Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

ADVANCED ASYNCHRONOUS PROGRAMMING IN JAVASCRIPT PROMISES, ASYNC/AWAIT

CHAPTER 1: INTRODUCTION TO ASYNCHRONOUS PROGRAMMING

1.1 What is Asynchronous Programming?

Asynchronous programming in JavaScript allows code execution **without blocking the main thread**, enabling applications to remain **responsive and efficient**. It is essential for:

- ✅ **Fetching data from APIs**
- ✅ **Handling user interactions**
- ✅ **Performing file operations**
- ✅ **Executing background tasks**

1.2 Synchronous vs. Asynchronous Execution

Synchronous Execution	Asynchronous Execution
Executes line by line	Executes tasks independently
Blocks execution until task completes	Allows execution of other code while waiting

Example: console.log("A"); console.log("B");	Example: fetch(url).then(data => console.log(data))

CHAPTER 2: UNDERSTANDING PROMISES IN JAVASCRIPT

2.1 What is a Promise?

A **Promise** in JavaScript represents a **future value** that will either:

- ✅ **Resolve (Success)** → Returns the expected data
- ❌ **Reject (Failure)** → Returns an error

Promises help handle **asynchronous operations** without deeply nested callbacks (**callback hell**).

2.2 States of a Promise

Promise State	Description
Pending	Initial state, waiting for completion
Fulfilled	Operation successful, returns a value
Rejected	Operation failed, returns an error

2.3 Creating a Basic Promise

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    let success = true; // Simulating a condition
    if (success) {
```

```
        resolve("Promise resolved successfully!");
    } else {
        reject("Promise rejected!");
    }
  }, 2000);
});

// Handling the Promise
myPromise
  .then(result => console.log(result))  // Handles success
  .catch(error => console.error(error)) // Handles failure
  .finally(() => console.log("Promise execution completed."));
```

✅ **Simulates a delayed operation (2 seconds).**
✅ **Uses .then() for success and .catch() for failure.**

2.4 Chaining Multiple Promises

Multiple asynchronous operations can be linked using **Promise chaining**.

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

```
}


fetchData()
  .then(data => {
    console.log(data);
    return "Processing data...";
  })
  .then(result => {
    console.log(result);
    return "Saving data...";
  })
  .then(finalStep => console.log(finalStep))
  .catch(error => console.error("Error:", error));
```

✅ **Executes steps sequentially using .then().**

CHAPTER 3: HANDLING ASYNCHRONOUS CODE WITH ASYNC/AWAIT

3.1 What is Async/Await?

✅ **Async/Await is a cleaner way to work with Promises**.
✅ **async function always returns a Promise**.
✅ **await pauses execution until the Promise resolves** (instead of using .then()).

3.2 Basic Example of Async/Await

```
async function fetchData() {
   return "Data retrieved!";
}

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

✅ **Returns a Promise without explicit new Promise().**

3.3 Using await Inside an Async Function

```
function delay(time) {
   return new Promise(resolve => setTimeout(resolve, time));
}

async function processTask() {
   console.log("Task started...");
   await delay(2000);  // Waits for 2 seconds
   console.log("Task completed after 2 seconds.");
}

processTask();
```

✅ **Executes code sequentially, pausing at await.**

3.4 Handling Errors with Try-Catch

Unlike .catch() in Promises, Async/Await uses try...catch for error handling.

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

fetchData();
```

✅ **Prevents unhandled promise rejections.**

Chapter 4: Comparing Promises vs. Async/Await

Feature	Promises	Async/Await
Syntax	Uses .then() and .catch()	Uses async and await

Readability	More complex for long chains	Cleaner and more readable
Error Handling	Uses .catch()	Uses try...catch
Execution Style	Non-blocking	Synchronous-style execution

💡 **Use Async/Await for cleaner and more maintainable code!**

CHAPTER 5: REAL-WORLD APPLICATIONS OF ASYNC/AWAIT

5.1 Fetching API Data

```
async function getUsers() {
  try {
    let response = await fetch("https://jsonplaceholder.typicode.com/users");
    let users = await response.json();
    console.log(users);
  } catch (error) {
    console.error("Error:", error);
  }
}

getUsers();
```

✅ **Retrieves and logs users from an API.**

5.2 Handling Multiple Asynchronous Requests

If multiple API calls need to be made, **Promise.all()** can execute them concurrently.

```
async function fetchMultipleData() {
  try {
    let [posts, comments] = await Promise.all([
      fetch("https://jsonplaceholder.typicode.com/posts/1").then(res => res.json()),

fetch("https://jsonplaceholder.typicode.com/comments/1").then(res => res.json())
    ]);

    console.log("Post:", posts);
    console.log("Comment:", comments);
  } catch (error) {
    console.error("Error fetching data:", error);
  }
}
```

```
fetchMultipleData();
```

✅ **Executes both requests in parallel, reducing execution time.**

CHAPTER 6: HANDS-ON EXERCISES

Exercise 1: Convert Callbacks to Promises

✅ **Rewrite a function using Promises instead of callbacks.**

Exercise 2: Use Async/Await to Fetch API Data

✅ **Fetch data from an API and display it in the console.**

Exercise 3: Handle API Errors Gracefully

✅ **Implement try...catch to handle network errors when fetching data.**

CONCLUSION

✅ **Promises provide a structured way to handle asynchronous operations.**
✅ **Async/Await simplifies promise handling, making code more readable.**
✅ **Use try...catch for better error handling in async functions.**
✅ **Promise.all() executes multiple promises in parallel, improving efficiency.**

🚀 **Next Steps:**

- **Use async/await in real-world applications (React, Node.js).**

- **Optimize performance with concurrent requests (Promise.all).**
- **Implement real-time data fetching using WebSockets with async/await.**

By mastering **Promises & Async/Await**, you can build **efficient, scalable, and non-blocking applications**! 🎉🚀

Advanced Asynchronous Programming: WebSockets for Real-Time Communication

Chapter 1: Introduction to WebSockets

1.1 What are WebSockets?

WebSockets provide **full-duplex communication** between a client and server over a single TCP connection. Unlike traditional HTTP, which follows a **request-response cycle**, WebSockets allow **real-time, bidirectional data transfer**.

1.2 Why Use WebSockets?

✅ **Real-Time Communication** – Ideal for **chat apps, live notifications, stock tickers, multiplayer games**.
✅ **Persistent Connection** – Reduces the overhead of repeated HTTP requests.
✅ **Low Latency** – Faster than polling or long-polling techniques.
✅ **Efficient Resource Usage** – Avoids unnecessary server load compared to HTTP polling.

Chapter 2: WebSockets vs HTTP

Feature	WebSockets	HTTP (REST API)
Connection	Persistent	Stateless (request-response)
Data Flow	Bidirectional	Client-to-server only
Latency	Low	Higher (due to repeated requests)

Use Case	Real-time apps	Standard web communication

Example:

- **Use WebSockets for**: Chat applications, live sports scores, real-time trading.
- **Use HTTP for**: Standard web pages, API calls, non-real-time data.

CHAPTER 3: SETTING UP WEBSOCKETS IN NODE.JS

3.1 Install Required Packages

We will use the **ws** library to create a WebSocket server in Node.js.

mkdir websocket-server

cd websocket-server

npm init -y

npm install ws express

✅ **ws** – A WebSocket library for Node.js.
✅ **express** – Used to serve an HTTP server (optional).

3.2 Creating a WebSocket Server

Create server.js:

const WebSocket = require("ws");

const express = require("express");

```
const app = express();
const PORT = 5000;

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

// Create WebSocket server
const wss = new WebSocket.Server({ server });

wss.on("connection", (ws) => {
  console.log("New client connected!");

  // Handle incoming messages
  ws.on("message", (message) => {
    console.log("Received:", message);
    ws.send(`Echo: ${message}`);  // Send message back to client
  });
```

```
  // Handle client disconnect
  ws.on("close", () => {
    console.log("Client disconnected");
  });
});
```

✅ **Runs a WebSocket server on ws://localhost:5000.**
✅ **Listens for client connections** and **sends responses**.

3.3 Creating a WebSocket Client

Create client.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>WebSocket Client</title>
</head>
<body>
  <h2>WebSocket Chat</h2>
  <input type="text" id="messageInput" placeholder="Enter message">
  <button onclick="sendMessage()">Send</button>
  <p id="response"></p>
```

```
  <script>
    const ws = new WebSocket("ws://localhost:5000");

    ws.onopen = () => console.log("Connected to server");
    ws.onmessage = (event) => {
      document.getElementById("response").innerText =
event.data;
    };

    function sendMessage() {
      const message =
document.getElementById("messageInput").value;
      ws.send(message);
    }
  </script>
</body>
</html>
```

✅ **Connects to WebSocket server** and **sends messages**.
✅ **Receives and displays real-time responses from the server**.

CHAPTER 4: BROADCASTING MESSAGES TO MULTIPLE CLIENTS

4.1 Sending Messages to All Connected Clients

Modify server.js to broadcast messages to **all clients**:

```
wss.on("connection", (ws) => {
  console.log("New client connected!");

  ws.on("message", (message) => {
    console.log("Received:", message);

    // Broadcast to all clients
    wss.clients.forEach(client => {
      if (client.readyState === WebSocket.OPEN) {
        client.send(`User: ${message}`);
      }
    });
  });
});
```

✅ **Ensures all connected clients receive messages** (used in chat apps).

CHAPTER 5: WEBSOCKETS WITH EXPRESS AND REACT

5.1 Setting Up a WebSocket Server with Express

Modify server.js to serve a simple HTTP route:

```
app.get("/", (req, res) => {
  res.send("WebSocket Server is running");
});
```

✅ **Now supports both WebSockets and HTTP requests.**

5.2 Creating a WebSocket Client in React

Create a React project and install Axios for HTTP requests:

```
npx create-react-app websocket-client
cd websocket-client
npm install axios
```

Modify App.js:

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

const ws = new WebSocket("ws://localhost:5000");

function App() {
  const [messages, setMessages] = useState([]);
  const [input, setInput] = useState("");

  useEffect(() => {
    ws.onmessage = (event) => {
```

```
        setMessages(prev => [...prev, event.data]);
      };
    }, []);

    const sendMessage = () => {
      ws.send(input);
      setInput("");
    };

    return (
      <div>
        <h1>WebSocket Chat</h1>
        <input value={input} onChange={(e) =>
setInput(e.target.value)} />
        <button onClick={sendMessage}>Send</button>
        <ul>
          {messages.map((msg, i) => <li key={i}>{msg}</li>)}
        </ul>
      </div>
    );
}
```

export default App;

✅ **Integrates React with WebSockets to send and receive messages in real-time.**

CHAPTER 6: WEBSOCKETS USE CASES IN REAL-WORLD APPLICATIONS

6.1 WebSockets in Chat Applications

- Enables **real-time messaging** without reloading the page.
- Uses **broadcasting** to send messages to all connected clients.

6.2 WebSockets for Live Notifications

- Used in applications like **Facebook, Twitter, and WhatsApp** for **instant notifications**.

6.3 WebSockets in Financial and Trading Apps

- **Stock market applications** use WebSockets to provide real-time stock updates.

6.4 WebSockets in Multiplayer Games

- WebSockets handle **game state synchronization** between multiple players.

Chapter 7: WebSockets vs Other Real-Time Solutions

Feature	WebSockets	Server-Sent Events (SSE)	Long Polling
Connection	Persistent	Persistent	Multiple Requests
Data Flow	Bidirectional	Server to Client	Client to Server
Use Cases	Chat, Live updates	Live Feeds	Basic Updates
Efficiency	High	Medium	Low

✅ **WebSockets** are best for **bidirectional, low-latency communication**.
✅ **SSE** works for **one-way updates like live news feeds**.
✅ **Long polling** is an **older technique** that uses more server resources.

Chapter 8: Hands-on Exercises

Exercise 1: Build a Chat App with WebSockets

✅ Modify the React client to support **multiple users with usernames**.

Exercise 2: Implement Typing Indicators

✅ Show "User is typing..." messages using WebSockets.

Exercise 3: Secure WebSockets with Authentication

✅ Use JWT tokens to **authenticate WebSocket connections**.

CONCLUSION

✅ **WebSockets enable real-time, bidirectional communication.**
✅ **They outperform HTTP polling for real-time applications.**
✅ **Used in chat apps, live notifications, trading apps, and gaming.**
✅ **Can be integrated with Express, React, and MongoDB for full-stack development.**

🚀 **Next Steps:**

- Add **JWT authentication** to WebSockets.
- Deploy WebSockets using **Socket.io** for advanced features.
- Build a **real-time collaborative whiteboard**!

By mastering **WebSockets**, you can build **real-time, scalable applications** that enhance user experience! 🎉🚀

Testing & Performance Optimization: Unit Testing with Jest in Node.js & React

Chapter 1: Introduction to Unit Testing with Jest

1.1 What is Unit Testing?

Unit testing is a software testing method where **individual components** of an application are tested **in isolation** to ensure they function correctly.

1.2 What is Jest?

Jest is a **JavaScript testing framework** developed by Facebook. It is commonly used for testing applications built with **Node.js, React, and other JavaScript frameworks**.

✅ **Why Use Jest?**

✓ Simple API & easy setup.

✓ Supports **mocking** and **snapshot testing**.

✓ Built-in **test runner & assertion library**.

✓ Works with both **backend (Node.js) & frontend (React.js)**.

Chapter 2: Installing Jest in Node.js (Backend Testing)

2.1 Install Jest in an Express.js Project

npm install --save-dev jest supertest

✅ **Dependencies:**

- **jest** – The testing framework.

- **supertest** – For testing Express.js APIs.

2.2 Update package.json for Jest Configuration

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

✅ Now, you can run tests using:

```
npm test
```

CHAPTER 3: WRITING UNIT TESTS FOR EXPRESS.JS APIS

3.1 Create a Sample Express Route (routes/userRoutes.js)

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

// Sample user data
const users = [{ id: 1, name: "Alice" }, { id: 2, name: "Bob" }];

// Get all users
router.get("/", (req, res) => {
   res.status(200).json(users);
});
```

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

3.2 Create server.js for Testing

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

const app = express();

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


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


module.exports = app; // Exporting for testing
```

3.3 Create a Jest Test File (tests/userRoutes.test.js)

```
const request = require("supertest");

const app = require("../server"); // Import the app


describe("User API Endpoints", () => {
  it("should return a list of users", async () => {
    const res = await request(app).get("/api/users");
    expect(res.statusCode).toEqual(200);
    expect(res.body).toHaveLength(2);
    expect(res.body[0]).toHaveProperty("name", "Alice");
  });
});
```

✅ **Run Tests:**

npm test

✅ **Output:**

PASS tests/userRoutes.test.js

✓ should return a list of users

CHAPTER 4: WRITING UNIT TESTS FOR REACT COMPONENTS

4.1 Install Jest & React Testing Library

npm install --save-dev jest @testing-library/react @testing-library/jest-dom

4.2 Create a Sample React Component (UserList.js)

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

import axios from "axios";


function UserList() {

  const [users, setUsers] = useState([]);


  useEffect(() => {

    axios.get("http://localhost:5000/api/users").then(response => setUsers(response.data));

  }, []);
```

```
  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}

export default UserList;
```

4.3 Write a Unit Test for UserList.js (__tests__/UserList.test.js)

```
import { render, screen } from "@testing-library/react";
import UserList from "../UserList";
import axios from "axios";

jest.mock("axios"); // Mock API calls

test("renders user list", async () => {
  axios.get.mockResolvedValue({ data: [{ id: 1, name: "Alice" }] });
```

```
  render(<UserList />);


  const userElement = await screen.findByText(/Alice/i);
  expect(userElement).toBeInTheDocument();
});
```

✅ **Run Tests:**

```
npm test
```

✅ **Output:**

```
PASS  __tests__/UserList.test.js
✓ renders user list
```

CHAPTER 5: MOCKING API CALLS IN JEST

5.1 Why Mock API Calls?

Mocking API requests ensures tests run **without depending on actual servers**.

5.2 Example: Mocking a Fetch Request

```
import axios from "axios";
import { getUsers } from "../api";

jest.mock("axios");
```

```
test("fetches users successfully", async () => {
  const mockData = { data: [{ id: 1, name: "Alice" }] };
  axios.get.mockResolvedValue(mockData);

  const users = await getUsers();
  expect(users).toEqual(mockData.data);
});
```

✅ **Output:**

```
PASS  __tests__/api.test.js
✓ fetches users successfully
```

CHAPTER 6: SNAPSHOT TESTING IN REACT

6.1 What is Snapshot Testing?

Snapshot testing ensures UI components **do not change unexpectedly**.

6.2 Example: Testing UserList.js Component

```
import { render } from "@testing-library/react";
import UserList from "../UserList";

test("matches snapshot", () => {
  const { asFragment } = render(<UserList />);
```

```
  expect(asFragment()).toMatchSnapshot();
});
```

✅ **Run Tests:**

```
npm test -- -u
```

✅ Jest creates a **snapshot file** to compare future changes.

Chapter 7: Measuring Test Coverage

7.1 Run Jest with Coverage Report

```
npm test -- --coverage
```

✅ **Sample Coverage Report:**

```
----------------------|---------|----------|---------|---------|-------------------
File                  | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #
----------------------|---------|----------|---------|---------|-------------------
All files             |   90.00 |    85.00 |   80.00 |   88.00 |
 src/UserList.js      |   100.0 |    90.00 |   90.00 |  100.00 |
----------------------|---------|----------|---------|---------|-------------------
```

Chapter 8: Best Practices for Unit Testing

✅ **Write Tests for Critical Features** – Focus on business logic & UI rendering.

✅ **Use Mocks for API Calls** – Prevent external dependencies from

affecting tests.

✅ **Run Tests Before Deployment** – Ensure no breaking changes in production.

✅ **Keep Tests Small & Independent** – Each test should verify **one behavior**.

✅ **Monitor Test Coverage** – Aim for **80%+ coverage** for robust applications.

CHAPTER 9: HANDS-ON PRACTICE & ASSIGNMENTS

Exercise 1: Test a New API Endpoint

1. Add a new route **/api/products** that returns an array of products.
2. Write a Jest test to verify it returns data correctly.

Exercise 2: Test User Authentication API

1. Write a Jest test for the login API (/api/auth/login).
2. Verify if it returns a **JWT token** upon successful login.

Exercise 3: Test Form Validation in React

1. Create a **LoginForm.js** component with input fields.
2. Write a **Jest test** to check if validation messages appear correctly.

CONCLUSION

🎉 You have successfully learned:

✅ **Unit testing with Jest in Node.js & React**

✅ **Mocking API calls for isolated testing**

✅ **Snapshot testing for UI stability**

✅ **Tracking test coverage**

🚀 **Next Steps:**

- Implement **integration tests** with **Supertest & Jest**.
- Learn **end-to-end testing** using **Cypress**.
- Deploy **CI/CD pipelines** to automate tests on GitHub Actions!

Happy Testing! 🎉 🚀

TESTING & PERFORMANCE OPTIMIZATION: API TESTING USING POSTMAN

CHAPTER 1: INTRODUCTION TO API TESTING

1.1 What is API Testing?

API testing is a type of software testing that focuses on verifying whether an **Application Programming Interface (API)** meets the expected functionality, reliability, performance, and security requirements.

1.2 Why Use Postman for API Testing?

Postman is a widely used tool for testing APIs because it provides:

✅ **User-friendly interface** – No coding required for basic testing.

✅ **Automation** – Supports writing test scripts using JavaScript.

✅ **Collection Runner** – Allows batch testing of multiple API requests.

✅ **Environment Variables** – Stores and reuses API keys, tokens, or URLs.

✅ **Monitoring & Debugging** – Helps inspect API responses.

CHAPTER 2: SETTING UP POSTMAN FOR API TESTING

2.1 Install Postman

1. Download **Postman** from https://www.postman.com/downloads/.
2. Install and open the application.

2.2 Understanding the Postman Interface

When you open Postman, you will see:

- **Collections** – A group of related API requests.
- **Request Builder** – Allows setting up HTTP methods (GET, POST, PUT, DELETE).
- **Response Section** – Displays API responses, status codes, and response times.
- **Environments & Variables** – Stores base URLs, authentication tokens, and other settings.

CHAPTER 3: PERFORMING API REQUESTS IN POSTMAN

3.1 Testing a GET Request

Example: Retrieve All Tasks from a Task Management API

1. Open Postman and create a new request.
2. Set the request type to **GET**.
3. Enter the API endpoint:
4. http://localhost:5000/api/tasks
5. Click **Send**.
6. Expected response:

```
[
  {
    "_id": "60b75d6c2f62a731b875a512",
    "title": "Complete API Testing",
    "status": "pending"
```

```
  },
  {
    "_id": "60b75d6c2f62a731b875a513",
    "title": "Review API responses",
    "status": "completed"
  }
]
```


✅ **Confirms that the GET request successfully retrieves all tasks.**

3.2 Testing a POST Request (Create Data)

Example: Add a New Task

1. Set the request type to **POST**.
2. Enter the API endpoint:
3. http://localhost:5000/api/tasks
4. Click on the **Body** tab and select **raw**, then set it to **JSON**.
5. Enter request data:


```
{
  "title": "Write API documentation",
  "status": "in progress"
}
```


5. Click **Send**.

6. Expected response:


```
{
  "_id": "60b75d6c2f62a731b875a514",
  "title": "Write API documentation",
  "status": "in progress",
  "createdAt": "2023-06-15T12:00:00.000Z"
}
```


✅ **Confirms that the API successfully creates a new task.**

3.3 Testing a PUT Request (Update Data)

Example: Update an Existing Task

1. Set the request type to **PUT**.
2. Use the endpoint with a task ID:
3. http://localhost:5000/api/tasks/60b75d6c2f62a731b875a514
4. Enter the new task details in the **Body** section:


```
{
  "status": "completed"
}
```


4. Click **Send**.
5. Expected response:


```
{
```

"_id": "60b75d6c2f62a731b875a514",

"title": "Write API documentation",

"status": "completed"

}

✅ **Confirms that the API successfully updates the task status.**

3.4 Testing a DELETE Request (Remove Data)

Example: Delete a Task

1. Set the request type to **DELETE**.
2. Use the endpoint with a task ID:
3. http://localhost:5000/api/tasks/60b75d6c2f62a731b875a514
4. Click **Send**.
5. Expected response:

{

"message": "Task deleted successfully"

}

✅ **Confirms that the task has been deleted.**

CHAPTER 4: AUTOMATING API TESTS IN POSTMAN

4.1 Writing Test Scripts

Postman allows writing JavaScript test scripts inside the **Tests** tab.

Example: Validate Response Status Code

1. Open Postman and send a **GET** request.
2. Go to the **Tests** tab and add the following script:

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

3. Click **Send**.
   ✅ **If the response is successful, Postman confirms the test passed.**

4.2 Validating JSON Response Data

```
pm.test("Response contains task title", function () {
  let jsonData = pm.response.json();
  pm.expect(jsonData[0]).to.have.property("title");
});
```

✅ **Ensures that the response contains the expected property.**

CHAPTER 5: USING POSTMAN COLLECTIONS & ENVIRONMENT VARIABLES

5.1 Creating Collections

1. Click **New Collection** and name it **Task Management API**.

2. Save API requests inside this collection.
   ✅ **Organizes API tests for easy access and execution.**

5.2 Using Environment Variables

1. Click **Environments** and create a new environment.
2. Add a variable for the API base URL:
3. Variable: base_url
4. Initial Value: http://localhost:5000/api
5. Use the variable in requests:
6. {{base_url}}/tasks

✅ **Allows quick updates when switching environments (local, staging, production).**

CHAPTER 6: RUNNING AUTOMATED API TESTS IN POSTMAN

6.1 Using the Collection Runner

1. Go to **Collections** and click **Run**.
2. Select the **Task Management API** collection.
3. Click **Run Collection**.
   ✅ **Executes all API requests and tests automatically.**

Chapter 7: Performance Optimization Techniques

7.1 Optimize API Response Time

- **Use Indexes in MongoDB**

```
TaskSchema.index({ title: 1 });
```

✅ **Speeds up search queries.**

Use Pagination for Large Datasets

```
app.get("/tasks", async (req, res) => {
  let { page, limit } = req.query;
  page = parseInt(page) || 1;
  limit = parseInt(limit) || 10;

  const tasks = await Task.find().limit(limit).skip((page - 1) * limit);
  res.json(tasks);
});
```

✅ **Improves performance when handling large datasets.**

Enable Caching Using Redis

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

app.get("/tasks", async (req, res) => {
  client.get("tasks", async (err, tasks) => {
    if (tasks) return res.json(JSON.parse(tasks));
```

```
        const dbTasks = await Task.find();
        client.setex("tasks", 3600, JSON.stringify(dbTasks));
        res.json(dbTasks);
    });
});
```

✅ **Reduces database load by caching responses.**

Chapter 8: Hands-on Practice & Exercises

Exercise 1: Perform CRUD API Testing in Postman

1. Create a new collection and add **GET, POST, PUT, DELETE** requests.
2. Use **environment variables** for API URLs.
3. Run automated tests using the **Collection Runner**.

Exercise 2: Implement API Performance Optimization

1. Add **pagination and sorting** to API responses.
2. Enable **MongoDB indexing** for faster queries.
3. Use **Redis caching** to improve API response time.

Conclusion

✅ **Postman simplifies API testing with request execution, automation, and validation.**
✅ **Automated tests ensure API functionality remains stable.**

✅ **Performance optimizations (pagination, caching, indexing) improve API speed.**

🚀 **Next Steps:**

- Implement **JWT authentication in API requests**.
- Deploy API on **Heroku or AWS**.
- Use **Newman CLI to automate Postman tests in CI/CD pipelines**!

By following this guide, you can now **test, validate, and optimize APIs like a pro!** 🚀 🔥

TESTING & PERFORMANCE OPTIMIZATION IN JAVASCRIPT

PERFORMANCE OPTIMIZATION TECHNIQUES

CHAPTER 1: INTRODUCTION TO PERFORMANCE OPTIMIZATION

1.1 What is Performance Optimization?

Performance optimization refers to techniques that **improve the speed, efficiency, and scalability** of a web application. Optimized applications provide:

✅ **Faster Load Times** – Improves user experience.

✅ **Better Responsiveness** – Reduces delays in UI interactions.

✅ **Efficient Resource Usage** – Lowers memory and CPU consumption.

✅ **Scalability** – Handles more users with minimal resource overhead.

1.2 Key Areas for Optimization

- **JavaScript Execution** – Reduce unnecessary computations.
- **Network Requests** – Optimize API calls and asset loading.
- **Rendering Performance** – Improve UI responsiveness.
- **Database Efficiency** – Optimize queries and indexing.

CHAPTER 2: JAVASCRIPT PERFORMANCE OPTIMIZATION TECHNIQUES

2.1 Minimize DOM Manipulations

Excessive **DOM updates** slow down rendering. Optimize by:

✅ **Batching updates** instead of modifying the DOM multiple times.

✅ **Using documentFragment** for multiple element insertions.

Example: Optimized DOM Update

Instead of:

```
for (let i = 0; i < 1000; i++) {
   let div = document.createElement("div");
   div.textContent = `Item ${i}`;
   document.body.appendChild(div);
}
```

Use **documentFragment** to reduce reflows:

```
let fragment = document.createDocumentFragment();
for (let i = 0; i < 1000; i++) {
   let div = document.createElement("div");
   div.textContent = `Item ${i}`;
   fragment.appendChild(div);
}
document.body.appendChild(fragment);
```

✅ **Improves rendering speed by reducing reflows.**

2.2 Optimize Loops & Iterations

Large loops impact performance. Use **optimized loop techniques**:

✅ **Use forEach, map, or reduce instead of for loops** where applicable.

✅ **Avoid unnecessary computations inside loops.**

Example: Avoid Unnecessary Computations in Loops

Instead of:

```
for (let i = 0; i < array.length; i++) {
  console.log(array[i] * 2);
}
```

Optimize by:

```
const doubledArray = array.map(num => num * 2);
console.log(doubledArray);
```

✅ **Reduces computational overhead.**

2.3 Debounce and Throttle Expensive Functions

Continuous event listeners **(e.g., scroll, resize, keypress)** can cause performance issues.

- **Debouncing** – Ensures a function is called **after a delay** instead of on every trigger.
- **Throttling** – Ensures a function is called **at most once per interval**.

Example: Debouncing Input Search

```
function debounce(func, delay) {
  let timer;
```

```
  return function(...args) {
    clearTimeout(timer);
    timer = setTimeout(() => func.apply(this, args), delay);
  };
}

const searchInput = document.getElementById("search");
searchInput.addEventListener("input", debounce(event => {
  console.log("Fetching search results for:", event.target.value);
}, 300));
```

✅ **Prevents excessive API calls while typing in a search bar.**

2.4 Optimize Event Listeners

Instead of adding **multiple event listeners**, use **event delegation**:

Example: Event Delegation

Instead of:

```
document.querySelectorAll("button").forEach(button => {
  button.addEventListener("click", () => console.log("Button clicked"));
});
```

Use **one event listener on the parent**:

```
document.getElementById("container").addEventListener("click",
event => {
  if (event.target.tagName === "BUTTON") {
    console.log("Button clicked");
  }
});
```

✅ **Reduces memory usage and improves efficiency.**

CHAPTER 3: OPTIMIZING NETWORK PERFORMANCE

3.1 Reduce API Calls with Caching

Fetching the same data repeatedly wastes resources. Use **localStorage, sessionStorage, or caching techniques**.

Example: Caching API Responses with localStorage

```
async function fetchData(url) {
  let cachedData = localStorage.getItem(url);
  if (cachedData) {
    return JSON.parse(cachedData);
  }

  const response = await fetch(url);
  const data = await response.json();
  localStorage.setItem(url, JSON.stringify(data));
```

```
    return data;
}
```

✅ **Prevents unnecessary API requests for repeated data.**

3.2 Optimize Image & Asset Loading

Images and assets impact page speed. Optimize by:

✅ **Using Lazy Loading** – Load images only when visible.

✅ **Compressing Images** – Reduce file size for faster loading.

Example: Lazy Loading Images

```
<img src="placeholder.jpg" data-src="actual-image.jpg" class="lazy-load">

<script>
document.addEventListener("DOMContentLoaded", function () {
    let lazyImages = document.querySelectorAll(".lazy-load");
    lazyImages.forEach(img => {
        img.src = img.dataset.src;
    });
});
</script>
```

✅ **Improves page speed by loading images only when required.**

CHAPTER 4: OPTIMIZING RENDERING PERFORMANCE

4.1 Use Efficient CSS Selectors

✅ **Avoid deep nested CSS selectors.**
✅ **Use class selectors (.class) instead of universal (*) selectors.**

Example: Optimized CSS Selectors

```
/* Avoid */

div.container ul li a { color: blue; }


/* Use */

.container a { color: blue; }
```

✅ **Reduces reflow and repaint costs.**

4.2 Reduce Layout Reflows

- **Avoid setting width/height dynamically.**
- **Use CSS transform instead of top and left for animations.**

Example: Optimized Animation Using transform

```
/* Avoid */

element { position: absolute; top: 50px; }


/* Use */

element { transform: translateY(50px); }
```

✅ **Improves rendering performance.**

Chapter 5: Database & Backend Performance Optimization

5.1 Indexing in MongoDB for Faster Queries

Use **indexes** to improve query performance.

Example: Create an Index on email Field

```
db.users.createIndex({ email: 1 });
```

✅ **Speeds up searches for users by email.**

5.2 Use Pagination to Optimize Large Data Retrieval

Instead of loading all data at once**, use pagination**.

Example: Pagination in MongoDB (Express.js API)

```
app.get("/users", async (req, res) => {
  let page = parseInt(req.query.page) || 1;
  let limit = 10;
  let users = await User.find().skip((page - 1) * limit).limit(limit);
  res.json(users);
});
```

✅ **Loads data in smaller chunks, improving performance.**

Chapter 6: Hands-on Exercises

Exercise 1: Implement Debouncing for Search Input

✅ **Modify an existing search bar to use debouncing.**

Exercise 2: Optimize API Calls Using Axios Interceptors

✅ **Cache API responses using Axios interceptors.**

Exercise 3: Improve Rendering Performance with Lazy Loading

✅ **Convert a webpage to use lazy-loaded images.**

Conclusion

✅ **Reduce DOM manipulations using documentFragment**
✅ **Optimize event handling with event delegation**
✅ **Use caching and pagination for efficient API calls**
✅ **Lazy load images and assets to improve page speed**
✅ **Optimize MongoDB queries with indexing**

🚀 **Next Steps:**

- Use **Web Workers** for multi-threaded JavaScript execution.
- Implement **Server-Side Rendering (SSR) for React/Vue apps**.
- Optimize **database schema design** for large-scale applications.

By applying these **performance optimization techniques**, you can build **fast, scalable, and efficient web applications!** 🚀 🎯

HANDS-ON PRACTICE: BUILD A LIVE CHAT APPLICATION USING WEBSOCKETS

CHAPTER 1: INTRODUCTION TO LIVE CHAT APPLICATION

1.1 What is a Live Chat Application?

A **live chat application** allows multiple users to communicate in real-time using WebSockets. Unlike traditional HTTP-based messaging, WebSockets enable **instant, bidirectional communication** between clients and the server.

1.2 Why Use WebSockets for Live Chat?

✅ **Instant Messaging** – Messages appear instantly without refreshing the page.
✅ **Persistent Connection** – A WebSocket connection remains open, reducing server overhead.
✅ **Efficient Resource Usage** – Reduces repeated HTTP requests compared to polling.
✅ **Scalability** – Supports multiple active users in chat rooms.

CHAPTER 2: SETTING UP THE PROJECT

2.1 Prerequisites

- **Node.js** and **npm** (Download)
- **WebSockets (ws package for Node.js)**
- **React.js for the frontend**

2.2 Create the Backend Project

mkdir websocket-chat

cd websocket-chat

npm init -y

2.3 Install Required Dependencies

npm install express ws cors

Package	Description
express	Handles HTTP requests
ws	WebSocket library for Node.js
cors	Enables frontend access

Chapter 3: Creating the WebSocket Server

3.1 Set Up WebSocket Server

Create a server.js file:

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

const WebSocket = require("ws");

const cors = require("cors");


const app = express();

const PORT = 5000;


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

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

// Create WebSocket server
const wss = new WebSocket.Server({ server });

wss.on("connection", (ws) => {
  console.log("New client connected");

  // Receive message and broadcast to all clients
  ws.on("message", (message) => {
    console.log("Received:", message);
    wss.clients.forEach(client => {
      if (client.readyState === WebSocket.OPEN) {
        client.send(message);
      }
    });
  });
```

```
  // Handle client disconnect
  ws.on("close", () => console.log("Client disconnected"));
});
```

✅ **Creates a WebSocket server on ws://localhost:5000**
✅ **Broadcasts received messages to all connected clients**

CHAPTER 4: SETTING UP THE FRONTEND (REACT.JS)

4.1 Create the React Project

```
npx create-react-app chat-client
cd chat-client
npm install axios
```

4.2 Create WebSocket Connection in React

Modify App.js:

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

const ws = new WebSocket("ws://localhost:5000");

function App() {
  const [messages, setMessages] = useState([]);
  const [input, setInput] = useState("");
```

```
  useEffect(() => {
    ws.onmessage = (event) => {
      setMessages(prev => [...prev, event.data]);
    };
  }, []);

  const sendMessage = () => {
    ws.send(input);
    setInput("");
  };

  return (
    <div>
      <h1>Live Chat</h1>
      <div>
        {messages.map((msg, i) => <p key={i}>{msg}</p>)}
      </div>
      <input value={input} onChange={(e) =>
setInput(e.target.value)} />
      <button onClick={sendMessage}>Send</button>
    </div>
```

```
  );
}

export default App;
```

✅ **Connects React frontend to WebSocket server**
✅ **Receives and displays messages in real-time**
✅ **Sends messages when clicking the button**

Chapter 5: Enhancing the Chat Application

5.1 Adding Usernames to Messages

Modify the WebSocket client to include a username:

```
const [username, setUsername] = useState("");
const [input, setInput] = useState("");

const sendMessage = () => {
  ws.send(JSON.stringify({ username, message: input }));
  setInput("");
};
```

Modify the WebSocket server to process JSON messages:

```
ws.on("message", (data) => {
  const parsedData = JSON.parse(data);
```

```
  const message = `${parsedData.username}: ${parsedData.message}`;


  wss.clients.forEach(client => {
    if (client.readyState === WebSocket.OPEN) {
      client.send(message);
    }
  });
});
```

✅ **Now messages include the sender's username**

5.2 Displaying Online Users

Modify the server to keep track of connected users:

```
const clients = new Set();


wss.on("connection", (ws) => {
  clients.add(ws);


  ws.on("close", () => {
    clients.delete(ws);
  });
```

```
  ws.on("message", (message) => {
    clients.forEach(client => {
      if (client.readyState === WebSocket.OPEN) {
        client.send(message);
      }
    });
  });
});
```

✅ **Tracks active users and updates when someone disconnects**

5.3 Adding Notifications for User Join/Leave

Modify the WebSocket connection handler:

```
wss.on("connection", (ws) => {
  ws.send("Welcome to the chat!");

  ws.on("close", () => {
    wss.clients.forEach(client => {
      if (client.readyState === WebSocket.OPEN) {
        client.send("A user has left the chat.");
      }
```

```
    });
  });
});
```

✅ **Displays messages when a user joins or leaves**

Chapter 6: Deploying the Chat Application

6.1 Deploy Backend on Render

1️⃣ Push Code to GitHub

```
git init
git add .
git commit -m "Initial commit"
git branch -M main
git remote add origin <your-repo-url>
git push -u origin main
```

2️⃣ Deploy on Render

- Go to https://render.com
- Create a **new web service**
- Connect your **GitHub repository**
- Deploy with **WebSocket support**

6.2 Deploy Frontend on Vercel

1️⃣ Install Vercel CLI

npm install -g vercel

2️⃣ Deploy

vercel

✅ **Now the chat app is accessible globally!**

CHAPTER 7: HANDS-ON EXERCISES

Exercise 1: Add Private Chat (Direct Messaging)

✅ Modify the WebSocket server to allow **private messages** between users.

Exercise 2: Implement Message Storage in MongoDB

✅ Store chat messages in **MongoDB** for later retrieval.

Exercise 3: Add Typing Indicator

✅ Show "User is typing..." messages using WebSockets.

Conclusion

✅ **WebSockets provide real-time, bidirectional communication**
✅ **A chat app can be built with Node.js, Express, and WebSockets**
✅ **React.js efficiently handles frontend WebSocket communication**
✅ **Scalable deployment using Render (backend) and Vercel (frontend)**

🚀 Next Steps:

- Add **authentication with JWT**
- Implement **chat rooms and groups**
- Integrate **video calling using WebRTC**

By completing this **hands-on project,** you have built a **real-time chat app** that can be extended for **large-scale applications!** 🎉🚀

Hands-on Practice: Optimize a MERN App for Performance

Chapter 1: Introduction to Performance Optimization in MERN Stack

1.1 Why Optimize a MERN App?

Performance optimization ensures that a **MERN (MongoDB, Express.js, React, Node.js)** application runs smoothly with:

✅ **Faster Page Load Times** – Improves user experience.
✅ **Reduced Server Load** – Efficient API requests reduce server strain.
✅ **Optimized Database Queries** – Prevents slow responses.
✅ **Scalability** – Supports more users efficiently.

Chapter 2: Backend Optimization in Express.js & MongoDB

2.1 Optimize Database Queries

Indexing for Faster Queries

Indexes speed up **search operations** in MongoDB.

Example: Add an index on the email field

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


const UserSchema = new mongoose.Schema({
  name: String,
```

```
  email: { type: String, unique: true, index: true },
  age: Number
});

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

✅ **Now, queries on email will be significantly faster.**

Use lean() for Read Queries

lean() improves performance by **returning plain JavaScript objects** instead of Mongoose documents.

```
const users = await User.find().lean(); // Faster than normal find()
```

Paginate Large Queries

Fetching large data sets slows down the API. Use pagination to **limit results**.

```
const limit = 10;
const page = req.query.page || 1;
const users = await User.find()
  .skip((page - 1) * limit)
  .limit(limit);
```

✅ **This reduces memory usage by fetching only required records.**

2.2 Optimize Express.js API Performance

Enable Compression

Compress responses to reduce **data transfer size**.

npm install compression

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

app.use(compression());
```

✅ **Improves response time for API calls.**

Use Caching to Reduce Database Calls

Cache frequently used data in **Redis**.

npm install redis

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

const client = redis.createClient();


// Middleware to check cache

const cacheMiddleware = (req, res, next) => {

  const { id } = req.params;

  client.get(id, (err, data) => {

    if (data) return res.json(JSON.parse(data));

    next();

  });
```

```
};


// Apply caching
app.get("/user/:id", cacheMiddleware, async (req, res) => {
  const user = await User.findById(req.params.id);
  client.setex(req.params.id, 3600, JSON.stringify(user)); // Store for 1 hour
  res.json(user);
});
```

✅ **This prevents repeated database queries for the same data.**

Optimize Middleware & Avoid Unnecessary Processing

Move **heavy calculations** to background jobs (e.g., using **Bull** for job queues).

```
npm install bull
const Queue = require("bull");
const emailQueue = new Queue("email");


app.post("/send-email", async (req, res) => {
  emailQueue.add({ email: req.body.email }); // Background task
  res.send("Email queued!");
});
```

✅ **Improves response time by deferring background tasks.**

Chapter 3: Frontend Optimization in React

3.1 Optimize React Rendering & Re-renders

Use React.memo to Avoid Unnecessary Rerenders

```
import React, { memo } from "react";


const UserCard = memo(({ user }) => {
  console.log("Rendering:", user.name);
  return <p>{user.name}</p>;
});
```

✅ **Prevents re-renders when props haven't changed.**

Use useCallback for Stable Functions

```
const handleClick = useCallback(() => {
  console.log("Button clicked!");
}, []);
```

✅ **Prevents unnecessary recreation of functions.**

Use Virtualization for Large Lists

Install react-window to optimize large lists.

npm install react-window

import { FixedSizeList } from "react-window";

const Row = ({ index, style }) => Row {index} ;

const ListView = () => (

<FixedSizeList height={400} width={300} itemSize={50} itemCount={1000}>

{Row}

</FixedSizeList>

);

✅ **Renders only visible items instead of loading all at once.**

3.2 Optimize Static Assets

Enable Gzip Compression in React Build

Add this to package.json:

"homepage": ".",

"build": "react-scripts build && gzip -r build"

✅ **Reduces JavaScript bundle size for faster page loads.**

Use Code Splitting for Faster Initial Load

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

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

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

✅ **Loads components only when needed.**

Use Lazy Loading for Images

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

✅ **Speeds up initial page load by deferring image loading.**

CHAPTER 4: NETWORK OPTIMIZATION

4.1 Optimize API Calls with Debouncing

Prevents excessive API calls on input changes.

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


function useDebounce(value, delay) {
  const [debouncedValue, setDebouncedValue] = useState(value);


  useEffect(() => {
    const handler = setTimeout(() => setDebouncedValue(value), delay);
    return () => clearTimeout(handler);
  }, [value, delay]);


  return debouncedValue;
}
```

✅ **Delays API calls until user stops typing.**

4.2 Enable HTTP/2 for Faster Requests

Modify **Express.js** server to use HTTP/2.

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

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


const options = {
  key: fs.readFileSync("./server.key"),
  cert: fs.readFileSync("./server.cert")
};


spdy.createServer(options, app).listen(PORT, () => {
  console.log(`Server running on HTTPS/2 at port ${PORT}`);
});
```

✅ **Reduces latency by multiplexing multiple requests over a single connection.**

CHAPTER 5: HANDS-ON PRACTICE & ASSIGNMENTS

Exercise 1: Optimize Backend Performance

1. **Enable response compression** in Express.
2. **Implement MongoDB indexing** for the email field in UserSchema.
3. **Cache responses** for the /api/users endpoint using Redis.

Exercise 2: Optimize React Performance

1. **Use React.memo** to prevent unnecessary re-renders.
2. **Lazy load images** in your React app.

3. **Implement API call debouncing** in a search bar.

Exercise 3: Reduce Network Latency

1. **Enable HTTP/2** in the Express server.
2. **Implement pagination** in the /api/products endpoint.

Conclusion

🎉 You have successfully learned how to:
✅ Optimize **MongoDB queries** using indexing & pagination.
✅ Reduce **API response time** using caching & compression.
✅ Improve **React performance** using memoization, lazy loading, and virtualization.
✅ Optimize **network requests** with HTTP/2 and API debouncing.

🚀 **Next Steps:**

- Implement **Server-Side Rendering (SSR) with Next.js**.
- Deploy optimized MERN apps on **AWS, Heroku, or Vercel**.
- Monitor performance using **Lighthouse & New Relic**.

Happy Coding! 🎉 🚀

ASSIGNMENTS

- DEVELOP A REAL-TIME CHAT APP WITH SOCKET.IO
- WRITE TEST CASES FOR A MERN API USING JEST

Assignment Solution & Step-by-Step Guide: Develop a Real-Time Chat App with Socket.io

Objective

In this assignment, we will build a **real-time chat application** using **Socket.io, Node.js, Express.js, and React.js**. The app will allow users to:

✅ **Join a chat room** and select a username.
✅ **Send and receive messages in real-time**.
✅ **See a list of active users in the chat room**.

Step 1: Setting Up the Backend with Node.js & Express.js

1.1 Initialize a Node.js Project

Create a new project folder and initialize a Node.js project:

mkdir socket-chat-app

cd socket-chat-app

npm init -y

This creates a package.json file.

1.2 Install Required Dependencies

npm install express socket.io cors dotenv

- **express** – Web framework for handling API routes.
- **socket.io** – Enables real-time WebSocket communication.
- **cors** – Allows frontend and backend to communicate.
- **dotenv** – Manages environment variables.

1.3 Set Up the Express Server

Create a file named server.js and set up the Express and Socket.io server:

```
require("dotenv").config();
const express = require("express");
const http = require("http");
const { Server } = require("socket.io");
```

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

const app = express();
const server = http.createServer(app);
const io = new Server(server, {
  cors: {
    origin: "http://localhost:3000", // Allow requests from React frontend
    methods: ["GET", "POST"]
  }
});

app.use(cors());

io.on("connection", (socket) => {
  console.log(`User connected: ${socket.id}`);

  // Listen for incoming messages
  socket.on("send_message", (data) => {
    io.emit("receive_message", data); // Broadcast message to all users
  });

  // Handle user disconnect
  socket.on("disconnect", () => {
    console.log(`User disconnected: ${socket.id}`);
  });
});

const PORT = process.env.PORT || 5000;
server.listen(PORT, () => {
```

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

✅ **Sets up a basic Socket.io server** that listens for connections and broadcasts messages.

Step 2: Setting Up the Frontend with React.js

2.1 Create a React App

Open a new terminal and create a React project:

```
npx create-react-app chat-frontend
cd chat-frontend
```

2.2 Install Required Dependencies

```
npm install socket.io-client
```

- **socket.io-client** – Enables the frontend to connect with the backend.

2.3 Create a Simple Chat Component

Inside src/, create a file Chat.js:

```
import React, { useState, useEffect } from "react";
import io from "socket.io-client";

const socket = io.connect("http://localhost:5000");

function Chat() {
  const [message, setMessage] = useState("");
  const [messages, setMessages] = useState([]);

  useEffect(() => {
    socket.on("receive_message", (data) => {
      setMessages((prevMessages) => [...prevMessages, data]);
    });
  }, []);
```

```
  const sendMessage = () => {
    if (message.trim() !== "") {
      socket.emit("send_message", message);
      setMessage("");
    }
  };

  return (
    <div>
      <h2>Real-Time Chat</h2>
      <div>
        {messages.map((msg, index) => (
          <p key={index}>{msg}</p>
        ))}
      </div>
      <input
        type="text"
        value={message}
        onChange={(e) => setMessage(e.target.value)}
        placeholder="Type a message..."
      />
      <button onClick={sendMessage}>Send</button>
    </div>
  );
}

export default Chat;
```

✅ **Establishes a connection with the backend and handles real-time messages.**

Step 3: Integrating the Chat Component in App.js

Modify src/App.js to include the chat component:

```
import React from "react";
import Chat from "./Chat";

function App() {
  return (
    <div>
      <h1>Welcome to the Real-Time Chat App</h1>
      <Chat />
    </div>
  );
}

export default App;
```

Step 4: Running the Application

4.1 Start the Backend

```
node server.js
```

or

```
nodemon server.js
```

4.2 Start the React Frontend

In a new terminal, run:

```
npm start
```

4.3 Open Multiple Browser Tabs

1. Open http://localhost:3000 in multiple tabs.
2. Type messages in the chat box and see real-time updates across tabs.

✅ **Your real-time chat app is now fully functional!**

Step 5: Enhancing the Chat Application

5.1 Add Usernames

Modify the Chat.js file to include usernames:

```
const [username, setUsername] = useState("");
const [userJoined, setUserJoined] = useState(false);


const joinChat = () => {
  if (username.trim() !== "") {
    setUserJoined(true);
  }
};
```

Modify the UI to show a username input before joining:

```
return (
  <div>
    {!userJoined ? (
      <div>
        <input
          type="text"
          placeholder="Enter your name..."
          onChange={(e) => setUsername(e.target.value)}
        />
        <button onClick={joinChat}>Join Chat</button>
      </div>
    ) : (
      <div>
        <h2>Real-Time Chat</h2>
        <div>
          {messages.map((msg, index) => (
```

```
            <p key={index}><strong>{msg.username}:</strong> {msg.text}</p>
          ))}
        </div>
        <input
          type="text"
          value={message}
          onChange={(e) => setMessage(e.target.value)}
          placeholder="Type a message..."
        />
        <button onClick={sendMessage}>Send</button>
      </div>
    )}
  </div>
);
```

Modify the sendMessage function to include the username:

```
const sendMessage = () => {
  if (message.trim() !== "") {
    const messageData = { username, text: message };
    socket.emit("send_message", messageData);
    setMessage("");
  }
};
```

Modify the backend in server.js to support usernames:

```
socket.on("send_message", (data) => {
  io.emit("receive_message", data);
});
```

✅ **Users can now enter a username before joining the chat.**

Step 6: Deploying the Chat App

6.1 Deploying the Backend to Heroku

1. Install Heroku CLI:
2. npm install -g heroku
3. Initialize Git:
4. git init
5. git add .
6. git commit -m "Initial commit"
7. Create a Heroku app and deploy:
8. heroku create socket-chat-app
9. git push heroku master

✅ **Your backend is now deployed on Heroku!**

6.2 Deploying the Frontend to Netlify

1. Build the React app:
2. npm run build
3. Upload the build/ folder to Netlify.
   ✅ **Your frontend is now live on Netlify!**

Step 7: Assignments & Enhancements

Assignment 1: Add Chat Rooms

- Modify the backend to allow multiple chat rooms.
- Use socket.join(roomName) to manage rooms.

Assignment 2: Implement Message History

- Store chat messages in a **MongoDB database**.
- Fetch old messages when a user joins.

Assignment 3: Add Typing Indicators

- Show "User is typing..." when a user types.
- Use socket.emit("typing") and socket.on("typing") events.

Conclusion

✅ **Built a fully functional real-time chat app using Socket.io, Node.js, and React.js.**
✅ **Enhanced the chat app with usernames and rooms.**
✅ **Deployed the app on Heroku & Netlify.**

🚀 **Next Steps:**

- Implement **user authentication with JWT**.
- Add **private messaging** between users.
- Build a **mobile-friendly UI with Material-UI or TailwindCSS**!

By following this guide, you can now create and enhance **real-time chat applications with Socket.io!** 🎉 🔥

Assignment Solution: Write Test Cases for a MERN API Using Jest

Objective

The goal of this assignment is to **write and execute test cases** for a **MERN (MongoDB, Express.js, React, Node.js) API** using **Jest** and **Supertest**. You will test CRUD operations for an API that manages user data.

Step 1: Setup the MERN API for Testing

1.1 Install Required Dependencies

Navigate to your **MERN backend project** and install the following testing libraries:

npm install --save-dev jest supertest mongodb-memory-server dotenv

Dependencies:

- jest – JavaScript testing framework.
- supertest – For testing API endpoints.
- mongodb-memory-server – Creates an in-memory database for testing.
- dotenv – Loads environment variables for the database.

1.2 Update package.json for Jest Configuration

Modify the package.json file to include:

```
"scripts": {
  "test": "jest --testTimeout=10000"
}
```

✅ **Ensures Jest runs with an extended timeout.**

Step 2: Setup Express.js API for Testing

2.1 Sample Express.js API (server.js)

Create an Express.js API for user management:

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

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

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

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

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

// Get All Users
app.get("/users", async (req, res) => {
  const users = await User.find();
  res.json(users);
});
```

```
// Get User by ID
app.get("/users/:id", async (req, res) => {
  const user = await User.findById(req.params.id);
  if (!user) return res.status(404).json({ error: "User not found" });
  res.json(user);
});

// Update User
app.put("/users/:id", async (req, res) => {
  const user = await User.findByIdAndUpdate(req.params.id, req.body, { new: true });
  if (!user) return res.status(404).json({ error: "User not found" });
  res.json(user);
});

// Delete User
app.delete("/users/:id", async (req, res) => {
  const user = await User.findByIdAndDelete(req.params.id);
  if (!user) return res.status(404).json({ error: "User not found" });
  res.json({ message: "User deleted" });
});

module.exports = app;
```

✅ **This API supports CRUD operations for users.**

Step 3: Setting Up Jest for Testing

3.1 Create Test Environment (test/setup.js)

We need to use **mongodb-memory-server** to create a temporary MongoDB instance for tests.

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

```
const { MongoMemoryServer } = require("mongodb-memory-server");

let mongoServer;

module.exports = {
  connect: async () => {
    mongoServer = await MongoMemoryServer.create();
    const uri = mongoServer.getUri();
    await mongoose.connect(uri, { useNewUrlParser: true, useUnifiedTopology: true
});
  },
  disconnect: async () => {
    await mongoose.disconnect();
    await mongoServer.stop();
  }
};
```

✅ **Ensures tests run in an isolated in-memory database.**

Step 4: Writing Test Cases Using Jest & Supertest

4.1 Create tests/user.test.js

```
const request = require("supertest");
const mongoose = require("mongoose");
const app = require("../server");
const { connect, disconnect } = require("./setup");

beforeAll(async () => {
  await connect();
});
```

```
afterAll(async () => {
  await disconnect();
});

// Test Case 1: Create a User
describe("POST /users", () => {
  it("should create a new user", async () => {
    const res = await request(app).post("/users").send({
      name: "John Doe",
      email: "johndoe@example.com"
    });

    expect(res.statusCode).toBe(201);
    expect(res.body).toHaveProperty("_id");
    expect(res.body.name).toBe("John Doe");
  });
});

// Test Case 2: Get All Users
describe("GET /users", () => {
  it("should fetch all users", async () => {
    const res = await request(app).get("/users");
    expect(res.statusCode).toBe(200);
    expect(Array.isArray(res.body)).toBeTruthy();
  });
});

// Test Case 3: Get User by ID
describe("GET /users/:id", () => {
```

```
  it("should fetch a user by ID", async () => {
    const user = await request(app).post("/users").send({
      name: "Alice",
      email: "alice@example.com"
    });

    const res = await request(app).get(`/users/${user.body._id}`);
    expect(res.statusCode).toBe(200);
    expect(res.body.name).toBe("Alice");
  });

  it("should return 404 if user is not found", async () => {
    const fakeId = new mongoose.Types.ObjectId();
    const res = await request(app).get(`/users/${fakeId}`);
    expect(res.statusCode).toBe(404);
  });
});

// Test Case 4: Update a User
describe("PUT /users/:id", () => {
  it("should update an existing user", async () => {
    const user = await request(app).post("/users").send({
      name: "Charlie",
      email: "charlie@example.com"
    });

    const res = await request(app).put(`/users/${user.body._id}`).send({
      name: "Charlie Updated"
    });
```

```
    expect(res.statusCode).toBe(200);
    expect(res.body.name).toBe("Charlie Updated");
  });
});

// Test Case 5: Delete a User
describe("DELETE /users/:id", () => {
  it("should delete a user", async () => {
    const user = await request(app).post("/users").send({
      name: "David",
      email: "david@example.com"
    });

    const res = await request(app).delete(`/users/${user.body._id}`);
    expect(res.statusCode).toBe(200);
    expect(res.body.message).toBe("User deleted");
  });

  it("should return 404 if user is not found", async () => {
    const fakeId = new mongoose.Types.ObjectId();
    const res = await request(app).delete(`/users/${fakeId}`);
    expect(res.statusCode).toBe(404);
  });
});
```

✅ **Covers all CRUD operations with Jest and Supertest.**

Step 5: Running Tests

5.1 Execute Jest Tests

Run the following command in your terminal:

npm test

✅ **Jest will execute all test cases and display the results.**

Step 6: Expected Output of Test Cases

PASS tests/user.test.js

✓ should create a new user (50 ms)

✓ should fetch all users (30 ms)

✓ should fetch a user by ID (40 ms)

✓ should return 404 if user is not found (20 ms)

✓ should update an existing user (35 ms)

✓ should delete a user (25 ms)

✓ should return 404 if user is not found (15 ms)

✅ **All test cases should pass if the API is correctly implemented.**

CONCLUSION

- **Jest & Supertest allow for comprehensive API testing**.
- **Mongodb-memory-server creates an isolated test database**.
- **Tests ensure API endpoints work correctly before deployment**.
- **Automated testing saves debugging time and prevents breaking changes**.

🚀 **Next Steps:**

- Implement **authentication tests (JWT verification)**.
- Add **pagination tests for large datasets**.
- Integrate tests in **CI/CD pipelines (GitHub Actions, Jenkins)**.

By following this **step-by-step guide**, you have successfully **written and executed Jest test cases** for a **MERN API**! 🎉🚀