Backend Development Lab Curriculum

AUTHOR: @Mithun Kumar S R

INITIAL DRAFT: 13 AUGUST 2024

LAST UPDATED: 8 OCTOBER 2024

Lab Curriculum Plan

Week 1: Introduction to Backend Development

- **Objective:** Understand the basics of backend development.
- **Lab Activities:**
  - Overview of backend vs. frontend development.
  - Introduction to server-side programming languages (Node.js, Python, Java).
  - Setting up the development eods and RESTful API principles.
- **Lab Activities:**
  - Overview of HTTP methods (GET, POST, PUT, DELETE).
  - Introduction to REST architecture.
  - Building a simple RESTful API.
- **Sample Project:** Create an API for managing a list of items (e.g., a to-do list).

Week 2: HTTP and RESTful APIs

- **Objective:** Learn about HTTP methods and RESTful API principles.
- **Lab Activities:**
  - Overview of HTTP methods (GET, POST, PUT, DELETE).
  - Introduction to REST architecture.
  - Building a simple RESTful API.
- **Sample Project:** Create an API for managing a list of items (e.g., a to-do list).

Week 3: Database Integration

- **Objective:** Integrate databases with backend applications.
- **Lab Activities:**

    - Introduction to relational a Performing basic CRUD operations.
    - nd non-relational databases (MySQL, MongoDB).
    - Setting up and connecting to a database.
- **Sample Project:** Extend the to-do list API to store items in a database.

Week 4: Authentication and Authorization

- **Objective:** Implement user authentication and authorization.
  Lab Activities:
    - Overview of authentication methods (Basic, OAuth, JWT).
    - Implementing user registration and login functionality.
    - Securing API routes based on user roles.
- **Sample Project:** Add user authentication to the to-do list API.

Week 5: Middleware and Routing

- **Objective:** Understand middleware functions and advanced routing.
- **Lab Activities:**
    - Introduction to middleware in backend frameworks.
    - Creating custom middleware for logging, error handling, etc.
    - Advanced routing techniques (nested routes, dynamic routes).
- **Sample Project:** Implement logging and error handling middleware for the API.

Week 6: Data Validation and Error Handling

- **Objective:** Learn to validate data and handle errors gracefully.
- **Lab Activities:**
    - Implementing input validation with libraries (e.g., Joi, Express Validator).
    - Structuring and handling API errors.
    - Sending proper HTTP status codes and error messages.
- **Sample Project:** Add data validation and error handling to the API.

Week 7: Working with External APIs

- **Objective:** Integrate third-party APIs into backend applications.
- **Lab Activities:**
    - Overview of working with external APIs (e.g., weather, payment, or social media APIs).
    - Fetching data from external APIs using HTTP clients (Axios, Requests).

    - Parsing and processing API responses.
- **Sample Project:** Integrate a third-party API (e.g., weather data) into the application.

Week 8: File Uploads and Management

- **Objective:** Implement file upload functionality in backend applications.
- **Lab Activities:**
    - Handling file uploads in the backend.
    - Storing files locally vs. using cloud storage services.
    - Managing file uploads and downloads.
- **Sample Project:** Add an image upload feature to the API and store images on the server.

Week 9: WebSocket and Real-time Communication

- **Objective:** Implement real-time communication using WebSockets.
- **Lab Activities:**
    - Introduction to WebSockets and real-time data transfer.
    - Setting up a WebSocket server.
    - Building a real-time chat or notification system.
- **Sample Project:** Add a real-time chat feature to the API using WebSockets.

Week 10: Deployment and Scaling

- **Objective:** Learn to deploy and scale backend applications.
- **Lab Activities:**
    - Introduction to deployment options (Heroku, AWS, DigitalOcean).
    - Deploying the backend application to a cloud platform.
    - Basics of scaling applications and load balancing.
- **Sample Project:** Deploy the to-do list API to a cloud service.

Week 11: Testing and Debugging

- **Objective:** Implement testing strategies for backend applications.
- **Lab Activities:**
    - Overview of testing types (unit, integration, end-to-end).
    - Writing and running tests with tools like Mocha, Jest, or PyTest.
    - Debugging common backend issues.

- **Sample Project:** Write unit tests for the to-do list API.

Week 12: Capstone Project

- **Objective:** Apply learned concepts in a comprehensive project.
- **Lab Activities:**
  - Students choose or are assigned a capstone project.
  - Work on developing a fully functional backend application.
  - Prepare for project presentation and code review.
- **Sample Project:** Build a complete backend system (e.g., an e-commerce API, a social media backend, or a blog platform).

Week 1 Introduction and setup

1. Node.js with Express

Step 1: Install Node.js

- Download and install Node.js from the official website.
- Verify the installation by running:

```
node -v
npm -v
```

Step 2: Set Up the Project

- Create a new directory for your project and navigate into it:

```
mkdir hello-world-node
cd hello-world-node
```

- Initialize a new Node.js project:

```
npm init -y
```

Step 3: Install Express

- Install the Express.js framework:

```
npm install express
```

Step 4: Create the Server File

- Create a file named index.js in the project directory and add the following code:

```
const express = require('express');
const app = express();
const port = 3000;

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

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

Step 5: Run the Server

- Start the server by running:

```
node index.js
```

- Open your browser and go to http://localhost:3000, or use a tool like Postman to see the "Hello, World!" message.

Sample Node.js App: Simple To-Do List API

1. Set Up the Project

1. **Initialize the Project** Open a terminal and create a new directory for the project, then navigate into it:

```
mkdir todo-api
cd todo-api
```

2. **Initialize npm** Initialize a new Node.js project with default settings:

```
npm init -y
```

3. **Install Dependencies** Install Express.js and a few other useful packages:

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

2. Create the Project Files

4. **Create server.js**

In the todo-api directory, create a file named server.js and add the following code:

javascript

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

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

app.use(bodyParser.json());

// Sample in-memory database
let todos = [
  { id: 1, task: 'Learn Node.js', completed: false },
  { id: 2, task: 'Build a RESTful API', completed: false }
];

// Get all todos
app.get('/todos', (req, res) => {
  res.json(todos);
});

// Get a single todo by ID
app.get('/todos/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const todo = todos.find(t => t.id === id);
  if (todo) {
    res.json(todo);
  } else {
    res.status(404).send('Todo not found');
  }
});

// Create a new todo
app.post('/todos', (req, res) => {
  const newTodo = req.body;
  newTodo.id = todos.length ? todos[todos.length - 1].id + 1 : 1;
  todos.push(newTodo);
  res.status(201).json(newTodo);
});
```

```
// Update a todo
app.put('/todos/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const index = todos.findIndex(t => t.id === id);
  if (index !== -1) {
    todos[index] = { id, ...req.body };
    res.json(todos[index]);
  } else {
    res.status(404).send('Todo not found');
  }
});

// Delete a todo
app.delete('/todos/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const index = todos.findIndex(t => t.id === id);
  if (index !== -1) {
    todos.splice(index, 1);
    res.status(204).send();
  } else {
    res.status(404).send('Todo not found');
  }
});

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

5. **Run the Server** Start the server by running:

bash

```
node server.js
```

The API will be available at http://localhost:3000.

3. Testing the API

You can use Postman or cURL to test the API endpoints.

6. **Get All Todos**
    - **URL**: GET http://localhost:3000/todos
    - **Response**: List of all todos.
7. **Get a Single Todo**
    - **URL**: GET http://localhost:3000/todos/{id} (replace {id} with a specific ID)
    - **Response**: Details of the specified todo.
8. **Create a New Todo**
    - **URL**: POST http://localhost:3000/todos
    - **Body**: JSON object, e.g., { "task": "New Task", "completed": false }
    - **Response**: Created todo object.
9. **Update a Todo**
    - **URL**: PUT http://localhost:3000/todos/{id} (replace {id} with a specific ID)
    - **Body**: JSON object with updated fields, e.g., { "task": "Updated Task", "completed": true }
    - **Response**: Updated todo object.
10. **Delete a Todo**
    - **URL**: DELETE http://localhost:3000/todos/{id} (replace {id} with a specific ID)
    - **Response**: No content (204 status) or error message if not found.

Week 2: Hands-On Exercise: Explore RESTful APIs with Postman or cURL

Introduction to RESTful APIs

REST (Representational State Transfer) is an architectural style used for designing networked applications. RESTful APIs are web services that adhere to REST principles, allowing systems to communicate over HTTP in a simple, stateless manner. Here's an introduction to the key concepts:

1. What is a RESTful API?

- **API (Application Programming Interface):** A set of rules that allows different software entities to communicate with each other. It defines the methods and data formats that applications use to interact.
- **RESTful API:** An API that conforms to the principles of REST. It's commonly used to expose web services and allow interaction with web-based resources.

2. Key REST Principles

11. **Statelessness:**
    - Each request from a client to the server must contain all the information needed to understand and process the request. The server does not store any session state about the client.
    - This principle simplifies server design and improves scalability, as each request is independent.
12. **Client-Server Architecture:**
    - The client and server are separate entities that interact over a network. The client is responsible for the user interface and user experience, while the server handles data storage and processing.
    - This separation allows for greater flexibility and modularity.
13. **Uniform Interface:**
    - A uniform and consistent interface between the client and server simplifies the architecture. This includes standardized methods (e.g., HTTP methods), URIs, and data formats (typically JSON or XML).
    - A uniform interface improves interoperability between different systems.
14. **Resource-Based:**

- Everything in REST is considered a resource, which can be an object, data, or service that can be accessed via a URL (Uniform Resource Locator).
- Resources are typically represented as JSON or XML, and they are manipulated using HTTP methods.

15. **Layered System:**
    - The API architecture is divided into layers, each with its specific role. For example, one layer might handle security, another caching, and another data storage.
    - This layering allows for more modular and scalable systems.
16. **Cacheability:**
    - Responses from the server should be explicitly marked as cacheable or non-cacheable to improve efficiency. If a response is cacheable, clients can store and reuse it without contacting the server again.
    - Proper caching can significantly reduce the load on the server and improve response times.
17. **Code on Demand (Optional):**
    - Servers can extend the functionality of a client by transferring executable code. For example, the server can send JavaScript that the client executes.
    - This principle is optional and is used less frequently in RESTful APIs.

3. RESTful API Structure

18. **Resources:**
    - Resources are the primary entities exposed by a RESTful API. Each resource has a unique URL, and resources can be anything: users, articles, products, etc.
    - For example:
        - GET /users - Retrieves a list of users.
        - GET /users/1 - Retrieves the user with ID 1.
19. **HTTP Methods:**
    - RESTful APIs use standard HTTP methods to interact with resources:
        - **GET:** Retrieve data from the server (e.g., GET /items).
        - **POST:** Create a new resource on the server (e.g., POST /items).
        - **PUT:** Update an existing resource (e.g., PUT /items/1).
        - **DELETE:** Remove a resource from the server (e.g., DELETE /items/1).
20. **HTTP Status Codes:**
    - Status codes are used to indicate the outcome of an API request:
        - **200 OK:** The request was successful.

        - **201 Created:** A new resource was successfully created.
        - **400 Bad Request:** The request was invalid or cannot be processed.
        - **404 Not Found:** The requested resource does not exist.
        - **500 Internal Server Error:** An error occurred on the server.

21. **Data Format:**
    - RESTful APIs typically use JSON (JavaScript Object Notation) to represent data, though XML is also used in some cases. JSON is lightweight, easy to read, and widely supported.

Example of a JSON response:

json

```
{
  "id": 1,
  "name": "Item 1",
  "description": "This is item 1"
}
```

4. Advantages of RESTful APIs

- **Scalability:** RESTful APIs are stateless, allowing them to scale easily with client requests.
- **Flexibility:** RESTful APIs can handle different types of calls, return different data formats, and even allow changes structurally with the hypermedia.
- **Performance:** Caching responses can significantly improve performance.
- **Simplicity:** REST uses standard HTTP methods and is easy to understand and implement.

5. Use Cases for RESTful APIs

- **Web and Mobile Applications:** RESTful APIs are commonly used to provide the backend for web and mobile applications.
- **Microservices:** RESTful APIs enable communication between microservices, making them suitable for complex systems with multiple interacting services.
- **Integration:** RESTful APIs allow different systems and applications to interact and exchange data, facilitating integration across platforms.

Hands-on exercise for week 2:

Objective

To familiarize students with using Postman or cURL for interacting with and testing RESTful APIs.

Materials Needed

- **Postman**: A tool for testing APIs with a user-friendly interface.
- **cURL**: A command-line tool for making HTTP requests.

Preparation

1. **API Endpoint**: Provide a simple, public API endpoint or a sample API you have set up for the exercise. For instance, you could use a mock API service like JSONPlaceholder for this exercise.
2. **Instructions Document**: Create a document with step-by-step instructions and screenshots to guide students through the exercise.

Exercise Steps

Using Postman

3. **Introduction to Postman**
    - **Download and Install**: Ensure students have Postman installed on their computers. If not, guide them to Postman's download page.
    - **Basic Navigation**: Show students how to navigate the Postman interface.
4. **Create a New Request**
    - **Open Postman**: Launch Postman.
    - **New Request**: Click on "New" and select "Request".
    - **Request Setup**: Name the request (e.g., "Test GET Request") and choose a collection or create a new one to save it.
5. **Send a GET Request**
    - **URL**: Enter a URL for a GET request. For example, https://jsonplaceholder.typicode.com/posts.

- **Send Request**: Click on “Send” and observe the response.
- **Analyze Response**: Review the response status, headers, and body.

6. **Send a POST Request**
   - **Setup**: Change the HTTP method to POST and enter a URL for a POST request (e.g., https://jsonplaceholder.typicode.com/posts).
   - **Body**: Go to the “Body” tab, select “raw”, and enter a JSON payload (e.g., {"title": "foo", "body": "bar", "userId": 1}).
   - **Send Request**: Click “Send” and review the response.
7. **Explore Other Methods**
   - **PUT/PATCH**: Show how to update resources using PUT or PATCH methods.
   - **DELETE**: Demonstrate how to delete a resource using the DELETE method.
8. **Save and Document**
   - **Save Requests**: Save all requests in a collection.
   - **Documentation**: Use Postman’s built-in documentation feature to generate API documentation.

Using cURL

9. **Introduction to cURL**
   - **Install cURL**: Ensure students have cURL installed. They can download it from cURL’s website if necessary.
   - **Basic Commands**: Explain basic cURL commands and syntax.
10. **Send a GET Request**
    - **Command**: Open a terminal and run curl https://jsonplaceholder.typicode.com/posts.
    - **Analyze Response**: Review the response directly in the terminal.
11. **Send a POST Request**
    - **Command**: Use curl -X POST https://jsonplaceholder.typicode.com/posts -H "Content-Type: application/json" -d '{"title": "foo", "body": "bar", "userId": 1}'.
    - **Analyze Response**: Review the response in the terminal.
12. **Explore Other Methods**
    - **PUT/PATCH**: Demonstrate how to update resources using curl -X PUT or curl -X PATCH.
    - **DELETE**: Show how to delete a resource with curl -X DELETE.
13. **Save and Document**
    - **Save Commands**: Encourage students to save their cURL commands in a text file for future reference.

- **Document**: Ask students to document the commands and responses in their exercise report.

Post-Exercise Discussion

- **Review**: Discuss the different types of requests and responses.
- **Q&A**: Answer any questions students may have about using Postman or cURL.
- **Challenges**: Provide additional challenges or questions to deepen their understanding (e.g., explore authentication or error handling).

Assessment

- **Submission**: Have students submit screenshots of their Postman requests and responses or a text file with their cURL commands and responses.
- **Evaluation**: Assess their ability to correctly use Postman or cURL to interact with the API and understand the responses.

This exercise should give students a practical understanding of how to interact with RESTful APIs and prepare them for more complex tasks.

Detailed Steps for Building a Simple RESTful API with Node.js using Express

This tutorial walks through building a basic CRUD RESTful API in Node.js using the Express framework. The API will manage a collection of items.

1. Setup Environment

Step 1.1: Install Node.js and npm

- Ensure Node.js and npm are installed on your machine. You can check this by running the following commands:

```
node -v
npm -v
```

Step 1.2: Create a Project Directory

- Create a directory for your project and navigate into it:

```
mkdir rest-api-example
cd rest-api-example
```

Step 1.3: Initialize the Project

- Initialize a new Node.js project. This will create a package.json file:

```
npm init -y
```

Step 1.4: Install Express

- Install the express framework along with nodemon (optional) for automatic server restarts:

```
npm install express
npm install --save-dev nodemon
```

2. Create the Basic Server

Step 2.1: Create the index.js File

- In the project directory, create a new file named index.js:

```
touch index.js
```

Step 2.2: Import Express and Set Up Basic Server

- Inside index.js, import Express and set up a basic server to listen on a specific port:

```
const express = require('express');
const app = express();
const port = 3000;

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

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

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

Step 2.3: Run the Server

- Run your server using Node.js or nodemon:

```
node index.js
# or use nodemon
npx nodemon index.js
```

- You should see the server running on http://localhost:3000 with the message "Hello World!" when you visit that URL in your browser.

3. Building CRUD Endpoints

We'll build a simple API that manages a list of items. Each item will have an id and a name.

Step 3.1: Create an In-Memory Data Store

- In index.js, create an in-memory data store to simulate a database. This will be an array of items:

```
javascript

let items = [
  { id: 1, name: 'Item One' },
  { id: 2, name: 'Item Two' }
];
```

Step 3.2: Define Routes

GET /items - Retrieve All Items

- Define a route to get all items:

javascript

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

GET /items/

- Retrieve a Single Item by ID

- Define a route to get an item by id:

javascript

```
app.get('/items/:id', (req, res) => {
  const item = items.find(i => i.id === parseInt(req.params.id));
  if (!item) return res.status(404).send('Item not found');
  res.json(item);
});
```

POST /items - Create a New Item

- Define a route to create a new item:

javascript

```
app.post('/items', (req, res) => {
  const newItem = {
    id: items.length + 1,
    name: req.body.name
  };
  items.push(newItem);
```

```
  res.status(201).json(newItem);
});
```

PUT /items/

- Update an Existing Item

- Define a route to update an existing item:

javascript

```
app.put('/items/:id', (req, res) => {
  const item = items.find(i => i.id === parseInt(req.params.id));
  if (!item) return res.status(404).send('Item not found');

  item.name = req.body.name;
  res.json(item);
});
```

DELETE /items/

- Delete an Item

- Define a route to delete an item:

javascript

```
app.delete('/items/:id', (req, res) => {
  const itemIndex = items.findIndex(i => i.id === parseInt(req.params.id));
  if (itemIndex === -1) return res.status(404).send('Item not found');

  const deletedItem = items.splice(itemIndex, 1);
  res.json(deletedItem);
});
```

4. Testing the API

Step 4.1: Testing with Postman or curl

22. **Retrieve All Items:**
    - Method: GET
    - URL: http://localhost:3000/items
    - Response:

json

```
[
 { "id": 1, "name": "Item One" },
 { "id": 2, "name": "Item Two" }
]
```

23. **Retrieve Single Item:**
    - Method: GET
    - URL: http://localhost:3000/items/1
    - Response:

json

```
{ "id": 1, "name": "Item One" }
```

24. **Create a New Item:**
    - Method: POST
    - URL: http://localhost:3000/items
    - Body (JSON):

json

```
{ "name": "Item Three" }
```

    - Response:

json

```
{ "id": 3, "name": "Item Three" }
```

25. **Update an Existing Item:**
    - Method: PUT
    - URL: http://localhost:3000/items/2
    - Body (JSON):

json

{ "name": "Updated Item Two" }

    - Response:

json

{ "id": 2, "name": "Updated Item Two" }

26. **Delete an Item:**
    - Method: DELETE
    - URL: http://localhost:3000/items/1
    - Response:

json

{ "id": 1, "name": "Item One" }

5. Enhancements

Step 5.1: Add Validation

- Add validation to check if the input is valid (e.g., ensure name is not empty) before adding or updating an item.

Step 5.2: Persist Data

- Optionally, replace the in-memory array with a real database (e.g., MongoDB, PostgreSQL).

Step 5.3: Error Handling

- Improve error handling for cases such as invalid data types, missing parameters, or server errors.

Week 3: Database Integration

Slides: Introduction_to_Relational_and_Non-Relational_Databases.pptx

Detailed Instructions for Installing MongoDB Locally and Connecting to Node.js

1. Installing MongoDB Locally

For Windows:

27. **Download MongoDB Installer**:
    - Visit the MongoDB Download Center.
    - Select the appropriate version for your operating system (usually, the MSI package for Windows).
    - Download the installer.
28. **Run the Installer**:
    - Double-click the downloaded .msi file.
    - Follow the prompts to install MongoDB.
    - Choose the "Complete" installation option.
    - Ensure that "Install MongoDB as a Service" is checked.
    - Optionally, you can also install MongoDB Compass, a GUI tool for managing your databases.
29. **Verify Installation**:
    - Open Command Prompt and type mongo --version to check if MongoDB was installed correctly.
30. **Starting MongoDB**:
    - MongoDB runs as a service by default. If it isn't running, you can start it by typing net start MongoDB in the Command Prompt.

For macOS:

31. **Install Homebrew (if not already installed)**:
    - Open Terminal and run the following command to install Homebrew (if you don't have it already):

/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

32. **Install MongoDB**:
    - Run the following command to install MongoDB using Homebrew:

brew tap mongodb/brew
brew install mongodb-community@6.0

33. **Start MongoDB**:
    - To start MongoDB as a service, run:

brew services start mongodb/brew/mongodb-community

34. **Verify Installation**:
    - Check if MongoDB is running by typing mongo --version in the Terminal.

For Linux (Ubuntu):

35. **Import the Public Key**:
    - Open Terminal and run:

bash

wget -qO - https://www.mongodb.org/static/pgp/server-6.0.asc | sudo apt-key add -

36. **Create the List File**:
    - Run the following command to create a list file for MongoDB:

bash

echo "deb [arch=amd64,arm64] https://repo.mongodb.org/apt/ubuntu $(lsb_release -cs)/mongodb-org/6.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-6.0.list

37. **Reload the Package Database**:
    - Update your package list:

bash

```
sudo apt-get update
```

38. **Install MongoDB**:
    - Install MongoDB by running:

bash

```
sudo apt-get install -y mongodb-org
```

39. **Start MongoDB**:
    - Start the MongoDB service:

bash

```
sudo systemctl start mongod
```

    - Enable MongoDB to start on boot:

bash

```
sudo systemctl enable mongod
```

40. **Verify Installation**:
    - Check if MongoDB is running by typing mongo --version in the Terminal.

2. Connecting to MongoDB from Node.js

Step 1: Create a Node.js Project

41. **Initialize a Node.js Project**:
    - Create a new directory for your project:

bash

```
mkdir mongodb-demo
cd mongodb-demo
```

- Initialize a new Node.js project:

bash

```
npm init -y
```

42. **Install Required Packages**:
    - Install mongoose, a popular MongoDB ODM (Object Data Modeling) library:

bash

```
npm install mongoose
```

Step 2: Connect to MongoDB

43. **Create a Connection Script**:
    - In the project directory, create a file named app.js.
    - Add the following code to app.js:

javascript

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

// Replace <username>, <password>, and <dbname> with your actual MongoDB credentials and database name
const uri = 'mongodb://localhost:27017/testdb';

mongoose.connect(uri, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log('Connected to MongoDB...'))
  .catch(err => console.error('Could not connect to MongoDB...', err));
```

```
const db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', function() {
  console.log("We're connected!");
});

// Define a simple schema
const Schema = mongoose.Schema;
const testSchema = new Schema({
  name: String,
  age: Number
});

// Compile model from schema
const TestModel = mongoose.model('TestModel', testSchema);

// Create a document
const testDoc = new TestModel({ name: 'John Doe', age: 30 });

// Save the document
testDoc.save((err, doc) => {
  if (err) return console.error(err);
  console.log('Document saved:', doc);
});
```

44. **Run the Application**:
    - Run the Node.js application by typing:

bash

```
node app.js
```

    - You should see the messages Connected to MongoDB... and We're connected! in the console, confirming a successful connection.
    - A document should be saved to the testdb database, and you should see the saved document details in the console.

Step 3: Verify the Connection in MongoDB Compass

45. **Open MongoDB Compass**:
    - If you installed MongoDB Compass, open it to visualize your database.
    - Connect to mongodb://localhost:27017 in Compass.
46. **Check the Database**:
    - Navigate to the testdb database.
    - Verify that the TestModel collection exists and contains the document you saved.

Performing Basic CRUD Operations

- **Duration**: 1.5-2 hours
- **Objective**: Implement Create, Read, Update, and Delete (CRUD) operations in a Node.js application using MySQL and MongoDB.
- **Tasks**:
    - **MySQL**:
        i. **Create**: Insert a new record into the database.
        ii. **Read**: Retrieve records from the database.
        iii. **Update**: Modify an existing record.
        iv. **Delete**: Remove a record from the database.
    - **MongoDB**:
        v. **Create**: Insert a new document into a collection.
        vi. **Read**: Retrieve documents from a collection.
        vii. **Update**: Modify an existing document.
        viii. **Delete**: Remove a document from a collection.
- **Resources**:
    - Example Node.js project with code for basic CRUD operations.
    - Documentation for MySQL queries and MongoDB operations.
- **Activity**:
    - Implement each CRUD operation in class, first for MySQL and then for MongoDB.
    - Encourage students to test their code by adding, viewing, updating, and deleting records/documents in their databases.
    - Discuss common issues and troubleshooting tips.

Part 1: Setting Up the Environment

Step 1: Initialize the Node.js Project

47. **Create a New Project Directory**:
    - Open your terminal and create a new directory:

bash

```
mkdir crud-lab
cd crud-lab
```

48. **Initialize a Node.js Project**:
    - Initialize the project with:

bash

```
npm init -y
```

49. **Install Required Packages**:
    - For MySQL:

bash

```
npm install express mysql2
```

    - For MongoDB:

```
npm install express mongoose
```

    - Also, install nodemon for easier development:

```
npm install --save-dev nodemon
```

50. **Set Up the Project Structure**:
    - Create the following files:
        - app.js (main application file)

- config.js (for configuration details like database connection strings)
- routes.js (for defining routes)
- controllers/ (directory for CRUD logic)
- models/ (directory for database schemas)

Part 2: Implementing CRUD Operations with MySQL

Step 1: Setting Up MySQL Connection

51. **Configure the Database Connection**:
    - In config.js, add the following code:

javascript

```
module.exports = {
  mysql: {
    host: 'localhost',
    user: 'your_username',
    password: 'your_password',
    database: 'crud_mysql'
  }
};
```

52. **Create a MySQL Database and Table**:
    - Open MySQL Workbench or use the command line to create a database and table:

sql

```
CREATE DATABASE crud_mysql;
USE crud_mysql;
CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100),
  email VARCHAR(100),
  age INT
);
```

53. **Connect to MySQL in app.js**:
    - Add the following code to connect to MySQL:

javascript

```
const express = require('express');
const mysql = require('mysql2');
const config = require('./config');
const app = express();

const connection = mysql.createConnection(config.mysql);

connection.connect((err) => {
 if (err) {
  console.error('Error connecting to MySQL:', err.stack);
  return;
 }
 console.log('Connected to MySQL as id', connection.threadId);
});

app.use(express.json());
app.use('/api/users', require('./routes'));

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

Step 2: Implement CRUD Operations

54. **Create a Controller for CRUD Logic**:
    - In controllers/mysqlController.js, add the following CRUD operations:

javascript

```
const connection = require('../app').connection;

exports.createUser = (req, res) => {
 const { name, email, age } = req.body;
```

```
  const query = 'INSERT INTO users (name, email, age) VALUES (?, ?, ?)';
  connection.query(query, [name, email, age], (err, result) => {
    if (err) return res.status(500).json(err);
    res.status(201).json({ id: result.insertId, name, email, age });
  });
};

exports.getUsers = (req, res) => {
  connection.query('SELECT * FROM users', (err, results) => {
    if (err) return res.status(500).json(err);
    res.status(200).json(results);
  });
};

exports.updateUser = (req, res) => {
  const { id } = req.params;
  const { name, email, age } = req.body;
  const query = 'UPDATE users SET name = ?, email = ?, age = ? WHERE id = ?';
  connection.query(query, [name, email, age, id], (err, result) => {
    if (err) return res.status(500).json(err);
    res.status(200).json({ message: 'User updated successfully' });
  });
};

exports.deleteUser = (req, res) => {
  const { id } = req.params;
  connection.query('DELETE FROM users WHERE id = ?', [id], (err, result) => {
    if (err) return res.status(500).json(err);
    res.status(200).json({ message: 'User deleted successfully' });
  });
};
```

55. **Define Routes in routes.js**:
    - Add the following code to handle routes:

javascript

```
const express = require('express');
const router = express.Router();
const mysqlController = require('./controllers/mysqlController');

router.post('/', mysqlController.createUser);
router.get('/', mysqlController.getUsers);
router.put('/:id', mysqlController.updateUser);
router.delete('/:id', mysqlController.deleteUser);

module.exports = router;
```

56. **Test the CRUD Operations**:
    - Run the application using nodemon:

bash

```
npx nodemon app.js
```

    - Use tools like Postman or cURL to test the CRUD operations by sending POST, GET, PUT, and DELETE requests to http://localhost:3000/api/users.

Part 3: Implementing CRUD Operations with MongoDB

Step 1: Setting Up MongoDB Connection

57. **Configure the Database Connection**:
    - In config.js, add MongoDB configuration:

```
module.exports = {
  mongodb: 'mongodb://localhost:27017/crud_mongodb'
};
```

58. **Connect to MongoDB in app.js**:
    - Update app.js to include MongoDB:

```
const mongoose = require('mongoose');
const config = require('./config');

mongoose.connect(config.mongodb, { useNewUrlParser: true, useUnifiedTopology: true })
```

```
  .then(() => console.log('Connected to MongoDB'))
  .catch(err => console.error('Could not connect to MongoDB...', err));

app.use('/api/users', require('./routes'));
```

Step 2: Implement CRUD Operations

59. **Create a Mongoose Model**:
    - In models/user.js, define the MongoDB schema:

javascript

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

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  age: { type: Number, required: true }
});

module.exports = mongoose.model('User', userSchema);
```

60. **Create a Controller for CRUD Logic**:
    - In controllers/mongodbController.js, add the following CRUD operations:

javascript

```
const User = require('../models/user');

exports.createUser = async (req, res) => {
  try {
    const user = new User(req.body);
    await user.save();
    res.status(201).json(user);
  } catch (err) {
    res.status(500).json(err);
  }
```

```
};

exports.getUsers = async (req, res) => {
 try {
  const users = await User.find();
  res.status(200).json(users);
 } catch (err) {
  res.status(500).json(err);
 }
};

exports.updateUser = async (req, res) => {
 try {
  const user = await User.findByIdAndUpdate(req.params.id, req.body, { new: true });
  res.status(200).json(user);
 } catch (err) {
  res.status(500).json(err);
 }
};

exports.deleteUser = async (req, res) => {
 try {
  await User.findByIdAndDelete(req.params.id);
  res.status(200).json({ message: 'User deleted successfully' });
 } catch (err) {
  res.status(500).json(err);
 }
};
```

61. **Define Routes in routes.js**:
    - Replace the current route definitions with:

javascript

```
const express = require('express');
const router = express.Router();
const mongodbController = require('./controllers/mongodbController');
```

```
router.post('/', mongodbController.createUser);
router.get('/', mongodbController.getUsers);
router.put('/:id', mongodbController.updateUser);
router.delete('/:id', mongodbController.deleteUser);

module.exports = router;
```

62. **Test the CRUD Operations**:
    - Ensure that MongoDB is running.
    - Restart the application with nodemon.
    - Use Postman or cURL to send CRUD requests to http://localhost:3000/api/users.

Week 4: Authentication and Authorization

Slides: authentication_authorization_lab.pptx

Here's a complete example of a Node.js application that includes user registration, login, and securing API routes using JWTs. This example is structured to match the lab activities, along with detailed explanations.

1. Setup

First, ensure you have the required packages installed:

bash

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

2. Directory Structure

bash

```
/auth-lab
  ├── server.js
  ├── models
  │   └── User.js
  ├── middleware
  │   └── auth.js
  ├── routes
  │   └── auth.js
  ├── .env
  └── config
      └── db.js
```

3. Environment Configuration (.env)

Create a .env file for storing environment variables:

env

```
PORT=5000
MONGO_URI=mongodb://localhost:27017/auth_lab
JWT_SECRET=your_jwt_secret
```

4. Database Connection (config/db.js)

Set up the connection to MongoDB using Mongoose:

javascript

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

const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGO_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log('MongoDB connected');
  } catch (err) {
    console.error(err.message);
    process.exit(1);
  }
};

module.exports = connectDB;
```

5. User Model (models/User.js)

Define the User schema and model using Mongoose:

javascript

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

const UserSchema = new mongoose.Schema({
  username: {
    type: String,
    required: true,
    unique: true,
  },
  password: {
```

```
    type: String,
    required: true,
  },
  role: {
    type: String,
    required: true,
    enum: ['user', 'admin'],
  },
});

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

6. Authentication Middleware (middleware/auth.js)

Create middleware to protect routes and handle authorization:

javascript

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

const authMiddleware = (req, res, next) => {
  const token = req.header('Authorization')?.replace('Bearer ', '');
  if (!token) {
    return res.status(401).json({ msg: 'No token, authorization denied' });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (err) {
    res.status(401).json({ msg: 'Token is not valid' });
  }
};

const roleMiddleware = (role) => (req, res, next) => {
  if (req.user.role !== role) {
```

```
    return res.status(403).json({ msg: 'Access denied' });
  }
  next();
};

module.exports = { authMiddleware, roleMiddleware };
```

7. Authentication Routes (routes/auth.js)

Implement routes for user registration, login, and a protected route:

javascript

```
const express = require('express');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const User = require('../models/User');
const { authMiddleware, roleMiddleware } = require('../middleware/auth');

const router = express.Router();

// Register a new user
router.post('/register', async (req, res) => {
  const { username, password, role } = req.body;

  try {
    let user = await User.findOne({ username });
    if (user) {
      return res.status(400).json({ msg: 'User already exists' });
    }

    user = new User({
      username,
      password: await bcrypt.hash(password, 10),
      role,
    });
```

```
    await user.save();
    res.status(201).json({ msg: 'User registered successfully' });
  } catch (err) {
    res.status(500).json({ msg: 'Server error' });
  }
});

// Login a user
router.post('/login', async (req, res) => {
  const { username, password } = req.body;

  try {
    const user = await User.findOne({ username });
    if (!user) {
      return res.status(400).json({ msg: 'Invalid credentials' });
    }

    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) {
      return res.status(400).json({ msg: 'Invalid credentials' });
    }

    const token = jwt.sign(
      { userId: user._id, role: user.role },
      process.env.JWT_SECRET,
      { expiresIn: '1h' }
    );

    res.json({ token });
  } catch (err) {
    res.status(500).json({ msg: 'Server error' });
  }
});

// Protected route example (Admin only)
router.get('/admin', authMiddleware, roleMiddleware('admin'), (req, res) => {
  res.json({ msg: 'Welcome to the admin area' });
});
```

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

8. Main Server File (server.js)

Set up the server, connect to the database, and include routes:

javascript

```
const express = require('express');
const connectDB = require('./config/db');
require('dotenv').config();

const app = express();

// Connect to database
connectDB();

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

// Routes
app.use('/api/auth', require('./routes/auth'));

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

9. Running the Application

Start the application:

bash

```
node server.js
```

The server will run on http://localhost:5000. You can now test the endpoints using tools like Postman.

Week 5: Introduction to Middleware

Slides: Introduction_to_Middleware.pptx

Github: https://github.com/mithunkumarsr/LearnNodeWithMithun/blob/main/express-middleware-routing.zip

1. Overview

Middleware is a fundamental concept in backend development, especially in frameworks like Express.js. It refers to functions that have access to the request object (req), the response object (res), and the next function in the application's request-response cycle.

These functions can execute any code, make changes to the request and response objects, end the request-response cycle, or call the next middleware in the stack.

Key Concepts:

- **Request-Response Cycle:** The process through which a server receives a client request, processes it, and sends back a response.
- **Middleware Stack:** The series of middleware functions that are executed sequentially for a request.

2. Types of Middleware

Application-Level Middleware:

- Defined at the application level using app.use() or app.METHOD().
- Applies to every incoming request or specific routes.

Example:

javascript

```
app.use((req, res, next) => {
  console.log('Time:', Date.now());
  next();
});
```

Router-Level Middleware:

- Attached to an instance of express.Router() and applies only to routes handled by that router.

Example:

javascript

```
const router = express.Router();
router.use((req, res, next) => {
  console.log('Router Time:', Date.now());
```

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

Error-Handling Middleware:

- Used to handle errors that occur during the request-response cycle.
- Should be defined after all other app.use() and routes calls.

Example:

javascript

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

Built-In Middleware:

- Provided by Express, such as express.json() to parse JSON bodies or express.static() to serve static files.

Example:

javascript

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

Third-Party Middleware:

- Middleware provided by the community, such as morgan for logging, body-parser for parsing request bodies, etc.

Example:

javascript

```
const morgan = require('morgan');
app.use(morgan('tiny'));
```

3. Common Middleware Functions

- **Logging:** Middleware that logs details about each request.
- **Authentication:** Middleware that verifies the identity of a user.
- **Authorization:** Middleware that checks whether the authenticated user has permission to access a resource.
- **Request Parsing:** Middleware that parses incoming requests, like JSON or URL-encoded data.
- **Error Handling:** Middleware that catches and handles errors in the application.

4. Example: Middleware in Express

javascript

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

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

// Custom Logging Middleware
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});

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

// Error-Handling Middleware
```

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});

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

5. Demo

1. Setting Up the Project

First, create a new Node.js project and install the necessary dependencies.

bash

```
mkdir express-middleware-routing
cd express-middleware-routing
npm init -y
npm install express morgan
```

2. Basic Express Setup

Create an index.js file and set up a basic Express server.

javascript

```
// index.js

const express = require('express');
const app = express();
const port = 3000;

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

```
// Sample route
app.get('/', (req, res) => {
  res.send('Welcome to the Middleware and Routing Example!');
});

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

3. Creating Custom Middleware

a. Logging Middleware

javascript

```
// loggingMiddleware.js

const fs = require('fs');
const path = require('path');

const logStream = fs.createWriteStream(path.join(__dirname, 'access.log'), { flags: 'a' });

const loggingMiddleware = (req, res, next) => {
  const log = ` ${new Date().toISOString()} - ${req.method} ${req.originalUrl}\n` ;
  console.log(log);
  logStream.write(log);
  next();
};

module.exports = loggingMiddleware;
```

b. Error Handling Middleware

javascript

```
// errorHandlingMiddleware.js

const errorHandlingMiddleware = (err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ message: 'Something went wrong!', error: err.message });
};

module.exports = errorHandlingMiddleware;
```

4. Integrating Middleware into Express

Update index.js to include the custom middleware.

javascript

```
// index.js

const express = require('express');
const app = express();
const loggingMiddleware = require('./loggingMiddleware');
const errorHandlingMiddleware = require('./errorHandlingMiddleware');
const port = 3000;

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

// Use the custom logging middleware
app.use(loggingMiddleware);

// Sample route
app.get('/', (req, res) => {
  res.send('Welcome to the Middleware and Routing Example!');
});

// Route that triggers an error
app.get('/error', (req, res) => {
  throw new Error('This is a forced error!');
```

```
});

// Use the custom error-handling middleware (should be after all routes)
app.use(errorHandlingMiddleware);

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

5. Advanced Routing Techniques

a. Nested Routes Example

Let's create nested routes for users and posts.

javascript

```
// routes/users.js

const express = require('express');
const router = express.Router();

// Nested route for posts under a specific user
router.get('/:userId/posts/:postId', (req, res) => {
  res.send(` User ID: ${req.params.userId}, Post ID: ${req.params.postId}` );
});

// Route to get all users
router.get('/', (req, res) => {
  res.send('List of all users');
});

module.exports = router;
```

b. Dynamic Routes Example

In the users routes, we are already using dynamic routes with :userId and :postId.

6. Organizing and Using Routes

Update index.js to use the organized routes.

javascript

```
// index.js

const express = require('express');
const loggingMiddleware = require('./loggingMiddleware');
const errorHandlingMiddleware = require('./errorHandlingMiddleware');
const usersRouter = require('./routes/users');
const app = express();
const port = 3000;

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

// Use the custom logging middleware
app.use(loggingMiddleware);

// Root route
app.get('/', (req, res) => {
  res.send('Welcome to the Middleware and Routing Example!');
});

// Use the users router for /users routes
app.use('/users', usersRouter);

// Route that triggers an error
app.get('/error', (req, res) => {
  throw new Error('This is a forced error!');
});
```

```
// Use the custom error-handling middleware (should be after all routes)
app.use(errorHandlingMiddleware);

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

7. Running and Testing the Application

63. Start the server:

bash

```
node index.js
```

64. Test the logging middleware by making a request to the root route (http://localhost:3000/). Check the console and the access.log file to see the logged request.
65. Test the error handling by visiting the /error route (http://localhost:3000/error).
66. Test the nested and dynamic routes by visiting a nested route like http://localhost:3000/users/1/posts/10.

8. Sample Project Structure

Here's what your project directory might look like:

lua

```
express-middleware-routing/
│
├── access.log
├── errorHandlingMiddleware.js
├── index.js
├── loggingMiddleware.js
├── package.json
└── routes/
```

└── users.js

Summary

- **Logging Middleware:** Logs each request with details like the method, URL, and timestamp to a file and console.
- **Error-Handling Middleware:** Catches errors that occur during request processing and sends a formatted error response.
- **Nested and Dynamic Routing:** Demonstrates how to set up routes that include dynamic parameters and are nested within each other.

Exercise 1: Create a Custom Middleware for Logging

67. Create an Express application.
68. Implement a custom middleware function that logs the method and URL of each incoming request to the console.
69. Test the middleware by making requests to different routes.

Exercise 2: Implement an Error-Handling Middleware

70. Modify your existing Express application.
71. Add a route that deliberately throws an error.
72. Implement an error-handling middleware that catches this error and sends a user-friendly message to the client.
73. Test by accessing the route that throws an error.

Exercise 3: Use Third-Party Middleware

74. Install the morgan middleware using npm install morgan.
75. Integrate morgan into your Express application to log requests in the 'tiny' format.
76. Compare the output of morgan with your custom logging middleware.

Exercise 4: Apply Router-Level Middleware

77. Create a new router in your Express application.
78. Attach middleware to the router that logs a message every time a request is made to any route handled by that router.

79. Define a few routes under the router and verify that the middleware is executed.

Exercise 5: Chain Multiple Middleware Functions

80. Implement multiple middleware functions that each perform a simple task (e.g., add a property to the req object).
81. Chain these middleware functions together in a route.
82. Verify that each middleware function is executed in order.

6. Conclusion

Middleware is a powerful and flexible mechanism for handling different aspects of the request-response cycle in an Express application. Understanding how to create, apply, and manage middleware is crucial for building scalable and maintainable backend systems.

Week 6: Data Validation and Error Handling

Lab Objective:

- Learn how to implement data validation for API requests.
- Set up error handling strategies for both client and server errors.
- Send appropriate HTTP status codes and meaningful error messages.
- Implement this in a sample Node.js application using Express, Joi, and error handling middleware.

Step 1: Setting Up a Node.js Application

83. Initialize the Project:

bash

```
mkdir data-validation-lab
cd data-validation-lab
npm init -y
```

This initializes a new Node.js project.

84. **Install Required Packages:** Install the necessary packages for this lab.

bash

```
npm install express joi
```

- express: Web framework for building the API.
- joi: Validation library to ensure incoming request data is properly formatted.

Step 2: Setting Up Basic Express Server

85. Create app.js to start the server:

javascript

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

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

app.get('/', (req, res) => {
  res.send('API is running...');
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
```

```
  console.log(`Server is running on port ${PORT}`);
});
```

- This sets up a basic Express server that listens on port 3000.

Step 3: Implementing Input Validation with Joi

86. **Define Validation Schema for User Input:** We'll create an API that handles user data. The user input will be validated using Joi.

Validation Schema:

javascript

```
const Joi = require('joi');

const userSchema = Joi.object({
  name: Joi.string().min(3).required(),
  email: Joi.string().email().required(),
  age: Joi.number().integer().min(18).max(65).required()
});
```

- name: A string that must be at least 3 characters long.
- email: A valid email format.
- age: A number between 18 and 65.

87. **Create API Route to Handle User Data:**

javascript

```
app.post('/user', (req, res) => {
  const { error } = userSchema.validate(req.body);
  if (error) {
    return res.status(400).send(error.details[0].message);
  }
  res.send('User created successfully');
});
```

- This route accepts POST requests to /user and validates the incoming data against the userSchema.
- If validation fails, a 400 Bad Request status code is sent with the error message.

Step 4: Structuring and Handling API Errors

88. **Creating a Custom Error Class:** To handle errors more effectively, create a custom AppError class.

javascript

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

    Error.captureStackTrace(this, this.constructor);
  }
}
```

- AppError allows us to define custom error messages and HTTP status codes.

89. **Centralized Error Handling Middleware:** To ensure all errors are handled in one place, we create a centralized error handler.

javascript

```
app.use((err, req, res, next) => {
  if (err.isOperational) {
    return res.status(err.statusCode).json({
      status: 'fail',
      message: err.message
    });
  }
  res.status(500).json({
    status: 'error',
    message: 'Internal Server Error'
```

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

- This middleware catches all errors, both operational (e.g., validation errors) and internal server errors.

Step 5: Sending Proper HTTP Status Codes

90. **Handling Client Errors (4xx):** Let's create an example route that returns a 404 error if a user is not found.

Example Route:

javascript

```
app.get('/user/:id', (req, res, next) => {
  const user = getUserById(req.params.id); // A function that retrieves user by ID.
  if (!user) {
    return next(new AppError('User not found', 404));
  }
  res.status(200).json(user);
});
```

- If a user is not found, an AppError is thrown with a 404 status code.

91. **Handling Server Errors (5xx):** If something goes wrong in your code, return a 500 Internal Server Error.

Example:

javascript

```
app.get('/cause-error', (req, res, next) => {
  try {
    // Some buggy code that throws an error
    throw new Error('Something went wrong!');
  } catch (err) {
    next(new AppError('Internal Server Error', 500));
  }
```

});

- The error handler catches this and sends a 500 response.

Step 6: Sample Project – Integrating Validation and Error Handling

92. API for Creating Users:

- Validate the user input for name, email, and age using the userSchema.
- Send proper error messages and status codes when validation fails.

93. API for Retrieving Users:

- Return 404 if a user is not found.
- Return 500 for internal server errors.

Sample app.js Full Code:

javascript

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

app.use(express.json());

// Joi validation schema
const userSchema = Joi.object({
  name: Joi.string().min(3).required(),
  email: Joi.string().email().required(),
  age: Joi.number().integer().min(18).max(65).required()
});

// Custom Error Class
class AppError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.statusCode = statusCode;
    this.isOperational = true;
  }
}
```

```
// POST route to create a user
app.post('/user', (req, res, next) => {
  const { error } = userSchema.validate(req.body);
  if (error) {
    return next(new AppError(error.details[0].message, 400));
  }
  res.send('User created successfully');
});

// Get user by ID
app.get('/user/:id', (req, res, next) => {
  const user = getUserById(req.params.id);
  if (!user) {
    return next(new AppError('User not found', 404));
  }
  res.status(200).json(user);
});

// Centralized Error Handling Middleware
app.use((err, req, res, next) => {
  if (err.isOperational) {
    return res.status(err.statusCode).json({
      status: 'fail',
      message: err.message
    });
  }
  res.status(500).json({
    status: 'error',
    message: 'Internal Server Error'
  });
});

// Starting the server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
```

});

Lab Task Instructions:

94. **Create the API:**
    - Set up the project structure as shown above.
    - Implement validation and error handling as described.
95. **Test Various Scenarios:**
    - Send valid and invalid user data to the /user route and observe the results.
    - Try accessing non-existing resources and confirm the correct error messages and status codes are returned.

Conclusion:

By the end of this lab, you should have a good understanding of how to:

- Implement input validation using libraries like Joi.
- Structure error handling for client and server errors.
- Send appropriate HTTP status codes and descriptive error messages in your API responses.

Testing Using Postman

1. Test POST /user Route (Create User with Validation)

- **Purpose:** Validate the creation of a user and handle invalid inputs.
- **Valid Input:** Send a POST request to http://localhost:3000/user with the following JSON body:

json

```
{
  "name": "John Doe",
  "email": "john.doe@example.com",
  "age": 30
}
```

Expected Response:

sql

```
User created successfully
```

Status: 200 OK

- **Invalid Input (Missing Fields):** Send the following JSON with the name field missing:

json

```
{
  "email": "john.doe@example.com",
  "age": 30
}
```

Expected Response:

json

```
{
  "status": "fail",
  "message": "\"name\" is required"
}
```

Status: 400 Bad Request

- **Invalid Input (Invalid Email):** Send the following JSON with an invalid email format:

json

```
{
  "name": "John Doe",
  "email": "john.doe",
  "age": 30
}
```

Expected Response:

json

```
{
  "status": "fail",
  "message": "\"email\" must be a valid email"
}
```

Status: 400 Bad Request

2. Test GET /user/:id Route (Fetching a User)

- **Purpose:** Handle cases where the requested user does not exist.
- **Valid Input:** Send a GET request to http://localhost:3000/user/1.

Assuming the user with ID 1 exists, you should see the user data:

json

```
{
  "id": "1",
  "name": "John Doe"
}
```

Status: 200 OK

- **Invalid Input (User Not Found):** Send a GET request to http://localhost:3000/user/99.

Expected Response:

json

```
{
  "status": "fail",
  "message": "User not found"
}
```

Status: 404 Not Found

3. Test Error Handling (Internal Server Error)

- **Purpose:** Test server error handling by sending a request to the /cause-error route, which simulates a bug in the code.
- **Test:** Send a GET request to http://localhost:3000/cause-error.

Expected Response:

json

```
{
  "status": "error",
  "message": "Internal Server Error"
}
```

Status: 500 Internal Server Error

Step 3: Testing Using curl (Optional)

If you prefer using curl (or you are working in a terminal), you can test the API routes using the following commands:

1. Test POST /user (Valid Input):

bash

curl -X POST http://localhost:3000/user -H "Content-Type: application/json" -d '{"name": "John Doe", "email": "john.doe@example.com", "age": 30}'

Expected Output:

sql

User created successfully

2. Test POST /user (Invalid Input - Missing Name):

bash

curl -X POST http://localhost:3000/user -H "Content-Type: application/json" -d '{"email": "john.doe@example.com", "age": 30}'

Expected Output:

json

```
{
  "status": "fail",
  "message": "\"name\" is required"
}
```

3. Test GET /user/:id (User Not Found):

bash

curl -X GET http://localhost:3000/user/99

Expected Output:

json

```
{
  "status": "fail",
  "message": "User not found"
}
```

4. Test GET /cause-error (Simulate Server Error):

bash

curl -X GET http://localhost:3000/cause-error

Expected Output:

json

```
{
  "status": "error",
  "message": "Internal Server Error"
}
```

Step 4: Writing Automated Tests with Mocha and Chai

You can also automate the testing using testing frameworks like Mocha and Chai. Here's a quick overview of how to set that up:

96. Install Mocha and Chai:

bash

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

97. Create a test.js file:

javascript

```
const request = require('supertest');
const app = require('./app'); // Adjust path as necessary
const expect = require('chai').expect;

describe('POST /user', () => {
  it('should create a user with valid data', async () => {
    const response = await request(app)
      .post('/user')
      .send({
        name: 'John Doe',
        email: 'john.doe@example.com',
        age: 30
      });
    expect(response.status).to.equal(200);
    expect(response.text).to.equal('User created successfully');
  });

  it('should return 400 for invalid data', async () => {
    const response = await request(app)
      .post('/user')
      .send({
        email: 'john.doe@example.com',
        age: 30
      });
    expect(response.status).to.equal(400);
    expect(response.body.message).to.equal('"name" is required');
  });
```

});

98. **Run the Tests:** Run the following command to execute the tests:

bash

npx mocha

Exercise 1: Basic Validation

- **Objective:** Implement basic validation for creating a new user.
- **Task:**
    a. Create a new API endpoint /register that accepts POST requests.
    b. Validate the following fields:
        - username: Must be a string of at least 5 characters.
        - password: Must be at least 8 characters long.
        - email: Must be a valid email address.
    c. If any field fails validation, return a 400 Bad Request status with a clear error message.

Sample Input:

json

```
{
  "username": "user",
  "password": "12345",
  "email": "invalid-email"
}
```

Expected Output:

json

```
{
  "error": "Invalid email format"
}
```

Exercise 2: Custom Validation Messages

- **Objective:** Customize validation error messages.
- **Task:**
    d. Modify the existing /register route to return custom error messages if validation fails.
        - For username: "Username must be at least 5 characters long."
        - For password: "Password must be at least 8 characters long."
        - For email: "Please provide a valid email address."
    e. Test with different invalid inputs and verify the custom error messages.

Hint: Use the .messages() method in Joi to customize error messages.

Exercise 3: Handling Nested Objects

- **Objective:** Validate nested objects in requests.
- **Task:**
    f. Create an API route /createProfile that accepts POST requests.
    g. Validate the following:
        - name: Must be a string.
        - address: A nested object with:
            - street: Required, string.
            - city: Required, string.
            - postalCode: Must be a valid number and at least 6 digits.
    h. Ensure that both the parent object (address) and nested fields are properly validated.

Sample Input:

json

```
{
  "name": "John Doe",
  "address": {
    "street": "123 Main St",
    "city": "Mumbai",
    "postalCode": "ABCDE"
  }
}
```

Expected Output:

json

```
{
  "error": "postalCode must be a valid number"
}
```

Exercise 4: Custom Error Handling

- **Objective:** Implement centralized error handling for validation and other types of errors.
- **Task:**
    i. Modify your app to use centralized error handling middleware.
    j. Create an endpoint /product that accepts POST requests. Validate the following fields:
        - productName: Required, string.
        - price: Required, positive number.
        - quantity: Must be an integer between 1 and 100.
    k. Throw an error if the product already exists (simulate this scenario) and return a 409 Conflict status code.
    l. Use the centralized error handler to catch both validation errors and the custom Conflict error.

Hint: Use the custom AppError class to throw errors and send proper status codes.

Exercise 5: Handling Async Errors

- **Objective:** Implement error handling for asynchronous code.
- **Task:**
    m. Create a new route /async-user/:id that fetches user data asynchronously from a mock database function.
    n. If the user ID is not found, throw an error and return a 404 Not Found status.
    o. Ensure that errors in asynchronous code are properly caught and handled by your error middleware.

Mock Function:

javascript

```
const getUserById = async (id) => {
 // Simulating a database call that rejects if the user is not found
 return new Promise((resolve, reject) => {
  if (id === '1') {
   resolve({ id: '1', name: 'John Doe' });
  } else {
   reject(new Error('User not found'));
  }
 });
};
```

Exercise 6: Validation with Query Parameters

- **Objective:** Validate query parameters in API requests.
- **Task:**
    p. Create a GET route /search that accepts query parameters q (search query) and limit (number of results).
    q. Validate:
        - q: Required, string, must be at least 3 characters.
        - limit: Optional, number, between 1 and 50.
    r. If validation fails, return a 400 Bad Request with the appropriate message.

Sample Request:

bash

GET /search?q=js&limit=100

Expected Output:

json

```
{
  "error": "Limit must be between 1 and 50"
}
```

Exercise 7: Handling Validation for Array Data

- **Objective:** Validate arrays in API requests.
- **Task:**
    s. Create an endpoint /bulkUpload that accepts POST requests with an array of user objects.
    t. Each user object must have the following:
        - name: Required, string.
        - email: Required, valid email.
        - age: Required, integer between 18 and 65.
    u. Validate each user in the array and return a 400 error if any of the users in the array fails validation.

Sample Input:

json

```
{
  "users": [
    {
      "name": "John Doe",
      "email": "john@example.com",
      "age": 30
    },
```

```
  {
    "name": "Invalid User",
    "email": "invalid-email",
    "age": 17
  }
 ]
}
```

Expected Output:

json

```
{
  "error": "User age must be between 18 and 65"
}
```

Exercise 8: Validating File Uploads

- **Objective:** Validate uploaded files.
- **Task:**
    v. Create an API endpoint /upload that accepts file uploads.
    w. Validate:
        - The file type must be an image (jpg, jpeg, png).
        - The file size must be less than 1 MB.
    x. If the validation fails, return a 400 Bad Request.

Hint: You can use a package like multer for handling file uploads in Express.

Bonus Challenge: Creating Reusable Validation Middleware

- **Objective:** Refactor the validation logic into reusable middleware.
- **Task:**
    y. Create a validate middleware that takes a Joi schema and validates the request body.

z. Apply the middleware to different routes to validate various data.
aa. Ensure that all routes return proper error messages if validation fails.

Deliverables:

99. Complete the API routes for each exercise.
100. Test the API with different inputs and ensure validation works correctly.
101. Submit the code along with test cases for the validation and error handling.

Week 7: Working with External APIs

Slides: Week_7_External_API_Integration.pptx

Here's how you can send an SMS using the Twilio API in Node.js. Before you start, you'll need a Twilio account and a Twilio phone number to send messages. You can sign up at Twilio's website and get free credits for testing.

Step-by-Step Guide to Send SMS via Twilio API

1. Install Twilio SDK

First, install the Twilio Node.js library using npm.

bash

npm install twilio

2. Set Up Environment Variables

You'll need the following credentials from your Twilio account:

- ACCOUNT_SID: Your Twilio account SID.
- AUTH_TOKEN: Your Twilio auth token.
- TWILIO_PHONE_NUMBER: The Twilio phone number you purchased.

You can set these up as environment variables to keep your credentials secure.

In your .env file:

plaintext

```
TWILIO_ACCOUNT_SID=your_account_sid
TWILIO_AUTH_TOKEN=your_auth_token
TWILIO_PHONE_NUMBER=your_twilio_phone_number
```

3. Send an SMS with Node.js

Create a file (e.g., send_sms.js), and add the following code:

javascript

```
require('dotenv').config();
const twilio = require('twilio');

// Load environment variables
const accountSid = process.env.TWILIO_ACCOUNT_SID;
const authToken = process.env.TWILIO_AUTH_TOKEN;
const client = twilio(accountSid, authToken);

// Function to send SMS
const sendSMS = (toPhoneNumber, message) => {
  client.messages
    .create({
      body: message,            // SMS content
      from: process.env.TWILIO_PHONE_NUMBER, // Your Twilio number
      to: toPhoneNumber,        // Recipient's phone number
    })
    .then((message) => console.log(`Message sent: ${message.sid}`))
    .catch((error) => console.error(`Failed to send SMS: ${error}`));
```

```
};

// Example usage
const recipientPhoneNumber = '+1234567890';  // Replace with actual recipient number
const message = 'Hello! This is a test message from Twilio.';
sendSMS(recipientPhoneNumber, message);
```

4. Run the Code

To send the SMS, run the script using Node.js:

bash

```
node send_sms.js
```

5. Explanation:

- **twilio.messages.create()**: This method creates and sends an SMS. It takes an object containing the body (message), from (Twilio phone number), and to (recipient phone number).
- **Environment Variables**: You can securely store your credentials in a .env file and access them using process.env.

6. Handling Errors:

The code includes error handling using the .catch() block to catch and log any errors that occur during the API call.

Additional Notes:

- Ensure that the recipient's phone number is in the correct international format (e.g., +1 for the US).
- In a real-world app, you may want to handle different types of errors, like invalid phone numbers, network errors, or authentication failures.

Week 8: File Uploads and Management - Backend Development Lab

Objective

The objective of this lab is to implement file upload functionality in backend applications. We will focus on handling file uploads using local storage and Google Cloud Storage (GCS).

Lab Outline

1. Introduction

This section covers the importance of file uploads in backend systems, the difference between local and cloud storage, and an introduction to Google Cloud Storage (GCS).

2. Lab Setup

Set up the project environment and install necessary dependencies for file uploads. Configure Google Cloud SDK to handle uploads to Google Cloud Storage.

Commands:

npm install express multer @google-cloud/storage

Set up GCP credentials for Google Cloud Storage.

3. File Upload: Local Storage

This section involves using Multer to upload files and store them locally on the server.

Code Example:

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

```
const app = express();

const storage = multer.diskStorage({
    destination: function (req, file, cb) {
        cb(null, 'uploads/');
    },
    filename: function (req, file, cb) {
        const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1E9);
        cb(null, file.fieldname + '-' + uniqueSuffix + path.extname(file.originalname));
    }
});

const upload = multer({ storage: storage });

app.post('/upload', upload.single('file'), (req, res) => {
    res.send('File uploaded locally: ' + req.file.filename);
});

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

4. Integrating Google Cloud Storage (40 minutes)

Modify the upload functionality to store files in Google Cloud Storage (GCS) instead of local storage.

Code Example:

```
const { Storage } = require('@google-cloud/storage');
const multer = require('multer');
const path = require('path');
const fs = require('fs');
const express = require('express');

const app = express();
```

```
const storage = new Storage();
const bucketName = 'your-bucket-name';

const upload = multer({ dest: 'uploads/' });

app.post('/upload', upload.single('file'), async (req, res) => {
  const localFilePath = req.file.path;
  const gcsFilename = Date.now() + '-' + path.basename(req.file.originalname);
  try {
    await storage.bucket(bucketName).upload(localFilePath, {
      destination: gcsFilename,
      public: true,
      metadata: { cacheControl: 'public, max-age=31536000' },
    });
    fs.unlinkSync(localFilePath);
    res.send(` File uploaded to Google Cloud Storage as ${gcsFilename}` );
  } catch (err) {
    console.error(err);
    res.status(500).send('Error uploading file');
  }
});

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

5. File Download and Management

Implement file download functionality to allow users to retrieve files from GCS.

Code Example:

```
const getFileFromGCS = async (filename) => {
  const options = { destination: path.join(__dirname, 'downloads', filename) };
  await storage.bucket(bucketName).file(filename).download(options);
  console.log(` File downloaded to ${options.destination}` );
};
```

```
app.get('/download/:filename', async (req, res) => {
    const filename = req.params.filename;
    try {
        await getFileFromGCS(filename);
        res.download(path.join(__dirname, 'downloads', filename));
    } catch (err) {
        console.error(err);
        res.status(500).send('Error downloading file');
    }
});
```

6. Error Handling and File Validation

Add file type validation and size limits to the upload functionality.

Code Example:

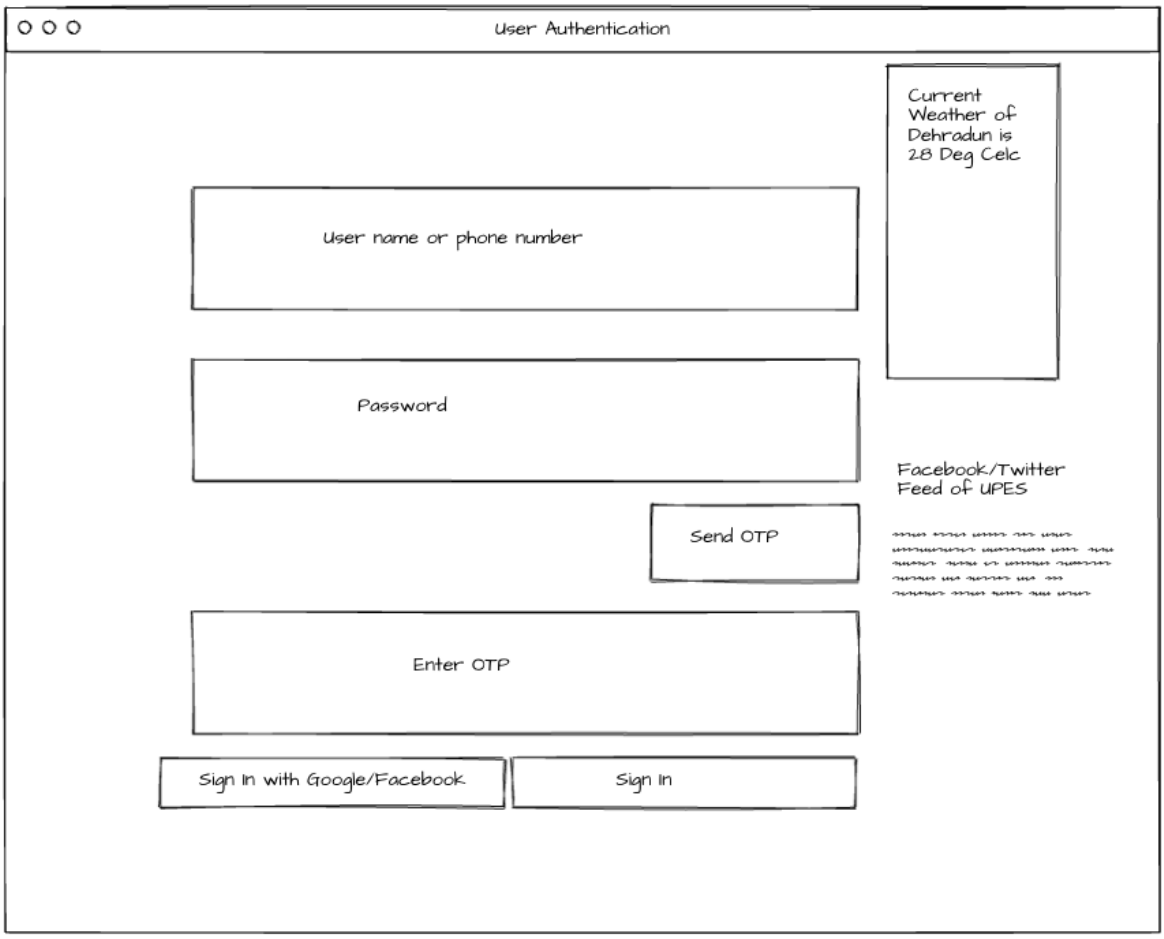
```
const upload = multer({
    dest: 'uploads/',
    limits: { fileSize: 2 * 1024 * 1024 },
    fileFilter: (req, file, cb) => {
        const filetypes = /jpeg|jpg|png/;
        const mimetype = filetypes.test(file.mimetype);
        const extname = filetypes.test(path.extname(file.originalname).toLowerCase());
        if (mimetype && extname) return cb(null, true);
        cb(new Error('Only images are allowed!'));
    }
});

app.post('/upload', upload.single('file'), (req, res) => {
    if (req.fileValidationError) return res.status(400).send(req.fileValidationError);
    res.send('File uploaded successfully');
});
```

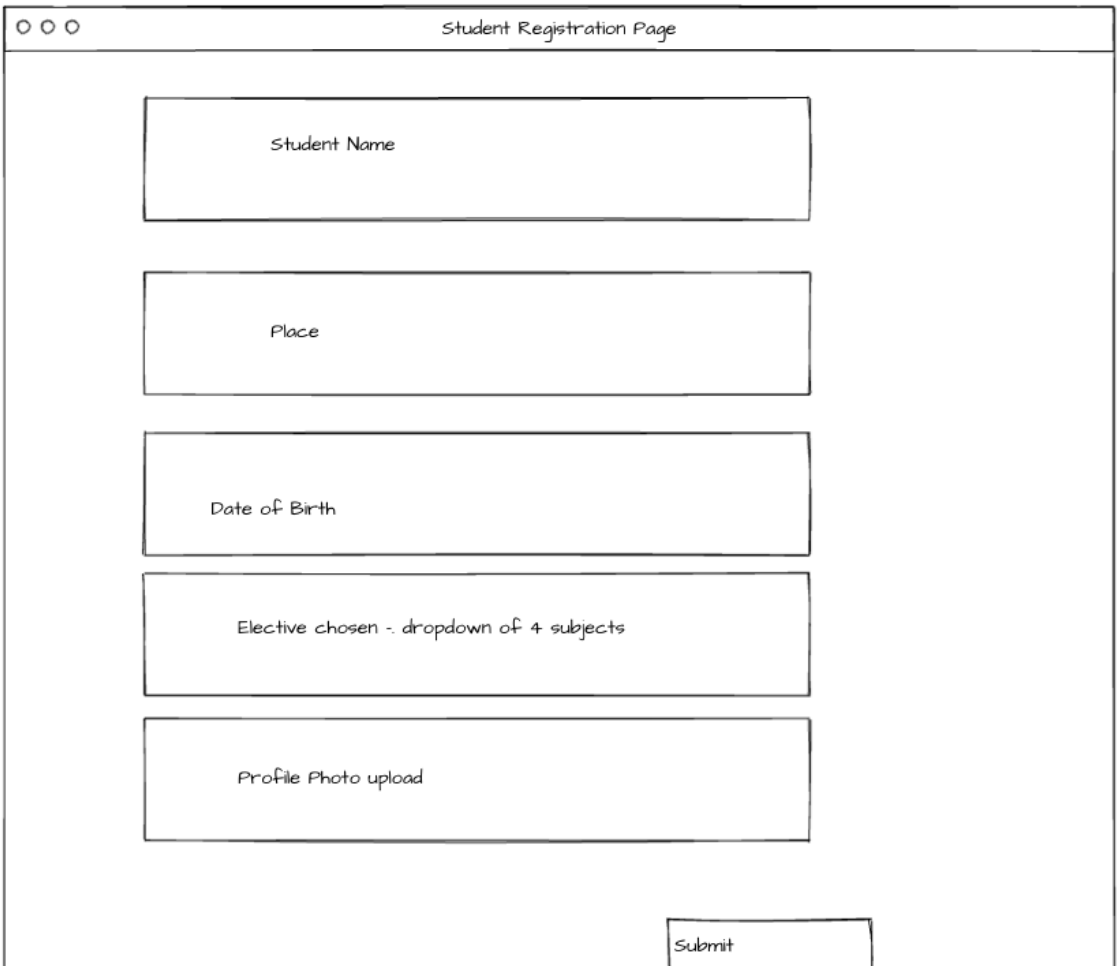
Lab Exercise:

Have a profile photo in the student registration page and have it uploaded locally (in the same file format)

Provide an API to download the picture for a particular user

Wireframe of the expectations until this lab is given below.


User Authentication
Current Weather of Dehradun is 28 Deg Celc
User name or phone number
Password
Facebook/Twitter Feed of UPES
Send OTP
Enter OTP
Sign In with Google/Facebook
Sign In


Student Registration Page
Student Name
Place
Date of Birth
Elective chosen -. dropdown of 4 subjects
Profile Photo upload
Submit

Week 9: Realtime communication with Websocket

https://github.com/mithunkumarsr/LearnNodeWithMithun/blob/main/websocket-lab.zip

Learning Objectives:

- Understand the concept and importance of WebSockets for real-time communication.
- Set up a WebSocket server in a Node.js application.
- Establish a connection between a client and the server using WebSocket.
- Implement real-time communication features like broadcasting messages and notifications.
- Explore handling WebSocket events such as connection, message, and disconnection.

Server-side WebSocket Code (Node.js)

```
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 8080 });

// Store clients by room
const rooms = {};

// Handle WebSocket connection
wss.on('connection', (ws) => {
console.log('New client connected');
// Default room assignment (can be changed later)
ws.room = null;

// Handle incoming messages
ws.on('message', (message) => {
const parsedMessage = JSON.parse(message);
const { type, room, text } = parsedMessage;

if (type === 'join') {
// When a client joins a room
ws.room = room;
if (!rooms[room]) {
rooms[room] = new Set(); // Create the room if it doesn't exist
}
rooms[room].add(ws);
console.log(`Client joined room: ${room}`);
} else if (type === 'message' && ws.room) {
```

```
// When a client sends a message to the room
const currentRoom = ws.room;
console.log(`Message from room ${currentRoom}: ${text}`);

// Broadcast the message only to clients in the same room
rooms[currentRoom].forEach((client) => {
if (client.readyState === WebSocket.OPEN && client.room === currentRoom) {
client.send(JSON.stringify({ room: currentRoom, text }));
}
});
}
});

// Handle client disconnect
ws.on('close', () => {
if (ws.room) {
rooms[ws.room].delete(ws);
if (rooms[ws.room].size === 0) {
delete rooms[ws.room]; // Remove the room if empty
}
}
console.log('Client disconnected');
});
});

console.log('WebSocket server is running on ws://localhost:8080');
```

Client-side WebSocket Code (HTML + JavaScript)}

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>WebSocket Client with Rooms</title>
</head>
<body>
<h1>WebSocket Client with Rooms</h1>

<div>
<label for="roomInput">Enter Room ID:</label>
<input type="text" id="roomInput" placeholder="Room ID">
<button id="joinRoomButton">Join Room</button>
</div>
<div id="messages"></div>

<input type="text" id="messageInput" placeholder="Enter a message">
<button id="sendButton" disabled>Send</button>
```

```
<script>
const ws = new WebSocket('ws://localhost:8080');
const messagesDiv = document.getElementById('messages');
const messageInput = document.getElementById('messageInput');
const sendButton = document.getElementById('sendButton');
const roomInput = document.getElementById('roomInput');
const joinRoomButton = document.getElementById('joinRoomButton');

let currentRoom = null;

// Join a room
joinRoomButton.onclick = () => {
const room = roomInput.value;
if (room.trim() !== '') {
currentRoom = room;
ws.send(JSON.stringify({ type: 'join', room: currentRoom }));
sendButton.disabled = false;
}
};

// Handle incoming messages
ws.onmessage = (event) => {
const message = JSON.parse(event.data);
const messageElem = document.createElement('div');
messageElem.textContent = `Room ${message.room}: ${message.text}`;
messagesDiv.appendChild(messageElem);
};

// Send a message to the room
sendButton.onclick = () => {
const text = messageInput.value;
if (text.trim() !== '' && currentRoom) {
ws.send(JSON.stringify({ type: 'message', room: currentRoom, text }));
messageInput.value = '';
}
};
</script>
</body>
</html>
```

Exercise 1 Real-time Chat Between Admins and Students

Objective: Build a real-time chat feature between students and the admission admin to address queries.

- **Exercise**:
    - Implement a real-time chat window on both the student registration page and the admin dashboard.
    - When a student submits a question or query, it should appear instantly for the admin.
    - The admin should be able to respond in real-time, and the student should receive the reply immediately.
- **Hint**: Use WebSockets for bi-directional communication and create separate chat rooms for each student.

Exercise 2: Real-time Admin Monitoring of Registration Progress

Objective: Allow the admin to monitor in real-time which students are currently filling out the registration form.

- **Exercise**:
    - Use WebSockets to send updates when a student begins and completes filling out the registration form.
    - The admin should see a live list of students currently in the process of registering.
- **Hint**: Send a message when the student starts typing their information, and display it on the admin's dashboard in real-time.