Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

TESTING, DEBUGGING & PERFORMANCE OPTIMIZATION (WEEKS 37-42)

DEBUGGING IN CHROME DEVTOOLS

CHAPTER 1: INTRODUCTION TO CHROME DEVTOOLS

1.1 What is Chrome DevTools?

Chrome DevTools is a set of **developer tools built into Google Chrome** that helps debug, inspect, and optimize web applications.

- **Why Use Chrome DevTools?**

✓ Identify **JavaScript errors** in real time.
✓ Debug **HTML & CSS issues** with live editing.
✓ Analyze **network requests** for performance optimization.
✓ Inspect and modify **DOM elements dynamically**.

- **How to Open Chrome DevTools?**

Method	**Shortcut**
Right-click → Inspect	-
Keyboard Shortcut (Windows/Linux)	Ctrl + Shift + I or F12
Keyboard Shortcut (Mac)	Cmd + Option + I

CHAPTER 2: DEBUGGING JAVASCRIPT IN CHROME DEVTOOLS

2.1 Viewing JavaScript Errors in the Console

📌 Open the Console Tab

1. Open DevTools (F12 → **Console** tab).
2. Look for **red error messages**.
3. Click on the error to **see the exact location in the code**.

📌 Example Error:

Uncaught ReferenceError: myVariable is not defined

✓ Occurs when myVariable is used but **not declared**.

2.2 Setting Breakpoints in JavaScript

A **breakpoint** pauses execution at a specific line of code, allowing developers to **inspect variables and execution flow**.

📌 How to Set a Breakpoint:

1. Open DevTools → Go to the **Sources** tab.
2. Open your JavaScript file (from the left panel).
3. Click on the **line number** where you want execution to pause.
4. Refresh the page to hit the breakpoint.

📌 Example: Debugging a Function in DevTools

```
function calculateSum(a, b) {
  let sum = a + b;
```

```
  console.log("Sum:", sum); // Set a breakpoint here
  return sum;
}

calculateSum(5, 10);
```

✓ Execution **pauses at the breakpoint,** allowing inspection of variables.

2.3 Using the Watch Panel to Track Variables

The **Watch panel** allows developers to **monitor specific variables** during debugging.

📌 **How to Use Watch Panel:**

1. Open the **Sources** tab.
2. Click on **"Watch"** → Add a variable name manually.
3. Step through the code to see how the variable changes.

✓ Helps in tracking **variable mutations over time**.

2.4 Stepping Through Code (Debugging Controls)

📌 **Debugging Options in Sources Tab:**

Control	Description	Shortcut
▶️ **Resume**	Continue execution until next breakpoint	F8

⏭ **Step Over**	Move to the next line in the same function	F10
⏬ **Step Into**	Enter into a function call	F11
⏹ **Step Out**	Exit the current function and return	Shift + F11

✓ Helps **trace function calls and execution flow**.

CHAPTER 3: DEBUGGING HTML & CSS IN CHROME DEVTOOLS

3.1 Inspecting and Modifying HTML Elements

📌 **How to Edit HTML in Real-Time:**

1. Open DevTools ⟶ Go to the **Elements** tab.
2. Hover over an element ⟶ Right-click ⟶ **Edit as HTML**.
3. Modify the HTML ⟶ Press Enter to see live changes.

✓ Useful for **testing layout changes without modifying source files**.

3.2 Debugging CSS Issues

📌 **How to Edit CSS in Real-Time:**

1. Open DevTools ⟶ **Elements** tab ⟶ Click on an element.
2. Go to **Styles Panel** (Right Side).
3. Modify **CSS properties live** (color, padding, margins, etc.).

📌 **Example: Changing Background Color in DevTools**

```
body {
  background-color: red;
}
```

✓ Helps test **UI changes instantly**.

3.3 Finding Unused CSS Rules

📌 **Use the Coverage Tool:**

1. Open DevTools → **Run Command (Ctrl + Shift + P)**.
2. Search for **"Show Coverage"**.
3. Click **Start Recording** to see **unused CSS rules**.

✓ Helps **remove unnecessary CSS** for better performance.

CHAPTER 4: DEBUGGING NETWORK REQUESTS

4.1 Monitoring API Calls in the Network Tab

📌 **How to Check API Requests in DevTools:**

1. Open DevTools → Go to **Network** tab.
2. Refresh the page → Look for **XHR or Fetch requests**.
3. Click on a request to view **Response, Headers, and Status Codes**.

✓ Helps **debug failed API requests** and **slow network responses**.

4.2 Simulating Slow Internet Speeds

📌 **How to Test API Performance in Low Network Conditions:**

1. Open DevTools → **Network** tab.
2. Click on **"No Throttling"** dropdown.
3. Select **"Slow 3G" or "Offline"**.

✓ Simulates **real-world network conditions**.

CHAPTER 5: PERFORMANCE OPTIMIZATION USING DEVTOOLS

5.1 Measuring Page Load Time

📌 **Use the Performance Tab:**

1. Open DevTools → **Performance** tab.
2. Click **Start Profiling & Reload Page**.
3. View **Render, Script Execution, and Load Time Graphs**.

✓ Identifies **slow-loading scripts and render-blocking resources**.

5.2 Reducing Render Blocking Resources

📌 **Use the Coverage Tool to Detect Unused JS & CSS**

Ctrl + Shift + P → Show Coverage

✓ Helps **remove unused files** to speed up page loading.

Case Study: How Netflix Uses Chrome DevTools for Debugging

Challenges Faced by Netflix

✓ Optimizing **video streaming performance**.
✓ Debugging **API failures in real-time**.

Solutions Implemented

✓ Used **Network tab** to track API response times.
✓ Used **Performance tab** to measure rendering speeds.
✓ Optimized **CSS & JavaScript** using **Coverage tool**.

🔹 **Key Takeaways from Netflix's Debugging Approach:**

✓ **DevTools help identify API bottlenecks & network failures.**
✓ **Performance profiling improves rendering speeds.**
✓ **Live editing & debugging save development time.**

📝 **Exercise**

✅ Open a website and **identify console errors** using DevTools.
✅ Set a **breakpoint in JavaScript** and track variable changes.
✅ Modify **CSS styles live** in DevTools.
✅ Use the **Network tab** to analyze API requests and responses.

Conclusion

✓ **Chrome DevTools provides powerful debugging capabilities.**
✓ **JavaScript breakpoints help inspect runtime issues.**
✓ **Live HTML/CSS editing improves frontend development.**
✓ **Performance tools optimize load times & resource usage.**

HANDLING ERRORS IN BACKEND & FRONTEND

CHAPTER 1: INTRODUCTION TO ERROR HANDLING

1.1 What is Error Handling?

Error handling is the process of **detecting, managing, and responding** to unexpected issues in a web application. It ensures that applications **continue to function smoothly** even when errors occur.

- **Why is Error Handling Important?**

✓ Prevents applications from **crashing unexpectedly**.
✓ Improves **user experience** by displaying meaningful messages.
✓ Helps developers **debug and troubleshoot issues** efficiently.
✓ Protects **sensitive data** by avoiding exposure of system errors.

- **Types of Errors:**

Error Type	Description	Example
Syntax Errors	Mistakes in code structure	unexpected token in JSON
Runtime Errors	Occur during code execution	null reference exception
Logical Errors	Incorrect implementation of logic	Fetching wrong data
Validation Errors	Invalid user inputs	Email is required

CHAPTER 2: HANDLING ERRORS IN THE BACKEND (NODE.JS + EXPRESS.JS)

2.1 Using Try-Catch for Error Handling

📌 Example: Handling Errors in an Express API Route

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

const app = express();


app.get("/api/data", async (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"));
```

✓ try block attempts execution.
✓ catch block captures **unexpected errors** and sends a response.

2.2 Using Middleware for Centralized Error Handling

📌 Example: Global Error Middleware in Express.js

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

```
  res.status(500).json({ error: "Internal Server Error" });
});
```

✓ Prevents **repetitive try-catch blocks** in every route.
✓ Logs errors **without exposing sensitive details**.

2.3 Handling Validation Errors with express-validator

📌 Step 1: Install express-validator

```
npm install express-validator
```

📌 Step 2: Validate Inputs in an API Route

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

app.post("/register", [
  body("email").isEmail().withMessage("Invalid email format"),
  body("password").isLength({ min: 6 }).withMessage("Password must be at least 6 characters long")
], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });

  res.json({ message: "User registered successfully" });
});
```

✓ Validates **email format and password length**.

✓ Returns **detailed validation errors**.

2.4 Handling Database Errors (MongoDB & Mongoose)

📌 **Example: Catching Mongoose Errors in Express.js**

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


app.post("/users", async (req, res) => {
  try {
    const newUser = new User(req.body);
    await newUser.save();
    res.status(201).json(newUser);
  } catch (error) {
    res.status(400).json({ message: error.message });
  }
});
```

✓ Catches **duplicate email errors** and **schema validation failures**.

CHAPTER 3: HANDLING ERRORS IN THE FRONTEND (REACT.JS)

3.1 Using Try-Catch in API Calls

📌 **Example: Handling Errors in Axios Requests**

```
import { useState, useEffect } from "react";
```

```
import axios from "axios";

function App() {
  const [data, setData] = useState(null);
  const [error, setError] = useState("");

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await
axios.get("http://localhost:5000/api/data");
        setData(response.data);
      } catch (err) {
        setError("Failed to fetch data. Please try again later.");
      }
    };

    fetchData();
  }, []);

  return (
    <div>
      {error ? <p style={{ color: "red" }}>{error}</p> : <p>{data}</p>}
```

```
    </div>
  );
}

export default App;
```

✓ Displays a **friendly error message** instead of crashing.
✓ Uses setError() to **update the UI dynamically**.

3.2 Handling Validation Errors in Forms

📌 **Example: Form Validation in React**

```
import { useState } from "react";

function LoginForm() {
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [error, setError] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();
    if (!email.includes("@")) return setError("Invalid email address");
    if (password.length < 6) return setError("Password must be at least 6 characters");
```

```
    setError(""); // Clear errors if valid
    console.log("Form submitted");
  };

  return (
    <div>
      <form onSubmit={handleSubmit}>
        <input type="email" placeholder="Email" onChange={(e) =>
setEmail(e.target.value)} />
        <input type="password" placeholder="Password"
onChange={(e) => setPassword(e.target.value)} />
        {error && <p style={{ color: "red" }}>{error}</p>}
        <button type="submit">Login</button>
      </form>
    </div>
  );
}

export default LoginForm;
```

✓ Prevents **invalid submissions** before making an API call.
✓ Displays **error messages in the UI**.

3.3 Handling 404 Errors in React (Not Found Page)

📌 Example: Displaying a Custom 404 Page in React Router

```
import { BrowserRouter as Router, Route, Routes, Navigate } from "react-router-dom";


function NotFound() {
  return <h1>404 - Page Not Found</h1>;
}


function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<h1>Home Page</h1>} />
        <Route path="*" element={<NotFound />} />
      </Routes>
    </Router>
  );
}


export default App;
```

✓ Redirects users to a **custom 404 page** for invalid URLs.

CHAPTER 4: LOGGING AND MONITORING ERRORS

4.1 Using Winston for Backend Error Logging

📌 Install Winston:

```
npm install winston
```

📌 Configure Winston for Logging Errors (logger.js)

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

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

module.exports = logger;
```

📌 Use Winston in server.js to Log Errors:

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

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

✓ Logs **server errors into a file (errors.log)**.

Case Study: How Amazon Handles Errors Efficiently

Challenges Faced by Amazon

✓ Handling **millions of product queries** daily.
✓ Preventing **checkout failures during high traffic**.

Solutions Implemented

✓ Used **try-catch blocks** to handle API errors.
✓ Implemented **error tracking with logging tools** (like Winston).
✓ Displayed **meaningful frontend messages instead of raw errors**.

🔹 **Key Takeaways from Amazon's Error Handling Strategy:**

✓ **Backend logging improves debugging efficiency.**
✓ **Proper frontend error messages improve user experience.**
✓ **Centralized error handling prevents crashes.**

📝 Exercise

✅ Implement **global error handling middleware** in Express.js.
✅ Use **try-catch in React API requests** to handle API failures.
✅ Display a **404 error page for undefined routes in React Router**.
✅ Implement **form validation** to prevent invalid user input.

Conclusion

✓ **Error handling improves system stability and user experience.**
✓ **Backend validation ensures only valid data is processed.**
✓ **Frontend error handling prevents UI crashes and improves**

messaging.

✓ Logging helps track issues for debugging and monitoring.

WRITING TESTS WITH JEST & MOCHA

CHAPTER 1: INTRODUCTION TO TESTING IN JAVASCRIPT

1.1 What is Testing?

Testing ensures that a **JavaScript application works correctly** by verifying functions, APIs, and UI components. Jest and Mocha are popular testing frameworks that help developers write **automated tests**.

- **Why is Testing Important?**

✓ Prevents **bugs** before deploying applications.
✓ Ensures **code reliability** and maintainability.
✓ Helps in **continuous integration (CI/CD)**.

- **Types of Testing:**

Type	Purpose	Example
Unit Testing	Tests individual functions/components	Checking a login function
Integration Testing	Tests how different parts work together	Verifying API calls
End-to-End (E2E) Testing	Tests the entire application workflow	User authentication flow

CHAPTER 2: WRITING TESTS WITH JEST

2.1 Installing Jest in a Project

📌 **Step 1: Install Jest**

npm install --save-dev jest

📌 Step 2: Add Jest to package.json

```
"scripts": {
  "test": "jest"
}
```

✓ jest runs **all test files automatically**.

2.2 Writing a Simple Unit Test in Jest

📌 Create a file math.js with a function to test:

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

module.exports = add;
```

📌 Create a test file math.test.js

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

test("adds 2 + 3 to equal 5", () => {
  expect(add(2, 3)).toBe(5);
});
```

📌 Run Jest Tests

npm test

✓ Jest runs **all tests and verifies outputs**.

2.3 Using Matchers in Jest

📌 Example: Common Jest Matchers

```
test("Check string match", () => {
  expect("hello world").toMatch(/hello/);
});

test("Array contains specific item", () => {
  expect([1, 2, 3]).toContain(2);
});

test("Object has required property", () => {
  expect({ name: "Alice" }).toHaveProperty("name", "Alice");
});
```

✓ Jest provides **various matchers** for testing strings, arrays, and objects.

2.4 Testing Asynchronous Functions with Jest

📌 Example: Testing an API Call

```
async function fetchData() {
  return new Promise((resolve) => {
```

```
    setTimeout(() => resolve("Success"), 1000);
  });
}

test("fetchData returns 'Success'", async () => {
  const data = await fetchData();
  expect(data).toBe("Success");
});
```

✓ Jest **handles async functions** using async/await.

CHAPTER 3: WRITING TESTS WITH MOCHA & CHAI

3.1 Installing Mocha & Chai

📌 **Install Mocha and Chai for Testing**

```
npm install --save-dev mocha chai
```

✓ **Mocha** → Test framework
✓ **Chai** → Assertion library

3.2 Writing Unit Tests with Mocha & Chai

📌 **Create math.js with a function to test:**

```
function multiply(a, b) {
  return a * b;
}
```

```
module.exports = multiply;
```

📌 **Create test/math.test.js and Add Tests:**

```
const { expect } = require("chai");
const multiply = require("../math");

describe("Math Functions", () => {
  it("should multiply 2 * 3 to equal 6", () => {
    expect(multiply(2, 3)).to.equal(6);
  });
});
```

📌 **Run Mocha Tests**

```
npx mocha
```

✓ Mocha **executes tests and logs results**.

3.3 Testing Asynchronous Code with Mocha

📌 **Example: Testing an API Call with Mocha**

```
const { expect } = require("chai");

async function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Success"), 1000);
```

```
  });
}

describe("Async Function", () => {
  it("should return 'Success'", async () => {
    const result = await fetchData();
    expect(result).to.equal("Success");
  });
});
```

✓ Mocha **supports async tests** using async/await.

CHAPTER 4: TESTING EXPRESS.JS APIS WITH JEST & SUPERTEST

4.1 Installing Supertest for API Testing

📌 **Install Required Packages:**

```
npm install --save-dev supertest jest
```

✓ supertest helps **test API endpoints**.

4.2 Writing API Tests for Express.js

📌 **Create server.js (Express API)**

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

```
app.get("/api", (req, res) => {
  res.json({ message: "API is running" });
});

module.exports = app;
```

📌 **Create server.test.js for API Testing**

```
const request = require("supertest");
const app = require("./server");

test("GET /api should return API message", async () => {
  const res = await request(app).get("/api");
  expect(res.status).toBe(200);
  expect(res.body.message).toBe("API is running");
});
```

📌 **Run Jest Tests**

```
npm test
```

✓ Verifies API **returns correct responses**.

Case Study: How Netflix Uses Jest & Mocha for Testing

Challenges Faced by Netflix

✓ Handling **millions of API requests per second**.
✓ Ensuring **video streaming works on all devices**.
✓ Maintaining **bug-free updates** across multiple platforms.

Solutions Implemented

✓ **Used Jest for unit tests** on React frontend.
✓ **Implemented Mocha for backend API testing**.
✓ **Ran integration tests** to simulate real-user interactions.

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

✓ **Unit tests catch bugs before deployment**.
✓ **Automated API tests ensure high availability**.
✓ **Continuous testing improves application stability**.

📝 Exercise

✅ Write a Jest test to **check if a function returns even numbers**.
✅ Create a Mocha test for **an Express.js API endpoint**.
✅ Write a Jest test for **fetching user data from an API**.
✅ Integrate **Supertest to test a CRUD API**.

Conclusion

✓ **Jest is great for frontend and backend unit testing**.
✓ **Mocha provides flexible testing for Node.js applications**.
✓ **Supertest allows seamless API testing for Express.js**.
✓ **Automated tests improve application reliability and performance**.

API TESTING WITH POSTMAN

CHAPTER 1: INTRODUCTION TO API TESTING WITH POSTMAN

1.1 What is API Testing?

API Testing is the process of verifying that an API **functions correctly, returns expected responses, and handles errors properly**. It is crucial for ensuring that APIs work as intended before they are used in production environments.

- **Why Use Postman for API Testing?**

✓ **User-friendly interface** for testing APIs.
✓ Supports **automated testing and scripting**.
✓ Handles **authentication, query parameters, and headers**.
✓ **Integrates with CI/CD pipelines** for continuous testing.

- **Types of API Testing in Postman:**

Type	Purpose	Example
Functional Testing	Checks if API returns correct data	Verifying user login
Performance Testing	Measures API speed & response time	Load testing
Security Testing	Ensures authentication and authorization	Testing JWT token validity
Error Handling Testing	Tests API failure scenarios	Checking incorrect data inputs

CHAPTER 2: SETTING UP POSTMAN

2.1 Installing Postman

📌 **Download and install Postman from** https://www.postman.com/downloads/.

2.2 Creating a New API Request

1. Open Postman and click **"New"** → **"Request"**.
2. Enter the API **endpoint URL** (e.g., http://localhost:5000/api/users).
3. Select **HTTP method** (GET, POST, PUT, DELETE).
4. Click **"Send"** to execute the request.

🔹 **Example: Sending a GET request to Fetch Users**

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

CHAPTER 3: PERFORMING CRUD OPERATIONS WITH POSTMAN

3.1 Sending a POST Request (Create a New User)

📌 **Example: Creating a New User**

- **Method:** POST
- **URL:** http://localhost:5000/api/users
- **Headers:**
  - Content-Type: application/json
- **Body (JSON):**

{

"name": "Alice",

"email": "alice@example.com",

"password": "securepassword"

}

- Click **"Send"** → **201 Created** response confirms successful creation.

3.2 Sending a GET Request (Retrieve User Data)

📌 **Example: Fetching All Users**

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

🔹 **Expected Response (JSON):**

[

{ "id": 1, "name": "Alice", "email": "alice@example.com" },

{ "id": 2, "name": "Bob", "email": "bob@example.com" }

]

✓ Confirms that API returns the **correct list of users**.

3.3 Sending a PUT Request (Update a User's Info)

📌 **Example: Updating a User's Email**

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

- **Headers:**
  - Content-Type: application/json
- **Body (JSON):**

{

"email": "alice_updated@example.com"

}

✓ API should return **200 OK** with updated user details.

3.4 Sending a DELETE Request (Remove a User)

📌 **Example: Deleting a User by ID**

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

✓ Response **204 No Content** confirms successful deletion.

CHAPTER 4: AUTHENTICATION & AUTHORIZATION TESTING IN POSTMAN

4.1 Testing JWT Authentication with Postman

📌 **Step 1: Obtain a JWT Token**

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

{

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

✓ Response returns a **JWT token**:

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cC..."
}
```

📌 Step 2: Access Protected Routes Using Token

- **Method:** GET
- **URL:** http://localhost:5000/api/profile
- **Headers:**
  - Authorization: Bearer YOUR_JWT_TOKEN

✓ The API should return **the user's profile data** if authentication is successful.

CHAPTER 5: AUTOMATING API TESTS IN POSTMAN

5.1 Writing Postman Tests with JavaScript

Postman allows adding **test scripts** to automate response validation.

📌 Example: Checking Response Status Code

```
pm.test("Status code is 200", function () {
  pm.response.to.have.status(200);
});
```

✓ Fails the test if API does **not return 200 OK**.

📌 **Example: Validating JSON Response**

```
pm.test("Response contains user email", function () {
  var jsonData = pm.response.json();
  pm.expect(jsonData.email).to.eql("alice@example.com");
});
```

✓ Ensures the API **returns the correct user email**.

5.2 Running Tests with Postman Collection Runner

📌 **Steps to Automate API Testing:**

1. **Create a Collection** → Add all API requests.
2. **Click "Runner"** in Postman.
3. **Select the Collection & Run Tests**.

✓ Postman automatically **executes and validates API responses**.

Case Study: How GitHub Uses API Testing

Challenges Faced by GitHub

✓ Managing **millions of API requests daily**.
✓ Ensuring **secure authentication for repositories**.

Solutions Implemented

✓ Used **Postman tests** to validate API endpoints.
✓ Implemented **JWT authentication for secure API access**.
✓ Automated API tests to **catch errors before deployment**.

🔹 **Key Takeaways from GitHub's API Testing Strategy:**

✓ **Automated API tests improve stability and reliability.**

✓ **JWT authentication ensures secure API communication.**

✓ **Using Postman collections streamlines large-scale API validation.**

📝 Exercise

✅ Set up Postman requests to **test a CRUD API**.

✅ Write **test scripts** to validate JSON responses.

✅ Use Postman to **test authentication and authorization**.

✅ Automate API tests using **Postman Collection Runner**.

Conclusion

✓ **Postman simplifies API testing with an easy-to-use interface.**

✓ **Supports automated testing with JavaScript scripts.**

✓ **Handles authentication & role-based access control (RBAC).**

✓ **Improves API reliability by automating test cases.**

CODE SPLITTING & LAZY LOADING

CHAPTER 1: INTRODUCTION TO CODE SPLITTING & LAZY LOADING

1.1 What is Code Splitting?

Code splitting is a **technique used in JavaScript applications** to break large bundles into smaller chunks, allowing parts of the application to load **only when needed**.

- **Why Use Code Splitting?**

✓ **Improves page load speed** by reducing initial bundle size.
✓ **Enhances performance** by loading JavaScript on demand.
✓ **Optimizes bandwidth usage** for better user experience.

- **How Code Splitting Works in React & Webpack?**
  - **Splitting the code into chunks** (instead of one big file).
  - **Loading JavaScript dynamically** as the user interacts with the app.

CHAPTER 2: CODE SPLITTING IN REACT USING REACT.LAZY & SUSPENSE

2.1 What is React.lazy()?

React's **lazy() function** allows you to **load components dynamically** when they are needed instead of at the initial load.

- **Why Use React.lazy()?**

✓ **Reduces initial bundle size** by loading components on demand.
✓ **Optimizes performance for large applications**.

📌 Example: Without Code Splitting (All Components Load Together)

```
import Home from "./Home";
import About from "./About";

function App() {
  return (
    <div>
      <Home />
      <About />
    </div>
  );
}

export default App;
```

✓ Loads **all components immediately**, increasing initial load time.

2.2 Implementing Code Splitting with React.lazy()

📌 Example: Using React.lazy() for Lazy Loading

```
import React, { lazy, Suspense } from "react";

const Home = lazy(() => import("./Home"));
```

```
const About = lazy(() => import("./About"));


function App() {
  return (
    <Suspense fallback={<h2>Loading...</h2>}>
      <Home />
      <About />
    </Suspense>
  );
}


export default App;
```

✓ lazy(() => import("./Home")) loads **Home component only when needed**.
✓ Suspense provides a **fallback (loading message) until component loads**.

2.3 Lazy Loading with React Router

📌 Example: Lazy Loading Routes in React Router

```
import React, { lazy, Suspense } from "react";
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
```

```
const Home = lazy(() => import("./Home"));
const About = lazy(() => import("./About"));

function App() {
  return (
    <Router>
      <Suspense fallback={<h2>Loading...</h2>}>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/about" element={<About />} />
        </Routes>
      </Suspense>
    </Router>
  );
}

export default App;
```

✓ Loads **routes dynamically** only when accessed.
✓ Improves **navigation performance** for large apps.

CHAPTER 3: CODE SPLITTING IN WEBPACK

3.1 Webpack's Automatic Code Splitting

Webpack **automatically splits code** for better performance using import().

📌 **Example: Dynamic Import in JavaScript**

```
document.getElementById("loadModule").addEventListener("click", () => {
  import("./module").then(module => {
    module.default();
  });
});
```

✓ Loads module.js **only when the button is clicked**.

3.2 Webpack SplitChunksPlugin for Code Splitting

Webpack **splits vendor and common code** into separate files.

📌 **Modify webpack.config.js for Code Splitting:**

```
module.exports = {
  optimization: {
    splitChunks: {
      chunks: "all",
    },
  },
};
```

✓ Automatically **separates vendor libraries (React, Lodash) from the main bundle**.

Chapter 4: Lazy Loading Images & Assets

4.1 Why Lazy Load Images?

✓ Improves **initial page load speed**.
✓ Reduces **bandwidth usage**.

📌 **Example: Lazy Loading Images Using loading="lazy" (HTML)**

```
<img src="large-image.jpg" alt="Example" loading="lazy">
```

✓ Defers image loading **until the user scrolls to it**.

📌 **Example: Lazy Loading Images in React**

```
const LazyImage = ({ src, alt }) => (
  <img src={src} alt={alt} loading="lazy" />
);
```

✓ Loads images **only when they come into view**.

Chapter 5: Optimizing Performance with Code Splitting

5.1 Measuring Bundle Size with Webpack Bundle Analyzer

📌 **Install Webpack Bundle Analyzer:**

```
npm install --save-dev webpack-bundle-analyzer
```

📌 **Modify webpack.config.js to Analyze Bundles:**

```
const { BundleAnalyzerPlugin } = require("webpack-bundle-analyzer");


module.exports = {
```

```
  plugins: [new BundleAnalyzerPlugin()],
};
```

✓ Helps **visualize which files contribute most to bundle size**.

5.2 Prefetching & Preloading Resources

📌 **Preloading Important Resources for Faster Load:**

```
<link rel="preload" href="styles.css" as="style">
```

✓ Ensures **critical assets load faster**.

📌 **Prefetching Non-Essential Resources:**

```
<link rel="prefetch" href="about.js">
```

✓ Loads **"About" page script in the background** for faster navigation.

Case Study: How YouTube Uses Code Splitting & Lazy Loading

Challenges Faced by YouTube

✓ Handling **billions of video requests** efficiently.
✓ Ensuring **faster page loads across devices**.

Solutions Implemented

✓ Used **lazy loading for video thumbnails**.
✓ Split JavaScript **to load only required features**.
✓ Implemented **prefetching for recommended videos**.

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

✓ **Lazy loading images & scripts improves performance.**

✓ **Code splitting reduces JavaScript bundle sizes.**

✓ **Prefetching improves next-page load speeds.**

📝 Exercise

✅ Implement **lazy loading in a React app** using React.lazy().

✅ Use **React Router to lazy load pages** only when visited.

✅ Measure **bundle size using Webpack Bundle Analyzer**.

✅ Add **lazy loading for images using loading="lazy"**.

Conclusion

✓ **Code splitting reduces initial JavaScript load times.**

✓ **React.lazy() + Suspense loads components only when needed.**

✓ **Lazy loading images & assets improves website performance.**

✓ **Using Webpack for automatic code splitting enhances optimization.**

IMAGE & ASSET OPTIMIZATION

CHAPTER 1: INTRODUCTION TO IMAGE & ASSET OPTIMIZATION

1.1 What is Image & Asset Optimization?

Image and asset optimization is the process of **reducing file sizes** while maintaining quality to improve **website performance, load times, and SEO rankings**. It involves **compressing images, lazy loading assets, and caching static files** to enhance user experience.

- **Why Optimize Images & Assets?**

✓ **Faster website load times** → Reduces bounce rate.

✓ **Improves SEO** → Google ranks faster websites higher.

✓ **Reduces bandwidth usage** → Saves server costs.

✓ **Enhances user experience** → Improves engagement on mobile devices.

- **Common Assets to Optimize:**

Asset Type	Optimization Method	Example
Images	Compression, lazy loading	PNG, JPEG, WebP, SVG
CSS & JS Files	Minification, bundling	style.css, main.js
Fonts	Subsetting, caching	Google Fonts, FontAwesome
Videos	Streaming, compression	MP4, WebM

CHAPTER 2: OPTIMIZING IMAGES FOR WEB PERFORMANCE

2.1 Choosing the Right Image Format

📌 Which Image Format Should You Use?

Format	Best For	Compression	Supports Transparency?
JPEG	Photos & colorful images	High	No
PNG	Images with transparency	Medium	Yes
WebP	Modern, best for web	Very High	Yes
SVG	Logos, icons, vector graphics	Scalable	Yes
GIF	Animations	Low	Yes

🔹 WebP vs. PNG vs. JPEG Compression Comparison

Format	File Size Reduction
PNG ⟶ WebP	25-30% smaller
JPEG ⟶ WebP	30-50% smaller

📌 Convert PNG/JPEG to WebP Using ImageMagick

```
convert image.png -quality 80 image.webp
```

✓ **Reduces file size significantly** without losing quality.

2.2 Using Image Compression Tools

📌 Online Image Compression Tools:

- **TinyPNG** → tinypng.com
- **Squoosh** → squoosh.app

📌 Automate Image Compression with sharp in Node.js

npm install sharp

📌 Example: Compress and Convert Images to WebP

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


sharp("input.jpg")
  .resize(800)
  .webp({ quality: 80 })
  .toFile("output.webp", (err, info) => {
    if (err) console.error(err);
    else console.log("Image optimized:", info);
  });
```

✓ Automatically resizes and **compresses images efficiently**.

2.3 Implementing Lazy Loading for Images

Lazy loading **defers image loading** until they enter the viewport, reducing initial page load time.

📌 Example: Lazy Loading Images in HTML

```
<img src="placeholder.jpg" data-src="image.jpg" class="lazyload" />
```

📌 Enable Lazy Loading in React (LazyLoadImage)

```
import { LazyLoadImage } from "react-lazy-load-image-component";


function App() {
  return (
    <LazyLoadImage src="image.jpg" alt="Optimized Image"
effect="blur" />
  );
}
```

✓ Improves **performance by loading images only when needed**.

CHAPTER 3: OPTIMIZING CSS, JAVASCRIPT, AND FONTS

3.1 Minifying CSS and JavaScript

Minification removes unnecessary characters (whitespace, comments) to **reduce file size**.

📌 **Minify CSS with csso (Command Line)**

```
npm install -g csso-cli

csso style.css -o style.min.css
```

📌 **Minify JavaScript with terser**

```
npm install -g terser

terser main.js -o main.min.js --compress --mangle
```

✓ **Speeds up page rendering** by reducing asset size.

3.2 Bundling & Code Splitting in React

Webpack splits JavaScript files into **smaller chunks,** loading only what's needed.

📌 Enable Code Splitting in React (React.lazy())

```
import React, { lazy, Suspense } from "react";


const Dashboard = lazy(() => import("./Dashboard"));


function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <Dashboard />
    </Suspense>
  );
}
```

✓ Improves initial load speed by loading JavaScript on demand.

3.3 Optimizing Google Fonts

Fonts can **slow down performance** if not optimized.

📌 Reduce Font Requests by Using Only Required Weights

```
<link rel="preconnect" href="https://fonts.googleapis.com">

<link rel="stylesheet"
href="https://fonts.googleapis.com/css2?family=Roboto:wght@400
;700&display=swap">
```

✓ **Loads only 400 and 700 weights,** reducing font file size.

Chapter 4: Caching and Content Delivery Networks (CDNs)

4.1 Caching Images & Static Assets

CDN caching **stores static files globally,** reducing server requests.

📌 **Enable Caching in Express.js (server.js)**

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

const app = express();


app.use(express.static("public", { maxAge: "1y" })); // Cache for 1 year


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

✓ **Stores assets in browser cache** for faster future loads.

4.2 Using a CDN for Faster Image Delivery

Cloudflare, AWS CloudFront, and Google Cloud CDN **distribute assets globally**.

📌 **Steps to Use Cloudflare CDN:**

1. Sign up at Cloudflare.
2. Add your website and enable **CDN caching**.
3. Store images in a **dedicated CDN subdomain (e.g., cdn.example.com)**.

📌 **Example: Serve Images via Cloudflare CDN**

<img src="https://cdn.example.com/images/photo.jpg" alt="Fast image">

✓ Delivers **optimized images from the nearest data center**.

Case Study: How Netflix Optimizes Assets for Faster Streaming

Challenges Faced by Netflix

✓ Handling **high-resolution images and videos** efficiently.
✓ Ensuring **fast content delivery across the world**.
✓ Reducing **bandwidth costs and improving load times**.

Solutions Implemented

✓ Used **WebP for thumbnails and previews**.
✓ Implemented **CDN caching to serve images instantly**.
✓ Applied **lazy loading to avoid unnecessary data usage**.

🔹 **Key Takeaways from Netflix's Optimization Strategy:**
✓ **Using modern image formats like WebP reduces bandwidth.**
✓ **CDN caching speeds up asset delivery for global users.**
✓ **Lazy loading minimizes unnecessary resource loading.**

📝 **Exercise**

✅ Convert PNG/JPEG images to **WebP** format and compare file sizes.
✅ Implement **lazy loading in React** using react-lazy-load-image-component.
✅ Use **Cloudflare or AWS CloudFront** to cache images.
✅ Minify **CSS and JavaScript** files for faster loading.

Conclusion

✓ Optimizing images reduces file size and improves page speed.
✓ Using WebP and lazy loading enhances user experience.
✓ Minifying CSS & JS speeds up rendering and execution.
✓ CDNs cache assets globally, reducing server load.

ASSIGNMENT:

OPTIMIZE AND TEST A WEB APPLICATION FOR PERFORMANCE

Assignment Solution: Optimize and Test a Web Application for Performance

Step 1: Measuring Application Performance

1.1 Analyzing Frontend Performance with Lighthouse

📌 **Run Lighthouse in Chrome DevTools:**

1. Open the **web application** in Chrome.
2. Press F12 → Go to **Lighthouse**.
3. Click **"Generate Report"** and analyze:
    - **Performance** (Load time, Render speed)
    - **Accessibility** (Color contrast, ARIA labels)
    - **Best Practices** (Security checks, HTTPS)

📌 **Alternatively, Run Lighthouse via CLI:**

```
npx lighthouse https://yourwebsite.com --view
```

✓ Provides **detailed performance metrics** for optimization.

1.2 Profiling API Performance with Postman

📌 **Steps to Analyze API Response Time in Postman:**

1. Open Postman and send a request to an API endpoint.
2. Check **response time in milliseconds** (bottom-right corner).
3. Optimize if response time exceeds **200ms**.

✓ Helps **identify slow API endpoints** for optimization.

Step 2: Optimizing Frontend Performance

2.1 Minifying and Compressing JavaScript & CSS

📌 **Install Terser for JavaScript Minification:**

```
npm install terser -g
terser src/index.js -o dist/index.min.js
```

✓ Reduces **JavaScript file size** for faster loading.

📌 **Minify CSS with clean-css-cli**

```
npm install clean-css-cli -g
cleancss -o dist/style.min.css src/style.css
```

✓ Improves **page speed by reducing CSS size**.

2.2 Using Lazy Loading for Images & Components

📌 **Lazy Load Images in React:**

```
const LazyImage = React.lazy(() => import("./ImageComponent"));

function App() {
  return (
    <React.Suspense fallback={<p>Loading...</p>}>
      <LazyImage />
    </React.Suspense>
```

```
  );
}
```

✓ Loads components **only when needed,** improving initial page load time.

Step 3: Optimizing Backend Performance

3.1 Caching API Responses Using Redis

📌 Install Redis for Fast Caching

```
npm install redis
```

📌 Implement Caching in Express API (server.js)

```
const redis = require("redis");
const client = redis.createClient();

app.get("/api/data", async (req, res) => {
  const cachedData = await client.get("data");
  if (cachedData) return res.json(JSON.parse(cachedData));

  const apiData = { message: "Hello from API!" };
  await client.setEx("data", 60, JSON.stringify(apiData)); // Cache for 60 seconds

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

✓ Reduces API response time **by serving cached data**.

3.2 Optimizing Database Queries with Indexing

📌 **Creating Index in MongoDB for Faster Queries:**

```
UserSchema.index({ email: 1 });
```

✓ Makes **search queries faster** on indexed fields.

Step 4: Writing Performance Tests with Jest & Supertest

4.1 Testing API Response Time with Jest

📌 **Install Jest and Supertest for API Testing**

```
npm install --save-dev jest supertest
```

📌 **Create server.test.js to Measure API Speed**

```
const request = require("supertest");
const app = require("./server");

test("GET /api/data should return response within 200ms", async () => {
  const start = Date.now();
  const res = await request(app).get("/api/data");
  const end = Date.now();

  expect(res.status).toBe(200);
```

```
  expect(end - start).toBeLessThan(200); // Expect API response in <200ms
});
```

✓ Ensures the **API responds quickly** for better user experience.

4.2 Testing Page Load Time with Puppeteer

📌 Install Puppeteer for Automated Browser Testing

```
npm install --save-dev puppeteer
```

📌 Create performance.test.js to Measure Page Speed

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

test("Page should load within 2 seconds", async () => {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  const start = Date.now();

  await page.goto("http://localhost:3000");

  const end = Date.now();
  expect(end - start).toBeLessThan(2000); // Expect load time <2s

  await browser.close();
```

});

✓ Ensures **fast page load times** under 2 seconds.

Step 5: Load Testing with Mocha & Artillery

5.1 Install Artillery for Load Testing

npm install -g artillery

📌 Create load-test.yml to Simulate High Traffic

```
config:
 target: "http://localhost:5000"
 phases:
  - duration: 60
    arrivalRate: 5  # 5 requests per second

scenarios:
 - flow:
    - get:
       url: "/api/data"
```

📌 Run Load Test

artillery run load-test.yml

✓ Tests **server stability under high traffic**.

Case Study: How Amazon Optimizes Performance

Challenges Faced by Amazon

✓ Handling **millions of user requests per second**.
✓ Ensuring **fast product searches and recommendations**.

Solutions Implemented

✓ Used **Redis caching** to reduce API load time.
✓ Implemented **database indexing** for faster queries.
✓ Optimized **frontend assets using lazy loading and minification**.

🔹 **Key Takeaways from Amazon's Performance Strategy:**

✓ **Caching reduces API response times significantly.**
✓ **Load testing ensures the system handles peak traffic efficiently.**
✓ **Frontend optimization minimizes load times for better UX.**

📝 Exercise

✅ Implement **Redis caching** in an Express API.
✅ Write **Jest tests to measure API response time**.
✅ Run **Puppeteer tests to check page load time**.
✅ Perform **load testing with Artillery** to simulate 100 users.

Conclusion

✓ **Frontend optimization speeds up page rendering.**
✓ **Backend caching reduces API response time.**
✓ **Jest & Mocha ensure APIs and UI components work correctly.**
✓ **Load testing simulates real-world traffic for better scalability.**