Library Management System using GraphQL

Project Overview

The Library Management System is a web application designed to manage aspects of a library. The system supports managing books, authors, libraries, clients, and borrowed books. It leverages GraphQL for efficient data querying and manipulation, allowing for flexible and optimized interactions between the frontend and backend.

Technologies Used

- Frontend
  - React: A JavaScript library for building user interfaces.
  - Apollo Client: A comprehensive state management library for JavaScript that enables you to manage both local and remote data with GraphQL.
- Backend
  - Node.js: A JavaScript runtime built on Chrome's V8 JavaScript engine.
  - Express.js: A minimal and flexible Node.js web application framework.
  - Apollo Server: A community-maintained open-source GraphQL server that works with any GraphQL schema.

Introduction to GraphQL

GraphQL is a query language for an API, and a runtime for executing those queries by using a type system defined for the data. GraphQL provides a complete and understandable description of the data in the API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.

Key Features of GraphQL

- **Declarative Data Fetching:** Clients specify the exact structure of the response they need. This eliminates over-fetching or under-fetching of data, which are common problems in REST APIs.
- **Strongly Typed Schema**: The schema defines the types and relationships in your data model. This provides a clear contract between the client and the server and allows for powerful tools like static analysis and autocomplete in IDEs.
- **Single Endpoint**: GraphQL operates over a single endpoint, simplifying the API structure and reducing the need to manage multiple routes.
- **Real-time Data with Subscriptions**: GraphQL supports subscriptions to allow clients to receive real-time updates, making it suitable for applications that require live data feeds.

GraphQL Schema

The schema is the core of any GraphQL server. It defines the types, queries, mutations, and subscriptions available in the API. Main components of a GraphQL schema:

- Types
    - **Object Types**: Define the shape of the data. For example, Author, Book, Library, Client, and BorrowedBook are object types in our schema.
    - **Scalar Types**: Represent the primitive data types like String, Int, Float, Boolean, and ID.
    - **Enum Types**: Define a set of possible values for a field.

- Queries
    - Queries are used to fetch data from the server. They define what data can be requested by the client. In our schema, we have queries like authors, author, books, book, libraries, library, clients, client, borrowedBooks, and borrowedBook.

- Mutations
    - Mutations are used to modify data on the server. They define what data can be created, updated, or deleted. Our schema includes mutations like addAuthor, addBook, updateAuthor, updateBook, updateLibrary, deleteClient.

- Subscriptions
    - Subscriptions allow clients to listen for real-time updates from the server. While not included in the current schema, they are useful for applications that require live updates, such as notifications or live feeds.

- GraphQL Resolvers
    - Resolvers are functions that handle the execution of a query or mutation. Each field in a GraphQL query corresponds to a resolver function that returns the data for that field. Resolvers can be asynchronous and can fetch data from any source, such as a database, a REST API, or a third-party service.

Backend Implementation

The backend is built using Node.js and Express.js, with Apollo Server handling GraphQL requests. The server defines schemas, resolvers, queries, and mutations.

Frontend Implementation

The Frontend is built using React and Apollo Client. It consists of various components that interact with the GraphQL API to perform CRUD operations.

Example Components:

- AddBook: Form to add a new book to the library.
- BookList: Displays a list of books with options to edit or delete each book.
- ClientManagement: Manages the clients of the library, including adding, updating, and deleting clients.

Setup Instructions

- Clone the repository:

  git clone https://github.com/OlteanBianca/Framework-Design-Server.git

  git clone https://github.com/OlteanBianca/Framework-Design-Client.git

- Start the backend server at http://localhost:4000/graphql:

  node src/index.js

- Start the frontend server at http://localhost:3000:
  npm start

Usage

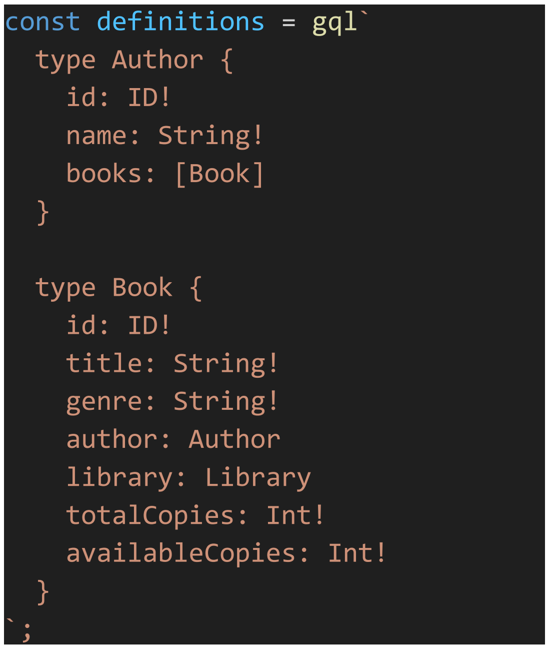
The application provides a user-friendly interface to manage books, authors, libraries, clients, and borrowed books. Users can add, update, and delete records through the web interface, with data being fetched and modified using GraphQL.

The main endpoint for the GraphQL API is “/graphql”, which handles all queries and mutations.

Schema

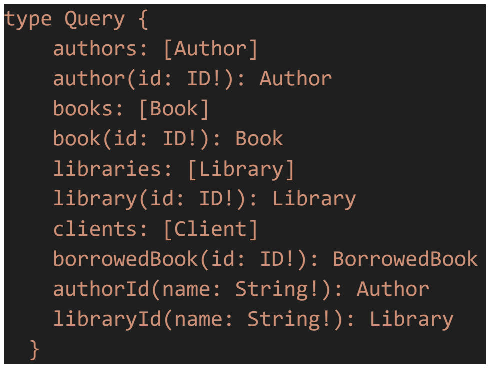
First the definitions of the entities and operations are declared:

- Example of the declaration of the Author and Book:

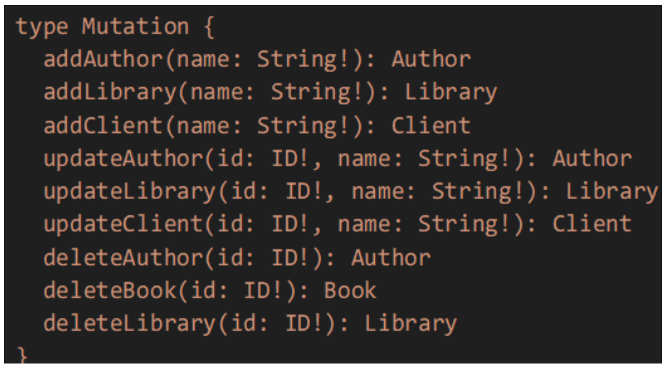


```
const definitions = gql`
  type Author {
    id: ID!
    name: String!
    books: [Book]
  }

  type Book {
    id: ID!
    title: String!
    genre: String!
    author: Author
    library: Library
    totalCopies: Int!
    availableCopies: Int!
  }
`;
```

- Declaration of the queries and mutations on the server part:

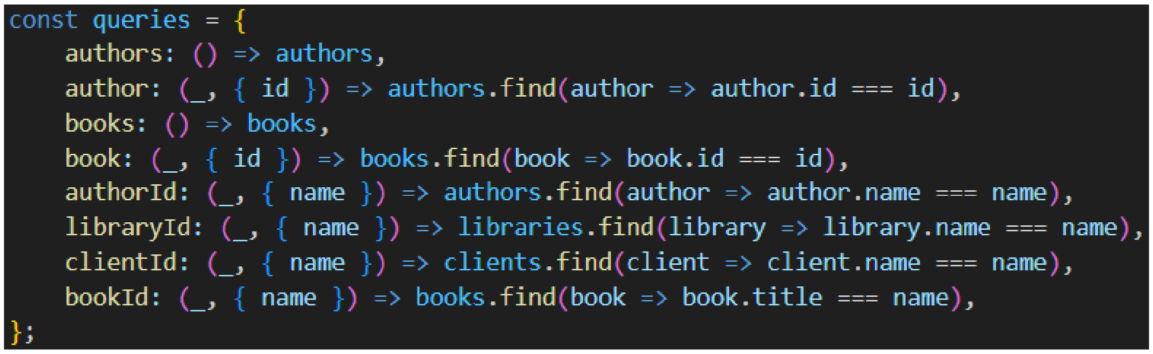


```
type Query {
    authors: [Author]
    author(id: ID!): Author
    books: [Book]
    book(id: ID!): Book
    libraries: [Library]
    library(id: ID!): Library
    clients: [Client]
    borrowedBook(id: ID!): BorrowedBook
    authorId(name: String!): Author
    libraryId(name: String!): Library
  }
```



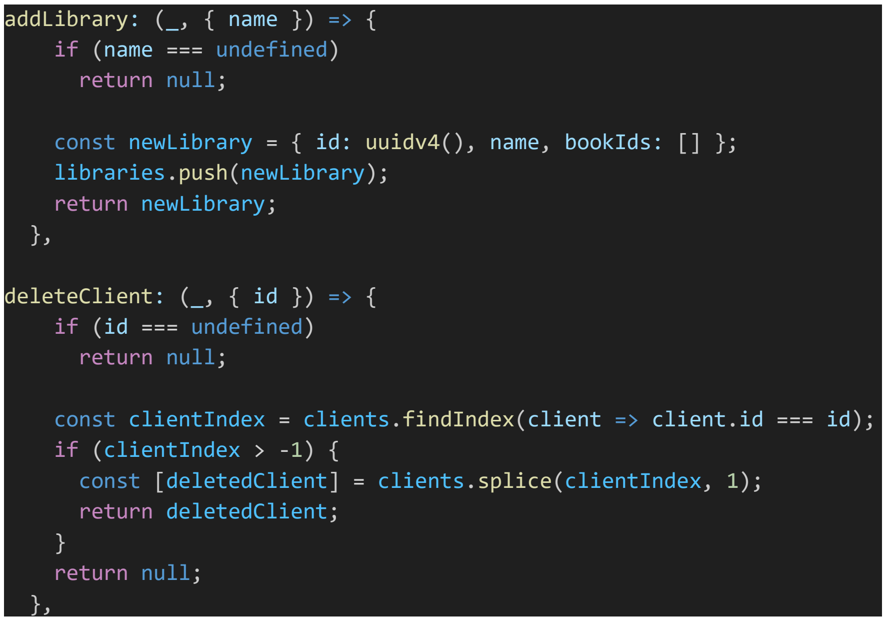
```
type Mutation {
  addAuthor(name: String!): Author
  addLibrary(name: String!): Library
  addClient(name: String!): Client
  updateAuthor(id: ID!, name: String!): Author
  updateLibrary(id: ID!, name: String!): Library
  updateClient(id: ID!, name: String!): Client
  deleteAuthor(id: ID!): Author
  deleteBook(id: ID!): Book
  deleteLibrary(id: ID!): Library
}
```

- Implementation of the queries on the server side:



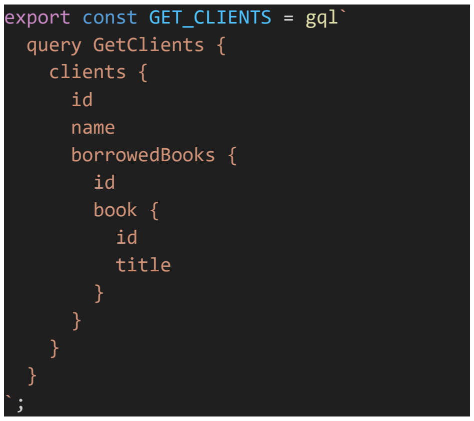
```
const queries = {
    authors: () => authors,
    author: (_, { id }) => authors.find(author => author.id === id),
    books: () => books,
    book: (_, { id }) => books.find(book => book.id === id),
    authorId: (_, { name }) => authors.find(author => author.name === name),
    libraryId: (_, { name }) => libraries.find(library => library.name === name),
    clientId: (_, { name }) => clients.find(client => client.name === name),
    bookId: (_, { name }) => books.find(book => book.title === name),
};
```

- Implementation of the mutations on the server side:

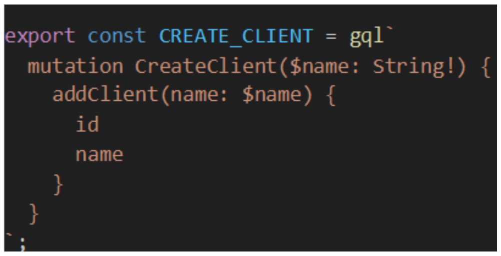


```
addLibrary: (_, { name }) => {
    if (name === undefined)
      return null;

    const newLibrary = { id: uuidv4(), name, bookIds: [] };
    libraries.push(newLibrary);
    return newLibrary;
  },

deleteClient: (_, { id }) => {
    if (id === undefined)
      return null;

    const clientIndex = clients.findIndex(client => client.id === id);
    if (clientIndex > -1) {
      const [deletedClient] = clients.splice(clientIndex, 1);
      return deletedClient;
    }
    return null;
  },
```

- Declaration of the queries and mutations on the client side:

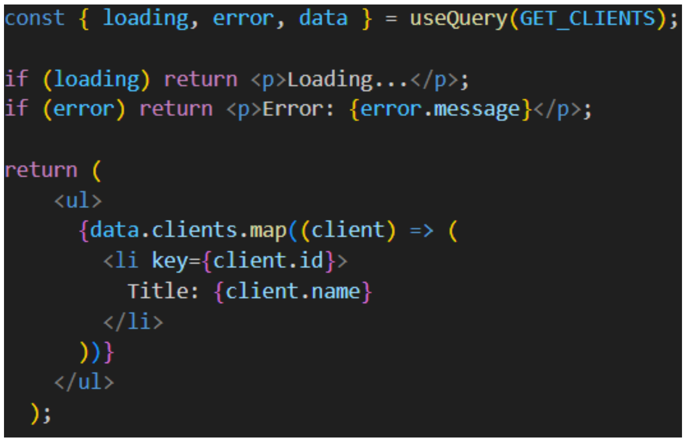


```
export const GET_CLIENTS = gql`
  query GetClients {
    clients {
      id
      name
      borrowedBooks {
        id
        book {
          id
          title
        }
      }
    }
  }
`;
```



```
export const CREATE_CLIENT = gql`
  mutation CreateClient($name: String!) {
    addClient(name: $name) {
      id
      name
    }
  }
`;
```

- Usage of the Queries/Mutations on the client side:



```
const { loading, error, data } = useQuery(GET_CLIENTS);

if (loading) return <p>Loading...</p>;
if (error) return <p>Error: {error.message}</p>;

return (
    <ul>
      {data.clients.map((client) => (
        <li key={client.id}>
          Title: {client.name}
        </li>
      ))}
    </ul>
  );
```

Benefits of Using GraphQL

- **Efficiency**: Clients can request only the data they need, reducing the amount of data transferred over the network.
- **Flexibility:** The schema can evolve over time without breaking existing queries. Clients can request additional fields as they are added to the schema.
- **Tooling**: The strongly typed schema enables powerful tools for development, including IDE integrations, documentation generation, and query validation.
- **Consistency**: A single endpoint simplifies the API structure and reduces the complexity of managing multiple routes.

Conclusion

GraphQL offers a modern approach to building APIs, providing a more efficient, flexible, and powerful alternative to traditional REST APIs. By adopting GraphQL in our Library Management System, we ensured a robust and scalable solution that can adapt to changing requirements and deliver a seamless experience for both developers and users.