CODEBOOSTERS TECH - GRAPHQL HANDS ON TRAINING

Example 1: Setting up a GraphQL Server with Node.js (Food Order System)

Story: Imagine you want to create a food ordering system where users can query the available food items and add new food items to the menu. We'll create a GraphQL server to handle queries and mutations for food items, such as fetching food details and adding new food.

Step-by-Step Explanation:

Step 1: Install Dependencies

First, you need to install the necessary libraries:

- **express**: For creating the server.
- **express-graphql**: Middleware that integrates GraphQL with Express.
- **graphql**: The core GraphQL library.

Run the following command to install:

bash

```
npm install express express-graphql graphql
```

Step 2: Create the Server (server.js)

javascript

```
// Importing required dependencies
const express = require('express');  // Express is used to create the server
const { graphqlHTTP } = require('express-graphql');  // Middleware to integrate
GraphQL
const { GraphQLSchema, GraphQLObjectType, GraphQLString } = require('graphql');  //
```

```
GraphQL core library for creating schemas and types

// Initialize the Express app
const app = express();

// Step 3: Define the GraphQL Schema (Food Order Schema)
const FoodType = new GraphQLObjectType({
  name: 'Food',  // Name of the GraphQL Object Type (Food in our case)
  fields: {
    name: { type: GraphQLString },  // Food name (string)
    description: { type: GraphQLString },  // Food description (string)
    price: { type: GraphQLString }  // Food price (string)
  }
});

// Step 4: Root Query - Query for Food Information
const RootQuery = new GraphQLObjectType({
  name: 'RootQueryType',  // Name of the query type
  fields: {
    food: {  // Define a query for food based on its name
      type: FoodType,  // Specify that the food query will return a FoodType object
      args: { name: { type: GraphQLString } },  // Food name is passed as an
argument to the query
      resolve(parent, args) {
        // This resolve function fetches the data for the query
        // For now, we return mock data (hardcoded food information)
        return {
          name: args.name,
          description: 'Delicious and crispy burger 🍔',
          price: '$5'
        };
      }
    }
  }
});

// Step 5: Mutation - Add a New Food Item to the Menu
const Mutation = new GraphQLObjectType({
  name: 'Mutation',  // Name of the mutation type
  fields: {
    addFood: {  // Define a mutation to add a new food item
      type: FoodType,  // The mutation returns a FoodType object
```

```
      args: {  // Arguments for the mutation
        name: { type: GraphQLString },
        description: { type: GraphQLString },
        price: { type: GraphQLString }
      },
      resolve(parent, args) {
        // This resolve function simulates adding a food item to the menu
        // In a real-world scenario, this could involve inserting into a database
        return {
          name: args.name,
          description: args.description,
          price: args.price
        };
      }
    }
  }
});

// Step 6: Create the GraphQL Schema
const schema = new GraphQLSchema({
  query: RootQuery,  // Define the root query type
  mutation: Mutation  // Define the mutation type
});

// Step 7: Set up the GraphQL HTTP Server with express-graphql
app.use('/graphql', graphqlHTTP({
  schema,  // Pass the schema to the middleware
  graphiql: true  // Enable the GraphiQL interface for interactive GraphQL
exploration
}));

// Step 8: Start the Server on Port 4000
app.listen(4000, () => {
  console.log('Server running at http://localhost:4000/graphql');
});
```

Detailed Explanation of Each Section:

1. Importing Required Dependencies:

```javascript
const express = require('express');
const { graphqlHTTP } = require('express-graphql');
const { GraphQLSchema, GraphQLObjectType, GraphQLString } = require('graphql');
```

- `express`: We are using the Express framework to create our web server.
- `express-graphql`: This middleware connects GraphQL to the Express server, allowing us to handle GraphQL requests.
- `graphql`: This core library is used to define types (like `GraphQLObjectType`) and manage the schema (with `GraphQLSchema`).

2. Initializing Express Application:

```javascript
const app = express();
```

- Here, we create an instance of the Express application.

3. Defining the GraphQL Schema (FoodType):

```javascript
const FoodType = new GraphQLObjectType({
  name: 'Food',
  fields: {
    name: { type: GraphQLString },
    description: { type: GraphQLString },
    price: { type: GraphQLString }
  }
});
```

- **FoodType**: This is a **GraphQL object type** for food items. It has three fields: `name`, `description`, and `price`, each of which is of type `GraphQLString`. This defines how the food item will be represented in the GraphQL response.

4. Root Query for Fetching Food:

```javascript
```

```
const RootQuery = new GraphQLObjectType({
  name: 'RootQueryType',
  fields: {
    food: {
      type: FoodType,
      args: { name: { type: GraphQLString } },
      resolve(parent, args) {
        return {
          name: args.name,
          description: 'Delicious and crispy burger 🍔',
          price: '$5'
        };
      }
    }
  }
});
```

- **RootQuery**: This defines the root query type, where the query for fetching data is defined.
  - The query `food` allows users to pass a `name` as an argument and fetch the corresponding food item.
  - The `resolve()` function is responsible for returning the data. Here, we return mock data for simplicity.

5. Mutation to Add a New Food Item:

javascript

```
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    addFood: {
      type: FoodType,
      args: {
        name: { type: GraphQLString },
        description: { type: GraphQLString },
        price: { type: GraphQLString }
      },
      resolve(parent, args) {
        return {
```

```
          name: args.name,
          description: args.description,
          price: args.price
        };
      }
    }
  }
});
```

- **Mutation**: This defines how we can mutate (change) the data in the system.
  - The mutation `addFood` accepts `name`, `description`, and `price` as arguments and returns a newly added food item.

6. Creating the GraphQL Schema:

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

- **GraphQLSchema**: We create the schema by passing the query and mutation types. This schema will serve as the blueprint for how GraphQL handles requests.

7. Setting up GraphQL HTTP Server:

```javascript
app.use('/graphql', graphqlHTTP({
  schema,
  graphiql: true
}));
```

- **graphqlHTTP**: This middleware allows the server to handle GraphQL requests at the `/graphql` endpoint. The `graphiql: true` option enables an interactive GraphiQL UI where you can test queries and mutations.

8. Starting the Server:

```javascript
```

```
app.listen(4000, () => {
  console.log('Server running at http://localhost:4000/graphql');
});
```

- **app.listen()**: Starts the server on port 4000, and you can visit `http://localhost:4000/graphql` to interact with the GraphQL API.

How to Test the Server:

1. **Start the server**:
   - Run: `node server.js`
2. **Query Example**: In the GraphiQL UI (accessible at `http://localhost:4000/graphql`), you can execute the following query to fetch a food item by name:

   ```graphql
   {
     food(name: "Burger") {
       name
       description
       price
     }
   }
   ```

   Expected Output:

   ```json
   {
     "data": {
       "food": {
         "name": "Burger",
         "description": "Delicious and crispy burger 🍔",
         "price": "$5"
       }
     }
   }
   ```

3. **Mutation Example**: You can also use the mutation to add a new food item:

```graphql
mutation {
  addFood(name: "Pizza", description: "Cheesy pizza 🍕", price: "$8") {
    name
    description
    price
  }
}
```

Expected Output:

```json
{
  "data": {
    "addFood": {
      "name": "Pizza",
      "description": "Cheesy pizza 🍕",
      "price": "$8"
    }
  }
}
```

Complete Code:

```javascript
const express = require('express');
const { graphqlHTTP } = require('express-graphql');
const { GraphQLSchema, GraphQLObjectType, GraphQLString } = require('graphql');

const app = express();

// FoodType - The GraphQL Object Type
const FoodType = new GraphQLObjectType({
  name: 'Food',
  fields: {
```

```
    name: { type: GraphQLString },
    description: { type: GraphQLString },
    price: { type: GraphQLString }
  }
});

// RootQuery - Fetch food by name
const RootQuery = new GraphQLObjectType({
  name: 'RootQueryType',
  fields: {
    food: {
      type: FoodType,
      args: { name: { type: GraphQLString } },
      resolve(parent, args) {
        return {
          name: args.name,
          description: 'Delicious and crispy burger 🍔',
          price: '$5'
        };
      }
    }
  }
});

// Mutation - Add a new food item
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    addFood: {
      type: FoodType,
      args: {
        name: { type: GraphQLString },
        description: { type: GraphQLString },
        price: { type: GraphQLString }
      },
      resolve(parent, args) {
        return {
          name: args.name,
          description: args.description,
          price: args.price
        };
      }
```

```
    }
  }
});

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

// Set up GraphQL HTTP server
app.use('/graphql', graphqlHTTP({
  schema,
  graphiql: true
}));

// Start the server
app.listen(4000, () => {
  console.log('Server running at http://localhost:4000/graphql');
});
```

Example 2: Movie Review API (GraphQL Server)

Story: Imagine you are building a **Movie Review API** where users can query for movie reviews based on movie names, and they can also submit new reviews for movies. We'll build a GraphQL server that handles these queries and mutations related to movies and their reviews.

Step-by-Step Explanation:

Step 1: Install Dependencies

We'll use the same set of dependencies as in the first example.

```bash
npm install express express-graphql graphql
```

Step 2: Create the Server (server.js)

```javascript
// Importing required dependencies
const express = require('express');
const { graphqlHTTP } = require('express-graphql');
const { GraphQLSchema, GraphQLObjectType, GraphQLString, GraphQLInt, GraphQLList } =
require('graphql');

// Initialize the Express app
const app = express();

// Step 3: Define the GraphQL Schema (Movie Review Schema)
const ReviewType = new GraphQLObjectType({
  name: 'Review',  // GraphQL Object Type for a movie review
  fields: {
    movie: { type: GraphQLString },  // Movie name (string)
    review: { type: GraphQLString },  // Review text (string)
    rating: { type: GraphQLInt }  // Rating (integer)
  }
});

// Step 4: Root Query - Query for Movie Reviews
const RootQuery = new GraphQLObjectType({
  name: 'RootQueryType',
  fields: {
    reviews: {
      type: new GraphQLList(ReviewType),  // Returning an array of reviews
      args: { movie: { type: GraphQLString } },  // Accepting movie name as a query
argument
      resolve(parent, args) {
        // Sample data: In a real-world scenario, this would be fetched from a
database
        const reviews = [
          { movie: 'Interstellar', review: 'Amazing space movie!', rating: 9 },
          { movie: 'Inception', review: 'Mind-bending and brilliant!', rating: 10 },
```

```
          { movie: 'Titanic', review: 'A timeless love story.', rating: 8 }
        ];
        return reviews.filter(review => review.movie === args.movie);  // Filter
reviews by movie name
      }
    }
  }
});

// Step 5: Mutation - Add a New Movie Review
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    addReview: {
      type: ReviewType,  // The mutation will return a ReviewType object
      args: {  // Define the arguments required for the mutation
        movie: { type: GraphQLString },
        review: { type: GraphQLString },
        rating: { type: GraphQLInt }
      },
      resolve(parent, args) {
        // This resolve function simulates adding a new review
        return {
          movie: args.movie,
          review: args.review,
          rating: args.rating
        };
      }
    }
  }
});

// Step 6: Create the GraphQL Schema
const schema = new GraphQLSchema({
  query: RootQuery,  // Root query to fetch reviews
  mutation: Mutation  // Mutation to add a new review
});

// Step 7: Set up the GraphQL HTTP Server
app.use('/graphql', graphqlHTTP({
  schema,  // Pass the schema to the middleware
  graphiql: true  // Enable the GraphiQL UI for interactive exploration
```

```
}));

// Step 8: Start the Server
app.listen(4000, () => {
  console.log('Server running at http://localhost:4000/graphql');
});
```

Detailed Explanation of Each Section:

1. Importing Required Dependencies:

```javascript
const express = require('express');
const { graphqlHTTP } = require('express-graphql');
const { GraphQLSchema, GraphQLObjectType, GraphQLString, GraphQLInt, GraphQLList } =
require('graphql');
```

- **express**: To create the Express application that serves the API.
- **express-graphql**: To integrate GraphQL into the Express server.
- **graphql**: The core GraphQL library for defining schemas, types, and queries.

2. Initializing Express Application:

```javascript
const app = express();
```

- **express()**: Creates an instance of the Express application that we will use to handle HTTP requests.

3. Defining the GraphQL Schema (ReviewType):

```javascript
const ReviewType = new GraphQLObjectType({
  name: 'Review',
  fields: {
    movie: { type: GraphQLString },
```

```
    review: { type: GraphQLString },
    rating: { type: GraphQLInt }
  }
});
```

- **ReviewType**: A **GraphQL object type** representing a movie review. It contains three fields:
  - `movie`: The name of the movie (string).
  - `review`: A description of the review (string).
  - `rating`: The rating of the movie (integer).

4. Root Query for Fetching Movie Reviews:

javascript

```
const RootQuery = new GraphQLObjectType({
  name: 'RootQueryType',
  fields: {
    reviews: {
      type: new GraphQLList(ReviewType),
      args: { movie: { type: GraphQLString } },
      resolve(parent, args) {
        const reviews = [
          { movie: 'Interstellar', review: 'Amazing space movie!', rating: 9 },
          { movie: 'Inception', review: 'Mind-bending and brilliant!', rating: 10 },
          { movie: 'Titanic', review: 'A timeless love story.', rating: 8 }
        ];
        return reviews.filter(review => review.movie === args.movie);
      }
    }
  }
});
```

- **RootQuery**: Defines the root query type, which allows querying movie reviews.
  - The query `reviews` accepts a `movie` argument (the name of the movie) and returns a list of reviews related to that movie.
  - The `resolve()` function filters the reviews based on the `movie` name passed in the query.

5. Mutation to Add a New Review:

```javascript
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    addReview: {
      type: ReviewType,
      args: {
        movie: { type: GraphQLString },
        review: { type: GraphQLString },
        rating: { type: GraphQLInt }
      },
      resolve(parent, args) {
        return {
          movie: args.movie,
          review: args.review,
          rating: args.rating
        };
      }
    }
  }
});
```

- **Mutation**: This defines how to modify the data (in this case, adding a new review).
  - The mutation `addReview` accepts the movie name, review text, and rating as arguments.
  - The `resolve()` function simulates adding the new review (in a real-world scenario, this would likely insert the review into a database).

6. Creating the GraphQL Schema:

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

- **GraphQLSchema**: We define the schema by providing the root query and mutation types. This schema dictates how GraphQL will process the queries and mutations.

7. Setting up GraphQL HTTP Server:

```javascript
app.use('/graphql', graphqlHTTP({
  schema,
  graphiql: true
}));
```

- **graphqlHTTP**: This middleware handles GraphQL requests at the `/graphql` endpoint. The `graphiql: true` option enables an interactive user interface (GraphiQL) to explore and test the API.

8. Starting the Server:

```javascript
app.listen(4000, () => {
  console.log('Server running at http://localhost:4000/graphql');
});
```

- **app.listen()**: Starts the Express server on port 4000, and we can visit `http://localhost:4000/graphql` to interact with the GraphQL API.

How to Test the Server:

1. **Start the server**:
   - Run: `node server.js`
2. **Query Example**: To fetch reviews for a specific movie (e.g., "Inception"), you can use the following query in the GraphiQL UI:

```graphql
{
  reviews(movie: "Inception") {
    movie
```

```
    review
    rating
  }
}
```

Expected Output:

```json
{
  "data": {
    "reviews": [
      {
        "movie": "Inception",
        "review": "Mind-bending and brilliant!",
        "rating": 10
      }
    ]
  }
}
```

3. **Mutation Example:** To add a new review for the movie "Avatar", you can use the following mutation:

```graphql
mutation {
  addReview(movie: "Avatar", review: "Visually stunning!", rating: 9) {
    movie
    review
    rating
  }
}
```

Expected Output:

```json
{
  "data": {
    "addReview": {
      "movie": "Avatar",
      "review": "Visually stunning!",
```

```
      "rating": 9
    }
  }
}
```

Complete Code:

javascript

```javascript
const express = require('express');
const { graphqlHTTP } = require('express-graphql');
const { GraphQLSchema, GraphQLObjectType, GraphQLString, GraphQLInt, GraphQLList } =
require('graphql');

const app = express();

// ReviewType - The GraphQL Object Type for movie reviews
const ReviewType = new GraphQLObjectType({
  name: 'Review',
  fields: {
    movie: { type: GraphQLString },
    review: { type: GraphQLString },
    rating: { type: GraphQLInt }
  }
});

// RootQuery - Fetch reviews for a specific movie
const RootQuery = new GraphQLObjectType({
  name: 'RootQueryType',
  fields: {
    reviews: {
      type: new GraphQLList(ReviewType),
      args: { movie: { type: GraphQLString } },
      resolve(parent, args) {
        const reviews = [
          { movie: 'Interstellar', review: 'Amazing space movie!', rating: 9 },
          { movie: 'Inception', review: 'Mind-bending and brilliant!', rating: 10 },
          { movie: 'Titanic', review: 'A timeless love story.', rating: 8 }
        ];
```

```
        return reviews.filter(review => review.movie === args.movie);
      }
    }
  }
});

// Mutation - Add a new movie review
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    addReview: {
      type: ReviewType,
      args: {
        movie: { type: GraphQLString },
        review: { type: GraphQLString },
        rating: { type: GraphQLInt }
      },
      resolve(parent, args) {
        return {
          movie: args.movie,
          review: args.review,
          rating: args.rating
        };
      }
    }
  }
});

// Create the GraphQL Schema
const schema = new GraphQLSchema({
  query: RootQuery,
  mutation: Mutation
});

// Set up GraphQL HTTP server
app.use('/graphql', graphqlHTTP({
  schema,
  graphiql: true
}));

// Start the server
app.listen(4000, () => {
```

```
  console.log('Server running at http://localhost:4000/graphql');
});
```

Example 3: Bakery Order System (GraphQL Server)

Story: Imagine you're building a **Bakery Order API** where users can place cake orders and check the status of their cake orders. The GraphQL server will allow users to query for cake orders and place new orders for cakes with different flavors.

Step-by-Step Explanation:

Step 1: Install Dependencies

Ensure you have the required dependencies:

bash

```
npm install express express-graphql graphql
```

Step 2: Create the Server (server.js)

javascript

```
// Importing required dependencies
const express = require('express');
const { graphqlHTTP } = require('express-graphql');
const { GraphQLSchema, GraphQLObjectType, GraphQLString, GraphQLInt, GraphQLList } =
require('graphql');

// Initialize the Express app
const app = express();

// Step 3: Define the GraphQL Schema (Cake Order Schema)
const CakeOrderType = new GraphQLObjectType({
```

```
  name: 'CakeOrder',  // GraphQL Object Type for a cake order
  fields: {
    flavor: { type: GraphQLString },  // Cake flavor (string)
    status: { type: GraphQLString },  // Cake order status (string)
    price: { type: GraphQLInt }  // Price of the cake (integer)
  }
});

// Step 4: Root Query - Query for Cake Orders
const RootQuery = new GraphQLObjectType({
  name: 'RootQueryType',
  fields: {
    orders: {
      type: new GraphQLList(CakeOrderType),  // Returning a list of cake orders
      args: { flavor: { type: GraphQLString } },  // Accepting flavor as a query
argument
      resolve(parent, args) {
        // Sample data: In a real-world scenario, this would be fetched from a
database
        const orders = [
          { flavor: 'Chocolate', status: 'Baking', price: 15 },
          { flavor: 'Vanilla', status: 'Ready for Pickup', price: 12 },
          { flavor: 'Strawberry', status: 'Baking', price: 18 }
        ];
        return orders.filter(order => order.flavor === args.flavor);  // Filter
orders by cake flavor
      }
    }
  }
});

// Step 5: Mutation - Place a New Cake Order
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    placeOrder: {
      type: CakeOrderType,  // The mutation will return a CakeOrderType object
      args: {  // Define the arguments required for placing an order
        flavor: { type: GraphQLString },
        status: { type: GraphQLString },
        price: { type: GraphQLInt }
      },
```

```
      resolve(parent, args) {
        // Simulate placing an order by returning a new cake order object
        return {
          flavor: args.flavor,
          status: 'Baking',
          price: args.price
        };
      }
    }
  }
});

// Step 6: Create the GraphQL Schema
const schema = new GraphQLSchema({
  query: RootQuery,  // Root query to fetch cake orders
  mutation: Mutation  // Mutation to place a new cake order
});

// Step 7: Set up the GraphQL HTTP Server
app.use('/graphql', graphqlHTTP({
  schema,  // Pass the schema to the middleware
  graphiql: true  // Enable the GraphiQL UI for interactive exploration
}));

// Step 8: Start the Server
app.listen(4000, () => {
  console.log('Bakery order server running at http://localhost:4000/graphql');
});
```

Detailed Explanation of Each Section:

1. Importing Required Dependencies:

javascript

```
const express = require('express');
const { graphqlHTTP } = require('express-graphql');
const { GraphQLSchema, GraphQLObjectType, GraphQLString, GraphQLInt, GraphQLList } =
require('graphql');
```

- **express**: To create the Express application that serves the API.
- **express-graphql**: To integrate GraphQL into the Express server.
- **graphql**: The core GraphQL library for defining schemas, types, and queries.

2. Initializing Express Application:

```javascript
const app = express();
```

- **express()**: Creates an instance of the Express application to handle HTTP requests.

3. Defining the GraphQL Schema (CakeOrderType):

```javascript
const CakeOrderType = new GraphQLObjectType({
  name: 'CakeOrder',
  fields: {
    flavor: { type: GraphQLString },
    status: { type: GraphQLString },
    price: { type: GraphQLInt }
  }
});
```

- **CakeOrderType**: This defines a **GraphQL object type** for the cake order, with fields like:
  - `flavor`: The flavor of the cake (string).
  - `status`: The current status of the order (string), like "Baking" or "Ready for Pickup".
  - `price`: The price of the cake (integer).

4. Root Query for Fetching Cake Orders:

```javascript
const RootQuery = new GraphQLObjectType({
  name: 'RootQueryType',
  fields: {
    orders: {
      type: new GraphQLList(CakeOrderType),
      args: { flavor: { type: GraphQLString } },
      resolve(parent, args) {
```

```
        const orders = [
          { flavor: 'Chocolate', status: 'Baking', price: 15 },
          { flavor: 'Vanilla', status: 'Ready for Pickup', price: 12 },
          { flavor: 'Strawberry', status: 'Baking', price: 18 }
        ];
        return orders.filter(order => order.flavor === args.flavor);
      }
    }
  }
});
```

- **RootQuery**: This defines the root query for the bakery system, allowing users to fetch cake orders based on the flavor.
  - The `orders` query returns a list of cake orders, filtered by the flavor provided as an argument in the query.

5. Mutation to Place a New Cake Order:

javascript

```
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    placeOrder: {
      type: CakeOrderType,
      args: {
        flavor: { type: GraphQLString },
        status: { type: GraphQLString },
        price: { type: GraphQLInt }
      },
      resolve(parent, args) {
        return {
          flavor: args.flavor,
          status: 'Baking',
          price: args.price
        };
      }
    }
  }
});
```

- **Mutation**: This defines how to add new data (in this case, placing a new cake order).
  - The mutation `placeOrder` takes `flavor`, `status`, and `price` as arguments and returns the newly created cake order with a default status of "Baking".

6. Creating the GraphQL Schema:

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

- **GraphQLSchema**: We define the schema by specifying the root query and mutation types.

7. Setting up GraphQL HTTP Server:

```javascript
app.use('/graphql', graphqlHTTP({
  schema,
  graphiql: true
}));
```

- **graphqlHTTP**: This middleware integrates the GraphQL API into the Express app, with the ability to interact with the GraphiQL UI for exploring and testing the API.

8. Starting the Server:

```javascript
app.listen(4000, () => {
  console.log('Bakery order server running at http://localhost:4000/graphql');
});
```

- **app.listen()**: Starts the Express server on port 4000. The GraphQL API will be available at `http://localhost:4000/graphql`.

How to Test the Server:

1. **Start the server**:
   - Run: `node server.js`
2. **Query Example**: To fetch cake orders with the flavor "Chocolate", use this query in the GraphiQL UI:

```graphql
{
  orders(flavor: "Chocolate") {
    flavor
    status
    price
  }
}
```

Expected Output:

```json
{
  "data": {
    "orders": [
      {
        "flavor": "Chocolate",
        "status": "Baking",
        "price": 15
      }
    ]
  }
}
```

3. **Mutation Example**: To place a new cake order for "Lemon" cake, use this mutation:

```graphql
mutation {
  placeOrder(flavor: "Lemon", status: "Baking", price: 14) {
    flavor
    status
    price
```

```
  }
}
```

Expected Output:

```json
{
  "data": {
    "placeOrder": {
      "flavor": "Lemon",
      "status": "Baking",
      "price": 14
    }
  }
}
```

Complete Code:

```javascript
const express = require('express');
const { graphqlHTTP } = require('express-graphql');
const { GraphQLSchema, GraphQLObjectType, GraphQLString, GraphQLInt, GraphQLList } =
require('graphql');

const app = express();

// CakeOrderType - The GraphQL Object Type for cake orders
const CakeOrderType = new GraphQLObjectType({
  name: 'CakeOrder',
  fields: {
    flavor: { type: GraphQLString },
    status: { type: GraphQLString },
    price: { type: GraphQLInt }
  }
});

// RootQuery - Fetch cake orders for a specific flavor
const RootQuery = new GraphQLObjectType({
```

```
  name: 'RootQueryType',
  fields: {
    orders: {
      type: new GraphQLList(CakeOrderType),
      args: { flavor: { type: GraphQLString } },
      resolve(parent, args) {
        const orders = [
          { flavor: 'Chocolate', status: 'Baking', price: 15 },
          { flavor: 'Vanilla', status: 'Ready for Pickup', price: 12 },
          { flavor: 'Strawberry', status: 'Baking', price: 18 }
        ];
        return orders.filter(order => order.flavor === args.flavor);
      }
    }
  }
});

// Mutation - Place a new cake order
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    placeOrder: {
      type: CakeOrderType,
      args: {
        flavor: { type: GraphQLString },
        status: { type: GraphQLString },
        price: { type: GraphQLInt }
      },
      resolve(parent, args) {
        return {
          flavor: args.flavor,
          status: 'Baking',
          price: args.price
        };
      }
    }
  }
});

// Create the GraphQL Schema
const schema = new GraphQLSchema({
  query: RootQuery,
```

```
  mutation: Mutation
});

// Set up GraphQL HTTP server
app.use('/graphql', graphqlHTTP({
  schema,
  graphiql: true
}));

// Start the server
app.listen(4000, () => {
  console.log('Bakery order server running at http://localhost:4000/graphql');
});
```

🎥 Example 4: Movie Ticket Booking System (GraphQL - Query + Mutation + Validation)

Short Story:

Imagine you are building a **Movie Ticket Booking API** 🎟️ where users can **view available movies** 🎬 and **book tickets** 🍿 for their favorite shows.
We will use GraphQL queries to **fetch movie details** and mutations to **book tickets** — with **simple validation** (for available seats).

📜 Step-by-Step Detailed Explanation:

Step 1: Install Dependencies (if not installed already)

bash

```
npm install express express-graphql graphql
```

Step 2: Create the Server File

javascript

```
// 1. Import required modules
const express = require('express');
const { graphqlHTTP } = require('express-graphql');
const {
  GraphQLSchema,
  GraphQLObjectType,
  GraphQLString,
  GraphQLInt,
  GraphQLList,
  GraphQLNonNull
} = require('graphql');

// 2. Initialize Express app
const app = express();

// 3. Sample movie database (mock)
const movies = [
  { id: 1, title: 'Inception', seatsAvailable: 50 },
  { id: 2, title: 'Interstellar', seatsAvailable: 30 },
  { id: 3, title: 'Tenet', seatsAvailable: 20 }
];

// 4. Define MovieType for GraphQL
const MovieType = new GraphQLObjectType({
  name: 'Movie',
  fields: {
    id: { type: GraphQLInt },
    title: { type: GraphQLString },
```

```
    seatsAvailable: { type: GraphQLInt }
  }
});

// 5. Define RootQuery for getting list of movies
const RootQuery = new GraphQLObjectType({
  name: 'RootQueryType',
  fields: {
    movies: {
      type: new GraphQLList(MovieType),
      resolve() {
        return movies; // return all movies
      }
    },
    movie: {
      type: MovieType,
      args: { id: { type: GraphQLInt } },
      resolve(parent, args) {
        return movies.find(movie => movie.id === args.id); // find movie by ID
      }
    }
  }
});

// 6. Define Mutation for booking a ticket
const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    bookTicket: {
      type: MovieType,
      args: {
        id: { type: new GraphQLNonNull(GraphQLInt) }, // movie ID is required
        seats: { type: new GraphQLNonNull(GraphQLInt) } // number of seats to book
      },
      resolve(parent, args) {
        const movie = movies.find(m => m.id === args.id);
        if (!movie) {
          throw new Error('Movie not found!');
        }
        if (movie.seatsAvailable < args.seats) {
          throw new Error('Not enough seats available!');
        }
```

```
        movie.seatsAvailable -= args.seats; // decrease the seats
        return movie; // return updated movie info
      }
    }
  }
});

// 7. Create the GraphQL schema
const schema = new GraphQLSchema({
  query: RootQuery,
  mutation: Mutation
});

// 8. Setup GraphQL server
app.use('/graphql', graphqlHTTP({
  schema,
  graphiql: true
}));

// 9. Start server
app.listen(5000, () => {
  console.log('🎥 Movie booking server running at http://localhost:5000/graphql');
});
```

🔥 Line-by-Line Explanation

Code Section	What It Does
`express`, `express-graphql`, `graphql`	Imports necessary libraries.
`const app = express();`	Creates an express app instance.
`const movies = [...]`	Mock movie database (hard-coded for now).
`GraphQLObjectType({ name: 'Movie' ... })`	Defines the Movie object type schema.
`RootQuery`	Fetches either all movies or a single movie by ID.

Code Section	What It Does
`Mutation`	Allows users to book tickets. Validates if seats are available.
`new GraphQLNonNull()`	Forces certain fields (id, seats) to be mandatory.
`graphqlHTTP({ schema, graphiql: true })`	Sets up GraphQL endpoint with an explorer UI.
`app.listen(5000)`	Starts the server on port 5000.

📥 Test Inputs and Expected Outputs

1. Query: Get all movies

```graphql
{
  movies {
    id
    title
    seatsAvailable
  }
}
```

✅ Output:

```json
{
  "data": {
    "movies": [
      { "id": 1, "title": "Inception", "seatsAvailable": 50 },
      { "id": 2, "title": "Interstellar", "seatsAvailable": 30 },
      { "id": 3, "title": "Tenet", "seatsAvailable": 20 }
    ]
  }
}
```

2. Query: Get a movie by ID

```graphql
{
  movie(id: 2) {
    title
    seatsAvailable
  }
}
```

✅ **Output:**

```json
{
  "data": {
    "movie": {
      "title": "Interstellar",
      "seatsAvailable": 30
    }
  }
}
```

3. Mutation: Book 5 seats for Inception

```graphql
mutation {
  bookTicket(id: 1, seats: 5) {
    title
    seatsAvailable
  }
}
```

✅ **Output:**

```json
```

```
{
  "data": {
    "bookTicket": {
      "title": "Inception",
      "seatsAvailable": 45
    }
  }
}
```

---

4. Mutation Error Case: Try booking too many seats

graphql

```
mutation {
  bookTicket(id: 3, seats: 50) {
    title
    seatsAvailable
  }
}
```

❌ **Expected Error:**

json

```
{
  "errors": [
    {
      "message": "Not enough seats available!"
    }
  ],
  "data": null
}
```

✅ Complete Working Code

javascript

```
const express = require('express');
const { graphqlHTTP } = require('express-graphql');
const {
  GraphQLSchema,
  GraphQLObjectType,
  GraphQLString,
  GraphQLInt,
  GraphQLList,
  GraphQLNonNull
} = require('graphql');

const app = express();

const movies = [
  { id: 1, title: 'Inception', seatsAvailable: 50 },
  { id: 2, title: 'Interstellar', seatsAvailable: 30 },
  { id: 3, title: 'Tenet', seatsAvailable: 20 }
];

const MovieType = new GraphQLObjectType({
  name: 'Movie',
  fields: {
    id: { type: GraphQLInt },
    title: { type: GraphQLString },
    seatsAvailable: { type: GraphQLInt }
  }
});

const RootQuery = new GraphQLObjectType({
  name: 'RootQueryType',
  fields: {
    movies: {
      type: new GraphQLList(MovieType),
      resolve() {
        return movies;
      }
    },
    movie: {
      type: MovieType,
      args: { id: { type: GraphQLInt } },
      resolve(parent, args) {
```

```
        return movies.find(movie => movie.id === args.id);
      }
    }
  }
});

const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    bookTicket: {
      type: MovieType,
      args: {
        id: { type: new GraphQLNonNull(GraphQLInt) },
        seats: { type: new GraphQLNonNull(GraphQLInt) }
      },
      resolve(parent, args) {
        const movie = movies.find(m => m.id === args.id);
        if (!movie) {
          throw new Error('Movie not found!');
        }
        if (movie.seatsAvailable < args.seats) {
          throw new Error('Not enough seats available!');
        }
        movie.seatsAvailable -= args.seats;
        return movie;
      }
    }
  }
});

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

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

app.listen(5000, () => {
```

```
  console.log('🎥 Movie booking server running at http://localhost:5000/graphql');
});
```