## Page:1

# **Moongose-Part-2-&MVC**

# 

**What is Mongoose?**

Mongoose is a tool that helps you work with MongoDB (a database) using JavaScript. It makes it easier to interact with the database by giving you a simple way to structure your data and perform actions like saving, updating, and deleting.

**What is a Schema?**

A Schema is like a blueprint or plan for how your data will look in the database. It defines the structure of the data, like what fields (e.g., name, age) it will have and what type of data (e.g., string, number) each field will hold.

**What is a Model?**

A Model is like a wrapper around the Schema. It allows you to work with the data in your database using the structure defined by the Schema. With a Model, you can create, read, update, and delete documents in your database.

**How to Create a Document Using Schema and Model?**

1. **Define a Schema**: First, you create a Schema to define the structure of your data. For example:

javascriptCopy code

const userSchema = new mongoose.Schema({

name: String,

age: Number

## Page:2

});

Create a Model: Next, you create a Model based on the Schema.

javascriptCopy code

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

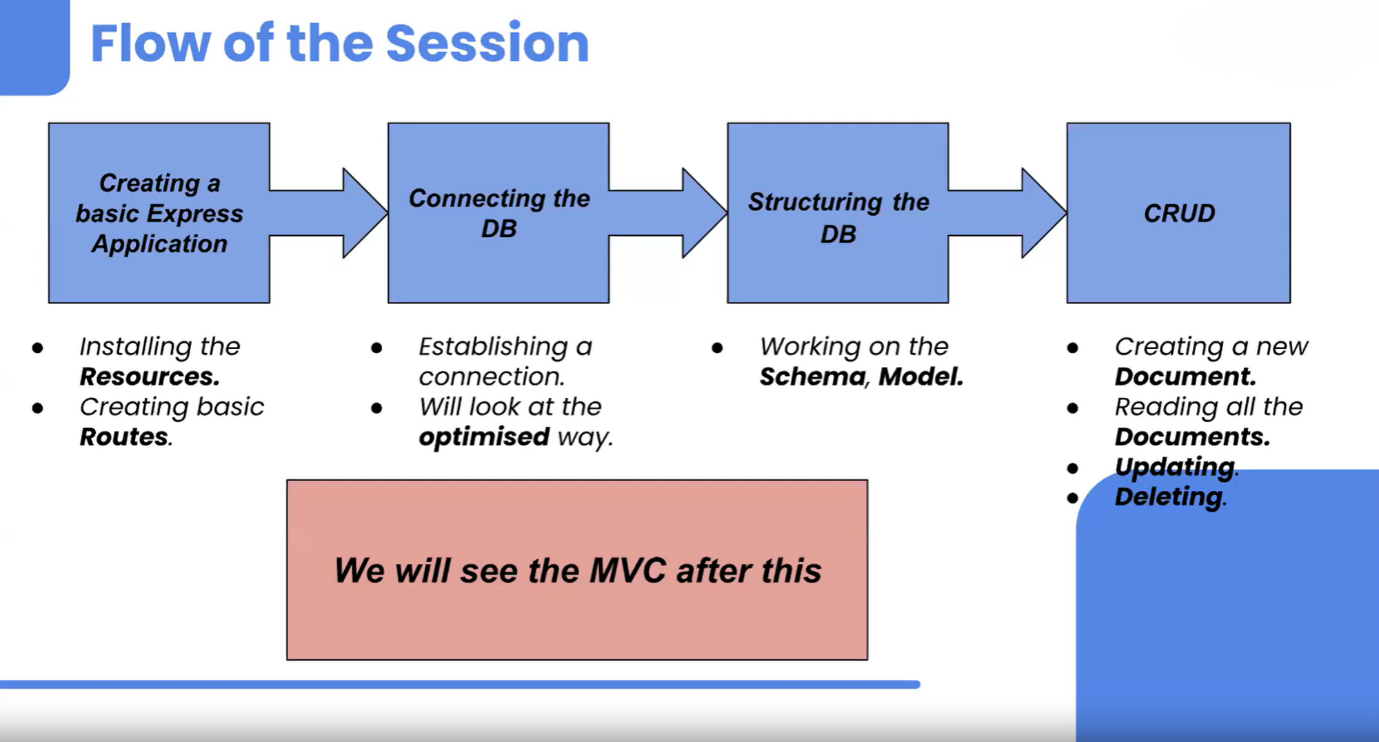
​

Create a Document: Finally, you use the Model to create a new document (a record in the database) and save it.

javascriptCopy code

const newUser = new User({ name: 'Alice', age: 25 });

newUser.save();



Part-1

**Step 1: Set Up the Project**

Create a New Directory:

Open your terminal and create a new directory for your project:

bashCopy code

mkdir my-express-app

## Page:3

cd my-express-app

**Initialize a New Node.js Project:**

Initialize your project with npm (Node Package Manager):

bashCopy code

npm init -y

This command creates a package.json file with default settings.

**Step 2: Install Express**

Install Express:

Install Express, a web framework for Node.js, by running:

bashCopy code

npm install express

**Step 3: Create the Basic Express Application**

Create an Entry File:

Create a file named app.js (or index.js) in the root of your project directory:

bashCopy code

touch index.js

​

**Write the Basic Express Code:**

Open index.js in your text editor and add the following code:

javascriptCopy code

const express = require('express');

const app = express();

// Define a route

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

res.send('Hello, World!');

});

## Page:4

**// Start the server**

const PORT = 8080;

app.listen(PORT, () => {

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

});

**Step 4: Run the Application**

Run the Express Application:

In the terminal, run the application using Node.js:

bashCopy code

node app.js

Access the Application:

Open your web browser and navigate to http://localhost:3000. You should see "Hello, World!" displayed on the page.

## Optional: Install Additional Basic Resources

If you need more packages like nodemon for automatic server restarts, you can install it as a development dependency:

Shell

Copy

bashCopy code

npm install nodemon

​

Run the app with nodemon for automatic restarts:

Shell

Copy

bashCopy code

nodemon index.js

## Page:5

**Part-2(Connection-TO-Db)**

**Step 1: Install Mongoose**

First, install Mongoose by running the following command in your project directory:

Shell

Copy

bashCopy code

npm install mongoose

​

**Step 2: Create a Database Connection File**

Create a new file called db.js in the root of your project:

Shell

Copy

bashCopy code

touch db.js

**Step 3: Establish a Database Connection**

Open the db.js file and add the following code. Here, we'll directly include the MongoDB URI in the code:

javascriptCopy code

const mongoose = require('mongoose');

const connection = await mongoose.connect('url');

module.exports = connection ;

**Step 4: Modify index.js to Use the Database Connection**

Now, create and open your index.js file and modify it to use the connectDB function:

bashCopy code

touch index.js

Then, add the following code to index.js:

javascriptCopy code

const express = require('express');

## Page:6

const connection= require('./db');

const app = express();

// Define a route

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

res.send('Hello, World!');

});

// Start the server

const PORT = 8080;

app.listen(PORT, async() => {

// Connect to the database

try{

await connection

}catch{

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

}

});

**Step 5: Run Your Application**

Finally, run your application using the following command:

bashCopy code

node index.js

**Explanation:**

* **Direct URI**: The MongoDB connection URI is hard-coded in the db.js file. This approach can be useful for quick setups but is not recommended for production environments where sensitive information should be kept secure.
* **Async/Await**: The database connection function uses async/await for cleaner and more manageable asynchronous code.
* **Error Handling**: The application will log any connection errors and exit if the connection fails.
* **Index.js**: The application is started from index.js, as you requested.

This setup provides a straightforward way to connect your Express application to MongoDB without using environment variables, keeping all the configurations within the code.

## Page:7

**Part-3(Strucutring The DB)**

**Step 1: Create the Schema and Model**

We'll define a User Schema and Model inside the db.js file.

**db.js:**

1. Open your db.js file and define the User Schema and Model.

javascriptCopy code

const mongoose = require('mongoose');

const connection = await mongoose.connect('url');

// Define User Schema

const userSchema = new mongoose.Schema({

name: String,

email: String,

password: String,

age: String,

date: Date // Automatically sets the current date

});

// Create User Model

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

module.exports = { connection , User };

**Explanation of the Code:**

* **User Schema**:
  + **name**: A required string field. The trim option ensures there are no spaces before or after the name.
  + **email**: A required and unique string field. The lowercase option converts the email to lowercase.
  + **password**: A required string field that would typically store a hashed password.
  + **age**: An optional number field that defaults to 0 if not provided.

## Page:8

* + **date**: A date field that automatically gets the current date and time when a new user is created.
* **User Model**: The User model is created from the userSchema. This model allows you to interact with the users collection in your MongoDB database.

**Step 2: Use the Model in index.js**

Next, we'll use the User model in our index.js file to create a new user.

**index.js:**

1. Open your index.js file and modify it as follows:

javascriptCopy code

const express = require('express');

const { connection , User } = require('./db');

const app = express();

// Middleware to parse JSON

app.use(express.json());

// Connect to the database

// Define a route to create a new user

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

try {

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

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

await user.save();

res.status(201).json(user); // Send back the created user

} catch (error) {

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

## Page:9

}

});

const PORT = 8080;

app.listen(PORT, async() => {

// Connect to the database

try{

await connection

}catch{

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

}

});

**Explanation of the Code:**

* **Express JSON Middleware**: app.use(express.json()) allows the server to parse incoming JSON requests.
* **Route to Create a New User**:
  + The POST /users route accepts user data in the request body and creates a new user document using the User model.
  + The user is saved to the database, and a success response with the created user is returned.
  + If there's an error (e.g., missing required fields), it sends a 400 error with the error message.

**Step 3: Run the Application**

1. Start your server:

bashCopy code

node index.js

You can now send a POST request to http://localhost:3000/users with a JSON body like:

jsonCopy code

{

"name": "Alice",

"email": "alice@example.com",

## Page:10

"password": "secret123",

"age": 25

}

**Part-4(Cred)**

**Step 1: Set Up the Project and Install Dependencies**

Make sure you've already set up your project with Express and Mongoose as described earlier.

**Step 2: Create and Update index.js**

Now, let’s write the code for each CRUD operation in index.js.

javascriptCopy code

const express = require('express');

const { connection , User } = require('./db');

const app = express();

// Middleware to parse JSON

app.use(express.json());

;

// CREATE: Add a new user

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

try {

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

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

await user.save();

res.status(201).json(user); // Return the created user

} catch (error) {

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

## Page:11

}

});

// READ: Get all users

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

try {

const users = await User.find(); // Retrieve all users

res.status(200).json(users);

} catch (error) {

res.status(500).json({ message: error.message });

}

});

// READ: Get a user by ID

app.get('/users/:id', async (req, res) => {

try {

const user = await User.findById(req.params.id); // Retrieve user by ID

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

res.status(200).json(user);

} catch (error) {

res.status(500).json({ message: error.message });

}

});

// UPDATE: Update a user by ID

app.put('/users/:id', async (req, res) => {

try {

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

const user = await User.findByIdAndUpdate(

## Page:12

req.params.id,

{ name, email, password, age },

);

res.status(200).json(user);

} catch (error) {

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

}

});

// DELETE: Delete a user by ID

app.delete('/users/:id', async (req, res) => {

try {

const user = await User.findByIdAndDelete(req.params.id); // Delete user by ID

res.status(200).json({ message: 'User deleted successfully' });

} catch (error) {

res.status(500).json({ message: error.message });

}

});

const PORT = 8080;

app.listen(PORT, async() => {

// Connect to the database

try{

await connection

}catch{

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

## Page:13

}

});

**Explanation of CRUD Operations:**

1. **CREATE**:
   * **Route**: POST /users
   * **Description**: Adds a new user to the database using the data provided in the request body. Responds with the created user document.
2. **READ (All Users)**:
   * **Route**: GET /users
   * **Description**: Retrieves all users from the database and responds with an array of user documents.
3. **READ (Single User by ID)**:
   * **Route**: GET /users/:id
   * **Description**: Retrieves a single user by their unique ID. Responds with the user document if found, otherwise returns a 404 error.
4. **UPDATE**:
   * **Route**: PUT /users/:id
   * **Description**: Updates an existing user’s details using their unique ID. The updated user document is returned if the update is successful.
5. **DELETE**:
   * **Route**: DELETE /users/:id
   * **Description**: Deletes a user by their unique ID. Responds with a success message if the deletion is successful.

### Step 3: Run the Application

Start your Express application:

bashCopy code

node index.js

**Testing the CRUD Operations**

You can test these routes using tools like **Postman** or **cURL**:

1. **Create a User**:
   * POST <http://localhost:3000/users>

## Page:14

* + Body (JSON):

jsonCopy code

{

"name": "Alice",

"email": "alice@example.com",

"password": "secret123",

"age": 25

}

​

**Get All Users**:

GET http://localhost:3000/users

**Get a User by ID**:

GET http://localhost:3000/users/:id

Replace :id with the actual user ID.

**Update a User**:

PUT http://localhost:3000/users/:id

Body (JSON):

jsonCopy code

{

"name": "Alice Updated",

"email": "alice.new@example.com",

"password": "newsecret123",

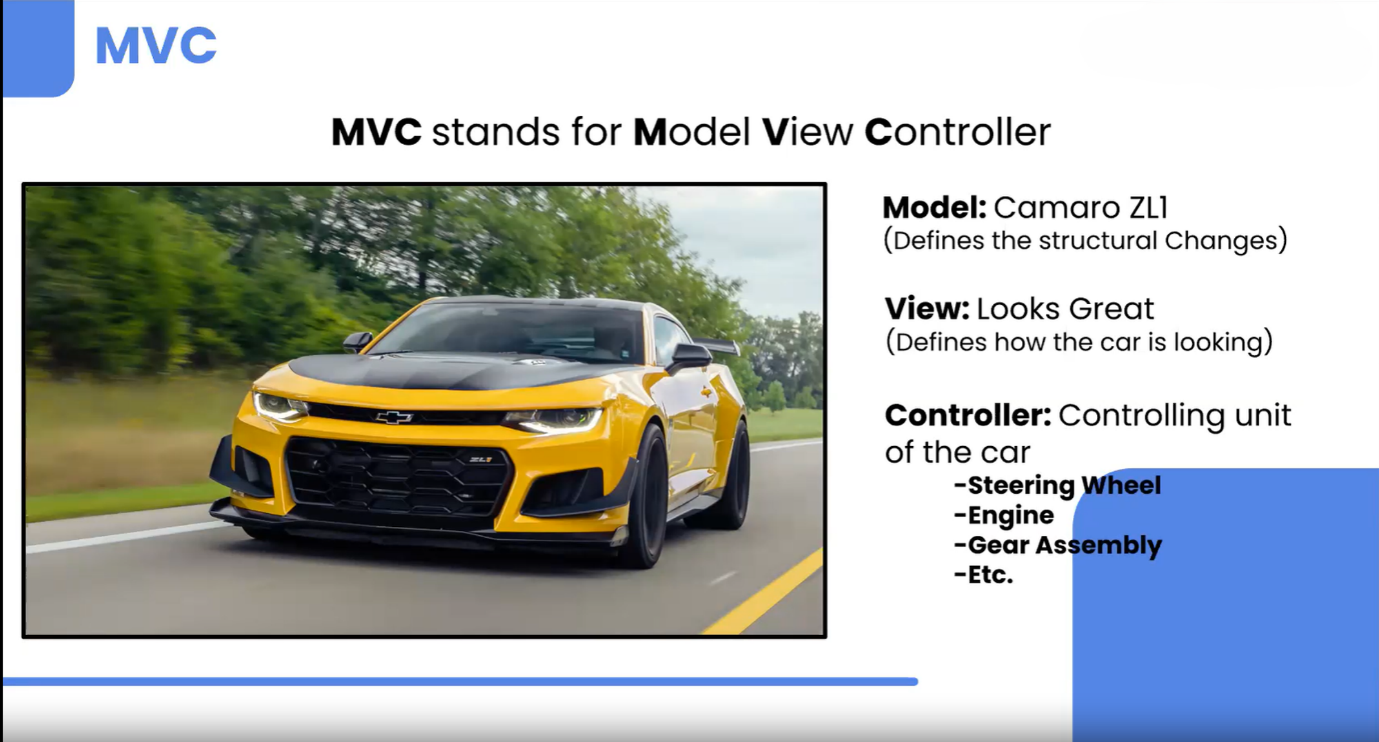
## Page:15

"age": 26

}

1. **Delete a User**:
   * DELETE <http://localhost:3000/users/:id>

This code provides a full implementation of CRUD operations using Express and Mongoose, all within the index.js file.



**Understanding MVC (Model-View-Controller)**

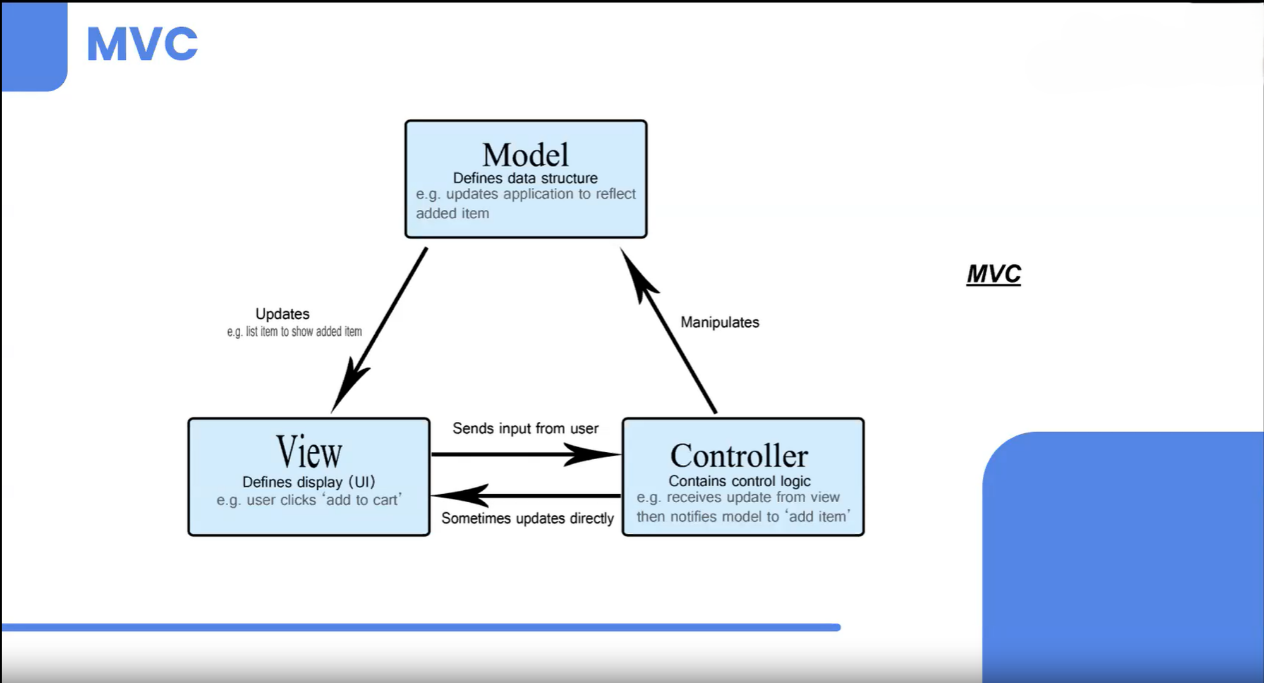
**MVC** is a software design pattern commonly used for developing user interfaces that separates an application into three interconnected components:

1. **Model**:
   * **What It Is**: The Model represents the data and the business logic of the application. It directly manages the data, logic, and rules of the application.
   * **Responsibilities**:
     + Interacts with the database to retrieve, update, or delete data.
     + Applies business rules and logic.
     + Notifies the View when data changes.
   * **Example**: In a blogging application, the Model would handle the data structure for blog posts, comments, and user information.
2. **View**:
   * **What It Is**: The View represents the user interface of the application. It displays the data provided by the Model in a specific format.

## Page:16

* + **Responsibilities**:
    - Presents data to the user.
    - Receives user input and sends it to the Controller.
  + **Example**: The View could be a web page displaying a list of blog posts with options to add, edit, or delete posts.

1. **Controller**:
   * **What It Is**: The Controller acts as an intermediary between the Model and the View. It listens to user inputs from the View, processes them (often updating the Model), and returns the results to the View.
   * **Responsibilities**:
     + Handles user input.
     + Calls the appropriate actions on the Model.
     + Updates the View with the new data.
   * **Example**: The Controller could process a user’s request to submit a new blog post, save it to the database (via the Model), and update the View to display the new post.



### MVC Folder Structure in a Node.js/Express Application

my-express-app/

│

├── models/ # Model Layer

│ ├── userModel.js # Defines the User schema and model

## Page:17

│ └── postModel.js # Defines the Post schema and model (e.g., for a blogging app)

│

├── views/ # View Layer

│ ├── userViews/ # Views related to users

│ │ ├── profile.ejs # User profile view

│ │ └── login.ejs # Login page view

│ └── postViews/ # Views related to posts

│ ├── postList.ejs # View to display a list of posts

│ └── postDetail.ejs # View to display a single post detail

│

├── controllers/ # Controller Layer

│ ├── userController.js # Handles user-related actions (login, register, etc.)

│ └── postController.js # Handles post-related actions (create, delete, etc.)

│

├── routes/ # Define routes and map them to controllers

│ ├── userRoutes.js # Routes for user actions

│ └── postRoutes.js # Routes for post actions

│

## Page:18

**Advantages of the MVC Pattern:**

* **Separation of Concerns**: Each component (Model, View, Controller) has a distinct responsibility, making the codebase easier to manage and maintain.
* **Scalability**: The application is easier to scale because different parts of the code are decoupled.
* **Reusability**: Components can be reused in different parts of the application or even in other applications.

This structure helps keep your code organized, making it easier to manage, debug, and scale as your application grows.