Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

BACKEND DEVELOPMENT WITH NODE.JS & EXPRESS.JS (WEEKS 13-18)

UNDERSTANDING JAVASCRIPT RUNTIME & EVENT LOOP

CHAPTER 1: INTRODUCTION TO JAVASCRIPT RUNTIME

1.1 What is the JavaScript Runtime?

The **JavaScript Runtime** is the **environment** where JavaScript code runs. It includes:

✓ **The JavaScript Engine** (e.g., V8 for Chrome, SpiderMonkey for Firefox).

✓ **The Call Stack** (where JavaScript keeps track of function execution).

✓ **The Heap** (memory space for storing variables and objects).

✓ **The Event Loop** (manages asynchronous code execution).

✓ **Web APIs** (browser features like setTimeout, DOM, and Fetch API).

- **Why is Understanding JavaScript Runtime Important?**
  - Helps optimize **code performance**.

- Prevents **blocking the main thread** in the browser.
- Explains **how asynchronous operations (e.g., API calls) work**.

1.2 Components of JavaScript Runtime

JavaScript uses a **single-threaded** model, meaning it executes one command at a time. However, **asynchronous tasks** (API calls, timers) run in the background and return results later.

🔹 **Key Components of JavaScript Runtime:**

Component	Purpose	Example
Call Stack	Tracks function execution	Function calls
Heap	Allocates memory	Storing objects & arrays
Web APIs	Handle async tasks	setTimeout, Fetch API
Event Queue	Holds async callbacks	click, keydown events
Event Loop	Moves tasks to Call Stack	Ensures smooth execution

📌 **Example: JavaScript Execution Flow**

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

setTimeout(() => console.log("Delayed"), 1000);

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

Output:

Start

End

Delayed

✓ **Why is "Delayed" printed last?** The **Event Loop** waits for the timer to finish before executing it.

CHAPTER 2: UNDERSTANDING THE CALL STACK

2.1 What is the Call Stack?

The **Call Stack** is a **data structure** that tracks **function execution order**. JavaScript executes **one function at a time** in a **Last In, First Out (LIFO)** order.

📌 Example: Call Stack Execution

```
function first() {
  console.log("First function");
  second();
}
function second() {
  console.log("Second function");
}
first();
```

🔹 Call Stack Execution Steps:

1️⃣ first() is pushed to the **Call Stack**.
2️⃣ Inside first(), second() is called → **added to the Call Stack**.
3️⃣ second() executes, then **removed from the Call Stack**.
4️⃣ first() finishes execution and **exits the Call Stack**.

✓ **Final Output:**

First function

Second function

2.2 Call Stack Overflow

If a function **calls itself indefinitely,** it leads to a **stack overflow error**.

📌 **Example: Infinite Recursion (Stack Overflow)**

```
function infinite() {
  infinite(); // Calls itself indefinitely
}
infinite();
```

- **Error:** Uncaught RangeError: Maximum call stack size exceeded

✓ **Fix:** Use a **base case** to stop recursion.

CHAPTER 3: UNDERSTANDING THE EVENT LOOP

3.1 What is the Event Loop?

The **Event Loop** is the mechanism that allows JavaScript to **handle asynchronous tasks** efficiently. It:

✓ **Checks the Call Stack** for pending tasks.

✓ **Moves async tasks** from the Event Queue to the Call Stack when it's empty.

✓ Prevents the browser from **freezing while waiting for slow tasks**.

📌 **Example: Understanding the Event Loop**

```
console.log("1");
```

```
setTimeout(() => console.log("2"), 0);

console.log("3");
```

✓ **Output Order:**

```
1

3

2
```

- **Why?**
  - console.log("1") runs **immediately**.
  - setTimeout **sends callback to Web API** and waits in the Event Queue.
  - console.log("3") executes **before** the event loop picks up setTimeout.

3.2 Tasks in the Event Loop: Microtasks vs. Macrotasks

JavaScript schedules asynchronous tasks in **two separate queues**:

Task Type	Example	Execution Priority
Microtasks	Promise.then(), MutationObserver	**Higher**
Macrotasks	setTimeout, setInterval, fetch()	Lower

📌 **Example: Microtasks Execute Before Macrotasks**

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


setTimeout(() => console.log("Timeout"), 0);
```

```
Promise.resolve().then(() => console.log("Promise"));


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

✓ Output Order:

Start

End

Promise

Timeout

- **Why?**
  - Promise.then() (Microtask) executes **before** setTimeout (Macrotask).

CHAPTER 4: HANDLING ASYNCHRONOUS CODE IN JAVASCRIPT

4.1 Callbacks & the Callback Hell Problem

A **callback function** is executed **after an asynchronous task finishes**.

📌 Example: Nested Callbacks (Callback Hell)

```
function fetchData(callback) {
  setTimeout(() => {
    console.log("Data fetched");
    callback();
  }, 1000);
}
```

```
fetchData(() => {
  console.log("Processing data...");
});
```

✓ Nested callbacks **become hard to manage** (Callback Hell).

4.2 Promises: Handling Asynchronous Operations

A **Promise** represents a value **that will be available in the future**.

📌 **Example: Using Promises**

```
function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Data loaded"), 1000);
  });
}

fetchData().then(data => console.log(data));
```

✓ **Removes nesting issues from callbacks**.

4.3 Async/Await: Cleaner Way to Handle Asynchronous Code

📌 **Example: Fetching Data with Async/Await**

```
async function fetchData() {
```

```
  let response = await
fetch("https://jsonplaceholder.typicode.com/posts/1");
  let data = await response.json();
  console.log(data);
}

fetchData();
```

✓ **Waits for data before continuing execution**.
✓ **More readable than Promises**.

Case Study: How Netflix Uses JavaScript Event Loop for Streaming

Challenges Faced by Netflix

✓ Handling **real-time data streaming** efficiently.
✓ Preventing **buffering issues during playback**.

Solutions Implemented

✓ Used **JavaScript's Event Loop** to handle video chunks asynchronously.
✓ Implemented **Microtasks (Promises) for priority tasks** like loading subtitles.
✓ Optimized **Web APIs like Fetch for real-time streaming**.

- **Key Takeaways from Netflix's Event Loop Strategy:**

✓ **Microtasks handle high-priority tasks smoothly**.
✓ **Efficient scheduling prevents video buffering**.
✓ **JavaScript's async model ensures smooth user experience**.

📝 Exercise

✅ Write a **JavaScript function** that logs numbers **1, 2, and 3** but prints 2 **after 1 second**.

✅ Create a **Promise** that resolves after **3 seconds** and logs "Data Loaded" to the console.

✅ Modify a function to **use async/await instead of Promises**.

✅ Use **setTimeout inside a loop** and fix the **unexpected output issue**.

Conclusion

✓ JavaScript is single-threaded but handles async tasks via the Event Loop.

✓ The Call Stack executes synchronous code first.

✓ Microtasks (Promise.then()) execute before Macrotasks (setTimeout).

✓ Async/Await simplifies handling API calls and background tasks.

✓ Understanding the Event Loop helps build scalable web applications.

WORKING WITH FILE SYSTEM & MODULES IN NODE.JS

CHAPTER 1: INTRODUCTION TO FILE SYSTEM & MODULES IN NODE.JS

1.1 What is the File System in Node.js?

The **File System (fs) module** in Node.js provides methods to interact with the **file system**, allowing developers to **read, write, update, and delete files**.

- **Why Use the File System Module?**

✓ Allows **reading/writing files** for data storage.
✓ Enables **logging, configuration, and data management**.
✓ Helps in **handling JSON files for APIs**.

- **Common Operations with fs Module:**

Operation	Method	Example
Read File	fs.readFile()	Read contents of a file
Write File	fs.writeFile()	Create or overwrite a file
Append File	fs.appendFile()	Add new content to a file
Delete File	fs.unlink()	Remove a file
Rename File	fs.rename()	Rename a file

CHAPTER 2: WORKING WITH THE FILE SYSTEM (FS MODULE)

2.1 Reading a File (fs.readFile)

📌 **Example: Reading a Text File Asynchronously**

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


fs.readFile("example.txt", "utf8", (err, data) => {
  if (err) {
    console.error("Error reading file:", err);
  } else {
    console.log("File Content:", data);
  }
});
```

✓ Uses fs.readFile() to **asynchronously read a file**.
✓ "utf8" ensures **text is readable** instead of binary data.

2.2 Writing to a File (fs.writeFile)

📌 Example: Creating a New File & Writing Content

```
fs.writeFile("newFile.txt", "Hello, Node.js!", (err) => {
  if (err) console.error("Error writing file:", err);
  else console.log("File created successfully!");
});
```

✓ Creates newFile.txt if it **does not exist**.
✓ **Overwrites** existing content if the file exists.

2.3 Appending to a File (fs.appendFile)

📌 Example: Adding More Content to a File

```
fs.appendFile("newFile.txt", "\nThis is additional content.", (err) => {
  if (err) console.error("Error appending file:", err);
  else console.log("Content added successfully!");
});
```

✓ Adds **new content** without deleting previous data.

2.4 Deleting a File (fs.unlink)

📌 **Example: Removing a File**

```
fs.unlink("newFile.txt", (err) => {
  if (err) console.error("Error deleting file:", err);
  else console.log("File deleted successfully!");
});
```

✓ Permanently **removes the file from the system**.

2.5 Renaming a File (fs.rename)

📌 **Example: Changing File Name**

```
fs.rename("oldFile.txt", "renamedFile.txt", (err) => {
  if (err) console.error("Error renaming file:", err);
  else console.log("File renamed successfully!");
});
```

✓ Changes oldFile.txt to renamedFile.txt.

CHAPTER 3: WORKING WITH DIRECTORIES (FOLDERS)

3.1 Creating a Directory (fs.mkdir)

📌 Example: Creating a New Folder

```
fs.mkdir("newFolder", (err) => {
  if (err) console.error("Error creating folder:", err);
  else console.log("Folder created successfully!");
});
```

✓ Creates newFolder in the **current directory**.

3.2 Checking if a File Exists (fs.existsSync)

📌 Example: Verifying a File Before Reading

```
if (fs.existsSync("example.txt")) {
  console.log("File exists!");
} else {
  console.log("File does not exist.");
}
```

✓ Ensures a file exists before performing **read/write operations**.

CHAPTER 4: UNDERSTANDING NODE.JS MODULES

4.1 What Are Modules in Node.js?

A **module** is a self-contained **block of reusable code** that can be imported into other files.

- **Types of Node.js Modules:**

Module Type	Description	Example
Built-in Modules	Provided by Node.js	fs, http, path
Custom Modules	User-defined modules	require('./myModule')
Third-Party Modules	Installed via npm	express, axios, lodash

4.2 Importing Built-in Modules (require())

📌 Example: Using the path Module

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


console.log("Directory Name:", path.dirname(__filename));
console.log("File Extension:", path.extname(__filename));
console.log("File Name:", path.basename(__filename));
```

✓ Extracts **file paths, names, and extensions** dynamically.

4.3 Creating and Exporting a Custom Module

📌 Step 1: Create math.js (Custom Module)

```
function add(a, b) {
  return a + b;
}


module.exports = { add };
```

✓ Defines an add() function and **exports** it.

📌 **Step 2: Import and Use math.js in Another File**

```
const math = require("./math");


console.log(math.add(5, 3)); // Output: 8
```

✓ Imports and **uses the add() function** from math.js.

CHAPTER 5: INSTALLING & USING THIRD-PARTY MODULES

5.1 Installing Packages with npm

📌 **Example: Installing lodash (Utility Library)**

```
npm install lodash
```

✓ Installs lodash package locally.

📌 **Example: Using Lodash in a Node.js File**

```
const _ = require("lodash");


const numbers = [10, 5, 8, 2];
console.log("Sorted:", _.sortBy(numbers));
```

✓ Uses _.sortBy() to **sort an array**.

Case Study: How Netflix Uses File System & Modules in Node.js

Challenges Faced by Netflix

✓ Storing **user watch history** and logs efficiently.
✓ Managing **large-scale file uploads for media streaming**.

Solutions Implemented

✓ Used **Node.js fs module** for logging and caching.
✓ Modularized **code using Node.js modules** for better scalability.
✓ Integrated **third-party libraries** (like multer) for file uploads.

🔹 **Key Takeaways from Netflix's Strategy:**
✓ **File system operations are crucial for data storage.**
✓ **Modules keep code organized and reusable.**
✓ **Third-party libraries enhance Node.js functionality.**

📝 **Exercise**

✅ Create a new **text file** and write "Hello, World!" into it using fs.writeFile().
✅ Create a **custom module** that exports a function to **multiply two numbers**.
✅ Use the **path module** to get the **file extension** of index.js.
✅ Install and use **Lodash** to sort an array of numbers.

Conclusion

✓ **Node.js fs module allows reading, writing, updating, and deleting files.**
✓ **Directories can be managed using fs.mkdir() and fs.existsSync().**
✓ **Modules in Node.js make code reusable and maintainable.**
✓ **Third-party libraries (via npm) extend the functionality of Node.js applications.**

ASYNCHRONOUS PROGRAMMING IN NODE.JS

CHAPTER 1: INTRODUCTION TO ASYNCHRONOUS PROGRAMMING IN NODE.JS

1.1 What is Asynchronous Programming?

Asynchronous programming is a **non-blocking programming model** that allows multiple operations to execute **without waiting for previous tasks to complete**.

🔹 **Why is Asynchronous Programming Important in Node.js?**

✓ Improves **performance** by handling multiple requests at once.
✓ Prevents **blocking of code execution** when performing I/O operations.
✓ Enables **faster response times** in web applications.

🔹 **Synchronous vs. Asynchronous Execution**

📌 **Synchronous Code (Blocking)**

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

const result = readFileSync("file.txt", "utf8"); // Blocks execution

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

📌 **Asynchronous Code (Non-Blocking)**

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

fs.readFile("file.txt", "utf8", (err, data) => {

  console.log("File Read");

});
```

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

✓ In asynchronous code, **execution continues** while the file is being read.

CHAPTER 2: CALLBACKS IN NODE.JS

2.1 What are Callbacks?

A **callback function** is a function passed as an argument **to another function** to be executed later.

- **Example: Asynchronous File Reading with Callbacks**

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


fs.readFile("file.txt", "utf8", (err, data) => {
  if (err) console.error(err);
  else console.log(data);
});
console.log("Reading file...");
```

✓ fs.readFile() **reads the file asynchronously,** executing the callback function once done.
✓ **"Reading file..."** logs **before the file content** because the function is non-blocking.

2.2 Callback Hell (The Pyramid of Doom)

When multiple callbacks are **nested inside each other,** it leads to **callback hell,** making the code difficult to read and debug.

📌 Example: Callback Hell in Nested Async Functions

```
fs.readFile("file1.txt", "utf8", (err, data1) => {
  fs.readFile("file2.txt", "utf8", (err, data2) => {
    fs.readFile("file3.txt", "utf8", (err, data3) => {
      console.log(data1, data2, data3);
    });
  });
});
```

✓ **The nested structure makes debugging difficult.**
✓ **Solution:** Use **Promises or Async/Await** to write cleaner code.

CHAPTER 3: PROMISES IN NODE.JS

3.1 What are Promises?

A **Promise** represents a value that **will be available in the future**. It replaces callbacks for better readability and error handling.

- **Promise States:**

State	Description
Pending	Initial state, not resolved or rejected.
Fulfilled	Operation completed successfully.
Rejected	Operation failed.

3.2 Creating and Using Promises

📌 Example: Creating a Simple Promise

```
const myPromise = new Promise((resolve, reject) => {
  let success = true;
  setTimeout(() => {
    if (success) resolve("Task Completed!");
    else reject("Task Failed!");
  }, 2000);
});

myPromise
  .then(result => console.log(result))  // If resolved
  .catch(error => console.log(error));  // If rejected
```

✓ resolve("Task Completed!") executes .then().
✓ reject("Task Failed!") executes .catch().

3.3 Using Promises with File System Operations

📌 Example: Reading a File with Promises

```
const fs = require("fs").promises;

fs.readFile("file.txt", "utf8")
  .then(data => console.log(data))
  .catch(err => console.error("Error:", err));
```

✓ fs.readFile() returns a **Promise**, eliminating callback nesting.

CHAPTER 4: ASYNC/AWAIT IN NODE.JS

4.1 What is Async/Await?

async and await provide a cleaner way to work with **asynchronous code** than Promises or callbacks.

🔹 **Why Use Async/Await?**

✓ Improves **code readability** by avoiding .then() chaining.

✓ Handles **asynchronous errors** easily with try...catch.

4.2 Converting a Promise-Based Function to Async/Await

📌 **Example: Using Async/Await with File Reading**

```
const fs = require("fs").promises;


async function readFileAsync() {
  try {
    const data = await fs.readFile("file.txt", "utf8");
    console.log(data);
  } catch (err) {
    console.error("Error:", err);
  }
}


readFileAsync();
```

✓ await **pauses execution** until the file is read.

✓ try...catch handles **errors gracefully**.

4.3 Fetching API Data Using Async/Await

📌 **Example: Fetching Data from an API**

```
const fetch = require("node-fetch");


async function fetchData() {
  try {
    const response = await
fetch("https://jsonplaceholder.typicode.com/posts/1");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("API Error:", error);
  }
}

fetchData();
```

✓ await fetch(url) waits for the API response **without blocking execution**.

CHAPTER 5: COMBINING CALLBACKS, PROMISES, AND ASYNC/AWAIT

📌 Example: Converting Callback-Based Code to Promises and Async/Await

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

// Callback Version
fs.readFile("file.txt", "utf8", (err, data) => {
  if (err) console.error(err);
  else console.log("Callback:", data);
});

// Promise Version
fs.promises.readFile("file.txt", "utf8")
  .then(data => console.log("Promise:", data))
  .catch(err => console.error(err));

// Async/Await Version
async function readFileAsync() {
  try {
    const data = await fs.promises.readFile("file.txt", "utf8");
    console.log("Async/Await:", data);
  } catch (err) {
    console.error(err);
  }
```

```
}

readFileAsync();
```

✓ Each version performs the same task but with different approaches.

Case Study: How Netflix Uses Asynchronous Programming in Node.js

Challenges Faced by Netflix

- **Handling millions of concurrent requests** for streaming.
- **Ensuring low latency** for seamless video playback.

Solutions Implemented

- Used **Async/Await to handle API calls efficiently**.
- Implemented **non-blocking I/O for streaming videos**.
- Used **event-driven architecture with Node.js** to improve performance.

🔹 **Key Takeaways from Netflix's Strategy:**

✓ Non-blocking architecture improves scalability.

✓ Async functions help optimize real-time API requests.

📝 **Exercise**

✅ Convert a **callback-based file reading function** into a Promise-based function.

✅ Fetch API data using both **Promises and Async/Await**.

✅ Implement a function that **writes data to a file asynchronously**.

✅ Create a function that **performs multiple async operations in sequence**.

Conclusion

✓ Asynchronous programming in Node.js prevents blocking operations.
✓ Callbacks were the original method but lead to callback hell.
✓ Promises simplify async execution with .then() and .catch().
✓ Async/Await provides the cleanest syntax for handling async tasks.

SETTING UP EXPRESS.JS

CHAPTER 1: INTRODUCTION TO EXPRESS.JS

1.1 What is Express.js?

Express.js is a **fast, unopinionated, and minimalist web framework for Node.js**. It simplifies the process of building **server-side applications and REST APIs** by providing built-in middleware, routing, and HTTP utilities.

🔹 **Why Use Express.js?**

✓ **Lightweight and fast** – Minimal setup required.

✓ **Easy routing system** – Handles GET, POST, PUT, DELETE requests.

✓ **Middleware support** – Manages authentication, logging, and security.

✓ **Compatible with databases** – Works with MongoDB, PostgreSQL, MySQL, etc.

🔹 **Common Use Cases:**

- Building **RESTful APIs**.
- Creating **backend services** for web applications.
- Handling **user authentication and authorization**.

CHAPTER 2: INSTALLING AND SETTING UP EXPRESS.JS

2.1 Installing Node.js and npm

📌 **Before installing Express.js, make sure Node.js and npm are installed:**

node -v # Check Node.js version

npm -v # Check npm version

If not installed, download it from Node.js Official Website.

2.2 Creating an Express.js Project

📌 Steps to Set Up an Express Project:

1. Create a new project folder:
2. mkdir express-app && cd express-app
3. Initialize a **Node.js project**:
4. npm init -y

This generates a package.json file.

5. Install Express.js:
6. npm install express

2.3 Creating a Basic Express Server

📌 Example: server.js (Basic Express Server)

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

const app = express();


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

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

});
```

```
const PORT = 3000;

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

✓ express() creates an **Express app instance**.
✓ app.get("/") defines a **route for the homepage**.
✓ app.listen(PORT, callback) starts the **server on port 3000**.

📌 **Run the server:**

```
node server.js
```

- **Open in browser:** http://localhost:3000/

CHAPTER 3: SETTING UP ROUTES IN EXPRESS.JS

3.1 Handling Different HTTP Methods

Express supports **multiple HTTP methods** for handling API requests.

📌 **Example: Basic Routing in Express.js**

```
app.get("/users", (req, res) => {
  res.send("GET request to /users");
});


app.post("/users", (req, res) => {
  res.send("POST request to /users");
```

```
});


app.put("/users/:id", (req, res) => {
  res.send(`PUT request to update user with ID ${req.params.id}`);
});


app.delete("/users/:id", (req, res) => {
  res.send(`DELETE request to remove user with ID ${req.params.id}`);
});
```

✓ GET – Fetch data.
✓ POST – Add new data.
✓ PUT – Update existing data.
✓ DELETE – Remove data.

3.2 Using Route Parameters and Query Strings

📌 **Example: Handling Route Parameters (req.params)**

```
app.get("/product/:id", (req, res) => {
  res.send(`Product ID: ${req.params.id}`);
});
```

- **Request:** GET /product/123
- **Response:** Product ID: 123

📌 **Example: Handling Query Strings (req.query)**

```
app.get("/search", (req, res) => {
  res.send(`Searching for: ${req.query.q}`);
});
```

- **Request:** GET /search?q=laptop
- **Response:** Searching for: laptop

CHAPTER 4: USING MIDDLEWARE IN EXPRESS.JS

4.1 What is Middleware?

Middleware functions **execute during the request-response cycle**. They process requests before sending a response.

- **Types of Middleware in Express.js:**

Middleware Type	Purpose	Example
Built-in	Parses request bodies	express.json()
Third-party	Adds extra functionality	cors, helmet, morgan
Custom	User-defined logic	Logging, validation

4.2 Using Built-in Middleware

📌 **Example: Parsing JSON Requests**

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


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

```
  res.send(`User Created: ${req.body.name}`);
});
```

- **Sends a JSON request:**

```
{ "name": "Alice" }
```

- **Response:** User Created: Alice

4.3 Creating Custom Middleware

📌 **Example: Logging Requests**

```
const logger = (req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next(); // Pass control to the next middleware
};

app.use(logger);
```

✓ Logs every request **before sending a response**.

CHAPTER 5: CONNECTING EXPRESS.JS WITH A DATABASE

5.1 Using MongoDB with Express.js

📌 **Install Mongoose (MongoDB ORM)**

```
npm install mongoose
```

📌 **Example: Connecting to MongoDB**

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

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

✓ Establishes a **connection with MongoDB**.

5.2 Defining a MongoDB Schema & Model

📌 Example: Creating a User Model

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

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

const User = mongoose.model("User", userSchema);
```

✓ Defines **users collection** in MongoDB.

5.3 CRUD Operations with MongoDB

📌 Example: Adding a New User to the Database

```
app.post("/users", async (req, res) => {
  const newUser = new User(req.body);
  await newUser.save();
  res.send("User Added");
});
```

✓ Stores user data **in MongoDB** dynamically.

Case Study: How Netflix Uses Express.js for Backend API Services

Challenges Faced by Netflix

✓ **Handling millions of concurrent users**.
✓ **Optimizing API response times** for streaming services.

Solutions Implemented

✓ Used **Express.js for lightweight, high-performance backend services**.
✓ Implemented **middleware for efficient request handling**.
✓ Integrated **MongoDB for user data management**.

- **Key Takeaways from Netflix's Backend Strategy:**

✓ **Express.js scales well for high-traffic applications.**
✓ **Middleware improves request processing efficiency.**
✓ **API optimization enhances streaming performance.**

📝 **Exercise**

✅ Create a **basic Express server** with a GET request.
✅ Implement **route parameters and query strings** in Express.

✅ Use **middleware** to log every request to the console.
✅ Connect Express to **MongoDB and perform CRUD operations**.

Conclusion

✓ **Express.js simplifies backend development** with minimal setup.
✓ **Routing and middleware enhance API efficiency**.
✓ **Connecting with databases allows dynamic data management**.
✓ **Using Express.js in production requires security and performance optimizations**.

MIDDLEWARE, ROUTES, AND CONTROLLERS

CHAPTER 1: INTRODUCTION TO MIDDLEWARE, ROUTES, AND CONTROLLERS

1.1 What Are Middleware, Routes, and Controllers?

Middleware, routes, and controllers are essential components in **backend development**, used for **handling HTTP requests, managing APIs, and processing data** in web applications.

- **Why Are They Important?**

✓ **Middleware** handles authentication, logging, and request validation.

✓ **Routes** define how URLs map to different functionalities.

✓ **Controllers** organize business logic for clean and maintainable code.

CHAPTER 2: UNDERSTANDING MIDDLEWARE IN EXPRESS.JS

2.1 What is Middleware?

Middleware is a **function that processes requests and responses** before reaching the final handler. It can:

✓ Modify request/response objects.

✓ Execute authentication checks.

✓ Log request details for debugging.

📌 **Example: Basic Middleware in Express.js**

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

const app = express();
```

```
const loggerMiddleware = (req, res, next) => {
  console.log(`${req.method} ${req.url} - ${new Date().toISOString()}`);
  next(); // Calls the next middleware in the stack
};

app.use(loggerMiddleware);

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

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

✓ Logs every incoming request **before it reaches the route handler**.

2.2 Types of Middleware

Middleware functions can be used for different purposes in Express.js applications.

Middleware Type	Purpose	Example Usage
Application-Level Middleware	Runs for all routes	Logging, authentication

Router-Level Middleware	Runs only for specific routes	API request validation
Built-in Middleware	Provided by Express.js	express.json() for parsing JSON
Error-Handling Middleware	Handles errors globally	Custom error messages

📌 **Example: Using express.json() Middleware for Parsing JSON Requests**

```
app.use(express.json()); // Parses JSON request bodies
```

✓ Enables API to **handle JSON request bodies**.

2.3 Implementing Authentication Middleware

📌 **Example: Protecting Routes with Authentication Middleware**

```
const authMiddleware = (req, res, next) => {
  const token = req.headers.authorization;
  if (token === "valid-token") {
    next(); // Allow access to the route
  } else {
    res.status(401).json({ message: "Unauthorized" });
  }
};

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

```
  res.send("Welcome to Dashboard");
});
```

✓ Blocks unauthorized access to the **dashboard route**.

Chapter 3: Understanding Routes in Express.js

3.1 What Are Routes?

Routes define **how URLs map to functionalities** in an application.

📌 **Basic Syntax of a Route in Express.js**

```
app.get("/home", (req, res) => {
  res.send("Welcome to Home Page");
});
```

✓ Handles **GET requests** to /home.

3.2 Types of Routes in Express.js

Route Type	Purpose	Example
GET	Fetch data	app.get("/users", callback)
POST	Send data	app.post("/users", callback)
PUT	Update data	app.put("/users/:id", callback)
DELETE	Remove data	app.delete("/users/:id", callback)

📌 **Example: Defining Multiple Routes in Express.js**

```
app.get("/users", (req, res) => res.send("Fetching all users"));
app.post("/users", (req, res) => res.send("Creating a new user"));
```

```
app.put("/users/:id", (req, res) => res.send(`Updating user ${req.params.id}`));

app.delete("/users/:id", (req, res) => res.send(`Deleting user ${req.params.id}`));
```

✓ Handles **CRUD operations** using routes.

3.3 Using Express Router for Modular Routing

📌 **Example: Using express.Router() for Cleaner Routes**

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

const router = express.Router();


router.get("/", (req, res) => res.send("Welcome to the API"));

router.get("/users", (req, res) => res.send("List of users"));


module.exports = router;
```

✓ This makes routes **modular and easier to manage**.

📌 **Using the Router in server.js**

```
const apiRoutes = require("./routes/api");

app.use("/api", apiRoutes);
```

✓ Routes are now accessed using /api/users.

CHAPTER 4: UNDERSTANDING CONTROLLERS IN EXPRESS.JS

4.1 What Are Controllers?

Controllers separate **business logic** from route definitions, making code more **structured**.

📌 Example: Without a Controller (Messy Code)

```
app.get("/users", (req, res) => {
  let users = [{ id: 1, name: "Alice" }, { id: 2, name: "Bob" }];
  res.json(users);
});
```

✓ Mixing **business logic and route handling** can make code difficult to maintain.

4.2 Using Controllers for Clean Code

📌 Step 1: Create a Controller (userController.js)

```
exports.getAllUsers = (req, res) => {
  let users = [{ id: 1, name: "Alice" }, { id: 2, name: "Bob" }];
  res.json(users);
};
```

📌 Step 2: Use Controller in Routes (userRoutes.js)

```
const express = require("express");
const router = express.Router();
const userController = require("../controllers/userController");

router.get("/users", userController.getAllUsers);
```

```
module.exports = router;
```

📌 Step 3: Import the Router in server.js

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

app.use("/api", userRoutes);
```

✓ **Routes remain clean, and controllers handle logic separately**.

Case Study: How GitHub Uses Middleware, Routes & Controllers

Challenges Faced by GitHub

✓ Needed to handle **millions of API requests per second**.
✓ Ensured **secure user authentication** for repositories.

Solutions Implemented

✓ Used **middleware for authentication** (OAuth, JWT tokens).
✓ Organized **routes using modular Express Router**.
✓ Separated **business logic into controllers** for efficiency.

🔹 **Key Takeaways from GitHub's API Design:**

✓ **Middleware ensures security** by authenticating API requests.
✓ **Modular routes improve maintainability** in large applications.
✓ **Controllers separate business logic** for better scalability.

📝 Exercise

✅ Create a **middleware function** that logs the time of each API request.
✅ Define **RESTful routes** (GET, POST, PUT, DELETE) for a products API.
✅ Implement **a controller** to handle fetching a list of users.

✅ Apply **authentication middleware** to protect a /dashboard route.

Conclusion

✓ Middleware processes requests before reaching the route handler.
✓ Routes define HTTP endpoints and CRUD operations.
✓ Controllers separate business logic for better maintainability.
✓ Using these concepts together ensures scalable backend development.

CRUD OPERATIONS WITH EXPRESS.JS

CHAPTER 1: INTRODUCTION TO EXPRESS.JS & CRUD OPERATIONS

1.1 What is Express.js?

Express.js is a **lightweight and fast web framework** for Node.js that simplifies building **backend APIs** and **web applications**. It helps manage **routes, requests, and responses** efficiently.

🔹 **Why Use Express.js?**

✓ Simplifies handling **HTTP requests & responses**.

✓ Supports **middleware** for authentication and validation.

✓ Enables **easy integration with databases (MongoDB, MySQL, PostgreSQL)**.

🔹 **What Are CRUD Operations?**

CRUD stands for:

- **C** → Create (Adding new data).
- **R** → Read (Fetching existing data).
- **U** → Update (Modifying data).
- **D** → Delete (Removing data).

📌 **Example: How CRUD Works in a Blog Application**

Operation	HTTP Method	API Endpoint	Example
Create	POST	/posts	Add a new blog post
Read	GET	/posts/:id	Retrieve a blog post
Update	PUT	/posts/:id	Edit a blog post

Delete	DELETE	/posts/:id	Remove a blog post

CHAPTER 2: SETTING UP EXPRESS.JS FOR CRUD OPERATIONS

2.1 Installing Express.js

📌 **Create a new Node.js project and install Express:**

mkdir express-crud

cd express-crud

npm init -y

npm install express

✓ npm init -y ⟶ Creates package.json for the project.
✓ npm install express ⟶ Installs Express.js.

2.2 Creating an Express Server

📌 **Create server.js and set up a basic Express server:**

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

const app = express();


// Middleware to parse JSON requests

app.use(express.json());


const PORT = 5000;
```

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

✓ express.json() enables parsing JSON data from requests.
✓ app.listen(5000) starts the server on **port 5000**.

CHAPTER 3: IMPLEMENTING CRUD OPERATIONS IN EXPRESS.JS

3.1 Creating Data Storage (Simulated Database)

📌 **Create a simple array to store blog posts:**

```
let posts = [
  { id: 1, title: "First Post", content: "This is my first post." },
  { id: 2, title: "Second Post", content: "This is my second post." }
];
```

✓ This will act as our **temporary database** before connecting to a real one.

3.2 Implementing the CREATE Operation (POST)

📌 **Define a route to create new posts:**

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

✓ req.body extracts data from the request.
✓ posts.push(newPost) **adds a new post** to the array.
✓ res.status(201).json(newPost) **returns the created post**.

🔹 **Testing with Postman or cURL:**

curl -X POST http://localhost:5000/posts -H "Content-Type: application/json" -d '{"title":"New Post","content":"This is a new blog post"}'

3.3 Implementing the READ Operation (GET)

📌 **Retrieve all posts:**

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

✓ Returns **all posts** in JSON format.

📌 **Retrieve a single post by ID:**

```
app.get("/posts/:id", (req, res) => {
  const post = posts.find(p => p.id === parseInt(req.params.id));
  post ? res.json(post) : res.status(404).json({ message: "Post not found" });
});
```

✓ req.params.id gets the **post ID from the URL**.
✓ find() retrieves the **matching post** or returns an error.

🔹 **Testing with Browser or cURL:**

curl http://localhost:5000/posts/1

3.4 Implementing the UPDATE Operation (PUT)

📌 Modify an existing post by ID:

```
app.put("/posts/:id", (req, res) => {
  const post = posts.find(p => p.id === parseInt(req.params.id));
  if (!post) return res.status(404).json({ message: "Post not found" });

  post.title = req.body.title || post.title;
  post.content = req.body.content || post.content;

  res.json(post);
});
```

✓ find() locates the **post to update**.
✓ req.body contains the **new values for update**.

🔹 Testing with Postman or cURL:

```
curl -X PUT http://localhost:5000/posts/1 -H "Content-Type: application/json" -d '{"title":"Updated Title"}'
```

3.5 Implementing the DELETE Operation (DELETE)

📌 Remove a post by ID:

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

});

✓ filter() **removes the matching post** from the array.
✓ Responds with a **success message**.

🔹 **Testing with Postman or cURL:**

curl -X DELETE http://localhost:5000/posts/1

CHAPTER 4: ENHANCING THE CRUD API

4.1 Adding Middleware for Logging Requests

📌 **Install morgan for logging:**

npm install morgan

📌 **Use it in server.js:**

const morgan = require("morgan");

app.use(morgan("dev"));

✓ Logs **every incoming request** for debugging.

4.2 Using CORS to Allow Frontend Requests

📌 **Install cors package:**

npm install cors

📌 **Enable CORS in server.js:**

const cors = require("cors");

app.use(cors());

✓ Prevents **cross-origin request issues** when calling the API from a frontend app.

Case Study: How Twitter Uses CRUD APIs for Tweets

Challenges Faced by Twitter

✓ Handling **millions of tweets per second**.
✓ Storing and retrieving **user-generated content** efficiently.
✓ Implementing **real-time updates for likes and retweets**.

Solutions Implemented

✓ Used **Express.js and Node.js for handling API requests**.
✓ Stored tweets in **NoSQL databases for fast retrieval**.
✓ Optimized API endpoints for **high-speed performance**.

- **Key Takeaways from Twitter's API Strategy:**

✓ **CRUD operations manage dynamic content efficiently.**
✓ **Middleware enhances logging and request handling.**
✓ **Optimized API responses ensure high performance.**

📝 Exercise

✅ Implement **pagination** to limit results when fetching posts.
✅ Add **timestamp fields** to store creation and update times.
✅ Improve API with **input validation** using express-validator.
✅ Connect Express.js CRUD operations to **MongoDB (using Mongoose)**.

Conclusion

✓ Express.js simplifies building REST APIs with CRUD operations.
✓ Middleware enhances functionality with logging and security.
✓ API endpoints efficiently manage database interactions.
✓ Real-world applications like Twitter use CRUD APIs for user-generated content.

JWT (JSON Web Tokens) for Authentication

Chapter 1: Introduction to JWT (JSON Web Tokens)

1.1 What is JWT?

JWT (JSON Web Token) is a **secure and compact way to transmit authentication information** between a client (browser, mobile app) and a server. It is commonly used for **user authentication and secure API communication**.

🔹 **Why Use JWT for Authentication?**

✓ **Stateless authentication** – No need for server-side sessions.

✓ **Compact and secure** – Uses JSON format with cryptographic signing.

✓ **Works across different platforms** – Web, mobile, API services.

🔹 **Real-World Use Cases:**

- User **logins** in web and mobile apps.
- API **authorization** (e.g., protected API routes).
- **Single sign-on (SSO)** across multiple applications.

Chapter 2: Structure of a JWT Token

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

📌 **Example JWT:**

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

.

eyJ1c2VySWQiOiIxMjMoNTYiLCJpYXQiOjE2MzM5MTg2MDB9

.

dFvGzV8E5RmGpTIle7K1F5Ow3cNfzGJx2v98M1DL4YI

- **Parts of a JWT Token:**

Part	Description	Example Data
Header	Defines token type & signing algorithm	{ "alg": "HS256", "typ": "JWT" }
Payload	Contains user data (claims)	{ "userId": "12345", "role": "admin" }
Signature	Ensures token integrity	Encrypted hash of header + payload

📌 **Decoded JWT Payload Example:**

```
{
  "userId": "123456",
  "role": "admin",
  "iat": 1633918600
}
```

✓ iat = Issued at timestamp.
✓ role = User role (e.g., admin, user).

CHAPTER 3: IMPLEMENTING JWT AUTHENTICATION IN NODE.JS

3.1 Installing Required Packages

To use JWT in a **Node.js + Express** application, install:

npm install express jsonwebtoken dotenv

✓ express – Web framework for handling routes.
✓ jsonwebtoken – Library for signing & verifying JWTs.
✓ dotenv – For managing environment variables securely.

3.2 Generating a JWT Token on User Login

📌 **Example: Creating a JWT Token in Express**

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

require("dotenv").config();


const generateToken = (user) => {

   return jwt.sign({ userId: user.id, role: user.role },
process.env.JWT_SECRET, { expiresIn: "1h" });

};


// Example usage

const user = { id: "123", role: "admin" };

const token = generateToken(user);

console.log("Generated Token:", token);
```

✓ jwt.sign() generates a token with **user ID & role**.
✓ expiresIn: "1h" makes the token **expire after 1 hour**.

CHAPTER 4: VERIFYING JWT TOKENS FOR PROTECTED ROUTES

4.1 Creating a Middleware to Protect API Routes

To secure API endpoints, **verify the JWT** before granting access.

📌 Example: Middleware to Verify JWT

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

const authenticateToken = (req, res, next) => {
  const token = req.header("Authorization")?.split(" ")[1];
  if (!token) return res.status(401).json({ message: "Access denied" });

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded; // Attach user data to request
    next();
  } catch (error) {
    res.status(403).json({ message: "Invalid token" });
  }
};

module.exports = authenticateToken;
```

✓ jwt.verify() ensures the token is **valid and not tampered with**.
✓ If valid, req.user is set to the **decoded user data**.

4.2 Securing an API Route with JWT Authentication

📌 Example: Protecting a Dashboard Route

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

const authenticateToken = require("./authMiddleware");


const app = express();


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

  res.json({ message: "Welcome to the dashboard!", user: req.user });

});


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

✓ Only **authenticated users** with a valid token can access /dashboard.
✓ req.user contains **user information extracted from JWT**.

CHAPTER 5: REFRESHING & EXPIRING JWT TOKENS

5.1 Implementing Token Expiry

JWT tokens should have an **expiration time** to enhance security.

📌 Example: Setting Expiry on JWT Token

```
const token = jwt.sign({ userId: "123" }, process.env.JWT_SECRET, { expiresIn: "15m" });

console.log("Token Expires in 15 Minutes:", token);
```

✓ The token will **automatically expire** after **15 minutes**.

5.2 Refreshing Expired JWT Tokens

If a token expires, users need to **request a new one** without logging in again.

📌 **Example: Refresh Token Implementation**

```
const refreshToken = (req, res) => {
  const oldToken = req.body.token;
  if (!oldToken) return res.status(401).json({ message: "No token provided" });

  try {
    const decoded = jwt.verify(oldToken, process.env.JWT_SECRET);
    const newToken = jwt.sign({ userId: decoded.userId }, process.env.JWT_SECRET, { expiresIn: "15m" });
    res.json({ token: newToken });
  } catch (error) {
    res.status(403).json({ message: "Invalid or expired token" });
  }
};
```

✓ **Generates a new token** when the old one expires.

Case Study: How Amazon Uses JWT for Secure User Authentication

Challenges Faced by Amazon

✓ Securing **millions of user sessions** globally.
✓ Preventing **unauthorized access to user accounts**.

Solutions Implemented

✓ Used **JWT authentication** for secure login sessions.
✓ Implemented **token expiration & refresh mechanisms**.
✓ Stored **tokens in HTTP-only cookies** to prevent XSS attacks.

🔹 **Key Takeaways from Amazon's Strategy:**
✓ **JWT improves security & scalability**.
✓ **Short-lived tokens minimize security risks**.
✓ **Token-based authentication supports microservices efficiently**.

📝 Exercise

✅ Create a **JWT token** for user authentication in Node.js.
✅ Implement **a middleware** to protect API routes using JWT.
✅ Set an **expiration time** for JWT and refresh the token after expiry.
✅ Use **Postman** to test a protected route by sending a JWT in the request header.

Conclusion

✓ **JWT is a secure way to handle authentication in web applications.**
✓ **JWT consists of a header, payload, and cryptographic**

signature.

✓ Tokens can be verified and decoded to retrieve user information.

✓ Using middleware, we can restrict access to protected routes.

✓ Token expiration and refresh mechanisms improve security.

HASHING PASSWORDS WITH BCRYPT

CHAPTER 1: INTRODUCTION TO PASSWORD HASHING

1.1 What is Password Hashing?

Password hashing is the process of **converting a plain-text password into a fixed-length encrypted string** using a mathematical function. It is a critical security practice that prevents storing passwords in plain text and enhances **user data protection**.

- **Why Use Password Hashing?**

✓ **Prevents password leaks** – Even if the database is compromised, hashed passwords are unreadable.

✓ **One-way encryption** – Cannot be reversed into the original password.

✓ **Adds security layers** – Includes **salting** to protect against dictionary and brute-force attacks.

- **Common Hashing Algorithms:**

Algorithm	**Security Level**	**Example**
MD5	Weak (vulnerable to attacks)	098f6bcd4621d373cade4e832627b4f6
SHA-256	Stronger but still crackable	5e884898da28047151d0e56f8dc62927
bcrypt	**Most Secure**	$2b$10$N9qo8uLOickgx2ZMRZoMye.PP6

	(adaptive hashing)	

🔹 **Why Use bcrypt?**

✓ Uses **adaptive hashing** (computational cost increases over time).

✓ Automatically generates a **salt** (random data added to hashing).

✓ **Slows down brute-force attacks,** making it more secure.

CHAPTER 2: INSTALLING AND SETTING UP BCRYPT

2.1 Installing bcrypt in a Node.js Project

📌 **Run the following command to install bcrypt:**

npm install bcrypt

or

yarn add bcrypt

2.2 Importing bcrypt in a JavaScript File

After installing, import bcrypt in your Node.js project:

const bcrypt = require("bcrypt");

CHAPTER 3: HASHING PASSWORDS WITH BCRYPT

3.1 Hashing a Password with bcrypt

bcrypt requires a **salt,** which adds a random value to each password before hashing, ensuring unique hashes for identical passwords.

📌 **Example: Hashing a Password**

const bcrypt = require("bcrypt");

```
const saltRounds = 10;

const password = "mySecurePassword";


bcrypt.hash(password, saltRounds, (err, hash) => {
   if (err) throw err;
   console.log("Hashed Password:", hash);
});
```

✓ saltRounds = 10 defines **how secure the hash should be** (higher values are more secure but slower).
✓ bcrypt.hash() generates a **unique hashed password**.

Example Output:

Hashed Password:
$2b$10$EIXKj4.H48YImi.YsTqFqOyk6OJeDMeqpC9UtvLeh1vN2Mq/uH8XW

🔹 **Even if the same password is hashed multiple times, it produces different hashes!**

3.2 Comparing a Hashed Password with User Input

When a user logs in, we need to **compare the hashed password stored in the database with the entered password**.

📌 **Example: Verifying Passwords with bcrypt.compare()**

```
const storedHash =
"$2b$10$EIXKj4.H48YImi.YsTqFqOyk6OJeDMeqpC9UtvLeh1vN2Mq/uH8XW"; // Example hash
```

```
bcrypt.compare("mySecurePassword", storedHash, (err, result) => {
  if (result) {
    console.log("Password is correct!");
  } else {
    console.log("Invalid password.");
  }
});
```

✓ bcrypt.compare() **compares** the input password with the stored hash.
✓ **Returns true if passwords match,** otherwise returns false.

CHAPTER 4: IMPLEMENTING BCRYPT IN A USER AUTHENTICATION SYSTEM

4.1 Hashing Passwords Before Storing in a Database

📌 **Example: Registering a New User and Storing Hashed Password**

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

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

```
const users = []; // Simulated database


app.post("/register", async (req, res) => {
  try {
    const hashedPassword = await bcrypt.hash(req.body.password, 10);
    users.push({ username: req.body.username, password: hashedPassword });
    res.status(201).send("User Registered!");
  } catch {
    res.status(500).send("Error registering user");
  }
});


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

✓ bcrypt.hash() **encrypts the password before saving it**.
✓ The **hashed password is stored,** instead of plain text.

4.2 Verifying Passwords During Login

📌 Example: Checking Passwords During User Login

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

```
  if (!user) return res.status(400).send("User not found");


  try {
    if (await bcrypt.compare(req.body.password, user.password)) {
      res.send("Login Successful!");
    } else {
      res.send("Incorrect Password");
    }
  } catch {
    res.status(500).send("Error during login");
  }
});
```

✓ bcrypt.compare() **checks if the entered password matches the stored hash**.
✓ Returns true for a correct password, otherwise false.

CHAPTER 5: BEST PRACTICES FOR PASSWORD HASHING WITH BCRYPT

- **1. Always Hash Passwords Before Storing**

Never store **plain-text passwords** in the database.

- **2. Use a Strong Salt Factor (saltRounds)**

✓ Recommended **salt rounds: 10-12** (higher values increase security but slow down hashing).

- **3. Avoid Hardcoding Secrets in Code**

Use environment variables (.env) to store sensitive values.

📌 Example: Using Environment Variables for Salt Rounds

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

const saltRounds = process.env.SALT_ROUNDS || 10;
```

- **4. Regularly Update Hashing Algorithms**

✓ bcrypt is secure **but may be replaced by better algorithms in the future**.

- **5. Implement Account Lockouts for Multiple Failed Attempts**

✓ Prevent **brute-force attacks** by limiting login attempts.

Case Study: How Facebook Uses Password Hashing for User Security

Challenges Faced by Facebook

✓ **Storing millions of user passwords securely**.
✓ **Preventing brute-force attacks** on user accounts.

Solutions Implemented

✓ Used **bcrypt with adaptive hashing** to protect stored passwords.
✓ Implemented **multi-factor authentication (MFA)** for added security.
✓ Monitored **login attempts to prevent credential stuffing attacks**.

- **Key Takeaways from Facebook's Security Model:**

✓ **Using bcrypt improves password security**.
✓ **Adaptive hashing protects against evolving threats**.
✓ **Implementing MFA reduces unauthorized access attempts**.

📝 Exercise

✅ **Hash a password using bcrypt** and store it in an array (simulate a database).
✅ **Create a function to compare a stored hash** with an entered password.
✅ **Implement a simple Express.js server** to handle user registration and login with bcrypt.
✅ **Enhance security** by adding environment variables for salt rounds.

Conclusion

✓ **bcrypt ensures passwords are stored securely** by hashing them before saving.
✓ **Password verification is done with bcrypt.compare()** to check login attempts.
✓ **Salting and adaptive hashing** protect against brute-force attacks.
✓ **Implementing bcrypt in real-world applications** improves overall security.

PROTECTING API ROUTES WITH AUTHENTICATION

CHAPTER 1: INTRODUCTION TO API AUTHENTICATION

1.1 What is API Authentication?

API authentication ensures that only **authorized users or systems** can access protected routes and resources in an application.

- **Why is API Authentication Important?**

✓ Prevents **unauthorized access** to sensitive data.
✓ Ensures **secure communication** between client and server.
✓ Allows role-based access control (**admin vs. user permissions**).

- **Common Authentication Methods:**

Method	Description	Example Usage
API Keys	Unique identifier used in requests	Public APIs (e.g., weather data)
JWT (JSON Web Token)	Token-based authentication	Securing user logins
OAuth	Third-party authentication	Login via Google, Facebook
Session-Based Auth	Server-side sessions stored in cookies	Traditional web apps

CHAPTER 2: SETTING UP A SECURE EXPRESS API

2.1 Installing Required Dependencies

To implement authentication in Express.js, install the following:

npm install express jsonwebtoken bcryptjs dotenv

✓ **jsonwebtoken (JWT)** → Creates and verifies authentication tokens.
✓ **bcryptjs** → Hashes user passwords securely.
✓ **dotenv** → Manages environment variables securely.

2.2 Creating a Basic Express API

📌 **Example: Setting Up Express Server (server.js)**

```
const express = require("express");
const dotenv = require("dotenv");
dotenv.config();

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

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

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

✓ Express **initializes the API** and listens for requests.
✓ **dotenv loads environment variables** (e.g., secret keys).

CHAPTER 3: USER REGISTRATION AND SECURE PASSWORD STORAGE

3.1 Hashing Passwords with Bcrypt.js

Storing passwords **in plaintext is insecure**. Instead, **hash passwords before saving them**.

📌 **Example: User Registration API with Password Hashing (authController.js)**

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


exports.register = async (req, res) => {
  const { username, password } = req.body;

  // Hash password before storing
  const salt = await bcrypt.genSalt(10);
  const hashedPassword = await bcrypt.hash(password, salt);

  // Simulate saving user in database
  const user = { id: 1, username, password: hashedPassword };
  res.status(201).json({ message: "User registered successfully", user });
};
```

✓ Uses bcrypt.hash() to **securely hash passwords** before storing them.

Chapter 4: Implementing JWT Authentication for API Protection

4.1 What is JWT (JSON Web Token)?

JWT is a **self-contained, encrypted token** used for authentication. It consists of:

1️⃣ **Header** → Defines the token type (JWT) and algorithm (HS256).

2️⃣ **Payload** → Contains user information (e.g., id, role).

3️⃣ **Signature** → Secures the token using a secret key.

📌 Example: Structure of a JWT Token

```
{
 "header": { "alg": "HS256", "typ": "JWT" },
 "payload": { "userId": 1, "username": "Alice" },
 "signature": "HMACSHA256(encodedHeader + encodedPayload, secret)"
}
```

✓ JWT tokens are **signed and verified** using a secret key.

4.2 Generating a JWT Token on Login

📌 Example: User Login API with JWT (authController.js)

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

```
exports.login = async (req, res) => {
  const { username, password } = req.body;

  // Simulated stored user (normally from a database)
  const storedUser = { id: 1, username: "Alice", password: "$2a$10$hashedpassword" };

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

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

  res.json({ message: "Login successful", token });
};
```

✓ jwt.sign() generates a **JWT token valid for 1 hour**.
✓ bcrypt.compare() verifies **hashed passwords** before issuing a token.

CHAPTER 5: PROTECTING API ROUTES USING MIDDLEWARE

5.1 Creating Authentication Middleware

Middleware ensures that **only authenticated users can access certain routes**.

📌 Example: Authentication Middleware (authMiddleware.js)

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


exports.verifyToken = (req, res, next) => {
  const token = req.header("Authorization");


  if (!token) return res.status(401).json({ message: "Access Denied" });


  try {
    const verified = jwt.verify(token.replace("Bearer ", ""), process.env.JWT_SECRET);
    req.user = verified;
    next();
  } catch (error) {
    res.status(400).json({ message: "Invalid Token" });
  }
};
```

✓ jwt.verify() **validates the token** before allowing access.
✓ **If invalid, the request is rejected with a 401 Unauthorized status**.

5.2 Protecting API Routes with Authentication Middleware

📌 Example: Securing the Dashboard Route (routes.js)

```
const express = require("express");
const { verifyToken } = require("../middleware/authMiddleware");
const router = express.Router();

router.get("/dashboard", verifyToken, (req, res) => {
  res.json({ message: `Welcome, ${req.user.username}!` });
});

module.exports = router;
```

✓ Only **authenticated users** with a valid JWT token can access /dashboard.

📌 Example: Sending Authenticated Requests Using Fetch API (Frontend)

```
fetch("http://localhost:5000/dashboard", {
  headers: { Authorization: "Bearer YOUR_JWT_TOKEN" }
})
.then(response => response.json())
```

```
.then(data => console.log(data))

.catch(error => console.error("Error:", error));
```

✓ Frontend **attaches JWT token** in request headers for authentication.

Case Study: How Amazon Secures API Routes with JWT

Challenges Faced by Amazon

✓ Ensuring **secure authentication** for millions of users.
✓ Protecting **sensitive order & payment APIs**.

Solutions Implemented

✓ Used **JWT tokens** for scalable authentication.
✓ Enforced **role-based access control (admin vs. users)**.
✓ Secured sensitive data with **middleware protections**.

- **Key Takeaways from Amazon's API Security Model:**

✓ JWT ensures session security without storing sensitive data on the server.
✓ Middleware protects critical API endpoints from unauthorized access.
✓ Role-based authentication prevents unauthorized admin actions.

📝 **Exercise**

✅ Create a **user authentication API** that issues JWT tokens upon login.
✅ Implement **middleware** to protect an API route.

✅ Fetch a protected API endpoint using **JWT token in headers**.
✅ Implement **role-based authentication** (Admin vs. User).

Conclusion

✓ **Middleware protects API routes** by validating JWT tokens.
✓ **JWT provides scalable, stateless authentication**.
✓ **Password hashing secures user credentials**.
✓ **Protecting API endpoints prevents unauthorized data access**.

ASSIGNMENT:

DEVELOP A USER AUTHENTICATION SYSTEM WITH NODE.JS & EXPRESS.JS.

Assignment Solution: Develop a User Authentication System with Node.js & Express.js

Step 1: Setting Up the Project

1.1 Initialize a Node.js Project

📌 Open the terminal and run:

```
mkdir auth-system
cd auth-system
npm init -y
```

✓ Creates a Node.js project with package.json.

1.2 Install Required Packages

📌 Install Express, bcrypt, JWT, dotenv, and MongoDB driver:

```
npm install express bcryptjs jsonwebtoken mongoose dotenv cors
```

✓ express → Web framework for handling requests.
✓ bcryptjs → Hashes passwords securely.
✓ jsonwebtoken → Generates and verifies authentication tokens.
✓ mongoose → Connects to MongoDB.
✓ dotenv → Manages environment variables securely.
✓ cors → Allows frontend to access the API.

Step 2: Setting Up Express Server

📌 Create server.js and set up Express:

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

const mongoose = require("mongoose");

const dotenv = require("dotenv");


dotenv.config();

const app = express();


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

app.use(require("cors")()); // Enable CORS for frontend


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

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

✓ Parses **JSON requests** and **enables CORS**.
✓ Starts the server on **port 5000**.

Step 3: Setting Up MongoDB with Mongoose

3.1 Connect to MongoDB

📌 Create a .env file and add your MongoDB URL:

```
MONGO_URI=mongodb+srv://your_user:your_password@cluster.mongodb.net/authDB

JWT_SECRET=your_jwt_secret
```

📌 Modify server.js to connect to MongoDB:

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

✓ Securely **stores credentials in .env**.
✓ Uses **MongoDB Atlas or local MongoDB**.

Step 4: Creating User Model

📌 Create a folder models/ and inside, create User.js:

```
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 }
});

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

✓ Defines a **User model with username, email, and password**.
✓ Ensures **unique usernames and emails**.

Step 5: Implementing User Registration

📌 Create a routes/auth.js file for authentication routes:

```
const express = require("express");
const bcrypt = require("bcryptjs");
const User = require("../models/User");

const router = express.Router();

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

    // Check if user exists
    if (await User.findOne({ email })) {
      return res.status(400).json({ message: "Email already exists" });
    }

    // Hash password
    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(password, salt);

    // Save user to database
```

```
    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({ error: error.message });
  }
});

module.exports = router;
```

✓ **Hashes passwords** before storing them.
✓ Prevents duplicate registrations.
✓ Returns **success response** after saving the user.

- **Testing the Registration API with Postman:**
  - Method: **POST**
  - URL: http://localhost:5000/auth/register
  - Body (JSON):

```
{
  "username": "john_doe",
  "email": "john@example.com",
  "password": "securepassword"
}
```

Step 6: Implementing User Login with JWT Authentication

📌 Add a login route in routes/auth.js:

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


// Login User
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({ token, user: { username: user.username, email: user.email } });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

✓ Verifies **email and password** before logging in.
✓ Uses **JWT to generate an authentication token**.
✓ Returns **user details and token** on successful login.

🔹 **Testing the Login API with Postman:**

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

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

Step 7: Protecting Routes Using Middleware

7.1 Creating Authentication Middleware

📌 **Create middleware/authMiddleware.js to protect routes:**

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

```
module.exports = (req, res, next) => {
  const token = req.header("Authorization");
  if (!token) return res.status(401).json({ message: "Access Denied" });

  try {
    const verified = jwt.verify(token, process.env.JWT_SECRET);
    req.user = verified;
    next();
  } catch (error) {
    res.status(400).json({ message: "Invalid Token" });
  }
};
```

✓ Checks if **token is provided** in headers.
✓ Verifies **token validity** before allowing access.

7.2 Creating a Protected Route

📌 **Modify routes/auth.js to add a protected route:**

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

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

```
  const user = await User.findById(req.user.userId).select("-
password");
  res.json(user);
});
```

✓ Only authenticated users can access **/profile**.

- **Testing the Protected API with Postman:**
  - Method: **GET**
  - URL: http://localhost:5000/auth/profile
  - Headers:
    - Authorization: Bearer YOUR_JWT_TOKEN

Case Study: How Amazon Uses Authentication Systems

Challenges Faced by Amazon

✓ Handling **millions of user logins securely**.
✓ Preventing **unauthorized access to user accounts**.

Solutions Implemented

✓ Used **bcrypt for secure password storage**.
✓ Implemented **JWT authentication for secure session handling**.
✓ Used **middleware to protect user data**.

- **Key Takeaways from Amazon's Authentication System:**

✓ **Secure password hashing prevents leaks**.
✓ **JWT provides fast, stateless authentication**.
✓ **Middleware restricts unauthorized access to user data**.

📝 Exercise

✅ Extend the authentication system to **allow password resets**.
✅ Implement **role-based authentication** (Admin vs. User).
✅ Add **email verification** using nodemailer.

Conclusion

✓ Express.js provides a simple backend for authentication systems.
✓ bcrypt ensures passwords are securely stored.
✓ JWT enables secure and scalable authentication.
✓ Middleware protects private routes from unauthorized access.