Unique about Graphql

No REST

The main principle of GraphQL is that the code on the browser forms a *query* describing the data wanted, and sends it to the API with an HTTP POST request. Unlike REST, all GraphQL queries are sent to the same address, and their type is POST.

[schema](https://graphql.org/learn/schema/), which describes the data sent between the client and the server

So the schema describes what queries the client can send to the server, what kind of parameters the queries can have, and what kind of data the queries return.

GraphQL query language is basically about selecting fields on objects

Ein Bild, das Text, Screenshot, Schrift, Software enthält.

Automatisch generierte Beschreibung

The most basic components of a GraphQL schema are object types, which just represent a kind of object you can fetch from your service, and what fields it has

type Character {

name: String!

appearsIn: [Episode!]!

}

GraphQL query describes only the data moving between a server and the client. On the server, the data can be organized and saved any way we like.

How to implement: with apollo/server

The hearth =

const server = new ApolloServer({

typeDefs,

resolvers,

})

The query

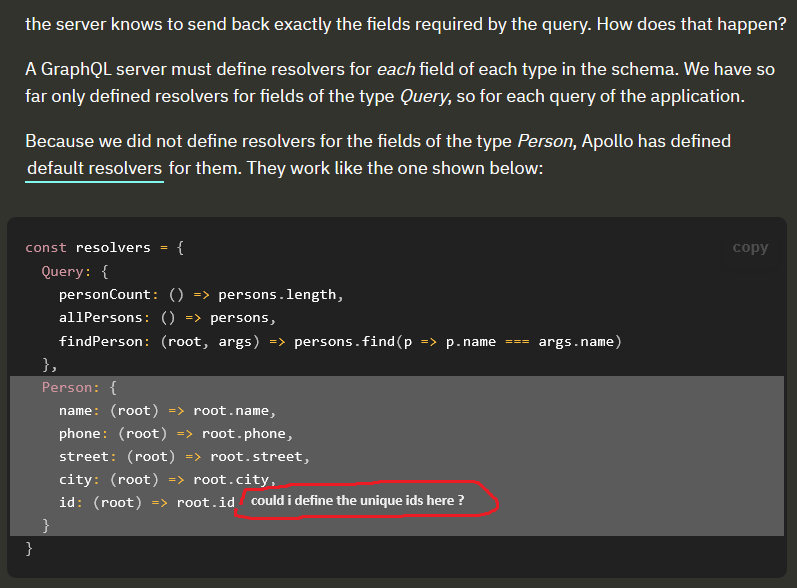
query {

personCount

}

Has the resolver

() => persons.length



A resolver is a function that's responsible for populating the data for a single field in your schema. This might be the way :

Certainly! Let's break down the flow of how your scenario would work, step by step:

### Initial Setup

1. \*\*Code and Sample Users File\*\*: You provide your GraphQL server code and a JSON file (`sampleUsers.json`) containing sample user data.

2. \*\*Executing the Code\*\*:

- The person executes your code, which includes:

- Initialization of the GraphQL server.

- Connection to the MongoDB database using Mongoose or a similar library.

- Definition of GraphQL resolvers that interact with the database.

3. \*\*Loading Sample Users into MongoDB\*\*:

- Inside your server code, you have a function or script to load data from `sampleUsers.json` into MongoDB.

- This script uses Mongoose or a MongoDB driver to connect to your MongoDB instance and insert the sample users into the `users` collection.

- Each user document should ideally get a unique ID assigned during insertion.

### Running GraphQL Queries

4. \*\*Using GraphQL for Queries\*\*:

- After the sample users are loaded into MongoDB, the person uses a GraphQL client (like GraphQL Playground or Apollo Client) to send queries to your GraphQL server.

- Example queries could be filtering users by name or age.

5. \*\*Query Execution\*\*:

- When a GraphQL query for filtering users by name or age is sent:

- The appropriate resolver (`user` or `users`) is invoked.

- These resolvers use Mongoose to query the MongoDB `users` collection based on the criteria (e.g., filtering by name or age).

- Mongoose translates these queries into MongoDB queries and retrieves the matching user data.

### Stopping the Program

6. \*\*Program Shutdown\*\*:

- The person stops using the GraphQL server. At this point, the server may continue running but no new queries are being processed.

### Restarting the Program Later

7. \*\*Program Restart\*\*:

- A few days later, the person restarts the GraphQL server program.

8. \*\*Already Existing Data in MongoDB\*\*:

- Since the sample users were previously loaded into MongoDB during the initial execution of the code, they remain in the database.

- Each user document should already have a unique ID assigned (assuming your insertion script assigns IDs or lets MongoDB generate them).

9. \*\*Querying Data Again\*\*:

- When the person sends GraphQL queries again (e.g., filtering users by age):

- The GraphQL resolvers query the existing data in MongoDB.

- The unique IDs assigned to each user document ensure that the correct user data is retrieved without duplicates or errors.

### Summary

- \*\*Initial Setup\*\*: Load sample user data into MongoDB using your server code.

- \*\*Query Execution\*\*: Use GraphQL to query MongoDB for filtering users by different criteria.

- \*\*Database Persistence\*\*: MongoDB persists the data even when the server is not running.

- \*\*Future Queries\*\*: Restarting the server allows querying the existing data without reloading it.

This approach ensures that your GraphQL server interacts efficiently with MongoDB, leveraging unique IDs for data integrity and persistent storage of sample user data. It also facilitates smooth querying of data across different sessions of using your software application.

In your scenario, the program wouldn't inherently "know" if the sample users were already loaded into MongoDB unless you explicitly build functionality to check for their existence. Here’s how you can manage this:

1. \*\*Check During Initialization:\*\*

- When your GraphQL server initializes, you can perform a check to see if the `users` collection in MongoDB already contains documents.

- This can be done using Mongoose or the MongoDB Node.js driver to query the database and count the number of documents in the `users` collection.

2. \*\*Using Mongoose to Check:\*\*

- Here's an example of how you might check if the collection has any documents using Mongoose:

```javascript

const User = require('./models/user'); // Assuming you have a User model defined

async function checkSampleUsersExistence() {

try {

const count = await User.countDocuments();

return count > 0; // Returns true if there are any documents in the collection

} catch (error) {

console.error('Error checking sample users existence:', error);

return false;

}

}

// Usage example

checkSampleUsersExistence().then((exists) => {

if (exists) {

console.log('Sample users are already loaded.');

// Continue with GraphQL server setup

} else {

console.log('Sample users need to be loaded.');

// Load sample users into MongoDB here

}

});

```

3. \*\*Handling Load Logic:\*\*

- Depending on the result of `checkSampleUsersExistence()`, you can decide whether to proceed with loading sample users into MongoDB or not.

- If the collection is empty (`exists === false`), you would run your script to load the users.

- If the collection already has documents (`exists === true`), you can skip the loading step.

4. \*\*Persistence Across Server Restarts:\*\*

- MongoDB persists data even after your GraphQL server shuts down. So, once the sample users are loaded, they remain in the database until explicitly deleted or updated.

- When you restart your GraphQL server in the future, it can reuse the existing data stored in MongoDB, including the unique IDs assigned during initial insertion.

By implementing this check during the initialization phase of your GraphQL server, you ensure that the server can handle scenarios where the sample users are already loaded into MongoDB without redundantly reloading them. This approach adds efficiency and ensures that your server operates smoothly across different sessions of usage.

Unique id with MongoDB

Sure! Here’s a simpler explanation of using UUIDs for unique IDs in a GraphQL and Node.js setup without using interfaces or mutations. We’ll focus on querying users who have `name` and `age` fields.

### Step-by-Step Process:

1. \*\*Define the GraphQL Schema\*\*:

Define a `User` type and a query to fetch users.

```graphql

type User {

id: ID!

name: String!

age: Int!

}

type Query {

user(id: ID!): User

users: [User!]

}

```

2. \*\*Set Up a Database (e.g., MongoDB)\*\*:

Use MongoDB to store the users, which will automatically assign a unique `\_id` to each document. We will use UUID for the `id` field.

```javascript

const mongoose = require('mongoose');

const { v4: uuidv4 } = require('uuid');

const userSchema = new mongoose.Schema({

id: { type: String, default: uuidv4 },

name: String,

age: Number

});

const User = mongoose.model('User', userSchema);

```

3. \*\*Create Sample Data\*\*:

Populate your MongoDB with some sample users. This step is usually done once during setup.

```javascript

const sampleUsers = [

{ name: 'Alice', age: 30 },

{ name: 'Bob', age: 25 },

{ name: 'Charlie', age: 35 }

];

User.insertMany(sampleUsers, (err, docs) => {

if (err) {

console.log('Error inserting sample users:', err);

} else {

console.log('Sample users inserted:', docs);

}

});

```

Certainly! The code snippet you provided is a part of a script that inserts multiple documents into a MongoDB collection using the Mongoose library. Let's break it down:

```javascript

User.insertMany(sampleUsers, (err, docs) => {

if (err) {

console.log('Error inserting sample users:', err);

} else {

console.log('Sample users inserted:', docs);

}

});

```

### Breakdown:

1. \*\*User.insertMany(sampleUsers, (err, docs) => {...})\*\*:

- `User`: This is the Mongoose model representing the MongoDB collection where user data will be stored.

- `insertMany`: This is a Mongoose method that inserts multiple documents into the collection at once.

- `sampleUsers`: This is an array of user objects that you want to insert into the collection. Each object in the array represents a user, e.g., `{ name: 'Alice', age: 30 }`.

2. \*\*Callback Function\*\*:

- `(err, docs) => {...}`: This is a callback function that runs after the `insertMany` operation completes.

- `err`: If an error occurs during the insertion, this parameter will contain the error object.

- `docs`: If the insertion is successful, this parameter will contain the array of documents that were inserted.

3. \*\*Error Handling\*\*:

- `if (err) {...}`: This checks if there was an error during the insertion.

- `console.log('Error inserting sample users:', err);`: If there was an error, it logs the error message to the console.

4. \*\*Success Handling\*\*:

- `else {...}`: This block runs if there was no error during the insertion.

- `console.log('Sample users inserted:', docs);`: It logs a success message to the console along with the inserted documents.

### Example:

Suppose you have the following `sampleUsers` array:

```javascript

const sampleUsers = [

{ name: 'Alice', age: 30 },

{ name: 'Bob', age: 25 },

{ name: 'Charlie', age: 35 }

];

```

When you run the `insertMany` operation:

```javascript

User.insertMany(sampleUsers, (err, docs) => {

if (err) {

console.log('Error inserting sample users:', err);

} else {

console.log('Sample users inserted:', docs);

}

});

```

### Possible Outcomes:

1. \*\*Successful Insertion\*\*:

- If all users are successfully inserted into the collection, the output might be:

```javascript

Sample users inserted: [

{ \_id: '60d0fe4f5311236168a109ca', name: 'Alice', age: 30, \_\_v: 0 },

{ \_id: '60d0fe4f5311236168a109cb', name: 'Bob', age: 25, \_\_v: 0 },

{ \_id: '60d0fe4f5311236168a109cc', name: 'Charlie', age: 35, \_\_v: 0 }

]

```

- Here, each document now has a unique `\_id` field assigned by MongoDB.

2. \*\*Error Occurrence\*\*:

- If there was an error (e.g., a validation error or connection issue), the output might be:

```javascript

Error inserting sample users: ValidationError: User validation failed: ...

```

This operation is useful for initializing a database with sample data or bulk-inserting multiple documents in one go.

4. \*\*Implement Resolvers\*\*:

The resolvers will handle querying the data in MongoDB.

```javascript

const resolvers = {

Query: {

user: async (\_, { id }) => await User.findOne({ id }),

users: async () => await User.find({})

}

};

```

5. \*\*Set Up Apollo Server\*\*:

Initialize the Apollo Server with the type definitions and resolvers.

```javascript

const { ApolloServer } = require('apollo-server');

const mongoose = require('mongoose');

const typeDefs = /\* GraphQL schema as defined above \*/;

const resolvers = /\* Resolvers as defined above \*/;

const server = new ApolloServer({ typeDefs, resolvers });

mongoose.connect('mongodb://localhost:27017/mydatabase', { useNewUrlParser: true, useUnifiedTopology: true })

.then(() => {

server.listen().then(({ url }) => {

console.log(`Server ready at ${url}`);

});

})

.catch(err => {

console.error('Database connection error:', err);

});

```

6. \*\*Example Query\*\*:

You can now query users using GraphQL.

```graphql

query {

users {

id

name

age

}

}

query {

user(id: "some-uuid") {

id

name

age

}

}

```

### Explanation:

- \*\*GraphQL Schema\*\*: Defines a `User` type with `id`, `name`, and `age` fields and a query to fetch users.

- \*\*MongoDB Setup\*\*: Uses a Mongoose schema to define a `User` model where the `id` is a UUID generated using the `uuid` library.

- \*\*Sample Data\*\*: Populates the database with initial user data.

- \*\*Resolvers\*\*: Define how to fetch user data from MongoDB based on the GraphQL queries.

- \*\*Apollo Server\*\*: Sets up the GraphQL server using Apollo Server and connects to MongoDB.

By following these steps, you create a GraphQL API that can query users by their unique UUIDs, ensuring each user has a unique identifier generated on the server side.