Course Outline

Course Title: Back-End Web Development

Duration: 6 Months

Tech Stack: HTML, CSS, JavaScript, Node.js, Express.js, MongoDB, Mongoose, Payment Integration (Stripe, Paystack), Passport.js, Git/GitHub, and More

---

This heading aligns with the structure of the front-end course and clearly highlights the key technologies covered.

Month 1: Web Development Fundamentals

Week 1: Introduction to Web Development

Objective: Introduce students to the overall landscape of web development, explain the roles of front-end and back-end, and familiarize them with essential tools like the command line, Git, and GitHub.

1. **Overview of Front-end and Back-end Roles:**
   - **Front-end Development:** The part of the website users interact with directly, like buttons, forms, and navigation. Front-end developers focus on technologies such as HTML, CSS, and JavaScript to create a user-friendly experience.
   - **Back-end Development:** Handles server-side logic, databases, and server-to-client communication. Back-end developers work with server-side languages (Node.js in this case), databases (like MongoDB), and APIs.
   - **Full-stack Development:** Explanation of how full-stack developers handle both front-end and back-end responsibilities.
2. **Introduction to the Command Line (CLI):**
   - Importance of the CLI in development (for running scripts, installing packages, etc.)
   - Basic commands: file navigation (`cd`, `ls`/`dir`), file management (`touch`, `mkdir`, `rm`), running programs, installing tools.
   - Installing Node.js and verifying the installation with `node -v` and `npm -v`.
3. **Version Control with Git and GitHub:**
   - **Git Basics:** Introduction to version control systems, how Git helps manage project versions, track changes, and collaborate with other developers.
   - **Core Concepts:**
     - `git init`: Initialize a Git repository.
     - `git add`, `git commit`: Staging and committing code.
     - `git log`: Viewing commit history.

- **Branching and Merging:**
    - Creating and managing branches (`git branch`, `git checkout`).
    - Merging branches back into the main branch.
- **GitHub:**
    - Pushing code to a GitHub repository.
    - Creating pull requests to collaborate with others.
    - Basic GitHub flow: Forking, cloning, committing, pushing, and merging.

4. **Basics of Hosting Platforms (Netlify, Vercel):**

    - Explanation of how hosting platforms work and their role in deploying static and dynamic websites.
    - Differences between static site hosting (Netlify) and serverless functions (Vercel).
    - Deploying a simple static website to Netlify or Vercel to give students a sense of how hosting works.

Week 2: HTML & CSS Essentials

Objective: Lay a solid foundation in HTML and CSS so students can build simple, responsive web pages.

1. **HTML Structure:**

    - Basics of HTML: Tags, elements, and attributes.
    - Structuring web pages with `<!DOCTYPE html>`, `<html>`, `<head>`, `<body>`, and metadata (`<meta>`).
    - **Key HTML Elements:**
        - Headings, paragraphs, lists, links, and images.
        - Forms: `<input>`, `<label>`, `<textarea>`, `<button>`.
    - **Semantic HTML:**
        - Using semantic tags (`<header>`, `<footer>`, `<section>`, `<article>`, `<nav>`) to improve accessibility and SEO.

2. **CSS Basics:**

    - **Box Model:** Understanding padding, margin, border, and content box.
    - **Layout Techniques:**
        - **Flexbox:** For aligning elements along the main and cross axes (`display: flex`, `justify-content`, `align-items`).
        - **CSS Grid:** For creating two-dimensional layouts (`grid-template-columns`, `grid-template-rows`).
    - **Styling Essentials:**
        - CSS selectors, properties, and values.
        - Colors, fonts, spacing, borders, and backgrounds.
        - Inline vs. internal vs. external styles.

3. **Building Static Web Pages:**

    - Creating simple, multi-section static web pages.

- Hands-on exercise: Build a portfolio website or landing page using HTML and CSS.

4. **Introduction to Responsive Design:**

   - **Media Queries:** Making websites adapt to different screen sizes.
   - Introduction to mobile-first design.
   - Using percentage-based widths and flexible units like `em`, `rem`, and `vw/vh`.

Week 3: Introduction to JavaScript

Objective: Introduce students to programming with JavaScript, covering fundamental concepts and how JavaScript interacts with HTML and the browser.

1. **JavaScript Basics:**

   - **Variables:** `var`, `let`, and `const`, and when to use each.
   - **Data Types:** Strings, numbers, booleans, arrays, and objects.
   - **Operators:** Arithmetic (`+`, `-`, `*`, `/`), comparison (`==`, `===`, `!=`), logical (`&&`, `||`).

2. **Functions, Loops, and Conditionals:**

   - **Functions:** Creating and invoking functions, passing parameters, and returning values.
   - **Conditionals:** `if`, `else if`, `else`, ternary operator.
   - **Loops:** `for`, `while`, and `do-while` loops for iteration.

3. **DOM Manipulation:**

   - **Selecting Elements:** `document.getElementById()`, `document.querySelector()`.
   - **Changing Content:** `textContent`, `innerHTML`, and manipulating attributes.
   - **Creating and Removing Elements:** `createElement()`, `appendChild()`, `removeChild()`.

4. **Basic Events and Event Listeners:**

   - **Event Handling:** `addEventListener()` for listening to user actions like clicks, form submissions, etc.
   - Example exercise: Building a basic interactive to-do list.

Week 4: JavaScript ES6+

Objective: Introduce modern JavaScript features (ES6+) to ensure students can write clean, maintainable code.

1. **ES6+ Features:**

   - **Arrow Functions:** Shorter syntax and the differences in the `this` keyword.
   - **Template Literals:** Using backticks for string interpolation.
   - **Destructuring:** Extracting data from arrays and objects.

2. **Promises and Async/Await:**

- **Promises:** Understanding how to handle asynchronous code, chaining `.then()` and `.catch()`.
- **Async/Await:** Simplified syntax for handling promises.
- Hands-on: Fetching data from an API using `fetch()` and displaying it on a web page.

3. **Object-Oriented Programming Basics:**

   - **Classes:** Defining classes, constructors, and methods.
   - **Inheritance:** Extending classes and overriding methods.

4. **Error Handling and Debugging:**

   - **Error Handling:** Using `try`, `catch`, and `finally` to handle errors.
   - **Debugging Tools:** Using `console.log()` effectively, Chrome DevTools, and breakpoints.

Month 2: Node.js Fundamentals

Week 5: Introduction to Node.js

Objective: Teach students the basics of Node.js, covering its history, environment setup, and event-driven architecture.

1. **What is Node.js?**

   - History and why Node.js is popular for building server-side applications.
   - Difference between traditional server-side languages (e.g., PHP) and Node.js.

2. **Setting Up a Node.js Environment:**

   - Installing Node.js and verifying the installation.
   - Running basic Node.js scripts using `node`.

3. **Understanding Node.js Event-Driven Architecture:**

   - Explanation of how Node.js is non-blocking and uses an event-driven model.
   - Handling asynchronous events using callback functions.

4. **Core Node.js Modules:**

   - Working with built-in modules like `fs` (File System), `http`, and `path`.
   - Building a simple HTTP server that responds with a web page.

Week 6: NPM & Project Structure

Objective: Introduce students to NPM for package management and guide them in structuring Node.js projects.

1. **Introduction to NPM:**

   - Installing Node packages (`npm install`).

- Managing dependencies with `package.json`.
- Global vs. local installations.

2. **Installing and Managing Packages:**
   - Installing essential packages for Node.js projects (e.g., `nodemon` for auto-restarts).
   - Versioning and updating packages (`npm update`).
3. **Project Structure and Modular Code:**
   - Best practices for organizing a Node.js project: separating routes, controllers, and models.
   - Creating and using modules with `module.exports` and `require()`.
4. **Environment Variables:**
   - Storing sensitive data (like API keys) using `dotenv`.
   - Setting up `.env` files and accessing environment variables in your code.

Week 7: Working with Node.js and Asynchronous Code

Objective: Dive deeper into asynchronous programming in Node.js, focusing on event-driven behavior and working with files.

1. **Understanding the Event Loop:**
   - Explanation of how the event loop works in Node.js.
   - Understanding blocking vs. non-blocking code.
2. **Handling Asynchronous Operations:**
   - Callbacks vs. promises.
   - Using `async` and `await` for cleaner asynchronous code.
3. **Working with Files:**
   - Reading from and writing to files using the `fs` module.
   - Asynchronous vs. synchronous file operations.
4. **Building a Command-Line App:**
   - Hands-on: Create a simple command-line app (e.g., a note-taking app) that uses file operations.

Week 8 Introduction to Express.js

Objective: Introduce students to Express.js, the most popular web framework for Node.js, and teach them how to build web applications with it.

1. **What is Express.js?**

- Overview of Express.js and its role in simplifying web application development.
- Setting up an Express project from scratch.

2. **Middleware:**

   - What middleware is and how it works in Express.js.
   - Creating and using custom middleware for logging, parsing requests, and more.

3. **Routing in Express.js:**

   - Defining and handling routes (GET, POST, PUT, DELETE).
   - Building a basic CRUD application (e.g., a simple blog or task manager).

4. **Handling Errors and 404 Responses:**

   - Setting up middleware for error handling.
   - Creating custom 404 pages for undefined routes.

Month 3: Databases (MongoDB & Mongoose)

In Month 3, the focus shifts toward databases, specifically MongoDB, a NoSQL database, and Mongoose, an object data modeling (ODM) library for MongoDB in Node.js. Students will be guided through connecting databases to their Node.js applications, performing CRUD operations, and building efficient, scalable systems.

Week 9: Introduction to MongoDB

Objective: Introduce students to MongoDB and the NoSQL paradigm, enabling them to perform basic database operations.

1. **What is MongoDB?**

   - **MongoDB Overview:** A NoSQL database that stores data in flexible, JSON-like documents. Unlike SQL databases that use tables and rows, MongoDB uses collections and documents.
   - **NoSQL vs. SQL:**
     - SQL databases (like MySQL, PostgreSQL) store data in a structured format with fixed schemas (tables, rows, columns).
     - NoSQL databases (like MongoDB) are schema-less and allow more flexibility for handling unstructured or semi-structured data.
     - Key differences: scalability, schema flexibility, query language (SQL vs. MongoDB’s query language).

2. **Setting up MongoDB:**

   - **MongoDB Locally:** Install MongoDB on the local system using the MongoDB Community Server.
   - **MongoDB Atlas:** Introduce students to cloud-based databases with MongoDB Atlas, allowing them to create a free-tier MongoDB cluster for easy cloud database management.

   - **Connecting Node.js to MongoDB:** Use the official MongoDB Node.js driver to establish a connection between an application and the database.

3. **Understanding Collections and Documents:**
   - **Collections:** Analogous to SQL tables, collections store documents, where each document is a JSON-like object.
   - **Documents:** Each document in MongoDB is a key-value pair, allowing for nested fields and arrays.
   - **Schemas:** Explain how MongoDB collections don't enforce schemas, offering flexibility to store a variety of data.

4. **CRUD Operations with MongoDB:**
   - **Create:** Insert documents into collections using `insertOne()` and `insertMany()`.
   - **Read:** Fetch data using `find()` and query filters.
   - **Update:** Update data using `updateOne()` and `updateMany()` with various query operators (`$set`, `$inc`).
   - **Delete:** Remove documents using `deleteOne()` and `deleteMany()`.
   - Hands-on Exercise: Building a basic MongoDB-based app to perform CRUD operations.

Week 10: Mongoose for MongoDB

Objective: Introduce students to Mongoose as an ODM for MongoDB, which helps in structuring and validating data, simplifying the integration between Node.js and MongoDB.

1. **Introduction to Mongoose:**
   - **What is Mongoose?** Mongoose is a popular ODM that provides a schema-based solution for structuring and validating data in MongoDB.
   - **Why use Mongoose?** It simplifies working with MongoDB by providing helpful abstractions such as models, validation, middleware, and easy-to-use query methods.

2. **Defining and Using Models/Schemas:**
   - **Schemas:** Define a blueprint for your MongoDB documents, specifying the structure, field types, and validation rules for each document in a collection.
   - **Models:** Create models from schemas, which allow you to interact with the corresponding MongoDB collection.
   - Example: Define a simple user schema with fields like `name`, `email`, `password`, and `role`.

3. **Mongoose Queries:**
   - **find():** Querying the database for documents.
   - **save():** Saving new data to the database.
   - **update():** Updating existing documents.
   - **delete():** Removing documents.
   - Hands-on: Build a simple blog app where users can create, edit, delete, and view posts using Mongoose models.

4. **Data Validation with Mongoose:**
   - **Schema validation:** Use Mongoose's built-in validation features to ensure data integrity (e.g., required fields, field lengths, email validation).
   - **Custom validators:** Create custom validation functions for more complex data rules.

Week 11: Advanced MongoDB & Mongoose

Objective: Delve deeper into advanced MongoDB and Mongoose features, such as referencing documents, optimizing performance, and working with advanced queries.

1. **Population and Referencing Documents:**
   - **Document Relationships:** Explain how MongoDB handles relationships (one-to-one, one-to-many) using document references or embedded documents.
   - **Mongoose Population:** Use the `.populate()` method in Mongoose to fetch related documents (e.g., linking users to posts or comments).
2. **Virtual Fields and Aggregation:**
   - **Virtual Fields:** Fields that aren't stored in the database but are computed at runtime (e.g., full name derived from first and last name).
   - **Aggregation:** Perform advanced data processing using MongoDB's aggregation pipeline (`$match`, `$group`, `$sort`, `$project`).
   - Example: Use the aggregation framework to generate reports (e.g., calculate total sales, group by category).
3. **Indexes and Performance Optimization:**
   - **Indexes:** Explain how indexes can improve query performance and when to use them. Use the `.createIndex()` method in MongoDB to speed up searches.
   - **Optimization Strategies:** Techniques for improving query performance, such as avoiding large documents and using projection to limit fields returned.
4. **Working with Timestamps and Lifecycle Hooks:**
   - **Timestamps:** Automatically track `createdAt` and `updatedAt` timestamps for documents in MongoDB using Mongoose's built-in timestamp functionality.
   - **Lifecycle Hooks:** Use Mongoose pre/post hooks (`pre('save')`, `post('remove')`) to trigger actions before or after certain events.

Week 12: Building a REST API with Express & MongoDB

Objective: Introduce students to RESTful API development using Express and MongoDB, with Mongoose for data handling.

1. **RESTful Architecture Overview:**

- **REST Principles:** Explain the concepts behind REST (Representational State Transfer), emphasizing statelessness and the importance of HTTP methods (`GET`, `POST`, `PUT`, `DELETE`).
- **Resources and Endpoints:** How RESTful services expose resources through URLs (e.g., `/users`, `/posts`).

2. **Setting Up Routes for a REST API:**
   - **Express Router:** Use Express's `router` to handle API routes.
   - Build RESTful routes (`/users`, `/posts`) to perform CRUD operations via HTTP methods.

3. **Handling Data with Mongoose in an API:**
   - **Data flow:** Use Mongoose models to handle database operations in your API endpoints.
   - Example: Build routes that allow users to create, read, update, and delete resources in a MongoDB collection.

4. **Error Handling in API Requests:**
   - Use Express error-handling middleware to manage errors gracefully (e.g., handling missing resources, invalid input).
   - Return appropriate HTTP status codes (`200`, `400`, `404`, `500`) and descriptive error messages.

5. **API Documentation with Postman:**
   - **Postman:** Introduce Postman as a tool for testing APIs and documenting requests and responses.
   - Hands-on Exercise: Test the CRUD operations of the API using Postman and create a simple API documentation.

Month 4: Authentication & Authorization

Month 4 is dedicated to securing applications, covering key topics like user authentication with Passport.js and JSON Web Tokens (JWT), OAuth, and encryption techniques.

Week 13: User Authentication with Passport.js

Objective: Introduce students to authentication and authorization concepts, focusing on using Passport.js for local authentication.

1. **Overview of Authentication and Authorization:**
   - **Authentication:** The process of verifying the identity of a user (e.g., login systems).
   - **Authorization:** The process of determining whether a user has access to specific resources or actions (e.g., role-based permissions).

2. **Setting up Passport.js for Local Authentication:**
   - **Local Strategy:** Use Passport's local authentication strategy to handle login with a username and password.

- **Session-based Authentication:** Store user authentication details in server sessions (using cookies and `express-session`).
- **User Registration and Login:** Create endpoints to register new users and authenticate them using Passport.

3. **Session-based vs. Token-based Authentication (JWT):**

    - **Session-based:** User authentication state is stored on the server (sessions).
    - **Token-based (JWT):** Authentication state is stored client-side, using JSON Web Tokens to validate requests.
    - Comparison: Discuss the pros and cons of each approach.

4. **Securing Routes and Middleware for Protected Endpoints:**

    - **Route Protection:** Use Passport to protect routes (e.g., requiring authentication to access certain pages or APIs).
    - **Custom Middleware:** Build custom middleware to check for authentication before allowing access to protected routes.

Week 14: OAuth with Passport.js

Objective: Teach students how to integrate OAuth for third-party login, such as Google and Facebook, and handle secure API access.

1. **Setting up Social Login (Google, Facebook):**

    - **OAuth 2.0:** Explain how OAuth works for authorizing access to third-party services.
    - **Passport OAuth Strategy:** Set up Passport's OAuth strategy for social logins (Google, Facebook).

2. **Securing API Endpoints with OAuth Tokens:**

    - Use OAuth tokens to authorize API requests, ensuring that users are authenticated via a third-party service before accessing your API.

3. **Role-based Access Control (RBAC):**

- **RBAC Concepts:** Introduce the concept of RBAC for controlling access based on user roles (e.g., admin, editor, viewer).
- **Implement RBAC:** Implement RBAC in Express.js routes, allowing certain actions (like delete or update) only for authorized roles.

Week 15: JSON Web Tokens (JWT) Authentication

Objective: Focus on using JWTs for authentication in stateless applications, providing a secure and scalable way to manage user sessions.

1. **What is JWT and How Does It Work?**

- **JWT Structure:** Explain the structure of a JWT (header, payload, signature) and how it works for stateless authentication.
- **Signing and Verifying:** Show how to sign tokens on the server and verify them on the client for secure communication.

2. **Implementing JWT Authentication in Express.js:**
    - Create an Express.js API with JWT-based authentication, allowing users to log in and receive a token.
3. **Securing Routes with JWT Middleware:**
    - Build JWT middleware to secure routes by verifying the presence of valid tokens in requests.
4. **Refresh Tokens and Token Expiry Strategies:**
    - **Token Expiry:** Explain how JWT tokens have expiry times and how to refresh tokens when they expire.
    - **Refresh Token Flow:** Implement refresh tokens for renewing access tokens securely.

Week 16: Securing Applications

Objective: Teach students essential security techniques for web applications, including encryption, data protection, and securing sensitive information.

1. **Encryption and Hashing (bcrypt.js):**
    - **Hashing Passwords:** Use `bcrypt.js` to hash user passwords before storing them in the database.
    - **Salting:** Add salt to hashed passwords to enhance security.
2. **Securing Sensitive Data:**
    - **Helmet:** Use `helmet` middleware to secure Express apps by setting various HTTP headers.
    - **Rate Limiting:** Implement rate limiting to prevent brute force attacks or DDoS attacks.
    - **CORS:** Configure Cross-Origin Resource Sharing (CORS) to control which domains can access your API.
3. **HTTPS Setup (SSL Certificates, Let's Encrypt):**
    - **SSL Certificates:** Set up HTTPS for your application using SSL certificates to encrypt data in transit.
    - **Let's Encrypt:** Use Let's Encrypt to get free SSL certificates for your website.
4. **Best Practices for Handling User Data and Security Audits:**
    - **Data Privacy:** Ensure compliance with privacy laws (like GDPR) and best practices for handling user data.
    - **Security Audits:** Teach students how to audit their applications for common vulnerabilities (e.g., using OWASP's top 10 security risks).

Month 5: Advanced Back-End Topics

This month delves into more advanced aspects of back-end development, focusing on Express.js, WebSockets for real-time communication, working with APIs (REST and GraphQL), testing, and debugging. By the end of the month, students will have a comprehensive understanding of how to build scalable and testable back-end systems, preparing them for real-world challenges.

Week 17: Advanced Express.js Topics

Objective: Enhance students' understanding of Express.js by introducing advanced topics such as reusable middleware, file management, and efficient routing.

1. **Building Reusable Middleware:**
    - **Middleware Basics:** Review the concept of middleware in Express.js and how it acts as a bridge between request and response.
    - **Reusable Middleware:** Learn how to create custom middleware functions that can be reused across routes, improving modularity and code organization. Example: Logging requests, rate limiting, authentication checks.
    - **Third-party Middleware:** Explore popular middleware packages such as `helmet` (security headers), `morgan` (logging), and `cors` (Cross-Origin Resource Sharing).
    - Hands-on: Create custom middleware for logging requests, checking user authentication, and handling errors globally.
2. **Express.js Error Handling Best Practices:**
    - **Error-handling Middleware:** Understand how to define error-handling middleware that captures errors and provides meaningful feedback to the client.
    - **Synchronous vs. Asynchronous Errors:** Learn how to handle errors in synchronous and asynchronous code (e.g., `try-catch` for async/await).
    - **Best Practices:** Tips for managing errors effectively (e.g., never exposing sensitive data in error messages, using HTTP status codes properly).
3. **Express Router for Modular Routes:**
    - **Modularizing Routes:** Organize large Express.js applications by splitting routes into separate modules using `express.Router()`.
    - **Route Grouping:** Implement route grouping for different entities (e.g., `/users`, `/posts`, `/products`), improving maintainability and scalability.
    - Hands-on: Refactor a large Express app by splitting routes into separate modules and applying middleware to specific groups of routes.
4. **Uploading and Managing Files with Multer.js:**
    - **File Uploads:** Learn how to handle file uploads in Express.js using the `Multer.js` middleware.

- **File Validation:** Ensure proper file types (e.g., images, PDFs) and sizes before storing them on the server.
- **File Management:** Explore strategies for managing uploaded files (e.g., saving files to local storage or cloud storage like AWS S3).
- Hands-on: Build a simple application that allows users to upload profile pictures, with proper validation and error handling.

Week 18: Building Real-Time Apps with WebSockets

Objective: Introduce students to real-time communication using WebSockets, allowing them to build applications that require instant updates such as chat systems and notifications.

1. **What are WebSockets?**
    - **Real-time Communication:** Explain WebSockets as a protocol for bidirectional, real-time communication between the server and client.
    - **HTTP vs. WebSockets:** Compare WebSockets to the traditional request-response model of HTTP, emphasizing the persistent connection of WebSockets.
    - **Use Cases:** Discuss scenarios where WebSockets are ideal (e.g., live chat, notifications, stock price updates).
2. **Setting up WebSockets with Socket.io in Node.js:**
    - **Socket.io:** Introduce Socket.io, a library that simplifies the process of using WebSockets in Node.js applications.
    - **Setting up a WebSocket Server:** Implement a basic WebSocket server using Socket.io.
    - **Real-time Communication:** Establish a persistent connection between the client and server to send and receive real-time messages.
    - Hands-on: Create a simple chat application that allows multiple users to send and receive messages in real-time.
3. **Implementing Basic Real-Time Features:**
    - **Chat Application:** Build a chat system where users can join rooms, send messages, and receive messages in real-time.
    - **Notifications:** Implement real-time notifications for events such as new messages, updates, or alerts.
    - **Broadcasting:** Use Socket.io's broadcasting feature to send messages to multiple clients simultaneously.
4. **Scaling WebSockets with Redis:**
    - **Redis Pub/Sub:** Introduce Redis for scaling WebSocket applications by using its Pub/Sub (publish/subscribe) system to manage events across multiple instances of a Node.js server.
    - **Scaling WebSocket Servers:** Discuss strategies for horizontally scaling WebSocket servers, ensuring that real-time communication works seamlessly across distributed systems.

Week 19: Working with APIs (REST vs GraphQL)

Objective: Teach students the differences between REST and GraphQL, guiding them through building and using both API types in their Node.js applications.

1. **REST vs. GraphQL:**
    - **REST Overview:** Review the RESTful architecture, focusing on how resources are represented and how HTTP methods are used for different operations (GET, POST, PUT, DELETE).
    - **GraphQL Overview:** Introduce GraphQL, a query language for APIs that allows clients to request only the data they need.
    - **Differences and Use Cases:**
        - REST: Best for simple, well-defined resources and stateless communication.
        - GraphQL: Ideal for complex, nested resources and situations where the client needs more control over the data it retrieves.
    - **When to Use Each:** Discuss the scenarios where REST is preferable versus those where GraphQL would be more efficient.
2. **Building a Basic GraphQL Server in Node.js:**
    - **GraphQL Setup:** Install and configure Apollo Server to build a GraphQL API in Node.js.
    - **Schema Definition:** Define GraphQL types, queries, and mutations.
    - Hands-on: Create a simple GraphQL API for managing a blog system, allowing users to query and manipulate posts and comments.
3. **Working with GraphQL Queries and Mutations:**
    - **Queries:** Learn how to query data with GraphQL, specifying the exact fields and related entities needed.
    - **Mutations:** Use mutations to perform create, update, and delete operations.
    - Hands-on: Build a GraphQL-based blog that allows querying posts with nested comments and adding new posts or comments.
4. **Connecting GraphQL to MongoDB Using Mongoose:**
    - **GraphQL + Mongoose:** Combine GraphQL with Mongoose models to handle database interactions.
    - **Resolvers:** Implement resolvers in GraphQL to handle fetching and updating data from MongoDB.
    - Hands-on: Extend the blog system by connecting it to MongoDB, allowing users to interact with a database via GraphQL.

Week 20: Testing & Debugging

Objective: Equip students with skills to write robust tests and debug Node.js applications, ensuring their projects are reliable and maintainable.

1. **Introduction to Unit Testing (Mocha, Chai):**

- **Unit Testing Basics:** Introduce the concept of unit testing—testing individual pieces of code in isolation (e.g., functions, classes).
- **Mocha & Chai:** Use the Mocha testing framework with Chai assertion library to write unit tests for Node.js applications.
- Hands-on: Write unit tests for simple functions like calculating sums, validating input, or checking authentication logic.

2. **Testing Asynchronous Code in Node.js:**

   - **Asynchronous Tests:** Learn how to write tests for asynchronous code (e.g., promises, callbacks) using Mocha's `done()` callback or async/await.
   - Hands-on: Test asynchronous functions that interact with a database or external API.

3. **Integration Testing for APIs (Supertest):**

   - **Integration Testing:** Understand the concept of integration testing—testing how different parts of the application work together.
   - **Supertest:** Use Supertest to test Express.js APIs by sending HTTP requests and validating the responses.
   - Hands-on: Write tests to ensure that an API's routes (e.g., `/users`, `/posts`) behave as expected when receiving requests.

4. **Debugging with Node.js Built-In Debugger:**

   - **Node.js Debugger:** Learn how to use Node.js's built-in debugging tools (`node inspect`) to step through code and find bugs.
   - **VS Code Debugging:** Set up a Node.js debugging environment in VS Code for a more visual debugging experience.
   - Hands-on: Debug a broken Node.js application, identifying and fixing bugs using the debugger.

Month 6: Payment Integration & Deployment

Month 6 focuses on practical aspects of back-end development: integrating payment systems, sending emails and notifications, and deploying Node.js applications to cloud platforms. It concludes with a capstone project that ties all the learned concepts together.

Week 21: Introduction to Payment Integration

Objective: Teach students how to integrate payment gateways like Stripe and Paystack into their applications, enabling them to process payments securely.

1. **Overview of Payment Systems and Flows:**

   - **Payment Gateway:** Introduce payment gateways (e.g., Stripe, Paystack) and how they process payments on behalf of businesses.
   - **Payment Flow:** Explain the typical flow of an online payment (e.g., client initiates payment, server verifies transaction, payment gateway processes the payment, server confirms

success/failure).

2. **Setting up Stripe API for Payment Processing:**
    - **Stripe Integration:** Walk students through setting up a Stripe account and using the Stripe API to create and manage payments.
    - **Handling Payments:** Implement payment processing, allowing users to make one-time payments.
    - Hands-on: Create a checkout page where users can purchase products or services using Stripe.
3. **Creating a Payment System with Paystack:**
    - **Paystack Overview:** Introduce Paystack as an alternative to Stripe, especially popular in African countries.
    - **API Integration:** Set up Paystack in a Node.js application, allowing users to process payments in local currencies.
    - Hands-on: Implement Paystack integration for handling payments, including verifying and confirming transactions.
4. **Handling Payments, Subscriptions, and Refunds:**
    - **Subscription Payments:** Learn how to set up recurring subscription payments using Stripe's or Paystack's subscription services.
    - **Refunds:** Implement logic for handling refunds through the payment gateway's API.
    - Hands-on: Extend the payment system to support recurring subscriptions and issue refunds when necessary.

Week 22: Email & Notifications

Objective: Teach students how to send transactional emails and notifications, both essential for many web applications (e.g., order confirmations, password resets).

1. **Sending Transactional Emails (Nodemailer):**
    - **Nodemailer:** Use Nodemailer to send transactional emails (e.g., user registration, password reset).
    - **SMTP Configuration:** Set up an SMTP server (or use a service like SendGrid) to send emails from the application.
    - Hands-on: Create a system where users receive an email upon registration or after placing an order.
2. **Setting Up Email Templates and SMTP Configurations:**
    - **Email Templates:** Create and use HTML-based email templates for sending professional, branded emails.
    - **SMTP Services:** Discuss using external SMTP services (e.g., Mailgun, SendGrid) for large-scale email sending.
    - Hands-on: Design a reusable email template and configure SMTP for sending it.

3. **Implementing Notification Systems (Email, SMS):**
    - **Email Notifications:** Set up a system to send email notifications to users based on specific triggers (e.g., order confirmation, new message).
    - **SMS Notifications:** Integrate SMS services (e.g., Twilio) to send SMS notifications for important updates.
    - Hands-on: Implement a notification system that sends email and SMS alerts based on user actions.
4. **Push Notifications (Using Web Push):**
    - **Web Push:** Introduce Web Push to send real-time push notifications to users' browsers (even when the website is not open).
    - Hands-on: Implement Web Push notifications for real-time updates (e.g., new messages or order status changes).

Week 23: Cloud Deployment & Scaling

Objective: Equip students with knowledge of deploying and scaling Node.js applications in the cloud, focusing on continuous integration/continuous deployment (CI/CD) pipelines and application monitoring.

1. **Deploying Applications to Cloud Platforms (Heroku, AWS, DigitalOcean):**
    - **Cloud Platforms Overview:** Introduce different cloud platforms (Heroku, AWS, DigitalOcean) and their strengths.
    - **Deploying to Heroku:** Walk students through deploying an Express.js app to Heroku, covering environment variables, app scaling, and custom domains.
    - Hands-on: Deploy a fully functioning Node.js application to Heroku.
2. **Understanding CI/CD Pipelines (GitHub Actions):**
    - **Continuous Integration/Continuous Deployment (CI/CD):** Explain the importance of automated testing and deployment in modern web development.
    - **GitHub Actions:** Use GitHub Actions to create a CI/CD pipeline that automatically tests and deploys the application when changes are pushed to the repository.
    - Hands-on: Set up a simple CI/CD pipeline with GitHub Actions for a Node.js app.
3. **Scaling Node.js Applications (Clustering, PM2):**
    - **Node.js Clustering:** Use Node.js's built-in clustering to take advantage of multiple CPU cores, improving application performance under heavy load.
    - **PM2:** Introduce PM2, a process manager for Node.js, and use it to manage and scale Node.js applications in production.
    - Hands-on: Set up PM2 to manage a deployed Node.js application, enabling automatic restarts, scaling, and monitoring.
4. **Monitoring Applications (Log Management, APM Tools):**

    - **Log Management:** Set up centralized logging systems (e.g., using Winston or Loggly) to track application errors and performance in real-time.
    - **Application Performance Monitoring (APM):** Introduce APM tools (e.g., New Relic, Datadog) to monitor application performance, server load, and response times.
    - Hands-on: Set up basic application monitoring and log management for a deployed app, enabling early detection of issues.

Week 24: Capstone Project

Objective: Allow students to integrate everything they have learned over the last six months into a full-fledged, real-world application, which they will present as their final project.

1. **Building a Full-Stack Application:**

    - **Project Overview:** Students will build a full-stack application using Node.js, Express.js, MongoDB, and all the technologies they've learned (authentication, payment integration, real-time features, etc.).
    - **Project Features:** The project should include user authentication (using JWT or Passport.js), payment processing (Stripe/Paystack), real-time features (WebSockets), and more.
    - Hands-on: Spend two weeks building, testing, and refining the project.

2. **Final Project Presentation and Review:**

    - **Project Presentation:** Students will present their capstone projects to the class, explaining the technologies used, challenges faced, and solutions implemented.
    - **Code Review:** Conduct a thorough review of each student's project, providing constructive feedback and suggestions for improvement.

3. **Next Steps: Career Advice, Portfolio Building, Job Interviews:**

    - **Portfolio Building:** Guide students on how to showcase their final projects on a portfolio website.
    - **Resume Preparation:** Provide tips on how to highlight technical skills on resumes.
    - **Job Interviews:** Conduct mock technical interviews, focusing on back-end concepts and problem-solving skills.

Optional Extras/Extensions:

These topics can be covered as optional sessions or self-paced learning resources for advanced students.

- **Advanced TypeScript in Node.js:** Use TypeScript to add static types to Node.js applications, improving code quality and maintainability.
- **Microservices Architecture with Node.js:** Explore microservices architecture and how to split a monolithic Node.js application into smaller, independent services.
- **Dockerizing Node.js Applications:** Learn how to containerize Node.js applications using Docker, making them easier to deploy and scale.

- **Real-Time Collaboration Features with WebSockets:** Build real-time collaboration tools (e.g., shared document editing or real-time drawing apps) using WebSockets.
- **Working with Redis for Caching and Scaling:** Introduce Redis as a caching layer to speed up database queries and scale Node.js applications.
- **GraphQL API Optimizations with Apollo Server:** Optimize GraphQL APIs using Apollo Server, including caching, batching, and schema stitching for microservices.
