Full Stack Development – I

SQL Commands

1. CREATE Table and INSERT Data

- **CREATE TABLE**: Creates a new table with a specific schema (column definitions, data types, constraints).
    - **Data Types**:
        - INT: For integers.
        - VARCHAR(n): For text strings of a maximum length n.
        - DATE: For dates.
        - BOOLEAN: For true/false values.
    - **Constraints**:
        - PRIMARY KEY: Ensures each row has a unique identifier.
        - NOT NULL: Prevents null values in the column.
        - DEFAULT: Specifies a default value for a column.
    - Example:

```
CREATE TABLE Employees (
  EmployeeID INT PRIMARY KEY,
  Name VARCHAR(100) NOT NULL,
  Department VARCHAR(50),
  HireDate DATE,
  Salary DECIMAL(10, 2) DEFAULT 30000.00
);
```

- **INSERT INTO**: Adds records to the table.
    - Example:

```
INSERT INTO Employees (EmployeeID, Name)
VALUES (1, 'Alice Johnson');
```

2. READ, SELECT, and WHERE

- **SELECT**: Retrieves data from specific columns or all columns (*) of a table.
    - Example:

```
SELECT Name, Salary FROM Employees;
```

- **WHERE**: Filters records based on conditions.
    - Operators:
        - =: Equals.

        - != or <>: Not equal.
        - >, <, >=, <=: Comparison.
        - LIKE: Pattern matching (e.g., % for any characters).
    - Example:

      ```
      SELECT * FROM Employees WHERE Department = 'HR' AND Salary > 40000;
      ```

3. DELETE

- **DELETE**: Removes specific rows from a table.
    - Example:

      ```
      DELETE FROM Employees WHERE EmployeeID = 1;
      ```

4. Updating Single Values and Adding Columns

- **UPDATE**: Modifies data in existing rows.
    - Example:

      ```
      UPDATE Employees
      SET Salary = 45000.00
      WHERE EmployeeID = 1;
      ```

- **ALTER TABLE**: Modifies the structure of a table.
    - Adding a column:

      ```
      ALTER TABLE Employees ADD PhoneNumber VARCHAR(15);
      ```

5. Understanding SQL Relationships

SQL relationships define how data in one table connects to data in another.

- **One-to-One**: Each record in Table A relates to one record in Table B.
- **One-to-Many**: One record in Table A relates to multiple records in Table B.
- **Many-to-Many**: Many records in Table A relate to many in Table B (achieved via a junction table).

6. Foreign Keys and Inner Joins

- **Foreign Keys**: Ensure data consistency by linking two tables.
    - Example:

```
CREATE TABLE Orders (
  OrderID INT PRIMARY KEY,
  EmployeeID INT,
  Amount DECIMAL(10, 2),
  FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID)
);
```

- **INNER JOIN**: Combines rows from two tables based on a common field.
  - Example:

```
SELECT Employees.Name, Orders.Amount
FROM Employees
INNER JOIN Orders ON Employees.EmployeeID = Orders.EmployeeID;
```

Introduction to NoSQL

What is NoSQL?

- **Definition**: NoSQL databases provide flexible, schema-less storage for modern applications, especially those handling unstructured or semi-structured data.
- **Why NoSQL?**:
  - Scalability: Easily scales horizontally by adding servers.
  - Flexibility: Schema-less design allows quick adjustments to data structures.
  - High Performance: Optimized for fast read/write operations.

When to Use NoSQL?

- Applications that need to handle:
  - High-volume data.
  - Real-time processing.
  - Schema changes or diverse data types.

Types of NoSQL Databases

1. Key-Value Stores

- Stores data as key-value pairs.
- **Use Cases**:
  - Caching.
  - Session management.
  - Simple configurations.
- **Example (Redis)**:

```
SET user:1001 "John Doe"
GET user:1001
```

2. Document Stores

- Stores data as JSON-like documents.
- Supports rich querying on nested fields.
- **Use Cases**:
  - Content management systems.
  - E-commerce catalogs.
- **Example (MongoDB)**:

```
db.products.insertOne({
  _id: 1,
  name: "Laptop",
  brand: "Dell",
  price: 799.99,
  specs: {
    processor: "Intel i5",
    ram: "8GB"
  }
});

db.products.find({ "specs.ram": "8GB" });
```

3. Column-Family Stores

- Organizes data into rows and columns but with more flexibility than relational databases.
- **Use Cases**:
  - Analytics platforms.
  - High-volume transactional applications.
- **Example (Cassandra)**:

```
CREATE TABLE users (
  user_id UUID PRIMARY KEY,
  name TEXT,
  email TEXT,
  signup_date TIMESTAMP
);

INSERT INTO users (user_id, name, email, signup_date)
VALUES (uuid(), 'Alice', 'alice@example.com', now());
```

4. Graph Databases

- Models data as nodes and edges.
- **Use Cases**:
  - Social networks.
  - Fraud detection.
  - Recommendation engines.
- **Example (Neo4j)**:

```
CREATE (a:Person {name: 'John'})
CREATE (b:Person {name: 'Jane'})
CREATE (a)-[:FRIEND]->(b);
```

NoSQL Data Modeling

Design Principles:

1. **Data Denormalization**:
   - Combine related data into one document to avoid joins.
   - Example (MongoDB):

```
{
 _id: 1,
 customer: "Alice",
 orders: [
   { order_id: 101, amount: 49.99 },
   { order_id: 102, amount: 19.99 }
 ]
}
```

2. **Query-Driven Design**:
   - Design the schema based on the queries your application needs.
3. **Avoiding Joins**:
   - NoSQL databases prioritize embedding related data over linking it.

NoSQL Database Management

CRUD Operations in NoSQL

- **Create**:
  - MongoDB Example:

```
db.users.insertOne({ name: "Alice", age: 30 });
```

- **Read**:
  - MongoDB Example:

```
db.users.find({ age: { $gte: 25 } });
```

- **Update**:
  - MongoDB Example:

```
db.users.updateOne(
  { name: "Alice" },
  { $set: { age: 31 } }
);
```

- **Delete**:
  - MongoDB Example:

```
db.users.deleteOne({ name: "Alice" });
```

Querying and Indexing

- **Indexing**:
  - Increases query performance by creating indices.
  - Example:

```
db.users.createIndex({ age: 1 });
```

- **Querying Nested Fields**:
  - Example:

```
db.products.find({ "specs.processor": "Intel i5" });
```

Overview of Node.js

Node.js is an open-source, cross-platform JavaScript runtime environment that allows developers to execute JavaScript code server-side. Node.js uses the V8 JavaScript engine, which is the same engine used by Google Chrome, to convert JavaScript code into machine code.

Node.js was developed by Ryan Dahl in 2009, and it has become one of the most popular platforms for building web applications, APIs, and server-side applications due to its high performance and scalability.

Benefits of Using Node.js Compared to Traditional Web Servers

1. **Non-blocking I/O and Asynchronous Nature**:

    - Node.js uses non-blocking, event-driven architecture which allows handling multiple requests simultaneously. This is a significant advantage over traditional web servers like Apache, which use blocking I/O.
    - Asynchronous programming in Node.js ensures that operations like reading from a file or querying a database do not block the execution of other operations.
2. **Single Programming Language**:
    - With Node.js, both the client-side and server-side of an application can be written in JavaScript. This streamlines development, as developers can use a single language for the entire stack.
3. **High Performance**:
    - Node.js uses the V8 JavaScript engine, which compiles JavaScript to machine code before execution, resulting in faster performance.
    - The event-driven architecture and non-blocking I/O make Node.js highly efficient in handling large numbers of concurrent connections with minimal overhead.
4. **Scalability**:
    - Node.js applications can be scaled horizontally and vertically. Horizontal scaling involves adding more instances of the application, while vertical scaling involves adding more computational power (CPU, memory) to an existing instance.
5. **Rich Ecosystem**:
    - Node.js has a vast ecosystem of open-source libraries and modules available through the Node Package Manager (npm). This makes it easy to add functionality to applications without having to build everything from scratch.

Asynchronous Programming and Event-Driven Architecture

Node.js's architecture is centred around asynchronous programming and an event-driven model. This is achieved through the following mechanisms:

- **Event Loop**:
    - The event loop is the core of Node.js's runtime environment. It continuously checks for events and callbacks to process. When an operation completes, its callback is pushed to the event loop, which then processes it.
- **Callbacks**:

    - Functions passed as arguments to other functions are known as callbacks. They are invoked once an asynchronous operation completes. This allows other code to execute while waiting for the operation to finish.

- **Promises**:
    - Promises provide a way to handle asynchronous operations more cleanly than callbacks. They represent a value that may be available now, in the future, or never. Promises have .then(), .catch(), and .finally() methods to handle the results of asynchronous operations.
- **Async/Await**:
    - async and await are syntactic sugar built on top of promises, making asynchronous code look and behave more like synchronous code. async functions return promises, and await can be used to pause the execution of an async function until a promise is resolved.

Setting Up Node.js Environment

Installation and Configuration of Node.js

1. **Download and Install Node.js**:
    - Visit the official Node.js website (https://nodejs.org/) and download the installer for your operating system.
    - Follow the installation instructions specific to your OS. The installer includes Node.js and npm.
2. **Verify Installation**:
    - Open a terminal or command prompt and run the following commands to verify that Node.js and npm are installed correctly:

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

3. **Configuration**:

- Node.js doesn't require extensive configuration out of the box. However, you can configure the environment by setting environment variables such as `NODE_ENV` for different environments (development, production).

Introduction to the Node Package Manager (npm)

1. **What is npm?**:

- npm is the default package manager for Node.js. It is used to manage dependencies and packages for Node.js projects. It allows developers to easily install, share, and distribute code.

2. **Basic npm Commands**:

- **Initialize a Project**:

  npm init

  This command initializes a new Node.js project and creates a package.json file, which holds metadata about the project and its dependencies.

- **Install a Package**:

  npm install <package-name>

  This command installs a package and adds it to the node_modules directory. The package is also listed in the dependencies section of the package.json file.

- **Install a Package Globally**:

  npm -g install <package-name>

  This installs a package globally, making it available across all projects on the system.

- **Uninstall a Package**:

  npm uninstall <package-name>

  This command removes a package from the project and updates the package.json file.

- **Update Packages**:

```
npm update
```

This command updates all the packages listed in the package.json file to their latest versions.

- **List Installed Packages**:

```
npm list
```

This command lists all the packages installed in the current project.

3. **Package.json**:
   - The package.json file holds various metadata relevant to the project. This includes the project's name, version, description, main entry point, scripts, and dependencies.

Node Modules

An In-depth Look at Essential Node.js Modules

Node.js provides a rich set of built-in modules to handle various functionalities. Here are some essential modules:

1. **File System (fs) Module**: The fs module provides an API for interacting with the file system. It allows you to read, write, and manipulate files and directories.

   Example:

```
import fs from 'fs';

// Reading a file asynchronously
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data);
});

// Writing to a file asynchronously
const content = 'Hello, World!';
```

```
fs.writeFile('example.txt', content, (err) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log('File written successfully');
});
```

2. **HTTP Module**: The HTTP module allows Node.js to transfer data over the HyperText Transfer Protocol (HTTP). It can be used to create both HTTP clients and servers.

   Example:

```
import http from 'http';

// Creating an HTTP server
    const server = http.createServer((req, res) => {
        res.statusCode = 200;
        res.setHeader('Content-Type', 'text/plain');
        res.end('Hello, World!\n');
    });

    const hostname = '127.0.0.1';
    const port = 3000;
    server.listen(port, hostname, () => {
        console.log(`Server running at http://${hostname}:${port}/`);
    });
```

3. **Path Module**: The `path` module provides utilities for working with file and directory paths. It helps in resolving and constructing file paths in a platform-independent way.

   Example:

```
import path from 'path';

// Joining paths
```

```
        const fullPath = path.join('/users', 'joe', 'docs', 'file.txt');
        console.log(fullPath); // Output: /users/joe/docs/file.txt

// Resolving paths
        const resolvedPath = path.resolve('file.txt');
        console.log(resolvedPath); // Output: Absolute path to file.txt

// Extracting the file extension
        const ext = path.extname('file.txt');
        console.log(ext); // Output: .txt
```

Overview of Express.js :

1. **Introduction:** Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for building web and mobile applications. It's one of the most popular frameworks in the Node.js ecosystem, known for its simplicity and performance. Express.js allows developers to build server-side applications with Node.js easily, offering features like routing, middleware support, and template engines.

   Features of Express.js:

   - **Routing:** Express offers a powerful routing system that allows you to define routes for various HTTP methods and URL patterns.
   - **Middleware Support:** Middleware functions are central to Express. They can be used to process requests, add headers, authenticate users, handle errors, and more.
   - **Template Engines:** Express supports various template engines (e.g., Pug, EJS) that allow you to generate dynamic HTML. Default - jade
   - **Static File Serving:** Express can serve static files (images, CSS, JavaScript) easily with minimal configuration.
   - **Error Handling:** Express provides built-in mechanisms for handling errors effectively.
   - **Scalability:** Express can handle a large number of requests efficiently, making it suitable for building scalable applications.

   Benefits of Using Express.js:

Simplicity: Express is easy to learn and use, with a minimalistic approach that allows you to get up and running quickly.

- **Flexibility:** Express is unopinionated, meaning you have the freedom to structure your application as you see fit.
- **Middleware Ecosystem:** Express has a vast ecosystem of middleware modules, allowing you to add various functionalities like authentication, logging, and data parsing.
- **Integration:** Express integrates seamlessly with various databases, template engines, and other Node.js modules.
- **Community Support:** With a large community and extensive documentation, finding help and resources for Express is straightforward.

2. Setting Up Express.js

Steps to Install Express:

1. **Install Node.js:** Before using Express, you need to have Node.js installed on your machine. You can download it from the official Node.js website.
2. **Initialize a Node.js Project:**
   - Open your terminal or command prompt.
   - Navigate to your project directory.
   - Run npm init -y to create a package.json file.
3. **Install Express:**
   - Run npm install express to install Express in your project. This command will add Express as a dependency in your package.json file.

Creating a Basic Express Application:

1. **Create an Entry Point:**
   - Create a file named `app.js` or `index.js` in your project directory.
2. **Require Express:**

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

const app = express();
```

3. **Define a Route:**

- Add a basic route to handle HTTP GET requests to the root URL (/):

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

4. **Start the Server:**

- Add the following code to start the Express server on a specific port:

```
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

5. **Run the Application:**
   - Run `node app.js` in your terminal to start the server. Open your browser and navigate to `http://localhost:3000` to see your Express application in action.

3. Routing

Introduction to Routing: Routing refers to how an application's endpoints (URIs) respond to client requests. In Express, routing is handled using the app object. You can define routes for different HTTP methods and URL patterns.

Route Parameters: Route parameters are named URL segments that act as placeholders for specific values in the URL. They are defined by prefixing the variable name with a colon (:).

Example:

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

- If the user visits `/users/123`, the `userId` parameter will be `123`.

Query Strings: Query strings are the part of a URL that comes after the `?` symbol. They are typically used to send data to the server. In Express, you can access query string parameters using `req.query`.

Example:

```
app.get('/search', (req, res) => {
  const query = req.query.q;
  res.send(`Search query: ${query}`);
});
```

- If the user visits `/search?q=express`, the `q` parameter will be `express`.

Handling Different HTTP Methods: Express allows you to define routes for various HTTP methods like GET, POST, PUT, DELETE, etc.

Example:

```
app.post('/submit', (req, res) => {
  res.send('Form submitted!');
});
app.put('/update', (req, res) => {
  res.send('Resource updated!');
});
app.delete('/delete', (req, res) => {
  res.send('Resource deleted!');
});
```

Overview of Databases

SQL vs NoSQL Databases

SQL Databases:

- **Structured Query Language (SQL)** databases are relational databases where data is stored in tables (rows and columns).
- **Examples:** MySQL, PostgreSQL, SQLite, MS SQL Server.
- **Schema:** SQL databases have a fixed schema. This means the structure of the data must be defined before inserting data.
- **Data Integrity:** SQL databases enforce data integrity through constraints (primary key, foreign key, unique, etc.).
- **ACID Compliance:** SQL databases are ACID-compliant, ensuring atomicity, consistency, isolation, and durability for transactions.
- **Use Cases:** Suitable for applications with structured data and complex relationships (e.g., banking systems, accounting applications).

NoSQL Databases:

- **Not only SQL (NoSQL)** databases are non-relational and store data in various formats such as key-value pairs, documents, graphs, or columns.
- **Examples:** MongoDB, Cassandra, CouchDB, Redis.
- **Schema-less or Flexible Schema:** NoSQL databases often do not require a fixed schema, allowing for dynamic or evolving data structures.
- **Scalability:** NoSQL databases are often used for horizontal scaling (distributing data across many machines), making them highly scalable.
- **Eventual Consistency:** Many NoSQL databases focus on high availability and partition tolerance, so they prioritize "eventual consistency" over strict ACID compliance.
- **Use Cases:** Best for applications with unstructured or semi-structured data, real-time analytics, and large-scale data (e.g., social media apps, content management systems, IoT).

Comparison Summary:

Feature	**SQL**	**NoSQL**
Structure	Table-based (fixed schema)	Flexible (document, key-value, etc.)
Scalability	Vertical scaling (adding resources)	Horizontal scaling (adding servers)
Relationships	Strong (joins between tables)	Weak or none (embed data in documents)
Data Integrity	Strong (ACID compliance)	Eventual consistency
Examples	MySQL, PostgreSQL	MongoDB, Cassandra, Redis

Connecting to Databases: Using MongoDB with Mongoose

What is Mongoose?

- **Mongoose** is an Object Data Modeling (ODM) library for MongoDB and Node.js. It provides a schema-based solution to model your data and interact with MongoDB.
- It simplifies data validation, query building, and business logic integration.

Steps to Connect to MongoDB with Mongoose:

1. **Install Mongoose:**

   ```
   npm install mongoose
   ```

2. **Create a Connection to MongoDB:**

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

   mongoose.connect('mongodb://localhost/mydatabase', {
      useNewUrlParser: true,
      useUnifiedTopology: true
   })
   .then(() => console.log('Connected to MongoDB...'))
   .catch((err) => console.log('Error connecting to MongoDB:', err));
   ```

   - mongoose.connect() connects to the MongoDB database. Replace 'mongodb://localhost/mydatabase' with your MongoDB URI.

Basic CRUD Operations with Mongoose

1. **Create:** To create new data, define a **Mongoose model** and use it to save data.

   ```
   const mongoose = require('mongoose');
   const userSchema = new mongoose.Schema({
      name: String,
      email: String,
      age: Number
   });
   const User = mongoose.model('User', userSchema);
   // Create a new user
   const user =  new User.create({
      name: 'John Doe',
      email: 'john.doe@example.com',
   ```

```
    age: 25
});
user.save()
    .then(() => console.log('User saved'))
    .catch((err) => console.log('Error saving user:', err));
```

2. **Read:** To find data in the database, use the find() method.

```
User.find({ age: { $gt: 18 } })
    .then(users => console.log(users))
    .catch(err => console.log('Error fetching users:', err));
```

3. **Update:** To update an existing document, use the updateOne() or findOneAndUpdate() method.

```
User.updateOne({ name: 'John Doe' }, { age: 26 })
    .then(() => console.log('User updated'))
    .catch((err) => console.log('Error updating user:', err));
```

4. **Delete:** To remove a document, use the deleteOne() or findOneAndDelete() method.

```
User.deleteOne({ name: 'John Doe' })
    .then(() => console.log('User deleted'))
    .catch((err) => console.log('Error deleting user:', err));
```

Advanced Database Operations

Data Validation and Schema Design in Mongoose

In MongoDB, data does not follow a strict schema. However, with **Mongoose**, you can define schemas to enforce structure and data validation.

Schema Validation:

You can add validation rules to your Mongoose schema.

```
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,   // Name is mandatory
    minlength: [3, 'Name must be at least 3 characters']
  },
  email: {
    type: String,
    required: true,
```

```
    unique: true,     // Email must be unique
    match: [/\S+@\S+\.\S+/, 'Please provide a valid email address']
  },
  age: {
    type: Number,
    min: [18, 'Age must be 18 or older']
  }
});
```

In the above schema:

- **required** ensures the field is provided.
- **minlength** ensures the string is of a minimum length.
- **match** is used for regular expression matching (e.g., to validate email).
- **unique** ensures no duplicate emails.

Using Mongoose Virtuals:

A **virtual** is a property that doesn't get stored in the database but can be used like a regular property. It’s calculated from other fields.

Example: Virtual for Full Name

```
userSchema.virtual('fullName').get(function() {
  return this.firstName + ' ' + this.lastName;
});
```

You can use this fullName field as if it's part of the data, but it's not actually saved to the database.

Populating Data:

MongoDB allows you to reference documents in other collections. **Mongoose population** allows you to automatically replace the referenced object in queries.

Example: Using populate()

```
const postSchema = new mongoose.Schema({
  title: String,
  author: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User'  // References the 'User' model
  }
});
```

```
const Post = mongoose.model('Post', postSchema);

// Create a post with a reference to a user
const post = new Post({
  title: 'How to Learn MongoDB',
  author: user._id
});

post.save();

// Populate the 'author' field when querying posts
Post.find()
  .populate('author')
  .then(posts => console.log(posts))
  .catch(err => console.log('Error populating data:', err));
```

In this example:

- **Populate** automatically retrieves the full user data (instead of just the user's ID) when querying posts.

Understanding MVC Architecture

What is MVC?

MVC (Model-View-Controller) is a software design pattern that separates an application into three interconnected components:

- **Model:** Represents the data and business logic of the application. It communicates with the database to retrieve, store, and update data.
- **View:** Represents the user interface (UI). It displays the data received from the model to the user.
- **Controller:** Acts as an intermediary between the model and view. It receives user input from the view, processes it (with possible updates to the model), and sends data to the view for display.

Benefits of Using MVC in Web Applications

1. **Separation of Concerns:** MVC divides the application into three components, making it easier to manage and maintain the code.

2. **Reusability:** Since the components are independent, you can reuse them in other parts of the application or other applications.
3. **Maintainability:** Since the logic is separated, developers can easily identify and fix issues in a specific part of the application without affecting others.
4. **Scalability:** The architecture makes it easier to scale the application as new features are added.

When to Use MVC?

MVC is ideal for applications where the data and business logic are separate from the user interface, such as content management systems, online stores, and dashboards.

Implementing MVC in Node.js

Structuring a Node.js Application Using MVC Pattern

In a Node.js application, the MVC pattern helps organize the code efficiently. Below is an example of how you can structure a simple Node.js application using MVC:

```
myApp/
├── controllers/            # Contains all controller files
│   ├── userController.js   # Handles requests related to users
├── models/                 # Contains model files
│   ├── userModel.js        # Defines the structure of user data
├── views/                  # Contains views (UI templates)
│   ├── userView.ejs        # Displays user data
├── routes/                 # Contains route files
│   ├── userRoutes.js       # Manages routes for user operations
├── app.js                  # Main server setup
└── package.json            # Project configuration
```

Creating Models

Models define the structure of the data (such as a user) and how it should be manipulated. In Mongoose (for MongoDB), models are defined based on schemas.

Example: Defining a Model (userModel.js)

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

// Define the user schema
```

```
const userSchema = new mongoose.Schema({
  name: {
     type: String,
     required: true
  },
  email: {
     type: String,
     required: true,
     unique: true
  },
  age: {
     type: Number,
     required: true
  }
});

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

module.exports = User;
```

Creating Controllers

Controllers handle incoming requests, interact with models to fetch or modify data, and send responses to the views.

Example: User Controller (userController.js)

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

// Controller for getting all users
exports.getAllUsers = (req, res) => {
   User.find()
      .then(users => {
         res.render('userView', { users });
      })
      .catch(err => {
         res.status(500).send("Error fetching users");
```

```
        });
};

// Controller for adding a new user
exports.addUser = (req, res) => {
    const newUser = new User({
        name: req.body.name,
        email: req.body.email,
        age: req.body.age
    });

    newUser.save()
        .then(() => res.redirect('/users'))
        .catch(err => res.status(500).send("Error adding user"));
};
```

Creating Views

Views are typically templates that are rendered with dynamic data. In Node.js, you can use **EJS**, **Pug**, or other templating engines to create views.

Example: EJS View (userView.ejs)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>User List</title>
</head>
<body>
    <h1>Users</h1>
    <ul>
        <% users.forEach(function(user) { %>
            <li><%= user.name %> - <%= user.email %> - <%= user.age %></li>
        <% }); %>
    </ul>
</body>
</html>
```

In this example, the users array is passed to the view and rendered in an unordered list.

Integrating a Templating Engine into a Node.js Application

A **templating engine** allows you to generate dynamic HTML pages by embedding JavaScript code inside HTML. In this section, we will integrate **EJS** as the templating engine in a Node.js application.

Steps to Integrate EJS:

1. **Install EJS:**
   npm install ejs
2. **Set Up EJS in Your Application:** In your main app.js file, tell Express to use EJS as the view engine.

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

// Set the view engine to EJS
app.set('view engine', 'ejs');

// Body parser middleware to handle form submissions
app.use(express.urlencoded({ extended: true }));

// Define routes (for example, users route)
const userRoutes = require('./routes/userRoutes');
app.use('/users', userRoutes);

mongoose.connect('mongodb://localhost/myApp', { useNewUrlParser: true,
useUnifiedTopology: true })
   .then(() => console.log('Connected to MongoDB'))
   .catch(err => console.log('Error connecting to MongoDB', err));

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

Handling Routes in MVC

Defining and Managing Routes:

Routes are responsible for handling HTTP requests (GET, POST, PUT, DELETE) and directing them to the appropriate controller. Express.js is used to handle routing in Node.js applications.

Using Express.js for Routing:

You can define routes in separate route files and then import them into the main application file (app.js). Each route corresponds to a specific controller method.

Example: Defining Routes (userRoutes.js)

```
const express = require('express');
const router = express.Router();
const userController = require('../controllers/userController');
// Route to get all users
router.get('/', userController.getAllUsers);
// Route to add a new user
router.post('/add', userController.addUser);
module.exports = router;
```

In this example:

- **GET /users** will call the getAllUsers method from the userController.
- **POST /users/add** will call the addUser method from the userController.

Using Middleware:

Middleware functions in Express are used to modify request objects, add additional functionality (like authentication), or handle errors.

Example: Middleware for Logging

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

// Simple logging middleware
app.use((req, res, next) => {
   console.log(`${req.method} request made to: ${req.url}`);
   next();
});
```

Middleware is executed in the order in which it is defined. The next() function is used to pass control to the next middleware in the stack.

- **MVC Architecture** helps in organizing the code into distinct parts: Model (data), View (UI), and Controller (logic), which leads to cleaner and more maintainable code.
- **In Node.js**, the MVC pattern is commonly implemented using **Express.js** for routing, **Mongoose** for managing MongoDB models, and **EJS** for dynamic views.
- This separation of concerns makes it easier to scale the application, handle changes in the database or UI, and improve overall maintainability.
- **Routes** and **middleware** in Express.js play a significant role in managing the flow of requests and responses, making the application functional and efficient.

Overview of APIs

What is an API?

API stands for **Application Programming Interface**. It is a set of rules that allows different software applications to communicate with each other. APIs act as intermediaries, enabling data exchange and functionality between systems.

Why are APIs Important in Software Development?

APIs are crucial because:

- **Integration:** They allow different systems to interact and share data, even if they're built on different technologies.
- **Automation:** APIs allow systems to automatically request and exchange information, speeding up processes and reducing manual work.
- **Abstraction:** APIs hide the complexity of the underlying system, allowing developers to use pre-defined functions without needing to understand the internals.
- **Modularity:** They enable building modular applications where components can be updated or replaced independently.

REST Architecture:

REST (Representational State Transfer) is an architectural style for designing networked applications. It is based on stateless communication and uses standard HTTP methods like GET, POST, PUT, DELETE, etc., to perform operations on resources (data).

Advantages of REST:

- **Statelessness:** Each request from a client to a server must contain all the information the server needs to fulfill the request. The server doesn’t store any session state between requests.

- **Scalability:** REST APIs can handle large numbers of requests efficiently and scale easily.
- **Simplicity:** REST uses standard HTTP methods (GET, POST, PUT, DELETE) and simple URL-based endpoints for communication.
- **Language Agnostic:** Since REST uses HTTP, it can be used with any programming language.

API Requests

Structuring API Requests

API requests are usually made using the **HTTP protocol**. Here's how an API request is structured:

- **Method:** The HTTP method (GET, POST, PUT, DELETE, etc.) determines the action to be performed.
- **URL (Endpoint):** The URL defines the resource on which the action is performed (e.g., /users, /posts).
- **Headers:** Additional information like authorization tokens or content type (JSON, XML).
- **Body:** For POST, PUT, or PATCH requests, the body contains the data you want to send to the server (e.g., creating a new user, updating a post).

Making Server-Side API Requests:

In Node.js, we can use modules like **axios**, **fetch**, or **request** to make API calls from the server to another server.

Example using axios to make a GET request:

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

axios.get('https://jsonplaceholder.typicode.com/posts')
  .then(response => {
    console.log(response.data); // Data from API
  })
  .catch(error => {
    console.log('Error fetching data:', error);
  });
```

API Authentication:

Most APIs require some form of authentication to ensure that only authorized users can access the data or perform certain actions.

1. **Basic Authentication:** A username and password are sent in the request header.

```
axios.get('https://api.example.com/data', {
  auth: {
    username: 'user',
    password: 'password'
  }
});
```

2. **Bearer Token Authentication:** A token (e.g., JWT or OAuth token) is sent in the Authorization header.

```
axios.get('https://api.example.com/data', {
  headers: {
    'Authorization': 'Bearer YOUR_TOKEN_HERE'
  }
});
```

REST APIs:

REST APIs are widely used for web services. They are simple, use standard HTTP methods, and are easy to integrate with different clients (web apps, mobile apps, etc.).

Building Your Own API

Creating a Basic API for a Blog in Node.js

We'll use **Express.js** to set up a simple REST API for a blog, where we will implement CRUD operations.

1. **Install Express:**

```
npm install express
```

2. **Create the Application Structure:**

```
blog-api/
├── app.js              # Main file to run the application
├── controllers/        # Contains controller files
│   ├── postController.js  # Handles logic for blog posts
├── models/             # Contains model files
│   ├── postModel.js    # Defines post schema
```

```
├── routes/              # Contains route files
│   ├── postRoutes.js    # Routes for post operations
└── package.json         # Project dependencies
```

1. Define the Post Model (postModel.js):

Define the schema and model for blog posts using **Mongoose**.

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

// Define the post schema
const postSchema = new mongoose.Schema({
    title: { type: String, required: true },
    content: { type: String, required: true },
    author: { type: String, required: true },
    date: { type: Date, default: Date.now }
});

// Create a model for the posts
const Post = mongoose.model('Post', postSchema);

module.exports = Post;
```

2. Create the Post Controller (postController.js):

The controller contains logic for CRUD operations.

```
const Post = require('../models/postModel');

// Get all posts
exports.getPosts = (req, res) => {
    Post.find()
        .then(posts => res.json(posts))
        .catch(err => res.status(500).json({ error: 'Unable to fetch posts' }));
};

// Create a new post
exports.createPost = (req, res) => {
    const newPost = new Post({
```

```
        title: req.body.title,
        content: req.body.content,
        author: req.body.author
    });

    newPost.save()
        .then(post => res.json(post))
        .catch(err => res.status(500).json({ error: 'Unable to create post' }));
};

// Update a post
exports.updatePost = (req, res) => {
    Post.findByIdAndUpdate(req.params.id, req.body, { new: true })
        .then(post => res.json(post))
        .catch(err => res.status(500).json({ error: 'Unable to update post' }));
};

// Delete a post
exports.deletePost = (req, res) => {
    Post.findByIdAndDelete(req.params.id)
        .then(() => res.json({ message: 'Post deleted successfully' }))
        .catch(err => res.status(500).json({ error: 'Unable to delete post' }));
};
```

3. Define Routes for Post Operations (postRoutes.js):

This file maps the routes to the respective controller methods.

```
const express = require('express');
const router = express.Router();
const postController = require('../controllers/postController');

// GET all posts
router.get('/', postController.getPosts);

// POST a new post
router.post('/', postController.createPost);
```

```
// PUT to update a post
router.put('/:id', postController.updatePost);

// DELETE a post
router.delete('/:id', postController.deletePost);

module.exports = router;
```

4. Set Up the Server (app.js):

Set up Express and connect it to the routes and the database.

```
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');
const postRoutes = require('./routes/postRoutes');

const app = express();

// Middleware to parse JSON bodies
app.use(bodyParser.json());

// Routes
app.use('/api/posts', postRoutes);

// Connect to MongoDB
mongoose.connect('mongodb://localhost/blog-api', { useNewUrlParser: true,
useUnifiedTopology: true })
  .then(() => console.log('Connected to MongoDB'))
  .catch(err => console.log('Error connecting to MongoDB:', err));

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

API Routes

GET Route (Fetching all posts):

```
router.get('/', postController.getPosts);
```

- This route fetches all the posts from the database and sends them as a response in JSON format.

POST Route (Creating a new post):

```
router.post('/', postController.createPost);
```

- This route creates a new post by accepting data from the request body (e.g., title, content, and author).

PUT Route (Updating an existing post):

```
router.put('/:id', postController.updatePost);
```

- This route updates an existing post identified by the id parameter. It replaces the post's data with new information from the request body.

DELETE Route (Deleting a post):

```
router.delete('/:id', postController.deletePost);
```

- This route deletes a post identified by the id parameter.