ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

INTRODUCTION TO EXPRESS.JS & NODE.JS (WEEKS 1-2)

INTRODUCTION TO NODE.JS & EXPRESS.JS

CHAPTER 1: UNDERSTANDING NODE.JS

1.1 What is Node.js?

Node.js is a **runtime environment** that allows developers to run **JavaScript on the server-side**. It is built on **Chrome's V8 JavaScript engine** and enables **asynchronous, event-driven programming**.

- **Why Use Node.js?**

✓ **Non-blocking, event-driven architecture** for handling multiple requests efficiently.

✓ **Uses JavaScript for both frontend & backend**, simplifying development.

✓ **High scalability** for building APIs, real-time applications, and microservices.

- **Key Features of Node.js**

Feature	Description

Asynchronous & Non-blocking I/O	Handles multiple requests without blocking execution.
Single-threaded but highly scalable	Uses event loops instead of threads.
NPM (Node Package Manager)	Access to thousands of reusable libraries.
Built-in Modules	Includes fs, http, path, and more for system operations.

1.2 Installing Node.js

📌 Step 1: Download & Install Node.js

- Go to Node.js Official Website.
- Install **LTS (Long-Term Support) version** for stability.
- Verify installation:

```
node -v  # Check Node.js version
npm -v   # Check npm version
```

✓ Confirms Node.js and npm are installed.

1.3 Running Your First Node.js Script

📌 Create a File app.js and Add:

```
console.log("Hello, Node.js!");
```

📌 Run the Script in Terminal:

```
node app.js
```

✓ Prints "Hello, Node.js!" in the console.

CHAPTER 2: INTRODUCTION TO EXPRESS.JS

2.1 What is Express.js?

Express.js is a **minimal and flexible web framework for Node.js** that simplifies building APIs and web applications.

- **Why Use Express.js?**

✓ **Simplifies HTTP request handling** (GET, POST, PUT, DELETE).
✓ **Supports middleware for modular development.**
✓ **Efficient routing system for building REST APIs.**

- **Key Features of Express.js**

Feature	Description
Fast Routing	Simple and efficient route handling.
Middleware Support	Helps process requests before sending a response.
Template Engine Support	Works with EJS, Handlebars, Pug, etc.
REST API Support	Easily build APIs for web and mobile applications.

2.2 Installing Express.js

📌 **Step 1: Initialize a Node.js Project**

mkdir express-app

cd express-app

npm init -y

✓ Creates a package.json file for managing dependencies.

📌 Step 2: Install Express.js

npm install express

✓ Installs Express.js in the project.

2.3 Creating a Basic Express Server

📌 Create server.js and Add:

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

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

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

📌 Run the Server:

node server.js

✓ Starts a web server at http://localhost:3000.

CHAPTER 3: UNDERSTANDING ROUTING & MIDDLEWARE IN EXPRESS.JS

3.1 Handling Routes in Express.js

📌 Example: Defining Multiple Routes

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

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

✓ **Routes handle different URLs** in Express.js applications.

3.2 Using Middleware in Express.js

Middleware functions **process requests before sending a response**.

📌 Example: Using Middleware for Logging

```
const logger = (req, res, next) => {
  console.log(`Request Method: ${req.method}, URL: ${req.url}`);
  next();
};

app.use(logger);
```

✓ Logs every request before moving to the next function.

CHAPTER 4: WORKING WITH REST APIS IN EXPRESS.JS

4.1 Creating REST API Endpoints

📌 Example: Implementing CRUD Operations

```
app.use(express.json()); // Middleware for parsing JSON

let users = [{ id: 1, name: "Alice" }];

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

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

✓ Implements **GET and POST API routes** for managing users.

CHAPTER 5: ERROR HANDLING & DEPLOYMENT IN EXPRESS.JS

5.1 Handling Errors in Express.js

📌 Example: Using Express Error Middleware

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

});

✓ Captures and handles **unexpected errors**.

5.2 Deploying an Express.js App

📌 Deploying to Heroku

1. Initialize Git:

```
git init
git add .
git commit -m "Initial commit"
```

2. Deploy to Heroku:

```
heroku create express-app
git push heroku main
```

✓ The app is now **live on Heroku**.

Case Study: How Netflix Uses Node.js & Express.js

Challenges Faced by Netflix

✓ Handling **millions of concurrent video streams**.
✓ Ensuring **fast API response times**.

Solutions Implemented

✓ Used **Node.js for real-time data processing**.
✓ Built APIs using **Express.js for content delivery**.

- **Key Takeaways from Netflix's Tech Stack:**

✓ **Express.js improves API speed & performance.**

✓ **Middleware enhances security & logging.**

✓ **Scalable architecture allows handling high traffic.**

📝 Exercise

✅ Install **Node.js & Express.js** on your system.

✅ Create a **basic Express.js server** and run it.

✅ Implement **routes for Home, About, and Contact pages**.

✅ Set up a **simple API that returns a list of users**.

Conclusion

✓ **Node.js enables server-side JavaScript execution.**

✓ **Express.js simplifies web server and API development.**

✓ **Middleware helps process requests efficiently.**

✓ **RESTful APIs in Express.js provide scalable backend solutions.**

INSTALLING AND SETTING UP EXPRESS.JS

CHAPTER 1: INTRODUCTION TO EXPRESS.JS

1.1 What is Express.js?

Express.js is a **fast, minimalist, and flexible web framework** for Node.js. It simplifies the process of **building web applications, APIs, and backend services**.

- **Why Use Express.js?**

✓ Provides a **lightweight** framework with powerful features.
✓ Makes it easy to **handle HTTP requests and responses**.
✓ Supports **middleware**, which enhances functionality.
✓ Works well with **databases, authentication, and real-time applications**.

- **Common Use Cases of Express.js:**

Use Case	Example
REST APIs	Building a backend for a mobile or web app
Full-Stack Applications	Used in MERN (MongoDB, Express, React, Node) stack
Real-Time Applications	Used with WebSockets for chat apps
Middleware Implementation	Authentication, logging, request validation

CHAPTER 2: INSTALLING NODE.JS & EXPRESS.JS

2.1 Prerequisites for Installing Express.js

Before setting up Express.js, you need to **install Node.js and npm** (Node Package Manager).

📌 Check if Node.js is Installed

node -v

npm -v

✓ If Node.js is **not installed**, download and install it from:
👉 Node.js Official Website

2.2 Setting Up an Express.js Project

📌 Step 1: Create a Project Folder

mkdir express-app

cd express-app

✓ This creates and navigates into a **new project directory**.

📌 Step 2: Initialize a Node.js Project

npm init -y

✓ Generates a package.json file, which manages dependencies.

📌 Step 3: Install Express.js

npm install express

✓ This installs **Express.js and its dependencies** in the project.

CHAPTER 3: CREATING A BASIC EXPRESS SERVER

3.1 Writing the First Express Server

📌 **Create a file server.js and add:**

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

const app = express();


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

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

});


const PORT = 5000;

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

✓ Imports Express.js and creates a **basic server**.
✓ Handles a **GET request** on /.

📌 **Run the Server**

```
node server.js
```

✓ The server starts on **http://localhost:5000/**.

3.2 Testing the Express Server

📌 **Open a browser and visit:**

http://localhost:5000/

✓ You should see **"Hello, Express!"** displayed.

📌 **Using Postman or cURL for API Testing:**

curl http://localhost:5000/

✓ Confirms the server **responds correctly**.

CHAPTER 4: UNDERSTANDING EXPRESS.JS STRUCTURE

4.1 Key Components of an Express.js App

🔹 **Common Express.js Files & Folders:**

File/Folder	Purpose
server.js	Entry point for Express server
routes/	Defines API routes
middlewares/	Stores custom middleware functions
controllers/	Manages business logic
models/	Handles database schemas

📌 **Example Folder Structure for a Scalable Express App:**

```
express-app/
 ├── server.js
 ├── package.json
 ├── routes/
 │    ├── users.js
 ├── controllers/
 │    ├── userController.js
 ├── models/
 │    ├── userModel.js
```

```
│ ——— middlewares/
│    ├——— authMiddleware.js
```

✓ Helps organize **large Express applications** efficiently.

CHAPTER 5: USING NODEMON FOR AUTOMATIC SERVER RESTART

5.1 Installing Nodemon

Nodemon automatically **restarts the server** when file changes are detected.

📌 **Install Nodemon Globally:**

```
npm install -g nodemon
```

📌 **Run the Express Server with Nodemon:**

```
nodemon server.js
```

✓ The server **automatically restarts** on code changes.

Case Study: How Netflix Uses Express.js

Challenges Faced by Netflix

✓ Handling **millions of user requests** per second.
✓ Ensuring **fast API responses** for streaming services.

Solutions Implemented

✓ Used **Express.js for backend microservices**.
✓ Implemented **load balancing** to distribute traffic.
✓ Used **caching techniques** to optimize API responses.

🔹 **Key Takeaways from Netflix's Use of Express.js:**

✓ **Lightweight framework for high-performance APIs.**

✓ **Fast request processing for real-time streaming.**

✓ **Scalable architecture with Express.js microservices.**

📝 Exercise

✅ Install **Node.js & Express.js** and set up a new project.

✅ Create a **basic Express.js server** with a single route.

✅ Test the server using **Postman or a browser**.

✅ Use **nodemon** to automate server restarts.

Conclusion

✓ **Express.js simplifies backend development for web applications.**

✓ **Installing Express and setting up a server takes just a few minutes.**

✓ **Organizing Express projects improves scalability and maintainability.**

✓ **Tools like Nodemon enhance development efficiency.**

UNDERSTANDING MIDDLEWARE & ROUTING IN EXPRESS.JS

CHAPTER 1: INTRODUCTION TO MIDDLEWARE & ROUTING

1.1 What is Middleware?

Middleware is a function that **runs between the request and response cycle** in an Express.js application. It helps in modifying the request (req) and response (res) objects, executing code, handling errors, and managing authentication.

- **Why Use Middleware?**

✓ Processes **requests before reaching the route handler**.
✓ Handles **authentication, logging, error handling, and security**.
✓ Makes Express applications **modular and maintainable**.

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

Middleware Type	Purpose	Example
Built-in Middleware	Included in Express for common tasks	express.json()
Third-party Middleware	Installed from npm for added functionality	cors, helmet, morgan
Custom Middleware	User-defined functions for specific needs	Logging, Authentication, Error Handling

CHAPTER 2: USING MIDDLEWARE IN EXPRESS.JS

2.1 Built-in Middleware

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

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

const app = express();


app.use(express.json()); // Parses incoming JSON requests


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


app.listen(5000, () => console.log("Server running on port 5000"));
```

✓ Parses **incoming JSON request bodies** automatically.

2.2 Third-Party Middleware

📌 Example: Using cors for Cross-Origin Resource Sharing

```
npm install cors

const cors = require("cors");

app.use(cors()); // Enables CORS for all routes
```

✓ Allows frontend applications **to access backend APIs** from different domains.

📌 Example: Using morgan for Logging Requests

```
npm install morgan
```

```
const morgan = require("morgan");
app.use(morgan("dev")); // Logs HTTP requests
```

✓ Logs **incoming requests in the console** for debugging.

2.3 Creating Custom Middleware

📌 Example: Writing a Middleware Function to Log Requests

```
const requestLogger = (req, res, next) => {
  console.log(`${req.method} request made to ${req.url}`);
  next(); // Calls the next middleware or route handler
};

app.use(requestLogger);
```

✓ Logs **each request method and URL**.

📌 Example: Middleware for User Authentication

```
const authenticateUser = (req, res, next) => {
  if (!req.headers.authorization) {
    return res.status(401).json({ message: "Unauthorized" });
  }
  next();
};
```

app.use("/secure", authenticateUser); // Protects all "/secure" routes

✓ Ensures **only authenticated users** can access protected routes.

CHAPTER 3: UNDERSTANDING ROUTING IN EXPRESS.JS

3.1 What is Routing?

Routing defines how an **Express.js application responds** to client requests (URLs and HTTP methods).

🔹 **Basic Syntax for Routes:**

app.METHOD(PATH, HANDLER);

✓ METHOD → HTTP method (GET, POST, PUT, DELETE).
✓ PATH → URL endpoint.
✓ HANDLER → Function to process the request.

3.2 Creating Routes in Express.js

📌 **Example: Defining Basic Routes**

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

app.post("/submit", (req, res) => {
  res.send("Data received successfully!");
});
```

```
app.put("/update", (req, res) => {
   res.send("Data updated!");
});

app.delete("/delete", (req, res) => {
   res.send("Data deleted!");
});
```

✓ Handles **GET, POST, PUT, DELETE requests**.

3.3 Using Route Parameters

📌 **Example: Dynamic Route with URL Parameters**

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

✓ :id captures **dynamic values from the URL**.

📌 **Request Example:**

GET /users/123

✓ Response: "User ID: 123"

3.4 Using Query Parameters

📌 **Example: Handling Query Strings (?key=value)**

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

✓ **Request:**

GET /search?q=express

✓ **Response:** "Searching for express"

CHAPTER 4: ORGANIZING ROUTES WITH EXPRESS ROUTER

4.1 Using Express Router for Modular Routes

📌 **Step 1: Create routes/users.js for User Routes**

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

router.get("/", (req, res) => res.send("User List"));
router.post("/", (req, res) => res.send("Create User"));
router.get("/:id", (req, res) => res.send(`User ID: ${req.params.id}`));

module.exports = router;
```

📌 **Step 2: Use the Router in server.js**

```
const userRoutes = require("./routes/users");
app.use("/users", userRoutes);
```

✓ Organizes routes into **separate files for maintainability**.

Chapter 5: Error Handling in Middleware & Routes

5.1 Handling Route Not Found (404 Errors)

📌 Example: Catching Undefined Routes

```
app.use((req, res, next) => {
  res.status(404).json({ message: "Route not found" });
});
```

✓ Ensures **invalid URLs return a proper error message**.

5.2 Global Error Handling Middleware

📌 Example: Handling Server Errors

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

✓ Captures **uncaught errors** and prevents application crashes.

Case Study: How Uber Uses Middleware & Routing in Express.js

Challenges Faced by Uber

✓ Handling **millions of ride requests** efficiently.
✓ Implementing **secure authentication for drivers & users**.
✓ Managing **real-time ride updates**.

Solutions Implemented

✓ Used **Middleware for authentication, logging, and caching**.
✓ Implemented **modular routes** for drivers, passengers, and rides.
✓ Optimized API **with Express Router and caching strategies**.

🔹 **Key Takeaways from Uber's Express.js Strategy:**

✓ **Middleware simplifies authentication and error handling.**
✓ **Express Router keeps API routes modular and scalable.**
✓ **Optimized routing improves performance in large-scale applications.**

📝 **Exercise**

✅ Create a **custom middleware** that logs request details (method, URL).
✅ Implement **route parameters** to fetch user details by ID.
✅ Use **Express Router** to organize API routes.
✅ Add **global error handling middleware** for catching errors.

Conclusion

✓ **Middleware manages authentication, logging, and security.**
✓ **Built-in, third-party, and custom middleware improve app efficiency.**
✓ **Routing in Express.js allows handling multiple request types.**
✓ **Express Router organizes routes into modular components.**

MANAGING GET, POST, PUT, DELETE REQUESTS IN EXPRESS.JS

CHAPTER 1: INTRODUCTION TO HTTP METHODS

1.1 What are HTTP Methods?

HTTP methods define how clients interact with a server. The most common methods used in RESTful APIs are:

- **GET** → Retrieve data from the server.
- **POST** → Send data to the server to create a new resource.
- **PUT** → Update or replace an existing resource.
- **DELETE** → Remove a resource from the server.

- **Why Are HTTP Methods Important?**

✓ Ensure **structured communication** between frontend and backend.
✓ Help in building **RESTful APIs**.
✓ Allow CRUD (**Create, Read, Update, Delete**) operations.

CHAPTER 2: SETTING UP AN EXPRESS SERVER

2.1 Installing Express.js

📌 **Step 1: Initialize a Node.js Project**

mkdir express-api

cd express-api

npm init -y

📌 **Step 2: Install Express.js**

npm install express

📌 Step 3: Create server.js and Set Up Express Server

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

const app = express();


app.use(express.json()); // Middleware for JSON request body parsing


const PORT = 5000;

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

✓ Starts the **Express.js server on port 5000**.
✓ Uses **JSON middleware** to parse request bodies.

CHAPTER 3: HANDLING GET REQUESTS

3.1 What is a GET Request?

A **GET request** is used to **retrieve data from the server**. It does not modify any data.

📌 Example: Fetching a List of Users

```
const users = [
  { id: 1, name: "Alice" },
  { id: 2, name: "Bob" }
];
```

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

✓ Clients send a **GET request** to /api/users.
✓ The server **responds with a JSON array** of users.

3.2 Fetching Data by ID

📌 **Example: Retrieve a Specific User**

```
app.get("/api/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);
});
```

✓ Extracts id from req.params.
✓ Returns **404 Not Found** if the user doesn't exist.

📌 **Testing with Postman or cURL**

```
curl http://localhost:5000/api/users/1
```

✓ Should return { "id": 1, "name": "Alice" }.

CHAPTER 4: HANDLING POST REQUESTS

4.1 What is a POST Request?

A **POST request** is used to **send data to the server to create a new resource**.

📌 Example: Adding a New User

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

✓ req.body.name retrieves data from **the client request**.
✓ The new user is **added to the array** and returned with a 201 Created status.

📌 Testing with Postman

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

```
{
  "name": "Charlie"
}
```

✓ Should return { "id": 3, "name": "Charlie" }.

CHAPTER 5: HANDLING PUT REQUESTS

5.1 What is a PUT Request?

A **PUT request** is used to **update an existing resource** completely.

📌 Example: Updating a User's Name

```
app.put("/api/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; // Updating the user name
  res.json(user);
});
```

✓ Extracts id from req.params.
✓ Updates the user's **name from req.body**.

📌 Testing with Postman

- **Method:** PUT
- **URL:** http://localhost:5000/api/users/1
- **Body (JSON):**

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

✓ Should return { "id": 1, "name": "Alice Updated" }.

CHAPTER 6: HANDLING DELETE REQUESTS

6.1 What is a DELETE Request?

A **DELETE request** removes a resource from the server.

📌 Example: Deleting a User

```
app.delete("/api/users/:id", (req, res) => {
  const index = users.findIndex(u => u.id === parseInt(req.params.id));
  if (index === -1) return res.status(404).json({ message: "User not found" });

  users.splice(index, 1); // Remove user from the array
  res.json({ message: "User deleted successfully" });
});
```

✓ Finds the user **by ID** and removes it.
✓ Returns a **confirmation message** upon success.

📌 Testing with Postman

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

✓ Should return { "message": "User deleted successfully" }.

CHAPTER 7: ERROR HANDLING & MIDDLEWARE FOR API REQUESTS

7.1 Implementing Global Error Handling

📌 Example: Adding Error Handling Middleware

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

✓ Catches **unhandled errors** and sends a **500 response**.

7.2 Validating API Requests

📌 **Example: Validating User Input Before Creating a User**

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

app.post("/api/users",
  body("name").notEmpty().withMessage("Name is required"),
  (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });

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

✓ Uses **express-validator** to **validate request body** before processing.

📌 **Testing with Invalid Data in Postman**

- **Method:** POST
- **Body:** {}
  ✓ Should return { "errors": [{ "msg": "Name is required" }] }.

Case Study: How Twitter Manages API Requests

Challenges Faced by Twitter

✓ Handling **millions of GET requests for tweets**.
✓ Preventing **spam POST requests (fake tweets)**.
✓ Securing user data from **unauthorized updates and deletions**.

Solutions Implemented

✓ **Implemented caching** for frequently accessed tweets (GET).
✓ **Rate limiting & authentication** for sensitive operations (POST, PUT, DELETE).
✓ **Centralized error handling** for smooth API responses.

🔹 **Key Takeaways from Twitter's API Strategy:**
✓ **Proper validation & security prevent abuse.**
✓ **Middleware enhances API request processing.**
✓ **Handling errors properly improves user experience.**

📝 **Exercise**

✅ Build an Express API with **GET, POST, PUT, DELETE endpoints**.
✅ Implement **error handling middleware** for failed API calls.

✅ Validate input fields using **express-validator**.

✅ Secure API routes using **authentication middleware**.

Conclusion

✓ GET retrieves data, POST creates data, PUT updates data, and DELETE removes data.

✓ Middleware enhances security, error handling, and validation.

✓ Express.js simplifies API request handling efficiently.

✓ Testing API requests ensures reliable backend performance.

HANDLING QUERY PARAMETERS AND REQUEST BODIES IN EXPRESS.JS

CHAPTER 1: INTRODUCTION TO HANDLING API REQUESTS IN EXPRESS.JS

1.1 What are Query Parameters and Request Bodies?

In Express.js, data is sent to the server in **two main ways**:

1. **Query Parameters** – Passed in the URL as key-value pairs (?key=value).
2. **Request Body** – Sent in the HTTP body (commonly in POST or PUT requests).

- **Why are Query Parameters & Request Bodies Important?**

✓ Used to **filter, sort, and search data** in APIs.
✓ Allow sending **user inputs in API requests**.
✓ Help in **data validation and processing**.

- **Example API Usage:**

Request Type	Example URL	Purpose
Query Parameters	/users?age=25	Get users aged 25
Request Body	{ "name": "Alice" }	Create a new user

CHAPTER 2: HANDLING QUERY PARAMETERS IN EXPRESS.JS

2.1 What are Query Parameters?

Query parameters allow **sending extra data** in a URL using the ? symbol. Multiple parameters are joined with &.

📌 **Example: URL with Query Parameters**

http://localhost:3000/users?age=25&gender=female

✓ Retrieves users who are **25 years old and female**.

2.2 Extracting Query Parameters in Express.js

📌 **Example: Handling Query Parameters in an API Route**

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

const app = express();


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

  const { age, gender } = req.query; // Extract query params

  res.json({ message: `Filtering users by age: ${age}, gender: ${gender}` });

});


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

✓ req.query retrieves the **age and gender from the URL**.

📌 **Request Example:**

http://localhost:3000/users?age=30&gender=male

📌 **Response Example:**

```
{ "message": "Filtering users by age: 30, gender: male" }
```

2.3 Using Query Parameters for Searching & Filtering

📌 **Example: Searching Users by Name**

```
app.get("/search", (req, res) => {
  const { name } = req.query;
  if (!name) return res.status(400).json({ error: "Name is required" });

  res.json({ message: `Searching for user: ${name}` });
});
```

✓ Returns **error if no query parameter is provided.**

📌 **Request Example:**

http://localhost:3000/search?name=Alice

📌 **Response Example:**

```
{ "message": "Searching for user: Alice" }
```

CHAPTER 3: HANDLING REQUEST BODIES IN EXPRESS.JS

3.1 What is a Request Body?

Request bodies contain **data sent by clients** (like form submissions or JSON payloads). Used in POST, PUT, PATCH requests.

🔹 **Comparison: Query Parameters vs. Request Body**

Feature	Query Parameters	Request Body
Where Data is Sent	URL	HTTP Body
Used In	GET requests	POST, PUT, PATCH

Example	/users?name=John	{ "name": "John" }

3.2 Extracting Request Body in Express.js

📌 Example: Handling POST Requests with express.json() Middleware

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


app.post("/users", (req, res) => {
  const { name, email } = req.body;
  res.json({ message: `User ${name} with email ${email} added.` });
});
```

✓ express.json() enables **request body parsing** in Express.js.

📌 Request Example (Using Postman or cURL):

```
POST http://localhost:3000/users
Content-Type: application/json
{
  "name": "Alice",
  "email": "alice@example.com"
}
```

📌 Response Example:

```
{ "message": "User Alice with email alice@example.com added." }
```

3.3 Validating Request Body Data

📌 **Example: Checking for Missing Fields**

```
app.post("/register", (req, res) => {
  const { username, password } = req.body;
  if (!username || !password) {
    return res.status(400).json({ error: "Username and password required" });
  }
  res.json({ message: `User ${username} registered successfully` });
});
```

✓ Returns an **error if required fields are missing**.

📌 **Request Example:**

```
{
  "username": "JohnDoe"
}
```

📌 **Response Example:**

```
{ "error": "Username and password required" }
```

CHAPTER 4: USING URL PARAMETERS (REQ.PARAMS)

4.1 What are URL Parameters?

URL parameters are **dynamic values** passed within a URL instead of query parameters.

📌 **Example URL:**

http://localhost:3000/users/123

✓ 123 is a **URL parameter representing user ID**.

4.2 Extracting URL Parameters in Express.js

📌 **Example: Handling Dynamic User IDs**

```
app.get("/users/:id", (req, res) => {
  const userId = req.params.id; // Extracting ID from URL
  res.json({ message: `Fetching details for user ID: ${userId}` });
});
```

✓ req.params.id retrieves **the dynamic user ID from the URL**.

📌 **Request Example:**

http://localhost:3000/users/5

📌 **Response Example:**

```
{ "message": "Fetching details for user ID: 5" }
```

CHAPTER 5: BEST PRACTICES FOR HANDLING QUERY PARAMETERS & REQUEST BODIES

🔹 **1. Always Validate User Input**

✓ Use **express-validator** for request validation.

✓ Prevent **injection attacks** with proper sanitization.

🔹 **2. Use Default Values for Query Parameters**

✓ Example: ?limit=10&page=1 → Ensures pagination works smoothly.

🔹 3. Return Proper HTTP Status Codes

Status Code	Meaning
200 OK	Request successful
400 Bad Request	Invalid input parameters
404 Not Found	Resource doesn't exist
500 Internal Server Error	Server-side issue

Case Study: How Twitter Uses Query Parameters & Request Bodies

Challenges Faced by Twitter

✓ **Handling millions of search queries efficiently**.
✓ **Processing user-generated content securely**.

Solutions Implemented

✓ **Uses query parameters** for **search filters (hashtags, location, time range)**.
✓ **Validates request bodies** before **posting tweets and comments**.
✓ **Implements rate limiting** to prevent API abuse.

🔹 Key Takeaways from Twitter's API Strategy:

✓ **Query parameters enhance search capabilities**.
✓ **Proper request validation prevents abuse and errors**.
✓ **Express.js simplifies API request handling**.

📝 Exercise

✅ Modify the /users route to allow **filtering by age and gender** using query parameters.

✅ Create a POST API route to **add a new product** using request body data.

✅ Implement **error handling for missing request fields** in a registration API.

✅ Build a **GET API with URL parameters** to fetch user details dynamically.

Conclusion

✓ Query parameters allow filtering & searching API results.

✓ Request bodies are used for sending structured data in POST/PUT requests.

✓ Express.js simplifies handling API requests with req.query and req.body.

✓ Best practices like input validation & proper status codes ensure API security.

IMPLEMENTING ERROR HANDLING AND LOGGING IN EXPRESS.JS

CHAPTER 1: INTRODUCTION TO ERROR HANDLING & LOGGING

1.1 What is Error Handling?

Error handling in Express.js ensures that the application gracefully **handles failures** instead of crashing. It provides meaningful **error messages** and logs errors for debugging.

- **Why is Error Handling Important?**

✓ Prevents **application crashes**.
✓ Provides **useful error messages** to developers.
✓ Improves **security by hiding system details**.

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

Type	Description	Example
Synchronous Errors	Errors in synchronous code	JSON.parse(undefined)
Asynchronous Errors	Errors in async operations	await fetchData() with a bad API
Middleware Errors	Errors from custom middleware	Validation failures
Route Errors	Errors due to incorrect paths or logic	Cannot GET /unknown-route

CHAPTER 2: IMPLEMENTING ERROR HANDLING IN EXPRESS.JS

2.1 Using Try-Catch for Synchronous Errors

📌 Example: Basic Try-Catch in a Route

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

app.get("/", (req, res) => {
  try {
    throw new Error("Something went wrong!");
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

app.listen(5000, () => console.log("Server running on port 5000"));
```

✓ **Catches synchronous errors** and returns a proper response.

2.2 Handling Errors in Asynchronous Code

📌 Example: Using Try-Catch in an Async Route

```
app.get("/async-error", async (req, res) => {
  try {
    const data = await fetchData(); // Assume this fails
    res.json(data);
```

```
  } catch (error) {
    res.status(500).json({ message: "Failed to fetch data" });
  }
});
```

✓ Ensures that **async/await errors** are handled properly.

2.3 Using Express Error-Handling Middleware

📌 **Example: Creating a Global Error Handler**

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

✓ Automatically handles **all application errors**.

2.4 Creating Custom Error Classes

📌 **Example: Defining a Custom Error Class**

```
class AppError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.statusCode = statusCode;
  }
}
```

```
app.get("/custom-error", (req, res, next) => {
  next(new AppError("Custom error occurred", 400));
});
```

✓ Enables **custom error handling** for different cases.

CHAPTER 3: IMPLEMENTING LOGGING IN EXPRESS.JS

3.1 Why Use Logging?

Logging helps developers **track errors, monitor application health, and debug issues** effectively.

- **Common Logging Levels:**

Level	Description	Example
Info	General application logs	"User logged in"
Warning	Unexpected behavior, but app works	"High response time detected"
Error	Critical issues that need fixing	"Database connection failed"

3.2 Logging with console.log() (Basic Logging)

📌 **Example: Logging Requests in Middleware**

```
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url} at ${new Date().toISOString()}`);
```

```
  next();
});
```

✓ Prints **request logs** in the terminal.

3.3 Using Winston for Advanced Logging

📌 Step 1: Install Winston

```
npm install winston
```

📌 Step 2: Configure Winston Logger (logger.js)

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

const logger = winston.createLogger({
  level: "info",
  transports: [
    new winston.transports.Console(),
    new winston.transports.File({ filename: "error.log", level: "error" }),
  ],
});

module.exports = logger;
```

📌 Step 3: Use Logger in Express (server.js)

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

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

✓ **Stores errors in error.log** for debugging.

3.4 Logging API Requests with Morgan

📌 **Step 1: Install Morgan**

```
npm install morgan
```

📌 **Step 2: Configure Morgan as Middleware**

```
const morgan = require("morgan");
app.use(morgan("combined")); // Logs requests in Apache format
```

✓ Provides **detailed logs** for every API request.

CHAPTER 4: COMBINING ERROR HANDLING & LOGGING FOR PRODUCTION

4.1 Best Practices for Error Handling

✓ Always return **meaningful error messages**.
✓ Use **different status codes** (400, 500, etc.).
✓ Implement **global error-handling middleware**.
✓ Log **critical errors** for debugging.

4.2 Best Practices for Logging

✓ Use **Winston or Morgan** for structured logging.
✓ Store logs in **files instead of console**.
✓ Use **error levels** to separate warnings, errors, and debug logs.

Case Study: How PayPal Uses Error Handling & Logging

Challenges Faced by PayPal

✓ Handling **millions of transactions per second**.
✓ Preventing **server crashes due to unhandled errors**.

Solutions Implemented

✓ Used **global error-handling middleware** to catch all errors.
✓ Implemented **Winston for structured logging** and debugging.
✓ Used **asynchronous logging services** to monitor system health.

- **Key Takeaways from PayPal's Strategy:**

✓ **Centralized error handling improves reliability.**
✓ **Logging helps detect system failures before they impact users.**
✓ **Automated alerts notify developers about critical errors.**

📝 Exercise

✅ Implement a **global error handler** in Express.js.
✅ Use **Winston to log errors** to a file.
✅ Use **Morgan to log API requests**.
✅ Create a **custom error class** for handling different error types.

Conclusion

✓ Error handling prevents crashes and improves app stability.
✓ Middleware helps centralize error handling across the app.
✓ Winston and Morgan enable structured logging and debugging.
✓ A well-implemented logging system helps track and fix issues efficiently.

ASSIGNMENT:

BUILD A BASIC EXPRESS.JS API WITH ROUTE HANDLING AND MIDDLEWARE

Assignment Solution: Build a Basic Express.js API with Route Handling and Middleware

Step 1: Setting Up the Express.js Project

1.1 Create a New Project Folder

📌 Open the terminal and run:

```
mkdir express-api
cd express-api
```

✓ This creates and navigates into the **project directory**.

1.2 Initialize a Node.js Project

📌 Initialize the project with package.json:

```
npm init -y
```

✓ Generates a package.json file for managing dependencies.

1.3 Install Express.js

📌 Install Express and additional middleware:

```
npm install express morgan cors dotenv
```

✓ **Express** → Core framework for API development.
✓ **Morgan** → Logs HTTP requests for debugging.
✓ **CORS** → Enables Cross-Origin Resource Sharing.
✓ **Dotenv** → Manages environment variables.

Step 2: Creating the Express Server

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

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

const app = express();

// Middleware
app.use(express.json()); // Parses incoming JSON requests
app.use(morgan("dev")); // Logs requests
app.use(cors()); // Enables CORS for frontend applications

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

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

✓ **Parses JSON data** with express.json().
✓ **Logs requests** using morgan("dev").
✓ **Enables CORS** for frontend interaction.
✓ **Starts the server** on port 5000.

📌 **Run the Server:**

node server.js

✓ Visit http://localhost:5000/ to see:

Welcome to the Express API!

Step 3: Implementing Route Handling

3.1 Creating API Routes

📌 **Modify server.js to include route handling:**

```
const users = [
  { id: 1, name: "Alice" },
  { id: 2, name: "Bob" }
];

// Get All Users
app.get("/api/users", (req, res) => {
  res.json(users);
});

// Get a User by ID
```

```
app.get("/api/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);
});

// Create a New User
app.post("/api/users", (req, res) => {
  const { name } = req.body;
  if (!name) return res.status(400).json({ message: "Name is
required" });

  const newUser = { id: users.length + 1, name };
  users.push(newUser);
  res.status(201).json(newUser);
});

// Update User
app.put("/api/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;
```

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

// Delete User
app.delete("/api/users/:id", (req, res) => {
  const index = users.findIndex(u => u.id === parseInt(req.params.id));
  if (index === -1) return res.status(404).json({ message: "User not found" });

  users.splice(index, 1);
  res.json({ message: "User deleted successfully" });
});
```

✓ Implements **CRUD operations** using different routes.
✓ Uses **route parameters (:id)** to target specific users.
✓ **Validates user input** before processing requests.

Step 4: Creating Custom Middleware

4.1 Logging Middleware

📌 **Create a file middlewares/logger.js:**

```
const logger = (req, res, next) => {
  console.log(`${req.method} request made to ${req.url}`);
  next();
```

```
};


module.exports = logger;
```

✓ Logs **each incoming request method and URL**.

📌 **Use Middleware in server.js:**

```
const logger = require("./middlewares/logger");

app.use(logger);
```

✓ Now, every request **logs to the console**.

4.2 Authentication Middleware

📌 **Create middlewares/auth.js:**

```
const authMiddleware = (req, res, next) => {
  const token = req.headers["authorization"];
  if (!token) return res.status(401).json({ message: "Unauthorized" });


  // Simulating token verification (for actual use, implement JWT)
  if (token !== "secure-token") return res.status(403).json({ message: "Invalid token" });


  next();
};
```

```
module.exports = authMiddleware;
```

✓ **Blocks access to protected routes** without an authorization token.

📌 **Apply Middleware to a Secure Route in server.js:**

```
const authMiddleware = require("./middlewares/auth");


app.get("/api/secure", authMiddleware, (req, res) => {
  res.json({ message: "Access granted to secure route!" });
});
```

✓ **Requests to /api/secure now require a token** in headers:

```
Authorization: secure-token
```

Step 5: Handling Errors in Express.js

5.1 Handling 404 Routes

📌 **Catch Unhandled Routes in server.js:**

```
app.use((req, res, next) => {
  res.status(404).json({ message: "Route not found" });
});
```

✓ Returns a **404 error for invalid endpoints**.

5.2 Global Error Handling Middleware

📌 **Create middlewares/errorHandler.js:**

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

module.exports = errorHandler;
```

📌 **Apply Error Handling Middleware in server.js:**

```
const errorHandler = require("./middlewares/errorHandler");
app.use(errorHandler);
```

✓ **Prevents app crashes by handling unexpected errors.**

Step 6: Testing the API Using Postman or cURL

6.1 Testing the API Routes

HTTP Method	Endpoint	Expected Response
GET	/api/users	List of users
GET	/api/users/1	User with ID 1
POST	/api/users	New user created
PUT	/api/users/1	Updated user details
DELETE	/api/users/1	User deleted successfully
GET	/api/secure	401 Unauthorized

📌 **Using cURL to Test API**

curl -X GET http://localhost:5000/api/users

curl -X POST -H "Content-Type: application/json" -d '{"name": "Charlie"}' http://localhost:5000/api/users

curl -X DELETE http://localhost:5000/api/users/2

✓ Confirms the **API functions correctly**.

Conclusion & Summary

✓ **Created a basic Express.js API** with route handling.
✓ **Implemented middleware** for logging and authentication.
✓ **Added error handling** for invalid routes and unexpected issues.
✓ **Tested the API using Postman & cURL.**