Independent
Skill Development
Mission

Independent
Skill Development NXT
Mission
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

BUILDING APIS WITH NODE.JS & EXPRESS.JS (WEEKS 5-6)

SETTING UP AN HTTP SERVER WITH THE BUILT-IN HTTP MODULE

CHAPTER 1: INTRODUCTION TO THE HTTP MODULE

1.1 Understanding the http Module

Node.js provides a built-in **http module** that allows developers to create web servers without needing external dependencies. This module is essential for building **RESTful APIs, handling client requests, and serving web pages dynamically**.

Unlike traditional web servers like Apache or Nginx, the Node.js HTTP server is **lightweight, event-driven, and non-blocking**, making it highly efficient for handling multiple requests simultaneously.

Key Features of the http Module:
✓ **Built-in and requires no external installation**.
✓ **Non-blocking I/O** for handling multiple requests efficiently.
✓ **Easy to set up and configure** for various types of web applications.

Chapter 2: Creating a Basic HTTP Server

2.1 Writing a Simple HTTP Server

To create a basic web server, use the http.createServer() method.

Example: Creating a Basic HTTP Server

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


const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, World!');
});


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

2.2 Explanation of Code

- **require('http')** – Imports the built-in http module.
- **http.createServer()** – Creates an HTTP server that listens for requests.
- **req (Request Object)** – Contains details about the incoming request (URL, method, headers, etc.).
- **res (Response Object)** – Used to send data back to the client.

- **res.writeHead(200, { 'Content-Type': 'text/plain' })** – Sets the HTTP status code (200 OK) and response type (text/plain).
- **res.end('Hello, World!')** – Sends the response and closes the connection.
- **server.listen(3000, callback)** – The server listens for requests on **port 3000**.

2.3 Running the Server

1. Save the file as server.js.
2. Open a terminal and run:
3. node server.js
4. Open a web browser and visit:
5. http://localhost:3000
6. You should see:
7. Hello, World!

Chapter 3: Handling Different HTTP Requests

3.1 Understanding HTTP Methods

A web server needs to handle different types of HTTP requests. The most commonly used request methods include:

- **GET** – Retrieve data from the server.
- **POST** – Send data to the server.
- **PUT** – Update existing data.
- **DELETE** – Remove data from the server.

3.2 Handling Different Requests in the HTTP Server

Example: Handling GET and POST Requests

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


const server = http.createServer((req, res) => {
  if (req.method === 'GET') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('This is a GET request');
  } else if (req.method === 'POST') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('This is a POST request');
  } else {
    res.writeHead(405, { 'Content-Type': 'text/plain' });
    res.end('Method Not Allowed');
  }
});


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

3.3 Explanation of Request Handling

- **req.method** – Identifies the HTTP request type.
- **Checks for GET or POST requests** and sends appropriate responses.

- **For unsupported methods**, returns a 405 Method Not Allowed error.

To test a POST request, use **cURL** in the terminal:

```
curl -X POST http://localhost:3000
```

Chapter 4: Serving HTML Files Using HTTP Server

4.1 Why Serve HTML Files?

Instead of returning plain text, an HTTP server can serve full **HTML pages**, making it more useful for web applications.

4.2 Example: Serving an HTML File

Create a file named index.html:

```
<!DOCTYPE html>
<html>
<head>
  <title>Node.js Server</title>
</head>
<body>
  <h1>Welcome to My Node.js Server!</h1>
</body>
</html>
```

Modify the server to read and serve this HTML file:

```
const http = require('http');
const fs = require('fs');
const server = http.createServer((req, res) => {
  fs.readFile('index.html', (err, data) => {
    if (err) {
      res.writeHead(500, { 'Content-Type': 'text/plain' });
      res.end('Error loading page');
    } else {
      res.writeHead(200, { 'Content-Type': 'text/html' });
      res.end(data);
    }
  });
});

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

4.3 Explanation of Serving HTML

- **fs.readFile('index.html', callback)** – Reads the file asynchronously.
- **If an error occurs (err), returns 500 Internal Server Error**.
- **If successful, serves the HTML file with a 200 OK status**.

Case Study: How Netflix Uses Node.js HTTP Servers for High Performance

Background

Netflix, a leading streaming platform, serves millions of users worldwide. Initially, they faced **scalability and speed issues** due to traditional web servers.

Challenges

- **Slow response times** caused by blocking operations.
- **High server load** due to inefficient request handling.
- **Scalability issues** as user demand increased.

Solution: Implementing Node.js HTTP Servers

Netflix adopted **Node.js-based HTTP servers**, leading to:

✓ **70% faster API response times** due to non-blocking operations.
✓ **Better scalability**, allowing seamless streaming for millions of users.
✓ **Reduced server costs** by efficiently handling concurrent requests.

This case study highlights the **efficiency and scalability** of using Node.js HTTP servers in real-world applications.

Exercise

1. Modify the basic HTTP server to return "Welcome to My Server!" instead of "Hello, World!".
2. Write a server that serves a JSON response:
3. { "message": "Hello from JSON API!" }

Conclusion

In this section, we explored:

✓ **How to set up a basic HTTP server using Node.js**.

✓ **Handling different HTTP methods like GET and POST**.

✓ **Serving HTML files dynamically using fs.readFile()**.

✓ **How large-scale applications like Netflix use Node.js HTTP servers for performance and scalability**.

HANDLING GET, POST, PUT, DELETE REQUESTS IN NODE.JS

CHAPTER 1: INTRODUCTION TO HTTP METHODS IN NODE.JS

1.1 Understanding HTTP Methods

The **Hypertext Transfer Protocol (HTTP)** is the foundation of communication on the web. When a client (such as a web browser or mobile app) interacts with a server, it sends HTTP requests, and the server responds accordingly.

The four most commonly used HTTP methods in web development are:

- **GET** – Retrieve data from a server.
- **POST** – Send new data to the server.
- **PUT** – Update existing data on the server.
- **DELETE** – Remove data from the server.

These HTTP methods are essential when designing **RESTful APIs** in **Node.js**, allowing applications to perform CRUD (Create, Read, Update, Delete) operations efficiently.

CHAPTER 2: SETTING UP AN HTTP SERVER IN NODE.JS

2.1 Creating a Simple HTTP Server

Node.js provides the built-in **http module** to create an HTTP server and handle requests.

Example: Creating a Basic HTTP Server

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


const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, World!');
});


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

- The http.createServer() method creates an HTTP server.
- The callback function **handles incoming requests** and sends responses.
- The server listens on **port 3000** for connections.

Chapter 3: Handling GET Requests in Node.js

3.1 Understanding GET Requests

A **GET request** is used to retrieve data from a server. In RESTful APIs, GET requests typically return JSON data.

Example: Handling a GET Request

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

const url = require('url');
const server = http.createServer((req, res) => {
  const parsedUrl = url.parse(req.url, true);


  if (req.method === 'GET' && parsedUrl.pathname === '/hello') {
    res.writeHead(200, { 'Content-Type': 'application/json' });
    res.end(JSON.stringify({ message: 'Hello, World!' }));
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Not Found');
  }
});


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

- The url.parse() method extracts the **path and query parameters** from the request URL.
- The server responds with a JSON message when a GET request is made to **/hello**.
- If the request does not match any route, the server returns a **404 Not Found** error.

CHAPTER 4: HANDLING POST REQUESTS IN NODE.JS

4.1 Understanding POST Requests

A **POST request** is used to send **new data** to the server. It is commonly used for:

✓ Submitting forms
✓ Creating new database records
✓ Uploading files

Example: Handling a POST Request

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


const server = http.createServer((req, res) => {
  if (req.method === 'POST' && req.url === '/submit') {
    let body = '';


    req.on('data', chunk => {
      body += chunk.toString(); // Convert buffer to string
    });


    req.on('end', () => {
      res.writeHead(200, { 'Content-Type': 'application/json' });
      res.end(JSON.stringify({ message: 'Data received', data: body }));
    });
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Not Found');
  }
});


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

✓ The **req.on('data') event** collects chunks of data from the request body.
✓ The **req.on('end') event** processes the full body once the request is complete.
✓ The server responds with a JSON message confirming the received data.

Chapter 5: Handling PUT Requests in Node.js

5.1 Understanding PUT Requests

A **PUT request** is used to **update** existing data on the server. It is commonly used for:

✓ Updating user profiles
✓ Modifying database records
✓ Changing application settings

Example: Handling a PUT Request

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


const server = http.createServer((req, res) => {
  if (req.method === 'PUT' && req.url === '/update') {
    let body = '';


    req.on('data', chunk => {
      body += chunk.toString();
    });

    req.on('end', () => {
      res.writeHead(200, { 'Content-Type': 'application/json' });
      res.end(JSON.stringify({ message: 'Data updated',
updatedData: body }));
    });
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Not Found');
  }
});

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

✓ The **PUT method** is used when updating existing records rather than creating new ones.
✓ The request body contains the **new data** to be updated.

Chapter 6: Handling DELETE Requests in Node.js

6.1 Understanding DELETE Requests

A **DELETE request** removes data from a server. It is commonly used for:

✓ Deleting user accounts
✓ Removing records from a database
✓ Erasing files from a server

Example: Handling a DELETE Request

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


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

  if (req.method === 'DELETE' && req.url === '/delete') {

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

    res.end(JSON.stringify({ message: 'Resource deleted successfully' }));

  } else {

    res.writeHead(404, { 'Content-Type': 'text/plain' });

    res.end('Not Found');

  }

});


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

✓ The **DELETE method** is used when removing specific resources.
✓ In real-world applications, the request may include **authentication and validation** before deletion.

Case Study: How an E-Commerce Website Optimized API Performance Using Node.js

Background

An online e-commerce platform struggled with **slow API response times**, affecting product listings, order processing, and customer interactions.

Challenges

- **High server load** due to inefficient request handling.
- **Inconsistent data updates** across multiple endpoints.
- **Slow checkout process**, leading to abandoned carts.

Solution: Implementing a RESTful API with Node.js

The development team optimized API performance by:

✔ **Implementing efficient request handlers** for GET, POST, PUT, DELETE.
✔ **Reducing unnecessary database queries** to improve response times.
✔ **Using asynchronous programming** to handle concurrent requests efficiently.

Results

- **50% faster API responses**, improving product browsing.
- **Reduced server load**, handling thousands of concurrent users.
- **Smoother checkout process**, increasing completed orders.

This case study highlights the importance of **proper request handling** in optimizing server performance.

Exercise

1. Write a Node.js script that handles a **GET request** at /greet and responds with "Hello, User!".
2. Modify the script to handle a **POST request** at /register that logs the request body.
3. Extend the server to support **PUT and DELETE requests** for updating and deleting user data.

Conclusion

In this section, we explored:

✓ **How to handle GET, POST, PUT, and DELETE requests in Node.js**.
✓ **The role of HTTP methods in designing RESTful APIs**.
✓ **Real-world examples of processing request data efficiently**.

UNDERSTANDING REQUEST-RESPONSE CYCLES

CHAPTER 1: INTRODUCTION TO THE REQUEST-RESPONSE CYCLE

1.1 What is the Request-Response Cycle?

The **request-response cycle** is a fundamental concept in web development that defines how a client (such as a web browser or mobile app) communicates with a server. Whenever a user interacts with a website—whether by clicking a link, submitting a form, or making an API request—a request is sent to the server, which processes the request and returns a response.

The request-response cycle follows these key steps:

1. **Client sends a request** – A user initiates a request through a browser, API, or application.
2. **Server receives and processes the request** – The server determines how to handle the request (e.g., retrieving data, storing information).
3. **Server sends a response** – The server sends back a response containing the requested data, an error message, or an appropriate status.
4. **Client receives and processes the response** – The client displays or processes the received information.

Understanding this cycle is crucial for building efficient backend applications, especially in **Node.js**, where handling multiple requests simultaneously is a key strength.

CHAPTER 2: COMPONENTS OF A REQUEST-RESPONSE CYCLE

2.1 Understanding HTTP Requests

A request sent from a client to a server follows the **HTTP protocol**, consisting of:

- **Method** – Defines the type of request (GET, POST, PUT, DELETE, etc.).
- **Headers** – Provide metadata such as content type and authentication tokens.
- **Body** – (Optional) Contains data for requests like form submissions or API interactions.

Example: Sending a Request Using Fetch API

```
fetch('https://api.example.com/data', {
  method: 'GET',
  headers: {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer token123'
  }
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Error:', error));
```

- The fetch function sends a GET request to an API.
- Headers include a content type and authentication token.
- The response is converted to JSON and logged.

2.2 Understanding HTTP Responses

A response from a server contains:

- **Status Code** – Indicates success, failure, or redirection (e.g., 200 OK, 404 Not Found).
- **Headers** – Provide metadata about the response.
- **Body** – Contains data returned from the server.

Example: Server Responding to a Request in Node.js

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


const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, world!');
});


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

- The server listens for incoming requests.
- It responds with "Hello, world!" and a 200 OK status.

CHAPTER 3: LIFECYCLE OF A REQUEST-RESPONSE IN NODE.JS

3.1 Handling Requests in Node.js

Node.js handles requests asynchronously using event-driven programming. The request lifecycle consists of:

1. **Receiving the request** – The server detects a new request.
2. **Processing the request** – Node.js executes functions to determine how to handle it.
3. **Fetching or modifying data** – The server interacts with a database or file system.
4. **Sending the response** – The processed data or message is sent back to the client.

Example: Handling Different HTTP Methods in Node.js

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


const server = http.createServer((req, res) => {
  if (req.method === 'GET') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Received a GET request');
  } else if (req.method === 'POST') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Received a POST request');
  } else {
    res.writeHead(405, { 'Content-Type': 'text/plain' });
    res.end('Method Not Allowed');
  }
});
```

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

- The server **differentiates requests based on the HTTP method** (GET, POST).
- It **sends appropriate responses** for each method.

3.2 Middleware in Request-Response Handling

Middleware functions in **Express.js** allow processing requests before they reach the final handler.

Example: Using Middleware in Express.js

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

app.use((req, res, next) => {
  console.log(`Request received: ${req.method} ${req.url}`);
  next();
});

app.get('/', (req, res) => {
  res.send('Hello, world!');
});
```

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

- The middleware logs **every request** received by the server.
- next() ensures the request proceeds to the next function.

Case Study: How Twitter Handles Billions of Requests Efficiently

Background

Twitter, a social media platform, processes **millions of requests per second** globally.

Challenges

- Handling **massive concurrent user requests**.
- Ensuring **real-time updates** for tweets, likes, and retweets.
- **Minimizing server response times**.

Solution: Optimizing the Request-Response Cycle

Twitter optimized its request-response cycle using:

✓ **Load Balancers** – Distribute requests across multiple servers to prevent overload.
✓ **Asynchronous Processing (Node.js & Kafka)** – Queues requests for non-blocking execution.
✓ **Caching (Redis & Memcached)** – Stores frequently requested data to **reduce database queries**.

Results

- **50% reduction in server response time**.
- Ability to **handle billions of daily requests** without downtime.
- Improved **user experience with real-time updates**.

This case study highlights how optimizing the request-response cycle ensures **scalability and efficiency** in high-traffic applications.

Exercise

1. What are the four main steps of the request-response cycle?
2. Write a Node.js server that responds to both GET and POST requests.
3. How does Express.js middleware improve request handling?

Conclusion

In this section, we explored:

✓ **The request-response cycle and its importance in web development**.
✓ **How HTTP requests and responses are structured**.
✓ **How Node.js handles requests asynchronously**.

SETTING UP AN EXPRESS.JS PROJECT

CHAPTER 1: INTRODUCTION TO EXPRESS.JS

1.1 What is Express.js?

Express.js is a **minimal, fast, and flexible** web framework for **Node.js** that simplifies the process of building web applications and APIs. It provides a structured way to handle routing, middleware, and HTTP requests, making it a popular choice for building scalable applications.

Key Features of Express.js:

- **Lightweight and Minimalistic** – Express.js does not enforce a specific structure, giving developers flexibility.
- **Middleware Support** – Middleware functions allow easy handling of requests, authentication, logging, and error management.
- **Robust Routing System** – Express provides a simple way to define routes for handling different HTTP methods (GET, POST, PUT, DELETE).
- **Seamless Integration with Databases** – Works well with **MongoDB, PostgreSQL, MySQL**, and other databases.
- **Compatible with Template Engines** – Supports **EJS, Pug, and Handlebars** for rendering dynamic HTML pages.

Example: Why Use Express.js Over Native Node.js?

Native Node.js Without Express.js

const http = require('http');

```
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, World!');
});

server.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

Using Express.js (Simplified Version)

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

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

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

- Express.js **reduces boilerplate code** and simplifies request handling.
- It provides a **built-in routing mechanism**, making development easier.

CHAPTER 2: INSTALLING EXPRESS.JS

2.1 Prerequisites

Before setting up an Express.js project, ensure you have:

- **Node.js installed** – Check using:
- node -v
- **NPM (Node Package Manager) installed** – Verify with:
- npm -v

2.2 Initializing a New Node.js Project

Follow these steps to create a new Express.js project:

1. **Create a project directory:**
2. mkdir express-app
3. cd express-app
4. **Initialize a Node.js project:**
5. npm init -y
    - This creates a package.json file to manage project dependencies.
6. **Install Express.js:**
7. npm install express
    - This installs Express.js and adds it to package.json.

CHAPTER 3: CREATING THE FIRST EXPRESS.JS SERVER

3.1 Setting Up the Basic Server

1. **Create an index.js file:**
2. touch index.js
3. **Open index.js and add the following code:**
4. const express = require('express');
5. const app = express();
6.
7. app.get('/', (req, res) => {
8.   res.send('Welcome to Express.js!');
9. });
10.
11. app.listen(3000, () => {
12.   console.log('Server running on http://localhost:3000');
13. });
14. **Run the server:**
15. node index.js
16. **Open a browser and visit:**
17. http://localhost:3000
    - You should see "Welcome to Express.js!" displayed.

Chapter 4: Adding Middleware and Routes

4.1 Using Middleware in Express.js

Middleware functions allow us to process requests before they reach the route handlers.

Example: Logging Middleware

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

- This middleware **logs** request details before passing control to the next handler.

4.2 Defining Multiple Routes

We can define multiple routes to handle different types of requests.

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


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

Now, visiting http://localhost:3000/about or http://localhost:3000/contact will return the respective responses.

Case Study: How LinkedIn Uses Express.js for Backend Services

Background

LinkedIn, a professional networking platform, needed a scalable backend to handle **millions of user profiles, job listings, and connections**.

Challenges

- Performance issues with their existing **monolithic backend**.
- High **API response times** due to increasing traffic.
- Difficulty in **scaling services** while maintaining speed.

Solution: Adopting Express.js

LinkedIn adopted **Express.js for their backend microservices**, resulting in:

- **40% Faster API Responses** – By optimizing request handling with Express middleware.
- **Better Load Balancing** – Express's lightweight structure allowed seamless integration with **NGINX and load balancers**.
- **Scalability** – Migrating to an **Express.js-based microservices architecture** enabled LinkedIn to scale efficiently.

Results

- **Reduced API latency**, ensuring a faster user experience.
- **Easier debugging and maintenance** due to modular Express.js code.
- **Scalable backend**, supporting millions of concurrent users.

This case study showcases how **Express.js is ideal for high-traffic web applications**.

Exercise

1. What command is used to install Express.js?
2. Write a simple Express.js server that responds with "Hello, Express!".
3. What is middleware in Express.js, and how does it work?

Conclusion

In this section, we explored:

✓ **What Express.js is and why it's used for web development**.
✓ **How to install and set up an Express.js project**.
✓ **How to create a basic Express.js server with routing and middleware**.

ROUTING AND MIDDLEWARE IN EXPRESS.JS

CHAPTER 1: INTRODUCTION TO ROUTING AND MIDDLEWARE

1.1 Understanding Routing in Express.js

In web development, **routing** refers to defining **URLs (routes)** that a server should respond to. In **Express.js**, routing determines how the application responds to **client requests** for specific URLs using different HTTP methods such as **GET, POST, PUT, DELETE**.

Key Features of Routing in Express.js:

✓ Defines how the server responds to different URLs.
✓ Handles various HTTP methods for CRUD operations.
✓ Supports route parameters and query strings.
✓ Allows modular organization using route handlers.

1.2 Understanding Middleware in Express.js

Middleware functions in Express.js are functions that execute **before sending the final response** to the client. They have access to:

- The **request object (req)**.
- The **response object (res)**.
- The **next() function**, which passes control to the next middleware.

Key Features of Middleware in Express.js:

✓ Used for logging, authentication, request parsing, and error handling.
✓ Executes sequentially in the order they are defined.
✓ Enhances modularity and maintainability in applications.

Chapter 2: Setting Up Routing in Express.js

2.1 Creating a Basic Express Server with Routes

First, install **Express.js** if not already installed:

```
npm init -y

npm install express
```

Then, create a file server.js and add the following code:

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

const app = express();


// Define a basic route

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

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

});


// Start the server

app.listen(3000, () => {

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

});
```

2.2 Explanation of Code

- **express()** – Initializes an Express application.
- **app.get('/', callback)** – Defines a route for GET /.

- **req (Request Object)** – Contains request details like headers, URL, and parameters.
- **res (Response Object)** – Used to send responses to the client.
- **app.listen(3000, callback)** – Starts the server on port 3000.

2.3 Running the Express Server

1. Save the file as server.js.
2. Start the server:
3. node server.js
4. Open a browser and visit:
5. http://localhost:3000
6. The response **"Welcome to the Express.js Server!"** should be displayed.

Chapter 3: Handling Multiple Routes and HTTP Methods

3.1 Defining Routes for Different HTTP Methods

Express.js allows handling different request types:

```
app.get('/about', (req, res) => {
  res.send('This is the About page.');
});


app.post('/submit', (req, res) => {
  res.send('Form submitted successfully.');
});
```

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


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

3.2 Explanation of Route Handling

- **app.get('/about', callback)** – Handles GET requests to /about.
- **app.post('/submit', callback)** – Handles form submissions.
- **app.put('/update', callback)** – Handles data updates.
- **app.delete('/delete', callback)** – Handles data deletions.

3.3 Handling Route Parameters

Route parameters allow **dynamic values** in URLs.

Example: Using Route Parameters

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

Test It in the Browser:

Visit:

http://localhost:3000/user/123

Response:

User ID: 123

Chapter 4: Using Middleware in Express.js

4.1 Creating a Simple Middleware Function

Middleware functions execute **before the request reaches the final route handler**.

Example: Logging Middleware

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

app.use(loggerMiddleware);
```

4.2 Explanation of Middleware Execution

- **Middleware runs before handling a request**.
- **next()** moves the request to the next middleware or route.

4.3 Built-in Middleware in Express.js

Express provides **built-in middleware** for common tasks:

- **express.json()** – Parses incoming JSON requests.
- **express.urlencoded({ extended: true })** – Parses form data.
- **express.static('public')** – Serves static files like images and stylesheets.

Example: Using JSON Middleware

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


app.post('/data', (req, res) => {
  res.send(`Received data: ${JSON.stringify(req.body)}`);
});
```

Case Study: How Airbnb Uses Express.js Routing and Middleware

Background

Airbnb, a leading online marketplace for rental properties, required a robust **backend** to handle millions of users searching for accommodations.

Challenges

- **Efficiently managing thousands of API routes.**
- **Ensuring security through authentication middleware.**
- **Handling complex data processing in real-time.**

Solution: Implementing Express.js Routing & Middleware

✓ **Organized API routes using Express.js routing**.
✓ **Implemented authentication middleware for security**.
✓ **Used request logging middleware to monitor API activity**.

This solution allowed Airbnb to **scale seamlessly**, handling **millions of user requests** daily.

Exercise

1. Modify the Express server to handle a **POST** request at /contact that returns "Contact form received".
2. Write a middleware function that logs "Middleware executed!" before any request is processed.

Conclusion

In this section, we explored:

✓ **How to define routes in Express.js**.

✓ **Handling different HTTP methods like GET, POST, PUT, and DELETE**.

✓ **Using route parameters for dynamic URLs**.

✓ **Implementing middleware for logging, authentication, and data parsing**.

✓ **How Airbnb benefits from Express.js routing and middleware**.

ERROR HANDLING AND LOGGING IN APIS

CHAPTER 1: INTRODUCTION TO ERROR HANDLING IN NODE.JS APIS

1.1 Understanding Error Handling in APIs

Error handling is a crucial part of **API development**. A well-structured API should be able to:

- **Detect errors** before they cause system failures.
- **Handle unexpected issues** gracefully without crashing the server.
- **Provide meaningful error messages** to users and developers.
- **Log errors** for debugging and troubleshooting.

Common types of errors in **Node.js APIs** include:

✔ **Operational Errors** – Issues like missing files, network failures, or invalid user input.
✔ **Programming Errors** – Bugs in the code, such as undefined variables or syntax mistakes.
✔ **Unhandled Rejections** – Failures in promises that are not caught properly.

To manage these errors effectively, **Node.js APIs must implement structured error handling techniques**.

CHAPTER 2: HANDLING ERRORS USING TRY-CATCH IN APIS

2.1 Using Try-Catch for Synchronous Code

The simplest way to handle errors in JavaScript is with try...catch. This works well for **synchronous operations**.

Example: Basic Try-Catch Error Handling

```
function divideNumbers(a, b) {
  try {
    if (b === 0) throw new Error("Cannot divide by zero!");
    return a / b;
  } catch (error) {
    console.error("Error:", error.message);
  }
}


console.log(divideNumbers(10, 2)); // Outputs: 5
console.log(divideNumbers(10, 0)); // Outputs: Error: Cannot divide by zero!
```

✓ If division by zero is attempted, an error is thrown and caught gracefully.
✓ The program **does not crash**, and a meaningful error message is logged.

2.2 Using Try-Catch for Asynchronous Code

try...catch does not work with asynchronous operations unless used inside an **async function**.

Example: Handling Errors in Async/Await

```
async function fetchData() {
  try {
    let response = await fetch("https://invalid-url.com");
```

```
        let data = await response.json();
        console.log(data);
    } catch (error) {
        console.error("Failed to fetch data:", error.message);
    }
}

fetchData();
```

✓ If the URL is invalid, the error is **caught and logged** instead of crashing the program.

CHAPTER 3: HANDLING API ERRORS IN EXPRESS.JS

3.1 Creating a Custom Error Handler Middleware

Express.js provides built-in support for **middleware**, making it easy to handle errors in a centralized way.

Example: Basic Express.js API with Error Handling

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

// Middleware to handle errors globally
app.use((err, req, res, next) => {
    console.error(err.stack);
    res.status(500).json({ error: "Internal Server Error" });
});


// Sample route that throws an error
app.get('/error', (req, res, next) => {
  try {
    throw new Error("Something went wrong!");
  } catch (err) {
    next(err); // Pass error to middleware
  }
});


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

✓ The **error-handling middleware** ensures all errors are caught and logged.
✓ The next(err) function passes errors to the middleware for centralized processing.

Chapter 4: Logging API Errors in Node.js

4.1 Why Logging is Important?

Logging allows developers to:

✓ **Monitor server activity** in real-time.
✓ **Debug API issues** without stopping the server.
✓ **Track application performance** over time.

Node.js provides various logging methods, such as:

- **console.log()** – Basic logging for development.
- **console.error()** – Logs errors with stack traces.
- **Winston & Morgan** – Advanced logging libraries.

4.2 Using Winston for Structured Logging

Winston is a popular logging library that supports:

✔ **Custom log levels** (info, warning, error).
✔ **File-based logging** for persistent records.
✔ **Logging to external monitoring services** (e.g., AWS, Elasticsearch).

Example: Using Winston for Logging API Errors

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


// Configure logger

const logger = winston.createLogger({

  level: 'error',

  format: winston.format.json(),

  transports: [

    new winston.transports.File({ filename: 'errors.log' })

  ]

});


// Log an error message

logger.error("Database connection failed!");
```

✓ This will log errors to errors.log, allowing developers to analyze issues later.

Chapter 5: Handling Uncaught Errors and Promise Rejections

5.1 Handling Uncaught Exceptions

An **uncaught exception** occurs when an error is thrown but not handled. This can crash the Node.js process.

To prevent crashes, use:

```
process.on('uncaughtException', (err) => {

  console.error("Uncaught Exception:", err.message);

});
```

✓ This catches unexpected errors and logs them instead of stopping the server.

5.2 Handling Unhandled Promise Rejections

If a promise is rejected and there's no .catch() to handle it, use:

```
process.on('unhandledRejection', (err) => {

  console.error("Unhandled Promise Rejection:", err.message);

});
```

✓ This prevents **silent errors**, ensuring all promise failures are logged.

Case Study: How an E-Commerce Platform Improved API Reliability

Background

An online e-commerce platform frequently experienced API crashes, leading to **downtime during peak sales**.

Challenges

- Uncaught errors were **crashing the API server**.
- API response times were slow due to **lack of structured logging**.
- Debugging was difficult as **errors were not properly logged**.

Solution: Implementing Proper Error Handling and Logging

The development team adopted:

✓ **Centralized error-handling middleware** in Express.js.
✓ **Winston logging** to track API failures in real-time.
✓ **Process-level error handlers** to prevent server crashes.

Results

- **50% reduction** in server crashes.
- **Faster debugging** by accessing structured error logs.
- **Increased API uptime**, improving customer experience.

This case study demonstrates how **effective error handling and logging** can **increase API reliability**.

Exercise

1. Modify an Express.js API to include centralized error-handling middleware.
2. Implement Winston logging to record errors in a log file.

3. Write a script that handles an **uncaught exception** and logs it to the console.

Conclusion

In this section, we explored:

✓ **How to handle errors using Try-Catch in synchronous and asynchronous code**.

✓ **How to implement error-handling middleware in Express.js APIs**.

✓ **How to log API errors using Winston for better debugging**.

ASSIGNMENT:

DEVELOP A CRUD-BASED RESTFUL API USING EXPRESS.JS

SOLUTION GUIDE: DEVELOP A CRUD-BASED RESTFUL API USING EXPRESS.JS

Step 1: Set Up the Project

1.1 Install Node.js and Create a Project Folder

Ensure Node.js is installed by checking the version:

```
node -v
```

Then, create a project folder and navigate into it:

```
mkdir express-crud-api

cd express-crud-api
```

1.2 Initialize a Node.js Project

Run the following command to create a package.json file:

```
npm init -y
```

This file keeps track of project dependencies.

1.3 Install Required Dependencies

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

- **Express.js** – A lightweight Node.js framework for handling HTTP requests.
- **Body-parser** – Parses incoming JSON request bodies.
- **Nodemon** (optional) – Restarts the server automatically when files change.

1.4 Create the API File

Run the following command to create the main file:

```
touch index.js
```

Step 2: Create the Express Server

2.1 Import Dependencies and Initialize Express

Open index.js and add the following code:

```
const express = require('express');
const bodyParser = require('body-parser');


const app = express();
const PORT = 3000;


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


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

- **express()** initializes the Express application.
- **bodyParser.json()** allows Express to handle JSON data in requests.
- The server listens on **port 3000**.

Run the server using:

```
node index.js
```

Or use **Nodemon** for automatic restarts:

```
npx nodemon index.js
```

Step 3: Implement CRUD Operations

3.1 Create a Sample In-Memory Database

Since this example doesn't use a real database, we'll store data in an **array**.

Add the following mock data to index.js:

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

3.2 Implement the CRUD Routes

Create (POST Request) – Add a New User

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

✓ Accepts user data from the request body.

✓ Adds the user to the array.

✓ Responds with the newly created user.

Read (GET Request) – Fetch All Users

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

✓ Returns all users in JSON format.

Read (GET Request) – Fetch a Single User by ID

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

✓ Retrieves a user based on the id parameter.
✓ Sends a 404 Not Found response if the user doesn't exist.

Update (PUT Request) – Modify an Existing User

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

  const { name, email } = req.body;
  user.name = name || user.name;
  user.email = email || user.email;
  res.json({ message: 'User updated', user });
});
```

✓ Finds the user by id.
✓ Updates only the provided fields.
✓ Returns the updated user.

Delete (DELETE Request) – Remove a User

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

✓ Removes the user from the list.
✓ Returns a success message.

Step 4: Test the API Using Postman or CURL

4.1 Start the Server

```
node index.js
```

4.2 Test API Endpoints

Create a New User (POST Request)

```
curl -X POST http://localhost:3000/users -H "Content-Type: application/json" -d '{"name": "Charlie", "email": "charlie@example.com"}'
```

Fetch All Users (GET Request)

```
curl -X GET http://localhost:3000/users
```

Fetch a Single User by ID (GET Request)

```
curl -X GET http://localhost:3000/users/1
```

Update a User (PUT Request)

```
curl -X PUT http://localhost:3000/users/1 -H "Content-Type: application/json" -d '{"name": "Alice Johnson"}'
```

Delete a User (DELETE Request)

```
curl -X DELETE http://localhost:3000/users/1
```

Step 5: Improving the API (Optional Enhancements)

✓ **Validate Input Data** – Ensure the user provides valid data before adding/updating.

```
if (!name || !email) {
  return res.status(400).json({ message: 'Name and email are required' });
}
```

✓ **Connect to a Database** – Replace the in-memory array with **MongoDB or PostgreSQL**.
✓ **Add Authentication** – Secure the API using **JWT (JSON Web Tokens)**.
✓ **Use Environment Variables** – Store sensitive data in a .env file.

Conclusion

✓ We built a **RESTful API** using **Express.js** to perform CRUD operations.
✓ We implemented **routes for creating, reading, updating, and deleting users**.

✓ We tested the API using **Postman and CURL**.

✓ We explored **further improvements**, such as database integration and authentication.