Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

BUILDING RESTFUL & GRAPHQL APIS (WEEKS 7-8)

PRINCIPLES OF REST API DESIGN

CHAPTER 1: INTRODUCTION TO REST API DESIGN

1.1 What is a REST API?

A **REST API (Representational State Transfer API)** is a standardized way of building web services that allow systems to communicate over HTTP. REST APIs follow a **stateless, client-server model,** making them scalable and easy to use.

- **Why Use REST APIs?**

✓ **Scalable** – Designed for large-scale applications.
✓ **Stateless** – Each request contains all necessary information.
✓ **Platform-independent** – Works across multiple technologies.

- **Key Characteristics of REST APIs:**

Principle	Description
Stateless	Each request is independent and does not rely on session data.

Client-Server Architecture	Frontend and backend are separate, allowing independent scaling.
Uniform Interface	Consistent naming, request methods, and responses.
Cacheable	API responses can be cached to improve performance.
Layered System	Requests can pass through multiple layers (e.g., authentication, security).

CHAPTER 2: DESIGNING RESTFUL ENDPOINTS

2.1 Using Resource-Based URLs

REST APIs use **nouns** for URLs, not verbs. Endpoints should represent **resources**, and actions should be performed using **HTTP methods**.

📌 **Example: Correct & Incorrect RESTful URLs**

Incorrect	Correct	Reason
/getUsers	/users	Avoid verbs in URLs.
/updateUser/1	/users/1 (PUT)	Use HTTP methods for actions.
/deleteUser?id=1	/users/1 (DELETE)	Use **path parameters** for resource identification.

📌 **RESTful Endpoint Example:**

```
GET /users      # Fetch all users

GET /users/1    # Fetch user with ID 1

POST /users     # Create a new user
```

```
PUT /users/1    # Update user with ID 1
DELETE /users/1  # Delete user with ID 1
```

✓ Uses **clear, structured resource paths** for API requests.

2.2 Choosing the Right HTTP Methods

HTTP Method	Purpose	Example
GET	Retrieve data	GET /users
POST	Create a new resource	POST /users
PUT	Update an entire resource	PUT /users/1
PATCH	Update part of a resource	PATCH /users/1
DELETE	Remove a resource	DELETE /users/1

📌 **Example: Using HTTP Methods in Express.js**

```
app.get("/users", (req, res) => { /* Fetch users */ });
app.post("/users", (req, res) => { /* Create user */ });
app.put("/users/:id", (req, res) => { /* Update user */ });
app.delete("/users/:id", (req, res) => { /* Delete user */ });
```

✓ Ensures **consistency in API interactions**.

CHAPTER 3: STRUCTURING API RESPONSES

3.1 Standard Response Format

REST APIs should return **structured JSON responses**.

📌 **Example: Success Response**

```
{
  "status": "success",
  "data": {
    "id": 1,
    "name": "Alice",
    "email": "alice@example.com"
  }
}
```

✓ Uses a **consistent format** for API responses.

📌 **Example: Error Response**

```
{
  "status": "error",
  "message": "User not found",
  "code": 404
}
```

✓ Includes **status, error message, and HTTP code**.

3.2 Using HTTP Status Codes Correctly

Status Code	Meaning	Example Scenario
200 OK	Request successful	Data retrieval (GET)
201 Created	Resource successfully created	New user added (POST)

204 No Content	Request successful but no data	Successful DELETE request
400 Bad Request	Client error	Missing required parameters
401 Unauthorized	Authentication failed	Invalid API token
403 Forbidden	No permission	User lacks authorization
404 Not Found	Resource does not exist	Requesting a non-existent user
500 Internal Server Error	Server-side issue	Unexpected errors

📌 **Example: Sending Proper Status Codes in Express.js**

```
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.status(200).json(user);
});
```

✓ Ensures **correct error handling and response codes**.

CHAPTER 4: HANDLING PAGINATION, FILTERING & SORTING

4.1 Implementing Pagination in REST APIs

Large datasets should be **paginated** for better performance.

📌 **Example: Using Query Parameters for Pagination**

GET /users?page=2&limit=10

✓ Fetches **page 2 with 10 users per page**.

📌 **Example: Implementing Pagination in Express.js**

```
app.get("/users", async (req, res) => {
  const page = parseInt(req.query.page) || 1;
  const limit = parseInt(req.query.limit) || 10;
  const skip = (page - 1) * limit;

  const users = await User.find().skip(skip).limit(limit);
  res.json(users);
});
```

✓ Uses .skip() and .limit() for **pagination in MongoDB**.

4.2 Filtering & Sorting Data

📌 **Example: Filtering Users by Age**

GET /users?age=30

📌 **Example: Sorting Users by Name (Descending)**

GET /users?sort=-name

✓ -name sorts users in **descending order**.

📌 **Express.js Implementation for Sorting & Filtering**

```
app.get("/users", async (req, res) => {
  const filter = req.query.age ? { age: req.query.age } : {};
```

```
  const sort = req.query.sort || "name";


  const users = await User.find(filter).sort(sort);
  res.json(users);
});
```

✓ Allows **dynamic sorting and filtering** via query parameters.

CHAPTER 5: REST API SECURITY BEST PRACTICES

- **1. Use API Keys or JWT Authentication**

✓ Require **API keys or JWT tokens** for sensitive endpoints.

```
Authorization: Bearer YOUR_JWT_TOKEN
```

- **2. Implement CORS Restrictions**

✓ Allow **only trusted domains** to access the API.

```
app.use(cors({ origin: "https://trusted.com" }));
```

- **3. Rate Limiting to Prevent Abuse**

✓ Use express-rate-limit to **limit excessive requests**.

```
const rateLimit = require("express-rate-limit");
app.use("/api", rateLimit({ windowMs: 1 * 60 * 1000, max: 100 }));
```

- **4. Validate User Input**

✓ Prevent **SQL injection & XSS attacks** with validation.

```
const { body, validationResult } = require("express-validator");
app.post("/register", [body("email").isEmail()], (req, res) => {
```

```
  if (!validationResult(req).isEmpty()) return res.status(400).json({
error: "Invalid email" });

});
```

Case Study: How Twitter Designs REST APIs

Challenges Faced by Twitter

✓ Handling **millions of real-time tweets**.
✓ Implementing **rate-limiting & pagination for efficiency**.

Solutions Implemented

✓ **Paginated API responses** for fetching tweets.
✓ **JWT authentication & API keys** for user security.
✓ **RESTful endpoints (/tweets, /users)** for a clean architecture.

- **Key Takeaways from Twitter's API Design:**

✓ **Pagination & sorting improve performance**.
✓ **Security measures like rate limiting prevent abuse**.
✓ **RESTful endpoints simplify API usage**.

📝 Exercise

✅ Design a **REST API for an e-commerce platform**.
✅ Implement **pagination & filtering for products**.
✅ Add **JWT authentication to protect user profiles**.
✅ Ensure API responses follow **RESTful principles**.

Conclusion

✓ REST API design follows standardized principles for scalability & security.

✓ Proper HTTP methods & status codes improve clarity.

✓ Pagination, filtering & sorting enhance API efficiency.

✓ Security best practices prevent unauthorized access.

USING POSTMAN FOR API TESTING

CHAPTER 1: INTRODUCTION TO API TESTING WITH POSTMAN

1.1 What is Postman?

Postman is a **popular API testing tool** that allows developers to **send, test, and automate API requests** in an easy-to-use interface. It supports **RESTful, GraphQL, and SOAP APIs**.

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

✓ **User-friendly interface** for making API requests.
✓ **Automates testing** using test scripts.
✓ **Handles authentication** (JWT, OAuth, API keys).
✓ **Supports environment variables** for dynamic testing.

- **Postman API Testing Use Cases:**

Use Case	Description
Functional Testing	Verify API responses are correct.
Performance Testing	Measure response time & efficiency.
Security Testing	Check authentication & error handling.
Integration Testing	Ensure frontend and backend communicate correctly.

CHAPTER 2: INSTALLING AND SETTING UP POSTMAN

2.1 Installing Postman

📌 **Download and install Postman from** Postman Official Website.

2.2 Creating a New API Request

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

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

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

📌 **Example API Response (JSON):**

```
[
  { "id": 1, "name": "Alice", "email": "alice@example.com" },
  { "id": 2, "name": "Bob", "email": "bob@example.com" }
]
```

CHAPTER 3: PERFORMING CRUD OPERATIONS IN POSTMAN

3.1 Sending a POST Request (Create a New User)

📌 **Example: Creating a New User**

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

{
"name": "Charlie",
"email": "charlie@example.com",
"password": "mypassword"
}

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

3.2 Sending a GET Request (Retrieve User Data)

📌 **Example: Fetching All Users**

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

✓ Returns **all users in the database**.

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

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

- **Method:** PUT
- **URL:** http://localhost:5000/api/users/1
- **Headers:**
  - Content-Type: application/json
- **Body (JSON):**

{

```
  "email": "charlie_updated@example.com"
}
```

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

3.4 Sending a DELETE Request (Remove a User)

📌 **Example: Deleting a User by ID**

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

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

Chapter 4: Authentication & Authorization Testing in Postman

4.1 Testing JWT Authentication with Postman

📌 **Step 1: Obtain a JWT Token**

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

```
{
  "email": "charlie@example.com",
  "password": "mypassword"
}
```

✓ Response returns a **JWT token**:

```
{
```

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

📌 **Step 2: Access Protected Routes Using Token**

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

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

CHAPTER 5: AUTOMATING API TESTS IN POSTMAN

5.1 Writing Postman Tests with JavaScript

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

📌 **Example: Checking Response Status Code**

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

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

📌 **Example: Validating JSON Response**

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

});

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

5.2 Running Tests with Postman Collection Runner

📌 **Steps to Automate API Testing:**

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

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

Case Study: How GitHub Uses API Testing

Challenges Faced by GitHub

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

Solutions Implemented

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

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

✓ **Automated API tests improve stability and reliability.**
✓ **JWT authentication ensures secure API communication.**
✓ **Using Postman collections streamlines large-scale API validation.**

📝 Exercise

- ✅ Set up Postman requests to **test a CRUD API**.
- ✅ Write **test scripts** to validate JSON responses.
- ✅ Use Postman to **test authentication and authorization**.
- ✅ Automate API tests using **Postman Collection Runner**.

Conclusion

✓ Postman simplifies API testing with an easy-to-use interface.
✓ Supports automated testing with JavaScript scripts.
✓ Handles authentication & role-based access control (RBAC).
✓ Improves API reliability by automating test cases.

IMPLEMENTING PAGINATION & QUERY FILTERING IN EXPRESS.JS WITH MONGODB

CHAPTER 1: INTRODUCTION TO PAGINATION & QUERY FILTERING

1.1 What is Pagination?

Pagination is a technique used to **divide large datasets into smaller chunks (pages)** to improve performance and user experience. Instead of retrieving thousands of records at once, pagination allows fetching a **limited number of results per request**.

- **Why Use Pagination?**

✓ **Improves API performance** by reducing data load.
✓ **Enhances user experience** by displaying limited results per page.
✓ **Optimizes database queries**, preventing system overload.

- **Example API Pagination Request:**

GET /products?page=2&limit=10

✓ Retrieves **page 2** with **10 items per page**.

1.2 What is Query Filtering?

Query filtering allows **retrieving specific data** from a database based on conditions. Instead of fetching all data, we **filter records based on user criteria**.

- **Why Use Query Filtering?**

✓ **Reduces data transfer** by fetching only required records.
✓ **Speeds up queries** by reducing database load.
✓ **Allows dynamic searching** with different criteria.

🔹 **Example API Query Filtering Request:**

GET /products?category=electronics&price[lt]=1000

✓ Retrieves **electronics priced below $1000**.

CHAPTER 2: IMPLEMENTING PAGINATION IN EXPRESS.JS WITH MONGODB

2.1 Basic Pagination with Mongoose

📌 **Example: Implementing Pagination in an API Route**

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

const mongoose = require("mongoose");

const Product = require("./models/Product"); // Import Mongoose model


const app = express();

app.use(express.json());


// API Route with Pagination

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

  const page = parseInt(req.query.page) || 1; // Default to page 1

  const limit = parseInt(req.query.limit) || 10; // Default to 10 items per page

  const skip = (page - 1) * limit; // Calculate offset
```

```
  try {
    const products = await Product.find().skip(skip).limit(limit);
    res.json({ page, limit, products });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

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

✓ Uses .skip(skip).limit(limit) to **paginate results**.
✓ Extracts page and limit values **from query parameters**.

📌 **Example Request:**

```
GET /products?page=2&limit=5
```

📌 **Example Response:**

```
{
  "page": 2,
  "limit": 5,
  "products": [ /* Array of 5 products */ ]
}
```

2.2 Returning Total Pages & Metadata

📌 **Example: Adding Metadata to the Pagination Response**

```
app.get("/products", async (req, res) => {
  const page = parseInt(req.query.page) || 1;
  const limit = parseInt(req.query.limit) || 10;
  const skip = (page - 1) * limit;

  try {
    const totalProducts = await Product.countDocuments(); // Count total records
    const products = await Product.find().skip(skip).limit(limit);

    res.json({
      page,
      limit,
      totalPages: Math.ceil(totalProducts / limit),
      totalProducts,
      products
    });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

✓ Includes **total pages** and **total records count** in the response.

📌 **Example Response:**

```
{
  "page": 2,
  "limit": 5,
  "totalPages": 10,
  "totalProducts": 50,
  "products": [ /* Array of 5 products */ ]
}
```

✓ Helps **frontend developers** display pagination UI.

CHAPTER 3: IMPLEMENTING QUERY FILTERING IN EXPRESS.JS

3.1 Basic Query Filtering with Mongoose

📌 **Example: Filtering Products by Category**

```
app.get("/products", async (req, res) => {
  const { category } = req.query;
  let filter = {};

  if (category) filter.category = category; // Apply category filter

  try {
    const products = await Product.find(filter);
    res.json(products);
  } catch (error) {
```

```
    res.status(500).json({ error: error.message });
  }
});
```

✓ Allows filtering by **category name** dynamically.

📌 **Example Request:**

```
GET /products?category=electronics
```

📌 **Example Response:**

```
[
  { "name": "Laptop", "category": "electronics" },
  { "name": "Smartphone", "category": "electronics" }
]
```

3.2 Advanced Query Filtering (Price Ranges, Sorting, Searching)

📌 **Example: Filtering Products by Price Range**

```
app.get("/products", async (req, res) => {
  const { category, minPrice, maxPrice } = req.query;
  let filter = {};

  if (category) filter.category = category;
  if (minPrice || maxPrice) {
    filter.price = {};
    if (minPrice) filter.price.$gte = minPrice; // Greater than or equal
```

```
    if (maxPrice) filter.price.$lte = maxPrice; // Less than or equal
  }

  try {
    const products = await Product.find(filter);
    res.json(products);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

✓ Enables **dynamic filtering by price range**.

📌 **Example Request:**

```
GET /products?minPrice=500&maxPrice=1000
```

📌 **Example Response:**

```
[
  { "name": "Smartphone", "price": 800 },
  { "name": "Tablet", "price": 900 }
]
```

3.3 Implementing Sorting & Searching

📌 **Example: Sorting Products by Price in Descending Order**

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

```
  const sortOrder = req.query.sort === "asc" ? 1 : -1;
  try {
    const products = await Product.find().sort({ price: sortOrder });
    res.json(products);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

✓ Uses .sort({ price: 1 }) for **ascending order**.
✓ Uses .sort({ price: -1 }) for **descending order**.

📌 **Example Request:**

GET /products?sort=desc

3.4 Implementing Full-Text Search in MongoDB

📌 **Step 1: Define an Index for Search**

```
ProductSchema.index({ name: "text", description: "text" });
```

📌 **Step 2: Implement Full-Text Search in Express.js**

```
app.get("/products", async (req, res) => {
  const { search } = req.query;
  let filter = {};

  if (search) filter.$text = { $search: search };
```

```
  try {
    const products = await Product.find(filter);
    res.json(products);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

✓ Enables **searching across multiple fields**.

📌 **Example Request:**

```
GET /products?search=laptop
```

📌 **Example Response:**

```
[
  { "name": "Laptop", "description": "High-performance laptop" }
]
```

CHAPTER 4: BEST PRACTICES FOR PAGINATION & FILTERING

🔹 **1. Always Provide Default Pagination Values**

✓ Prevents **overloading the database** with large queries.

🔹 **2. Use Indexing for Fast Queries**

✓ Index frequently searched fields for **optimized performance**.

🔹 **3. Implement Query Caching for Faster Responses**

✓ Use **Redis or in-memory caching** to store paginated results.

Case Study: How Amazon Uses Pagination & Filtering

Challenges Faced by Amazon

✓ Handling **millions of product searches per second**.
✓ Ensuring **fast response times** for users worldwide.

Solutions Implemented

✓ Used **pagination to load products dynamically**.
✓ Implemented **query filtering for precise search results**.
✓ Optimized **database queries with indexing**.

🔹 **Key Takeaways from Amazon's API Strategy:**
✓ **Pagination reduces API load & improves speed.**
✓ **Query filtering enhances user experience.**
✓ **Indexing ensures fast product retrieval.**

📝 Exercise

✅ Implement **pagination with metadata (total pages & records)**.
✅ Create an API to **filter products by category, price, and search keyword**.
✅ Implement **sorting by price (asc/desc)**.
✅ Use **MongoDB indexing to optimize query performance**.

Conclusion

✓ **Pagination improves API performance by limiting data loads.**
✓ **Query filtering helps fetch specific results dynamically.**

✓ **Sorting & searching enhance user experience in applications.**

✓ **Using indexing and caching ensures fast query execution.**

SETTING UP GRAPHQL IN EXPRESS.JS

CHAPTER 1: INTRODUCTION TO GRAPHQL

1.1 What is GraphQL?

GraphQL is a **query language for APIs** that provides a flexible and efficient way to request and manage data. Unlike REST, which requires multiple endpoints, GraphQL allows clients to **query multiple resources from a single endpoint**.

- **Why Use GraphQL Over REST?**

✓ Fetch **only the required data** (avoids over-fetching or under-fetching).

✓ Reduce **multiple API calls** to a single request.

✓ Flexible querying using **schema-based structure**.

- **Comparison: GraphQL vs. REST APIs**

Feature	GraphQL	REST
Data Fetching	Client specifies fields	Fixed endpoints return all data
Number of Requests	Single request for multiple resources	Multiple requests for related data
API Versioning	No need for versions	Often requires versioning (v1, v2)

CHAPTER 2: INSTALLING GRAPHQL IN EXPRESS.JS

2.1 Prerequisites

📌 **Ensure Node.js & Express.js are Installed**

node -v

npm init -y

npm install express

2.2 Installing GraphQL & Apollo Server

📌 **Install GraphQL and Apollo Server for Express.js**

npm install graphql express-graphql apollo-server-express

✓ **graphql** – Core package for GraphQL.
✓ **express-graphql** – Middleware to integrate GraphQL with Express.js.
✓ **apollo-server-express** – Popular GraphQL server for Express.

CHAPTER 3: SETTING UP A BASIC GRAPHQL SERVER

3.1 Creating an Express Server with GraphQL

📌 **Step 1: Create server.js and Add GraphQL Middleware**

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

const { graphqlHTTP } = require("express-graphql");

const { buildSchema } = require("graphql");


const app = express();


// Define GraphQL Schema

const schema = buildSchema(`
```

```
  type Query {
    message: String
  }
`);

// Define Resolvers
const root = {
  message: () => "Hello, GraphQL!"
};

// Set up GraphQL Middleware
app.use("/graphql", graphqlHTTP({
  schema: schema,
  rootValue: root,
  graphiql: true
}));

const PORT = 5000;
app.listen(PORT, () => console.log(`GraphQL Server running at http://localhost:${PORT}/graphql`));
```

✓ Defines a **GraphQL schema** with a simple "message" query.
✓ Uses **express-graphql** middleware to handle GraphQL requests.
✓ Enables **GraphiQL** for testing queries in the browser.

3.2 Testing GraphQL Queries in GraphiQL

📌 Run the Server and Open GraphiQL

node server.js

✓ Go to **http://localhost:5000/graphql** and enter the following query:

```
{
  message
}
```

✓ Expected Response:

```
{
  "data": {
    "message": "Hello, GraphQL!"
  }
}
```

CHAPTER 4: DEFINING GRAPHQL TYPES & QUERIES

4.1 Creating a User Type in GraphQL

📌 Modify server.js to Add a User Type and Query

```
const schema = buildSchema(`
  type User {
    id: ID
```

```
    name: String
    email: String
  }

  type Query {
    user(id: ID!): User
    users: [User]
  }
`);

const usersData = [
  { id: "1", name: "Alice", email: "alice@example.com" },
  { id: "2", name: "Bob", email: "bob@example.com" }
];

// Define Resolvers
const root = {
  user: ({ id }) => usersData.find(user => user.id === id),
  users: () => usersData
};
```

✓ **user(id: ID!)** – Fetch a user by ID.
✓ **users: [User]** – Fetch all users.

📌 Testing Queries in GraphiQL

```
{
  users {
    id
    name
    email
  }
}
```

✓ Should return all users as JSON.

CHAPTER 5: HANDLING MUTATIONS IN GRAPHQL

5.1 What is a Mutation?

Mutations allow **creating, updating, and deleting data** in GraphQL.

📌 **Modify server.js to Add Mutations**

```
const schema = buildSchema(`
  type User {
    id: ID
    name: String
    email: String
  }

  type Query {
    users: [User]
```

```
  }

  type Mutation {
    addUser(name: String!, email: String!): User
  }
`);

let usersData = [];

const root = {
  users: () => usersData,
  addUser: ({ name, email }) => {
    const newUser = { id: usersData.length + 1, name, email };
    usersData.push(newUser);
    return newUser;
  }
};
```

✓ Defines a **mutation to create users** dynamically.

📌 Testing Mutation in GraphiQL

```
mutation {
  addUser(name: "Charlie", email: "charlie@example.com") {
    id
```

```
    name
    email
  }
}
```

✓ Adds a new user and returns their details.

CHAPTER 6: CONNECTING GRAPHQL WITH A DATABASE (MONGODB + SEQUELIZE)

6.1 Using GraphQL with MongoDB & Mongoose

📌 Install Mongoose for MongoDB

```
npm install mongoose
```

📌 Define a User Model (models/User.js)

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

const UserSchema = new mongoose.Schema({
  name: String,
  email: String
});

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

📌 Modify GraphQL Resolver to Fetch Data from MongoDB

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

```
const root = {
  users: async () => await User.find(),
  addUser: async ({ name, email }) => {
    const user = new User({ name, email });
    await user.save();
    return user;
  }
};
```

✓ GraphQL now **fetches and modifies data from MongoDB**.

6.2 Using GraphQL with MySQL & Sequelize

📌 Install Sequelize & MySQL Drivers

```
npm install sequelize mysql2
```

📌 Set Up Sequelize Connection (db.js)

```
const { Sequelize } = require("sequelize");
const sequelize = new Sequelize("database_name", "username", "password", {
  host: "localhost",
  dialect: "mysql"
});
module.exports = sequelize;
```

📌 Modify GraphQL Resolver to Use Sequelize

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


const root = {
  users: async () => await User.findAll(),
  addUser: async ({ name, email }) => {
    return await User.create({ name, email });
  }
};
```

✓ GraphQL now **fetches and modifies data from MySQL**.

Case Study: How GitHub Uses GraphQL for APIs

Challenges Faced by GitHub

✓ Managing **millions of repositories and users**.
✓ Reducing **API over-fetching issues** in REST.

Solutions Implemented

✓ Used **GraphQL to allow flexible data fetching**.
✓ Improved **performance by fetching only required fields**.
✓ Provided **GraphQL-powered GitHub API for developers**.

- **Key Takeaways from GitHub's GraphQL Strategy:**

✓ **Optimized queries improve API efficiency.**
✓ **Fetching only required fields reduces data load.**
✓ **GraphQL enhances API flexibility and performance.**

📝 Exercise

✅ Set up a **GraphQL API** in Express.js.

✅ Create **Queries & Mutations** for fetching and updating user data.

✅ Integrate **MongoDB or MySQL** with GraphQL using Sequelize/Mongoose.

✅ Test GraphQL queries in **GraphiQL or Postman**.

Conclusion

✓ GraphQL provides a flexible way to fetch data from APIs.

✓ Express.js can efficiently serve GraphQL queries and mutations.

✓ Sequelize & Mongoose help manage database interactions.

✓ GraphQL improves API performance compared to REST.

CREATING QUERIES, MUTATIONS & RESOLVERS IN GRAPHQL WITH EXPRESS.JS

CHAPTER 1: INTRODUCTION TO GRAPHQL

1.1 What is GraphQL?

GraphQL is a **query language for APIs** that allows clients to request **specific data** rather than retrieving fixed responses from traditional REST APIs. It is developed by **Facebook** and offers more flexibility in API communication.

🔹 **Why Use GraphQL?**

✓ Clients **request only the data they need,** reducing over-fetching.

✓ Combines **multiple REST API calls** into a single query.

✓ Provides **strongly typed schemas** for better structure and validation.

🔹 **GraphQL vs. REST API**

Feature	REST API	GraphQL
Data Fetching	Returns **fixed** response	Client requests **specific fields**
Multiple Requests	Needs **separate calls** for multiple resources	**Single query** fetches all required data
Versioning	Requires **new endpoints** for updates	Uses **schema evolution**

📌 **Example: Querying a REST API vs. GraphQL API**

✅ **REST API Request**

GET /users/1

GET /posts?userId=1

✓ Requires **multiple calls** to fetch user & posts.

✅ **GraphQL API Request**

```
{
  user(id: 1) {
    name
    email
    posts {
      title
    }
  }
}
```

✓ Retrieves **all data in a single request**.

CHAPTER 2: SETTING UP GRAPHQL WITH EXPRESS.JS

2.1 Installing Required Packages

📌 **Install GraphQL & Apollo Server**

npm install express graphql express-graphql apollo-server-express

✓ express-graphql → Integrates GraphQL with Express.js.
✓ apollo-server-express → Provides tools for schema management.

📌 **Create a Basic Express Server with GraphQL (server.js)**

const express = require("express");

```
const { graphqlHTTP } = require("express-graphql");
const { buildSchema } = require("graphql");

const app = express();

const schema = buildSchema(`
  type Query {
    hello: String
  }
`);

const root = {
  hello: () => "Hello, GraphQL!"
};

app.use("/graphql", graphqlHTTP({
  schema: schema,
  rootValue: root,
  graphiql: true
}));

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

📌 Start the Server

node server.js

✓ Opens GraphiQL at http://localhost:5000/graphql.

📌 Example Query in GraphiQL:

```
{
  hello
}
```

📌 Response:

```
{
  "data": {
    "hello": "Hello, GraphQL!"
  }
}
```

✓ Confirms **GraphQL API is working**.

CHAPTER 3: WRITING QUERIES IN GRAPHQL

3.1 What is a GraphQL Query?

A **GraphQL Query** is used to **fetch data** from a server.

📌 Example: Define a User Schema & Query (server.js)

```
const schema = buildSchema(`
  type User {
    id: ID
```

```
    name: String
    email: String
  }

  type Query {
    user(id: ID!): User
  }
`);

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

const root = {
  user: ({ id }) => users.find(user => user.id === id)
};
```

✓ Defines a **User type** with fields id, name, and email.
✓ Creates a **user query** to fetch data by id.

📌 **GraphQL Query Example:**

```
{
  user(id: "1") {
```

```
    name
    email
  }
}
```

📌 **Response:**

```
{
  "data": {
    "user": {
      "name": "Alice",
      "email": "alice@example.com"
    }
  }
}
```

✓ Fetches **specific user fields** instead of entire objects.

CHAPTER 4: WRITING MUTATIONS IN GRAPHQL

4.1 What is a Mutation?

A **GraphQL Mutation** allows **creating, updating, and deleting data**.

📌 **Example: Adding a Mutation to Create Users (server.js)**

```
const schema = buildSchema(`
  type User {
    id: ID
```

```
    name: String
    email: String
  }

  type Query {
    user(id: ID!): User
  }

  type Mutation {
    addUser(name: String!, email: String!): User
  }
`);

const users = [];

const root = {
  addUser: ({ name, email }) => {
    const newUser = { id: users.length + 1, name, email };
    users.push(newUser);
    return newUser;
  }
};
```

✓ Defines addUser mutation for **creating new users**.

📌 **Mutation Example in GraphiQL:**

```
mutation {
  addUser(name: "Charlie", email: "charlie@example.com") {
    id
    name
    email
  }
}
```

📌 **Response:**

```
{
  "data": {
    "addUser": {
      "id": "1",
      "name": "Charlie",
      "email": "charlie@example.com"
    }
  }
}
```

✓ Creates a **new user dynamically**.

CHAPTER 5: USING RESOLVERS IN GRAPHQL

5.1 What is a Resolver?

A **resolver** is a function that **fetches or modifies data** based on GraphQL queries or mutations.

🔹 How Resolvers Work in GraphQL

1. **Queries** → Fetch data from a database or array.
2. **Mutations** → Create, update, or delete data.

📌 Example: Implementing Resolvers for Queries & Mutations

```
const resolvers = {
  Query: {
    user: (_, { id }) => users.find(user => user.id === id)
  },
  Mutation: {
    addUser: (_, { name, email }) => {
      const newUser = { id: users.length + 1, name, email };
      users.push(newUser);
      return newUser;
    }
  }
};
```

✓ **Separates query & mutation logic** into dedicated resolvers.

📌 GraphQL Query Example:

```
{
```

```
  user(id: "1") {
    name
    email
  }
}
```

📌 **GraphQL Mutation Example:**

```
mutation {
  addUser(name: "Dave", email: "dave@example.com") {
    id
    name
  }
}
```

✓ Uses resolvers to **handle different GraphQL operations efficiently**.

Chapter 6: Best Practices for GraphQL APIs

🔹 **1. Use Schema-First Design**

✓ Define GraphQL schema **before implementing logic**.

🔹 **2. Implement Input Validation**

✓ Use GraphQL types to enforce required fields (!).

🔹 **3. Optimize Queries with Batching & Caching**

✓ Use **DataLoader** to minimize database queries.

🔹 **4. Secure APIs with Authentication & Rate Limiting**

✓ Require **JWT tokens** for private queries.

Case Study: How Shopify Uses GraphQL

Challenges Faced by Shopify

✓ Handling **millions of API requests from merchants**.
✓ Reducing **over-fetching & under-fetching of data**.

Solutions Implemented

✓ **GraphQL for flexible, efficient API requests**.
✓ **Implemented batch queries** to minimize API calls.
✓ **Optimized query execution time** using caching.

🔹 **Key Takeaways from Shopify's GraphQL API Strategy:**

✓ **GraphQL reduces API response time & load**.
✓ **Resolvers improve data retrieval efficiency**.
✓ **Schema-first design ensures maintainability**.

📝 Exercise

✅ Create a **GraphQL API with queries & mutations**.
✅ Implement a **resolver function for fetching user data**.
✅ Secure the GraphQL API with **JWT authentication**.
✅ Deploy the **GraphQL server using Apollo & Express.js**.

Conclusion

✓ **GraphQL simplifies API development with flexible queries.**
✓ **Resolvers fetch or modify data dynamically.**

✓ **Mutations enable real-time data updates.**

✓ **Optimizing queries improves API performance & scalability.**

HANDLING AUTHENTICATION IN GRAPHQL APIS

CHAPTER 1: INTRODUCTION TO AUTHENTICATION IN GRAPHQL

1.1 What is Authentication in GraphQL?

Authentication in GraphQL ensures that **only authorized users can access protected data**. It verifies user identity before allowing access to **queries, mutations, and subscriptions**.

- **Why is Authentication Important?**

✓ **Prevents unauthorized access** to sensitive information.
✓ **Ensures data integrity** by restricting actions to verified users.
✓ **Secures API endpoints** against malicious attacks.

- **Common Authentication Methods in GraphQL:**

Method	Description	Example
JWT (JSON Web Tokens)	Token-based authentication using Authorization headers.	Used in APIs & microservices.
Session-based Authentication	Stores session data on the server using cookies.	Best for web apps.
OAuth 2.0	Uses third-party authentication providers (Google, GitHub).	Social logins.
API Key Authentication	Clients provide an API key to access the API.	Used in public APIs.

CHAPTER 2: SETTING UP A GRAPHQL API WITH AUTHENTICATION

2.1 Installing Required Dependencies

📌 Step 1: Initialize a Node.js Project & Install Dependencies

mkdir graphql-auth

cd graphql-auth

npm init -y

npm install express express-graphql graphql bcryptjs jsonwebtoken cors dotenv mongoose

✓ express-graphql → Middleware for handling GraphQL requests.
✓ bcryptjs → Hashes passwords securely.
✓ jsonwebtoken → Generates & verifies JWT tokens.
✓ mongoose → Handles MongoDB database operations.

CHAPTER 3: IMPLEMENTING USER AUTHENTICATION IN GRAPHQL

3.1 Setting Up User Model with Mongoose

📌 Step 2: Create models/User.js

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

const bcrypt = require("bcryptjs");


const UserSchema = new mongoose.Schema({

  username: { type: String, required: true, unique: true },

  email: { type: String, required: true, unique: true },

  password: { type: String, required: true }

});
```

```
// Hash password before saving
UserSchema.pre("save", async function (next) {
  if (!this.isModified("password")) return next();
  this.password = await bcrypt.hash(this.password, 10);
  next();
});

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

✓ Ensures **passwords are hashed before storing in MongoDB**.

3.2 Creating GraphQL Schema with Authentication

📌 Step 3: Define Schema & Resolvers in schema.js

```
const { GraphQLObjectType, GraphQLString, GraphQLSchema } = require("graphql");
const User = require("./models/User");
const bcrypt = require("bcryptjs");
const jwt = require("jsonwebtoken");

// Define User Type
const UserType = new GraphQLObjectType({
  name: "User",
  fields: () => ({
```

```
    id: { type: GraphQLString },
    username: { type: GraphQLString },
    email: { type: GraphQLString },
    token: { type: GraphQLString }
  })
});

// Mutation: Register User
const RootMutation = new GraphQLObjectType({
  name: "Mutation",
  fields: {
    register: {
      type: UserType,
      args: {
        username: { type: GraphQLString },
        email: { type: GraphQLString },
        password: { type: GraphQLString }
      },
      async resolve(parent, args) {
        const hashedPassword = await bcrypt.hash(args.password,
10);
        const newUser = new User({ username: args.username,
email: args.email, password: hashedPassword });
```

```
        await newUser.save();
        return newUser;
      }
    },
    login: {
      type: UserType,
      args: {
        email: { type: GraphQLString },
        password: { type: GraphQLString }
      },
      async resolve(parent, args) {
        const user = await User.findOne({ email: args.email });
        if (!user) throw new Error("User not found");

        const isMatch = await bcrypt.compare(args.password,
user.password);
        if (!isMatch) throw new Error("Invalid credentials");

        const token = jwt.sign({ userId: user.id }, "mysecretkey", {
expiresIn: "1h" });
        return { ...user._doc, token };
      }
    }
```

```
  }
});

// Define Query Type
const RootQuery = new GraphQLObjectType({
  name: "Query",
  fields: {
    userProfile: {
      type: UserType,
      args: { id: { type: GraphQLString } },
      async resolve(parent, args, context) {
        if (!context.user) throw new Error("Unauthorized");
        return await User.findById(args.id);
      }
    }
  }
});

module.exports = new GraphQLSchema({
  query: RootQuery,
  mutation: RootMutation
});
```

✓ **Registers new users and generates JWT tokens on login.**

✓ **Protects user profile route using authentication middleware.**

CHAPTER 4: IMPLEMENTING AUTHENTICATION MIDDLEWARE IN GRAPHQL

4.1 Creating Authentication Middleware

📌 Step 4: Add authMiddleware.js

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


const authMiddleware = (req) => {
  const authHeader = req.headers.authorization;
  if (authHeader) {
    const token = authHeader.split(" ")[1];
    try {
      const user = jwt.verify(token, "mysecretkey");
      return { user };
    } catch (err) {
      throw new Error("Invalid/Expired Token");
    }
  }
  throw new Error("Authorization token required");
};
```

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

✓ Extracts **JWT from request headers** and verifies it.

4.2 Using Middleware in GraphQL Server

📌 Step 5: Integrate Middleware in server.js

```
const express = require("express");
const { graphqlHTTP } = require("express-graphql");
const schema = require("./schema");
const mongoose = require("mongoose");
const authMiddleware = require("./authMiddleware");

mongoose.connect("mongodb://127.0.0.1:27017/graphqlAuthDB", {
  useNewUrlParser: true,
  useUnifiedTopology: true
});

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

app.use(
  "/graphql",
  graphqlHTTP((req) => ({
```

```
    schema,
    graphiql: true,
    context: authMiddleware(req)
  }))
);

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

✓ Integrates **authentication middleware** with GraphQL context.
✓ Secures API endpoints by requiring **JWT authentication**.

CHAPTER 5: TESTING AUTHENTICATION IN POSTMAN

5.1 Registering a User

📌 **GraphQL Mutation Request (Register User)**

```
mutation {
 register(username: "johnDoe", email: "john@example.com", password: "mypassword") {
   id
   username
   email
 }
}
```

✓ Creates a **new user in MongoDB**.

5.2 Logging in to Get JWT Token

📌 GraphQL Mutation Request (Login User)

```
mutation {
 login(email: "john@example.com", password: "mypassword") {
  token
 }
}
```

✓ Returns **JWT token for authentication**.

5.3 Fetching User Profile (Requires Token)

📌 GraphQL Query Request (Using Token in Headers)

```
query {
 userProfile(id: "USER_ID_HERE") {
  username
  email
 }
}
```

✓ Authorization Header:

Authorization: Bearer YOUR_JWT_TOKEN

✓ Returns **user profile data only if authentication is valid**.

Case Study: How GitHub Uses GraphQL Authentication

Challenges Faced by GitHub

✓ Managing **millions of authenticated API requests**.
✓ Ensuring **secure OAuth authentication** for third-party apps.

Solutions Implemented

✓ Used **GraphQL authentication with OAuth & JWT**.
✓ Implemented **rate-limiting and token expiry** for security.

🔹 **Key Takeaways from GitHub's API Security:**

✓ **JWT ensures secure authentication across GraphQL APIs.**
✓ **OAuth enables third-party authentication.**
✓ **Using authorization headers helps secure API access.**

📝 Exercise

✅ Implement **JWT authentication in a GraphQL API**.
✅ Secure a **GraphQL query (userProfile) with authentication**.
✅ Test authentication using **Postman GraphQL API requests**.

Conclusion

✓ **GraphQL authentication secures API requests.**
✓ **JWT enables stateless, token-based authentication.**
✓ **Middleware verifies user identity before executing GraphQL queries.**

ASSIGNMENT:

DEVELOP A GRAPHQL API WITH EXPRESS.JS & MONGODB

Assignment Solution: Develop a GraphQL API with Express.js & MongoDB

Step 1: Setting Up the Project

1.1 Initialize a Node.js Project

📌 **Create a project directory and initialize Node.js:**

```
mkdir graphql-api

cd graphql-api

npm init -y
```

📌 **Install Required Packages:**

```
npm install express express-graphql graphql mongoose dotenv
```

✓ express → Web framework for handling requests.
✓ express-graphql → Middleware for integrating GraphQL with Express.
✓ graphql → GraphQL query language support.
✓ mongoose → ORM for MongoDB.
✓ dotenv → Load environment variables from .env.

Step 2: Setting Up MongoDB Connection

2.1 Create a .env File

📌 **Define MongoDB connection string:**

```
MONGO_URI=mongodb://127.0.0.1:27017/graphqlDB
```

```
PORT=5000
```

2.2 Connect Express.js to MongoDB

📌 Create server.js and add the MongoDB connection:

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

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

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

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

✓ Connects Express to MongoDB using **Mongoose**.

Step 3: Defining the Mongoose Schema & Model

📌 **Create models/User.js and define the User schema:**

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

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

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

✓ Defines a **User model** with name, email, and age.

Step 4: Setting Up GraphQL with Express.js

4.1 Import GraphQL & Define Schema

📌 **Modify server.js to include GraphQL setup:**

```
const { graphqlHTTP } = require("express-graphql");
const { GraphQLSchema, GraphQLObjectType, GraphQLString, GraphQLInt, GraphQLID, GraphQLList, GraphQLNonNull } = require("graphql");
const User = require("./models/User");

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

4.2 Creating a GraphQL Schema & Root Query

📌 Define the GraphQL User Type & Root Query:

```
const UserType = new GraphQLObjectType({
  name: "User",
  fields: () => ({
    id: { type: GraphQLID },
    name: { type: GraphQLString },
    email: { type: GraphQLString },
    age: { type: GraphQLInt }
  })
});

const RootQuery = new GraphQLObjectType({
  name: "RootQueryType",
  fields: {
    user: {
      type: UserType,
      args: { id: { type: GraphQLID } },
      resolve(parent, args) {
        return User.findById(args.id);
      }
```

```
    },
    users: {
      type: new GraphQLList(UserType),
      resolve() {
        return User.find();
      }
    }
  }
});
```

✓ Defines **GraphQL types and queries** for fetching users.

4.3 Implementing GraphQL Mutations

📌 **Define Mutations for Creating, Updating, and Deleting Users:**

```
const Mutation = new GraphQLObjectType({
  name: "Mutation",
  fields: {
    addUser: {
      type: UserType,
      args: {
        name: { type: new GraphQLNonNull(GraphQLString) },
        email: { type: new GraphQLNonNull(GraphQLString) },
        age: { type: GraphQLInt }
```

```
      },
      resolve(parent, args) {
        let user = new User({ name: args.name, email: args.email,
age: args.age });
        return user.save();
      }
    },
    updateUser: {
      type: UserType,
      args: {
        id: { type: new GraphQLNonNull(GraphQLID) },
        name: { type: GraphQLString },
        email: { type: GraphQLString },
        age: { type: GraphQLInt }
      },
      resolve(parent, args) {
        return User.findByIdAndUpdate(args.id, args, { new: true });
      }
    },
    deleteUser: {
      type: UserType,
      args: { id: { type: new GraphQLNonNull(GraphQLID) } },
      resolve(parent, args) {
```

```
            return User.findByIdAndDelete(args.id);
        }
    }
  }
});
```

✓ Implements **CRUD operations** with GraphQL Mutations.

Step 5: Integrating GraphQL with Express.js

📌 Finalize GraphQL Server Setup in server.js:

```
const schema = new GraphQLSchema({ query: RootQuery, mutation: Mutation });


app.use("/graphql", graphqlHTTP({
  schema,
  graphiql: true // Enable GraphiQL UI
}));


console.log("GraphQL API ready at http://localhost:5000/graphql");
```

✓ Uses express-graphql to expose GraphQL API on /graphql.

Step 6: Testing the GraphQL API in GraphiQL

6.1 Running the Server

📌 Start the Express Server:

node server.js

✓ API is now available at http://localhost:5000/graphql.

6.2 Running GraphQL Queries & Mutations

📌 Query to Fetch All Users

```
{
  users {
    id
    name
    email
    age
  }
}
```

📌 Mutation to Add a New User

```
mutation {
  addUser(name: "Alice", email: "alice@example.com", age: 25) {
    id
    name
  }
}
```

📌 Mutation to Update a User's Age

```
mutation {
  updateUser(id: "USER_ID_HERE", age: 30) {
    id
    name
    age
  }
}
```

📌 **Mutation to Delete a User**

```
mutation {
  deleteUser(id: "USER_ID_HERE") {
    id
    name
  }
}
```

✓ Executes CRUD operations using GraphQL.

Case Study: How GitHub Uses GraphQL for API Efficiency

Challenges Faced by GitHub

✓ Needed **faster API performance** for fetching user repositories.
✓ Wanted **flexible queries** to reduce unnecessary data fetching.

Solutions Implemented

✓ Used **GraphQL instead of REST APIs** for efficient data fetching.
✓ Allowed clients to **request only necessary fields**.

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

✓ **GraphQL reduces over-fetching** by retrieving only required fields.

✓ **Performance improves with optimized database queries**.

📝 **Exercise**

✅ Add a **new field (phoneNumber) to the User schema** and update queries.

✅ Implement a **GraphQL query to fetch users based on age range**.

✅ Create a **search query** that filters users by name.

✅ Deploy the GraphQL API to **Heroku or Vercel**.

Conclusion

✓ **GraphQL allows flexible data fetching compared to REST APIs.**

✓ **Express.js & MongoDB enable scalable backend APIs.**

✓ **GraphQL Mutations handle CRUD operations efficiently.**

✓ **Using express-graphql simplifies GraphQL server setup.**