Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

MODULE 3 - BACKEND DEVELOPMENT (NODE.JS & EXPRESS.JS)

SETUP NODE.JS AND EXPRESS.JS RUNTIME ENVIRONMENT & INSTALL NECESSARY SOFTWARE

CHAPTER 1: INTRODUCTION TO NODE.JS AND EXPRESS.JS

1.1 What is Node.js?

Node.js is an **open-source, server-side JavaScript runtime environment** built on **Google Chrome's V8 engine**. It allows developers to run JavaScript **outside the browser,** making it ideal for building **fast, scalable backend applications**.

1.2 What is Express.js?

Express.js is a **lightweight and flexible web framework for Node.js** that simplifies backend development by providing:

✅ **Routing mechanisms** – Manage HTTP requests efficiently.

✅ **Middleware support** – Process incoming requests before reaching the server logic.

✅ **Template engines** – Render dynamic web pages.

✅ **REST API development** – Create scalable APIs quickly.

1.3 Why Use Node.js & Express.js?

✅ **Fast and non-blocking I/O operations** – Ideal for real-time applications.
✅ **Full-stack JavaScript** – Use JavaScript for both frontend and backend.
✅ **Microservices support** – Build modular and scalable apps.
✅ **Active open-source community** – Plenty of plugins and tools available.

CHAPTER 2: INSTALLING NODE.JS AND NPM (NODE PACKAGE MANAGER)

2.1 Download and Install Node.js

To install Node.js:

1. Visit the official website: https://nodejs.org/
2. Choose the **LTS (Long-Term Support) version** (recommended for stability).
3. Download the installer for **Windows, macOS, or Linux**.
4. Run the installer and follow the on-screen setup instructions.

2.2 Verify Node.js and npm Installation

After installation, open the **terminal (Command Prompt or PowerShell)** and check the versions:

```
node -v   # Checks Node.js version

npm -v    # Checks npm version
```

✅ **If both commands return version numbers, Node.js and npm are successfully installed.**

Chapter 3: Setting Up a Node.js Project

3.1 Creating a New Node.js Project

1. Open a terminal and navigate to your desired project folder:
2. cd /path/to/project
3. Initialize a new Node.js project:
4. npm init -y

This command creates a package.json file, which manages dependencies and configurations.

3.2 Understanding package.json

The package.json file includes:

- **Project metadata** (name, version, description).
- **Scripts** (commands for starting the server).
- **Dependencies** (installed npm packages).

Chapter 4: Installing and Setting Up Express.js

4.1 Install Express.js

To install Express.js, run:

npm install express

This will download Express.js and add it to the **dependencies** section in package.json.

4.2 Create a Basic Express Server

1. Create a new file server.js inside your project folder.
2. Add the following code to create a simple Express server:

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

const app = express();

const PORT = 3000;


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

  res.send("Hello, Express Server!");

});


app.listen(PORT, () => {

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

});
```

3. Start the server by running:
4. node server.js
5. Open a browser and visit: http://localhost:3000/
   ✅ You should see the message **"Hello, Express Server!"** displayed in the browser.

CHAPTER 5: INSTALLING ADDITIONAL TOOLS FOR DEVELOPMENT

5.1 Installing Nodemon for Auto-Reloading

Instead of restarting the server manually, **Nodemon** automatically restarts it when file changes are detected.

npm install -g nodemon

Run the server using:

nodemon server.js

✅ **Now, any changes in server.js will automatically restart the server!**

5.2 Installing dotenv for Environment Variables

dotenv helps manage environment variables securely.

1. Install dotenv:
2. npm install dotenv
3. Create a .env file:
4. PORT=4000
5. Modify server.js to use dotenv:

```
require("dotenv").config();

const express = require("express");

const app = express();
```

```
const PORT = process.env.PORT || 3000;


app.get("/", (req, res) => {
   res.send("Hello, Express with dotenv!");
});


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

✅ **Now, the server reads the port number from the .env file.**

5.3 Installing CORS for Cross-Origin Requests

CORS (Cross-Origin Resource Sharing) is needed when working with **frontend-backend communication**.

```
npm install cors
```

Enable CORS in server.js:

```
const cors = require("cors");

app.use(cors());
```

✅ **This allows your API to be accessed by other domains (frontend applications).**

CHAPTER 6: FOLDER STRUCTURE FOR A SCALABLE EXPRESS APP

For large applications, use a structured project layout:

```
node-express-app/
 | -- node_modules/      # Installed dependencies
 | -- public/           # Static files (HTML, CSS)
 | -- src/
 |    | -- routes/       # API routes
 |    |    ├──── users.js    # User-related routes
 |    |    ├──── posts.js    # Blog post routes
 |    | -- controllers/    # Logic for handling requests
 |    | -- models/       # Database models
 |    | -- config/       # Configuration files
 |    | -- middleware/    # Custom middleware functions
 |    ├──── server.js      # Entry point
 | -- .env             # Environment variables
 | -- package.json      # Project configuration
 | -- README.md         # Documentation
```

✅ **This structure ensures scalability and maintainability.**

CHAPTER 7: HANDS-ON PRACTICE & EXERCISES

Exercise 1: Set Up a Node.js and Express.js Project

1. Install **Node.js** and **npm**.
2. Create a new Node.js project using npm init -y.
3. Install **Express.js** and set up a basic server.
4. Start the server and test it in the browser.

Exercise 2: Add Routes for Different Pages

1. Modify server.js to include multiple routes:
2. app.get("/about", (req, res) => {
3. res.send("About Page");
4. });
5.
6. app.get("/contact", (req, res) => {
7. res.send("Contact Page");
8. });
9. Restart the server and test http://localhost:3000/about and http://localhost:3000/contact.

Exercise 3: Implement Middleware

1. Install **body-parser** middleware:
2. npm install body-parser
3. Modify server.js to parse JSON requests:
4. const bodyParser = require("body-parser");
5. app.use(bodyParser.json());

6. Create a **POST route** to handle user input:
7. app.post("/submit", (req, res) => {
8. res.json({ message: "Data received", data: req.body });
9. });

Conclusion

Key Takeaways

✅ **Node.js** provides a runtime environment for executing JavaScript on the server.
✅ **Express.js** simplifies backend development by handling routes, middleware, and APIs.
✅ **Additional tools** like dotenv, cors, and nodemon enhance development efficiency.
✅ **Organizing the project structure** makes applications more scalable.

🚀 **Next Steps:**

- Learn **database integration with MongoDB (Mongoose)**.
- Explore **authentication with JWT (JSON Web Token)**.
- Deploy your Node.js application using **Heroku or Vercel**!

By following this guide, you are now ready to build **full-stack web applications with Node.js and Express.js**! 🎉 🔥

NODE.JS & EXPRESS.JS BASICS: UNDERSTANDING NODE.JS EVENT LOOP

CHAPTER 1: INTRODUCTION TO NODE.JS

1.1 What is Node.js?

Node.js is a **JavaScript runtime environment** that allows developers to run JavaScript code **outside the browser**. It is built on **Google Chrome's V8 engine**, making it **fast, scalable, and efficient** for backend development.

1.2 Why Use Node.js?

✅ **Non-blocking I/O operations** – Handles multiple requests asynchronously.
✅ **Single-threaded architecture** – Uses event-driven programming to improve performance.
✅ **Fast execution** – Built on the V8 engine, optimized for speed.
✅ **Full-stack JavaScript** – Enables developers to use **JavaScript for both frontend and backend**.
✅ **Supports real-time applications** – Used in chat apps, gaming platforms, and streaming services.

1.3 Installing Node.js

To install Node.js, follow these steps:

1. Visit https://nodejs.org/
2. Download the **LTS (Long-Term Support) version**.
3. Install it using the guided setup.
4. Verify installation by running:

5. node -v # Check Node.js version
6. npm -v # Check npm version

CHAPTER 2: UNDERSTANDING THE NODE.JS EVENT LOOP

2.1 What is the Event Loop in Node.js?

The **Node.js Event Loop** is a **mechanism that handles asynchronous operations** efficiently. Since Node.js is **single-threaded,** it uses an **event-driven architecture** to process multiple tasks without blocking the execution flow.

- Instead of waiting for operations like **database queries or file I/O,** the event loop allows **Node.js to handle other tasks while waiting for the result**.

2.2 How Does the Event Loop Work?

The event loop follows these **six phases**:

Phase	Description
Timers	Executes callbacks from setTimeout() and setInterval().
Pending Callbacks	Executes I/O callbacks deferred from the previous cycle.
Idle, Prepare	Internal Node.js operations (not commonly used by developers).
I/O Polling	Retrieves new I/O events, executes I/O callbacks.
Check Phase	Executes setImmediate() callbacks.

Close Callbacks	Executes clean-up callbacks for closed connections (e.g., socket.on('close')).

2.3 Event Loop in Action (Example)

Synchronous vs Asynchronous Code in Node.js

```
console.log("Start");


// Simulating asynchronous operation
setTimeout(() => {
   console.log("Timeout Callback");
}, 2000);


console.log("End");
```

💡 **Expected Output:**

```
Start
End
Timeout Callback  // Executed after 2 seconds
```

Explanation:

1. "Start" is printed first (synchronous operation).
2. setTimeout() is added to the event loop and scheduled to execute after 2 seconds.

3. "End" is printed immediately.
4. After 2 seconds, "Timeout Callback" is executed by the event loop.

2.4 Event Loop Order with setTimeout, setImmediate, and process.nextTick

```
console.log("Start");


setTimeout(() => {
  console.log("setTimeout");
}, 0);


setImmediate(() => {
  console.log("setImmediate");
});


process.nextTick(() => {
  console.log("process.nextTick");
});


console.log("End");
```

💡 **Expected Output:**

Start

End

process.nextTick

setTimeout / setImmediate (execution order may vary)

Explanation:

- process.nextTick() executes **before any asynchronous tasks** in the event loop.
- setTimeout() and setImmediate() both execute in the **next cycle,** but their order may change depending on execution timing.

CHAPTER 3: PRACTICAL USE CASES OF THE EVENT LOOP

3.1 Handling Multiple API Requests in a Web Server

A traditional blocking server would handle **one request at a time,** slowing down performance. Node.js **event loop** allows handling **thousands of concurrent requests** efficiently.

Example: Creating a Simple Web Server in Node.js

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

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

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

  res.end("Hello from Node.js Server!");

});
```

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

✅ **Handles multiple requests asynchronously without blocking execution.**

3.2 Real-Time Chat Applications

The **event loop** allows handling multiple **chat messages, notifications, and updates** in real-time.

Example: Using setInterval for Real-time Data Fetching

```
setInterval(() => {
  console.log("Fetching new messages...");
}, 3000);
```

✅ **Continuously executes without blocking other operations.**

3.3 File Handling in Node.js

Instead of blocking execution while reading files, the **event loop** allows other tasks to continue.

Example: Reading a File Asynchronously

```
const fs = require("fs");
fs.readFile("example.txt", "utf8", (err, data) => {
  if (err) throw err;
```

```
  console.log("File Content:", data);
});
console.log("Reading file...");
```

💡 **Expected Output:**

Reading file...

File Content: (contents of example.txt)

✅ **Node.js does not wait for readFile() to complete before executing the next line.**

Chapter 4: Best Practices for Optimizing the Event Loop

✅ **Use Asynchronous Functions** (fs.readFile, fetch) to avoid blocking execution.
✅ **Minimize Heavy Computations** in the main thread (use **Worker Threads** for CPU-intensive tasks).
✅ **Use process.nextTick() Carefully** – Overusing it can block other asynchronous tasks.
✅ **Optimize Database Queries** – Use **pagination and indexing** to reduce processing time.
✅ **Avoid Deeply Nested Callbacks** – Use **Promises or Async/Await** for better readability.

Chapter 5: Exercises on the Event Loop

Exercise 1: Simulating a Delayed Operation

1. Create a function that **logs a message** after 3 seconds using setTimeout().
2. Print "Start" before calling the function.
3. Print "End" immediately after calling the function.

Exercise 2: Experiment with process.nextTick() and setImmediate()

1. Create a Node.js script with process.nextTick() and setImmediate().
2. Observe which one executes first.

Exercise 3: Create an Asynchronous File Reader

1. Write a Node.js script that **reads a file asynchronously**.
2. Print "Reading file..." before the file is read.
3. Verify the output order.

CHAPTER 6: CASE STUDY - HOW NETFLIX USES NODE.JS EVENT LOOP

Problem

Netflix needed a **fast, scalable solution** to handle **millions of concurrent users** streaming video.

Solution

✅ Adopted **Node.js event loop** for **asynchronous processing of streaming data**.
✅ Used **non-blocking I/O** to handle **thousands of concurrent**

requests.

✅ Optimized performance with **load balancing and caching**.

Outcome

🔥 **40% faster startup time,** improving **user experience and scalability**.

CONCLUSION

Key Takeaways

✅ **Node.js is event-driven and uses a single-threaded model.**
✅ **The event loop efficiently manages asynchronous tasks.**
✅ **Asynchronous programming helps handle multiple requests without blocking execution.**
✅ **Optimizing the event loop is crucial for high-performance applications.**

🚀 **Next Steps:**

- Learn about **Node.js Streams** for handling large data.
- Explore **Worker Threads** for CPU-intensive tasks.
- Implement **Express.js** to build scalable APIs!

By mastering the **Node.js event loop,** you can build **fast, scalable web applications** that handle multiple requests efficiently. 🎉 🚀

NPM & PACKAGE MANAGEMENT IN REACT.JS

CHAPTER 1: INTRODUCTION TO NPM (NODE PACKAGE MANAGER)

1.1 What is npm?

npm (**Node Package Manager**) is the default **package manager** for Node.js. It is used to **install, manage, and share JavaScript libraries and dependencies** required for web development.

1.2 Why Use npm?

✅ **Manages dependencies** – Automates the installation of required packages.
✅ **Simplifies project setup** – Reduces manual downloads of libraries.
✅ **Supports version control** – Allows updating and maintaining package versions.
✅ **Access to a vast ecosystem** – Provides access to thousands of open-source packages.

1.3 Checking if npm is Installed

To check if npm is installed, open the terminal and run:

```
npm -v
```

If npm is installed, it will return a version number.

💡 **Note**: npm comes bundled with **Node.js**. If you don't have it installed, download it from https://nodejs.org/.

CHAPTER 2: INSTALLING AND USING NPM

2.1 Installing a Global npm Package

To install a package globally (**accessible from any project**), use:

npm install -g <package-name>

For example, install **create-react-app** globally:

npm install -g create-react-app

2.2 Installing a Local npm Package

To install a package **inside a specific project**, use:

npm install <package-name>

Example: Install **Axios** (used for API requests) in a project:

npm install axios

This creates a node_modules folder containing the package.

💡 **Tip**: Use npm i <package-name> as a shorthand for npm install.

CHAPTER 3: UNDERSTANDING PACKAGE.JSON & PACKAGE-LOCK.JSON

3.1 What is package.json?

The package.json file stores **metadata about a project** and its dependencies. It is **automatically created** when running:

npm init

or

npx create-react-app my-app

Example: package.json File

```
{
 "name": "my-react-app",
 "version": "1.0.0",
 "dependencies": {
  "react": "^18.0.0",
  "axios": "^0.24.0"
 },
 "scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build"
 }
}
```

3.2 What is package-lock.json?

package-lock.json **records the exact version** of installed dependencies to maintain consistency across environments.

💡 **Tip**: Never manually edit package-lock.json; npm automatically updates it when running npm install.

CHAPTER 4: MANAGING NPM PACKAGES

4.1 Installing a Specific Version of a Package

To install a specific version of a package, use:

npm install <package-name>@<version>

Example: Install React version 17.0.2 instead of the latest version:

npm install react@17.0.2

4.2 Updating npm Packages

To update a package to the latest version:

npm update <package-name>

To update **all dependencies**:

npm update

4.3 Uninstalling npm Packages

To remove an installed package, use:

npm uninstall <package-name>

Example: Uninstall Axios:

npm uninstall axios

💡 **Tip**: If a package is no longer needed, uninstall it to keep the project lightweight.

CHAPTER 5: USING NPM SCRIPTS

5.1 What are npm Scripts?

npm scripts are **custom commands** defined in package.json to automate tasks like **running the app, testing, and building**.

5.2 Running Default npm Scripts

The most common npm scripts are:

Command	Description
npm start	Starts the development server
npm run build	Builds the project for production
npm test	Runs unit tests
npm run eject	Removes the CRA abstraction (not recommended)

Example: Running a React app:

npm start

5.3 Adding Custom npm Scripts

You can define **custom scripts** inside package.json:

```
"scripts": {
  "dev": "react-scripts start",
  "deploy": "npm run build && firebase deploy"
}
```

Now, running:

npm run dev

will start the development server.

CHAPTER 6: NPM VS YARN – WHICH ONE TO USE?

Feature	npm	yarn
Speed	Slower	Faster

Lockfile	package-lock.json	yarn.lock
Security	Requires npm audit	More secure by default
Offline Mode	Limited	Better caching

💡 **Tip**: You can replace npm with yarn by running:

npm install -g yarn

Then use:

yarn install

CHAPTER 7: EXERCISES ON NPM & PACKAGE MANAGEMENT

Exercise 1: Install and Uninstall Packages

1. Install lodash in a React project.
2. Check if it appears in package.json.
3. Uninstall lodash and verify removal.

Exercise 2: Add and Run a Custom npm Script

1. Inside package.json, add:

```
"scripts": {
  "hello": "echo 'Hello, npm!'"
}
```

Run the script using:

```
npm run hello
```

Observe the output.

CHAPTER 8: CASE STUDY – MANAGING DEPENDENCIES IN A LARGE PROJECT

Scenario:

A development team is working on a **React-based e-commerce application**. The project has **multiple dependencies** such as:

- react-router-dom for navigation
- redux for state management
- axios for API requests

Challenges:

✅ Keeping dependencies **updated without breaking the app**.
✅ Preventing **inconsistent package versions across team members**.
✅ Managing dependencies in **production and development environments**.

Solution:

1. Use **package.json** to maintain project dependencies.
2. Run **npm install** with a package-lock.json file to ensure consistency.
3. Automate dependency updates using:
4. npm outdated # Check outdated packages
5. npm update # Update packages

Outcome: The team maintained a **stable, well-managed React project** with minimal dependency conflicts.

CONCLUSION

Key Takeaways:

✅ npm **installs, updates, and manages dependencies** in React.js projects.
✅ package.json tracks dependencies, while package-lock.json ensures **version consistency**.
✅ npm scripts **automate tasks** like starting the server, building projects, and running tests.
✅ Regularly updating and auditing packages **improves security and performance**.

🚀 **Next Steps:**

- Explore **monorepos** using npm workspaces.
- Learn **dependency injection** for better package management.
- Implement **CI/CD workflows** with npm for automated deployment!

SETTING UP AN EXPRESS SERVER IN NODE.JS

CHAPTER 1: INTRODUCTION TO EXPRESS.JS

1.1 What is Express.js?

Express.js is a **minimal and flexible web framework** for Node.js, used for building **server-side applications and APIs**. It simplifies **handling requests, routing, middleware integration, and database operations** in Node.js applications.

1.2 Why Use Express.js?

✅ **Lightweight & Fast** – Minimal overhead, making API responses quicker.
✅ **Routing System** – Easily create RESTful APIs with structured endpoints.
✅ **Middleware Support** – Allows processing of requests (authentication, logging).
✅ **Template Engine Compatibility** – Supports rendering dynamic HTML pages.

💡 **Common Use Cases of Express.js**

- Backend for **Web & Mobile Apps**
- RESTful **API Development**
- **Authentication Systems** (Login, JWT, OAuth)
- Real-time apps with **WebSockets**

CHAPTER 2: INSTALLING NODE.JS & EXPRESS.JS

2.1 Install Node.js and npm

To run Express.js, you need **Node.js** installed.

Step 1: Check if Node.js is Installed

node -v

npm -v

If not installed, download from https://nodejs.org/

2.2 Create a New Node.js Project

Step 2: Create a Project Folder

mkdir my-express-server

cd my-express-server

Step 3: Initialize a Node.js Project

npm init -y

This creates a package.json file containing project metadata.

2.3 Install Express.js

Step 4: Install Express

npm install express

This downloads Express and adds it to package.json.

CHAPTER 3: CREATING A BASIC EXPRESS SERVER

3.1 Setting Up the Server

Create a new file server.js in the project folder.

server.js (Basic Express Server)

```
const express = require("express");  // Import Express

const app = express();  // Initialize Express

const PORT = 3000;


// Define a simple route

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

   res.send("Welcome to Express Server!");

});


// Start the server

app.listen(PORT, () => {

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

});
```

💡 **Explanation:**

- express() initializes the app.
- app.get("/", callback) defines a route (GET /).
- app.listen(PORT, callback) starts the server on localhost:3000.

3.2 Running the Server

Run the server using:

```
node server.js
```

✅ Open a browser and visit **http://localhost:3000/** to see the response.

CHAPTER 4: HANDLING ROUTES IN EXPRESS.JS

4.1 Defining Multiple Routes

```
app.get("/about", (req, res) => {
  res.send("About Page");
});

app.get("/contact", (req, res) => {
  res.send("Contact Page");
});
```

✅ Navigate to **http://localhost:3000/about** or **http://localhost:3000/contact** to see responses.

4.2 Handling POST Requests

```
app.post("/submit", (req, res) => {
  res.send("Form Submitted!");
});
```

💡 **Use Postman or a frontend form** to test POST requests.

Chapter 5: Using Middleware in Express.js

5.1 What is Middleware?

Middleware functions **process requests** before they reach the final route handler.

5.2 Adding a Middleware Function

```
app.use((req, res, next) => {
  console.log(`Request received at ${new Date().toISOString()}`);
  next();  // Pass control to the next handler
});
```

💡 **Middleware is useful for logging, authentication, and request validation.**

Chapter 6: Serving Static Files with Express.js

6.1 Hosting HTML, CSS, and Images

To serve static files (e.g., index.html), place them in a public folder and add:

```
app.use(express.static("public"));
```

📁 **Project Structure**

```
my-express-server/
 | -- public/
 |    | -- index.html
 |    | -- styles.css
```

```
|-- server.js
|-- package.json
```

6.2 Example: index.html

Create public/index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Express Server</title>
</head>
<body>
  <h1>Welcome to My Express Server</h1>
</body>
</html>
```

✅ Visit **http://localhost:3000/index.html** to see the page.

CHAPTER 7: CONNECTING EXPRESS.JS TO A DATABASE (MONGODB)

7.1 Install Mongoose for MongoDB

```
npm install mongoose
```

7.2 Connect to MongoDB in server.js

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

```
mongoose.connect("mongodb://localhost:27017/mydatabase", {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => console.log("Connected to MongoDB"))
 .catch(err => console.error("MongoDB connection error:", err));
```

💡 **MongoDB is useful for storing user data, orders, and blog posts.**

CHAPTER 8: HANDLING JSON REQUESTS IN EXPRESS.JS

8.1 Using express.json() to Parse JSON Data

```
app.use(express.json());


app.post("/data", (req, res) => {
  console.log(req.body);
  res.send("Data received");
});
```

✅ Use **Postman** or a frontend form to send a JSON request:

```
{
  "name": "John",
  "email": "john@example.com"
}
```

Chapter 9: Deploying Express.js Server

9.1 Deploying with Heroku

Install Heroku CLI:

npm install -g heroku

Login to Heroku:

heroku login

Initialize Git and push to Heroku:

git init

git add .

git commit -m "Deploy Express App"

heroku create

git push heroku master

✅ Your Express server is now **live on Heroku**! 🎉

Chapter 10: Hands-on Practice & Assignments

Exercise 1: Set Up a Basic Express Server

1. Install **Node.js & Express.js**.
2. Create a basic server (server.js).
3. Start the server and visit **localhost:3000**.

Exercise 2: Create Multiple Routes

1. Define routes: /home, /services, /team.
2. Display different messages on each route.

Exercise 3: Implement Middleware Logging

1. Add a middleware function to **log request details**.
2. Ensure it logs every request before reaching the route handler.

Exercise 4: Serve Static Files

1. Add a public/ folder with an **HTML, CSS, and JS file**.
2. Serve them using express.static().

Exercise 5: Deploy Your Express Server

1. Create a GitHub repository and push the code.
2. Deploy the server on **Heroku** or **Render**.

Conclusion

✅ You have successfully learned **how to set up an Express.js server**!
This setup allows you to build **REST APIs, web applications, and full-stack projects**.

🚀 **Next Steps:**

- Learn **Authentication (JWT, OAuth)**.
- Build a **CRUD API** with Express & MongoDB.
- Explore **real-time features** using **WebSockets (Socket.io)**!

Happy Coding! 🎉 🚀

RESTful API Development: Creating APIs with CRUD Operations in Node.js & Express.js

Chapter 1: Introduction to RESTful APIs

1.1 What is a RESTful API?

A **RESTful API (Representational State Transfer API)** is an **architectural style** used to create **web services** that communicate using **HTTP requests**. It follows six guiding principles:

✅ **Client-Server** – The client and server operate independently.

✅ **Stateless** – Each request contains all necessary information; the server does not store client state.

✅ **Cacheable** – Responses should be cacheable for efficiency.

✅ **Layered System** – Multiple layers can exist between client and server.

✅ **Uniform Interface** – Standardized request formats (GET, POST, PUT, DELETE).

✅ **Code on Demand (Optional)** – Servers can send executable code to clients.

1.2 Why Use RESTful APIs?

✓ **Standardized communication** – Works with any frontend or mobile app.

✓ **Scalability** – Supports large-scale applications.

✓ **Reusability** – APIs can be used by multiple applications.

✓ **Flexibility** – Works with different programming languages.

CHAPTER 2: SETTING UP A NODE.JS & EXPRESS.JS REST API

2.1 Prerequisites

- Install **Node.js** (Download from here)
- Install **Postman** (for testing API requests)
- Basic knowledge of JavaScript & HTTP methods

2.2 Installing Express.js

Express.js is a **minimal and flexible web framework** for Node.js, making API development easier.

Step 1: Create a New Node.js Project

Run the following commands in the terminal:

```
mkdir rest-api

cd rest-api

npm init -y  # Initializes package.json
```

Step 2: Install Required Dependencies

```
npm install express cors body-parser nodemon
```

- express – Handles API routing.
- cors – Enables cross-origin requests.
- body-parser – Parses JSON request bodies.
- nodemon – Automatically restarts server on file changes.

Step 3: Create an Express Server (server.js)

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

const cors = require("cors");
```

```
const app = express();

app.use(cors());

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


const PORT = 5000;

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

✅ **The Express server is now set up and running!**

CHAPTER 3: UNDERSTANDING CRUD OPERATIONS IN RESTFUL APIS

CRUD stands for:

- **C**reate (POST) – Add new data
- **R**ead (GET) – Retrieve existing data
- **U**pdate (PUT/PATCH) – Modify existing data
- **D**elete (DELETE) – Remove data

CHAPTER 4: CREATING APIS WITH CRUD OPERATIONS

4.1 Creating a Sample Data Store

Since we are not using a database yet, let's create an **in-memory store** using an array.

```
let users = [
  { id: 1, name: "Alice", email: "alice@example.com" },
  { id: 2, name: "Bob", email: "bob@example.com" }
];
```

4.2 Implementing CRUD Endpoints

4.2.1 Create (POST) - Add a New User

The POST method **adds new data** to the server.

```
app.post("/users", (req, res) => {
  const { name, email } = req.body;
  const newUser = { id: users.length + 1, name, email };
  users.push(newUser);
  res.status(201).json(newUser);
});
```

✅ **Sends new user data in JSON format.**
✅ **Returns the newly created user object.**

💡 **Test with Postman:**

- **Method:** POST
- **URL:** http://localhost:5000/users
- **Body (JSON):**

```
{
```

```
  "name": "Charlie",
  "email": "charlie@example.com"
}
```

4.2.2 Read (GET) - Retrieve All Users

The GET method **fetches all user data**.

```
app.get("/users", (req, res) => {
  res.json(users);
});
```

✅ **Returns a list of users in JSON format.**

💡 **Test with Postman:**

- **Method:** GET
- **URL:** http://localhost:5000/users

4.2.3 Read (GET) - Retrieve a Single User by ID

The GET method fetches **a user based on their id**.

```
app.get("/users/:id", (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (!user) return res.status(404).json({ message: "User not found" });
  res.json(user);
});
```

✅ **Uses req.params.id to get the user's ID from the URL.**
✅ **Returns 404 Not Found if the user does not exist.**

💡 **Test with Postman:**

- **Method:** GET
- **URL:** http://localhost:5000/users/1

4.2.4 Update (PUT) - Modify a User

The PUT method **updates an existing user**.

```
app.put("/users/:id", (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (!user) return res.status(404).json({ message: "User not found" });

  user.name = req.body.name || user.name;
  user.email = req.body.email || user.email;
  res.json(user);
});
```

✅ **Updates the user's name or email.**
✅ **Returns the updated user object.**

💡 **Test with Postman:**

- **Method:** PUT
- **URL:** http://localhost:5000/users/1

- **Body (JSON):**

```
{
  "name": "Alice Johnson",
  "email": "alice.johnson@example.com"
}
```

4.2.5 Delete (DELETE) - Remove a User

The DELETE method **removes a user from the list**.

```
app.delete("/users/:id", (req, res) => {
  users = users.filter(u => u.id !== parseInt(req.params.id));
  res.json({ message: "User deleted successfully" });
});
```

✅ **Filters out the user with the specified id**.
✅ **Returns a success message**.

💡 **Test with Postman:**

- **Method:** DELETE
- **URL:** http://localhost:5000/users/1

CHAPTER 5: TESTING AND DEPLOYING THE REST API

5.1 Running the API Locally

Start the server:

node server.js

OR (for auto-restart on changes):

npx nodemon server.js

5.2 Deploying the API

1. **Push the code to GitHub**
2. **Deploy on Render:**
    - Create a free account at https://render.com
    - Add a new **web service** and connect your GitHub repo
    - Set **runtime** to Node.js
    - Deploy and get a live API URL

CHAPTER 6: HANDS-ON EXERCISES

Exercise 1: Extend the API

✅ Add a **PATCH endpoint** for updating only one field at a time.

Exercise 2: Add Database Support

✅ Replace the in-memory users array with **MongoDB (Mongoose)**.

Exercise 3: Implement Authentication

✅ Secure the API with **JWT authentication**.

CONCLUSION

✅ **RESTful APIs follow CRUD principles using HTTP methods.**
✅ **Express.js simplifies API development in Node.js.**
✅ **Postman helps test API endpoints efficiently.**
✅ **Deployment on Render makes the API accessible globally.**

🚀 **Next Steps:**

- Learn **MongoDB with Mongoose** for persistent data storage.
- Implement **JWT authentication** for secure API access.
- Build a **frontend in React.js to consume this API!**

RESTful API Development: Request Validation & Error Handling

Chapter 1: Introduction to RESTful APIs

1.1 What is a RESTful API?

A **RESTful API (Representational State Transfer API)** allows applications to communicate over the internet using **HTTP methods** like GET, POST, PUT, and DELETE. REST APIs follow **stateless communication,** meaning each request is independent and does not store session data on the server.

1.2 Why is Request Validation & Error Handling Important?

✅ Ensures **data integrity** by validating user inputs.
✅ Prevents **security vulnerabilities** such as SQL injection and cross-site scripting (XSS).
✅ Provides **better error messages** to guide developers and users.
✅ Improves **API reliability and user experience**.

Chapter 2: Setting Up a REST API with Express.js

2.1 Installing Node.js & Express

First, install **Node.js** and create an Express.js server:

mkdir rest-api

cd rest-api

npm init -y

npm install express

Create a file server.js and set up a basic API:

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

const app = express();

const port = 3000;


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


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

   res.send("Welcome to the REST API!");

});


app.listen(port, () => {

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

});
```

Run the server:

```
node server.js
```

Now, the API is available at http://localhost:3000/.

CHAPTER 3: REQUEST VALIDATION IN REST APIS

3.1 What is Request Validation?

Request validation ensures that API clients send **valid and expected data** before processing a request. Common validations include:

✅ **Required Fields** – Ensuring that mandatory fields are provided.

✅ **Data Types** – Verifying string, number, email, etc.

✅ **Value Constraints** – Checking length, minimum/maximum values.

✅ **Custom Rules** – Ensuring unique usernames, valid dates, etc.

3.2 Validating Requests Using Express Validator

We will use **express-validator**, a middleware for validating API requests.

Step 1: Install express-validator

```
npm install express-validator
```

Step 2: Validate User Registration Input

```
const { body, validationResult } = require("express-validator");


app.post("/register", [
  body("username").isLength({ min: 3 }).withMessage("Username must be at least 3 characters"),
  body("email").isEmail().withMessage("Invalid email format"),
  body("password").isLength({ min: 6 }).withMessage("Password must be at least 6 characters")
], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
```

```
    return res.status(400).json({ errors: errors.array() });
  }
  res.send("User registered successfully!");
});
```

💡 **How It Works:**

- The body() method checks for field values.
- If validation fails, validationResult() returns error messages.
- Errors are sent with a **400 Bad Request** status.

CHAPTER 4: ERROR HANDLING IN REST APIS

4.1 Importance of Proper Error Handling

Without error handling, users may receive **cryptic error messages** or incorrect data. A **good API** should:

✅ Return **clear, meaningful error messages**.
✅ Use **proper HTTP status codes** (400, 404, 500, etc.).
✅ Log errors for **debugging and monitoring**.

4.2 Handling Errors with Try-Catch Blocks

Wrap API logic inside try-catch blocks to handle errors gracefully.

Example: Handling Errors in User Registration

```
app.post("/register", async (req, res) => {
  try {
    let { username, email, password } = req.body;
```

```
    if (!username || !email || !password) {
      throw new Error("All fields are required");
    }

    // Simulating database interaction
    if (username === "admin") {
      throw new Error("Username already taken");
    }

    res.status(201).json({ message: "User registered successfully!" });
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});
```

✅ **Uses try-catch** to prevent server crashes.
✅ **Returns structured error responses** (error: error.message).

CHAPTER 5: GLOBAL ERROR HANDLING MIDDLEWARE

5.1 What is Middleware?

Middleware functions process requests **before sending responses**. A **global error handler** can catch errors from **any route** in the API.

5.2 Implementing Global Error Handling

Create a centralized error handler in server.js:

```
// Global Error Handler
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ error: "Internal Server Error" });
});

// Example route that triggers an error
app.get("/error", (req, res, next) => {
  try {
    throw new Error("Something went wrong!");
  } catch (err) {
    next(err);  // Pass error to the global handler
  }
});
```

✅ **Logs errors for debugging** (console.error(err.stack)).
✅ **Responds with a structured error message** (res.status(500)).

CHAPTER 6: USING HTTP STATUS CODES FOR ERROR HANDLING

6.1 Common HTTP Status Codes in REST APIs

Status Code	Meaning	Usage
200 OK	Success	Used for GET requests
201 Created	Resource created	Used for POST requests
400 Bad Request	Client-side validation error	Missing or invalid input
401 Unauthorized	Authentication failure	Wrong API key, token missing
403 Forbidden	Access denied	User does not have permissions
404 Not Found	Resource not found	Invalid URL or missing record
500 Internal Server Error	Server failure	Unexpected errors

💡 **Tip:** Always return **correct status codes** for better debugging.

CHAPTER 7: CASE STUDY – HANDLING ERRORS IN AN E-COMMERCE API

Scenario:

A company launched an **e-commerce REST API** to manage products and users. However, they faced **two critical issues**:

1. Users **entered invalid product details,** causing crashes.
2. Customers received **generic error messages** when placing orders.

Solution:

✅ **Implemented request validation** for adding products:

```
app.post("/products", [
  body("name").notEmpty().withMessage("Product name is required"),
  body("price").isFloat({ min: 1 }).withMessage("Price must be greater than 0")
], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  res.status(201).json({ message: "Product added successfully!" });
});
```

✅ **Added a global error handler** to return clear responses.

Outcome:

🚀 **API error rate reduced by 60%.**
🚀 **Improved user experience with clearer error messages.**

Chapter 8: Exercises on Request Validation & Error Handling

Exercise 1: Validate User Login Requests

1. Create a POST /login route that:
   - ✅ Requires email and password.
   - ✅ Returns 400 Bad Request for missing fields.
   - ✅ Returns 200 OK for valid login.

Exercise 2: Handle Product API Errors

1. Create a POST /products route.
2. Validate name (required) and price (must be positive).
3. Use try-catch to catch unexpected errors.

Conclusion

Key Takeaways:

✅ **Request validation** prevents invalid data from reaching the server.
✅ **Proper error handling** ensures APIs provide clear and useful messages.
✅ **HTTP status codes** help communicate API responses effectively.
✅ **Global error handling middleware** improves maintainability.

🚀 Next Steps:

- Implement **authentication middleware** to validate API tokens.
- Learn about **rate limiting** to prevent API abuse.

- Explore **logging tools like Winston and Morgan** for debugging!

RESTful API Development: Middleware in Express.js

Chapter 1: Introduction to Middleware in Express.js

1.1 What is Middleware?

Middleware in Express.js refers to **functions that process incoming HTTP requests** before they reach the final route handler.
Middleware functions can:

- ✅ Modify request and response objects
- ✅ Execute code before sending a response
- ✅ Handle authentication and authorization
- ✅ Serve static files
- ✅ Handle errors

1.2 Why Use Middleware?

Middleware helps in:

✓ **Logging and debugging** – Record incoming requests.
✓ **Request validation** – Check if required fields exist in a request.
✓ **Security measures** – Implement authentication.
✓ **Data processing** – Parse incoming JSON and URL-encoded data.

Chapter 2: Types of Middleware in Express.js

Express middleware can be categorized into four types:

2.1 Application-Level Middleware

Applies to all requests or specific routes.

2.2 Router-Level Middleware

Used for specific route handlers.

2.3 Built-in Middleware

Pre-defined middleware available in Express.js.

2.4 Error-Handling Middleware

Handles errors across the application.

CHAPTER 3: SETTING UP EXPRESS AND MIDDLEWARE

3.1 Install Express.js

npm install express

3.2 Create an Express Server with Middleware

```
const express = require("express");
const app = express();
const PORT = 3000;

// Custom Middleware Function
app.use((req, res, next) => {
  console.log(`Request Method: ${req.method}, URL: ${req.url}`);
  next();  // Pass control to the next middleware
});
```

```
app.get("/", (req, res) => {
  res.send("Welcome to the Express Server!");
});

app.listen(PORT, () => {
  console.log(`Server running at http://localhost:${PORT}`);
});
```

✅ **Logs every request before passing it to route handlers.**

CHAPTER 4: APPLICATION-LEVEL MIDDLEWARE

4.1 Using app.use() for Global Middleware

Application-level middleware runs **before reaching the routes**.

```
app.use((req, res, next) => {
  console.log("Application-Level Middleware Executed!");
  next();
});
```

💡 **Common Uses:**

- Logging requests
- Validating API keys
- Applying security policies

Chapter 5: Router-Level Middleware

5.1 Attaching Middleware to Specific Routes

Middleware can be applied to **specific routes** instead of globally.

Example: Middleware for Protected Routes

```
const checkAuth = (req, res, next) => {
  const auth = req.headers.authorization;
  if (!auth) {
    return res.status(403).json({ message: "Unauthorized" });
  }
  next();
};

app.get("/dashboard", checkAuth, (req, res) => {
  res.send("Welcome to the Dashboard!");
});
```

✅ **Prevents unauthorized access to /dashboard.**

Chapter 6: Built-in Middleware in Express.js

Express provides **predefined middleware** for common tasks:

6.1 express.json() – Parsing JSON Data

Parses JSON from incoming requests.

```
app.use(express.json());


app.post("/data", (req, res) => {
   console.log(req.body);  // Access parsed JSON
   res.send("Data received!");
});
```

6.2 express.urlencoded() – Parsing Form Data

Handles form submissions.

```
app.use(express.urlencoded({ extended: true }));


app.post("/form", (req, res) => {
   console.log(req.body);
   res.send("Form submitted successfully!");
});
```

6.3 express.static() – Serving Static Files

Hosts HTML, CSS, and images.

```
app.use(express.static("public"));
```

✅ **Visit http://localhost:3000/index.html to view static content.**

CHAPTER 7: ERROR-HANDLING MIDDLEWARE

7.1 Using Error-Handling Middleware

Express provides a built-in mechanism to catch errors across the application.

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ message: "Internal Server Error" });
});
```

💡 **Best Practice:**
Always define the **error-handling middleware last** in your middleware stack.

CHAPTER 8: THIRD-PARTY MIDDLEWARE IN EXPRESS.JS

Express allows integrating third-party middleware packages.

8.1 Using cors Middleware for Cross-Origin Requests

```
npm install cors
```

```
const cors = require("cors");
app.use(cors());
```

✅ **Allows requests from different domains (important for APIs).**

8.2 Using morgan for Logging Requests

```
npm install morgan
```

```
const morgan = require("morgan");
app.use(morgan("dev"));
```

✅ **Logs detailed HTTP request information.**

Chapter 9: Real-World Use Cases of Middleware

9.1 Authentication Middleware for Protected Routes

```
const verifyToken = (req, res, next) => {
  const token = req.headers["authorization"];
  if (!token) return res.status(401).send("Access Denied");
  next();
};

app.get("/protected", verifyToken, (req, res) => {
  res.send("Protected Route Accessed!");
});
```

✅ **Useful for JWT authentication in secure APIs.**

Chapter 10: Hands-on Practice & Assignments

Exercise 1: Implement Global Logging Middleware

1. Create an Express server.
2. Log each request's **method, URL, and timestamp**.

Exercise 2: Create Route-Specific Middleware

1. Define a route /admin.
2. Apply middleware that **blocks unauthorized users**.

Exercise 3: Implement JSON Request Parsing

1. Accept JSON requests via /submit-data.
2. Print request body in the terminal.

Exercise 4: Handle Errors with Middleware

1. Simulate an error in a route.
2. Implement an error-handling middleware to return a **500 error message**.

CONCLUSION

✅ Middleware enhances **Express.js functionality** by managing **requests, security, logging, and errors**.
✅ Built-in middleware like **express.json()** and **express.static()** simplify development.
✅ **Custom middleware** ensures controlled execution before processing requests.

🚀 **Next Steps:**

- Implement **JWT authentication middleware**.
- Build an Express API with **rate-limiting middleware**.
- Explore **GraphQL API middleware** for advanced data querying!

Happy Coding! 🎉 🚀

SECURITY & AUTHENTICATION: INTRODUCTION TO JWT (JSON WEB TOKEN)

CHAPTER 1: INTRODUCTION TO JWT (JSON WEB TOKEN)

1.1 What is JWT?

JWT (**JSON Web Token**) is a **secure, compact, and self-contained** way of transmitting information between parties as a **JSON object**. It is commonly used for **authentication and secure data exchange** in web applications. JWTs are **digitally signed** using **HMAC (Hash-Based Message Authentication Code) or RSA** to ensure integrity and security.

1.2 Why Use JWT?

✅ **Stateless Authentication** – No need to store session data on the server.
✅ **Compact & Fast** – JWTs are small and can be transmitted quickly.
✅ **Self-contained** – Contains all necessary user information, reducing database queries.
✅ **Secure** – Uses digital signatures for integrity verification.

1.3 Common Use Cases of JWT

- **User Authentication** – Securely verify users in login systems.
- **API Authorization** – Grant access to protected API routes.
- **Single Sign-On (SSO)** – Enable seamless authentication across multiple applications.
- **Microservices Communication** – Authenticate services within a distributed system.

CHAPTER 2: STRUCTURE OF A JWT

A **JWT token** consists of **three parts,** separated by dots (.):

header.payload.signature

Example of a JWT:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.

eyJ1c2VySWQiOiIxMjM0NTYiLCJyb2xlIjoiYWRtaW4iLCJleHAiOjE2NzM5MDAwMDB9.

SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

2.1 Header (Algorithm & Token Type)

The **header** contains metadata about the token, such as **signing algorithm and token type**.

{

"alg": "HS256",

"typ": "JWT"

}

- "alg": "HS256" → Uses HMAC SHA-256 algorithm for signing.
- "typ": "JWT" → Specifies that the token is a JSON Web Token.

2.2 Payload (User Data & Claims)

The **payload** contains **claims,** which are statements about the user and additional metadata.

{

```
 "userId": "123456",
 "role": "admin",
 "exp": 1673900000
}
```

- "userId" ⟶ Identifies the authenticated user.
- "role" ⟶ Defines user privileges (admin, user, etc.).
- "exp" ⟶ Expiration timestamp (in UNIX format).

2.3 Signature (Verifies Token Integrity)

The **signature** is used to **verify that the JWT has not been altered**. It is generated using:

HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secretKey)

- This ensures that the token is **tamper-proof** and **originates from a trusted source**.

CHAPTER 3: IMPLEMENTING JWT IN NODE.JS & EXPRESS.JS

3.1 Install Required Dependencies

First, install the necessary npm packages:

```
npm install express jsonwebtoken dotenv
```

3.2 Setting Up Express.js Server

Create a server.js file and set up an Express server:

```
require("dotenv").config();
```

```
const express = require("express");
const jwt = require("jsonwebtoken");


const app = express();
app.use(express.json());


const SECRET_KEY = process.env.JWT_SECRET || "supersecretkey";
```

✅ **dotenv** is used to store the secret key securely in an .env file.

3.3 Generating a JWT Token

Create a **login route** to authenticate users and issue JWT tokens:

```
app.post("/login", (req, res) => {
  const { username } = req.body;


  if (!username) {
    return res.status(400).json({ error: "Username is required" });
  }


  // Generate JWT token
  const token = jwt.sign(
    { username: username, role: "user" },
    SECRET_KEY,
```

```
    { expiresIn: "1h" }
  );

  res.json({ token });
});
```

✅ **jwt.sign()** creates a JWT token with the user's details.
✅ **expiresIn: "1h"** sets token validity for 1 hour.

CHAPTER 4: PROTECTING ROUTES WITH JWT AUTHENTICATION

4.1 Creating an Authentication Middleware

To protect API routes, use middleware to **verify JWT tokens** before granting access.

```
const verifyToken = (req, res, next) => {
  const token = req.headers["authorization"];

  if (!token) {
    return res.status(403).json({ error: "Access Denied: No Token Provided" });
  }

  try {
    const decoded = jwt.verify(token.split(" ")[1], SECRET_KEY);
```

```
    req.user = decoded; // Store user data in request object
    next();
  } catch (err) {
    res.status(401).json({ error: "Invalid Token" });
  }
};
```

✅ **Extracts the token from the Authorization header.**
✅ **Verifies token using jwt.verify().**
✅ **Calls next() to allow access if valid.**

4.2 Protecting API Routes

Apply the middleware to secure protected routes:

```
app.get("/profile", verifyToken, (req, res) => {
  res.json({ message: "Welcome to the protected profile page!", user: req.user });
});
```

Now, only users with a **valid JWT token** can access /profile.

✅ **Test the API using Postman or CURL**

1. **Login to get a token**:
2. POST /login
3. Body: { "username": "john_doe" }
4. **Use the token to access protected routes**:

5. GET /profile
6. Headers: { Authorization: "Bearer <your_token>" }

CHAPTER 5: REFRESH TOKENS FOR ENHANCED SECURITY

5.1 Why Use Refresh Tokens?

Access tokens expire quickly (e.g., **1 hour**) to enhance security. **Refresh tokens** provide a way to generate a new access token **without requiring the user to log in again**.

5.2 Generating a Refresh Token

Modify the /login route to return both **access** and **refresh tokens**:

```
app.post("/login", (req, res) => {
  const { username } = req.body;

  const accessToken = jwt.sign({ username }, SECRET_KEY, { expiresIn: "15m" });
  const refreshToken = jwt.sign({ username }, SECRET_KEY, { expiresIn: "7d" });

  res.json({ accessToken, refreshToken });
});
```

5.3 Implementing Token Refresh API

Create a new endpoint to **exchange refresh tokens for a new access token**:

```
app.post("/refresh", (req, res) => {
  const { refreshToken } = req.body;

  if (!refreshToken) return res.status(401).json({ error: "Refresh Token Required" });

  try {
    const decoded = jwt.verify(refreshToken, SECRET_KEY);
    const newAccessToken = jwt.sign({ username: decoded.username }, SECRET_KEY, { expiresIn: "15m" });
    res.json({ accessToken: newAccessToken });
  } catch {
    res.status(403).json({ error: "Invalid Refresh Token" });
  }
});
```

✅ **If a valid refresh token is provided, a new access token is generated.**

CHAPTER 6: HANDS-ON PRACTICE & EXERCISES

Exercise 1: Implement JWT Authentication

1. Create an Express server with JWT authentication.
2. Implement **login, protected routes,** and **middleware**.

3. Store the secret key securely in an .env file.

Exercise 2: Add Role-Based Authentication

1. Modify JWT payload to include a **user role** (admin, user).
2. Restrict access to specific routes based on the role.

Exercise 3: Implement Logout Functionality

1. Store issued refresh tokens in a **database or cache**.
2. Create an API to **invalidate refresh tokens upon logout**.

CONCLUSION

Key Takeaways

✅ **JWT provides a secure way to authenticate users without storing session data.**
✅ **Tokens contain user data and expiration information.**
✅ **Middleware ensures only authenticated users can access protected routes.**
✅ **Refresh tokens improve security by allowing token renewal without re-login.**

🚀 **Next Steps:**

- Implement **OAuth authentication (Google, Facebook login)**.
- Secure APIs using **HTTPS & helmet.js**.
- Deploy the authentication system using **Docker & Kubernetes**!

By mastering JWT, you are now equipped to build **secure and scalable authentication systems** for real-world applications! 🔒🚀

Security & Authentication: Password Hashing with bcrypt.js in Node.js & Express.js

Chapter 1: Introduction to Password Hashing & Security

1.1 Why Is Password Security Important?

Storing plain-text passwords is a major **security risk**. If a database is compromised, all user credentials can be **easily accessed by attackers**. To prevent this, we use **password hashing** to store passwords in a secure manner.

1.2 What Is Password Hashing?

✅ **Hashing** is a one-way process that converts plain text into an irreversible string of characters.
✅ Hashing ensures that even if **data is leaked, actual passwords remain protected**.
✅ Hash functions create **unique outputs for different inputs** (no two passwords should have the same hash).

💡 **Example of a hashed password using bcrypt.js:**
Plain-text: "mypassword"
Hashed version:
$2b$10$4aaRr6eWc9XyF9HJ2Wk1eOR05bMygJNzUpeTx13ZmTvhz8hPmOjS.

Chapter 2: Introduction to bcrypt.js

2.1 What Is bcrypt.js?

bcrypt.js is a popular **password-hashing library** for Node.js that provides **strong encryption and protection**. It uses the **Blowfish cipher** to generate **salted and hashed passwords**, making them more resistant to brute-force attacks.

2.2 Why Use bcrypt.js?

✅ **Automatic Salting** – bcrypt **generates a unique salt** for every password.
✅ **Slow Hashing Function** – Deliberately slows down computation to **thwart brute-force attacks**.
✅ **Cross-Platform Support** – Works with Node.js, Python, and other languages.
✅ **Secure Hashing Algorithm** – Uses **adaptive hashing** that can increase in complexity over time.

2.3 Installing bcrypt.js in Node.js

To use bcrypt.js in your Node.js application, install it via npm:

```
npm install bcrypt
```

CHAPTER 3: IMPLEMENTING PASSWORD HASHING WITH BCRYPT.JS

3.1 Setting Up a Node.js Project

Step 1: Create a New Node.js Project

```
mkdir bcrypt-auth

cd bcrypt-auth

npm init -y  # Initializes package.json
```

Step 2: Install Required Dependencies

npm install express bcrypt body-parser

- express – Web framework for creating APIs.
- bcrypt – For password hashing.
- body-parser – Parses request body (for handling user input).

Step 3: Create server.js

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

const bcrypt = require("bcrypt");


const app = express();

app.use(express.json()); // Middleware to parse JSON request body


const users = []; // In-memory user store


app.listen(5000, () => console.log("Server running on http://localhost:5000"));
```

✅ **Server is now set up!**

3.2 Hashing Passwords Before Storing in Database

When a user signs up, **hash the password before storing it**.

Add a User Registration Route (/register)

```
app.post("/register", async (req, res) => {
```

```
  try {
    const { username, password } = req.body;

    // Generate a salt and hash the password
    const saltRounds = 10;
    const hashedPassword = await bcrypt.hash(password, saltRounds);

    // Store the new user
    users.push({ username, password: hashedPassword });

    res.status(201).json({ message: "User registered successfully!" });
  } catch (error) {
    res.status(500).json({ error: "Registration failed" });
  }
});
```

✅ **Uses bcrypt.hash() to hash the password before storing it.**
✅ **Salt rounds (10) define complexity (higher = more secure but slower).**

💡 **Test with Postman:**

- **Method:** POST
- **URL:** http://localhost:5000/register

- **Body (JSON):**

```
{
  "username": "johndoe",
  "password": "securepassword"
}
```

- **Response:**

```
{
  "message": "User registered successfully!"
}
```

- Check the users array, and you'll see the **hashed password stored instead of plain text**.

CHAPTER 4: VERIFYING HASHED PASSWORDS DURING LOGIN

Once a password is stored in hashed format, users need to **verify it during login**.

Add a Login Route (/login)

```
app.post("/login", async (req, res) => {
  try {
    const { username, password } = req.body;

    // Find user in database (in-memory array here)
    const user = users.find(u => u.username === username);
```

```
    if (!user) return res.status(404).json({ error: "User not found" });


    // Compare the provided password with the hashed password
    const isMatch = await bcrypt.compare(password, user.password);


    if (isMatch) {
      res.json({ message: "Login successful!" });
    } else {
      res.status(401).json({ error: "Invalid credentials" });
    }
  } catch (error) {
    res.status(500).json({ error: "Login failed" });
  }
});
```

✅ **Uses bcrypt.compare() to check if the provided password matches the hashed password.**
✅ **Returns 401 Unauthorized if the password is incorrect.**

💡 **Test with Postman:**

- **Method:** POST
- **URL:** http://localhost:5000/login
- **Body (JSON):**

```
{
  "username": "johndoe",
  "password": "securepassword"
}
```

- **Response:**

```
{
  "message": "Login successful!"
}
```

- If you enter the wrong password, the response will be:

```
{
  "error": "Invalid credentials"
}
```

Chapter 5: Best Practices for Secure Password Storage

5.1 Never Store Plain-Text Passwords

Always **hash passwords** before storing them in a database.

5.2 Use a High Number of Salt Rounds

More salt rounds increase security but **affect performance**. Recommended:

- **10 rounds** for regular applications.
- **12+ rounds** for sensitive applications (e.g., banking apps).

5.3 Use Asynchronous Hashing (bcrypt.hash())

Asynchronous hashing **prevents blocking** the event loop.

5.4 Implement Rate Limiting for Login Attempts

To prevent **brute-force attacks,** limit the number of login attempts per user.

Chapter 6: Hands-On Exercises

Exercise 1: Create a Password Reset Route

✅ Add an endpoint /reset-password that allows users to update their password securely.

Exercise 2: Implement a JWT-Based Authentication System

✅ Store user sessions using **JSON Web Tokens (JWT)** after successful login.

Exercise 3: Integrate MongoDB with Mongoose

✅ Replace the **in-memory users array** with **MongoDB to store users persistently**.

Chapter 7: Case Study - How bcrypt.js Helped Secure Facebook User Data

Problem

Facebook faced an issue where some old **user passwords were stored in plain text**. This led to a **major security concern**.

Solution

✅ **Implemented bcrypt.js for hashing passwords** before storing them.
✅ **Increased salt rounds** to improve password security.
✅ **Forced password resets** for affected users.

Outcome

🚀 **Improved user security** and prevented unauthorized access to accounts.

Conclusion

Key Takeaways

✅ **bcrypt.js is essential for secure password hashing in Node.js.**
✅ **Hashed passwords prevent data breaches and hacking attempts.**
✅ **bcrypt.hash() ensures passwords are safely stored, and bcrypt.compare() validates user logins.**
✅ **Follow best practices like using salt rounds, asynchronous hashing, and rate limiting.**

🚀 **Next Steps:**

- Integrate **JWT authentication** for secure user sessions.
- Implement **OAuth (Google/Facebook Login)** for better security.
- Store passwords securely in **MongoDB with Mongoose**.

By mastering **bcrypt.js password hashing**, you can **enhance security and prevent unauthorized access** in any Node.js application! 🎉 🚀

SECURITY & AUTHENTICATION IN RESTFUL APIS: IMPLEMENTING ROLE-BASED ACCESS CONTROL (RBAC)

CHAPTER 1: INTRODUCTION TO ROLE-BASED ACCESS CONTROL (RBAC)

1.1 What is Role-Based Access Control (RBAC)?

Role-Based Access Control (RBAC) is a security model that restricts system access **based on a user's role** within an application. Instead of granting permissions to individual users, permissions are assigned to **roles**, and users are assigned **roles**.

1.2 Why Use RBAC?

✅ **Enhances Security** – Limits user access based on their role.
✅ **Reduces Administrative Overhead** – Easily assign or modify user permissions.
✅ **Ensures Data Protection** – Prevents unauthorized access to sensitive information.
✅ **Improves Compliance** – Meets security standards (e.g., GDPR, HIPAA).

1.3 Example Use Case

Consider a **content management system (CMS)** with different user roles:

Role	Permissions
Admin	Can create, update, delete all users and content
Editor	Can create and update content but cannot delete

Viewer	Can only view content

CHAPTER 2: SETTING UP ROLE-BASED ACCESS CONTROL IN AN EXPRESS API

2.1 Prerequisites

Before implementing RBAC, ensure you have:

- **Node.js** and **Express.js** installed.
- **jsonwebtoken (JWT)** for authentication.
- **bcryptjs** for password hashing.

2.2 Install Required Packages

npm install express jsonwebtoken bcryptjs dotenv

2.3 Setting Up the Express Server

Create a file server.js and set up a basic Express app:

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

const app = express();

require("dotenv").config();


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


const PORT = process.env.PORT || 3000;

app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

CHAPTER 3: IMPLEMENTING USER AUTHENTICATION

3.1 Creating a User Model with Roles

For simplicity, we use an **in-memory user database** instead of a real database.

```
const users = [
  { id: 1, username: "admin", password: "$2a$10$Wz7jR", role: "admin" },
  { id: 2, username: "editor", password: "$2a$10$XyZp8", role: "editor" },
  { id: 3, username: "viewer", password: "$2a$10$AbCd9", role: "viewer" }
];
```

3.2 User Login & JWT Token Generation

Create a **login route** that authenticates users and assigns a JWT token with their role.

```
const jwt = require("jsonwebtoken");
const bcrypt = require("bcryptjs");

app.post("/login", async (req, res) => {
  const { username, password } = req.body;
  const user = users.find((u) => u.username === username);
```

```
  if (!user || !(await bcrypt.compare(password, user.password))) {
    return res.status(401).json({ error: "Invalid credentials" });
  }

  const token = jwt.sign({ id: user.id, role: user.role },
process.env.JWT_SECRET, { expiresIn: "1h" });
  res.json({ token });
});
```

✅ **Checks user credentials.**
✅ **Generates a JWT token** with the user's **role**.
✅ **Returns the token** to be used for authentication in protected routes.

CHAPTER 4: PROTECTING ROUTES WITH MIDDLEWARE

4.1 Creating Authentication Middleware

To protect routes, create a middleware that verifies the **JWT token**.

```
const authenticate = (req, res, next) => {
  const token = req.header("Authorization");
  if (!token) return res.status(401).json({ error: "Access denied. No
token provided." });

  try {
```

```
    const decoded = jwt.verify(token.split(" ")[1],
process.env.JWT_SECRET);

    req.user = decoded;

    next();

  } catch (err) {

    res.status(400).json({ error: "Invalid token" });

  }

};
```

✅ **Extracts the token from request headers.**
✅ **Verifies the JWT token.**
✅ **Attaches the decoded user data to the request object.**

CHAPTER 5: IMPLEMENTING ROLE-BASED ACCESS CONTROL (RBAC)

5.1 Creating an RBAC Middleware

This middleware **checks if the user has the required role**.

```
const authorize = (roles) => {

  return (req, res, next) => {

    if (!roles.includes(req.user.role)) {

      return res.status(403).json({ error: "Access denied. Insufficient
permissions." });

    }

    next();
```

```
  };
};
```

✅ **Checks if the user's role is in the allowed roles list.**
✅ **Blocks unauthorized users with a 403 Forbidden error.**

CHAPTER 6: APPLYING RBAC TO API ROUTES

6.1 Defining API Endpoints with Role Restrictions

```
app.get("/admin", authenticate, authorize(["admin"]), (req, res) => {
  res.json({ message: "Welcome Admin! You have full access." });
});

app.get("/editor", authenticate, authorize(["admin", "editor"]), (req, res) => {
  res.json({ message: "Welcome Editor! You can edit content." });
});

app.get("/viewer", authenticate, authorize(["admin", "editor", "viewer"]), (req, res) => {
  res.json({ message: "Welcome Viewer! You can view content." });
});
```

6.2 How the RBAC System Works

✅ **Admin users can access all routes.**
✅ **Editors can access /editor and /viewer.**
✅ **Viewers can only access /viewer.**

CHAPTER 7: TESTING ROLE-BASED AUTHENTICATION

7.1 Using Postman for API Testing

1. **Login with different users** (/login) and retrieve a token.
2. **Make requests to protected routes** (/admin, /editor, /viewer).
3. **Include the token** in the request headers:
4. Authorization: Bearer <your-token>
5. **Check access permissions** based on the role.

Chapter 8: Handling Role-Based Access Errors

8.1 Common Error Scenarios & Solutions

Error	Possible Cause	Solution
401 Unauthorized	No token provided	Ensure the user includes the token in the request header
403 Forbidden	Insufficient role permissions	Verify the user's assigned role
400 Bad Request	Invalid token format	Ensure the token is correctly structured (Bearer <token>)

Chapter 9: Case Study – Securing a SaaS Platform with RBAC

Scenario:

A SaaS company providing an **online learning platform** needed to implement **role-based access control** for:

- **Admins**: Can manage users, create courses.
- **Instructors**: Can create and manage their courses.
- **Students**: Can only enroll in courses.

Solution:

✅ Used JWT for **user authentication**.
✅ Implemented RBAC with authorize(["admin", "instructor"]) for course management.
✅ Restricted student access to **only viewing courses**.

Outcome:
🚀 **Improved security by preventing unauthorized modifications.**
🚀 **Simplified role assignments with minimal admin effort.**

Chapter 10: Exercises on Role-Based Access Control

Exercise 1: Add a New Role (Moderator)

1. Modify the RBAC middleware to include a moderator role.
2. Create an /moderator route accessible only to admins and moderators.

Exercise 2: Implement Role-Based Content Creation

1. Modify /editor to allow only **editors** to create new content.
2. Implement a **PUT request** to update content based on roles.

CONCLUSION

Key Takeaways

✅ **RBAC ensures restricted access based on user roles.**
✅ **JWT authentication secures API endpoints.**
✅ **Middleware functions simplify authentication and authorization.**
✅ **Proper error handling improves API security.**

🚀 **Next Steps:**

- Implement **Refresh Tokens** for extended sessions.
- Use **MongoDB/PostgreSQL** to store user roles dynamically.
- Learn **OAuth2 and OpenID Connect** for advanced authentication!

HANDS-ON PRACTICE: BUILD A REST API FOR A BLOG APPLICATION USING EXPRESS.JS & MONGODB

OBJECTIVE

In this hands-on practice, you will build a **RESTful API** for a **Blog Application** using **Express.js** and **MongoDB**. The API will allow users to:

- ✅ Create new blog posts
- ✅ Retrieve all posts or a single post
- ✅ Update blog posts
- ✅ Delete posts

CHAPTER 1: SETTING UP THE DEVELOPMENT ENVIRONMENT

1.1 Install Required Software

Ensure that the following are installed:

- **Node.js & npm** – Download here
- **MongoDB (Locally or MongoDB Atlas)** – Download MongoDB

1.2 Initialize a New Node.js Project

Create a project folder and initialize npm:

```
mkdir blog-api

cd blog-api

npm init -y
```

✅ This creates a package.json file for dependency management.

CHAPTER 2: INSTALL DEPENDENCIES

2.1 Install Express.js & Mongoose

npm install express mongoose dotenv cors morgan body-parser

✅ **Dependencies Explained:**

- **express** – Web framework for handling API routes.
- **mongoose** – ODM (Object Data Modeling) for MongoDB.
- **dotenv** – Manages environment variables.
- **cors** – Enables Cross-Origin Resource Sharing.
- **morgan** – Logs incoming API requests.
- **body-parser** – Parses incoming request bodies.

CHAPTER 3: SETTING UP AN EXPRESS SERVER

3.1 Create server.js

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

const mongoose = require("mongoose");

const dotenv = require("dotenv");

const cors = require("cors");

const morgan = require("morgan");


// Load environment variables
```

```
dotenv.config();


const app = express();
const PORT = process.env.PORT || 5000;

// Middleware
app.use(express.json()); // Parse JSON requests
app.use(cors()); // Enable CORS for cross-origin requests
app.use(morgan("dev")); // Log requests

// Default route
app.get("/", (req, res) => {
  res.send("Welcome to the Blog API!");
});

// Start the server
app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

✅ **Run the server:**

```
node server.js
```

You should see:

Server running on http://localhost:5000

✅ Open http://localhost:5000/ in a browser to test the setup.

CHAPTER 4: CONNECT MONGODB TO EXPRESS.JS

4.1 Set Up a MongoDB Connection in .env

Create a .env file:

MONGO_URI=mongodb://localhost:27017/blogdb

(Use MongoDB Atlas if you prefer a cloud database.)

4.2 Modify server.js to Connect MongoDB

```
// Connect to MongoDB
mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log("MongoDB Connected"))
.catch(err => console.error("MongoDB connection error:", err));
```

✅ Restart the server to apply changes.

CHAPTER 5: DEFINE THE BLOG POST MODEL

5.1 Create a models/Blog.js File

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


const BlogSchema = new mongoose.Schema({
  title: { type: String, required: true },
  content: { type: String, required: true },
  author: { type: String, required: true },
  createdAt: { type: Date, default: Date.now }
});


module.exports = mongoose.model("Blog", BlogSchema);
```

✅ **The schema defines a blog post with title, content, author, and creation date.**

CHAPTER 6: IMPLEMENT CRUD OPERATIONS FOR BLOG POSTS

6.1 Create Routes for the Blog API

Create routes/blogRoutes.js:

```
const express = require("express");
const Blog = require("../models/Blog");


const router = express.Router();
```

```
// 1. Create a new blog post (POST /api/blogs)
router.post("/", async (req, res) => {
  try {
    const newBlog = new Blog(req.body);
    await newBlog.save();
    res.status(201).json(newBlog);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});

// 2. Get all blog posts (GET /api/blogs)
router.get("/", async (req, res) => {
  try {
    const blogs = await Blog.find();
    res.json(blogs);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});
```

```
// 3. Get a single blog post by ID (GET /api/blogs/:id)
router.get("/:id", async (req, res) => {
  try {
    const blog = await Blog.findById(req.params.id);
    if (!blog) return res.status(404).json({ message: "Blog not found" });
    res.json(blog);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});

// 4. Update a blog post (PUT /api/blogs/:id)
router.put("/:id", async (req, res) => {
  try {
    const updatedBlog = await Blog.findByIdAndUpdate(req.params.id, req.body, { new: true });
    res.json(updatedBlog);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});
```

```
// 5. Delete a blog post (DELETE /api/blogs/:id)
router.delete("/:id", async (req, res) => {
  try {
    await Blog.findByIdAndDelete(req.params.id);
    res.json({ message: "Blog deleted successfully" });
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});

module.exports = router;
```

✅ **Handles all CRUD operations for blog posts.**

CHAPTER 7: REGISTER ROUTES IN SERVER.JS

Modify server.js to use the new route:

```
const blogRoutes = require("./routes/blogRoutes");

// Use blog routes
app.use("/api/blogs", blogRoutes);
```

✅ **Now, API requests will be handled at /api/blogs.**

Chapter 8: Testing the API with Postman

8.1 Test API Endpoints

Method	Endpoint	Description
POST	/api/blogs	Create a blog post
GET	/api/blogs	Retrieve all blog posts
GET	/api/blogs/:id	Retrieve a single blog post
PUT	/api/blogs/:id	Update a blog post
DELETE	/api/blogs/:id	Delete a blog post

Use **Postman** or **cURL** to test API requests:

Create a blog post:

curl -X POST http://localhost:5000/api/blogs -H "Content-Type: application/json" -d '{"title": "My First Blog", "content": "Hello World!", "author": "John Doe"}'

Retrieve all blog posts:

curl http://localhost:5000/api/blogs

Retrieve a single blog post:

curl http://localhost:5000/api/blogs/{id}

Chapter 9: Deploying the Blog API

9.1 Deploy to Heroku

heroku login

heroku create blog-api-app

git push heroku main

heroku open

✅ **Your API is now live!** 🎉

Chapter 10: Assignments

1. **Add a Comment System** – Create a comments schema and associate it with blogs.
2. **Implement User Authentication** – Add JWT-based authentication for blog access.
3. **Enhance the API with Pagination & Search** – Improve /api/blogs with filters.

Conclusion

You have successfully built a **RESTful Blog API** using Express.js and MongoDB! 🚀

✅ **Next Steps:**

- Connect the API to a **React frontend**.
- Implement **file uploads** for blog images.
- Add **email notifications** for new blog posts!

Happy Coding! 🎉 🚀

HANDS-ON PRACTICE: IMPLEMENT JWT AUTHENTICATION IN NODE.JS & EXPRESS.JS

CHAPTER 1: INTRODUCTION TO JWT AUTHENTICATION

1.1 What is JWT (JSON Web Token)?

JWT (JSON Web Token) is a **compact and secure way** to transmit user authentication data between a client and server. It is **stateless**, meaning authentication is managed **without storing session data on the server**.

1.2 Why Use JWT for Authentication?

✅ **Stateless** – No need to store session data on the server.
✅ **Secure** – Uses **digital signatures** (HMAC or RSA) to verify tokens.
✅ **Compact** – Encoded as a small **Base64 string**, making it efficient.
✅ **Cross-platform** – Works on **React, Angular, Mobile Apps, and APIs**.

CHAPTER 2: SETTING UP JWT AUTHENTICATION IN NODE.JS & EXPRESS.JS

2.1 Prerequisites

- Install **Node.js** (Download from here)
- Install **Postman** for testing API requests
- Basic knowledge of **Express.js, bcrypt.js, and REST APIs**

2.2 Install Required Dependencies

npm init -y # Initialize package.json

npm install express jsonwebtoken bcrypt cors dotenv body-parser

- express – Web framework for handling API requests.
- jsonwebtoken – Generates and verifies JWT tokens.
- bcrypt – Hashes passwords securely.
- cors – Allows cross-origin requests.
- dotenv – Loads environment variables from .env file.
- body-parser – Parses JSON request bodies.

CHAPTER 3: SETTING UP EXPRESS.JS SERVER

3.1 Create server.js and Configure Express.js

Create a new file server.js and add the following:

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

const jwt = require("jsonwebtoken");

const bcrypt = require("bcrypt");

const cors = require("cors");

require("dotenv").config();


const app = express();

app.use(express.json());

app.use(cors());
```

```
// Secret key for JWT (Store this securely)
const SECRET_KEY = process.env.JWT_SECRET || "supersecretkey";

// Temporary in-memory user database
const users = [];

const PORT = 5000;
app.listen(PORT, () => console.log(`Server running on http://localhost:${PORT}`));
```

✅ **The Express server is now set up and ready to use!**

CHAPTER 4: USER REGISTRATION WITH JWT TOKEN GENERATION

4.1 Hash Password and Store User in Database

Create a **registration endpoint** that:

- **Hashes the password** using bcrypt.js
- **Stores user details** in an array
- **Generates a JWT token** for authentication

```
app.post("/register", async (req, res) => {
  try {
    const { username, password } = req.body;
```

```
    // Check if user already exists
    const existingUser = users.find(u => u.username === username);
    if (existingUser) return res.status(400).json({ message: "User already exists" });

    // Hash password before saving
    const hashedPassword = await bcrypt.hash(password, 10);
    const newUser = { username, password: hashedPassword };
    users.push(newUser);

    // Generate JWT token
    const token = jwt.sign({ username }, SECRET_KEY, { expiresIn: "1h" });

    res.status(201).json({ message: "User registered successfully!", token });
  } catch (error) {
    res.status(500).json({ error: "Registration failed" });
  }
});
```

✅ **Uses bcrypt.hash() to securely store passwords.**
✅ **Generates a JWT token upon successful registration.**

💡 **Test with Postman:**

- **Method:** POST
- **URL:** http://localhost:5000/register
- **Body (JSON):**

```
{
  "username": "johndoe",
  "password": "mypassword"
}
```

- **Response:**

```
{
  "message": "User registered successfully!",
  "token": "eyJhbGciOi..."
}
```

CHAPTER 5: USER LOGIN & JWT TOKEN AUTHENTICATION

5.1 Verify Password and Return JWT Token

Create a **login endpoint** that:

- **Finds user by username**
- **Compares the password** using bcrypt.js

- **Generates and returns a JWT token**

```
app.post("/login", async (req, res) => {
  try {
    const { username, password } = req.body;

    // Find user
    const user = users.find(u => u.username === username);
    if (!user) return res.status(404).json({ error: "User not found" });

    // Compare hashed password
    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) return res.status(401).json({ error: "Invalid credentials" });

    // Generate JWT token
    const token = jwt.sign({ username }, SECRET_KEY, { expiresIn: "1h" });

    res.json({ message: "Login successful!", token });
  } catch (error) {
    res.status(500).json({ error: "Login failed" });
```

```
  }
});
```

✅ **Uses bcrypt.compare() to validate the password.**
✅ **Returns a JWT token for future authentication.**

💡 **Test with Postman:**

- **Method:** POST
- **URL:** http://localhost:5000/login
- **Body (JSON):**

```
{
  "username": "johndoe",
  "password": "mypassword"
}
```

- **Response:**

```
{
  "message": "Login successful!",
  "token": "eyJhbGciOi..."
}
```

CHAPTER 6: PROTECTING ROUTES WITH JWT MIDDLEWARE

6.1 Create Middleware to Verify JWT Tokens

Middleware checks if the **request contains a valid JWT token**.

```
const authenticateToken = (req, res, next) => {
  const token = req.headers["authorization"];
  if (!token) return res.status(403).json({ error: "Access denied. No token provided." });

  jwt.verify(token.split(" ")[1], SECRET_KEY, (err, user) => {
    if (err) return res.status(403).json({ error: "Invalid token" });
    req.user = user;
    next();
  });
};
```

✅ **Extracts the token from Authorization header.**
✅ **Verifies the token using jwt.verify().**
✅ **Calls next() to proceed to the requested route.**

6.2 Create a Protected Route (/dashboard)

Only users with a **valid JWT token** can access this route.

```
app.get("/dashboard", authenticateToken, (req, res) => {
  res.json({ message: `Welcome, ${req.user.username}! This is your dashboard.` });
});
```

✅ **Uses authenticateToken middleware to restrict access.**

💡 **Test with Postman:**

- **Method:** GET
- **URL:** http://localhost:5000/dashboard
- **Headers:**
- Authorization: Bearer <your-jwt-token>
- **Response:**

```
{
  "message": "Welcome, johndoe! This is your dashboard."
}
```

🔹 If the token is missing or invalid, you get:

```
{
  "error": "Access denied. No token provided."
}
```

CHAPTER 7: BEST PRACTICES FOR JWT AUTHENTICATION

✅ **Use Environment Variables** – Store the JWT secret key securely in .env.

✅ **Set Token Expiry** – Use **short expiry times (e.g., 1 hour)** to reduce risks.

✅ **Implement Refresh Tokens** – Generate a new token without re-authentication.

✅ **Use HTTPS in Production** – Prevent token interception.

✅ **Use Strong Password Hashing** – Always hash passwords before storing them.

Chapter 8: Hands-on Exercises

Exercise 1: Implement Role-Based Authentication

✅ Modify JWT payload to include role: "admin" and restrict access to certain routes.

Exercise 2: Implement Token Refresh Mechanism

✅ Create an endpoint /refresh-token to issue a new JWT token when the old one expires.

Exercise 3: Store Users in MongoDB

✅ Replace the **in-memory users array** with **MongoDB for persistent storage**.

Conclusion

✅ **JWT provides a stateless, secure authentication mechanism.**
✅ **bcrypt.js ensures passwords are hashed before storage.**
✅ **Middleware (authenticateToken) protects routes from unauthorized access.**
✅ **Implementing JWT authentication makes APIs scalable & secure.**

🚀 **Next Steps:**

- Integrate JWT with **React.js or Angular** for a **full-stack authentication system**.

- Implement **OAuth (Google/Facebook login)**.
- Explore **session-based authentication for comparison**.

By completing this **hands-on practice,** you are now ready to **implement secure authentication** in real-world applications! 🎉 🚀

ASSIGNMENTS:

- DEVELOP A SECURE USER AUTHENTICATION SYSTEM USING JWT
- BUILD A CRUD API WITH EXPRESS & MONGODB

Assignment Solution: Develop a Secure User Authentication System Using JWT in Express.js

Objective

The goal of this assignment is to **implement secure user authentication** in a RESTful API using **JSON Web Tokens (JWT)** in an **Express.js** application. The authentication system will include:

✅ **User Registration** – Create an account with hashed passwords.
✅ **User Login** – Validate credentials and return a JWT token.
✅ **JWT Authentication Middleware** – Protect routes from unauthorized access.
✅ **Token Expiry & Verification** – Ensure session security.

Step 1: Setup Express Project

1.1 Install Required Dependencies

Create a new project and install dependencies:

mkdir jwt-auth

cd jwt-auth

npm init -y

Install required npm packages:

npm install express mongoose dotenv bcryptjs jsonwebtoken cors body-parser

✅ **Dependencies Explained:**

- **express** – Web framework for API handling.
- **mongoose** – MongoDB ORM for user storage.
- **dotenv** – Loads environment variables securely.
- **bcryptjs** – Hashes passwords for secure storage.
- **jsonwebtoken (JWT)** – Generates and verifies authentication tokens.
- **cors** – Allows cross-origin API requests.
- **body-parser** – Parses incoming request bodies.

Step 2: Setup Express Server

Create a file named **server.js** and initialize Express:

```
const express = require("express");
const dotenv = require("dotenv");
const cors = require("cors");
const mongoose = require("mongoose");

// Load environment variables
dotenv.config();

const app = express();
const PORT = process.env.PORT || 5000;
```

```
// Middleware
app.use(express.json()); // Parses JSON requests
app.use(cors()); // Enable CORS for API calls

// Default route
app.get("/", (req, res) => {
  res.send("JWT Authentication API is running!");
});

// Connect to MongoDB
mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => console.log("MongoDB Connected"))
.catch(err => console.error("MongoDB connection error:", err));

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

✅ **Run the server:**

node server.js

Your API should now be running at http://localhost:5000/ 🎉

Step 3: Create the User Model

Create a folder named **models/** and inside it, create **User.js** to define the user schema:

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


const UserSchema = new mongoose.Schema({
  username: { type: String, required: true, unique: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true }
}, { timestamps: true });


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

✅ **Fields Explained:**

- **username** – Unique identifier for users.
- **email** – Required for login verification.
- **password** – Stored securely in hashed format.
- **timestamps** – Auto-generates createdAt and updatedAt.

Step 4: Implement User Registration with Password Hashing

Create a folder **routes/** and inside it, create **authRoutes.js**:

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

const bcrypt = require("bcryptjs");

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


const router = express.Router();


// Register a new user

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

  try {

    const { username, email, password } = req.body;


    // Check if user already exists

    const existingUser = await User.findOne({ email });

    if (existingUser) return res.status(400).json({ message: "User already exists" });


    // Hash password before saving

    const salt = await bcrypt.genSalt(10);

    const hashedPassword = await bcrypt.hash(password, salt);
```

```
    const newUser = new User({ username, email, password: hashedPassword });
    await newUser.save();

    res.status(201).json({ message: "User registered successfully!" });
  } catch (error) {
    res.status(500).json({ message: "Internal Server Error" });
  }
});

module.exports = router;
```

✅ **Registration Flow:**

1. Checks if user already exists.
2. Hashes password using **bcrypt.js**.
3. Saves the new user in MongoDB.

Step 5: Implement User Login and JWT Token Generation

Modify **authRoutes.js** to include login functionality:

```
const jwt = require("jsonwebtoken");
```

```
// User login
router.post("/login", async (req, res) => {
  try {
    const { email, password } = req.body;

    // Check if user exists
    const user = await User.findOne({ email });
    if (!user) return res.status(400).json({ message: "Invalid credentials" });

    // Validate password
    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) return res.status(400).json({ message: "Invalid credentials" });

    // Generate JWT Token
    const token = jwt.sign({ userId: user._id }, process.env.JWT_SECRET, { expiresIn: "1h" });

    res.json({ message: "Login successful", token });
  } catch (error) {
    res.status(500).json({ message: "Internal Server Error" });
```

```
  }
});
```

✅ **Login Flow:**

1. Verifies user existence.
2. Compares password using **bcrypt**.
3. Generates a **JWT token** with an expiry time.

Step 6: Protect Routes with JWT Authentication Middleware

Create a new folder **middleware/** and inside it, create **authMiddleware.js**:

```
const jwt = require("jsonwebtoken");


const verifyToken = (req, res, next) => {
  const token = req.header("Authorization");
  if (!token) return res.status(403).json({ message: "Access Denied" });

  try {
    const verified = jwt.verify(token.split(" ")[1], process.env.JWT_SECRET);
    req.user = verified;
    next();
```

```
  } catch (error) {
    res.status(401).json({ message: "Invalid Token" });
  }
};

module.exports = verifyToken;
```

✅ **Middleware Logic:**

- Extracts token from Authorization header.
- Verifies token using jwt.verify().
- Grants access if the token is valid.

Step 7: Protect an API Route

Modify server.js to use authentication middleware:

```
const authRoutes = require("./routes/authRoutes");
const verifyToken = require("./middleware/authMiddleware");

app.use("/api/auth", authRoutes);

// Protected route example
app.get("/api/protected", verifyToken, (req, res) => {
  res.json({ message: "This is a protected route", user: req.user });
```

```
});
```

✅ **Only authenticated users with a valid JWT can access /api/protected.**

Step 8: Test the API using Postman

8.1 Register a New User

Endpoint:

POST /api/auth/register

Request Body (JSON):

```
{
  "username": "JohnDoe",
  "email": "john@example.com",
  "password": "securepass"
}
```

8.2 Log In and Receive JWT Token

Endpoint:

POST /api/auth/login

Request Body (JSON):

```
{
  "email": "john@example.com",
  "password": "securepass"
```

}

✅ **Response:**

{

"message": "Login successful",

"token": "eyJhbGciOiJIUz..."

}

8.3 Access a Protected Route

Endpoint:

GET /api/protected

Headers:

Authorization: Bearer <your_jwt_token>

CONCLUSION

🎉 You have successfully built a **JWT-based Authentication System** with:

✅ **Secure password hashing**
✅ **JWT token generation & verification**
✅ **Middleware for protecting API routes**

🚀 **Next Steps:**

- Implement **role-based authentication** (Admin/User).
- Store JWT in **HttpOnly cookies** for better security.
- Use **Refresh Tokens** for extended sessions.

Happy Coding! 🎉 🚀

Assignment Solution & Step-by-Step Guide: Build a CRUD API with Express & MongoDB

Objective

In this assignment, we will build a **CRUD API (Create, Read, Update, Delete)** using **Express.js and MongoDB**. The API will allow users to manage a collection of **blog posts**.

Step 1: Set Up the Development Environment

1.1 Install Node.js and npm

If you haven't installed **Node.js,** download it from:
🔗 https://nodejs.org/

Verify installation by running:

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

✅ If both commands return version numbers, Node.js and npm are installed.

1.2 Create a New Project Folder

```
mkdir express-mongo-crud
cd express-mongo-crud
```

Initialize a new Node.js project:

```
npm init -y
```

This creates a package.json file to manage dependencies.

Step 2: Install Required Packages

Run the following command to install necessary dependencies:

npm install express mongoose dotenv cors body-parser

- **express** – Web framework for creating the API.
- **mongoose** – ODM (Object Data Modeling) for MongoDB.
- **dotenv** – Manages environment variables.
- **cors** – Enables cross-origin requests.
- **body-parser** – Parses incoming request bodies.

Step 3: Set Up Express.js Server

3.1 Create server.js File

Inside the project folder, create a file named **server.js** and add the following code:

```
require("dotenv").config();

const express = require("express");

const mongoose = require("mongoose");

const cors = require("cors");


const app = express();
```

```
const PORT = process.env.PORT || 5000;


// Middleware
app.use(cors());
app.use(express.json());


// MongoDB Connection
mongoose.connect(process.env.MONGO_URI, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log("MongoDB Connected"))
  .catch(err => console.error(err));


// Default Route
app.get("/", (req, res) => {
  res.send("Welcome to the CRUD API with Express & MongoDB");
});


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

3.2 Create .env File for Environment Variables

Inside the project folder, create a **.env** file and add your MongoDB connection string:

MONGO_URI=mongodb://localhost:27017/blogdb

PORT=5000

✅ **The server is now set up and connected to MongoDB!**

Step 4: Define the Blog Model

Create a new folder **models/** and inside it, create **Blog.js**:

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


const BlogSchema = new mongoose.Schema({
  title: { type: String, required: true },
  content: { type: String, required: true },
  author: { type: String, required: true },
  createdAt: { type: Date, default: Date.now }
});


module.exports = mongoose.model("Blog", BlogSchema);
```

✅ **Defines a schema for blog posts with fields: title, content, author, and createdAt.**

Step 5: Implement CRUD Operations

Create a new folder **routes/** and inside it, create **blogRoutes.js**:

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

const Blog = require("../models/Blog");


const router = express.Router();


// Create a new blog post (POST)

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

  try {

    const newBlog = new Blog(req.body);

    await newBlog.save();

    res.status(201).json(newBlog);

  } catch (err) {

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

  }

});


// Get all blog posts (GET)

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

  try {
```

```
    const blogs = await Blog.find();
    res.status(200).json(blogs);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});

// Get a single blog post by ID (GET)
router.get("/:id", async (req, res) => {
  try {
    const blog = await Blog.findById(req.params.id);
    if (!blog) return res.status(404).json({ error: "Blog not found" });
    res.status(200).json(blog);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});

// Update a blog post (PUT)
router.put("/:id", async (req, res) => {
  try {
```

```
    const updatedBlog = await Blog.findByIdAndUpdate(req.params.id, req.body, { new: true });
    res.status(200).json(updatedBlog);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});

// Delete a blog post (DELETE)
router.delete("/:id", async (req, res) => {
  try {
    await Blog.findByIdAndDelete(req.params.id);
    res.status(200).json({ message: "Blog deleted successfully" });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});

module.exports = router;
```

✅ **Implements full CRUD (Create, Read, Update, Delete) functionality using Mongoose.**

Step 6: Integrate Routes in server.js

Modify server.js to use the blog routes:

```
const blogRoutes = require("./routes/blogRoutes");

app.use("/api/blogs", blogRoutes);
```

✅ **Now, all blog-related requests will be handled under /api/blogs/.**

Step 7: Start the Server and Test with Postman

7.1 Start the Server

```
node server.js
```

or

```
nodemon server.js
```

7.2 Test API Endpoints

HTTP Method	Endpoint	Description
POST	/api/blogs/	Create a new blog post
GET	/api/blogs/	Get all blog posts
GET	/api/blogs/:id	Get a single blog post by ID
PUT	/api/blogs/:id	Update a blog post
DELETE	/api/blogs/:id	Delete a blog post

Step 8: Deploying the API to Cloud (Bonus)

8.1 Deploy to Heroku

1. Install **Heroku CLI** from https://devcenter.heroku.com/articles/heroku-cli
2. Login to Heroku:
3. heroku login
4. Initialize Git and deploy the app:
5. git init
6. git add .
7. git commit -m "Initial commit"
8. heroku create blog-api
9. git push heroku master

✅ **Your API is now live on Heroku!**

Step 9: Assignments & Enhancements

Assignment 1: Implement User Authentication

- Install jsonwebtoken (npm install jsonwebtoken).
- Secure routes using authentication middleware.

Assignment 2: Implement Comments for Blog Posts

- Create a **Comment schema** linked to blog posts.
- Add **nested routes** to handle comments (/api/blogs/:id/comments).

Assignment 3: Implement Pagination & Search

- Use limit() and skip() in Mongoose to paginate results.
- Implement a **search endpoint** using req.query.

CONCLUSION

✅ **Built a fully functional CRUD API using Node.js, Express, and MongoDB.**
✅ **Implemented database operations using Mongoose.**
✅ **Secured environment variables using dotenv.**
✅ **Tested the API using Postman.**
✅ **Deployed the API to Heroku.**

🚀 **Next Steps:**

- Add **file uploads** (Multer) for blog images.
- Implement **rate limiting** to prevent abuse.
- Create a **frontend (React.js) to consume this API!** 🎉 🔥