GraphQL API for E-Commerce

A PROJECT REPORT

Submitted by

OMPRAKASH REDDY-23BAI70296

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE & ENGINEERING

Chandigarh University

NOVEMBER 2025

GraphQL API for E-Commerce

Project Description

This project presents a **GraphQL API for an E-Commerce Platform**, designed and implemented using **Node.js (Express)**, **Apollo Server**, and **MongoDB**. The system offers a **unified, flexible, and high-performance API layer** for managing products, categories, orders, and users — providing a foundation for modern, scalable e-commerce applications.

Unlike traditional REST APIs, this GraphQL implementation allows clients to request **exactly the data they need**, reducing over-fetching and under-fetching while improving response performance. The project emphasizes **secure data access**, **efficient query resolution**, and **field-level control** through GraphQL's powerful schema and resolver architecture.

Objectives:

- Provide a unified API for Products, Categories, Orders, and Users.
- Support precise and efficient querying using GraphQL's type system.
- Implement **pagination, sorting, and filtering** for large product catalogs.
- Ensure **secure authentication and role-based authorization** using JWT.

Core Features and Implementation

The GraphQL API for E-Commerce is engineered with a focus on **data efficiency**, **secure access**, and **developer-friendly schema design**. The architecture ensures predictable performance even on large datasets by combining **GraphQL's field selection** with MongoDB's lean queries.

---

1. Unified GraphQL Schema

- The schema defines core entities such as Product, Category, Order, and User, along with input types for filtering, pagination, and mutations.
- Custom scalar types (DateTime, Decimal) are implemented for precision in timestamps and prices.
- The schema acts as a single source of truth, exposing relationships through nested resolvers (e.g., a category's products or a user's orders).

---

2. Data Access and Performance Optimization

- The backend integrates **DataLoader** for batching and caching queries to prevent the N+1 problem.
- Queries are executed using **lean MongoDB projections**, ensuring only requested fields are fetched.
- Compound indexes (e.g., categoryId, price, createdAt) enhance performance for filtering and sorting operations.

---

3. Authentication and Authorization

- **JWT-based authentication** is implemented in middleware to validate tokens for each request.
- The token stores both the **user ID** and **role** (admin or customer).
- Role-based checks are enforced in resolvers — only admins can modify product data or update order statuses.

---

4. Pagination, Filtering, and Sorting

- Supports both **cursor-based (Relay style)** and **offset/limit** pagination.
- Provides dynamic filtering on fields such as price range, category, inStock, and rating.
- Sorting options include price, createdAt, and popularity, supporting both ascending and descending orders.

---

5. Error Handling and Input Validation

- Structured error extensions are used (e.g., GRAPHQL_VALIDATION_FAILED, UNAUTHENTICATED) for consistent error reporting.

- Input validation ensures only correct data types and value ranges are accepted for mutations.

- All exceptions are logged and returned with descriptive paths and codes for debugging and schema transparency.

---

6. Caching and Field-Level Query Optimization

- GraphQL's @cacheControl directive is used to mark public fields as cacheable.
- The API dynamically adjusts MongoDB projections based on GraphQL's resolveInfo object to fetch only requested fields.

- This significantly reduces payload size and response time for large data sets.

---

7. Observability and Logging

- Each request is tagged with a unique request ID and logged with timing data for observability.

- Slow queries are flagged and logged for optimization insights.

- Metrics such as cache hit rate and resolver latency are monitored for performance tracking.

---

8. Mutations and Admin Operations

- Admins can perform **CRUD operations** on products and categories.

- Customers can use **addToCart**, **placeOrder**, and **updateOrder** mutations.

- Input types include validation logic to ensure product stock and pricing constraints.

---

9. Development and Deployment

- The project uses environment-driven configuration for local and production setups.
- Docker Compose is used to orchestrate the API and MongoDB containers for consistent development environments.
- Seed scripts automatically populate the database with sample products, categories, and users for quick testing.
- **Summary**

The **GraphQL API for E-Commerce** was designed to deliver a modern, secure, and efficient backend solution for online retail systems. The unified GraphQL schema enables flexible data querying, while DataLoader and lean MongoDB projections ensure scalable performance.

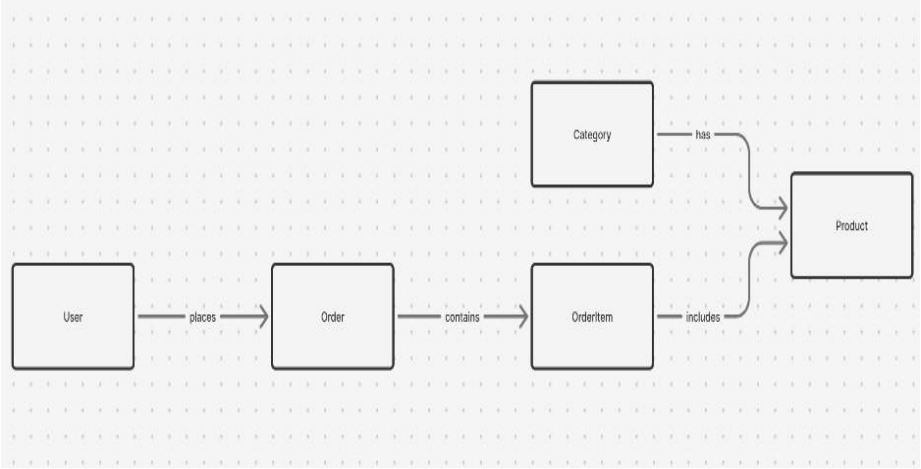
JWT-based authentication provides secure role-specific access, and robust pagination, filtering, and sorting features enhance usability. The system's architecture ensures minimal data redundancy, predictable performance, and strong documentation support through Apollo Playground.

This API provides a foundation for extending to frontend applications, real-time updates via subscriptions, and future integration with payment gateways and analytics modules.

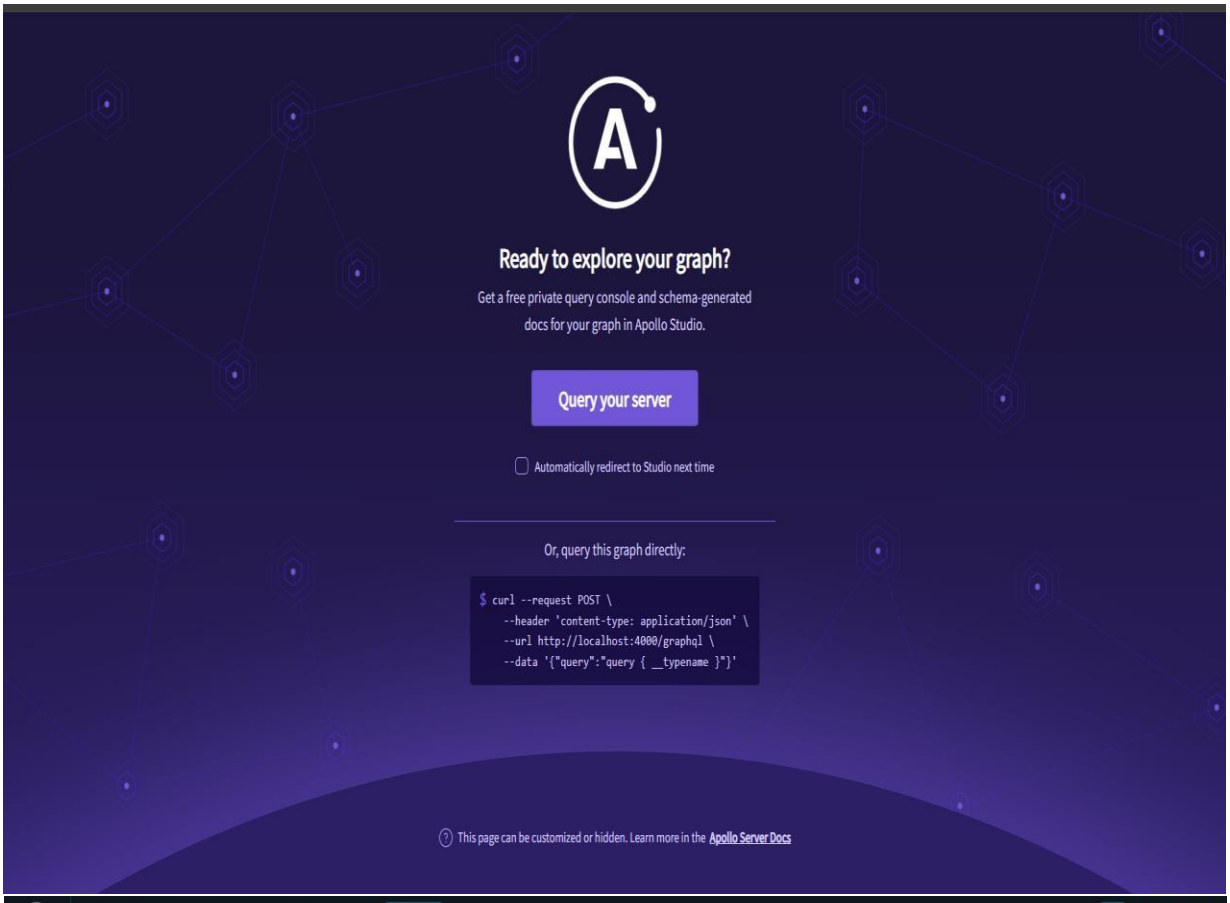
Hardware/Software Requirements Hardware:

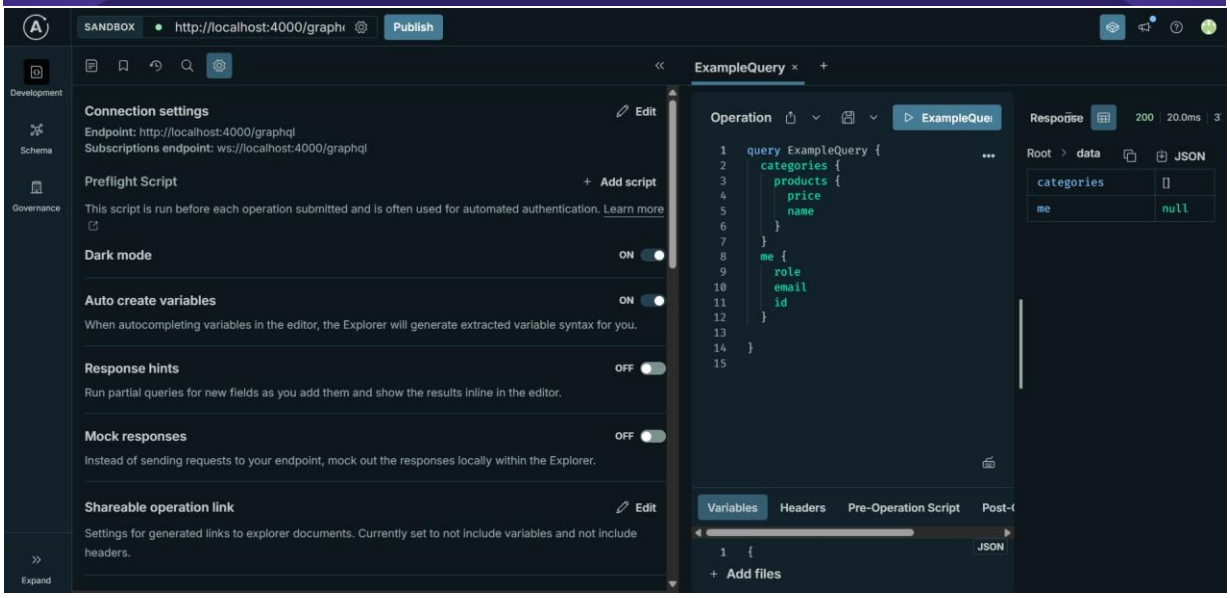
Category	Specification
Processor	Intel Core i5 or higher
RAM	Minimum 8 GB
Storage	250 GB HDD / SSD
Operating System	Windows 10 / 11 or Ubuntu 22.04 LTS
Backend Framework	Node.js (Express) with Apollo Server
Database	MongoDB (Local or Atlas)
Authentication	JSON Web Token (JWT)
Development Tools	VS Code, Postman, GraphQL Playground

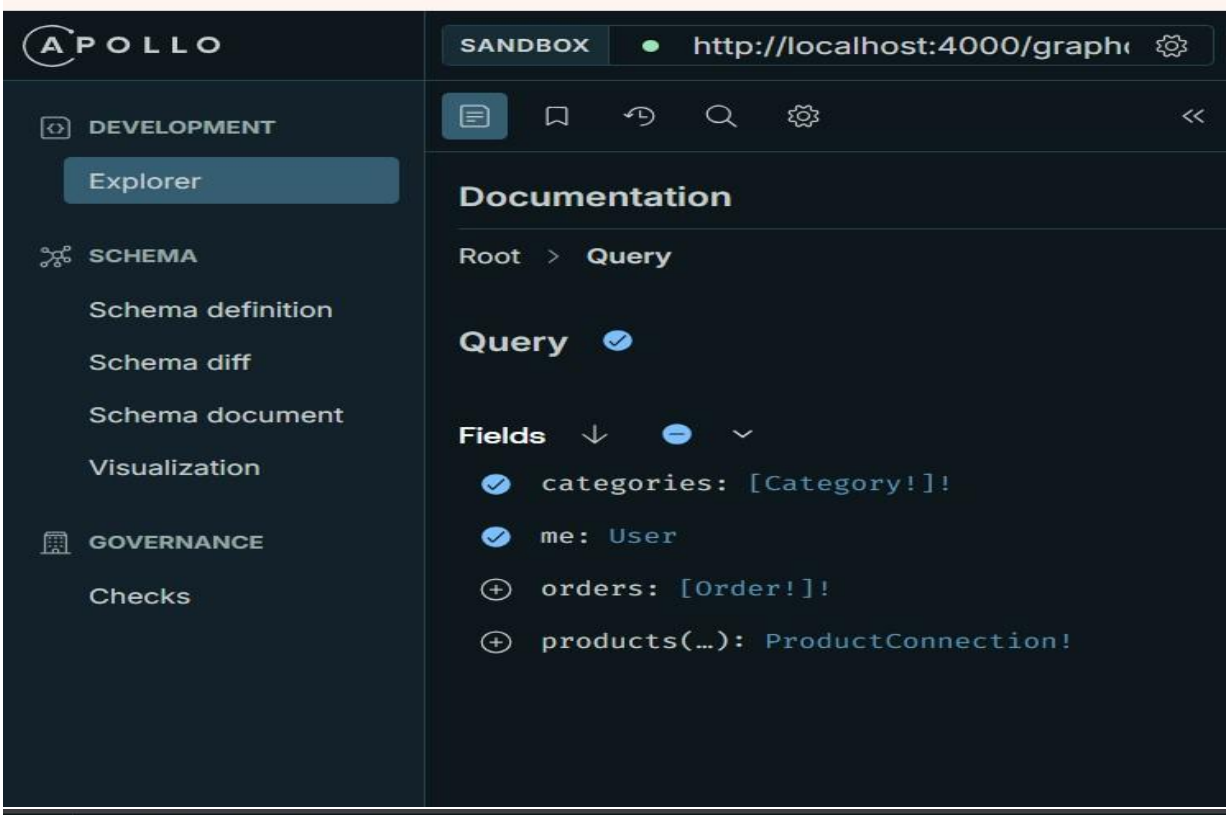
ER Diagram

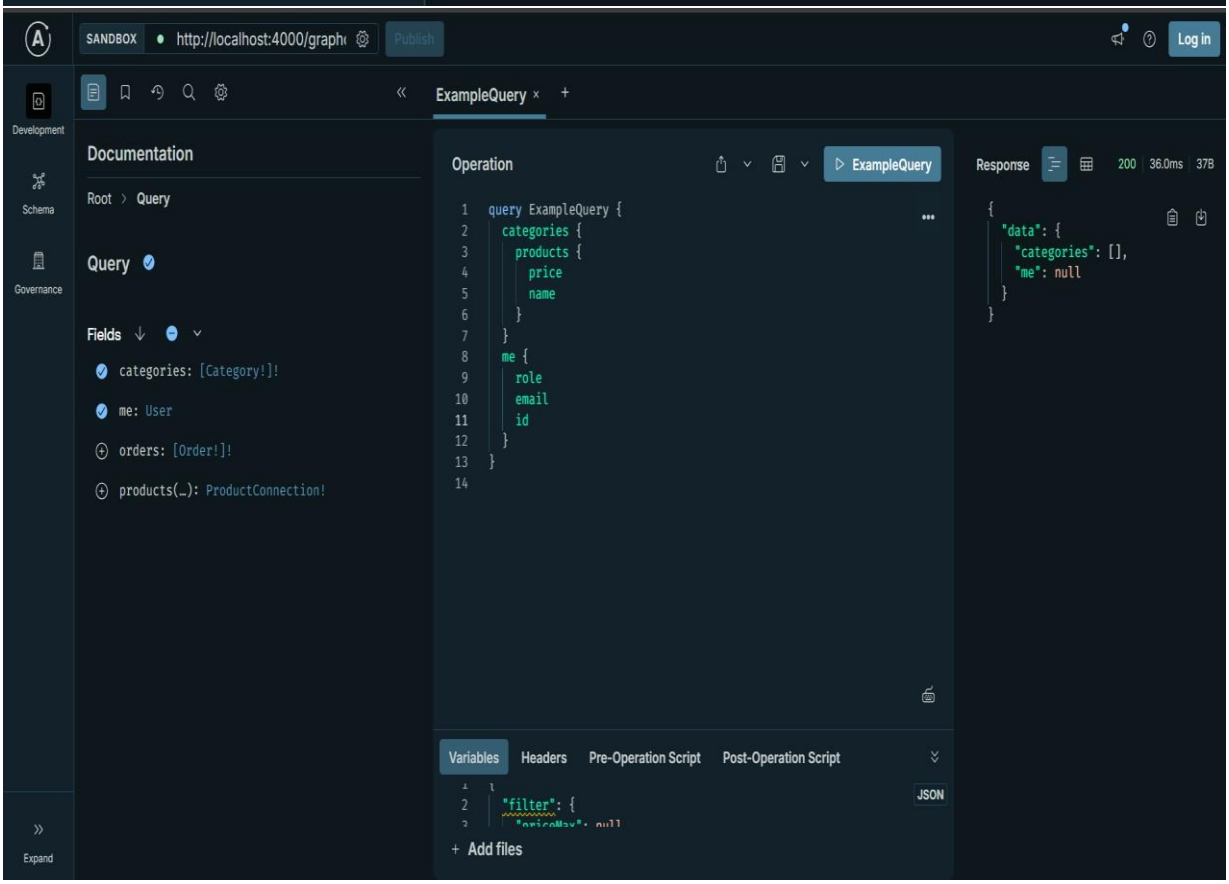

Category
has
Product
User
places
Order
contains
OrderItem
includes

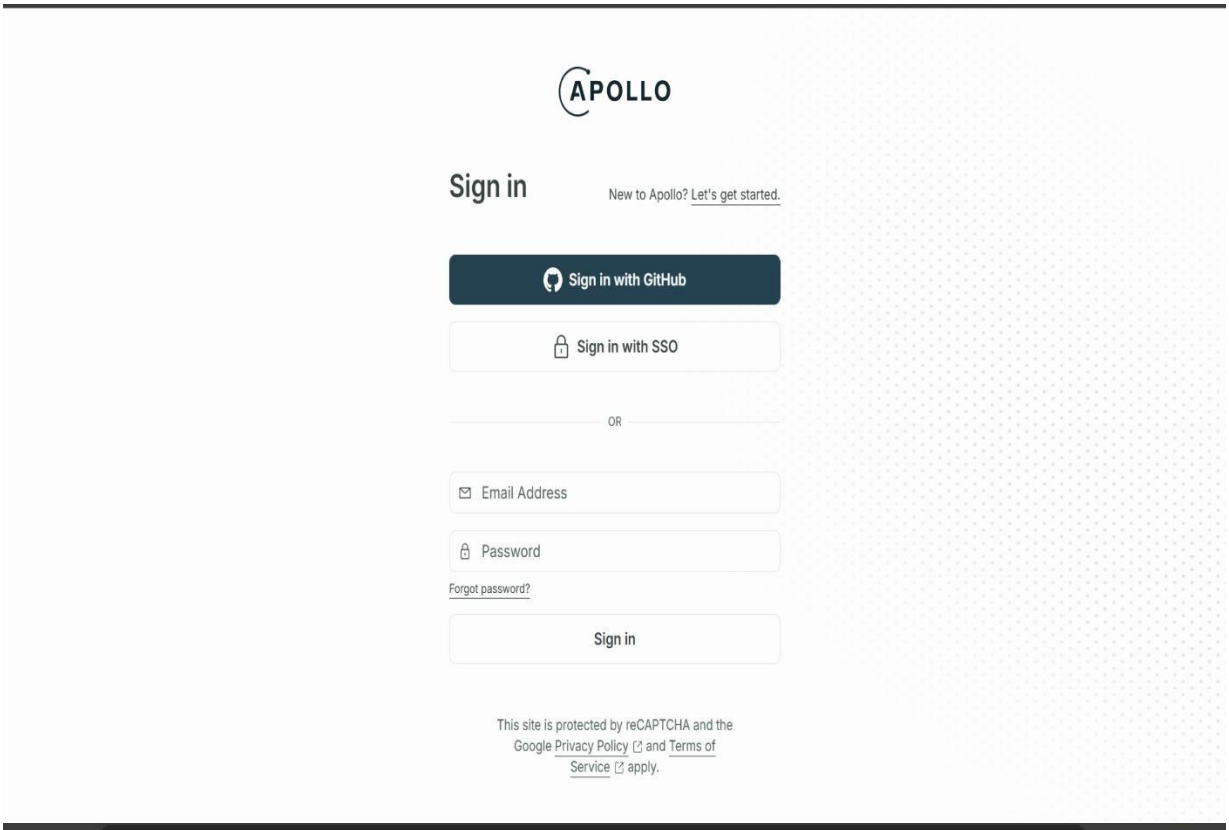
Front-End Screens

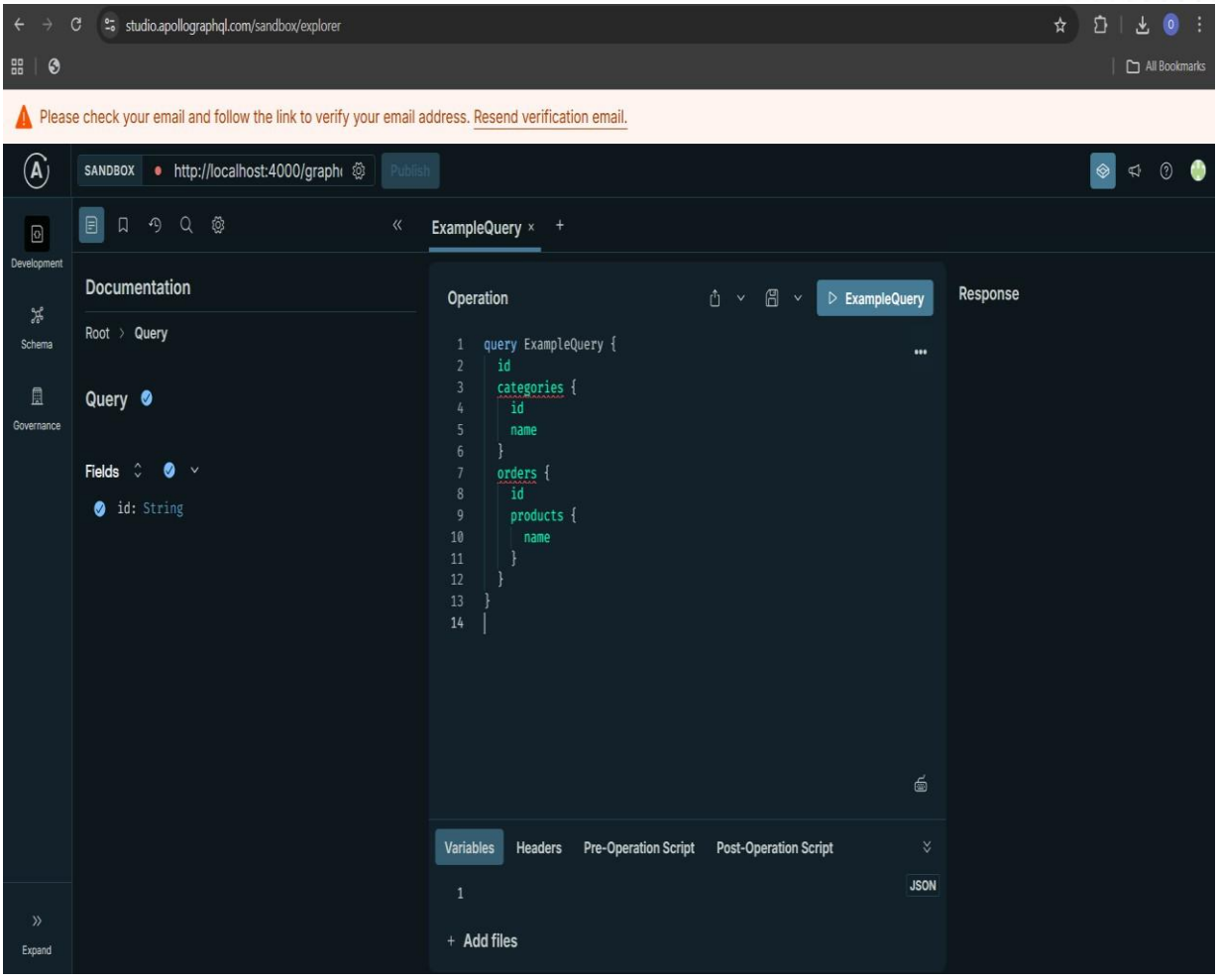


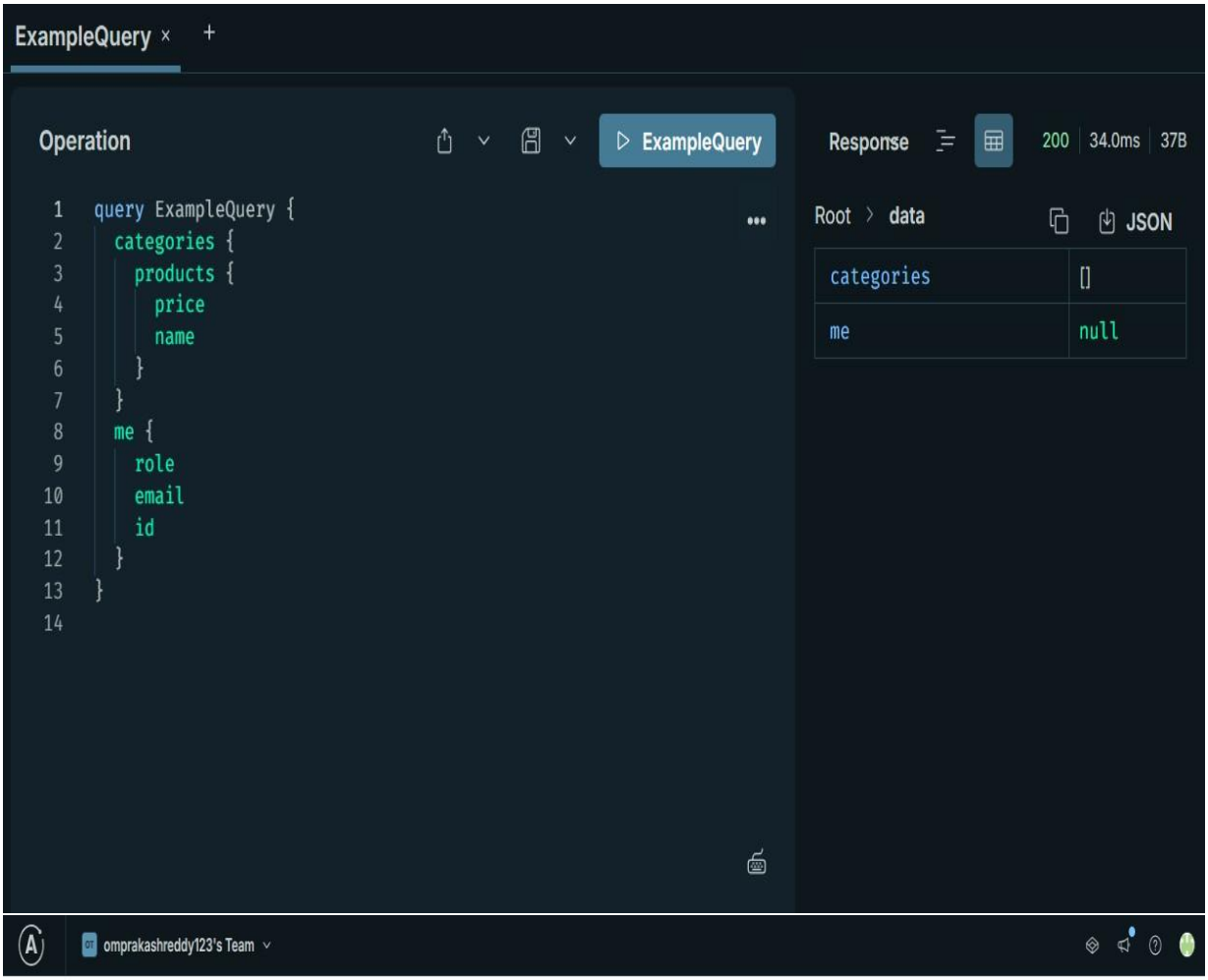


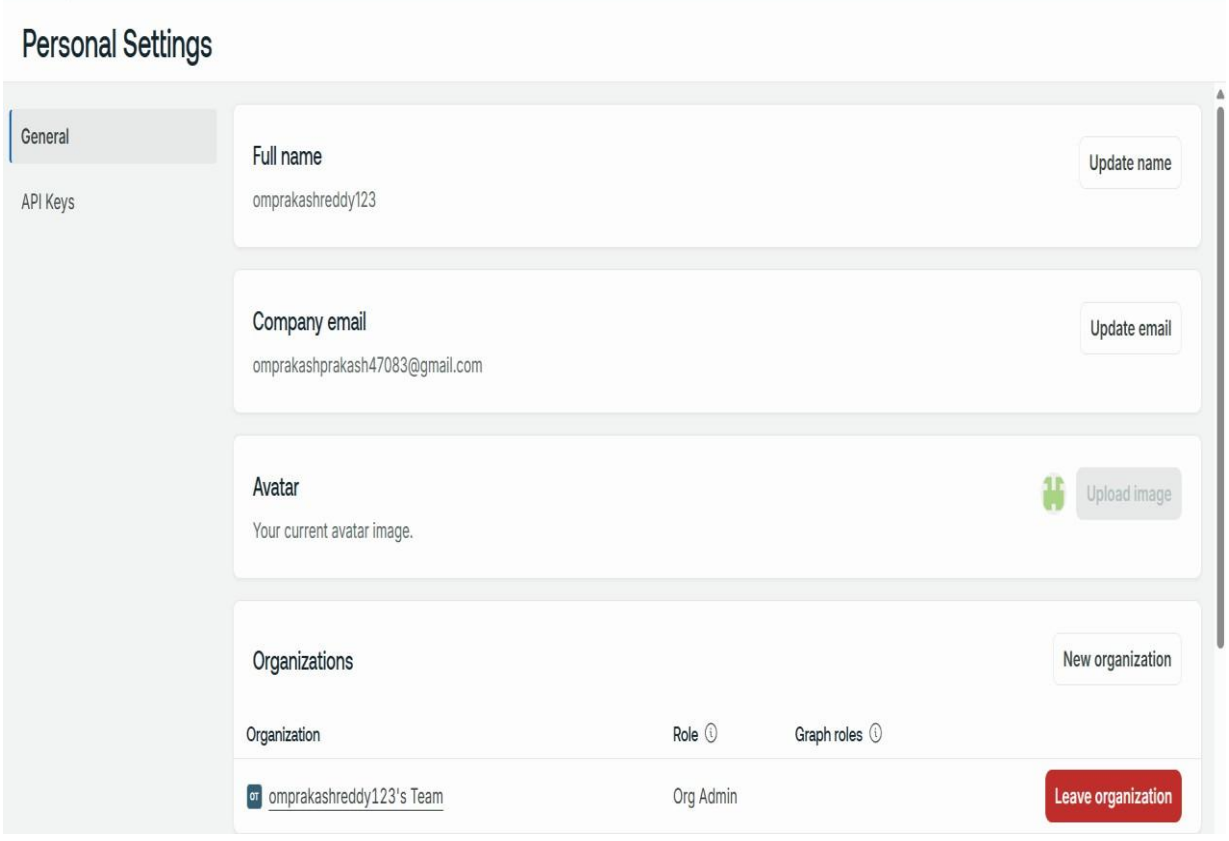

APOLLO
SANDBOX
http://localhost:4000/graph
DEVELOPMENT
Explorer
SCHEMA
Schema definition
Schema diff
Schema document
Visualization
GOVERNANCE
Checks
Documentation
Root > Query
Query
Fields
categories: [Category!]!
me: User
orders: [Order!]!
products(…): ProductConnection!


SANDBOX
http://localhost:4000/graph
Publish
Log in
Development
Schema
Governance
Expand
ExampleQuery
Documentation
Root › Query
Query
Fields
categories: [Category!]!
me: User
orders: [Order!]!
products(…): ProductConnection!
Operation
ExampleQuery
query ExampleQuery {
categories {
products {
price
name
}
}
me {
role
email
id
}
}
Response
200
36.0ms
37B
{
"data": {
"categories": [],
"me": null
}
}
Variables
Headers
Pre-Operation Script
Post-Operation Script
JSON
"filter": {
+ Add files


APOLLO
Sign in
New to Apollo? Let's get started.
Sign in with GitHub
Sign in with SSO
OR
Email Address
Password
Forgot password?
Sign in
This site is protected by reCAPTCHA and the Google Privacy Policy and Terms of Service apply.


studio.apollographql.com/sandbox/explorer
All Bookmarks
Please check your email and follow the link to verify your email address. Resend verification email.
SANDBOX
http://localhost:4000/graph
Publish
Development
Schema
Governance
Expand
Documentation
Root › Query
Query
Fields
id: String
ExampleQuery
Operation
ExampleQuery
Response
query ExampleQuery {
id
categories {
id
name
}
orders {
id
products {
name
}
}
}
Variables
Headers
Pre-Operation Script
Post-Operation Script
JSON
1
+ Add files


ExampleQuery
Operation
ExampleQuery
query ExampleQuery {
categories {
products {
price
name
}
}
me {
role
email
id
}
}
Response
200
34.0ms
37B
Root
data
JSON
categories
[]
me
null
omprakashreddy123's Team


Personal Settings
General
API Keys
Full name
omprakashreddy123
Update name
Company email
omprakashprakash47083@gmail.com
Update email
Avatar
Your current avatar image.
Upload image
Organizations
New organization
Organization
Role
Graph roles
omprakashreddy123's Team
Org Admin
Leave organization

Limitations:

1. The current GraphQL API system is optimized for small to mid-sized e-commerce platforms. Scaling beyond 50+ active tenants would require the implementation of distributed database sharding or a multi-cluster architecture for efficient load balancing.
2. Real-time data synchronization across multiple devices is not yet fully implemented, which may lead to minor delays in reflecting updates between clients.
3. The present version provides limited support for external third-party integrations such as CRM, billing, and ERP systems.
4. The system does not currently include predictive analytics or AI-driven personalization features to enhance customer experience.
5. Role-Based Access Control (RBAC) is implemented at a basic level and lacks support for fine-grained permission hierarchies and tenant-specific policy rules.
6. The platform currently relies on a single-region deployment setup, which may introduce latency for users from geographically distant regions and reduce resilience to regional outages.
7. The solution operates entirely online and lacks an offline-first mode, making continuous internet connectivity mandatory.
8. Backup and disaster recovery processes are semi-automated and not yet integrated into the CI/CD deployment pipeline.

Future Scope:

1. Integrating an **AI-powered Recommendation Engine** to provide personalized product suggestions based on user purchase patterns and browsing behavior.
2. Implementing **multi-region deployment** using Kubernetes clusters and load balancers to achieve high scalability, fault tolerance, and low-latency access for global users.
3. Introducing **Payment Gateway integrations** such as Razorpay, Stripe, and PayPal, along with automated invoicing and tax management.
4. Developing an **Advanced Analytics Dashboard** featuring revenue visualization, customer segmentation, churn prediction, and sales forecasting.
5. Migrating to a **Microservices Architecture** to allow independent scaling of core modules like authentication, product catalog, and order management.
6. Enhancing the **RBAC system** with granular permission levels, tenant-based policies, and customizable admin roles.
7. Supporting **Custom Domain Mapping and White-Label Branding** for each ecommerce tenant (e.g., storeName.com) to enhance business identity.
8. Enabling **Third-Party Marketplace Integrations** with platforms like Amazon, Flipkart, and Shopify, and accounting systems such as Zoho Books or Tally.
9. Adding **Single Sign-On (SSO)** support for Google, Microsoft, and GitHub to simplify authentication and onboarding.
10. Implementing **Offline Mode with Progressive Web App (PWA) caching**, allowing users to browse products and manage carts even without active internet connectivity.

GitHub URL https://github.com/vashistsalar/ecommerce-graphql-api

Project Code

File: **auth.js**

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

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

exports.getUserFromToken = async
(token) => {

  if (!token) return null;

  try {

    const payload = jwt.verify(token,
process.env.JWT_SECRET);

    const user = await
User.findById(payload.id);

    return user ? { id: user.id, role:

user.role, email: user.email } : null;

} catch {

    return null;  }

};
```

File: loadder.js const DataLoader =
require('dataloader')const Category =
require('../models/Category');

const batchCategories = async
(categoryIds) => {

  const categories = await
Category.find({ _id: { $in:
categoryIds } });

  return categoryIds.map(id =>
categories.find(c => c.id == id));

};

module.exports = { categoryLoader:
() => new
DataLoader(batchCategories)

}; **File: validation.js:**

const yup =

require('yup');

const productSchema = yup.object({

  name: yup.string().required(),

```
  price:
yup.number().required().positive(),
  category: yup.string().required(),stock:
yup.number().required().min(0),
  description: yup.string().max(500)
});
const registerSchema = yup.object({
  email: yup.string().email().required(),
  password:
yup.string().min(6).required()
});
exports.validateProductInput = input =>
productSchema.validate(input);
exports.validateRegisterInput = input
=> registerSchema.validate(input);
```

File: index.js:

```
const Product = require('./Product');
const Category = require('./Category');
const Order = require('./Order');
```

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



module.exports = {

  Product,

  Category,

  Order,

  User,

};
```

File: order.js:

```
const mongoose = require('mongoose');
 const OrderSchema = new mongoose.Schema({   products: [{ type:
 mongoose.Schema.Types.ObjectId, ref: 'Product' }],   total: { type: Number,
 required: true },   user: { type: mongoose.Schema.Types.ObjectId, ref:
 'User', required: true },   status: { type: String, default: "pending" },
 createdAt: { type: Date, default: Date.now }
 });
 module.exports = mongoose.model('Order', OrderSchema);
```

File: products.js:

```
 const mongoose = require('mongoose'); const ProductSchema = new
 mongoose.Schema({   name: { type: String, required: true },   price: { type: Number,
 required: true },   category: { type: mongoose.Schema.Types.ObjectId, ref:
 'Category', required: true },   stock: { type: Number, default: 0, required: true },
 description: String
 });
 module.exports = mongoose.model('Product', ProductSchema);
```

File: users.js:

```
const mongoose = require('mongoose');
 const UserSchema = new mongoose.Schema({
   email: { type: String, required: true, unique: true },   password: {
 type: String, required: true },   role: { type: String, default:
 'customer', enum: ['customer', 'admin'] }
 });
 module.exports = mongoose.model('User', UserSchema);
```

Conclusion

The **GraphQL API for E-Commerce** project successfully demonstrates the design and implementation of a modern, scalable, and efficient backend architecture for online retail platforms. Built using **Node.js (Express)**, **Apollo Server**, and **MongoDB**, the system provides a unified GraphQL schema that allows seamless access to core entities such as products, categories, orders, and users.

The project emphasizes **performance optimization**, **data security**, and **developer flexibility**. With advanced features like **DataLoader-based caching**, **field-level projections**, and **JWT authentication**, the API delivers both robustness and efficiency while avoiding common pitfalls like the N+1 query problem. Moreover, support for **pagination**, **filtering**, and **sorting** empowers frontend clients (e.g., React applications) to request precise and optimized datasets, improving overall performance and user experience.

From a deployment perspective, the integration of **Docker Compose** ensures consistent development and production environments, while seed scripts facilitate quick testing and data population. The project also adheres to modern software engineering principles such as **modularity**, **observability**, and **role-based access control**, making it adaptable to real-world e-commerce systems.

In summary, this project serves as a strong foundation for building cloud-ready, API-driven ecommerce solutions. Future enhancements — such as **AI-driven recommendations**, **payment gateway integration**, and **multi-region scalability** — can further extend its capabilities and transform it into a comprehensive platform for next-generation online retail ecosystems.

References

1. MongoDB Documentation – https://www.mongodb.com/docs/
2. Express.js Official Guide – https://expressjs.com/
3. React.js Documentation – https://react.dev/
4. Docker Documentation – https://docs.docker.com/
5. NGINX Reverse Proxy Guide – https://www.nginx.com/
6. Vercel Documentation – Deployment & Serverless – https://vercel.com/docs
7. Render Deployment Docs for Node.js – https://docs.render.com/deploy-node-expressapp

8. JWT Authentication Guide – https://jwt.io/introduction
9. CORS Explained – MDN Web Docs – https://developer.mozilla.org/enUS/docs/Web/HTTP/CORS
10. REST API Best Practices – Microsoft Docs – https://learn.microsoft.com/enus/azure/architecture/bestpractices/api-design
11. Twelve-Factor App Methodology for SaaS – https://12factor.net/
12. NPM Package Management Documentation – https://docs.npmjs.com/