Node.js & Express Framework: Introduction and Environment Setup

1. Introduction to Node.js

- **Node.js** is a runtime environment that allows you to run JavaScript on the server side.
- It's built on **Google Chrome's V8 JavaScript engine**.
- Node.js is non-blocking, event-driven, and designed for building scalable network applications.

2. Introduction to Express Framework

- **Express.js** is a fast, minimal, and flexible Node.js web application framework.
- It simplifies building web applications and APIs by providing robust features for routing and middleware handling.

3. Environment Setup

Step 1: Install Node.js

1. Visit the official Node.js website.
2. Download and install the **LTS version** for your OS (Windows, macOS, or Linux).
3. Verify installation:

```
node -v
npm -v
```

Step 2: Set Up a New Project

1. Create a new directory for your project:

```
mkdir my-express-app
cd my-express-app
```

2. Initialize a new Node.js project:

```
npm init -y
```

This creates a `package.json` file.

Step 3: Install Express

1. Install Express in your project:

   ```
   npm install express --save
   ```

2. Verify Express is added to `package.json` under **dependencies**.

Step 4: Create a Basic Express Server

1. Create a new file named `app.js`:

   ```
   touch app.js
   ```

2. Add this basic Express code to `app.js`:

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

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

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

Step 5: Run the Server

1. Run the Express server:

   ```
   node app.js
   ```

2. Open your browser and visit `http://localhost:3000`. You should see "Hello World!"

You have now set up a basic Node.js environment and created a simple Express web server.

Serving static resources:

In **Express.js**, serving **static resources** refers to delivering files like images, CSS, JavaScript, or any other assets to the client (browser). These files don't change on the server and are directly sent in response to client requests.

Express makes it easy to serve static files using built-in middleware called **express.static().**

How to Serve Static Resources

1. **Place your static files** (like images, CSS, or JS files) in a directory, e.g., **`public/`**.

2. **Use `express.static()` middleware** to serve these files. You need to tell Express where to find these files.

Example:

1. Create a folder named `public/` in your project directory, and put your static files in it (e.g., CSS, images, JS).

2. Then, in your `app.js`, configure Express to serve static files from this folder.

Example:

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

const app = express();

const port = 3000;

// Use the 'public' folder to serve static files

app.use(express.static('public'));

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

  res.send('Home Page');

});

app.listen(port, () => {

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

});
```

Folder Structure Example:

```
/my-express-app
  └── public/
     ├── styles.css
     ├── script.js
     └── image.jpg
  └── app.js
```

How It Works:

1. If a user visits http://localhost:3000/image.jpg, Express will automatically look inside the public/ folder and serve the file image.jpg.

2. Similarly, requests to http://localhost:3000/styles.css or http://localhost:3000/script.js will serve the respective static files.

Why Use express.static()?

Simplifies serving files (e.g., CSS, JS, images) without manually creating routes for each.

What is a Template Engine?

A template engine allows you to generate dynamic HTML content by combining templates (with placeholders) and data. In web applications, instead of writing static HTML files, template engines let you use variables, control structures (like loops and conditionals), and render content dynamically.

Popular template engines in Express.js include Vash and Jade (now called Pug).

1. Vash Template Engine

Vash is a template engine inspired by the Razor syntax (from ASP.NET). It allows you to embed JavaScript code within HTML to generate dynamic web pages.

Steps to use Vash in Express.js:

1. **Install Vash**:

   npm install vash –save

Set up Vash in Express: Modify your `app.js` to use Vash as the template engine:

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

const app = express();

const port = 3000;

// Set the view engine to Vash

app.set('view engine', 'vash');

// Define a route and render a Vash template

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

  res.render('index', { title: 'Welcome', message: 'Hello from Vash!' });

});

app.listen(port, () => {

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

});
```

Create a Vash Template: Inside a **`views`**`/` directory, create a `index.vash` file:

```
<!DOCTYPE html>

<html>

<head>

   <title>@title</title>

</head>

<body>

   <h1>@message</h1>

</body>

</html>
```

`@title` and **`@message`** are placeholders for dynamic data.

<u>Run the server</u>:

When you visit `http://localhost:3000`, it will render the dynamic HTML using the Vash template.

2. Jade (Pug) Template Engine

Jade (renamed to **Pug**) is another popular template engine with a minimalist syntax that omits closing tags and reduces boilerplate in HTML code.

Steps to use Jade/Pug in Express.js:

1. **Install Pug**:

   ```
   npm install pug
   ```

Set up Pug in Express: Modify your `app.js` to use Pug as the template engine:

Example:

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

const app = express();

const port = 3000;

// Set the view engine to Pug

app.set('view engine', 'pug');

// Define a route and render a Pug template

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

  res.render('index', { title: 'Welcome', message: 'Hello from Pug!' });

});

app.listen(port, () => {

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

});
```

Create a Pug Template: Inside the `views/` directory, create an **`index.pug`** file

```
doctype html
html
  head
    title= title
  body
    h1= message
```

The Pug syntax is indentation-based and doesn’t require closing tags.

Run the server: When you visit `http://localhost:3000`, it will render the dynamic HTML using the Pug template.

Key Differences between Vash and Jade (Pug)

Feature	Vash	Jade (Pug)
Syntax	Similar to Razor (C#)	Minimalistic, indentation-based
Usage	@ for variables	= for variables
HTML Structure	Uses traditional HTML structure	Reduces boilerplate by omitting closing tags
Popularity	Less common but powerful	Very popular, widely used

Conclusion:

- **Vash** is ideal if you're familiar with Razor or prefer embedding JavaScript in a more HTML-like syntax.
- **Pug** (Jade) is more concise and elegant, minimizing boilerplate HTML.

Both engines can be integrated easily into Express.js, providing dynamic HTML rendering capabilities for your web applications.

Steps to Connect Node.js to a MongoDB Database

To connect Node.js with a MongoDB database, you'll typically use the **Mongoose library** or the **MongoDB native driver**. Below are the steps for both methods, starting with Mongoose, which is the most commonly used.

Method 1: Using Mongoose

Step 1: Install Mongoose

First, you need to install Mongoose, which is an ODM (Object Data Modeling) library that provides a simple way to interact with MongoDB.

```
npm install mongoose
```

Step 2: Set Up MongoDB

1. **Install MongoDB** if you haven't already or use a cloud-based solution like **MongoDB Atlas**.
2. Create a new database (for example, `mydatabase`).
3. Get the connection string, which typically looks like:
   - For local MongoDB:

     ```
     mongodb://localhost:27017/mydatabase
     ```

   - For MongoDB Atlas:

     ```
     mongodb+srv://<username>:<password>@cluster.mongodb.ne
     t/mydatabase
     ```

Step 3: Connect to MongoDB in Node.js

1. Create a file called `app.js` or use your existing Node.js project.
2. Import Mongoose and connect to your MongoDB database using the **`mongoose.connect ()`** method.

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

// Replace with your MongoDB URI

const mongoURI =
'mongodb://localhost:27017/mydatabase';

// Connect to MongoDB
mongoose.connect (mongoURI, { useNewUrlParser: true,
useUnifiedTopology: true })
  .then(() => console.log('MongoDB connected
successfully!'))
  .catch(err => console.log('Failed to connect to
MongoDB', err));
```

Step 4: Define a Mongoose Model

To interact with MongoDB collections, you need to define a **Mongoose Schema** and **Model**.

```
// Define a schema for a collection (e.g., Users)
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  age: Number,
});

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

Step 5: Perform Database Operations

Once connected, you can perform CRUD operations (Create, Read, Update, Delete) using Mongoose.

```
// Create a new user
const newUser = new User({
  name: 'John Doe',
  email: 'john@example.com',
  age: 30,
});

// Save the user to the database
newUser.save()
  .then(() => console.log('User saved successfully!'))
  .catch(err => console.log('Error saving user', err));

// Read data (Find users)
User.find()
  .then(users => console.log(users))
  .catch(err => console.log(err));
```

Method 2: Using the MongoDB Native Driver

Step 1: Install MongoDB Driver

If you prefer using the MongoDB native driver instead of Mongoose, you need to install the MongoDB package.

```
npm install mongodb
```

Step 2: Connect to MongoDB

1. Create a file called `app.js` or use your existing Node.js project.
2. Import the **MongoClient** from the MongoDB library and connect to the database.

```
const { MongoClient } = require('mongodb');

// Replace with your MongoDB URI
const mongoURI = 'mongodb://localhost:27017';
const dbName = 'mydatabase';

// Create a new MongoClient
const client = new MongoClient(mongoURI, {
useNewUrlParser: true, useUnifiedTopology: true });

client.connect()
  .then(() => {
    console.log('Connected to MongoDB!');
    const db = client.db(dbName);

    // Perform operations here (CRUD)
    const users = db.collection('users');

    // Example: Insert a user
    users.insertOne({ name: 'John Doe', email:
'john@example.com', age: 30 })
      .then(result => {
        console.log('User inserted:', result);
        client.close(); // Close connection after
operation
      })
      .catch(err => console.log('Error inserting user',
err));
  })
  .catch(err => console.log('Failed to connect to
MongoDB', err));
```

Step 3: Perform Database Operations

Once connected, you can perform Create operations directly using the **MongoDB Native Driver**. Here's an example of reading data from the `users` collection.

```
// Find all users
   users.find().toArray()
  .then(users => console.log (users))
  .catch(err => console.log(err));
```

Summary of Key Steps

1. **Install Required Packages**:
   - Use `mongoose` or the `mongodb` driver.
2. **Connect to MongoDB**:
   - For local MongoDB or MongoDB Atlas, use the connection string.
3. **Create Models or Use Collections**:
   - With **Mongoose**, define schemas and models.
   - With the **MongoDB native driver**, directly access collections.
4. **Perform Create Operations**:
   - Insert documents from your MongoDB database.

These are steps to successfully connect Node.js with MongoDB, allowing you to manage data efficiently in your applications.

Mongoose Module:

Mongoose is an **Object Data Modeling (ODM) library** for **MongoDB** and **Node.js**. It provides a schema-based solution to model and manage MongoDB data in a structured way. With Mongoose, you can define **data schemas**, **create models**, and **interact with MongoDB collections** using simple, intuitive JavaScript methods.

Key Features of Mongoose

1. **Schemas**:
   - Mongoose allows you to define schemas for your MongoDB collections. Schemas define the structure of documents, including data types, default values, and validation rules.
   - Example:

   ```
   const mongoose = require('mongoose');
   const userSchema = new mongoose.Schema({
     name: String,
     email: String,
     age: Number,
   });
   ```

2. **Models**:
   - A **model** in Mongoose is a wrapper for a MongoDB collection. It allows you to create, update, delete, and query documents based on the defined schema.
   - Example:

   ```
   Const User = mongoose.model ('User',
   userSchema);
   ```

   - Here, `User` is the model representing the `users` collection in MongoDB, allowing you to perform CRUD operations.
3. **Validation**:
   - Mongoose provides built-in validation on schema fields (e.g., required fields, minimum/maximum values) to ensure data integrity before saving to MongoDB.
   - Example:

   ```
   const userSchema = new mongoose.Schema({
     name: { type: String, required: true },
   ```

```
  age: { type: Number, min: 18, max: 65 },
});
```

4. **Query Helpers**:
   - Mongoose offers chainable query methods for filtering, sorting, and selecting documents from the database.
   - Example:

```
User.find({ age: { $gt: 18 } })
  .sort('name')
  .select('name email')
  .exec((err, users) => {
    console.log(users);
  });
```

Benefits of Using Mongoose

- **Schema-based modeling**: Ensures data consistency with pre-defined schemas.
- **Validation**: Built-in validation to ensure data integrity.
- **Querying**: Chainable and flexible query methods.
- **Middleware**: Ability to execute logic before or after specific operations.
- **Population**: Easy handling of references between collections (joins-like behavior).

When to Use Mongoose?

- When you need structured and validated data models.
- When you want to define relationships between collections.
- When you want to avoid directly writing MongoDB queries and prefer an abstraction layer.

Conclusion

Mongoose is a powerful tool for working with MongoDB in Node.js. It simplifies database operations, enforces schema rules, and provides additional features like hooks, validation, and population, making it a great choice for structured MongoDB-based applications.

Creating REST APIs in Express.js

Creating REST APIs in Express.js is a common task for building backend services that communicate with clients (such as browsers or mobile apps) via HTTP requests (GET, POST, PUT, DELETE, etc.). Let's see how to creating a simple REST API using Express.js.

Step 1: Install Node.js and Express

1. **Initialize a new Node.js project**: Open your terminal, navigate to your project folder, and run the following command to create a `package.json` file:

   ```
   npm init -y
   ```

2. **Install Express**: Install Express.js to handle HTTP requests:

   ```
   npm install express
   ```

Step 2: Basic Setup for Express

Create a file called `app.js` (or `server.js`), and set up Express to handle requests.

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

// Middleware to parse JSON requests
app.use(express.json());

// Basic route for testing
app.get('/', (req, res) => {
  res.send('Welcome to the REST API');
});

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

Step 3: Define Routes for RESTful APIs

You'll typically create APIs for different **resources** (like users, products, etc.). For this example, let's create a REST API for managing a list of **books**.

REST API Endpoints for Books:

HTTP Method	URL	Description
GET	/api/books	Get all books
GET	/api/books/	Get a single book by ID
POST	/api/books	Add a new book
PUT	/api/books/	Update a book by ID
DELETE	/api/books/	Delete a book by ID

Step 4: Implement API Endpoints

Here's how to implement these REST API routes:

```
// In-memory data store for books (just for demonstration)

let books = [
  { id: 1, title: '1984', author: 'George Orwell' },
  { id: 2, title: 'To Kill a Mockingbird', author: 'Harper Lee'
},
];

// GET: Retrieve all books
app.get('/api/books', (req, res) => {
  res.json(books);
});

// GET: Retrieve a single book by ID
app.get('/api/books/:id', (req, res) => {
  const bookId = parseInt(req.params.id);
  const book = books.find(b => b.id === bookId);
  if (!book) return res.status(404).json({ message: 'Book not
found' });

  res.json(book);
});

// POST: Add a new book
app.post('/api/books', (req, res) => {
  const newBook = {
    id: books.length + 1,
    title: req.body.title,
    author: req.body.author,
  };

  books.push(newBook);
  res.status(201).json(newBook);
});

// PUT: Update a book by ID
app.put('/api/books/:id', (req, res) => {
  const bookId = parseInt(req.params.id);
  const book = books.find(b => b.id === bookId);

  if (!book) return res.status(404).json({ message: 'Book not
found' });

  book.title = req.body.title || book.title;
  book.author = req.body.author || book.author;

  res.json(book);
});

// DELETE: Remove a book by ID
app.delete('/api/books/:id', (req, res) => {
  const bookId = parseInt(req.params.id);
  const bookIndex = books.findIndex(b => b.id === bookId);

  if (bookIndex === -1) return res.status(404).json({ message:
'Book not found' });

  books.splice(bookIndex, 1);
  res.json({ message: 'Book deleted' });
});
```

Step 5: Test Your API

1. **Start your server** by running the following command:

   ```
   node app.js
   ```

2. **Test your API** using Postman, Curl, or directly in your browser (for GET requests):
   - **GET all books**: `http://localhost:3000/api/books`
   - **GET book by ID**: `http://localhost:3000/api/books/1`
   - **POST a new book**: Send a POST request to `http://localhost:3000/api/books` with the following JSON body:

     ```
     {
       "title": "The Great Gatsby",
       "author": "F. Scott Fitzgerald"
     }
     ```

   - **PUT to update a book**: Send a PUT request to `http://localhost:3000/api/books/1` with a body like:

     ```
     {
       "title": "Animal Farm",
       "author": "George Orwell"
     }
     ```

   - **DELETE a book**: Send a DELETE request to `http://localhost:3000/api/books/1`.

Step 6: Connecting to a Database (Optional)

You can integrate a database (like MongoDB, MySQL, or PostgreSQL) with your Express API.

Example using MongoDB and Mongoose:

1. **Install Mongoose**:

   ```
   npm install mongoose
   ```

2. **Connect to MongoDB**:

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

   mongoose.connect('mongodb://localhost:27017/mydatabase', {
   ```

```
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => console.log('Connected to MongoDB'))
  .catch(err => console.error('Could not connect to MongoDB',
err));
```

3. **Define a Model**:

```
javascript
Copy code
const bookSchema = new mongoose.Schema({
  title: String,
  author: String
});

const Book = mongoose.model('Book', bookSchema);
```

4. **CRUD Operations with MongoDB**: Replace your in-memory array with MongoDB operations using Mongoose models.

Conclusion

By following these steps, you can create a simple REST API using **Express.js**. You can expand this API by adding more routes, connecting it to databases, handling authentication, and using middleware to extend its functionality.

Express.js Framework:

Express.js is a minimal and flexible **Node.js web application framework** that provides a robust set of features to develop web and mobile applications. It simplifies the process of building servers and handling HTTP requests, making it the go-to choice for developers when creating REST APIs, web applications, and more.

Step 1: Install Node.js and NPM

Before starting with Express.js, you need to have **Node.js** and **npm** (Node Package Manager) installed on your machine.

- **Install Node.js**
- Verify installation by running:

```
node -v
npm -v
```

Step 2: Create a New Node.js Project

1. Create a new directory for your project.

```
mkdir my-express-app
cd my-express-app
```

2. Initialize a new Node.js project by creating a `package.json` file.

```
npm init -y
```

   This creates a basic `package.json` file with default settings.

Step 3: Install Express.js

1. **Install Express** as a dependency.

```
npm install express
```

2. After installing, Express is now available to be used in your application.

Step 4: Create a Basic Express Server

1. Create an entry file, typically named `app.js` or `server.js`:

```
touch app.js
```

2. In `app.js`, write the following code to set up a basic Express server:

   ```
   // Import the Express module
   const express = require('express');
   const app = express();
   const port = 3000;

   // Define a route for the root URL ('/')
   app.get('/', (req, res) => {
     res.send('Hello, Express!');
   });

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

3. Run the application by using:

   ```
   node app.js
   ```

4. Open your browser and go to `http://localhost:3000`. You should see **"Hello, Express!"**.

Step 5: Add Middleware

Middleware in Express is a function that processes requests before they reach the route handler. Middleware is commonly used for tasks like parsing incoming data or handling authentication.

1. Add **body-parser** middleware (built-in starting from Express 4.16.0) to parse incoming JSON data.

   ```
   app.use(express.json ()); // for parsing
   application/json
   ```

Step 6: Define Routes

Routes define how the server responds to specific HTTP requests. You can define routes for different HTTP methods like **GET**, **POST**, **PUT**, **DELETE**, etc.

1. **Basic routes**:

   ```
   // GET request
   app.get('/api/users', (req, res) => {
     res.send('Get all users');
   });

   // POST request
   app.post('/api/users', (req, res) => {
     res.send('Create a new user');
   });

   // PUT request
   app.put('/api/users/:id', (req, res) => {
     res.send(`Update user with ID ${req.params.id}`);
   });

   // DELETE request
   app.delete('/api/users/:id', (req, res) => {
     res.send(`Delete user with ID ${req.params.id}`);
   });
   ```

2. In the routes above, the `req` object holds the request information, and `res` is the response object.

Step 7: Serving Static Files

Express can serve static files such as images, CSS, and JavaScript directly to clients.

1. Create a directory `public` to store static assets:

   ```
   mkdir public
   ```

2. Place any static files (like `index.html`, CSS, or images) in this directory.
3. Use the following middleware to serve static files:

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

4. Now, if you visit `http://localhost:3000/filename`, Express will automatically serve files from the `public` folder.

Step 8: Using Template Engines (Optional)

A **template engine** allows you to generate dynamic HTML pages. Some common template engines are **Pug (formerly Jade)**, **EJS**, and **Vash**.

1. Install a template engine (e.g., **Pug**):

   ```
   npm install pug
   ```

2. Set up the template engine in your `app.js`:

   ```
   app.set('view engine', 'pug');
   app.set('views', './views'); // Specify the views directory
   ```

3. Create a `views` folder with a file named `index.pug`:

   ```
   doctype html
   html
     head
       title= title
     body
       h1 Hello #{name}
   ```

Step 9: Start the Server

Finally, start your server to run the application. You can also use **nodemon** for automatically restarting the server when code changes.

1. **Run the server**:

   ```
   node app.js
   ```

2. **Or use nodemon** (install it globally or as a dev dependency):

   ```
   npm install -g nodemon
   nodemon app.js
   ```

Conclusion

By following these steps, you can create a basic but fully functional web application using Express.js. Express offers many more features, such as advanced routing, middleware handling, and integration with various databases (e.g., MongoDB, MySQL). It is widely used in creating RESTful APIs and web apps because of its simplicity, flexibility, and extensive ecosystem.

MVC Pattern in Express.js:

MVC stands for **Model-View-Controller**, a software design pattern used for developing web applications. It divides the application into three interconnected components to separate internal representations of information from how it is presented to and accepted by the user.

The **MVC** pattern in **Express.js** helps organize your code structure by dividing it into:

- **Model**: Manages the data and the logic of the application. It represents the data (typically by interacting with a database) and defines the business logic.
- **View**: Defines how the data is presented to the user (usually HTML templates, or JSON if building an API).
- **Controller**: Acts as an intermediary between the Model and the View. It processes user input, interacts with the Model, and renders the appropriate View based on user actions.

Components of MVC in Express.js

1. Model

The **Model** represents the application's data and business logic. It is responsible for querying the database, validating data, and handling business rules. In **Express.js**, you can use libraries like **Mongoose** for MongoDB or **Sequelize** for SQL databases to handle models.

```
// models/User.js

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

  name: String,

  email: String,

  password: String,

});

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

module.exports = User;
```

2. View

The **View** is the UI representation of the data. In Express.js, Views can be rendered using a **template engine** like **Pug (formerly Jade)**, **EJS**, or others. The controller will send the necessary data to the view to display the content.

```
// views/user.pug

doctype html

html

  head

    title User Page

  body

    h1 Welcome, #{user.name}!
  p Email: #{user.email}
```

3. Controller

The **Controller** processes requests from the client, interacts with the **Model** to get or manipulate data, and then selects the appropriate **View** to render the data. *The Controller is the crucial part on an MVC pattern it is the part that does the request handling data fetching from the database through the model if needed and responding back to the client with a compiled view.*

```
// controllers/userController.js

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

// Controller function to render user page

const getUserPage = async (req, res) => {

 try {

  const user = await User.findById(req.params.id);

  if (!user) return res.status(404).send('User not found');

  res.render('user', { user });

 } catch (error) {

  res.status(500).send('Server Error');

 }

};

// Controller function to create a new user

const createUser = async (req, res) => {

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

    const newUser = new User({ name, email, password });

    await newUser.save();

    res.redirect(`/users/${newUser._id}`);

  } catch (error) {

    res.status(500).send('Server Error');

  }

};

module.exports = { getUserPage, createUser };
```

Step-by-Step Implementation of MVC in Express.js:

Step 1: Set up an Express Project

1. **Initialize a new Node.js project** and install necessary dependencies:

   ```
   mkdir express-mvc-app
   cd express-mvc-app
   npm init -y
   npm install express mongoose pug
   ```

2. **Create the project structure**:

   ```
   ├── app.js
   ├── controllers
   │   └── userController.js
   ├── models
   ```

```
│    └── User.js
├── views
│    └── user.pug
└── routes
     └── userRoutes.js
```

Step 2: Set Up Express Server

1. In `app.js`, set up the Express application, connect to MongoDB, and configure routing:

```
const express = require('express');
const mongoose = require('mongoose');
const app = express();
const port = 3000;

// Connect to MongoDB
mongoose.connect('mongodb://localhost:27017/express
mvc', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
}).then(() => console.log('MongoDB Connected'))
  .catch(err => console.log('MongoDB connection
error:', err));

// Set the view engine to Pug
app.set('view engine', 'pug');

// Middleware to parse JSON and form data
app.use(express.json());
app.use(express.urlencoded({ extended: false }));

// Import routes
const userRoutes = require('./routes/userRoutes');
app.use('/users', userRoutes);

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

Step 3: Create the Model

1. Define the **User Model** in `models/User.js` using **Mongoose** to handle database interactions.

```
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  password: String,
});

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

Step 4: Create the Controller

1. Create **controller functions** in `controllers/userController.js` to handle business logic for routes.

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

// Controller to get the user page
const getUserPage = async (req, res) => {
  try {
    const user = await User.findById(req.params.id);
    if (!user) return res.status(404).send('User not
found');
    res.render('user', { user });
  } catch (error) {
    res.status(500).send('Server Error');
  }
};

// Controller to create a new user
const createUser = async (req, res) => {
  try {
    const { name, email, password } = req.body;
    const newUser = new User({ name, email, password });
    await newUser.save();
    res.redirect(`/users/${newUser._id}`);
  } catch (error) {
    res.status(500).send('Server Error');
  }
};

module.exports = { getUserPage, createUser };
```

Step 5: Create Routes

1. Define routes for handling user-related operations in `routes/userRoutes.js`.

```
const express = require('express');
const { getUserPage, createUser } =
require('../controllers/userController');
const router = express.Router();

// Route to get a user by ID
router.get('/:id', getUserPage);

// Route to create a new user
router.post('/', createUser);

module.exports = router;
```

Step 6: Create Views

1. Create a Pug template for displaying the user in `views/user.pug`.

```
pug
Copy code
doctype html
html
  head
    title User Details
  body
    h1 User: #{user.name}
    p Email: #{user.email}
```

Step 7: Test the Application

1. Run your application:

```
node app.js
```

2. Test the **create user** route by sending a POST request (using Postman or any REST client) to `http://localhost:3000/users` with JSON data:

```
{
  "name": "John Doe",
  "email": "john@example.com",
  "password": "password123"
}
```

3. Open the browser and navigate to `http://localhost:3000/users/USER_ID` to see the user details.

The **MVC** pattern in **Express.js** helps in organizing your codebase by separating concerns:

- **Model**: Handles data and business logic (e.g., database).
- **View**: Handles presentation and user interfaces (e.g., templates).
- **Controller**: Handles requests, processes the business logic, and returns responses to the client (e.g., managing routes).

By using **MVC**, your code is more organized, easier to maintain, and scalable for large applications.

Routing in Express.js:

Routing in **Express.js** refers to the mechanism that defines how an application responds to client requests to different endpoints (URIs) using different HTTP methods (e.g., GET, POST, PUT, DELETE). Each route is associated with a callback function that handles the request and sends the appropriate response.

Routing is a fundamental part of Express.js for building web applications and APIs, as it allows developers to map URLs to specific pieces of logic in the application.

Key Concepts of Routing in Express.js

1. **Route Methods**: These correspond to HTTP methods like `GET`, `POST`, `PUT`, and `DELETE`, and define how the server responds to a request made to a specific endpoint.
2. **Route Paths**: Paths are the part of the URL that identifies the resource or action, like `/home`, `/users`, or `/about`.
3. **Route Handlers (Middleware)**: Functions that are executed when a request to a specific route and HTTP method is matched. You can have multiple handlers for a single route.
4. **Route Parameters**: Variables in the route path that can capture dynamic values from the URL, such as `/users/:id` (where `:id` is a parameter).

Basic Route Structure

A basic route in Express is defined using the following syntax:

```
app.METHOD(PATH, HANDLER)
```

- **app**: The Express instance.
- **METHOD**: The HTTP method (e.g., `get`, `post`, `put`, `delete`, etc.).
- **PATH**: The URL endpoint or path on the server.
- **HANDLER**: A function that is executed when the route is matched.

Example of Basic Routes

Here's an example that shows different types of routes using various HTTP methods:

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

// GET route for the homepage
app.get('/', (req, res) => {
  res.send('Welcome to the Home Page!');
});

// POST route for creating a new user
app.post('/users', (req, res) => {
  res.send('User created!');
});

// PUT route for updating a user by ID
app.put('/users/:id', (req, res) => {
  res.send(`User with ID ${req.params.id} updated!`);
});

// DELETE route for deleting a user by ID
app.delete('/users/:id', (req, res) => {
  res.send(`User with ID ${req.params.id} deleted!`);
});

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

In this example:

- `GET /` responds with "Welcome to the Home Page!".
- `POST /users` responds with "User created!".
- `PUT /users/:id` responds with a message showing the `id` from the URL.
- `DELETE /users/:id` deletes the user with the specific `id`.

Route Parameters

Route parameters allow you to capture values from the URL and use them in your route handler. These values are dynamic, making the route flexible for different requests.

Example:

```
app.get('/users/:id', (req, res) => {
  res.send(`User ID: ${req.params.id}`);
});
```

- If you visit `http://localhost:3000/users/123`, the response will be `User ID: 123`.
- `:id` is a route parameter that captures the dynamic part of the URL.

Cookies and Sessions in Express.js:

Cookies and **Sessions** are mechanisms used in web development to store data related to user interaction on a website, enabling persistent user sessions and maintaining state across multiple HTTP requests.

Here's an explanation of how both work in **Express.js** and their key differences:

Cookies in Express.js:

A **cookie** is a small piece of data that is sent from the server to the client (browser) and stored on the client-side. Cookies are used to remember stateful information (e.g., user preferences, session identifiers) between HTTP requests since HTTP is a stateless protocol.

Key Characteristics of Cookies:

- **Stored on the client-side** (browser).
- Can hold small amounts of data (typically up to 4 KB).
- Can be set with an expiration time to specify how long they should be kept.
- Can be configured as **secure** or **HTTP-only** to restrict access.

Setting and Accessing Cookies in Express.js:

In Express.js, you can use the `cookie-parser` middleware to handle cookies easily.

1. **Install `cookie-parser`**:

   ```
   npm install cookie-parser
   ```

2. **Setting up Cookies in Express:** Here's how to set and access cookies using the `cookie-parser` middleware.

   ```
   const express = require('express');
   const cookieParser = require('cookie-parser');
   const app = express();

   // Use cookie-parser middleware
   app.use(cookieParser());
   ```

```
// Set a cookie
app.get('/set-cookie', (req, res) => {
  res.cookie('user', 'John Doe', { maxAge: 900000,
httpOnly: true });
  res.send('Cookie has been set');
});

// Access a cookie
app.get('/get-cookie', (req, res) => {
  const user = req.cookies.user;
  res.send(`User stored in cookie: ${user}`);
});

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

- **Setting a cookie**: `res.cookie()` is used to send a cookie to the client. You can specify options like `maxAge` (expiration time) and `httpOnly` (making it accessible only via HTTP, not JavaScript).
- **Accessing a cookie**: `req.cookies` is used to retrieve the cookies sent by the client.

Sessions in Express.js:

A **session** stores data on the server side and associates it with a unique session ID that is sent to the client in the form of a cookie. This ID is used to maintain user-specific data between requests. Unlike cookies, sessions store larger amounts of data securely on the server and allow for more complex data structures (e.g., user login information).

Key Characteristics of Sessions:

- **Stored on the server-side**.
- A session ID is stored on the client-side in a cookie.
- Can hold more complex and sensitive data (e.g., authentication tokens).
- Session data persists until it expires or is manually destroyed.

Setting Up Sessions in Express.js:

To use sessions in Express, you need the `express-session` middleware.

1. **Install `express-session`**:

   ```
   npm install express-session
   ```

2. **Setting up Sessions in Express:** Here's how to create and manage sessions.

   ```
   const express = require('express');
   const session = require('express-session');
   const app = express();

   // Use express-session middleware
   app.use(session({
     secret: 'mysecretkey', // Secret key to sign the session
   ID cookie
     resave: false,          // Prevents session from being
   saved back to the store if it wasn't modified
     saveUninitialized: true, // Saves a new session even if
   it's not modified
     cookie: { maxAge: 60000 } // Set cookie expiration to 1
   minute
   }));

   // Set a session variable
   app.get('/set-session', (req, res) => {
     req.session.user = 'John Doe';
     res.send('Session data has been set');
   });

   // Access session data
   app.get('/get-session', (req, res) => {
     const user = req.session.user;
     res.send(`Session data for user: ${user}`);
   });

   // Destroy the session
   app.get('/logout', (req, res) => {
     req.session.destroy((err) => {
       if (err) {
         return res.status(500).send('Failed to destroy
   session');
       }
       res.send('Session has been destroyed');
   ```

```
  });
});

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

- **Session setup**: `express-session` creates a session with a unique session ID, stores session data on the server, and sends the session ID to the client as a cookie.
- **Setting session data**: You can store any data on `req.session` (e.g., `req.session.user = 'John Doe'`).
- **Accessing session data**: You can access session data in subsequent requests using `req.session`.
- **Destroying a session**: The session can be destroyed using `req.session.destroy()` (useful for logging out users).

Differences Between Cookies and Sessions

Feature	Cookies	Sessions
Storage location	Stored on the client-side (browser).	Stored on the server-side.
Data capacity	Limited to about 4 KB of data.	Can store large amounts of data.
Security	Less secure (as they are stored on the client-side).	More secure (data is stored on the server).
Data type	Stores only strings.	Can store any data type (objects, arrays, etc.).
Duration	Cookies persist even after closing the browser (if set).	Sessions typically expire when the user closes the browser or after a specified timeout.

Feature	Cookies	Sessions
Usage	Best for storing non-sensitive, small amounts of data.	Best for storing sensitive and complex data like authentication details.

When to Use Cookies vs. Sessions

- **Cookies**: Use cookies when you need to store small pieces of non-sensitive data like user preferences, session IDs, or items in a shopping cart.
- **Sessions**: Use sessions when you need to store sensitive information like user authentication, large datasets, or complex objects that are not suitable for storage on the client side.

Conclusion

- **Cookies**: Store small data directly in the browser and are suitable for lightweight, non-sensitive data that you want to persist between requests.
- **Sessions**: Store data on the server and are used for handling user data securely across requests, especially for user authentication, tracking logged-in states, and other sensitive information.

By using cookies and sessions appropriately in **Express.js**, you can manage state effectively and create more dynamic, secure, and personalized web applications.

HTTP Interaction in Express.js:

HTTP interaction refers to the communication between a client (browser, mobile app, etc.) and a server via the **HTTP protocol**. In Express.js, HTTP interaction happens when the server receives requests from the client and sends back appropriate responses.

Basic HTTP Interaction in Express.js

When a client sends a request to the server, the request typically consists of:

- **HTTP Method**: Specifies the action (e.g., `GET`, `POST`, `PUT`, `DELETE`).
- **Headers**: Additional metadata about the request (e.g., content type, authorization tokens).
- **Body**: Data sent in the request, usually in `POST` and `PUT` methods (e.g., form data or JSON).
- **URL**: Specifies the path or resource being requested.

User Authentication in Express.js

User Authentication is the process of verifying the identity of a user. In Express.js, this is typically done by validating credentials (e.g., username, password) and providing access tokens or session data to maintain a user's login state.

Authentication can be handled in various ways, but two common approaches are:

1. **Session-based authentication**.
2. **Token-based authentication (using JSON Web Tokens - JWT)**.

1. Session-Based Authentication

In **session-based authentication**, after a user successfully logs in, the server creates a session for the user, stores it in memory or a database, and sends a session ID to the client as a cookie. The client sends this session ID with every request to verify the user.

- **Login**: The user logs in with credentials. If valid, a session is created and stored on the server.
- **Protected Route**: A route like `/dashboard` requires an active session to allow access.

- **Logout**: Ends the session and logs the user out.

2. Token-Based Authentication (Using JWT)

In **token-based authentication**, after a user logs in, the server generates a **JWT (JSON Web Token)**, which is sent to the client. The client stores the token (usually in local storage) and sends it with every request for authentication.

Key Differences: Session-Based vs. Token-Based Authentication

Feature	Session-Based Authentication	Token-Based Authentication (JWT)
Storage location	Session data is stored on the server.	Token is stored on the client-side (localStorage, etc.).
Scalability	Not as scalable, as session data is stored server-side.	More scalable, as no server-side session storage is needed.
State	Server stores session state (session ID, user data).	Stateless, as the token itself contains all the information.
Data Size	Sessions can store more complex and larger data.	Tokens have a size limit, storing only basic info.
Security	More secure for sensitive data, as it’s stored server-side.	Tokens can be less secure if stored improperly on the client-side.