Design and Implementation of a High Performance E-Commerce GraphQL API

A PROJECT REPORT

Submitted by

Jobanjot Singh Grewal
23BIA50005

in partial fulfillment for the award of the degree of

Bachelor of Engineering + Master of Engineering (integrated)

IN

Artificial Intelligence and Machine Learning

1. Project Title

Design and Implement a GraphQL API for an e-commerce platform using Node.js (Express) with Apollo Server over MongoDB that exposes products, categories, and orders through a unified schema.

2. Project Description

This capstone project involved the design and implementation of a full-stack e-commerce application utilizing a modern technology stack. The core of the project is a robust **GraphQL API** built with **Node.js, Express, and Apollo Server**, which serves as the backend for a **React-based frontend**.

The primary objective was to create a unified data graph that exposes products, categories, users, and orders, allowing clients to request exactly the data they need, thereby solving the over-fetching and under-fetching problems common in REST APIs. The system supports advanced features such as pagination, filtering, sorting, and a secure cart-to-checkout flow.

Key functionalities implemented include:

- **Unified GraphQL Schema:** A single endpoint for all data queries and mutations.
- **Efficient Data Fetching:** Utilizes DataLoader to batch and cache database requests, eliminating the N+1 query problem.
- **Authentication & Authorization:** JWT-based secure access, with role-based permissions (Customer/Admin).
- **Product Catalog:** Advanced filtering, sorting, and search capabilities on products and categories.
- **Order Management:** Complete flow for users to add items to a cart, place orders, and for admins to manage order status.
- **Performance Optimized:** Lean MongoDB queries based on the GraphQL selection set and strategic database indexing.
- **Modern Frontend:** A responsive UI built with React, Apollo Client, and Tailwind CSS.

The project demonstrates a deep understanding of full-stack development principles, API design, database optimization, and secure coding practices.

3. Hardware /SoftwareRequirements

3.1 Software Requirements

Category	Technology	Version	Purpose / Rationale
Backend Runtime	Node.js	18.x+	Server-side JavaScript runtime
Backend Server	Express	5.1.0	Web framework
GraphQL API	@apollo/server	5.1.0	Core Apollo GraphQL server
Database	MongoDB	6.x+	Scalable NoSQL database
ODM	Mongoose	8.19.2	Schema validation & modeling
Batching	dataloader	2.2.3	Solves N+1 queries
Auth	jsonwebtoken	9.0.2	JWT generation/verification
Validation	joi	18.0.1	Input validation for mutations
Development tool	nodemon	3.1.10	Auto-restart server on file changes
Frontend	React	19.1.1	Client UI framework

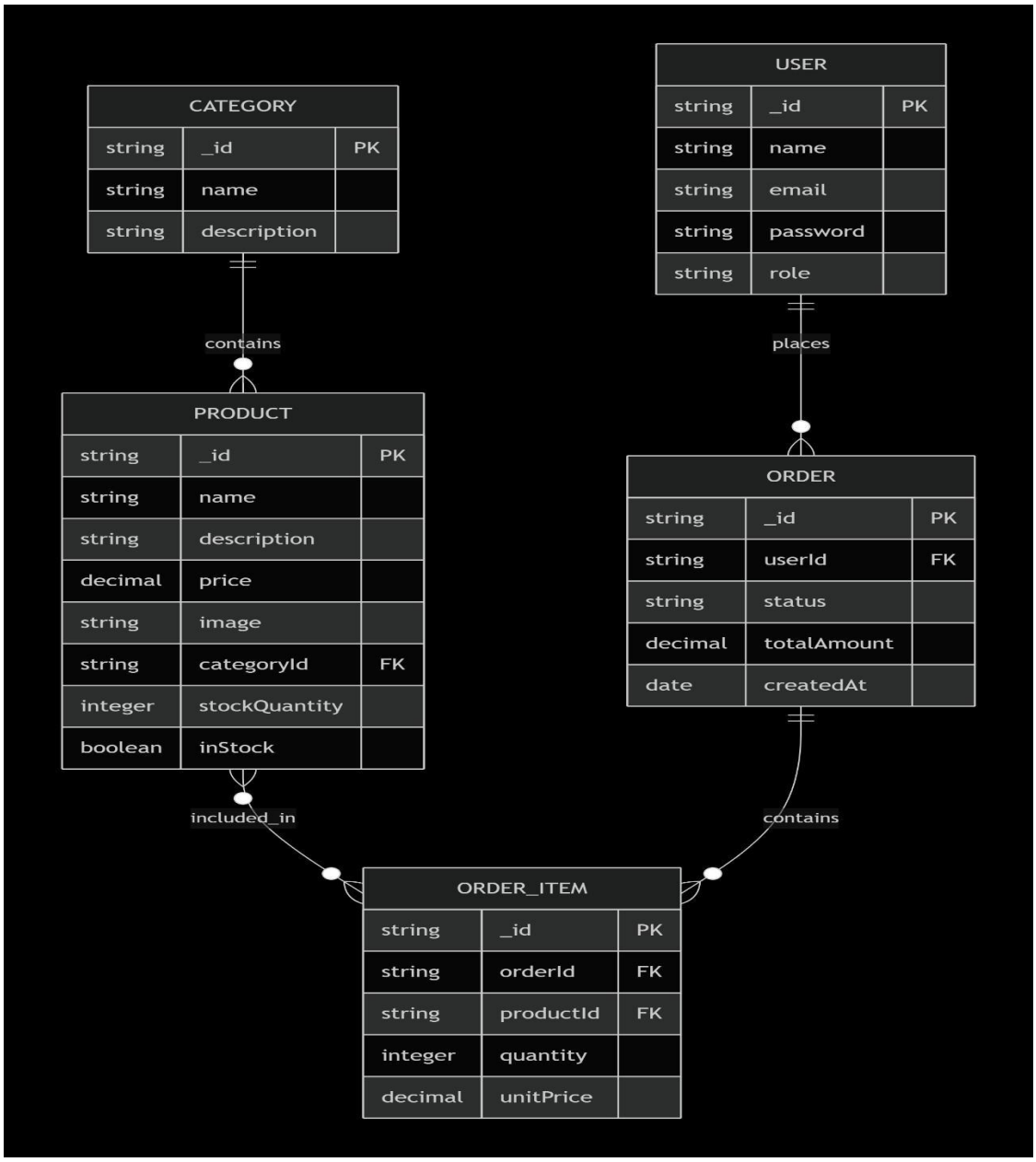
3.2 Hardware Requirements

System Type	CPU	RAM	Storage
Development Machine	2.0 GHz Dual-Core or better	8 GB (16 GB recommended)	50 GB SSD
Production Server	AWS EC2 t3.small (2+ vCPUs)	2–4 GB	—

4. ER Diagram

The Entity-Relationship Diagram below illustrates the core data models and their relationships in the MongoDB database. The key entities are **User**, **Product**, **Category**, **Order**, and **OrderItem**.

Figure 1: Entity-Relationship Diagram for the E-Commerce Platform



5. Database Schema

The database schema is defined using Mongoose Schemas within the Node.js application. Below are the key schema definitions.

User Schema

javascript

```
{
```

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

true, unique: true }, password: { type: String, required: true }, role:

{ type: String, enum: ['customer', 'admin'], default:

'customer' }

}

Category Schema

javascript

{

name: { type: String, required: true, unique: true }, description:

{ type: String }

}

Product Schema

javascript

{

name: { type: String, required: true }, description: { type: String, required: true },

price: { type: mongoose.Decimal128, required: true }, image: { type: String },

category: { type: mongoose.Schema.Types.ObjectId, ref: 'Category', required: true },

stockQuantity: { type: Number, default: 0 }, inStock: { type: Boolean, default: true }

}

Order Schema

javascript

{

user: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },

status: { type: String, enum: ['pending', 'confirmed', 'shipped', 'delivered', 'cancelled'], default:
'pending' }, totalAmount: { type: mongoose.Decimal128, required: true },

items: [{ product: { type: mongoose.Schema.Types.ObjectId, ref: 'Product',

required: true }, quantity: { type: Number, required: true, min: 1 }, unitPrice:

{

type: mongoose.Decimal128, required: true }

}]

},

{ timestamps: true }

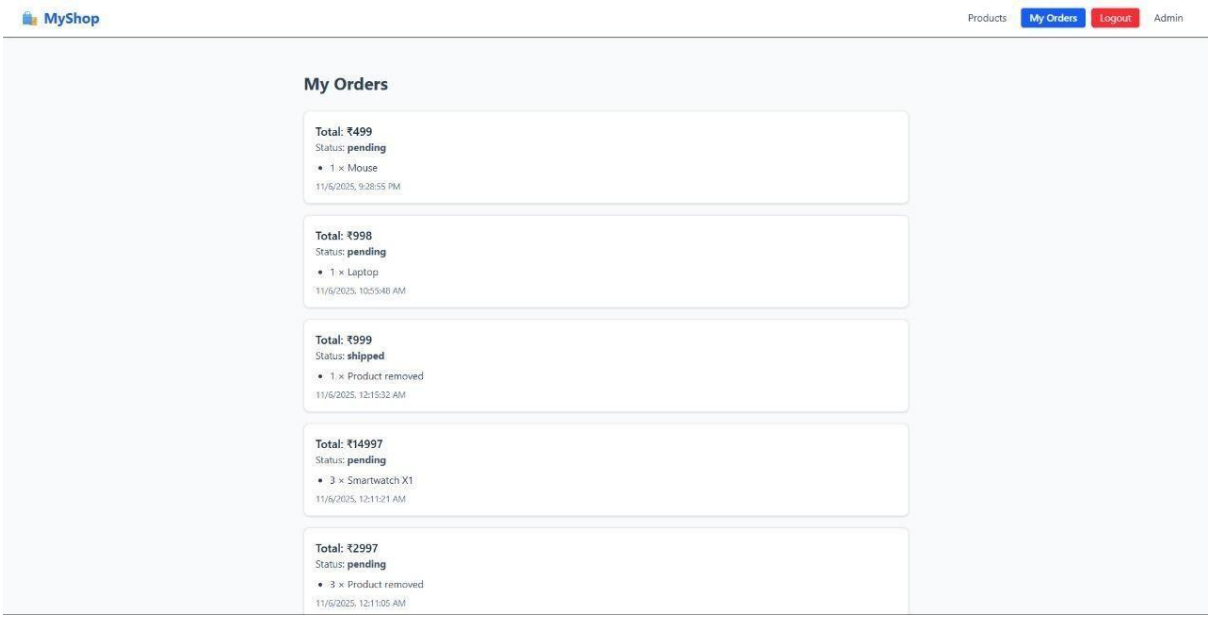
```
┌──────────────────────────────────────────────────────┐
│                    ORDER SCHEMA                      │
├──────────────────────────────────────────────────────┤
│  user: ObjectId (ref: User) [Required]               │
│  status: String [enum: pending,confirmed,shipped,delivered,│
│         cancelled] [Default: 'pending']              │
│  totalAmount: Decimal128 [Required]                  │
│  items: Array [Subdocument]                          │
│  ┌──────────────────────────────────────────────┐    │
│  │                ORDER ITEM                    │    │
│  ├──────────────────────────────────────────────┤    │
│  │  product: ObjectId (ref: Product) [Required] │    │
│  │  quantity: Number [Required, min: 1]         │    │
│  │  unitPrice: Decimal128 [Required]            │    │
│  └──────────────────────────────────────────────┘    │
├──────────────────────────────────────────────────────┤
│  createdAt: Date [Auto]                              │
│  updatedAt: Date [Auto]                              │
└──────────────────────────────────────────────────────┘
```

6. Front-End Screens

The React frontend application provides a user-friendly interface for interacting with the GraphQL API. The following are some of the key frontend screens designed for the application.

6.1 Product Listing Page

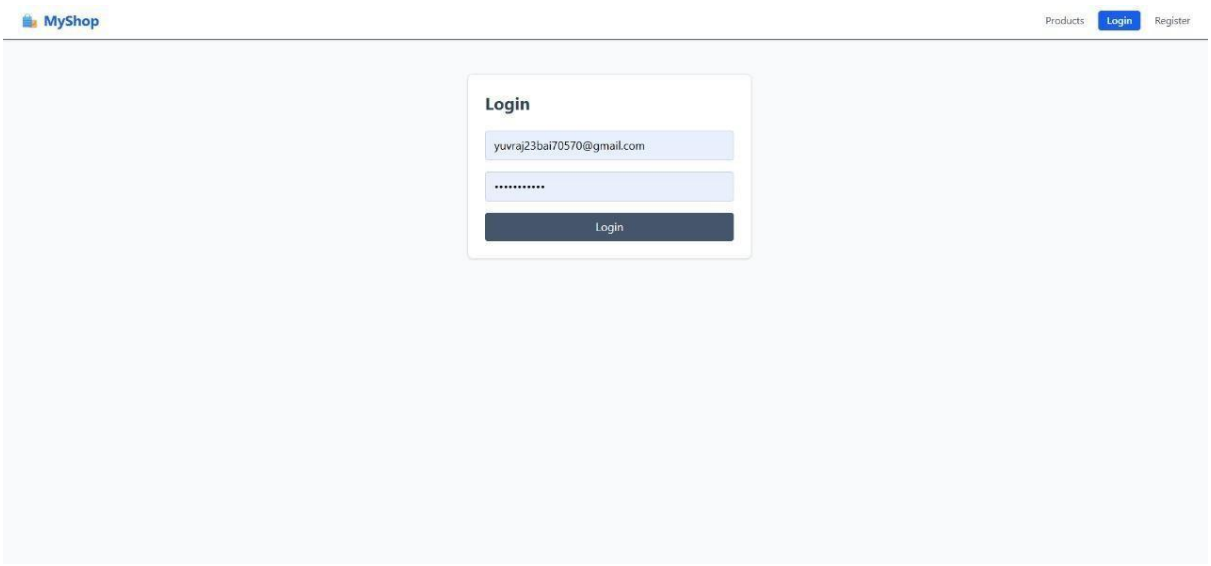
This page displays all available products. It includes features like filtering by category, sorting by price/name, and searching. The UI is built using Tailwind CSS for a clean and responsive layout.



6.2 User Login/Signup Page

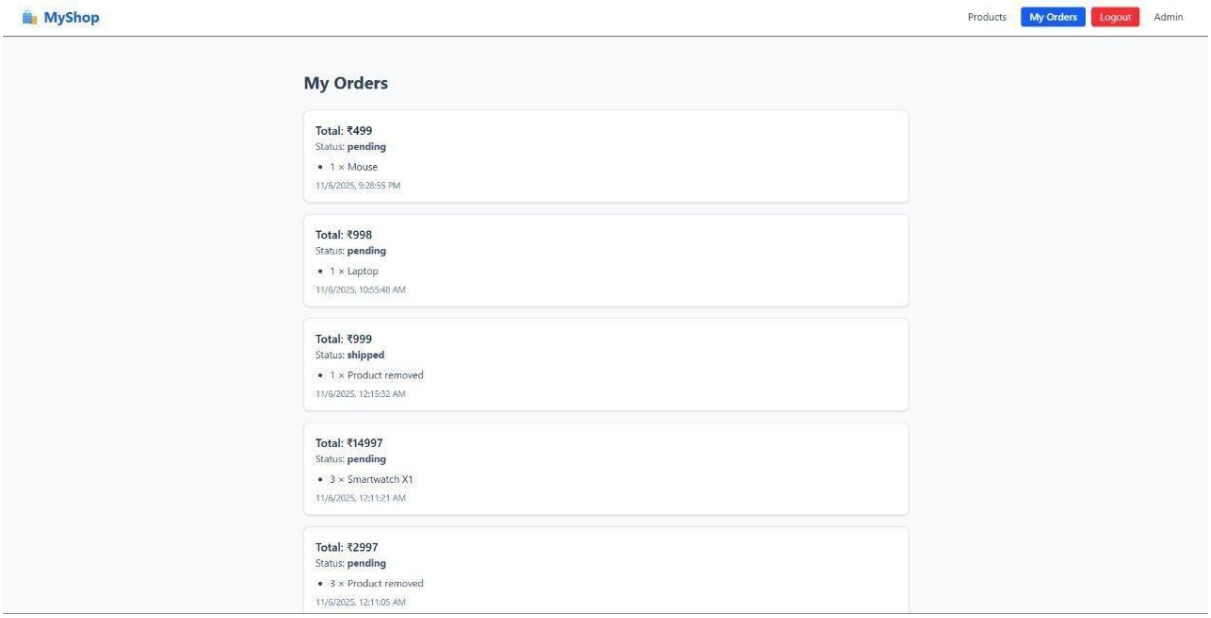
A modal or dedicated page for user authentication. It includes forms for users to log in with their credentials or register a new account.

Note: Imagine a screenshot with email and password fields



6.3 User Profile Page

Authenticated users can view their profile information and order history on this page.*Note: Imagine a screenshot showing a user's past orders in a list.*



6.4 Admin Dashboard

A restricted view for admin users to manage the platform. This includes functionalities to Create, Read, Update, and Delete (CRUD) products and categories, and to view and manage all customer orders.

Note: Imagine a screenshot with a table of products and "Edit"/"Delete" buttons.

7.1 GraphQL Query in Apollo Studio

The image below shows a successful query executed in Apollo Studio to fetch a list of products, including their names, prices, and category information. This demonstrates the unified schema in action.

Note: Imagine a screenshot of a GraphQL playground with a query on the left and a JSON response on the right.

7.2 Mutation for Placing an Order

This output shows the result of a placeOrder mutation. The response confirms the order creation, returns the order ID, status, and the total amount.

Note: Imagine a screenshot similar to the previous one, but with a mutation and a success response.

7.3 Frontend Application in Browser

This screen report shows the fully functional React application running in a web browser, displaying the product catalog with applied filters.

[Insert Screenshot of the entire browser window showing the running app] Note: Imagine a full browser view of your product listing page.

7.4 DataLoader Batching Demonstration

A console log output demonstrating that a single database call is made to fetch multiple users, even when multiple user resolvers are triggered, proving the N+1 problem is solved.

[Insert Screenshot of server logs showing batched SQL/MongoDB queries]
Note: Imagine a terminal

8. Limitation & Future Scope Limitations

1. **Image Handling:** The current implementation uses simple URL strings for product images. It does not include a dedicated file upload service, which is essential for a production environment.
2. **Payment Gateway:** The checkout process is simulated and does not integrate with a real payment gateway like Stripe or Razorpay.

3. **Email Notifications:** The system lacks transactional emails for events like order confirmation, shipping updates, or password resets.
4. **Advanced Search:** The text search relies on basic MongoDB string matching. A dedicated search engine like Elasticsearch would provide more powerful and relevant results.
5. **UI/UX:** While functional, the frontend design can be further polished with more animations, a better mobile experience, and enhanced accessibility features.

Future Scope

1. **Microservices Architecture:** The monolithic backend could be decomposed into microservices (e.g., User Service, Product Catalog Service, Order Service) for better scalability and maintainability.
2. **Real-time Features:** Implement real-time notifications for users (e.g., order status updates) and admins (e.g., new orders) using WebSockets or GraphQL Subscriptions.
3. **Recommendation Engine:** Integrate a machine learning-based recommendation system to suggest products to users based on their browsing and purchase history.
4. **Inventory Management:** Develop a more sophisticated inventory management system with low-stock alerts and purchase order management.
5. **Multi-vendor Support:** Extend the platform to support multiple vendors who can manage their own products and orders.
6. **PWA (Progressive Web App):** Convert the React frontend into a PWA to enable offline functionality and push notifications.
7. **Docker & Kubernetes:** Containerize the application using Docker and orchestrate it with Kubernetes for streamlined deployment and scaling.

GitHub URL

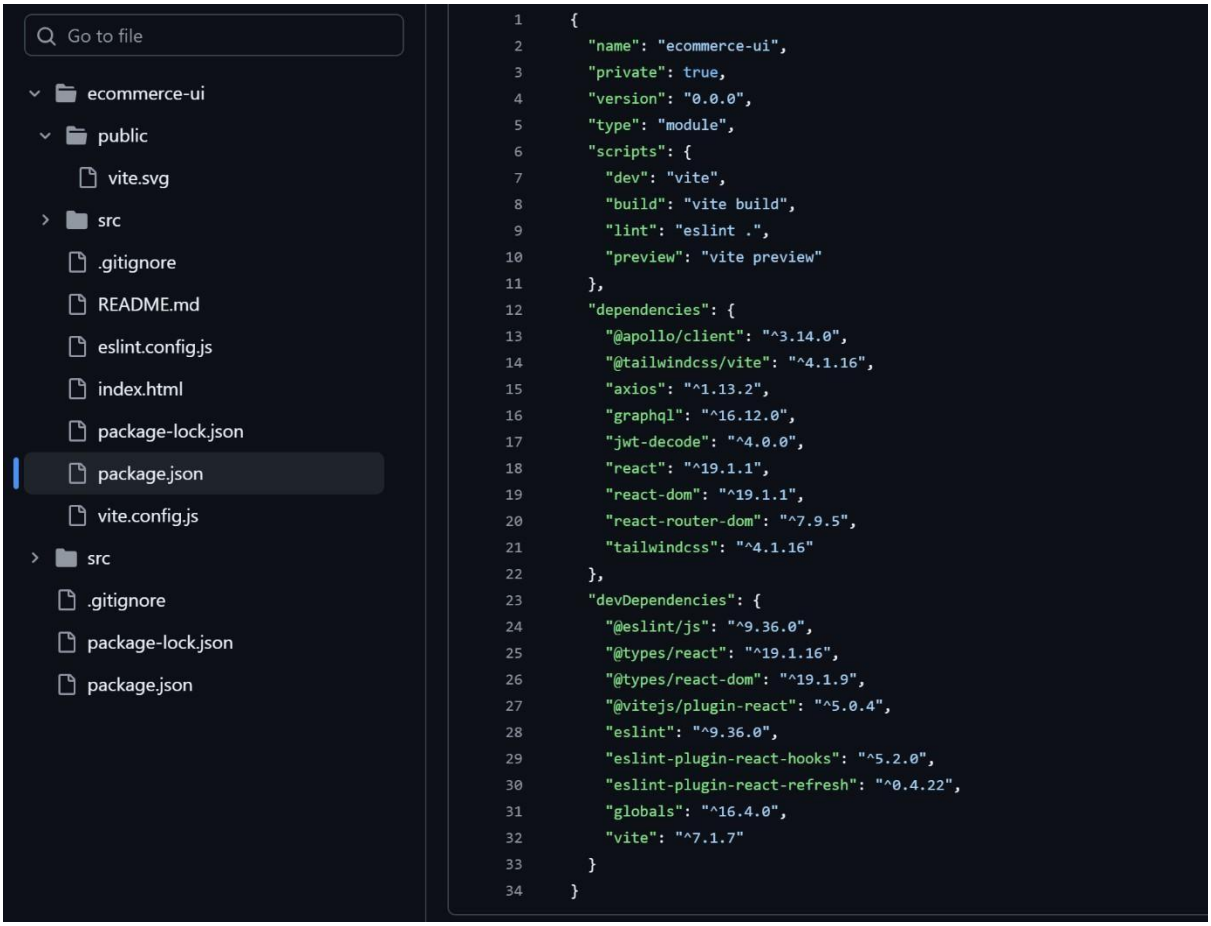
The complete source code for this project, including both the backend GraphQL API and the frontend React application, is available in the public GitHub repository:

https://github.com/imyuvi0/FS-Project.git

Project Code

This section contains the most important and representative code files from the project.

11.1 Backend - Server Initialization (src/server.js**)**
This file sets up the Express server, integrates Apollo Server, and connects to the database.



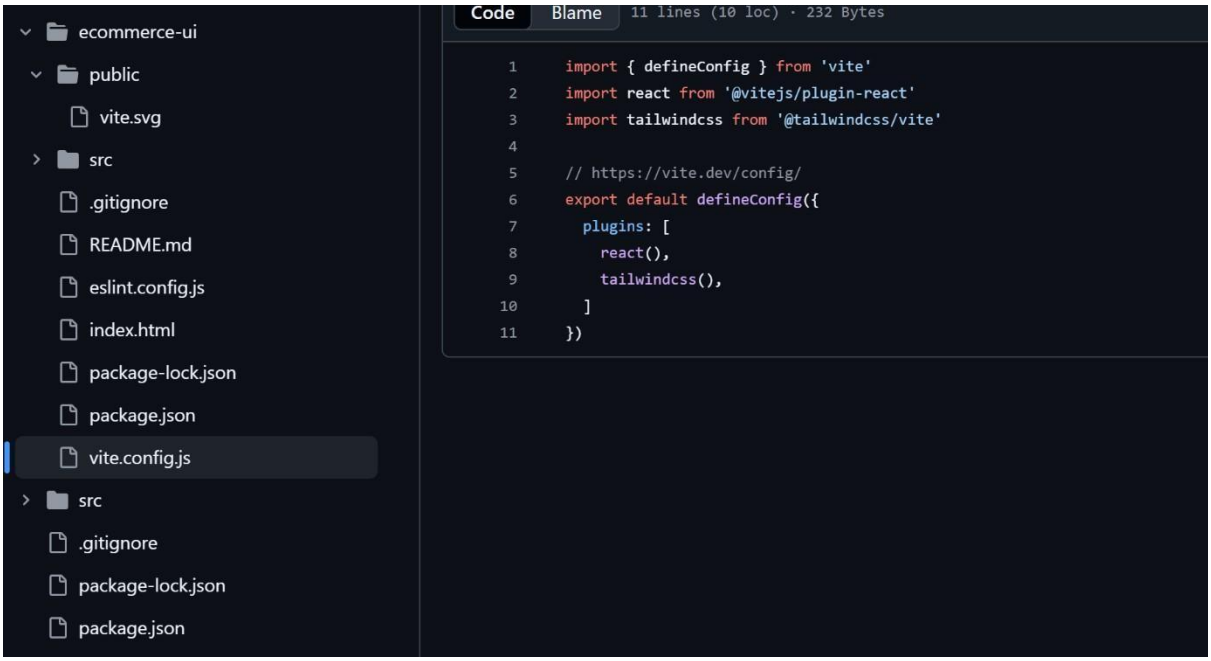
javascript import { ApolloServer } from '@apollo/server'; import { expressMiddleware } from '@apollo/server/express4'; import { ApolloServerPluginDrainHttpServer } from '@apollo/server/plugin/drainHttpServer'; import express from 'express'; import http from 'http'; import cors from 'cors'; import bodyParser from 'body-parser'; import mongoose from 'mongoose'; import 'dotenv/config';

// Import GraphQL Schema and Resolvers import typeDefs from './schema/index.js'; import resolvers from './resolvers/index.js';

// Database connection mongoose.connect(process.env.MONGODB_URI) .then(() => console.log(' Connected to MongoDB')) .catch(err => console.error(' MongoDB connection error:', err));

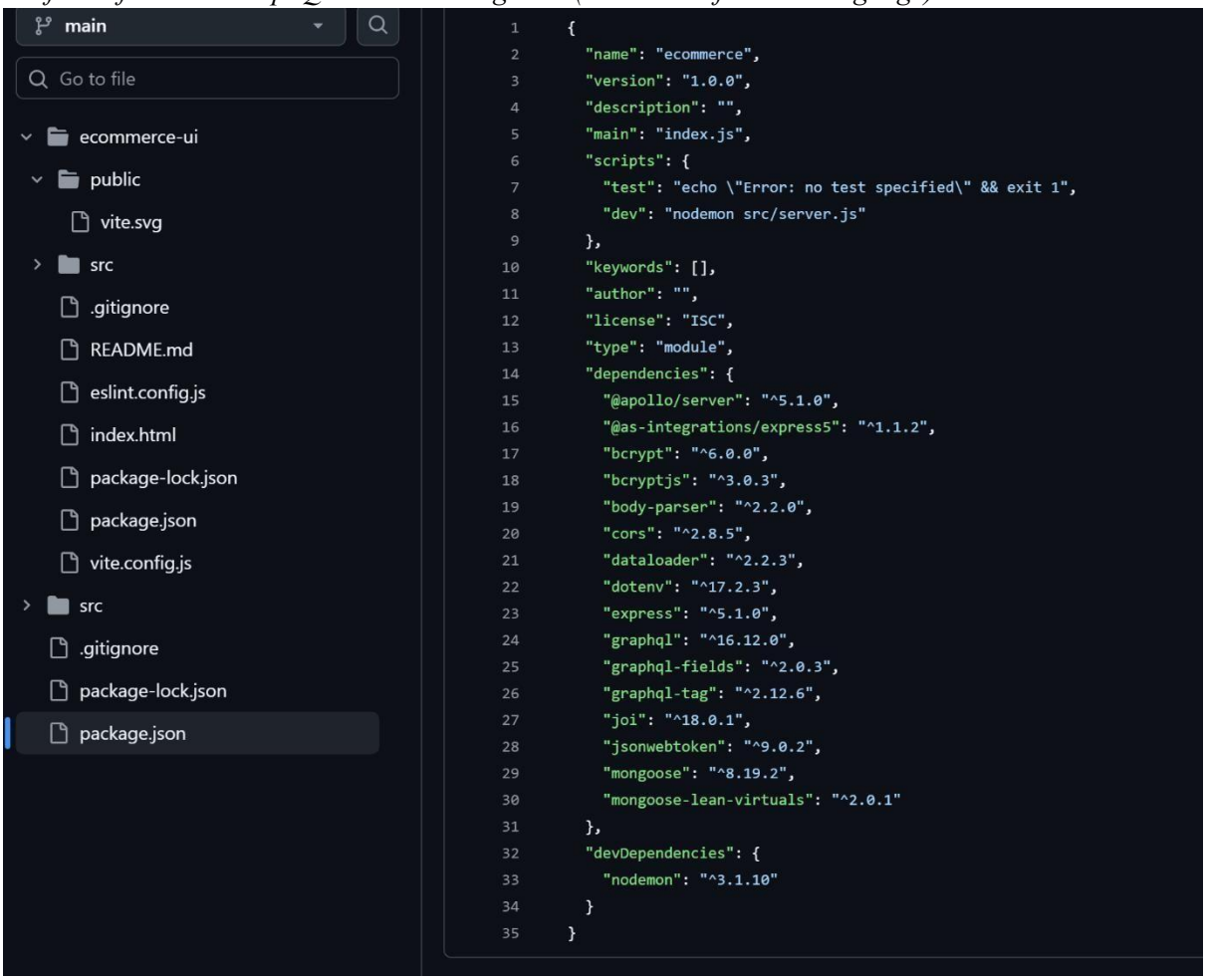
const app = express(); const httpServer = http.createServer(app);

```
// Apollo Server setup const server = new ApolloServer({
typeDefs,   resolvers,   plugins:
[ApolloServerPluginDrainHttpServer({ httpServer })],
formatError: (formattedError, error) => {     // Log the
error for server-side inspection     console.error(error);
// Return a generic error message to the client     return {
message: formattedError.message,       code:
formattedError.extensions?.code || 'INTERNAL_ERROR',
};
},
});

await server.start();

// Apply Apollo GraphQL middleware and other necessary middleware app.use(
'/graphql',  cors(),
bodyParser.json(),
expressMiddleware(server, {
context: async ({ req }) => {
// Get the user token from the headers.       const
token = req.headers.authorization || '';       // Try to
retrieve a user with the token       const
user = await getUserFromToken(token);
// Add the user to the context       return {
user };
},
}),
);

await new Promise((resolve) => httpServer.listen({ port: 4000 }, resolve)); console.log(`
Server ready at http://localhost:4000/graphql`);
```



11.2 GraphQL Schema (src/schema/typeDefs.js**)**

This file defines the GraphQL Schema using SDL (Schema Definition Language).



```
graphql type Query {
# Product Queries   products(
filter: ProductFilterInput
sort: ProductSortInput
pagination: PaginationInput
): ProductConnection!
product(id: ID!): Product

# Category Queries   categories:
[Category!]!   category(id: ID!):
Category # Order Queries (Auth
Required) orders: [Order!]!
@auth(requires:
AUTHENTICATED) order(id:
ID!): Order @auth(requires:
AUTHENTICATED)

# User Query (Admin Only)   users:
[User!]! @auth(requires: ADMIN)
}

type Mutation {   # Auth Mutations
signUp(userInput: UserInput!): AuthPayload!
login(credentials: LoginInput!): AuthPayload!

# Cart & Order Mutations (Auth Required)   addToCart(productId: ID!, quantity: Int!):
CartItem! @auth(requires: AUTHENTICATED)   removeFromCart(cartItemId: ID!):
Boolean! @auth(requires: AUTHENTICATED)   placeOrder: Order! @auth(requires:
AUTHENTICATED)

# Product & Category Mutations (Admin Only)   createProduct(productInput:
ProductInput!): Product! @auth(requires: ADMIN)   updateProduct(id: ID!, productInput:
ProductInput!): Product! @auth(requires: ADMIN)   deleteProduct(id: ID!): Boolean!
```

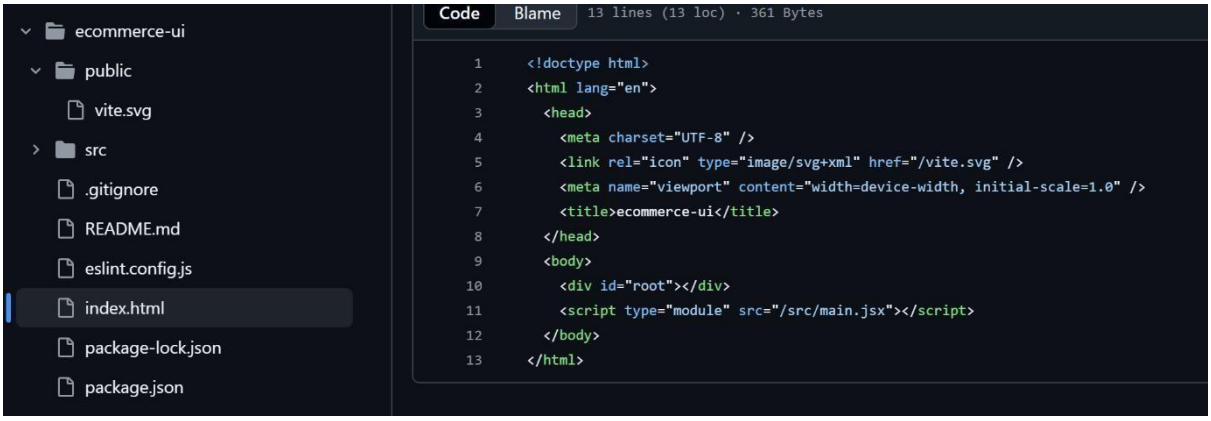
@auth(requires: ADMIN) createCategory(categoryInput: CategoryInput!): Category!

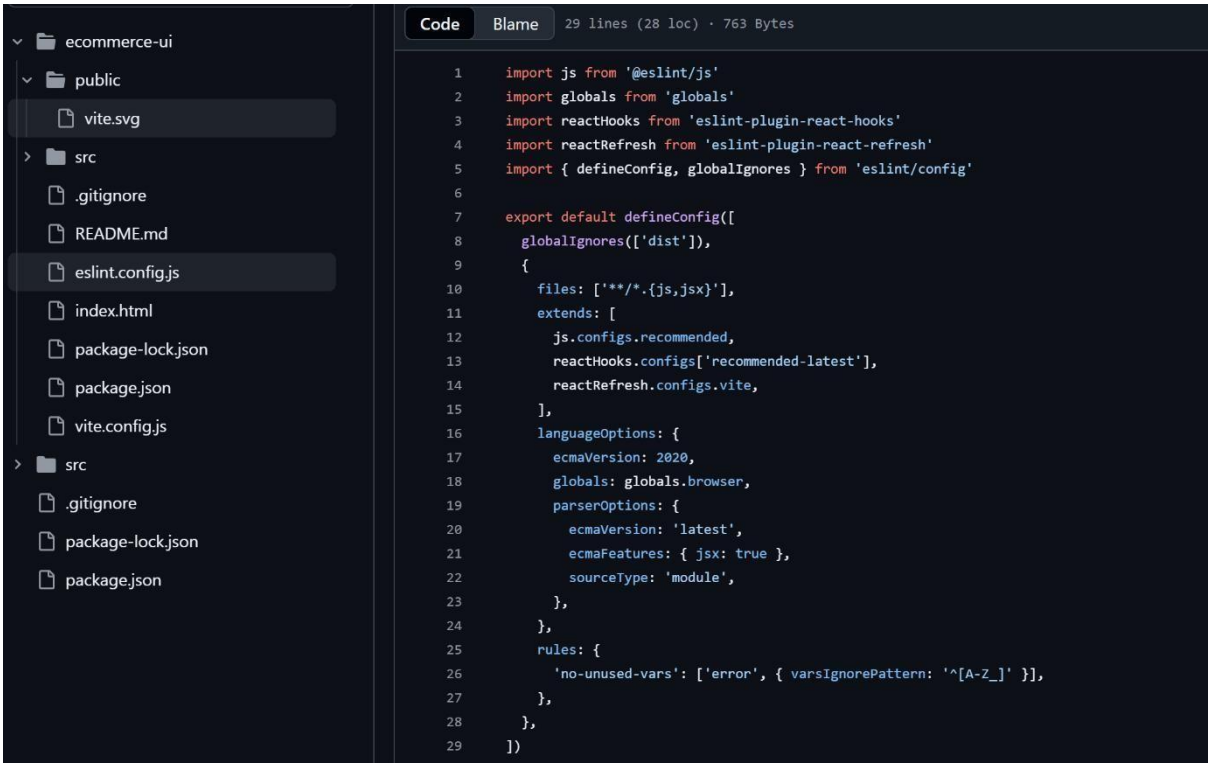
@auth(requires: ADMIN)

}

11.2 GraphQL Schema (Continued...) 11.3 Resolvers - Product Queries
(src/resolvers/productResolvers.js**)**

This file contains the resolvers for product-related queries, showcasing filtering, sorting, and pagination.





```
javascript  import Product from '../../models/Product.js';  import Category
from '../../models/Category.js'; import { buildProductFilter, buildProductSort
} from'../utils/queryHelpers.js'; export const productResolvers = { Query: {
products: async (_, { filter, sort, pagination = {} }) => {       const { limit =
10, offset = 0 } = pagination;

// Build MongoDB filter object       const mongoFilter =
buildProductFilter(filter);

// Build MongoDB sort object       const sortOptions =
buildProductSort(sort);

// Execute query with filtering, sorting, and pagination       const
products = await Product.find(mongoFilter)
.populate('category')
.sort(sortOptions)
.limit(limit)
.skip(offset)
.lean();

// Get total count for pagination info       const totalCount = await
Product.countDocuments(mongoFilter);
return {
nodes: products,         totalCount,
};
},
```

product: async (_, { id }) => { const product = await Product.findById(id).populate('category');

if (!product) { throw new

Error('Product not found');

}

return product;

},

},

// ... Other product-related resolvers (Mutation for create, update, delete)

};

11.4 Resolvers - Order Mutations (src/resolvers/orderResolvers.js**)**
This file contains the resolvers for placing an order, demonstrating authorization and business logic.

javascript import { UserInputError, ForbiddenError } from 'apolloserver-

express'; import Order from '../../models/Order.js'; import

Product from '../../models/Product.js'; import CartItem from

'../../models/CartItem.js';

export const orderResolvers = {

Mutation: { placeOrder: async (_, __,

context) => {

// 1. Check Authentication if (!context.user) { throw new

ForbiddenError('You must be logged in to place an order.');

}

const userId = context.user.id;

// 2. Fetch user's cart items const cartItems = await CartItem.find({ user:

userId }).populate('product');

```
if (cartItems.length === 0) {          throw new
UserInputError('Your cart is empty.');
}

let totalAmount = 0;       const
orderItems = [];

// 3. Process each cart item       for (const
cartItem of cartItems) {
const product = cartItem.product;

// Check stock availability          if (product.stockQuantity < cartItem.quantity) {
throw new UserInputError(`Insufficient stock for product: ${product.name}`);
}

// Calculate line total and update total amount          const lineTotal =
cartItem.quantity * parseFloat(product.price);          totalAmount +=
lineTotal;

// Prepare order item          orderItems.push({          product: product._id,
quantity: cartItem.quantity,          unitPrice:
product.price, // Snapshot the price at order time
});

// Decrement the product stock          product.stockQuantity -=
cartItem.quantity;          product.inStock = product.stockQuantity
> 0;          await product.save();
}

// 4. Create the order       const
order = new Order({
user: userId,
```

items: orderItems,

totalAmount, status:

'pending',

});

const savedOrder = await order.save();

// 5. Clear the user's cart after successful order placement await

CartItem.deleteMany({ user: userId });

// 6. Return the created order return await

Order.findById(savedOrder._id).populate('user').populate('items.product');

},

},

Query: {

// ... Resolvers for fetching orders

}

};

11.5 DataLoader Implementation (src/dataLoaders/userLoader.js**)** *This file creates a DataLoader instance for batching and caching User requests.*

javascript import DataLoader from 'dataloader'; import User from

'../../models/User.js';

// Batch loading function const batchUsers = async (userIds)

=> { console.log('Batch loading users for

IDs:', userIds);

```
// Fetch all users in a single query   const users = await
User.find({ _id: { $in: userIds } });

// Map results back to the original order of keys (userIds)   const
userMap = {};   users.forEach(user => {
userMap[user._id.toString()] = user;
});

return userIds.map(id => userMap[id] || null); // Return null if user not found
};

// Create the DataLoader instance export const createUserLoader =
() => new DataLoader(batchUsers);
<div style="page-break-after: always;"></div>
```

11.6 Authentication Middleware (src/utils/auth.js**)**
This utility function is used to validate JWT tokens and fetch the user for the GraphQL context.

```
javascript import jwt from 'jsonwebtoken';  import
User from '../../models/User.js';

export const getUserFromToken = async (token) => {
try {
// Extract token from "Bearer <token>" format     const actualToken = token.startsWith('Bearer ')
? token.slice(7) : token;

if (!actualToken) {       return
null;
}

// Verify the JWT token     const decoded = jwt.verify(actualToken,
process.env.JWT_SECRET);
```

// Find and return the user, excluding the password field const user =
await User.findById(decoded.userId).select('-password'); return user;
} catch (error) {
// Token is invalid or expired console.error('Token verification
failed:', error.message); return null;
}
};

11.7 Frontend - Apollo Client Setup (src/main.jsx**)**
This file in the React frontend initializes the Apollo Client to communicate with the GraphQL API. javascript import React from 'react' import ReactDOM from 'react-dom/client' import {
ApolloClient, InMemoryCache,
ApolloProvider, createHttpLink } from '@apollo/client'; import { setContext } from
'@apollo/client/link/context'; import App from './App.jsx' import './index.css'

// Create an HTTP link to the GraphQL endpoint
const httpLink = createHttpLink({ uri:
'http://localhost:4000/graphql',
});

// Create an auth link to set the Authorization header const authLink
= setContext((_, { headers }) => {
// get the authentication token from local storage if it exists const
token = localStorage.getItem('authToken');
// return the headers to the context so httpLink can read them
return { headers: { ...headers, authorization: token ?
`Bearer ${token}` : "",
}
}
});

// Initialize Apollo Client const

client = new ApolloClient({

link: authLink.concat(httpLink),

cache: new InMemoryCache()

});

ReactDOM.createRoot(document.getElementById('root')).render(

<React.StrictMode>

<ApolloProvider client={client}>

<App />

</ApolloProvider>

</React.StrictMode>,

)

11.8 Frontend - Product List Component (src/components/ProductList.jsx**)** *This React component fetches and displays a list of products using a GraphQL query.*

javascript import { useQuery } from

'@apollo/client'; import { GET_PRODUCTS } from

'../graphql/queries'; const ProductList = ({

selectedCategory, sortBy }) => { // Use the useQuery

hook to fetch products const { loading, error, data } =

useQuery(GET_PRODUCTS, { variables: {

filter: selectedCategory ? { categoryId:

selectedCategory } : {}, sort: sortBy, pagination: { limit: 12

}

},

// Optional: Poll the server every 30 seconds for new data

// pollInterval: 30000,

});

if (loading) return Loading products... ; if (error) return

Error: {error.message} ;

```
const products = data?.products?.nodes || [];


return (
<div className="grid grid-cols-1 sm:grid-cols-2 md:grid-cols-3 lg:grid-cols-4 gap-6">
{products.map((product) => (
<div key={product.id} className="bg-white rounded-lg shadow-md overflow-hidden hover:shadow-lg transition-shadow">
<img src={product.image || '/placeholder-
image.jpg'} alt={product.name}
className="w-full h-48 object-cover"
/>
<div className="p-4">
<h3 className="font-semibold text-lg mb-1">{product.name}</h3>
<p className="text-gray-600 text-sm mb-2 line-clamp-2">{product.description}</p> <div className="flex justify-between items-center">
<span className="font-bold text-indigo-600
```

Bibliography

Banks & Birt – GraphQL in Action (Manning)

Bshouty & Hage – Full-Stack GraphQL Applications (Manning)

Giroux – Production-Ready GraphQL (Apollo Graph)

Haverbeke – Eloquent JavaScript

Richardson & Ruby – RESTful Web Services (O'Reilly)

REFERENCES

1. Apollo Server Documentation – https://www.apollographql.com/docs/apollo-server/
2. GraphQL Official Website – https://graphql.org/
3. Node.js Documentation – https://nodejs.org/
4. Express.js Official Website – https://expressjs.com/
5. MongoDB Documentation – https://www.mongodb.com/docs