MEAN Final PYQ

2marks

d) how express handles the requests in MEAN project?

Ans=>In a MEAN (MongoDB, Express.js, Angular, Node.js) project, **Express.js** serves as the **backend web framework** that handles HTTP requests. It acts as middleware between the client (Angular) and the database (MongoDB). Here's how it works:

1. **Routing:** Express defines routes (e.g., GET, POST, PUT, DELETE) to handle requests from the Angular frontend.
2. **Middleware:** It can process requests through middleware functions to handle tasks like authentication, validation, logging, and parsing request data.
3. **Request Handling:** Express receives client requests, processes them, and communicates with MongoDB through Mongoose or native drivers.
4. **Response:** After processing, Express sends back a response (JSON data or status) to the Angular frontend.

Example:

```
app.get('/api/users', (req, res) => {
  User.find().then(users => res.json(users));
});
```

This route handles a GET request for users and sends a JSON response.

e) How to Structuring an AngularJS web application?

To build a scalable AngularJS application, it is important to organize code into a **modular and maintainable structure**. A typical AngularJS project can be structured as follows:

1. Folder Structure:

```
/app
  /controllers     → Contains controller files
  /services        → Reusable business logic and API calls
  /directives      → Custom directives
  /views           → HTML templates (partials)
  /filters         → Custom filters
  /assets          → CSS, images, etc.
  app.js           → Main AngularJS module and routing
  index.html       → Main HTML file
```

2. Use of Modules:

AngularJS encourages breaking the app into **modules** using angular.module() to separate concerns (e.g., userModule, productModule).

3. Routing Configuration:

Use ngRoute or ui-router to define routes:

```
app.config(function($routeProvider) {
  $routeProvider
    .when('/home', {
      templateUrl: 'views/home.html',
      controller: 'HomeController'
    });
});
```

4. Controller Separation:

Each controller should handle the logic of a specific view:

```
app.controller('HomeController', function($scope) {
  $scope.message = "Welcome to Home Page";
});
```

5. Services and Factories:

Use them to handle reusable logic and API calls:

```
app.factory('UserService', function($http) {
  return {
    getUsers: () => $http.get('/api/users')
  };
});
```

6. Use of Custom Directives:

To create reusable UI components:

```
app.directive('userCard', function() {
  return {
    templateUrl: 'views/user-card.html'
  };});
```

f) Differentiate the concept of One-way and Two-way data binding

Aspect	One-way Data Binding	Two-way Data Binding
Definition	Data flows from the component to the view only	Data flows between component and view in both directions
Direction	Single direction (Component → View)	Bidirectional (Component ↔ View)
DOM Update	View updates when data changes in the component	View and component both update when either changes
Control	More control over how data flows	Less control, as synchronization is automatic
Complexity	Simpler to debug and manage	Can become complex in large applications
Used In	Angular, React (by default)	AngularJS (with ng-model), Angular (with [(ngModel)])
Example	{{ name }} – binds data one-way to template	<input ng-model="name"> – syncs input and model

e) Differentiate between various data models used for designing in MongoDB

Data Model	Description	Use Case	Example
Embedded Data Model	Stores related data in the same document	When related data is mostly accessed together	A blog post with embedded comments
Referenced Data Model	Uses references (_id) to link documents in different collections	When data is reused in multiple places	A post references a user document by user ID
Flat Data Model	Stores data in a flat, non-nested structure	For simple datasets and fast querying	A user document with name, age, email fields
Normalized Data Model	Similar to relational model; separates data into collections	To avoid duplication, better for large data sets	Separate collections for students and courses
Denormalized Data Model	Combines related data to avoid joins and improve read performance	When read performance is more important than storage	Product document includes embedded category info

e) What is the main use of node in Mean Stack?

Main Uses of Node.js in MEAN Stack:

1. **Server-side Execution:**
   It allows JavaScript to run on the server, enabling full-stack development using a single language (JavaScript).
2. **Non-blocking I/O:**
   Node.js handles multiple client requests efficiently using its event-driven, asynchronous architecture.
3. **Backend Support:**
   Acts as the foundation for Express.js, which handles routing and APIs.
4. **Real-time Applications:**
   Useful for building real-time apps like chat apps and live notifications using WebSockets.
5. **Package Management:**
   Provides npm (Node Package Manager) to install libraries and dependencies easily.

Differentiate Express.js and Node.js

Aspect	**Node.js**	**Express.js**
Definition	Node.js is a runtime environment that allows execution of JavaScript on the server side.	Express.js is a lightweight web application framework built on top of Node.js.
Type	Runtime Environment	Framework
Purpose	Used to build server-side and networking applications	Used to build web applications and RESTful APIs
Level of Abstraction	Low-level APIs	High-level abstraction
HTTP Handling	Requires manual handling of requests and responses using http module	Provides simplified methods like app.get(), app.post() etc.
Speed of Development	Slower, more boilerplate code needed	Faster, with built-in routing and middleware support
Routing	Must be implemented manually	Built-in routing system makes it easy to define endpoints
Middleware Support	No built-in support	Built-in support for middleware (e.g., for logging, authentication)
File Structure	No structure enforced	Encourages MVC pattern and better organization

Aspect	Node.js	Express.js
Third-party Integration	Manually configured	Easy integration with third-party middleware like body-parser, cors, etc.
Learning Curve	Requires deeper understanding of core modules	Easier to learn after understanding Node.js basics

4Marks

Discuss the concepts of Mongoose schema. Also write a code how to a create schema.

Discuss the Concepts of Mongoose Schema

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It provides a way to define the **structure of documents** and **data validation** using **schemas**.

Concepts of Mongoose Schema:

1. **Schema Definition:**
   A Mongoose schema defines the structure of a MongoDB document — what fields it will have and what data types those fields are.
2. **Data Types:**
   Fields in a schema can have types like String, Number, Date, Boolean, Array, or even nested objects.
3. **Validation:**
   Schemas allow for built-in and custom validation rules (e.g., required, minLength).
4. **Defaults:**
   Fields can have default values if no data is provided.
5. **Methods and Statics:**
   You can define custom methods (for instances) and statics (for the model) within the schema.
6. **Middleware (Hooks):**
   Functions that run before or after certain actions (e.g., save, remove).
7. **Virtuals:**
   Computed properties that are not stored in the database but behave like regular fields.

Code Example: Creating a Schema in Mongoose

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

// Define a schema
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  age: { type: Number, default: 18 },
  createdAt: { type: Date, default: Date.now }
});

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

// Export the model
module.exports = User;
```

Q6. How to design a single-page web application with Angular?

A **Single Page Application (SPA)** loads a single HTML page and dynamically updates content without reloading the page. Angular is a powerful framework for building SPAs using components, routing, and services.

Steps to Design an SPA with Angular:

1. Set Up Angular Project:

Use Angular CLI to create a new project:

```
ng new my-spa-app
cd my-spa-app
ng serve
```

2. Create Components:

Each section of the SPA is built as a component.

```
ng generate component home
ng generate component about
```

3. Configure Routing:

Define navigation routes using Angular's Router module.

```
// app-routing.module.ts
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent }
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

4. Add Navigation Links:

Use routerLink to switch between views.

```
<!-- app.component.html -->
<nav>
  <a routerLink="/">Home</a>
  <a routerLink="/about">About</a>
</nav>
<router-outlet></router-outlet>
```

5. Use Services for Data Handling:

Create services to fetch and share data.

```
ng generate service data
```

6. Dynamic Content Loading:

Data is loaded and updated dynamically in components without page refresh.

Illustrate, how the cornponents of the simple Angular app fit together'? Explain its working.

How the Components of a Simple Angular App Fit Together

In an Angular application, different components, services, and modules work together to create a seamless user experience. Let's break down how they interact in a **simple Angular app**.

1. Angular Modules:

- **App Module (app.module.ts)**: This is the root module of an Angular application. It imports necessary Angular modules (like BrowserModule, FormsModule, etc.) and declares components that are part of the application.

2. Components:

- **Component (app.component.ts)**: The fundamental building block of an Angular application. A component controls a part of the user interface (UI). Each component consists of:
  - **HTML Template**: Defines the structure of the view.
  - **CSS Styles**: Defines the look and feel of the component.
  - **TypeScript Class**: Contains logic and properties for the component.

3. Templates:

- A component's template defines the **view**. It binds to the data in the component's class using Angular's **binding syntax** like {{variable}} for interpolation and [(ngModel)] for two-way data binding.

4. Services:

- **Services**: Angular services are used to **share data** and **business logic** across different components. Services are injected into components via **dependency injection**.

5. Routing:

- **Routing (app-routing.module.ts)**: Angular uses the RouterModule to handle navigation between different views or components. It enables the dynamic loading of components without reloading the page.

6. Dependency Injection:

- Angular uses **Dependency Injection** to provide services to components.

Illustration of a Simple Angular App Structure

/src

/app

app.component.ts -> Main component (UI and logic)

app.component.html -> Template for UI

app.component.css -> Styles for the main component

app.module.ts -> Root module (imports BrowserModule, declarations of components)

app-routing.module.ts -> Routing module (defines routes for navigation)

/services

data.service.ts -> A service to fetch or manage data

/components

home.component.ts -> A component for the home page

about.component.ts -> A component for the about page

/models

user.model.ts -> Model representing a user (optional)

How it Works:

1. **App Module**:
    - The root module imports BrowserModule and AppRoutingModule to make the app work in a browser and handle routing.
    - It declares AppComponent and any other components that are part of the application.
2. **Component and Template Binding**:
    - AppComponent binds data from its TypeScript class to the template using Angular's **data binding**.
    - When data in the component changes, the view automatically updates (two-way binding).
3. **Routing**:
    - The router configuration in app-routing.module.ts defines paths like /home and /about, mapping them to HomeComponent and AboutComponent.
    - When the user clicks on a link (<a routerLink="/home">Home), the corresponding component is dynamically loaded without refreshing the page.

4. **Service Injection**:
    - The service (data.service.ts) is injected into the component to handle data fetching, such as from an API.
    - The component calls methods from the service to fetch data and updates the view based on the received data.

How do you configure an Express application to render views using a templating engine? Provide a brief example.

Express.js can be configured to render views by using a **templating engine**. A templating engine allows us to separate the HTML structure from the logic of the application and provides dynamic rendering capabilities.

Common templating engines for Express include **EJS**, **Pug**, and **Handlebars**. Below is how to configure an Express application to use a templating engine, **EJS**, for rendering views.

Steps to Configure Express with a Templating Engine (e.g., EJS):

1. Install the Templating Engine:

First, install the templating engine via npm. For **EJS**, run:

npm install ejs

2. Set the View Engine in Express:

In the **Express** app, specify the view engine and the location where views will be stored. This is done using the app.set() method.

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

const app = express();

// Set EJS as the view engine

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

// Specify the folder where view files will be stored (optional, default is './views')

app.set('views', './views');
```

3. Create Views:

Create an **EJS** file (or other templating engine files) in the specified **views** folder.

For example, create a file called index.ejs in the views folder:

```
<!-- views/index.ejs -->

<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">
```

```
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Express Example</title>
</head>
<body>
  <h1>Welcome, <%= name %>!</h1>
  <p>Your age is: <%= age %></p>
</body>
</html>
```

4. Render the View:

In the route handler, render the view using the res.render() method. You can pass dynamic data to the view as an object.

```
app.get('/', (req, res) => {
  const data = {
    name: 'John Doe',
    age: 30
  };
  // Render the 'index' view and pass data
  res.render('index', data);
});
```

5. Start the Server:

Finally, start the Express server to see the rendered view in the browser.

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

Full Example:

```
const express = require('express');
const app = express();
// Set the view engine to EJS
app.set('view engine', 'ejs');
// Define a route
app.get('/', (req, res) => {
```

```
  const data = {
    name: 'John Doe',
    age: 30
  };
  // Render the view and pass the data
  res.render('index', data);
});
// Start the server
app.listen(3000, () => {
  console.log('Server is running on http://localhost:3000');
});
```

Q6. Write a Node.js script that sets up a basic web server using the http module. The server should respond with "Hello, World!" when accessed at the root URL.
for 12 marks
How do you create a simple server in Node.js that returns Hello World?

```
// Load the http module to create an HTTP server
const http = require('http');
// Create an HTTP server that handles requests and responses
const server = http.createServer((req, res) => {
  // Check if the request URL is the root URL '/'
  if (req.url === '/') {
    // Set the response HTTP header
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    // Send the "Hello, World!" response
    res.end('Hello, World!\n');
  } else {
    // For other URLs, respond with a 404 (Not Found)
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Not Found\n');
  }
});
```

```
// Set the server to listen on port 3000
server.listen(3000, () => {
  console.log('Server running at http://localhost:3000/');
});
```

How It Works:

1. **Require the http module**:
   - This module allows you to create HTTP servers in Node.js.
2. **Create the HTTP server**:
   - The http.createServer() function is used to create the server.
   - The callback function receives the request (req) and response (res) objects.
3. **Handle requests**:
   - Inside the callback, we check if the request URL is / (the root).
   - If the URL is /, we send a 200 OK response with the content "Hello, World!".
   - If the URL is not /, we send a 404 Not Found response.
4. **Listen on port 3000**:
   - The server is set to listen on port 3000. When you navigate to http://localhost:3000/, the server responds with "Hello, World!".

Write an AngularJS component that demonstrates two-way data binding. The component should include a text input field that updates a displayed message as the user types.

AngularJS Two-Way Data Binding Component Example

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>AngularJS Two-Way Data Binding Example</title>
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>
</head>
<body ng-app="myApp">
  <div ng-controller="myController">
```

```
    <h1>Two-Way Data Binding Example</h1>
<!-- Text Input Field with Two-Way Data Binding -->
    <input type="text" ng-model="message" placeholder="Type something..." />
    <!-- Display the message dynamically as the user types -->
    <p>Your message: {{ message }}</p>
  </div>
  <script>
    // Define the AngularJS application
    var app = angular.module('myApp', []);
    // Define the controller for the app
    app.controller('myController', function($scope) {
      // Initialize the model variable
      $scope.message = '';
    });
  </script>
</body>
</html>
```

Explanation:

1. **AngularJS Script**:
   The script is included from the **Google CDN** (angular.min.js), which makes it easy to add AngularJS functionality to the page.
2. **ng-app Directive**:
   The ng-app="myApp" directive initializes the AngularJS application. It binds the AngularJS framework to the HTML document.
3. **ng-controller Directive**:
   The ng-controller="myController" directive binds the myController controller to the tag, which encapsulates the logic for this component.
4. **Two-Way Data Binding**:
   - The ng-model="message" directive binds the input field to the message model on the $scope. This creates a two-way data binding, meaning that any changes to the input field will automatically update the message variable in the controller and vice versa.
   - The {{ message }} in the <p> tag will dynamically update the displayed message as the user types in the input field.

5. **Controller**:
    - The controller (myController) initializes the message variable in the $scope to an empty string.
    - As the user types in the text input, the message variable is automatically updated, and the displayed text (Your message: {{ message }}) reflects the change in real-time.

Explain the Working of Routing and Services in Angular with Example

1. Routing in Angular

Routing in Angular allows navigation between different views or components without reloading the entire page. Angular uses the **RouterModule** to define routes and load components dynamically.

Key Concepts:

- **Routes**: Define path-to-component mapping.
- **RouterModule**: Angular module that handles navigation.
- **routerLink**: Directive to link to a route in templates.
- **router-outlet**: Placeholder where the routed component is displayed.

Example of Routing:

```
// app-routing.module.ts
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent }
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

```
<!-- app.component.html -->
<nav>
  <a routerLink="/">Home</a>
  <a routerLink="/about">About</a>
</nav>
<router-outlet></router-outlet>
```

Explanation:

- When the user clicks "Home", HomeComponent is rendered inside <router-outlet>.
- When "About" is clicked, AboutComponent is rendered.

2. Services in Angular

Services are used to organize and share data or logic across multiple components. They are commonly used for:

- Fetching data from an API
- Business logic
- Shared state

Creating a Service:

```
ng generate service data
```

data.service.ts

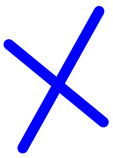
```
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: 'root'
})
export class DataService {
  getMessage() {
    return "Hello from Data Service!";
  }
}
```

Using Service in a Component:

```
// home.component.ts
import { Component, OnInit } from '@angular/core';
import { DataService } from '../data.service';
@Component({
  selector: 'app-home',
  template: `<h2>{{ message }}</h2>`
})
export class HomeComponent implements OnInit {
  message: string = '';
  constructor(private dataService: DataService) {}
  ngOnInit() {
    this.message = this.dataService.getMessage();
  }
}
```

Explanation:

- DataService provides a method getMessage().
- HomeComponent injects the service using **dependency injection** and calls the method in ngOnInit() to assign the message.



Q7. How to design a single-page web application with Angular?

A **Single-Page Application (SPA)** is a web app that loads a single HTML page and dynamically updates the view without reloading the whole page. **Angular** is well-suited for SPAs because of its powerful features like **components**, **routing**, and **data binding**.

Steps to Design a SPA Using Angular:

1. Set Up Angular Project

Use Angular CLI to generate a new project:

```
ng new my-spa-app
cd my-spa-app
```

2. Create Components

Each view/page in SPA is a separate Angular component.

Example:

```
ng generate component home
```

ng generate component about

3. Configure Routing

Define routes to connect URLs to components.

app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent }
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

4. Set Up Navigation

Use routerLink for navigation between components without reloading the page.

app.component.html

```
<nav>
  <a routerLink="/">Home</a>
  <a routerLink="/about">About</a>
</nav>
<router-outlet></router-outlet>
```

5. Run the Application

ng serve

Open your browser at http://localhost:4200.

What is the difference between MongoDB database, collection, and document?

In MongoDB, the terms **database**, **collection**, and **document** refer to different levels of data organization. Here's the difference between them:

1. Database

- A **database** is the highest level of data organization in MongoDB.
- It acts like a container for collections.
- Each database has its own set of files and is independent of others.
- Example: A database named school might contain data related to students, teachers, and courses.

2. Collection

- A **collection** is a group of MongoDB documents.
- It is the equivalent of a table in relational databases, but more flexible.
- Collections do **not enforce a schema**, so documents in the same collection can have different structures.
- Example: In the school database, you might have a collection called students.

3. Document

- A **document** is the basic unit of data in MongoDB.
- It is stored in **BSON** (Binary JSON) format, which allows rich data types.
- A document is like a row in a relational table but can store nested data.
- Example:

```
{
 "name": "Alice",
 "age": 20,
 "courses": ["Math", "Science"]
}
```

Summary Table:

Term	Description	Analogy in RDBMS
Database	Container for collections	Database
Collection	Group of documents	Table
Document	Individual data record in BSON format	Row (Record)

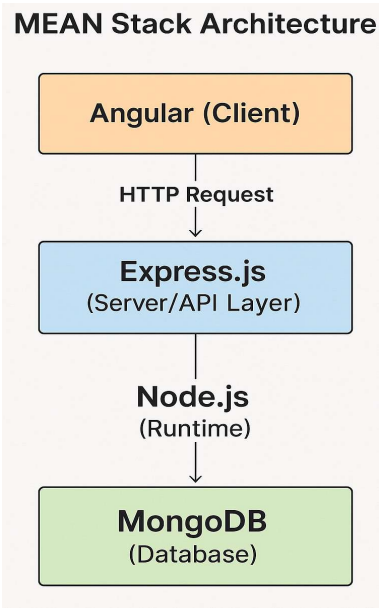
Discuss the role of Angular JS in the MEAN stackRole of AngularJS in MEAN Stack

1. **Front-End Development**: AngularJS provides a structured framework for building interactive and responsive web applications, handling UI components efficiently.
2. **Model-View-Controller (MVC) Architecture**: It organizes code in a logical manner, separating concerns between data (model), UI (view), and application logic (controller).
3. **Two-Way Data Binding**: AngularJS synchronizes the model and the view, reducing the need for manual DOM manipulations and making the development process more intuitive.
4. **RESTful API Integration**: It communicates with the Express.js backend via HTTP services, retrieving and updating data stored in MongoDB.
5. **Reusable Components**: Directives and components allow developers to create modular and reusable UI elements, enhancing code maintainability.
6. **Dependency Injection**: AngularJS simplifies service management and boosts efficiency by injecting dependencies where needed.
7. **Routing and Single-Page Applications (SPA)**: Using the built-in routing module, AngularJS ensures smooth transitions between views without requiring full page reloads.

12Marks
How to design a common MEAN stack architecture using a REST API built in

Node.js, Express, and MongoDB? Explain with detail examples.

Designing a common **MEAN stack architecture** involves integrating **MongoDB**, **Express.js**, **Angular**, and **Node.js** to build full-stack web applications. The **REST API** is typically built using **Node.js**, **Express**, and **MongoDB**, while **Angular** is used for the front-end.



✅ **Overview of MEAN Stack(Explain all it in details)**

- **MongoDB**: NoSQL database to store data in JSON-like format.
- **Express.js**: Web framework for Node.js to create API endpoints.
- **Angular**: Front-end framework for building dynamic SPAs (Single Page Applications).
- **Node.js**: JavaScript runtime for building the server-side logic.

🧱 Folder Structure (Backend - Node/Express)

```
/backend
│
├── server.js
├── /routes
│   └── studentRoutes.js
├── /models
│   └── Student.js
├── /controllers
│   └── studentController.js
```

⚙️ Step-by-Step: Build REST API with Node.js, Express, and MongoDB

1. Install Dependencies

```
npm init -y
npm install express mongoose cors body-parser
```

2. Connect to MongoDB (server.js)

```
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
const studentRoutes = require('./routes/studentRoutes');
const app = express();
app.use(cors());
app.use(express.json());
mongoose.connect('mongodb://localhost:27017/school', {
   useNewUrlParser: true,
   useUnifiedTopology: true
})
.then(() => console.log("MongoDB connected"))
.catch(err => console.error(err));
app.use('/api/students', studentRoutes);
app.listen(3000, () => console.log('Server running on port 3000'));
```

3. Define MongoDB Schema (models/Student.js)

```
const mongoose = require('mongoose');
const studentSchema = new mongoose.Schema({
   name: String,
   age: Number,
   courses: [String]
});
module.exports = mongoose.model('Student', studentSchema);
```

4. Create Controller (controllers/studentController.js)

```
const Student = require('../models/Student');
exports.getAllStudents = async (req, res) => {
   const students = await Student.find();
   res.json(students);
};
exports.createStudent = async (req, res) => {
   const newStudent = new Student(req.body);
   await newStudent.save();
   res.status(201).json(newStudent);
};



exports.getStudentById = async (req, res) => {
   const student = await Student.findById(req.params.id);
   res.json(student);
};
exports.updateStudent = async (req, res) => {
   const student = await Student.findByIdAndUpdate(req.params.id, req.body, { new: true });
   res.json(student);
};
exports.deleteStudent = async (req, res) => {
```

```
  await Student.findByIdAndDelete(req.params.id);
  res.json({ message: 'Student deleted' });
};
```

5. Set Up Routes (routes/studentRoutes.js)

```
const express = require('express');
const router = express.Router();
const studentController = require('../controllers/studentController');
router.get('/', studentController.getAllStudents);
router.post('/', studentController.createStudent);
router.get('/:id', studentController.getStudentById);
router.put('/:id', studentController.updateStudent);
router.delete('/:id', studentController.deleteStudent);
module.exports = router;
```

💡 **Sample API Endpoints**

- GET /api/students – List all students
- POST /api/students – Add a new student
- GET /api/students/:id – Get a student by ID
- PUT /api/students/:id – Update a student
- DELETE /api/students/:id – Delete a student

Write a code how the POST, GET, PUT, and DELETE ETTI) request methods to common CRUD operations in MEAN Web application with examples.

💻 **Backend: Node.js + Express + MongoDB**

1. Model (models/student.js)

```
const mongoose = require('mongoose');
const studentSchema = new mongoose.Schema({
  name: String,
  age: Number,
  course: String
```

```
});
module.exports = mongoose.model('Student', studentSchema);
```

2. Controller (controllers/studentController.js)

```
const Student = require('../models/student');
// GET all students
exports.getAll = async (req, res) => {
  const students = await Student.find();
  res.json(students);
};
// GET one student
exports.getById = async (req, res) => {
  const student = await Student.findById(req.params.id);
  res.json(student);
};
// POST create student
exports.create = async (req, res) => {
  const student = new Student(req.body);
  await student.save();
  res.status(201).json(student);
};

// PUT update student
exports.update = async (req, res) => {
  const updated = await Student.findByIdAndUpdate(req.params.id, req.body, { new: true });
  res.json(updated);
};
// DELETE student
exports.remove = async (req, res) => {
  await Student.findByIdAndDelete(req.params.id);
  res.json({ message: 'Deleted' });
```

```
};
```

3. Routes (routes/studentRoutes.js)

```
const express = require('express');
const router = express.Router();
const studentCtrl = require('../controllers/studentController');
router.get('/', studentCtrl.getAll);
router.get('/:id', studentCtrl.getById);
router.post('/', studentCtrl.create);
router.put('/:id', studentCtrl.update);
router.delete('/:id', studentCtrl.remove);
module.exports = router;
```

4. Server Entry (server.js)

```
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
const studentRoutes = require('./routes/studentRoutes');
const app = express();
app.use(cors());
app.use(express.json());
mongoose.connect('mongodb://localhost:27017/school', { useNewUrlParser: true });

app.use('/api/students', studentRoutes);
app.listen(3000, () => console.log('Server running on port 3000'));
```

🖥️ **Frontend: Angular Example Using HttpClient**

1. Angular Service (student.service.ts)

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
export interface Student {
  _id?: string;
```

```
  name: string;
  age: number;
  course: string;
}
@Injectable({
  providedIn: 'root'
})
export class StudentService {
  private apiUrl = 'http://localhost:3000/api/students';
  constructor(private http: HttpClient) {}
  getStudents(): Observable<Student[]> {
    return this.http.get<Student[]>(this.apiUrl);
  }
  getStudent(id: string): Observable<Student> {
    return this.http.get<Student>(`${this.apiUrl}/${id}`);
  }
  createStudent(data: Student): Observable<Student> {
    return this.http.post<Student>(this.apiUrl, data);
  }
  updateStudent(id: string, data: Student): Observable<Student> {
    return this.http.put<Student>(`${this.apiUrl}/${id}`, data);
  }
  deleteStudent(id: string): Observable<any> {
    return this.http.delete(`${this.apiUrl}/${id}`);
  }
}
```

2. Example Usage in Component (student.component.ts)

```
import { Component, OnInit } from '@angular/core';
import { StudentService, Student } from './student.service';
@Component({
```

```
  selector: 'app-student',
  template: `<ul><li *ngFor="let s of students">{{s.name}}</li></ul>`
})
export class StudentComponent implements OnInit {
  students: Student[] = [];
  constructor(private studentService: StudentService) {}
  ngOnInit() {
    this.studentService.getStudents().subscribe(data => {
      this.students = data;
    });
  }
  addStudent() {
    const newStudent: Student = { name: 'New', age: 20, course: 'CS' };
    this.studentService.createStudent(newStudent).subscribe();
  }
  updateStudent(id: string) {
    const updated: Student = { name: 'Updated', age: 22, course: 'Math' };
    this.studentService.updateStudent(id, updated).subscribe();
  }
  deleteStudent(id: string) {
    this.studentService.deleteStudent(id).subscribe();}}
```

1. Understanding Event-Driven Programming in JavaScript

JavaScript is **event-driven**, meaning it operates based on events such as user interactions, network requests, or timers. Instead of executing code sequentially, event-driven programming utilizes **callbacks** and **asynchronous functions** to respond to events efficiently.

Example:

```
const fs = require('fs');
fs.readFile('example.txt', 'utf8', (err, data) => {
    if (err) {
        console.error(err);
    } else {
```

```
        console.log(data);
    }
});
```

Here, the fs.readFile method reads a file asynchronously, and the callback function handles the response when the event completes.