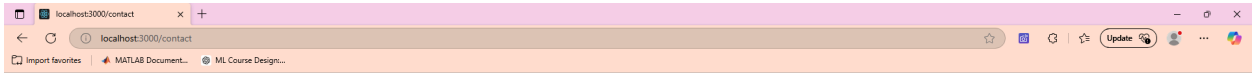


Welcome to the Home Page



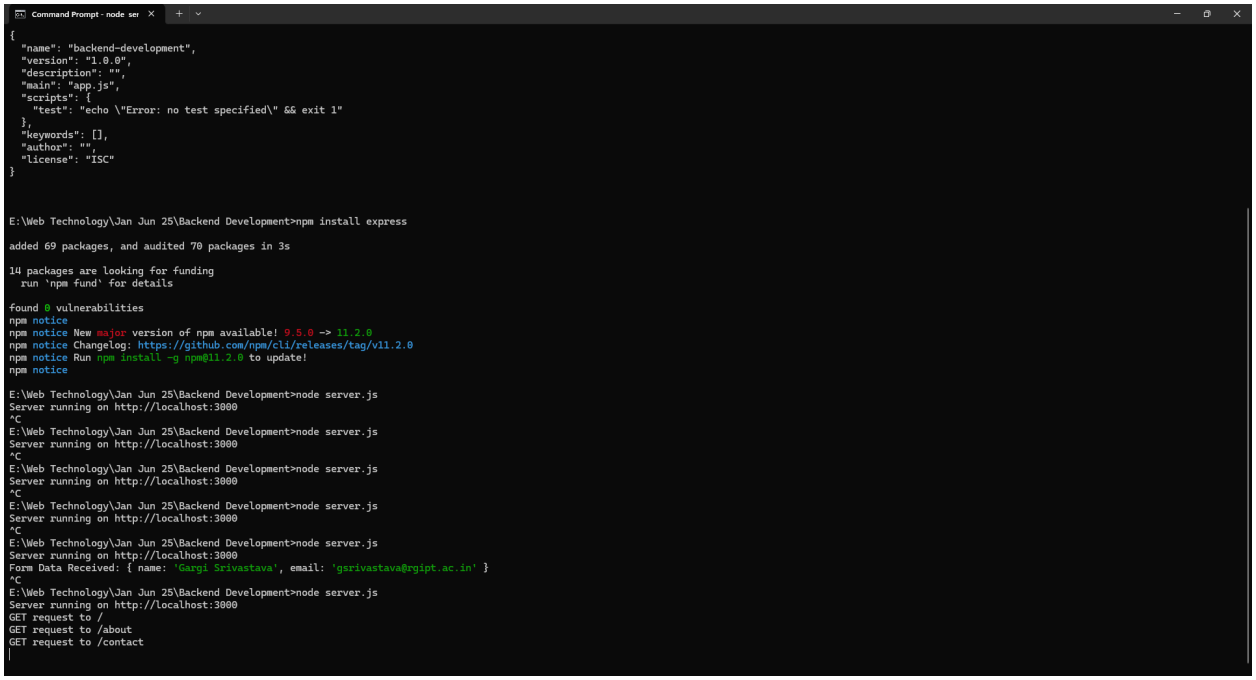
About Page

This is the about section.



Contact Page

Email us at contact@example.com



📌 **Use middleware for authentication, logging, and error handling.**

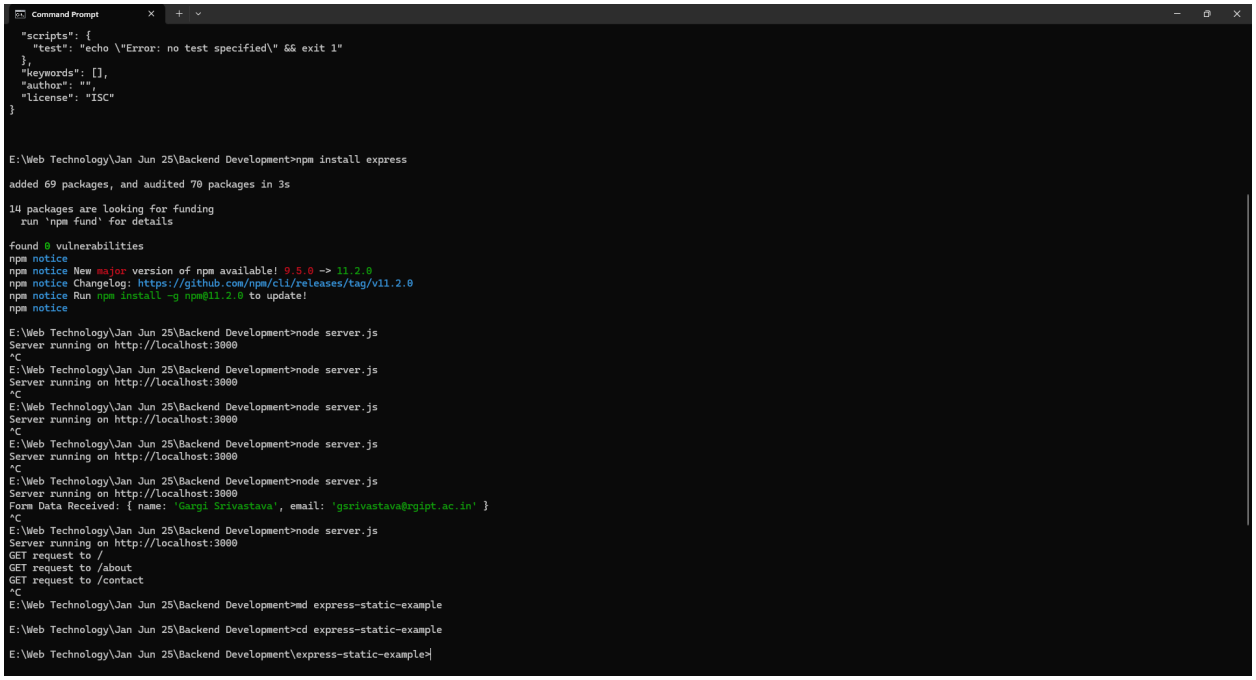
---

6. Serving Static Files

Express can serve HTML, CSS, and images.

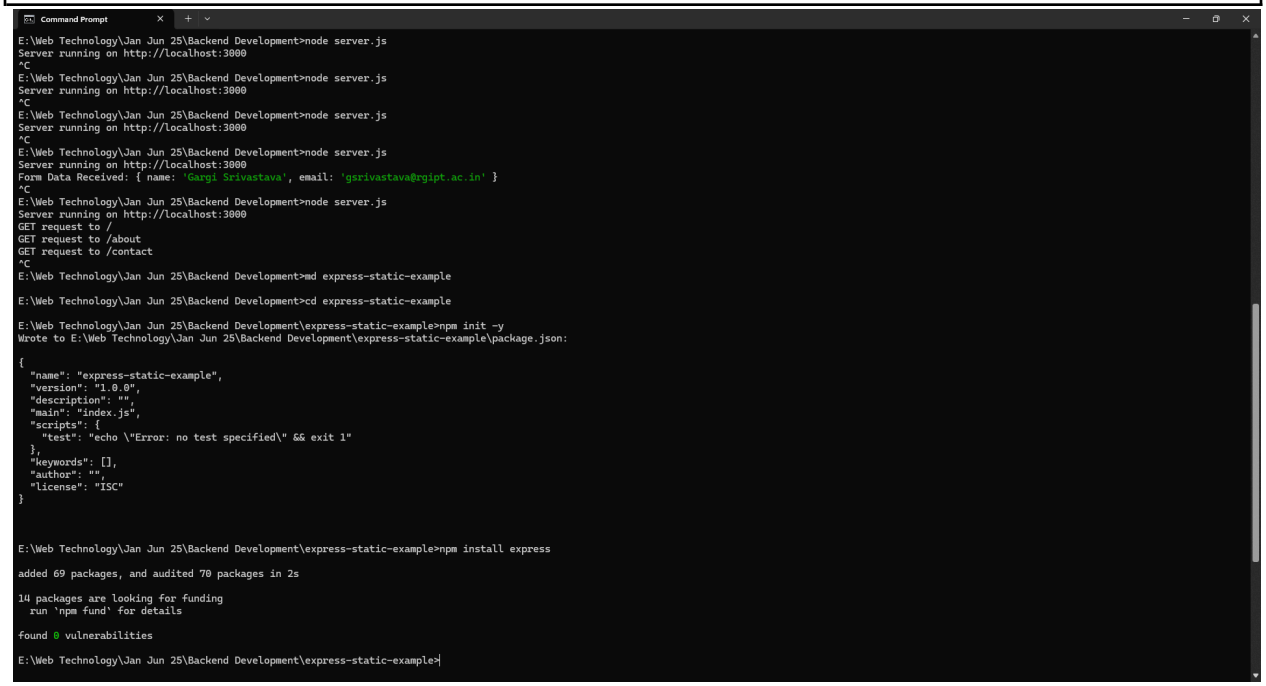
Example: Serving Static Files

```
md express-static-example
cd express-static-example
```



npm init -y

npm install express



md public

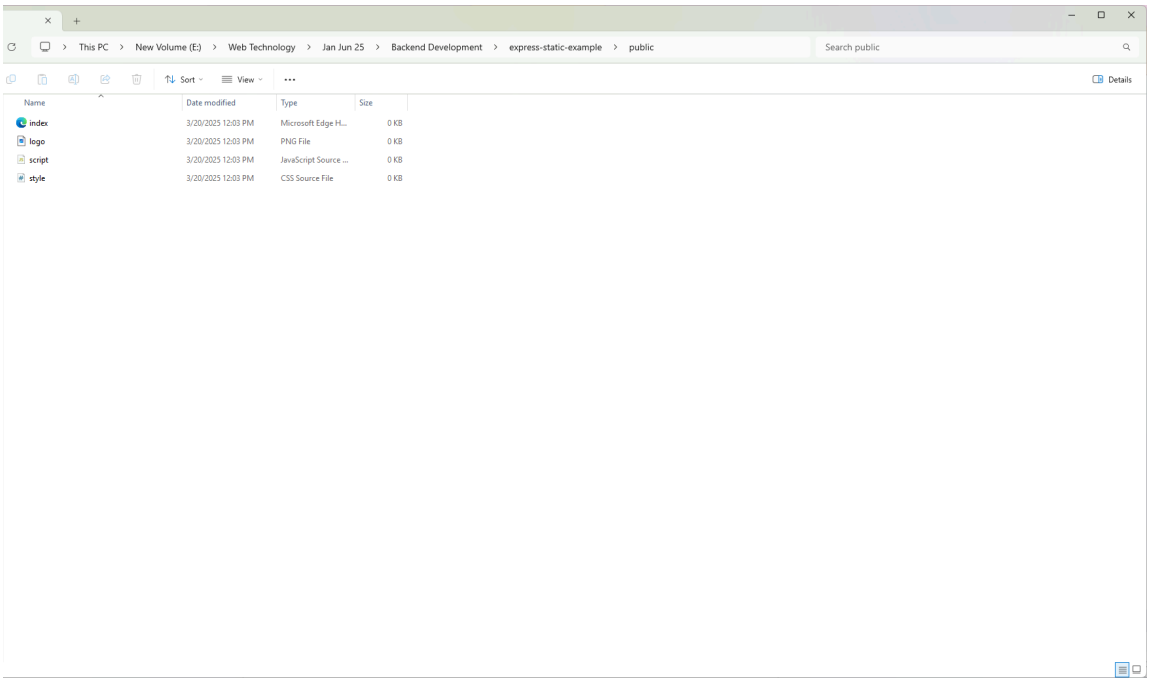
wsl

wsl --install -d Ubuntu-24.04

wsl

touch server.js public/index.html public/style.css public/script.js public/logo.png

```
rgipt@DESKTOP-5H5KH95: /n
Microsoft Windows [Version 10.0.26100.3476]
(c) Microsoft Corporation. All rights reserved.

C:\Users\RGIPT>E:

E:\>cd "E:\Web Technology\Jan Jun 25\Backend Development\express-static-example"

E:\Web Technology\Jan Jun 25\Backend Development\express-static-example>touch
'touch' is not recognized as an internal or external command,
operable program or batch file.

E:\Web Technology\Jan Jun 25\Backend Development\express-static-example>wsl touch
Windows Subsystem for Linux has no installed distributions.
You can resolve this by installing a distribution with the instructions below:

Use 'wsl.exe --list --online' to list available distributions
and 'wsl.exe --install <Distro>' to install.

E:\Web Technology\Jan Jun 25\Backend Development\express-static-example>wsl --install -d Ubuntu-24.04
Downloading: Ubuntu 24.04 LTS
Installing: Ubuntu 24.04 LTS
Distribution successfully installed. It can be launched via 'wsl.exe -d Ubuntu-24.04'

E:\Web Technology\Jan Jun 25\Backend Development\express-static-example>wsl
Provisioning the new WSL instance Ubuntu-24.04
This might take a while...
Create a default Unix user account: rgipt
New password:
Retype new password:
passwd: password updated successfully
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

Welcome to Ubuntu 24.04.2 LTS (GNU/Linux 5.15.167.4-microsoft-standard-WSL2 x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

 System information as of Thu Mar 20 06:31:48 UTC 2025

  System load:  0.12               Processes:             32
  Usage of /:   0.1% of 1006.85GB  Users logged in:       0
  Memory usage: 3%                 IPv4 address for eth0: 172.30.180.4
  Swap usage:   0%


This message is shown once a day. To disable it please create the
/home/rgipt/.hushlogin file.
rgipt@DESKTOP-5H5KH95:/mnt/e/Web Technology/Jan Jun 25/Backend Development/express-static-example$ touch server.js publi
rgipt@DESKTOP-5H5KH95:/mnt/e/Web Technology/Jan Jun 25/Backend Development/express-static-example$ touch server.js public/index.html public/style.css public/script.js public/logo.png
```



server.js

```
const express = require("express");
const app = express();
const port = 3000;

// Serve static files from the "public" directory
```

```
app.use(express.static("public"));

app.get("/", (req, res) => {
  res.sendFile(__dirname + "/public/index.html"); // Serve HTML file
});

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

public/index.html

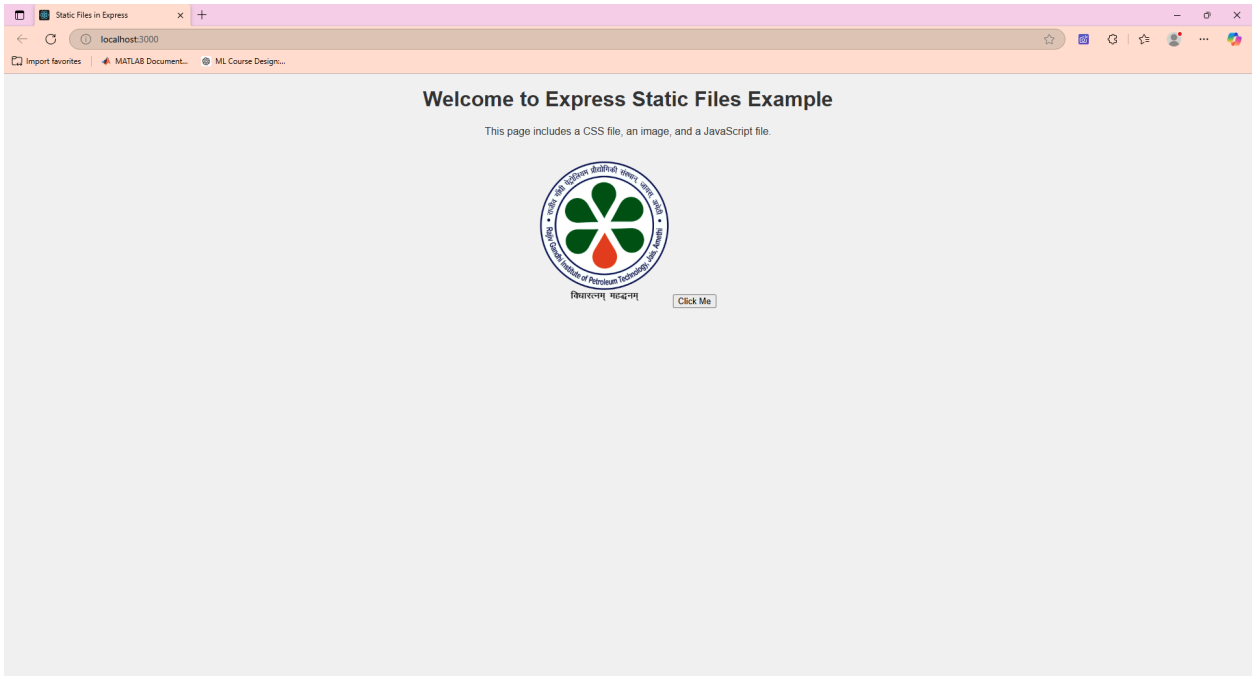
```
<head>
    <link rel="stylesheet" href="style.css">
</head>
<body>
  <h1>Welcome to Express Static Files Example</h1>
  <p>This page includes a CSS file, an image, and a JavaScript file.</p>
  <img src="logo.png" alt="Logo" width="200">
  <button onclick="showAlert()">Click Me</button>
  <script src="script.js"></script>
</body>
```

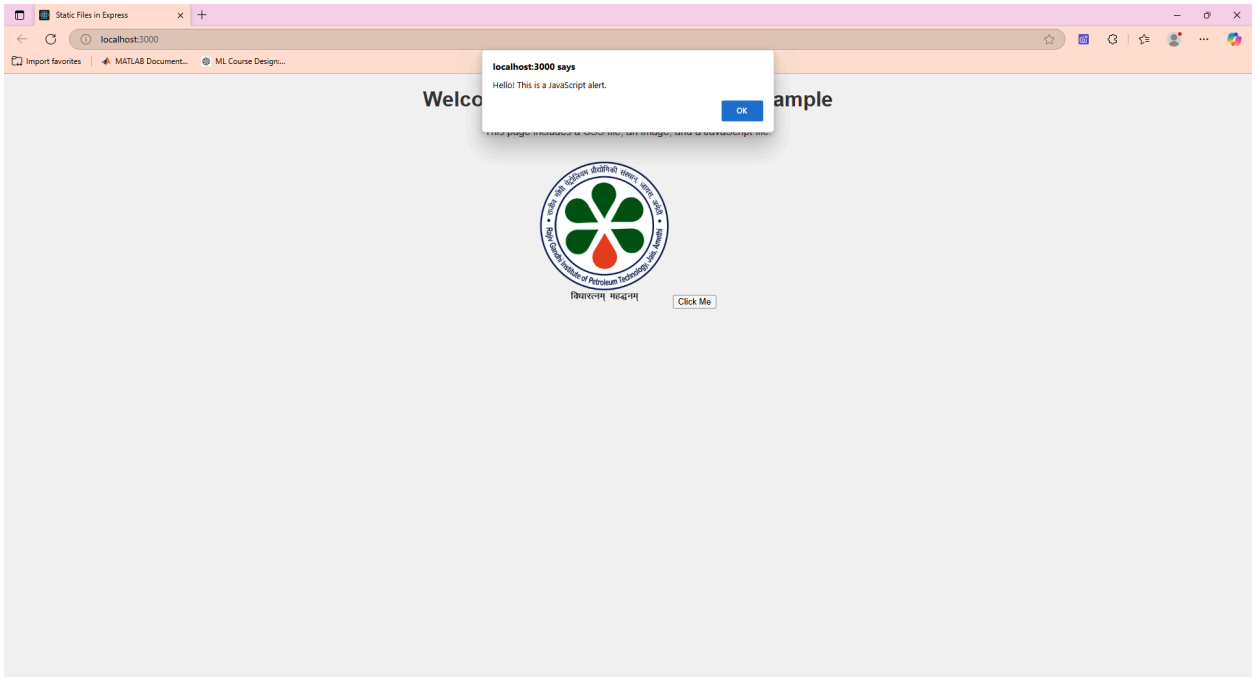
public/style.css

```
body {
  font-family: Arial, sans-serif;
  text-align: center;
  background-color: #f4f4f4;
  color: #333;
}
img {
  margin-top: 20px;
}
```

public/script.js

```
function showAlert() {
  alert("Hello! This is a JavaScript alert.");
}
```





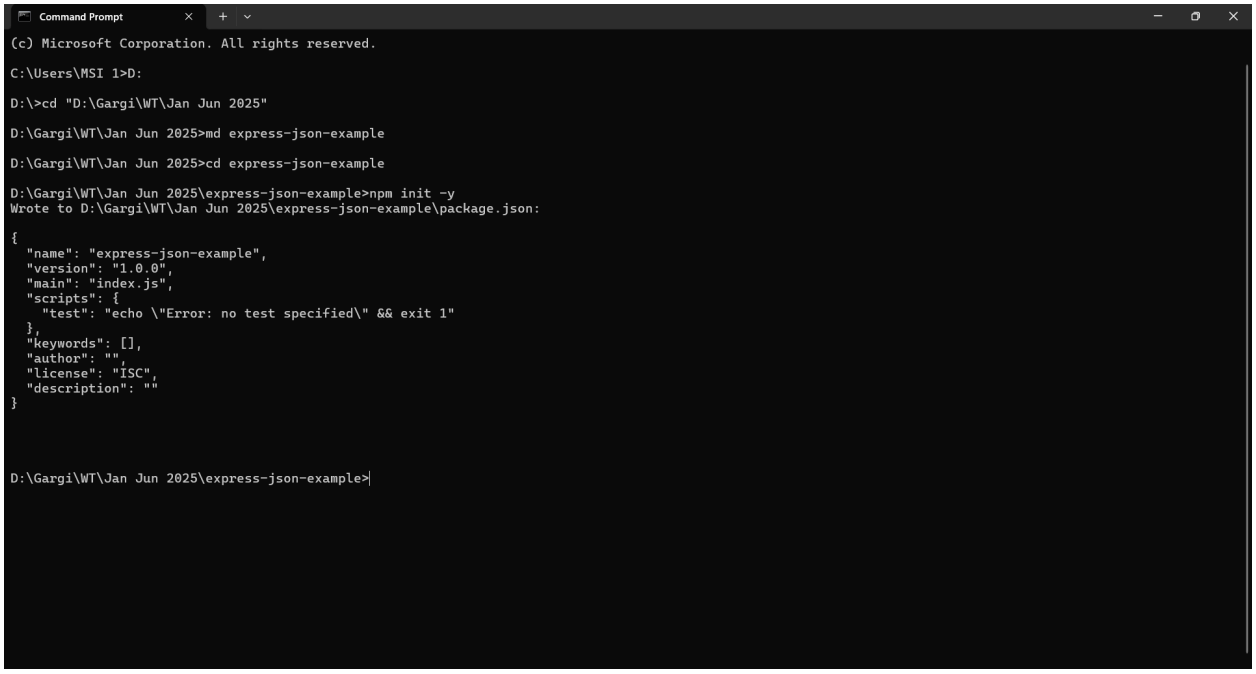
Place HTML/CSS files in the `public/` folder, and they'll be accessible at `http://localhost:3000/`.

---

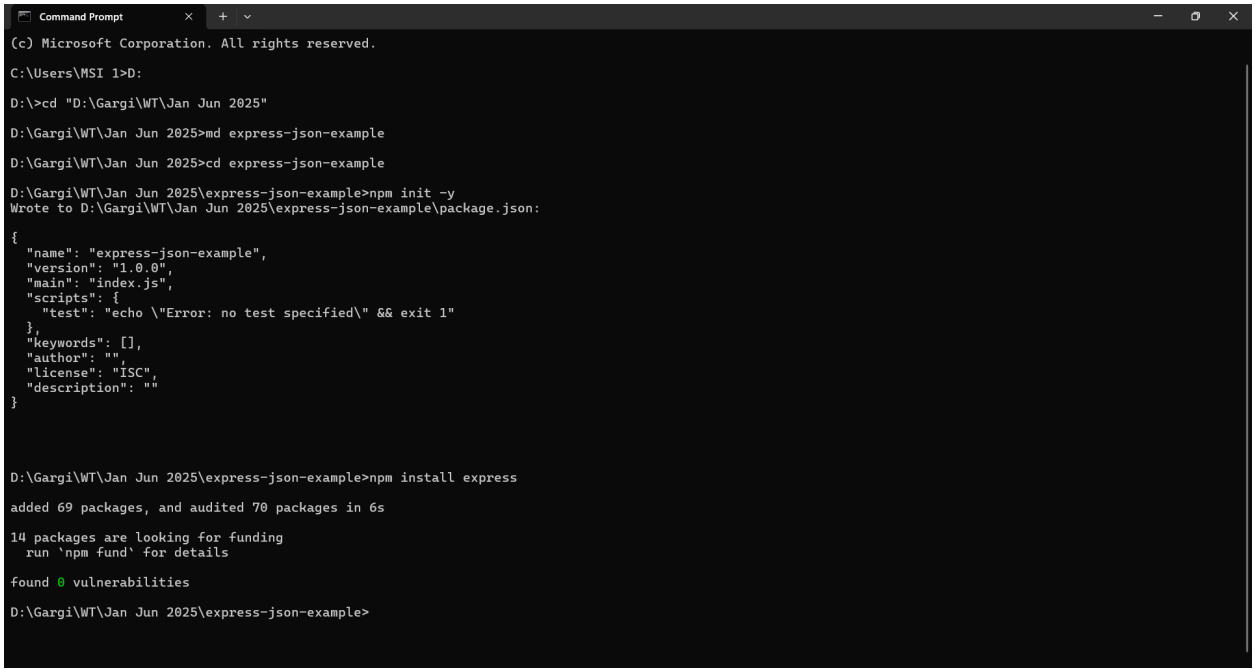
7. Handling JSON

Express can process JSON data.

```
md express-json-example

cd express-json-example

npm init -y
```

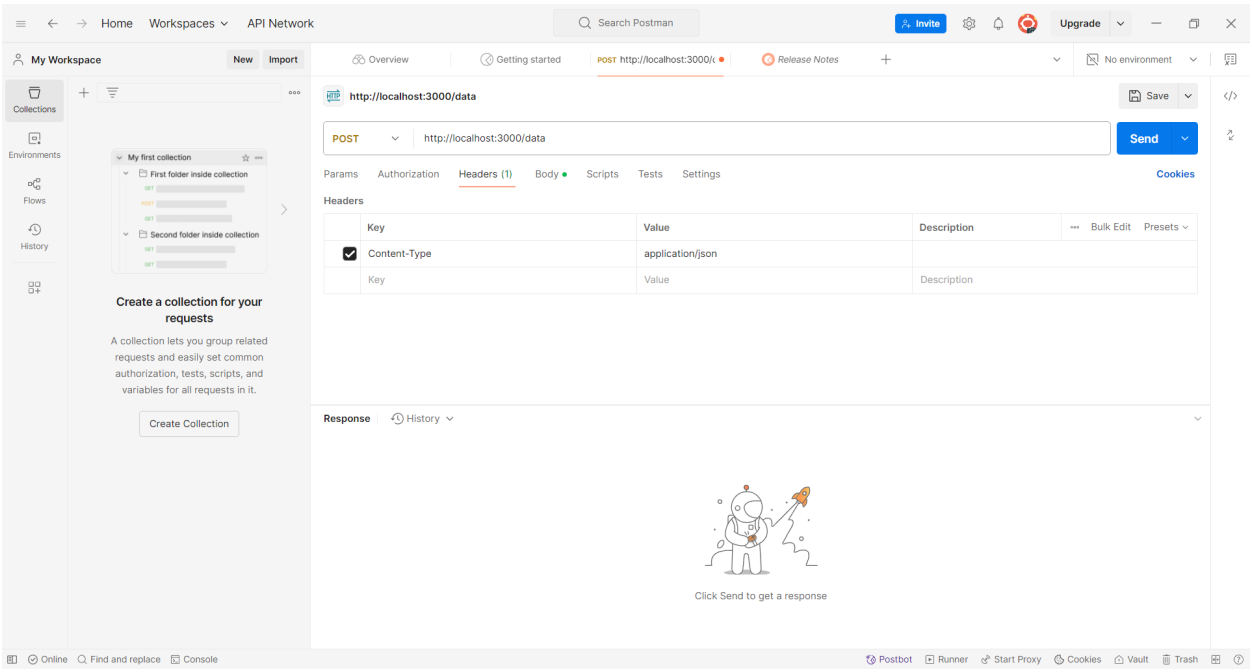


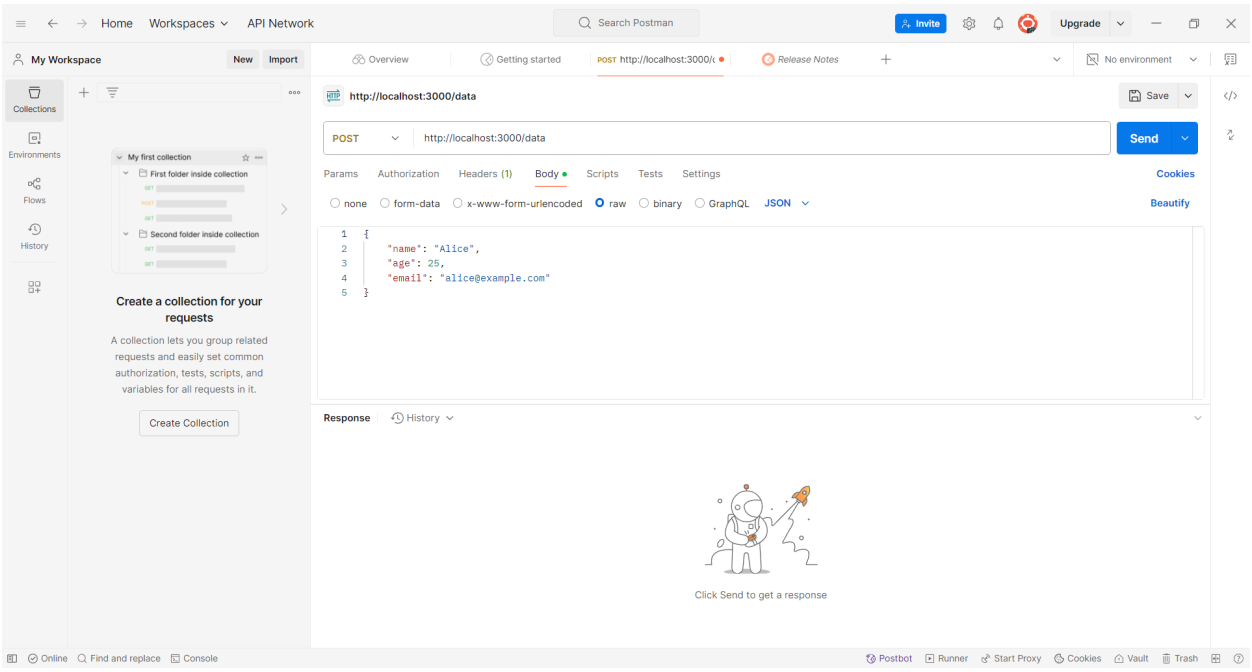

```
npm install express
```

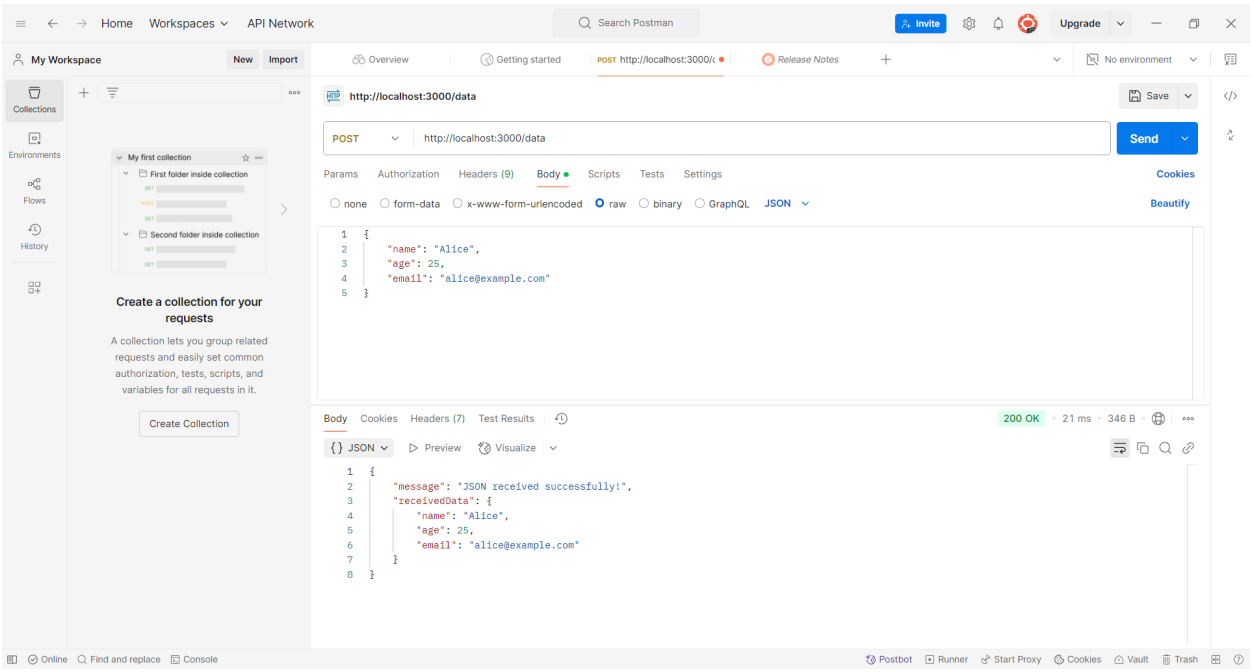


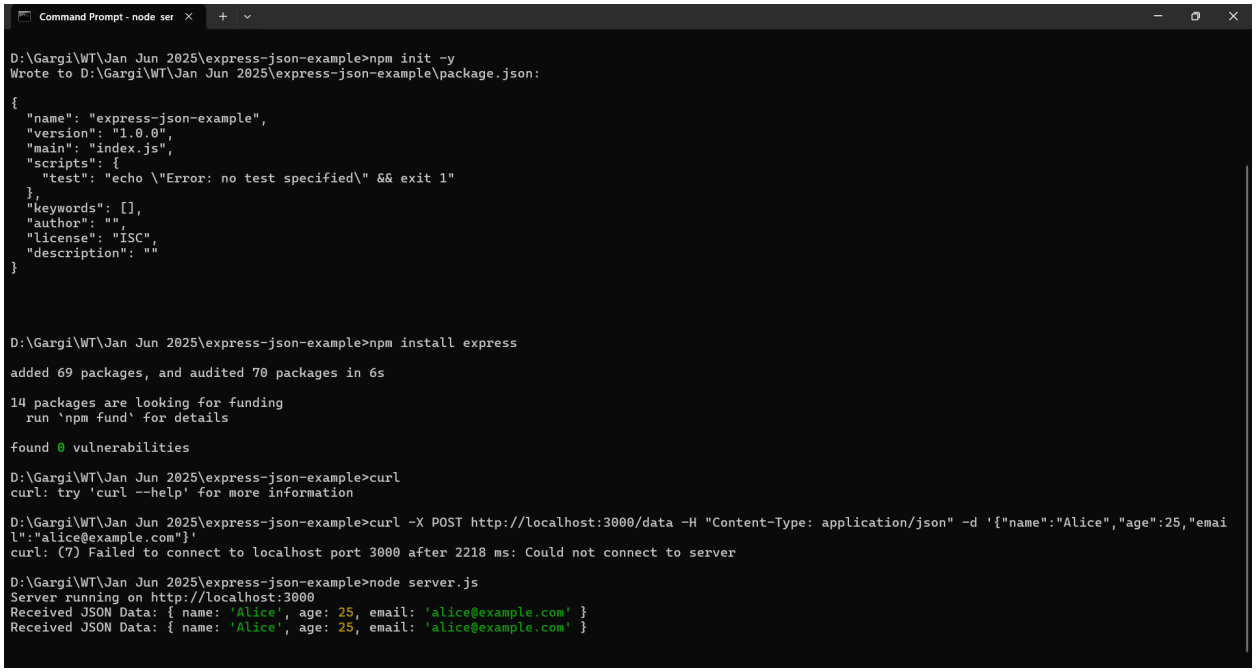

```
const express = require("express");

const app = express();

const port = 3000;



// Middleware to parse JSON request body

app.use(express.json());



// Route to handle a JSON POST request

app.post("/data", (req, res) => {

  console.log("Received JSON Data:", req.body);



  // Responding with received data

  res.json({
```

```
    message: "JSON received successfully!",
    receivedData: req.body
  });
});


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




Home
Workspaces
API Network
Search Postman
Invite
Upgrade
My Workspace
New
Import
Overview
Getting started
POST http://localhost:3000/(
Release Notes
No environment
Collections
Environments
Flows
History
http://localhost:3000/data
Save
POST
http://localhost:3000/data
Send
Params
Authorization
Headers (1)
Body
Scripts
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"name": "Alice",
"age": 25,
"email": "alice@example.com"
}
Response
History
Click Send to get a response
Create a collection for your requests
A collection lets you group related requests and easily set common authorization, tests, scripts, and variables for all requests in it.
Create Collection
Online
Find and replace
Console
Postbot
Runner
Start Proxy
Cookies
Vault
Trash


Home
Workspaces
API Network
Search Postman
Invite
Upgrade
My Workspace
New
Import
Overview
Getting started
POST http://localhost:3000/(
Release Notes
No environment
Collections
Environments
Flows
History
http://localhost:3000/data
Save
POST
http://localhost:3000/data
Send
Params
Authorization
Headers (9)
Body
Scripts
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"name": "Alice",
"age": 25,
"email": "alice@example.com"
}
Body
Cookies
Headers (7)
Test Results
200 OK
21 ms
346 B
{ } JSON
Preview
Visualize
{
"message": "JSON received successfully!",
"receivedData": {
"name": "Alice",
"age": 25,
"email": "alice@example.com"
}
}
Create a collection for your requests
A collection lets you group related requests and easily set common authorization, tests, scripts, and variables for all requests in it.
Create Collection
Online
Find and replace
Console
Postbot
Runner
Start Proxy
Cookies
Vault
Trash



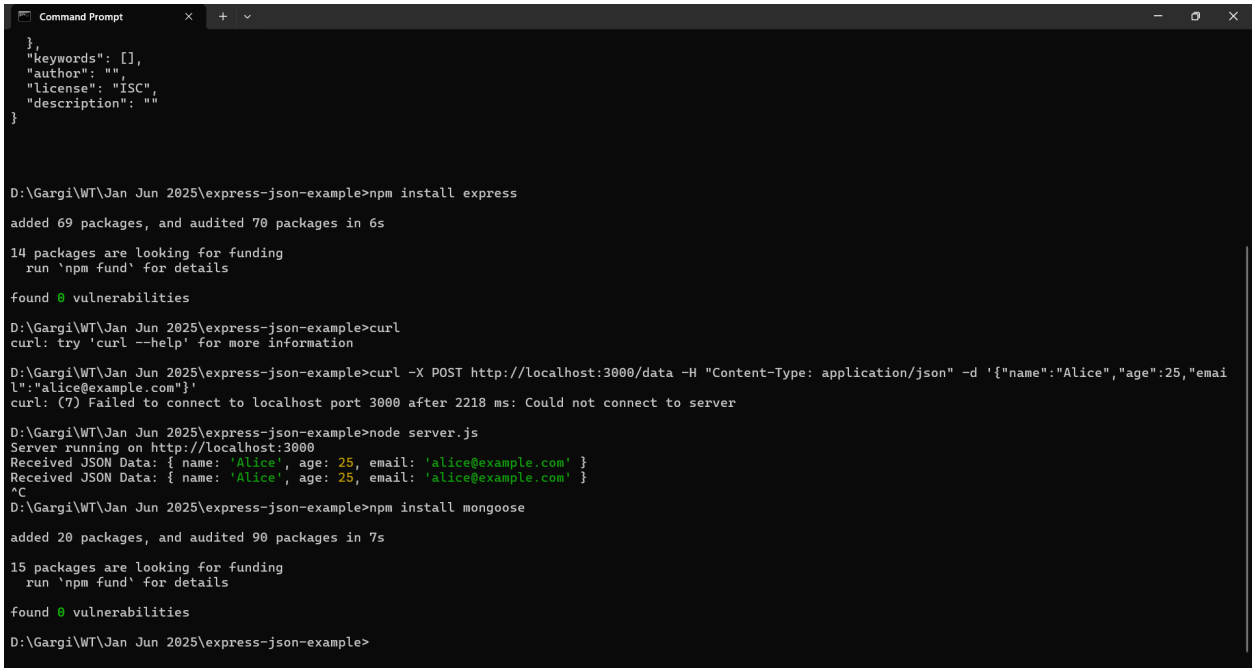
📌 **Use `express.json()` for JSON data.**

---

8. Connecting to a Database (MongoDB Example)

Install MongoDB and Mongoose:

```
npm install mongoose
```

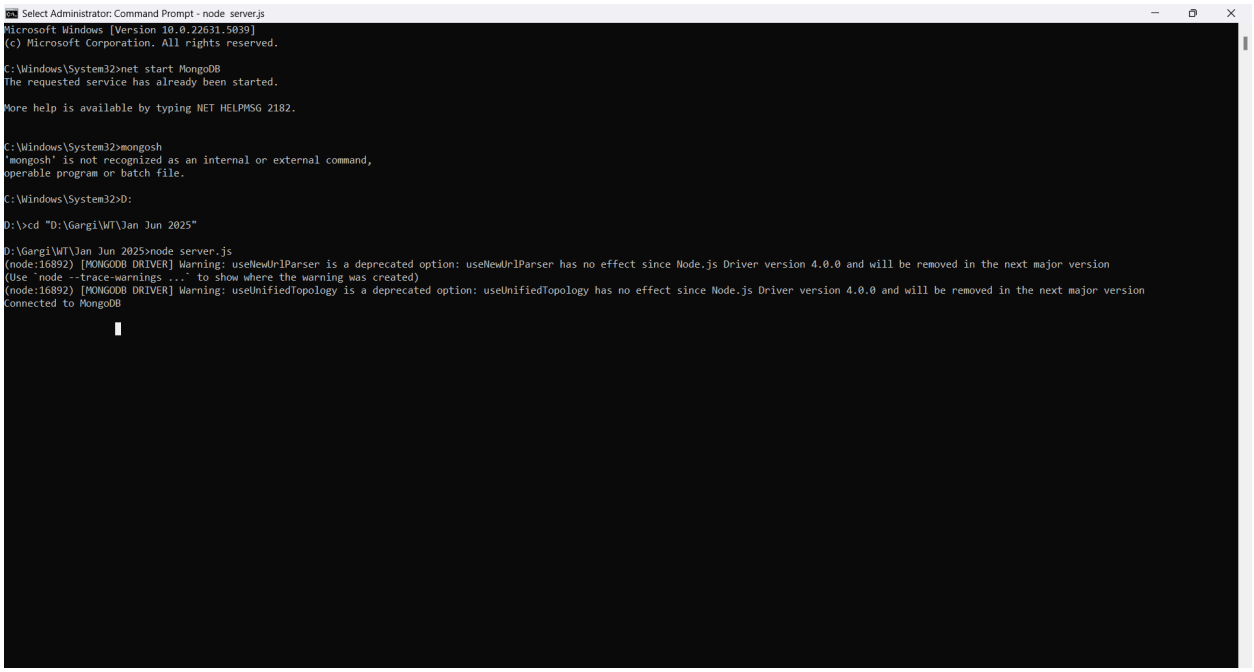


Example: Connecting Express to MongoDB

1. Download MongoDB community Server https://www.mongodb.com/try/download/community
2. Add MongoDB to System PATH C:\Program Files\MongoDB\Server\8.0\bin
3. mongod --version
4. Start MongoDB Compass, Add URI “mongodb://localhost:27017”, Click on Connect.

server.js

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

mongoose.connect("mongodb://localhost:27017/mydatabase", {
  useNewUrlParser: true,
  useUnifiedTopology: true
});

const db = mongoose.connection;
db.on("error", console.error.bind(console, "Connection error:"));
db.once("open", () => {
  console.log("Connected to MongoDB");
});
```



---

9. Conclusion

- **Node.js** allows running JavaScript on the server.
- **Express.js** simplifies backend development with easy routing and middleware.
- Together, they provide a powerful framework for building web applications and APIs.

🚀 **Next Steps:** Learn about authentication, REST APIs, and deploying Express apps!

---

1. Introduction to RESTful APIs

What is a RESTful API?

A **RESTful API** (Representational State Transfer API) is an architectural style used for designing networked applications. It allows communication between a client and a server using standard HTTP methods.

Why Use RESTful APIs?

✅ **Scalable & Flexible** – Suitable for large applications.
✅ **Language Independent** – Can be used with any programming language.
✅ **Stateless Communication** – No client context is stored on the server.
✅ **Widely Used** – Most modern web services follow REST principles.

2. REST Principles

1. Client-Server Architecture

- The client (frontend) and server (backend) are separate entities.
- The client requests resources, and the server responds.

2. Statelessness

- Each request from the client to the server must contain all necessary information.
- The server does not store session data.

3. Cacheability

- Responses can be cached to improve performance.
- APIs specify whether a response can be cached using HTTP headers.

4. Layered System

- The API can have multiple layers (security, load balancing, data transformation).

5. Uniform Interface

- Uses standard HTTP methods (GET, POST, PUT, DELETE).
- Uses URLs to represent resources.

3. HTTP Methods in REST APIs

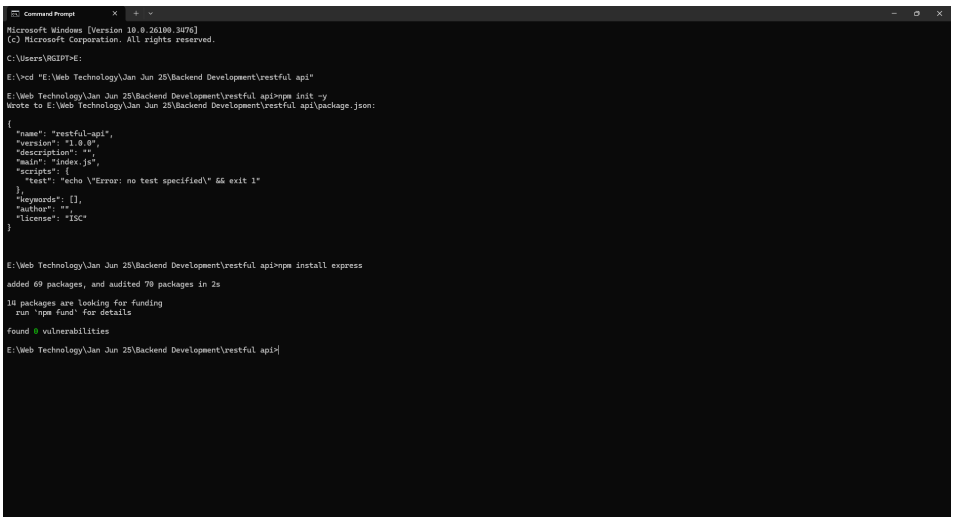
Method	Description	Example Endpoint

GET	Retrieve data	`/users` (Get all users)
POST	Create a new resource	`/users` (Create a user)
PUT	Update an existing resource	`/users/1` (Update user with ID 1)
DELETE	Remove a resource	`/users/1` (Delete user with ID 1)

4. REST API Example in Express.js

Installing Express.js

```
npm init -y

npm install express
```

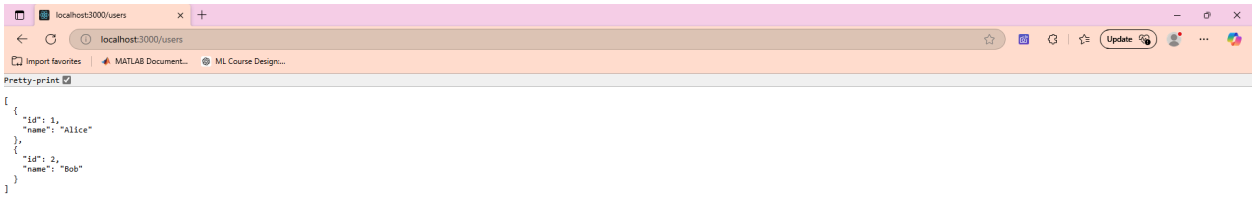


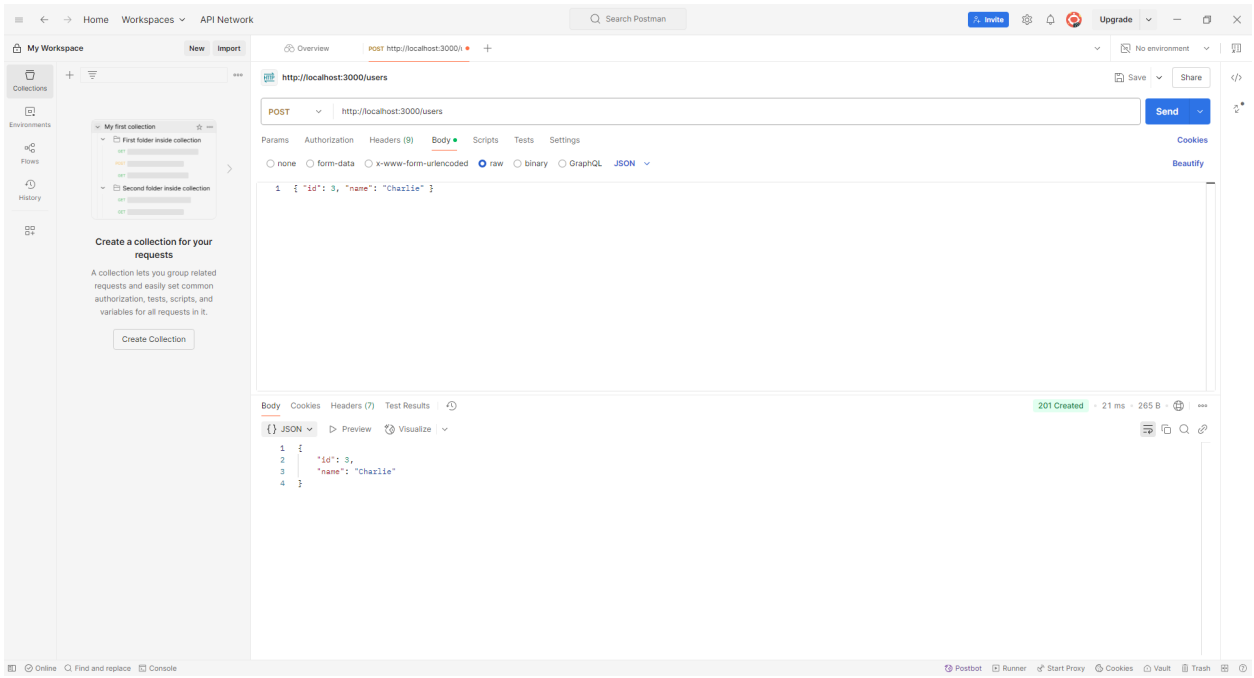
Creating a Basic RESTful API

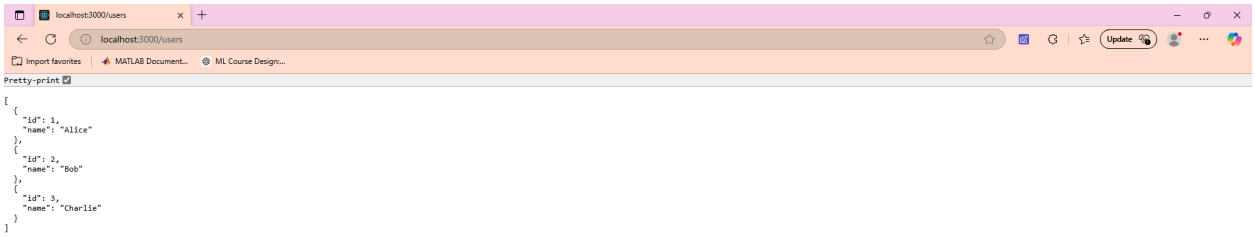
```
const express = require("express");
```

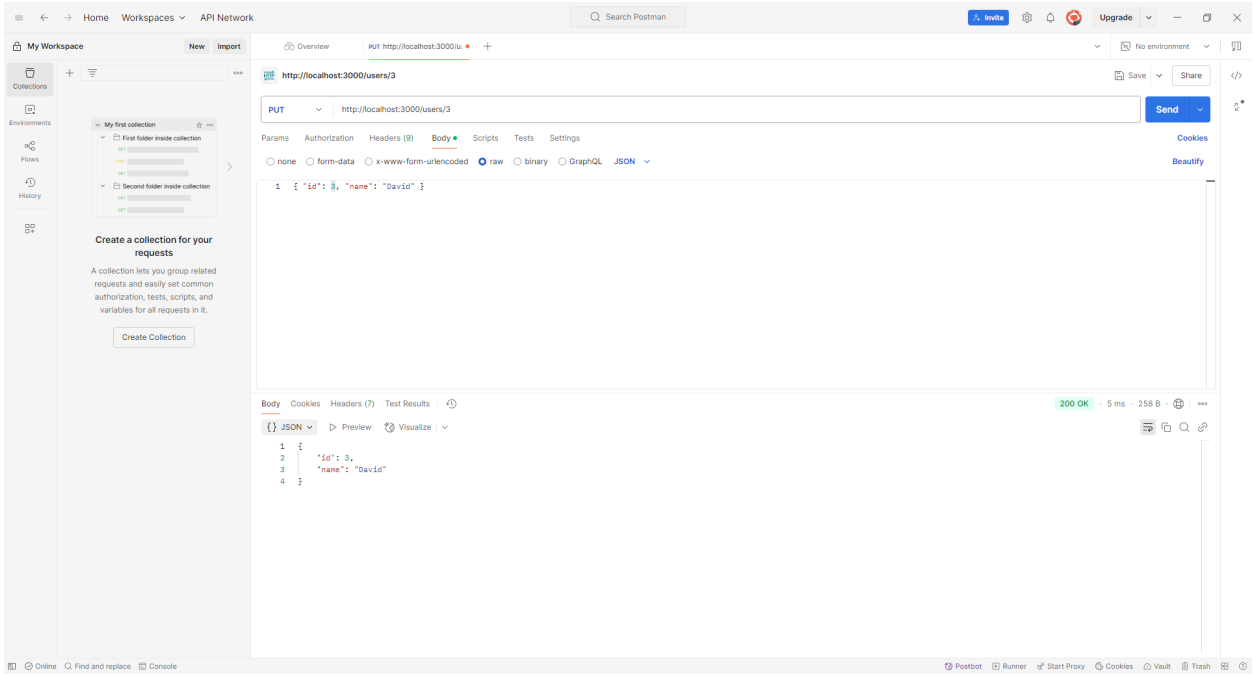
```
const app = express();

app.use(express.json());



let users = [

  { id: 1, name: "Alice" },

  { id: 2, name: "Bob" }

];



// GET: Fetch all users

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

  res.json(users);

});



// POST: Add a new user

app.post("/users", (req, res) => {

  const newUser = { id: users.length + 1, name: req.body.name };

  users.push(newUser);

  res.status(201).json(newUser);

});



// PUT: Update an existing user

app.put("/users/:id", (req, res) => {
```

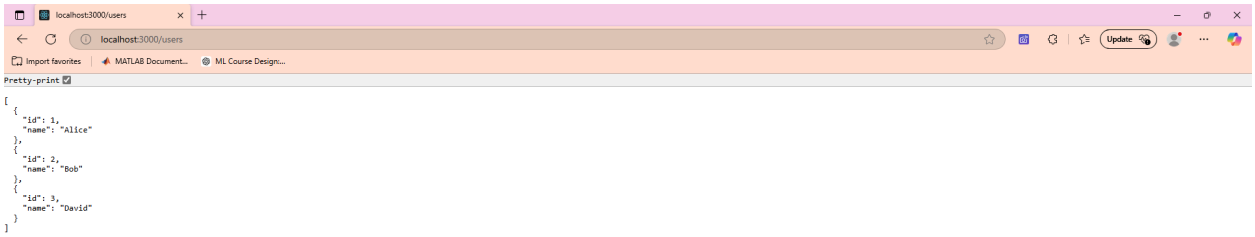
```
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (!user) return res.status(404).json({ message: "User not found" });
  user.name = req.body.name;
  res.json(user);
});

// DELETE: Remove a user
app.delete("/users/:id", (req, res) => {
  users = users.filter(u => u.id !== parseInt(req.params.id));
  res.json({ message: "User deleted" });
});

app.listen(3000, () => console.log("Server running on http://localhost:3000"));
```

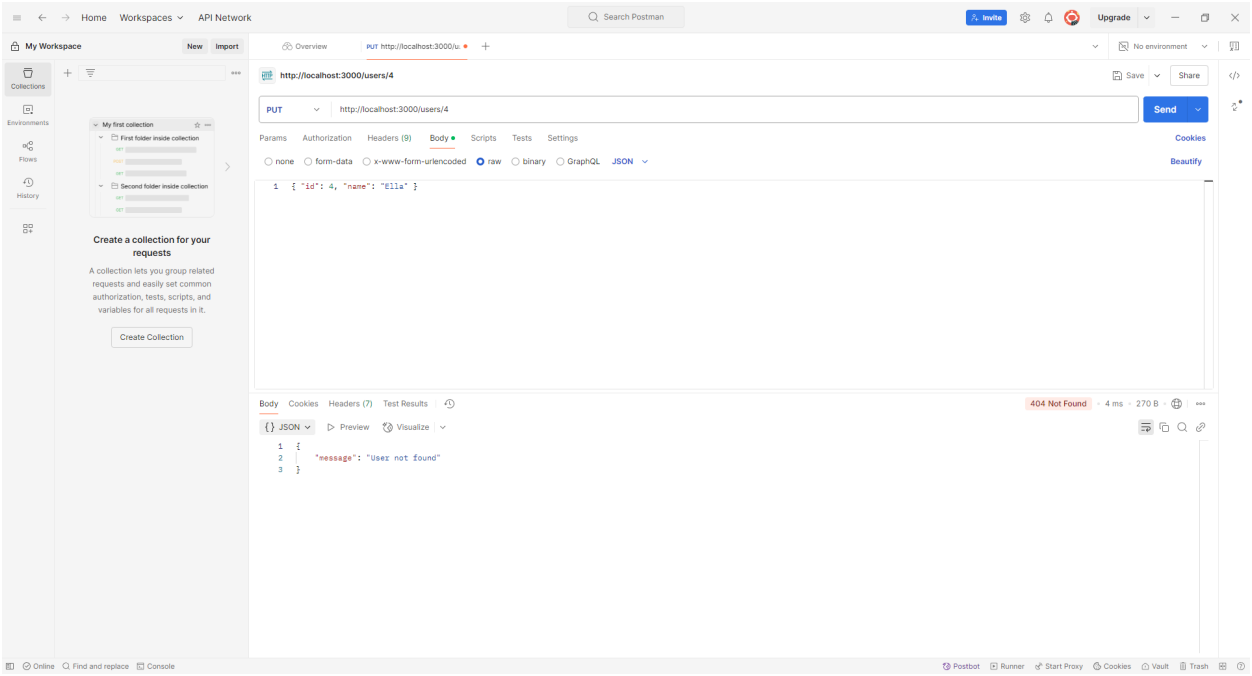


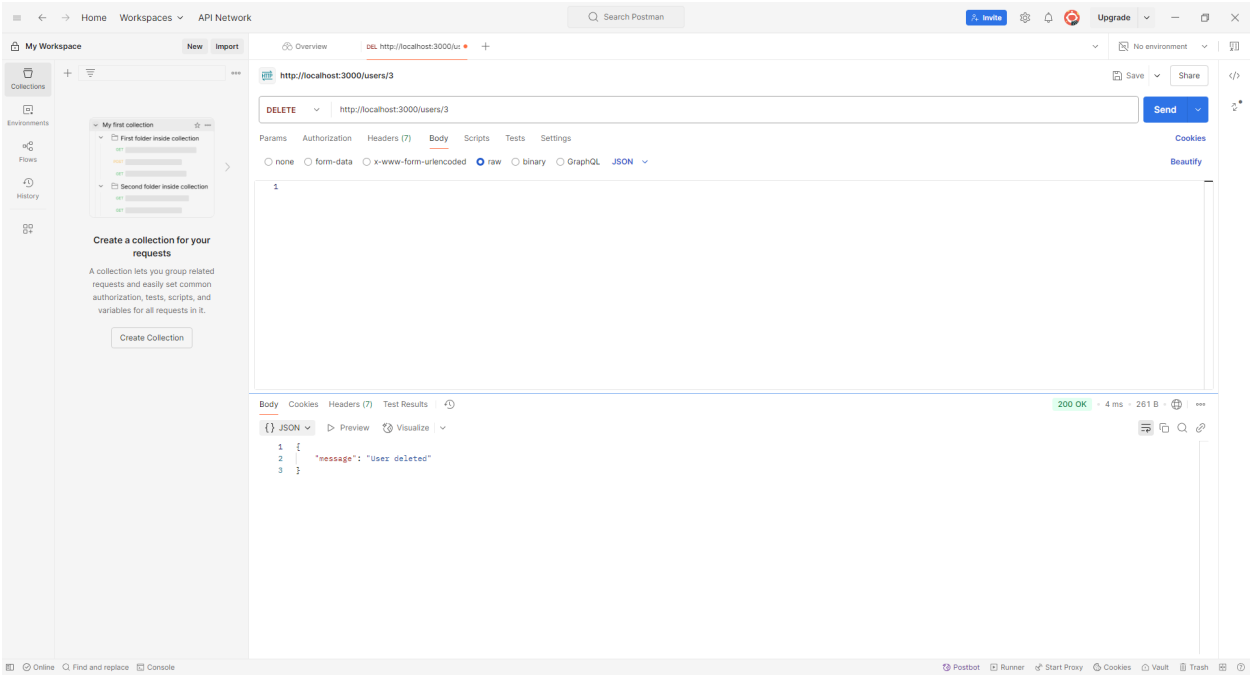

Home
Workspaces
API Network
Search Postman
Invite
Upgrade
My Workspace
New
Import
Collections
Environments
Flows
History
Overview
POST http://localhost:3000/
No environment
http://localhost:3000/users
Save
Share
POST
http://localhost:3000/users
Send
Params
Authorization
Headers (9)
Body
Scripts
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
1 { "id": 3, "name": "Charlie" }
Create a collection for your requests
A collection lets you group related requests and easily set common authorization, tests, scripts, and variables for all requests in it.
Create Collection
Body
Cookies
Headers (7)
Test Results
201 Created
21 ms
265 B
JSON
Preview
Visualize
1 {
2 "id": 3,
3 "name": "Charlie"
4 }
Online
Find and replace
Console
Postbot
Runner
Start Proxy
Cookies
Vault
Trash

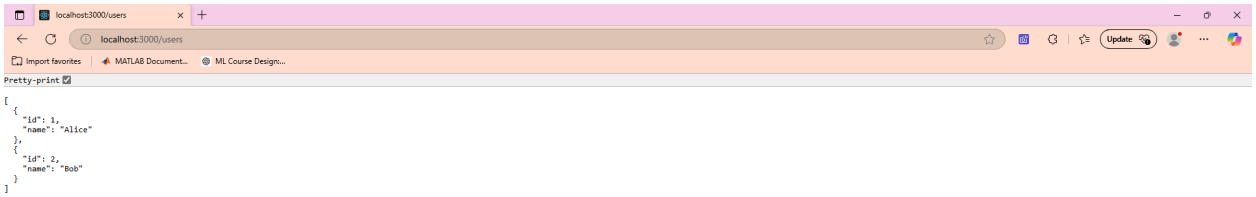

localhost:3000/users
localhost:3000/users
Update
Import favorites
MATLAB Document...
ML Course Design:...
Pretty-print
[
{
"id": 1,
"name": "Alice"
},
{
"id": 2,
"name": "Bob"
},
{
"id": 3,
"name": "Charlie"
}
]


Home
Workspaces
API Network
Search Postman
Invite
Upgrade
My Workspace
New
Import
Collections
Environments
Flows
History
Overview
PUT http://localhost:3000/u
No environment
http://localhost:3000/users/3
Save
Share
PUT
http://localhost:3000/users/3
Send
Params
Authorization
Headers (9)
Body
Scripts
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
1 { "id": 3, "name": "David" }
Create a collection for your requests
A collection lets you group related requests and easily set common authorization, tests, scripts, and variables for all requests in it.
Create Collection
Body
Cookies
Headers (7)
Test Results
200 OK
5 ms
258 B
JSON
Preview
Visualize
1 {
2 "id": 3,
3 "name": "David"
4 }
Online
Find and replace
Console
Postbot
Runner
Start Proxy
Cookies
Vault
Trash


localhost:3000/users
localhost:3000/users
Update
Import favorites
MATLAB Document...
ML Course Design:...
Pretty-print
[
{
"id": 1,
"name": "Alice"
},
{
"id": 2,
"name": "Bob"
},
{
"id": 3,
"name": "David"
}
]


Home
Workspaces
API Network
Search Postman
Invite
Upgrade
My Workspace
New
Import
Collections
Environments
Flows
History
Overview
PUT http://localhost:3000/u
No environment
http://localhost:3000/users/4
Save
Share
PUT
http://localhost:3000/users/4
Send
Params
Authorization
Headers (9)
Body
Scripts
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
1 { "id": 4, "name": "Ella" }
Create a collection for your requests
A collection lets you group related requests and easily set common authorization, tests, scripts, and variables for all requests in it.
Create Collection
Body
Cookies
Headers (7)
Test Results
404 Not Found
4 ms
270 B
JSON
Preview
Visualize
1 {
2 "message": "User not found"
3 }
Online
Find and replace
Console
Postbot
Runner
Start Proxy
Cookies
Vault
Trash


Home
Workspaces
API Network
Search Postman
Invite
Upgrade
My Workspace
New
Import
Collections
Environments
Flows
History
Overview
DEL http://localhost:3000/u
No environment
http://localhost:3000/users/3
Save
Share
DELETE
http://localhost:3000/users/3
Send
Params
Authorization
Headers (7)
Body
Scripts
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
1
Create a collection for your requests
A collection lets you group related requests and easily set common authorization, tests, scripts, and variables for all requests in it.
Create Collection
Body
Cookies
Headers (7)
Test Results
200 OK
4 ms
261 B
JSON
Preview
Visualize
1 {
2 "message": "User deleted"
3 }
Online
Find and replace
Console
Postbot
Runner
Start Proxy
Cookies
Vault
Trash



5. REST API Best Practices

1. Use Meaningful Resource Names

- `/users` instead of `/getUsers`
- `/products` instead of `/fetchProducts`

2. Use Plural Nouns for Collections

- `/users` (for multiple users)
- `/orders` (for multiple orders)

3. Use Proper HTTP Status Codes

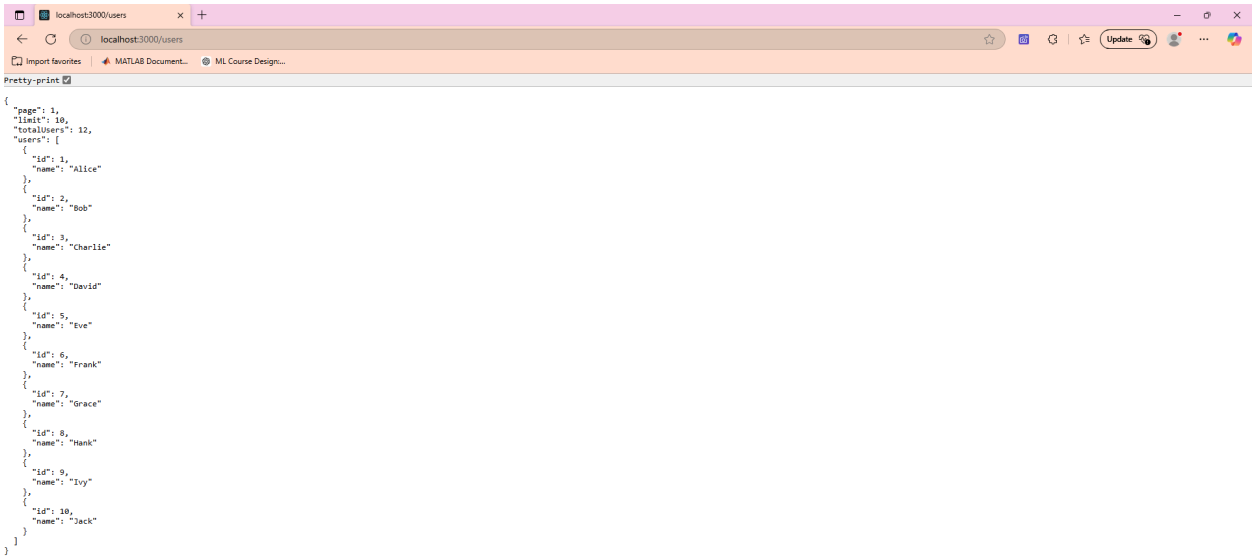
Code	Meaning
200	OK (Success)

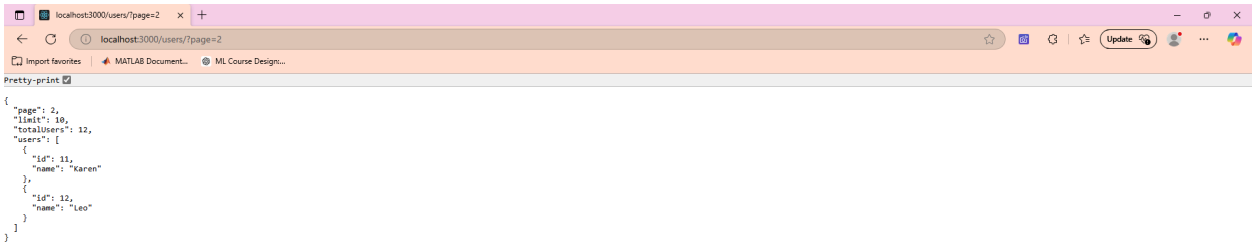
201	Created (New resource added)
400	Bad Request (Invalid input)
404	Not Found (Resource missing)
500	Internal Server Error

4. Use Pagination for Large Data

```
const express = require("express");
const app = express();
app.use(express.json());

let users = [
  { id: 1, name: "Alice" },
  { id: 2, name: "Bob" },
  { id: 3, name: "Charlie" },
  { id: 4, name: "David" },
  { id: 5, name: "Eve" },
  { id: 6, name: "Frank" },
  { id: 7, name: "Grace" },
  { id: 8, name: "Hank" },
  { id: 9, name: "Ivy" },
  { id: 10, name: "Jack" },
  { id: 11, name: "Karen" },
  { id: 12, name: "Leo" }
];

// GET: Fetch paginated users
app.get("/users", (req, res) => {
  let { page = 1, limit = 10 } = req.query;
  page = parseInt(page);
```

```
  limit = parseInt(limit);

  const startIndex = (page - 1) * limit;
  const endIndex = startIndex + limit;
  const paginatedUsers = users.slice(startIndex, endIndex);

  res.json({
    page,
    limit,
    totalUsers: users.length,
    users: paginatedUsers
  });
});

app.listen(3000, () => console.log("Server running on http://localhost:3000"));
```






```
/users?page=1&limit=10
```

5. Secure the API

- **Use authentication (JWT, OAuth)**
- **Validate user input to prevent attacks**
- **Enable CORS for cross-origin requests**

6. Conclusion

- RESTful APIs are **widely used** for modern web applications.
- They follow **standard HTTP methods** and **stateless architecture**.
- Best practices improve **performance, security, and maintainability**.

Connecting to Databases (MongoDB/PostgreSQL)

1. Introduction

Databases are essential for storing and managing application data. In modern web applications, two commonly used databases are:

- **MongoDB (NoSQL Database)**: Stores data in flexible, JSON-like documents.
- **PostgreSQL (SQL Database)**: A powerful, open-source relational database.

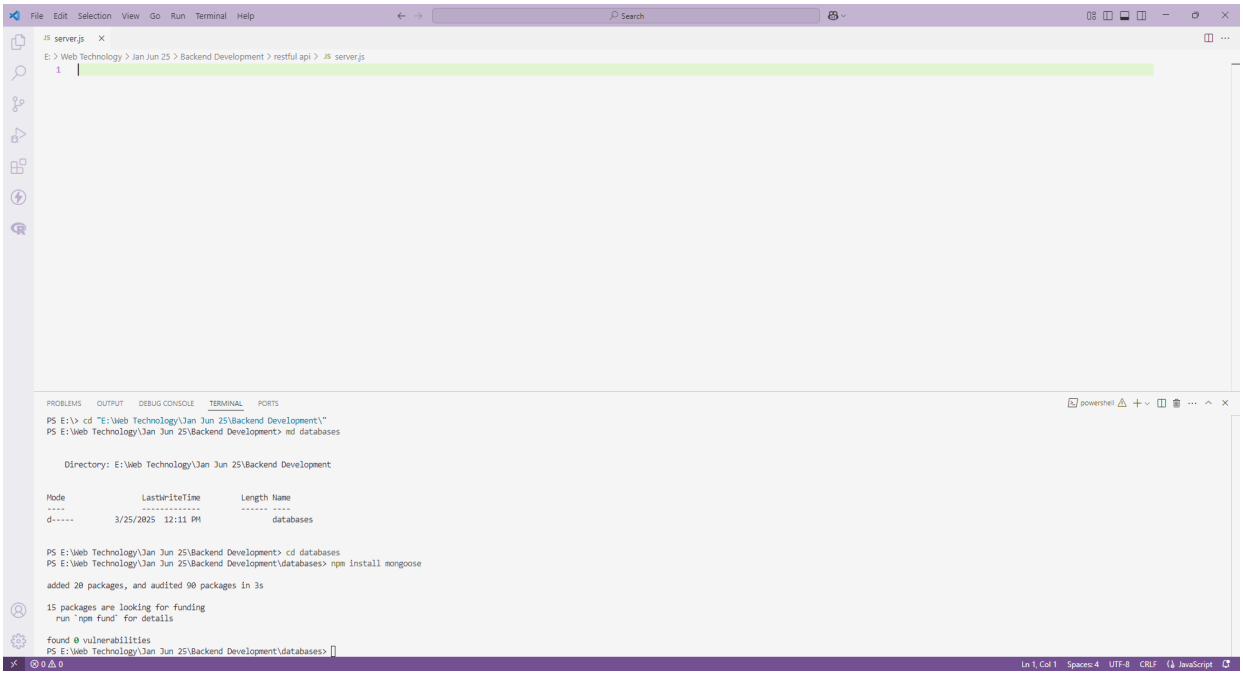
---

2. Connecting to MongoDB with Node.js and Mongoose

2.1 Installing MongoDB and Mongoose

To use MongoDB with Node.js, install **MongoDB** and **Mongoose** (an ODM for MongoDB):

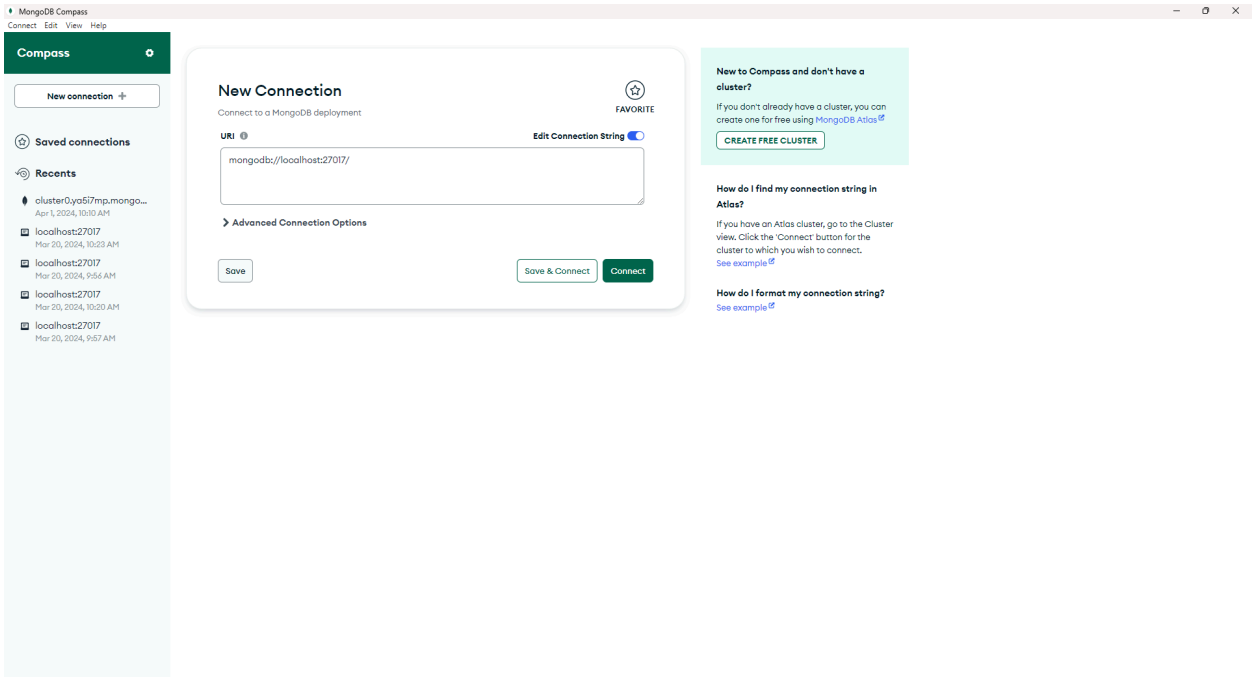
```
npm install mongoose
```



2.2 Setting Up MongoDB Connection

Create a file `db.js` for database connection:

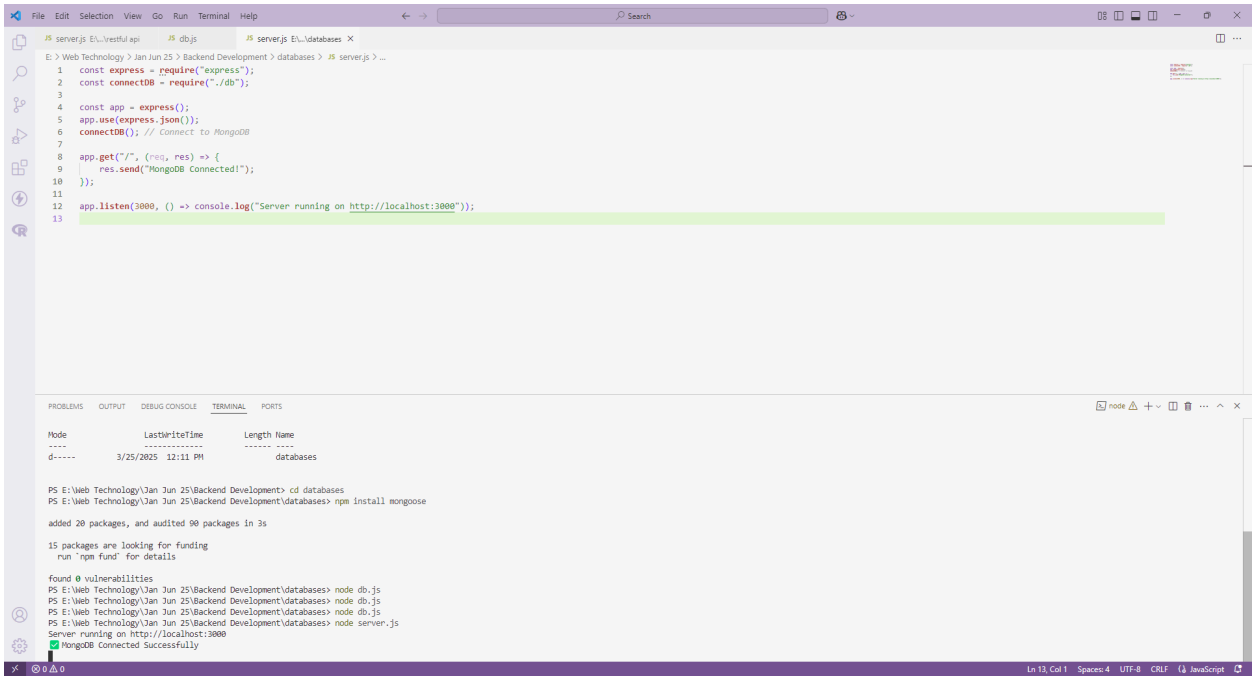
```
const mongoose = require("mongoose");



const connectDB = async () => {
  try {
    await mongoose.connect("mongodb://127.0.0.1:27017/mydatabase");
    console.log("MongoDB Connected Successfully");
  } catch (error) {
    console.error("MongoDB Connection Error:", error);
    process.exit(1);
  }
};



module.exports = connectDB;
```



2.3 Connecting MongoDB in an Express App

Modify `server.js` to use the database connection:

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

const connectDB = require("./db");



const app = express();

app.use(express.json());

connectDB(); // Connect to MongoDB



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

   res.send("MongoDB Connected!");

});



app.listen(3000, () => console.log("Server running on http://localhost:3000"));
```



2.4 Creating a Mongoose Model

Define a `User` model in `models/User.js`:

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



const UserSchema = new mongoose.Schema({

  name: { type: String, required: true },

  email: { type: String, required: true, unique: true },

  userId: { type: String, required: true, unique: true }

});



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

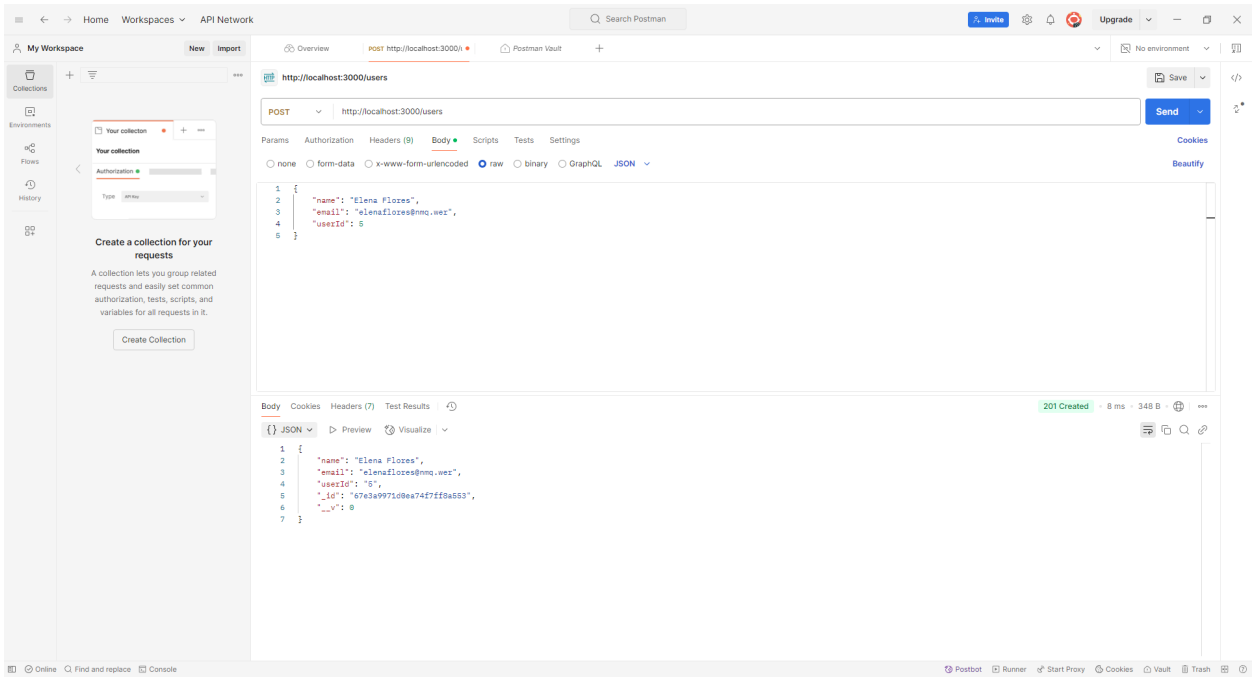
2.5 Performing CRUD Operations

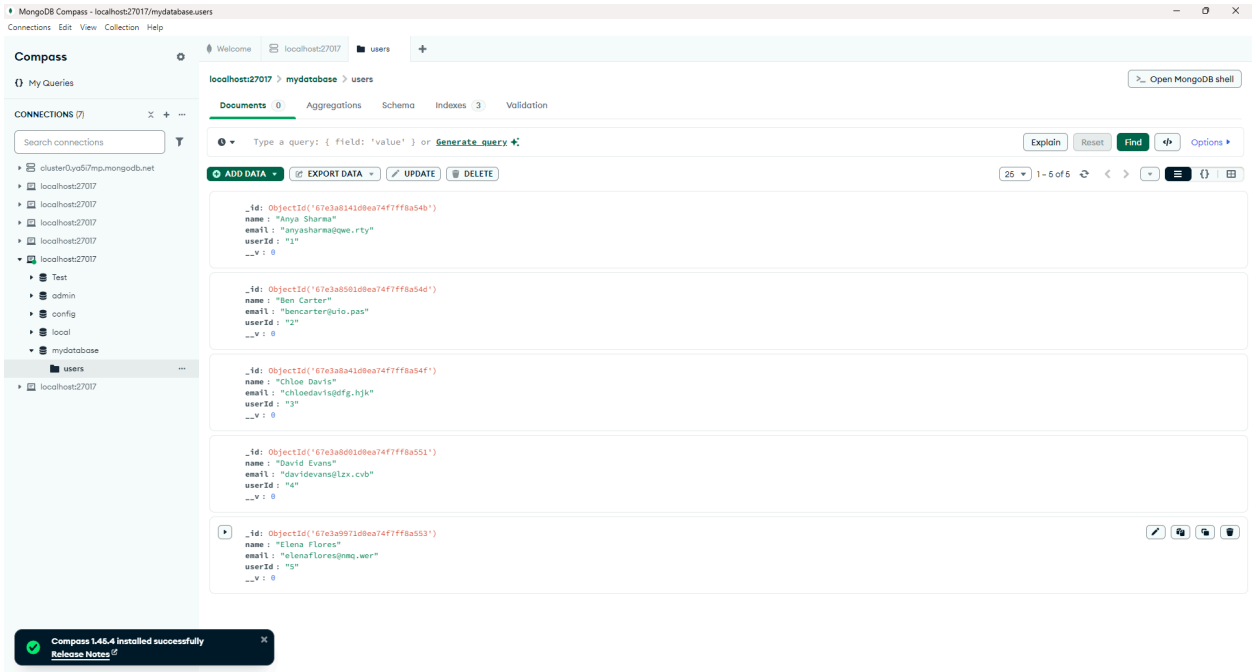
Create, Read, Update and Delete a User

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

const connectDB = require("./db");

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



const app = express();

app.use(express.json());

connectDB(); // Connect to MongoDB



// POST: Create a new user

app.post("/users", async (req, res) => {

  try {

    const user = new User(req.body);

    await user.save();

    res.status(201).json(user);

  } catch (error) {

    res.status(400).json({ message: error.message });

  }

});



// GET: Fetch all users

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

```
  const users = await User.find();
  res.json(users);
});

// PUT: Update a user by userId
app.put("/users/:userId", async (req, res) => {
  try {
    const user = await User.findOneAndUpdate({ userId: req.params.userId }, req.body, { new: true, runValidators: true });
    if (!user) {
      return res.status(404).json({ message: "User not found" });
    }
    res.json(user);
  } catch (error) {
    res.status(400).json({ message: error.message });
  }
});

// DELETE: Remove a user by userId
app.delete("/users/:userId", async (req, res) => {
  try {
    const user = await User.findOneAndDelete({ userId: req.params.userId });
    if (!user) {
```

```
        return res.status(404).json({ message: "User not found" });
      }
      res.json({ message: "User deleted successfully" });
    } catch (error) {
      res.status(500).json({ message: error.message });
    }
});


app.listen(3000, () => console.log("Server running on http://localhost:3000"));
```





---

3. Connecting to PostgreSQL with Node.js and Sequelize

3.1 Installing PostgreSQL and Sequelize

Sequelize is an ORM for SQL databases, including PostgreSQL. Install the required packages:

```
npm install pg pg-hstore sequelize
```

3.2 Setting Up PostgreSQL Connection

Create a `db.js` file:

```
const { Sequelize } = require("sequelize");


const sequelize = new Sequelize("mydatabase", "username", "password", {
  host: "localhost",
  dialect: "postgres"
```

```
});

const connectDB = async () => {
  try {
    await sequelize.authenticate();
    console.log("PostgreSQL Connected Successfully");
  } catch (error) {
    console.error("PostgreSQL Connection Error:", error);
  }
};

module.exports = { sequelize, connectDB };
```

3.3 Connecting PostgreSQL in an Express App

Modify `server.js`:

```
const express = require("express");
const { sequelize, connectDB } = require("./db");

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

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

```
  res.send("PostgreSQL Connected!");
});

app.listen(3000, () => console.log("Server running on http://localhost:3000"));
```

3.4 Defining a Model in Sequelize

Create `models/User.js`:

```
const { Sequelize, DataTypes } = require("sequelize");
const { sequelize } = require("../db");

const User = sequelize.define("User", {
  name: {
    type: DataTypes.STRING,
    allowNull: false
  },
  email: {
    type: DataTypes.STRING,
    allowNull: false,
    unique: true
  }
});

module.exports = User;
```

3.5 Performing CRUD Operations

Create a User

```
app.post("/users", async (req, res) => {
  const user = await User.create(req.body);
  res.json(user);
});
```

Read Users

```
app.get("/users", async (req, res) => {
  const users = await User.findAll();
  res.json(users);
});
```

---

4. Conclusion

- **MongoDB** is a NoSQL database suited for flexible, document-based storage.
- **PostgreSQL** is an SQL database with strong ACID compliance and relational structure.
- **Mongoose** provides an easy way to work with MongoDB, while **Sequelize** simplifies PostgreSQL operations.
- Connecting databases requires defining models, setting up connections, and performing CRUD operations.

---

1. Introduction

Authentication and authorization are crucial components of web security. They help ensure that users accessing a system are who they claim to be (authentication) and have the necessary permissions to perform specific actions (authorization).

2. Authentication

2.1 Definition

Authentication is the process of verifying the identity of a user, application, or system.

2.2 Types of Authentication

1. **Single-Factor Authentication (SFA)** – Uses one credential (e.g., password).
2. **Two-Factor Authentication (2FA)** – Requires an additional authentication factor like a code from a mobile device.
3. **Multi-Factor Authentication (MFA)** – Uses multiple authentication methods such as biometrics, OTPs, or security tokens.

2.3 Common Authentication Methods

1. **Username & Password** – Traditional method using stored credentials.
2. **Token-Based Authentication** – Uses a temporary token (e.g., JWT) for authentication.
3. **Biometric Authentication** – Uses fingerprints, facial recognition, or retina scans.
4. **OAuth** – A protocol that allows secure token-based authentication (used in Google, Facebook logins).
5. **SSO (Single Sign-On)** – Users authenticate once and gain access to multiple applications.

2.4 How Authentication Works

1. The user submits login credentials (username & password).
2. The server verifies credentials against stored data.
3. If valid, the server issues a session token or JWT.
4. The user includes this token in subsequent requests to prove identity.

2.5 Implementation of Authentication in Node.js (Example using JWT)

Bcrypt is a widely used library for securely hashing passwords. It relies on the Blowfish cipher and a technique called "salting" to hash passwords. Salting involves adding a random value (called the "salt") to

the original password before hashing it. This ensures that even if two users happen to have the same password, their hashed values will be different because of the unique salt added to each one.

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

const jwt = require('jsonwebtoken');

const bcrypt = require('bcryptjs');


const app = express();

app.use(express.json());


const users = []; // Temporary user storage


// Register Route

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

  const hashedPassword = await bcrypt.hash(req.body.password, 10);

  users.push({ username: req.body.username, password: hashedPassword });

  res.status(201).send('User registered');

});


// Login Route

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

  const user = users.find(u => u.username === req.body.username);

  if (!user || !(await bcrypt.compare(req.body.password, user.password))) {

    return res.status(403).send('Invalid credentials');
```

```
  }

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

  res.json({ token });

});


app.listen(3000, () => console.log('Server running on port 3000'));
```

3. Authorization

3.1 Definition

Authorization is the process of granting or restricting access to resources based on user roles or permissions.

3.2 Types of Authorization

1. **Role-Based Access Control (RBAC)** – Users are assigned roles with predefined permissions.
2. **Attribute-Based Access Control (ABAC)** – Access is granted based on attributes like department or location.
3. **Discretionary Access Control (DAC)** – Users control access to their own data.
4. **Mandatory Access Control (MAC)** – System enforces strict access policies.

3.3 How Authorization Works

1. The user is authenticated and receives a token.
2. The token contains role/permission details.
3. The server checks if the user has access to the requested resource.
4. If authorized, access is granted; otherwise, access is denied.

3.4 Implementation of Authorization in Node.js (Example with Role-Based Access Control)

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

const jwt = require('jsonwebtoken');
```

```
const app = express();

app.use(express.json());


const users = [
  { username: 'admin', password: 'admin123', role: 'admin' },
  { username: 'user', password: 'user123', role: 'user' }
];


const authenticateToken = (req, res, next) => {
  const token = req.header('Authorization');
  if (!token) return res.status(401).send('Access Denied');


  try {
    const verified = jwt.verify(token.split(" ")[1], 'secret'); //The token secret is a random string used to encrypt and decrypt data.
    req.user = verified;
    next();
  } catch (err) {
    res.status(403).send('Invalid Token');
  }
};
```

```
const authorizeRole = (role) => {
  return (req, res, next) => {
    if (req.user.role !== role) {
      return res.status(403).send('Access Forbidden');
    }
    next();
  };
};

app.get('/admin', authenticateToken, authorizeRole('admin'), (req, res) => {
  res.send('Welcome Admin');
});

app.listen(3000, () => console.log('Server running on port 3000'));
```

4. Best Practices for Authentication & Authorization

1. **Use Strong Password Hashing** – Use bcrypt or Argon2 for password storage.
2. **Implement Multi-Factor Authentication (MFA)** – Add an extra layer of security.
3. **Use HTTPS** – Encrypt data to prevent man-in-the-middle attacks.
4. **Limit Login Attempts** – Prevent brute force attacks.
5. **Use Secure Session Management** – Avoid storing sensitive information in cookies.
6. **Implement Least Privilege Principle** – Grant users only the permissions they need.
7. **Regularly Rotate API Keys and Tokens** – Reduce risk of token leakage.
8. **Use Secure Token Storage** – Store JWTs securely using HttpOnly and Secure flags.