**Module 1: Introduction to Full Stack Development**

**🔹 What is Web Development?**

Web development is the process of building and maintaining websites or web applications. It involves creating the structure, design, functionality, and interactivity of websites using various programming languages and tools. Web development can be divided into three main parts:

* **Frontend (Client-side):** What users see and interact with (e.g., buttons, forms, layouts).
* **Backend (Server-side):** The behind-the-scenes logic, servers, and databases that process and respond to user actions.
* **Database:** A storage system that holds and organizes data used by the backend.

Web development powers everything from blogs and e-commerce sites to complex social networks and cloud platforms.

**🔹 Client-side vs Server-side vs Database**

| **Aspect** | **Client-side (Frontend)** | **Server-side (Backend)** | **Database** |
| --- | --- | --- | --- |
| **Location** | Runs in the browser | Runs on the web server | Hosted on a database server |
| **Technology** | HTML, CSS, JavaScript | Node.js, Express, PHP, etc. | MongoDB, MySQL, PostgreSQL |
| **Responsibility** | UI/UX, layout, styling | Business logic, API handling | Data storage, retrieval |
| **Example** | Displaying a form | Validating form data | Saving user info |

**Example Flow:**  
A user fills out a signup form (client-side) → The data is sent to the server for processing (server-side) → The server stores the user’s information in the database.

**🔹 Overview of Frontend, Backend & Database**

* **Frontend (Presentation Layer):**
  + Technologies: HTML, CSS, JavaScript, React
  + Purpose: Building user interfaces and ensuring a responsive design
  + Example: A login form or a product listing page
* **Backend (Application Layer):**
  + Technologies: Node.js, Express.js
  + Purpose: Handles application logic, authentication, routing, etc.
  + Example: Processing login requests or handling payment transactions
* **Database (Data Layer):**
  + Technologies: MongoDB (NoSQL), MySQL (SQL)
  + Purpose: Stores and manages data in a structured or document-based format
  + Example: Storing user profiles or transaction history

**🔹 Introduction to MERN Stack**

The **MERN Stack** is a popular full stack JavaScript solution used to build dynamic web applications.

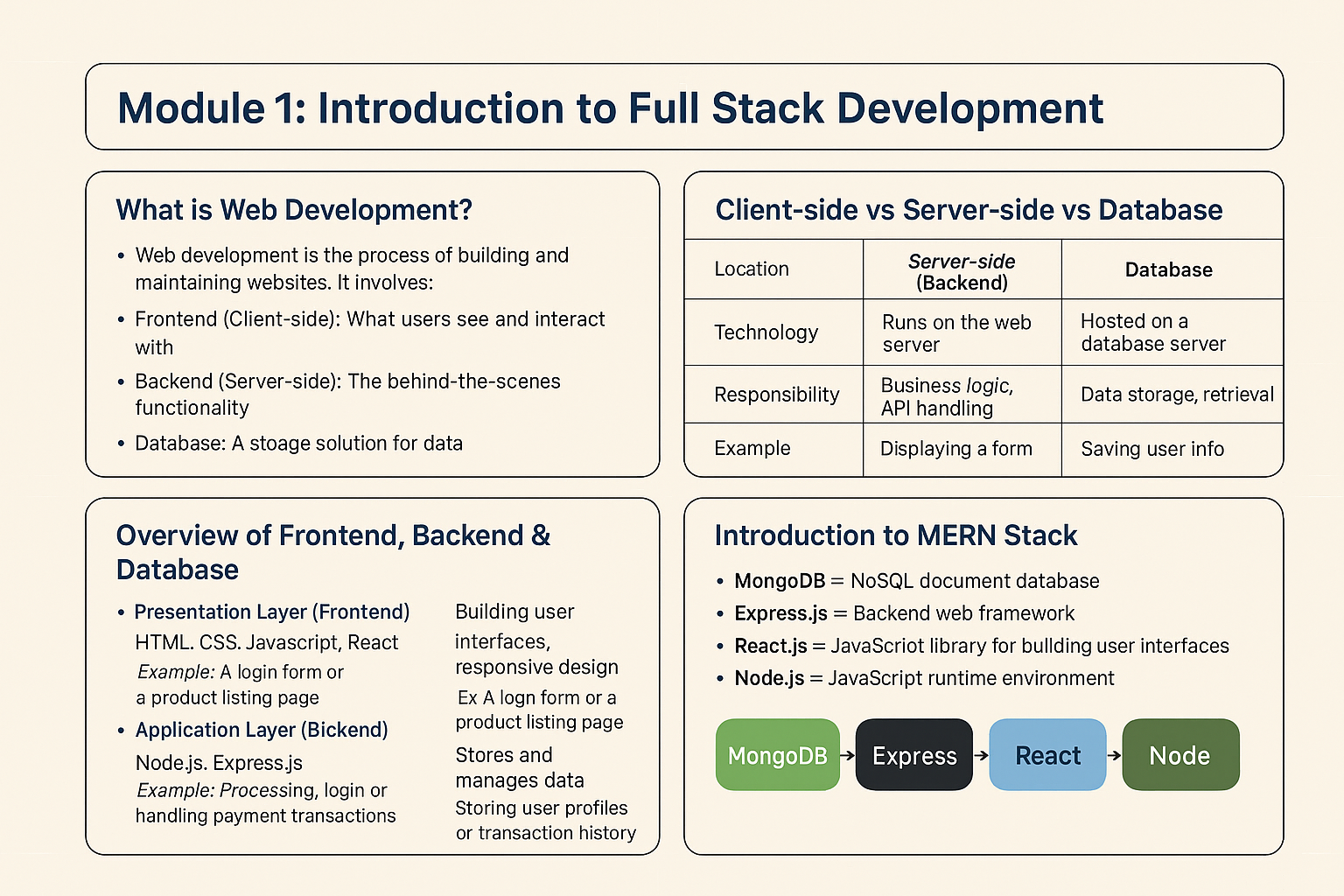
**MERN** stands for:

1. **MongoDB** (Database): A NoSQL document-based database used to store application data in flexible, JSON-like documents.
2. **Express.js** (Backend Framework): A minimal and flexible Node.js web application framework that handles routing, middleware, and server logic.
3. **React.js** (Frontend Library): A JavaScript library developed by Facebook for building modern, component-based user interfaces.
4. **Node.js** (Runtime Environment): A JavaScript runtime built on Chrome’s V8 engine that lets you run JavaScript on the server.

Together, these technologies allow developers to use JavaScript for both the **frontend and backend**, making development faster and more consistent.

**Example Workflow:**

* **React.js** builds the UI and sends requests to the server
* **Express.js + Node.js** handle the server-side logic
* **MongoDB** stores and retrieves the required data



**Module 2: Frontend Development Basics**

**🔹 HTML5: Structure of a Web Page & Semantic Elements**

**What is HTML5?**  
HTML5 (HyperText Markup Language) is the standard language for creating the structure of web pages.

**Basic Structure of a Web Page:**

<!DOCTYPE html>

<html>

<head>

<title>My First Page</title>

</head>

<body>

<h1>Welcome to My Website</h1>

<p>This is a paragraph.</p>

</body>

</html>

**Semantic Elements in HTML5:**  
Semantic elements = meaningful HTML.

Semantic elements in HTML5 are tags that clearly describe their meaning or role in a web page, both to the browser and to developers. Instead of using generic containers like <div> or <span>, semantic elements make the structure and purpose of the content more meaningful and easier to understand.

**Why Use Semantic Elements?**

* Improves code readability
* Helps search engines (SEO) understand your content
* Enhances accessibility for screen readers
* Makes maintenance easier

**Examples of Semantic Elements:**

| **Element** | **Meaning/Purpose** |
| --- | --- |
| **<header>** | **Represents the top section of a page or section (like a title or logo)** |
| **<nav>** | **Contains navigation links** |
| **<main>** | **The main content of the document** |
| **<section>** | **A thematic grouping of content** |
| **<article>** | **Self-contained content like blog posts or news articles** |
| **<aside>** | **Related side content (like ads, tips, or sidebars)** |
| **<footer>** | **The bottom section of a page or section** |
| **<figure> & <figcaption>** | **Used to display media and its caption** |

**Example with Semantic Elements:**

<body>

<header>

<h1>My Portfolio</h1>

</header>

<nav>

<a href="#about">About</a> |

<a href="#projects">Projects</a>

</nav>

<main>

<section id="about">

<h2>About Me</h2>

<p>I am a web developer.</p>

</section>

</main>

<footer>

<p>&copy; 2025 My Name</p>

</footer>

</body>

**🔹 CSS3: Styling Basics, Flexbox, Grid & Responsive Design**

**What is CSS?**  
**CSS** stands for **Cascading Style Sheets**. It is a language used to **style and design** web pages. While **HTML** provides the structure of a web page (like headings, paragraphs, images), **CSS** is what makes that page look good — by controlling **layout, colors, fonts, spacing, animations**, and more.

**Basic CSS Styling:**

<style>

body {

background-color: #f0f0f0;

font-family: Arial;

}

h1 {

color: darkblue;

text-align: center;

}

</style>

**Flexbox (Flexible Box Layout):**

Flexbox helps create flexible and responsive layouts.

<div style="display: flex; gap: 10px;">

<div style="background: lightblue; flex: 1;">Box 1</div>

<div style="background: lightgreen; flex: 1;">Box 2</div>

</div>

**CSS Grid:**

Grid provides a two-dimensional layout system.

<div style="display: grid; grid-template-columns: 1fr 1fr; gap: 10px;">

<div style="background: pink;">Box A</div>

<div style="background: orange;">Box B</div>

</div>

**Responsive Design (Media Queries):**

Make the site look good on all devices.

@media (max-width: 600px) {

body {

background-color: lightgray;

}

h1 {

font-size: 20px;

}

}

**🔹 JavaScript Basics: Variables, Loops, DOM Manipulation**

**What is JavaScript?**  
JavaScript is a **high-level, interpreted programming language** primarily used to create interactive and dynamic content on websites. It’s one of the **core technologies of the web**, alongside **HTML** (for structure) and **CSS** (for styling).

**Variables:**

let name = "John";

const age = 25;

var city = "New York";

console.log(name, age, city);

**Loops:**

for (let i = 1; i <= 5; i++) {

console.log("Number: " + i);

}

**DOM Manipulation:**

DOM = Document Object Model (the structure of your web page)

<p id="greeting">Hello!</p>

<button onclick="changeText()">Click Me</button>

<script>

function changeText() {

document.getElementById("greeting").innerText = "Welcome to JavaScript!";

}

</script>

**Mini Project Example: Responsive Profile Card**

<!DOCTYPE html>

<html>

<head>

<style>

.card {

width: 300px;

border: 1px solid #ddd;

border-radius: 10px;

padding: 20px;

text-align: center;

font-family: sans-serif;

}

@media (max-width: 600px) {

.card {

background-color: lightyellow;

}

}

</style>

</head>

<body>

<div class="card">

<h2 id="name">Jane Doe</h2>

<p>Web Developer</p>

<button onclick="changeName()">Change Name</button>

</div>

<script>

function changeName() {

document.getElementById("name").innerText = "Full Stack Developer";

}

</script>

</body>

</html>

**Module 3: React.js (Frontend Framework)**

**🔹 1. Introduction to React, JSX, and Components**

**What is React?**

React is a **JavaScript library** created by Facebook for building **user interfaces**. It is:

* **Component-based**: You build UI as small, reusable pieces called components.
* **Declarative**: You describe what you want to show, and React updates the UI automatically when data changes.
* **SPA-friendly**: Perfect for building Single Page Applications (SPAs) that load dynamically without full page reloads.

**What is JSX?**

JSX stands for **JavaScript XML**. It's a syntax extension that lets you write HTML-like code in your JavaScript files.

**Example:**

const element = <h1>Hello, world!</h1>;

JSX gets **compiled into JavaScript**, like:

React.createElement("h1", null, "Hello, world!");

**Components:**

A **component** is a function or class that returns JSX.

**Functional Component Example:**

function Hello() {

return <h1>Hello, React!</h1>;

}

You can reuse components:

<Hello />

<Hello />

**🔹 2. Props and State, Functional vs Class Components**

**Props (Properties):**

* Used to **send data from parent to child** components.
* Read-only, can’t be changed inside the child.

**Example:**

function Welcome(props) {

return <h2>Hello, {props.name}!</h2>;

}

// Usage:

<Welcome name="John" />

**State:**

* Used to **store and update data** that changes over time.
* Only available in components that manage it (usually functional components with hooks or class components).

**Example:**

const [count, setCount] = useState(0);

**Functional vs Class Components:**

**Functional Component (modern approach):**

function Greet() {

return <h1>Hi!</h1>;

}

**Class Component (older):**

class Greet extends React.Component {

render() {

return <h1>Hi!</h1>;

}

}

**🔹 3. useState and useEffect Hooks, Forms & Event Handling**

**useState:**

* A React Hook for managing local component state.

**Example:**

const [message, setMessage] = useState("Hello");

<button onClick={() => setMessage("Hi!")}>Click Me</button>

**useEffect:**

* A Hook that runs code **after rendering**.
* Great for **fetching data**, **updating the DOM**, or **setting timers**.

**Example:**

useEffect(() => {

console.log("Component mounted!");

}, []); // Empty array = run once

**Forms & Event Handling:**

React handles form events just like vanilla JavaScript, but you use JSX syntax.

**Example:**

function MyForm() {

const [name, setName] = useState("");

const handleSubmit = (e) => {

e.preventDefault();

alert(`Hello, ${name}`);

};

return (

<form onSubmit={handleSubmit}>

<input

type="text"

value={name}

onChange={(e) => setName(e.target.value)}

/>

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

</form>

);

}

**🔹 4. React Router Basics, Calling APIs, Mini Project**

**React Router Basics:**

Used to create **multiple pages (routes)** in a single-page React app.

**Installation:**

npm install react-router-dom

**Example Usage:**

import { BrowserRouter, Routes, Route } from 'react-router-dom';

function App() {

return (

<BrowserRouter>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/about" element={<About />} />

</Routes>

</BrowserRouter>

);

}

Now you can navigate using <Link>:

<Link to="/about">About</Link>

**Calling APIs:**

Use fetch() inside useEffect to get data from a server.

**Example:**

useEffect(() => {

fetch('https://jsonplaceholder.typicode.com/posts')

.then(res => res.json())

.then(data => setPosts(data));

}, []);

This is how your app can **show live data** from a server.

**Mini Project: React Contact List**

Let’s make a small project combining everything!

function ContactCard({ name, email }) {

return (

<div style={{ border: "1px solid gray", padding: "10px", margin: "10px" }}>

<h3>{name}</h3>

<p>{email}</p>

</div>

);

}

function App() {

const [contacts, setContacts] = useState([]);

useEffect(() => {

fetch('https://jsonplaceholder.typicode.com/users')

.then(res => res.json())

.then(data => setContacts(data));

}, []);

return (

<div>

<h1>Contact List</h1>

{contacts.map(c => (

<ContactCard key={c.id} name={c.name} email={c.email} />

))}

</div>

);

}

**2. Setting Up Node.js and npm**

**Step-by-Step Setup:**

1. **Download and install Node.js**  
   👉 Go to <https://nodejs.org> and install the LTS version.
2. **Check installation:**

\node -v # Check Node.js version

npm -v # Check npm version (comes with Node.js)

1. **Initialize a project folder:**

mkdir myapp

cd myapp

npm init -y # Creates a package.json file

1. **Install packages (example: Express):**

npm install express

npm = **Node Package Manager**, used to install JavaScript libraries and frameworks.

**🔹 3. Introduction to Backend Programming**

**What is Backend?**

* The **backend** is the **brain of a web app** — it handles:
  + Storing data (e.g., in databases)
  + Authentication (login/register)
  + Sending/receiving data from frontend
  + Business logic (calculations, conditions, etc.)

**Backend vs Frontend:**

| **Frontend (Client)** | **Backend (Server)** |
| --- | --- |
| HTML/CSS/JS | Node.js/Express.js |
| Runs in browser | Runs on server |
| Shows UI to user | Handles logic/data |

**Backend Flow Example:**

* User clicks "Submit"
* Frontend sends data to backend (Node.js)
* Backend stores it or returns a response
* Frontend shows success/failure message

**🔹 4. Building a Basic Server**

**Using Node’s built-in http module:**

Create a file server.js:

const http = require('http');

const server = http.createServer((req, res) => {

res.writeHead(200, {'Content-Type': 'text/plain'});

res.end('Hello from basic Node.js server!');

});

server.listen(3000, () => {

console.log('Server running on http://localhost:3000');

});

Run it:

node server.js

Visit http://localhost:3000 in your browser.

**🔹 5. Using Express.js (Simple Web Framework)**

**What is Express.js?**

Express.js is a minimal and flexible web application framework for Node.js. It is used to build web servers and APIs quickly and efficiently.

It simplifies common tasks like:

* Routing (handling URLs and requests)
* Middleware (handling requests/responses)
* Managing sessions, cookies, and forms
* A lightweight framework for Node.js
* Makes building web apps and APIs **much easier**
* Handles **routing**, **request/response**, **middleware**, and more

**Step-by-Step Setup:**

1. Install Express:

npm install express

1. Create a file app.js:

const express = require('express');

const app = express();

// Home route

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

res.send('Welcome to Express.js Server!');

});

// About route

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

res.send('This is the About page.');

});

// API route

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

res.json({ name: 'John', age: 30 });

});

// Start server

app.listen(3000, () => {

console.log('Express server running at http://localhost:3000');

});

Run it:

node app.js

Now you can visit:

* http://localhost:3000/ ➝ Shows welcome message
* http://localhost:3000/about ➝ Shows about info
* http://localhost:3000/api/user ➝ Returns JSON user data

**Bonus: Handling POST data with Express**

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

app.post('/api/contact', (req, res) => {

console.log(req.body);

res.send('Contact form submitted!');

});

Use tools like **Postman** or frontend forms to send POST requests to test this route.

**Summary:**

| **Topic** | **Summary** |
| --- | --- |
| Node.js | JavaScript runtime for backend development |
| npm | Package manager to install libraries |
| Backend Programming | Logic, data, APIs, database, user authentication |
| Basic Server (http module) | Create a simple server manually using Node.js |
| Express.js | Easy way to build web servers and APIs with routing |

**Module 4: Node.js (Backend Basics)**

**🔹 What is Node.js?**

**Node.js** is a **JavaScript runtime** built on **Chrome’s V8 engine** that lets you run JavaScript code **outside the browser**, typically on a server.

**Key Features:**

* Built on **non-blocking**, **event-driven** architecture.
* Ideal for building **fast**, **scalable** server-side applications.
* Enables **full-stack JavaScript** development (same language on client and server).

**Real-world Uses:**

* APIs and backends
* Real-time chat apps
* Streaming services

**🔹 Setting up Node.js and npm**

**Steps:**

1. Download Node.js from: <https://nodejs.org>
2. Install it (npm comes bundled with Node).
3. Check installation:

node -v # Shows Node.js version

npm -v # Shows npm version

**Initialize a project:**

mkdir myapp

cd myapp

npm init -y # Creates package.json

**🔹 Introduction to Backend Programming**

**Backend programming** refers to the server-side part of a web app:

* Handles logic, databases, APIs, user authentication, etc.
* Unlike frontend, it does not directly interact with users visually.

With Node.js + Express, you can:

* Create routes like /home, /users, /login
* Connect to databases
* Handle form submissions or API requests

**🔹 Building a Basic Server Using Express.js**

**Step 1: Install Express**

npm install express

**Folder structure:**

myapp/

├── node\_modules/

├── package.json

└── server.js

**server.js (Basic Server Code):**

// Import express

const express = require('express');

// Create an Express application

const app = express();

// Define a simple route

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

res.send('Welcome to My First Node.js Server using Express!');

});

// Start the server on port 3000

app.listen(3000, () => {

console.log('Server is running at http://localhost:3000');

});

**To run the server:**

node server.js

Then open your browser and visit: [**http://localhost:3000**](http://localhost:3000)

You’ll see: **"Welcome to My First Node.js Server using Express!"**

**Summary of Module 4:**

| **Topic** | **Description** |
| --- | --- |
| **Node.js** | Run JavaScript on the server |
| **npm** | Node package manager for installing libraries |
| **Backend** | Handles server logic, data, APIs |
| **Express.js** | Simplifies server creation and routing |

**Module 5: Express Middleware, Routing, and MongoDB Integration**

**What is Middleware in Express.js?**

**Middleware** functions are the **heart of Express.js**. They are functions that execute **during the request-response cycle**.

**Key Uses:**

* Logging
* Authentication
* Error handling
* Parsing request bodies

**Example:**

const express = require('express');

const app = express();

// Simple middleware to log every request

app.use((req, res, next) => {

console.log(`${req.method} ${req.url}`);

next(); // Pass control to the next middleware/route

});

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

res.send('Middleware Example');

});

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

**Routing in Express.js**

**Routing** determines how your app responds to different HTTP requests (GET, POST, PUT, DELETE).

**Example:**

const express = require('express');

const app = express();

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

res.send('Welcome Home!');

});

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

res.send('Form Submitted!');

});

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

res.send('Update successful!');

});

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

res.send('Delete completed!');

});

**Route Parameters Example:**

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

res.send(`User ID is: ${req.params.id}`);

});

**Connecting to MongoDB with Mongoose**

**MongoDB** is a NoSQL database, and **Mongoose** is an ODM (Object Data Modeling) library that makes it easier to use MongoDB with Node.js.

**Install Mongoose:**

npm install mongoose

**Connect to MongoDB:**

const mongoose = require('mongoose');

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

useNewUrlParser: true,

useUnifiedTopology: true,

}).then(() => {

console.log('MongoDB connected');

}).catch(err => {

console.error('Connection error', err);

});

**Creating a Simple CRUD API with MongoDB**

**Step-by-Step:**

**1. Create a Mongoose Schema:**

const mongoose = require('mongoose');

const UserSchema = new mongoose.Schema({

name: String,

email: String,

});

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

**2. Define Express Routes:**

const express = require('express');

const app = express();

app.use(express.json()); // For parsing JSON bodies

// Create a new user

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

const user = new User(req.body);

await user.save();

res.send(user);

});

// Read all users

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

const users = await User.find();

res.send(users);

});

// Update a user

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

const user = await User.findByIdAndUpdate(req.params.id, req.body, { new: true });

res.send(user);

});

// Delete a user

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

await User.findByIdAndDelete(req.params.id);

res.send({ message: 'User deleted' });

});

**Summary of Module 5:**

| **Concept** | **Description** |
| --- | --- |
| **Middleware** | Code that runs between request and response |
| **Routing** | Handles different HTTP methods and URLs |
| **MongoDB** | Database to store application data |
| **Mongoose** | Makes MongoDB easier to use in Node.js |
| **CRUD API** | Create, Read, Update, Delete operations on data |
|  |  |

**Mini Project: Simple User Management API**

**Project Goal:**

Build a backend API to **Create**, **Read**, **Update**, and **Delete** user profiles, using:

* **Express.js** for routing
* **Middleware** for logging
* **MongoDB + Mongoose** for data storage

**Project Structure:**

user-api/

├── models/

│ └── User.js

├── routes/

│ └── users.js

├── server.js

└── package.json

**Features to Implement:**

**1. Setup**

* Initialize Node project: npm init -y
* Install packages: npm install express mongoose

**2. User Schema (models/User.js)**

const mongoose = require('mongoose');

const UserSchema = new mongoose.Schema({

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

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

});

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

**3. User Routes (routes/users.js)**

const express = require('express');

const router = express.Router();

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

// Create user

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

const user = new User(req.body);

await user.save();

res.send(user);

});

// Get all users

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

const users = await User.find();

res.send(users);

});

// Update user

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

const user = await User.findByIdAndUpdate(req.params.id, req.body, { new: true });

res.send(user);

});

// Delete user

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

await User.findByIdAndDelete(req.params.id);

res.send({ message: 'User deleted' });

});

module.exports = router;

**4. Server File (server.js)**

const express = require('express');

const mongoose = require('mongoose');

const app = express();

const userRoutes = require('./routes/users');

// Middleware

app.use(express.json());

app.use((req, res, next) => {

console.log(`${req.method} ${req.url}`);

next();

});

// Connect to DB

mongoose.connect('mongodb://localhost:27017/userdb')

.then(() => console.log('MongoDB connected'))

.catch(err => console.error(err));

// Use routes

app.use('/users', userRoutes);

// Start server

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

**Optional Enhancements:**

* Add **validation** for fields
* Add **error handling middleware**
* Add **search** feature by name or email
* Add **pagination** for large user lists

**Learning Outcomes:**

* Apply Express routing and middleware
* Connect Node.js to MongoDB using Mongoose
* Practice building and testing RESTful APIs
* Understand basic project structure and modular code

**Module 6: MongoDB (Database Layer)**

This module focuses on MongoDB, a popular NoSQL database used with Node.js in the MERN stack. It introduces core concepts, how to use MongoDB with Mongoose (an ODM library), and best practices for schema design and data modeling.

**NoSQL and MongoDB Basics**

**Concept:**  
Unlike traditional relational databases, **NoSQL** databases like MongoDB store data in **flexible, JSON-like documents** instead of rows and tables.

**Key Features:**

* Schema-less structure
* High scalability
* Stores data in **collections** and **documents**

**Example:**  
A user document in MongoDB might look like this:

{

"\_id": "123abc",

"name": "John Doe",

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

"isAdmin": false

}

**Collections, Documents, CRUD Operations**

**Concepts:**

* **Collection**: A group of MongoDB documents (similar to a table)
* **Document**: A record in BSON (Binary JSON) format
* **CRUD**: Create, Read, Update, Delete operations

**Example (Using MongoDB Shell or Driver):**

// Create

db.users.insertOne({ name: "Alice", email: "alice@example.com" });

// Read

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

// Update

db.users.updateOne({ name: "Alice" }, { $set: { email: "alice@newmail.com" } });

// Delete

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

**Using MongoDB with Mongoose**

**Concept:**  
Mongoose is an **Object Data Modeling (ODM)** library for MongoDB and Node.js. It provides a schema-based solution to model your data.

**Example:**

const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/myapp');

const userSchema = new mongoose.Schema({

name: String,

email: String,

isAdmin: Boolean

});

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

// Create a new user

const newUser = new User({ name: "Bob", email: "bob@example.com", isAdmin: false });

newUser.save();

**Data Modeling and Schema Design**

**Concept:**  
Designing schemas helps enforce data structure and relationships. Mongoose lets you define data types, defaults, validation, and even relationships (refs).

**Example: One-to-Many Relationship (User & Posts):**

// Post Schema

const postSchema = new mongoose.Schema({

title: String,

content: String,

author: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }

});

// User Schema

const userSchema = new mongoose.Schema({

name: String,

email: String

});

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

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

// Create a user and a post

const user = new User({ name: "Sarah", email: "sarah@example.com" });

await user.save();

const post = new Post({ title: "My First Post", content: "Hello World", author: user.\_id });

await post.save();

**Learning Outcomes:**

By the end of this module, students will:

* Understand MongoDB’s NoSQL nature
* Perform CRUD operations directly and via Mongoose
* Design clean, scalable data models
* Implement relationships between collections

**Module 7: Integration – MERN Stack**

This module brings everything together—MongoDB, Express, React, and Node—into a working full-stack application with state management using Redux.

**1. Connecting MongoDB with Node and Express**

**Concept:**  
The backend connects to MongoDB using **Mongoose**. Express routes are used to create RESTful APIs to interact with the database.

**Example:**

// server.js (Node + Express + MongoDB)

const express = require('express');

const mongoose = require('mongoose');

const app = express();

mongoose.connect('mongodb://localhost:27017/myapp');

app.use(express.json());

const userSchema = new mongoose.Schema({ name: String });

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

// Create API route

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

const user = new User({ name: req.body.name });

await user.save();

res.send(user);

});

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

**2. Consuming Express APIs in React**

**Concept:**  
React (frontend) uses fetch or axios to make API calls to the Express server and update the UI.

**Example:**

// React Component (e.g., App.js)

import React, { useEffect, useState } from 'react';

import axios from 'axios';

function App() {

const [users, setUsers] = useState([]);

useEffect(() => {

axios.get('/api/users').then(res => setUsers(res.data));

}, []);

return (

<div>

<h1>Users:</h1>

<ul>{users.map(u => <li key={u.\_id}>{u.name}</li>)}</ul>

</div>

);

}

**3. State Management using Redux**

**Concept:**  
Redux manages global state across the application and handles complex UI states like loading, error, or updating lists.

**Redux Setup Example:**

1. **actions/userActions.js**

import axios from 'axios';

export const getUsers = () => async (dispatch) => {

dispatch({ type: 'USER\_LIST\_REQUEST' });

const { data } = await axios.get('/api/users');

dispatch({ type: 'USER\_LIST\_SUCCESS', payload: data });

};

1. **reducers/userReducer.js**

export const userListReducer = (state = { users: [] }, action) => {

switch (action.type) {

case 'USER\_LIST\_REQUEST':

return { loading: true };

case 'USER\_LIST\_SUCCESS':

return { loading: false, users: action.payload };

default:

return state;

}

};

1. **store.js**

import { configureStore } from '@reduxjs/toolkit';

import thunk from 'redux-thunk';

import { userListReducer } from './reducers/userReducer';

const store = configureStore({

reducer: { userList: userListReducer },

middleware: [thunk],

});

export default store;

1. **React Component (UserList.js)**

import React, { useEffect } from 'react';

import { useDispatch, useSelector } from 'react-redux';

import { getUsers } from './actions/userActions';

function UserList() {

const dispatch = useDispatch();

const { loading, users } = useSelector(state => state.userList);

useEffect(() => {

dispatch(getUsers());

}, [dispatch]);

return (

<div>

{loading ? "Loading..." : users.map(user => <p key={user.\_id}>{user.name}</p>)}

</div>

);

}

**4. Full-Stack CRUD Application (Mini Project)**

**Concept:**  
Build a small **User Manager App** with:

* Express API for backend
* MongoDB for data
* React + Redux frontend

**Functionality:**

* Add new users
* Display user list
* Edit or delete users

**Project Structure:**

/backend

└── models/User.js

└── routes/userRoutes.js

└── server.js

/frontend

└── components/UserList.js

└── redux/actions/userActions.js

└── redux/reducers/userReducer.js

└── App.js

**Learning Outcomes:**

By the end of this module, students will:

* Connect backend (Node/Express) to a MongoDB database
* Build and consume APIs in React
* Implement Redux for frontend state management
* Complete a working full-stack CRUD application

**Module 8: Deployment & Best Practices**

**1. Introduction to Git & GitHub**

**Concept:**  
Git is a version control system; GitHub is a platform to host code repositories, track changes, and collaborate.

**Commands Example:**

git init

git add .

git commit -m "Initial commit"

git remote add origin https://github.com/yourusername/blog-app.git

git push -u origin main

**Best Practice:**

* Commit often with meaningful messages
* Use .gitignore to avoid pushing node\_modules or .env files

**2. Hosting Frontend (Vercel/Netlify), Backend (Render/Heroku)**

**Frontend Deployment (Vercel/Netlify):**

1. Push React app to GitHub.
2. Import repo into Vercel or Netlify.
3. Configure build settings:
   * **Build Command**: npm run build
   * **Output Directory**: build/
4. Done!

**Backend Deployment (Render/Heroku):**

1. Create a Render or Heroku account.
2. Connect GitHub repo.
3. Set:
   * **Start Command**: node server.js
   * **Environment Variables**: MongoDB URI, PORT, etc.
4. Enable auto-deploy on push.

**3. Security Best Practices**

**Essentials:**

* Never expose credentials in code
* Use environment variables (dotenv)
* Validate incoming requests
* Sanitize inputs (prevent XSS, injection)
* Use helmet and CORS

**Example:**

const helmet = require('helmet');

const cors = require('cors');

app.use(helmet());

app.use(cors());

**Module 9: Capstone Project – Blog App**

Build and deploy a **Full Stack Blog Application** using MERN. This project will integrate everything learned.

**1. Idea Selection, Planning, and Wireframing**

**Goal:** A blogging platform where users can:

* Register/Login
* Create, Edit, Delete blog posts
* View blogs from others

**Tools:**

* Wireframing: Figma / Pen & paper
* Pages: Home, Post Detail, Create Post, Edit Post, Login, Register

**2. End-to-End Development and Deployment**

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

**Models:**

// models/User.js

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

username: String,

email: String,

password: String

});

// models/Post.js

const postSchema = new mongoose.Schema({

title: String,

content: String,

author: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }

});

**Routes:**

// routes/posts.js

app.post('/api/posts', authMiddleware, async (req, res) => {

const post = new Post({ title: req.body.title, content: req.body.content, author: req.user.\_id });

await post.save();

res.json(post);

});

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

const posts = await Post.find().populate('author', 'username');

res.json(posts);

});

**Frontend: React + Redux + Axios**

**Redux Action:**

// actions/postActions.js

export const getPosts = () => async (dispatch) => {

const { data } = await axios.get('/api/posts');

dispatch({ type: 'POST\_LIST\_SUCCESS', payload: data });

};

**Component:**

// PostList.js

function PostList() {

const dispatch = useDispatch();

const { posts } = useSelector(state => state.postList);

useEffect(() => {

dispatch(getPosts());

}, []);

return posts.map(post => (

<div key={post.\_id}>

<h2>{post.title}</h2>

<p>{post.content}</p>

<small>by {post.author.username}</small>

</div>

));

}

**Deployment Steps:**

* Push both frontend and backend to GitHub
* Deploy backend (Render) and set MONGO\_URI, JWT\_SECRET
* Deploy frontend (Vercel), set backend API URL in .env

**3. Final Presentation**

**Each student presents:**

* Live demo (frontend & backend)
* GitHub links
* Tools used (e.g., Figma, Vercel)
* Features implemented
* Challenges & solutions

**Bonus Ideas:**

* Add likes/comments
* Rich text editor (e.g., Quill.js)
* Search functionality

**Learning Outcomes:**

By the end of Module 9, students will:

* Understand real-world app development
* Practice best practices in deployment
* Gain hands-on experience with GitHub and hosting platforms
* Build confidence presenting and defending their work