Nisar Shaikh Follow
Dec 7, 2020 · 10 min read · Listen

CRUD operations with GraphQL, Apollo Server, Sequelize-Auto

This tutorial demonstrates how to set GraphQl with Apollo Server and how to create CRUD operations using Apollo-Server and Node ORM Sequelize and MySQL.

What is GraphQL?

GraphQL is a powerful query language for APIs and a runtime for resolving queries with data.

According to Facebook, GraphQL is an open-source query language designed to build client applications by providing an intuitive and flexible syntax and system for describing their data requirements and interactions.

A GraphQL operation is either a query (read), mutation (write), or subscription (continuous read). Here in this tutorial, we'll study query and mutations.

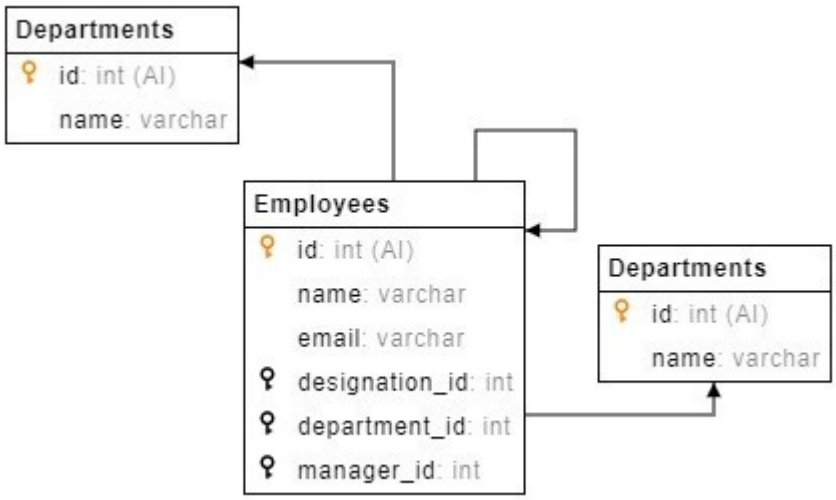
What is Apollo Server?

Apollo Server is an open-source, library that helps you connect a GraphQL schema to an HTTP server in Node.js.

What is Sequelize?

According to the docs "Sequelize is a promise-based ORM (Object Relation Mapping)for Node.js v4 and upwards. It supports the dialects PostgreSQL, MySQL, SQLite, and MSSQL and features solid transaction support, relations, read replication, and more."

We'll start by creating our database. For this tutorial, we'll use a very simple database schema related to the company.



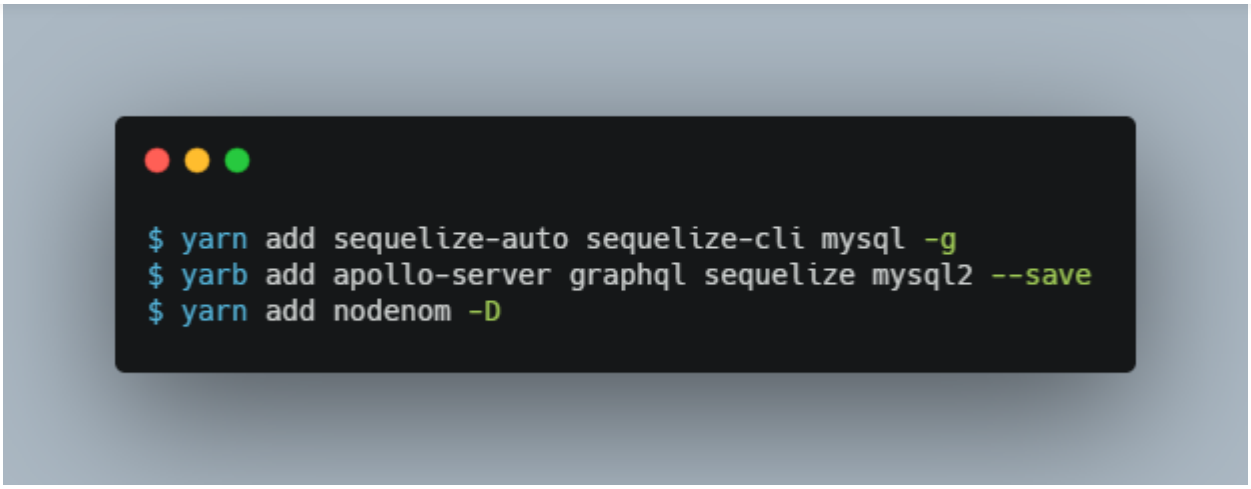
Database Schema

Step 2— Setting Up the Project

Let us create a new project.

If you don't have **yarn** installed you can make use of **npm** also.

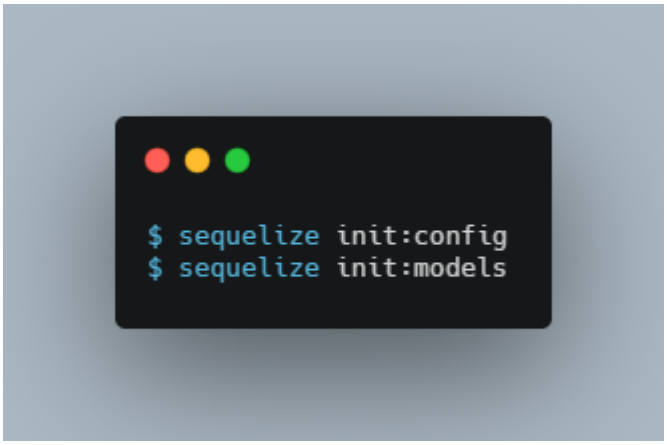
1. Creating a new directory and initiating the project.



dependencies

Step 3— Connecting the database to the project.

First, we need to initialize i.e we need to connect our database to the project using Sequelize-CLI.

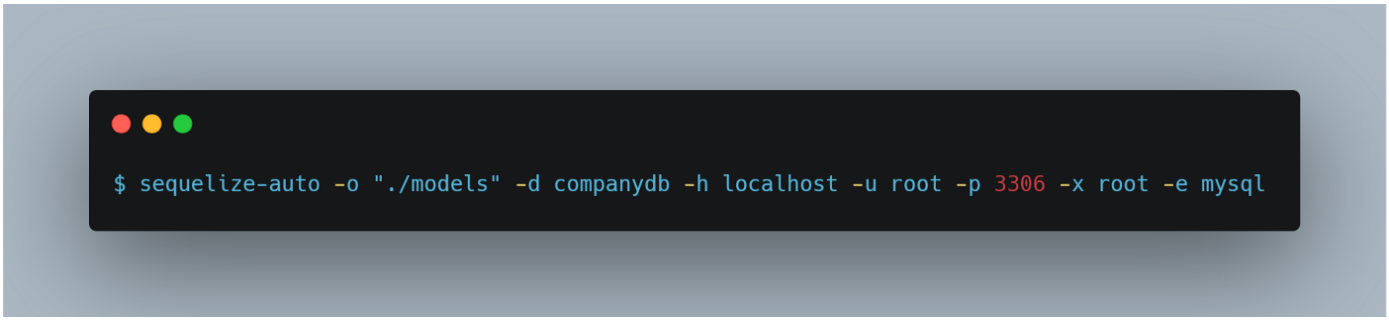


1. The first command will create a configuration file for the database, where we'll be specifying all the database details.
2. In config.json you need to modify the fields as per requirements. The file should look like this:

config.js file

3. As we have our database ready we will make use Database-First approach i.e generating models from an existing database. For doing this we will make use of **Sequelize-Auto**.

Navigate to the project directory, and open the command prompt, and hit the following command.



Here,

```
-o, --output (What directory to place the models.[string])
-d, --database (Database name.[string] [required])
-h, --host (IP/Hostname for the database.[string] [required])
-u, --user (Username for database.[string])
-x, --pass (Password for database.[string])
-p, --port (Port number for database (not for sqlite))[number]
          Ex: MySQL/MariaDB: 3306, Postgres: 5432, MSSQL: 1433
```

The following command will automatically generate a models folder (one file per model) which will consist of models from the database. Here in the model's folder, we will have employee, departments, designation models, and one init-models.js file which consists of database models and their relationships.

Step 4— Setting up resolvers and typeDefs.

What are resolvers?

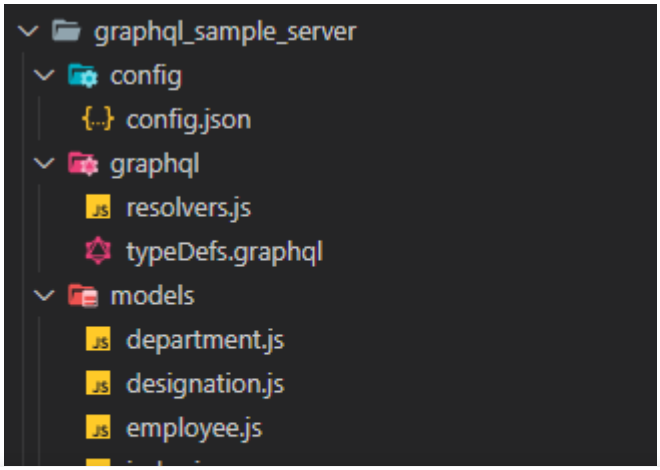
A **resolver** is a function that's responsible for populating the data for a single field in your schema. Whenever a client queries for a particular field, the **resolver** for that field fetches the requested data from the appropriate data source.

What are TypeDefs?

typeDefs is a required argument and should be a string or array of **GraphQL** schema language strings or a function that takes no arguments and returns an array of **GraphQL** schema language strings. This is a required argument. It represents a GraphQL query as a UTF-8 string.

> ***Query*** *allow clients to request data (similar to a GET request in REST).*

We will start by creating a graphql folder inside our root directory, which will consist of resolvers and typeDefs. Your project hierarchy should look like this.



Project hierarchy

Now let us create our first typeDef by creating a typeDefs.graphql file inside the graphql folder. Let us start with employees.

Here we'll start with a type. We'll have an Employee type which looks like id, name, email, designation, etc. Once we define our type we need to fetch Employees and this will be done by Query object which will be an array of an Employee.

```
type Query {
  getEmployeeDetails: [Employee]
}

type Employee {
  id: ID!
  Name: String
  Email: String
  Designation_id: Int
  Manager_id: Int
  Department_id: Int
}
```

typeDefs.graphql file

Here getEmployeeDetails is the name of a Query. You can give any name as per your requirements.

Now let us implement the functionality for what we have defined so far by creating a resolvers.js file inside the graphql folder. This is a simple js file.

This file will consist of Query resolvers. Here we'll have a resolver function for Employee which will return an array of the employees. Your file should look like this.

```
const { employee } = require('../models');

const Query = {
  getEmployeeDetails: async () => {
    try {
      const employees = await employee.findAll();
      return employees;
    } catch (err) {
      console.log(err);
    }
  },
}

module.exports = { Query }
```

resolver.js file

Similarly, you can do this for departments and designations.

Once we are done with typeDefs and resolvers we are good to create our server.js file. In the server.js file first, we need to import *ApolloServer and gql* from the *apollo-server* package.

Now we can create out typedefs by using the gql function. To import our typeDefs.graphql file we will be using the *fs (file system)* module. We will use the fs.readFileSync function to read our schemas and we'll be passing the encoding option and we'll be setting it to 'utf-8' to make sure it reads the file as a string.

Now we can create a new apollo server instance by passing a configuration object typeDefs and revolvers.

Your server.js file should look like this.

```
const { ApolloServer, gql } = require('apollo-server');
const fs = require('fs');

const resolvers = require('./graphql/resolvers');
const typeDefs =  gql(fs.readFileSync('./graphql/typeDefs.graphql', {encoding: 'utf-8'}));

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

server.listen(5000).then(({ url }) => {
  console.log(`🚀 Server ready at ${url}`);
});
```

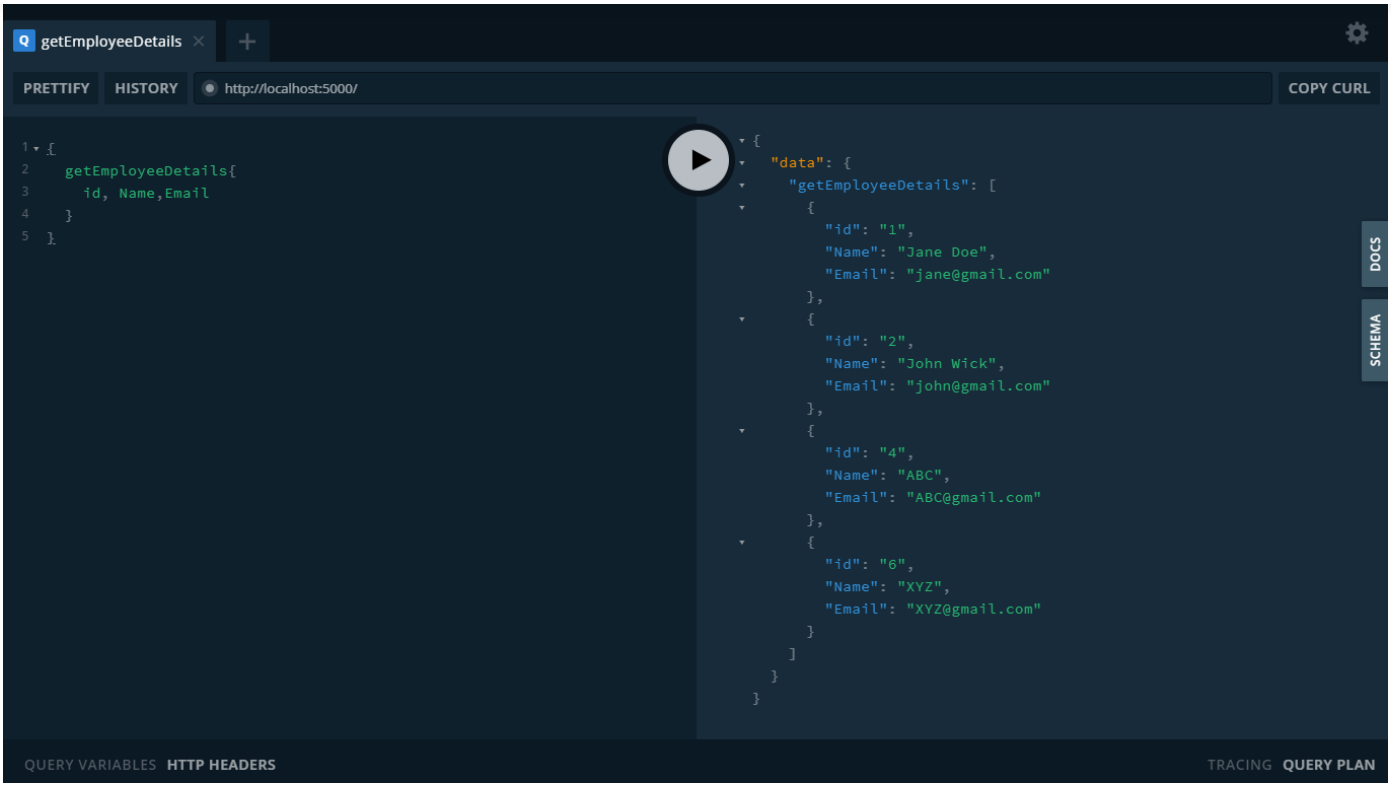
server.js file

Now time to run our server. To start our server go to the command prompt and hit **npm start**. This will be running on http://localhost:5000, and we will see GraphQL Playground running.

Let's try getting all employee detail:

```
{
  getEmployeeDetails{
    id,
    Name,
    Email
  }
}
```

We will see a result as below:



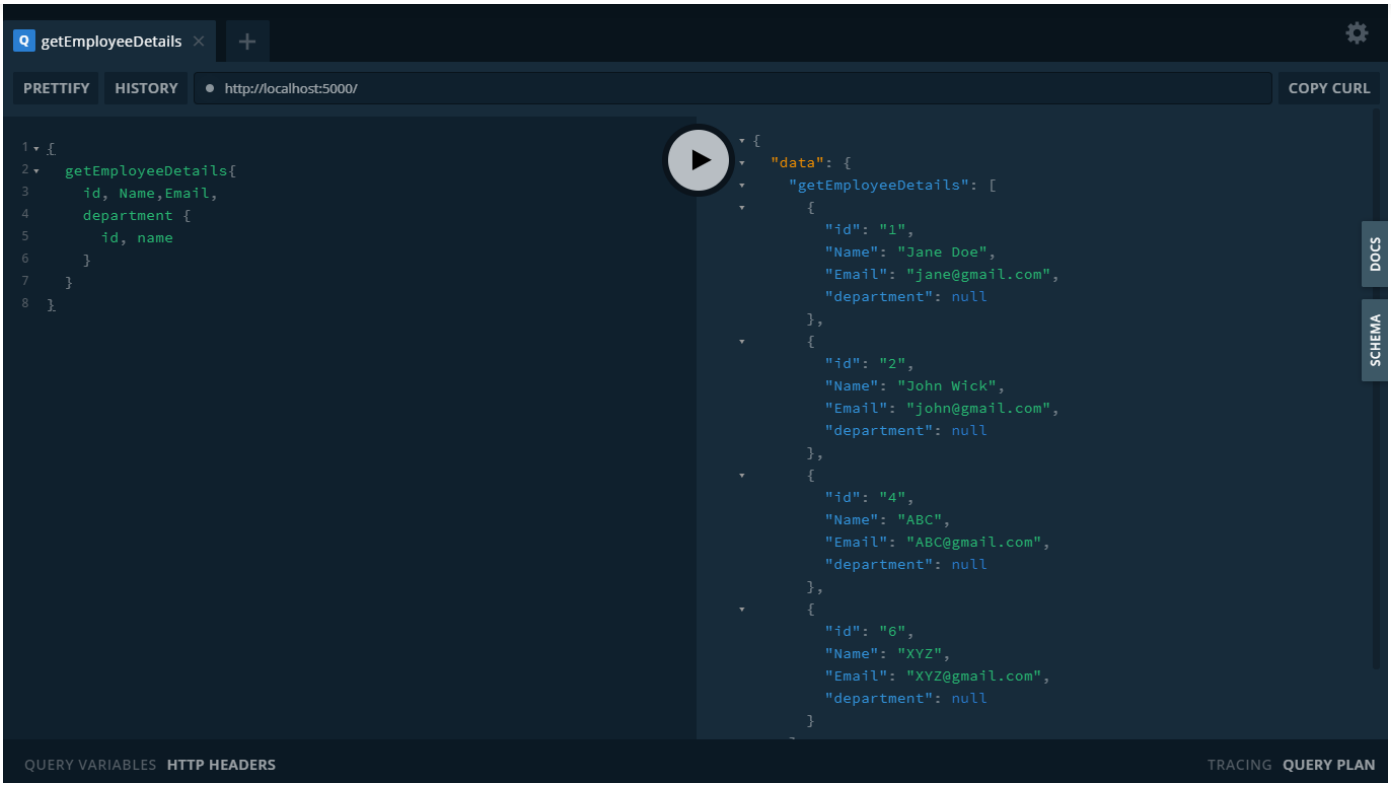
Here we are only fetching data from the employee table. What if we want to fetch data for employees from multiple tables? Yes, we can do this, as we have already defined the relationship. Now we just need to modify our schema for employee as well as we need to add a new type for the department, designation. Our modified schemas should look like this.

```
type Query {
  getEmployeeDetail(id: ID!): Employee
}

type Employee {
  id:ID!
  Name: String
  Email: String
  Designation_id: Int
  Manager_id: Int
  Department_id: Int,
  designation: Designation
  department: Department
  manager: Employee
}

type Department {
  id: ID!
  name: String
}

type Designation {
  id: ID!
  name: String
}
```

modified schema

here, we have added a designation field to employee and the type of this field is Designation. So in a custom type, like Employee, we can use another custom type like Designation, that's the way we represent the associations between different objects. Similarly, we can do it for Designation and Manager.

Now let us implement these associations in the resolvers. Actually, let's see what happens if we don't change our resolvers. We will query getEmployeeDetails again with id, Name, Email, department. Since the department is an object with fields, we need to specify



You can see that we get some data back, but since we didn't update our resolver the department in each employee is always null. So how can we return a department for employees?

We already have relationships defined in our database. We have department_id as a foreign key in the employee table. Here we can make use of department_id as a reference to fetch the department related data. As we already have declared in our schemas that every employee has a department field which is an object. Our resolver should reflect our schema.

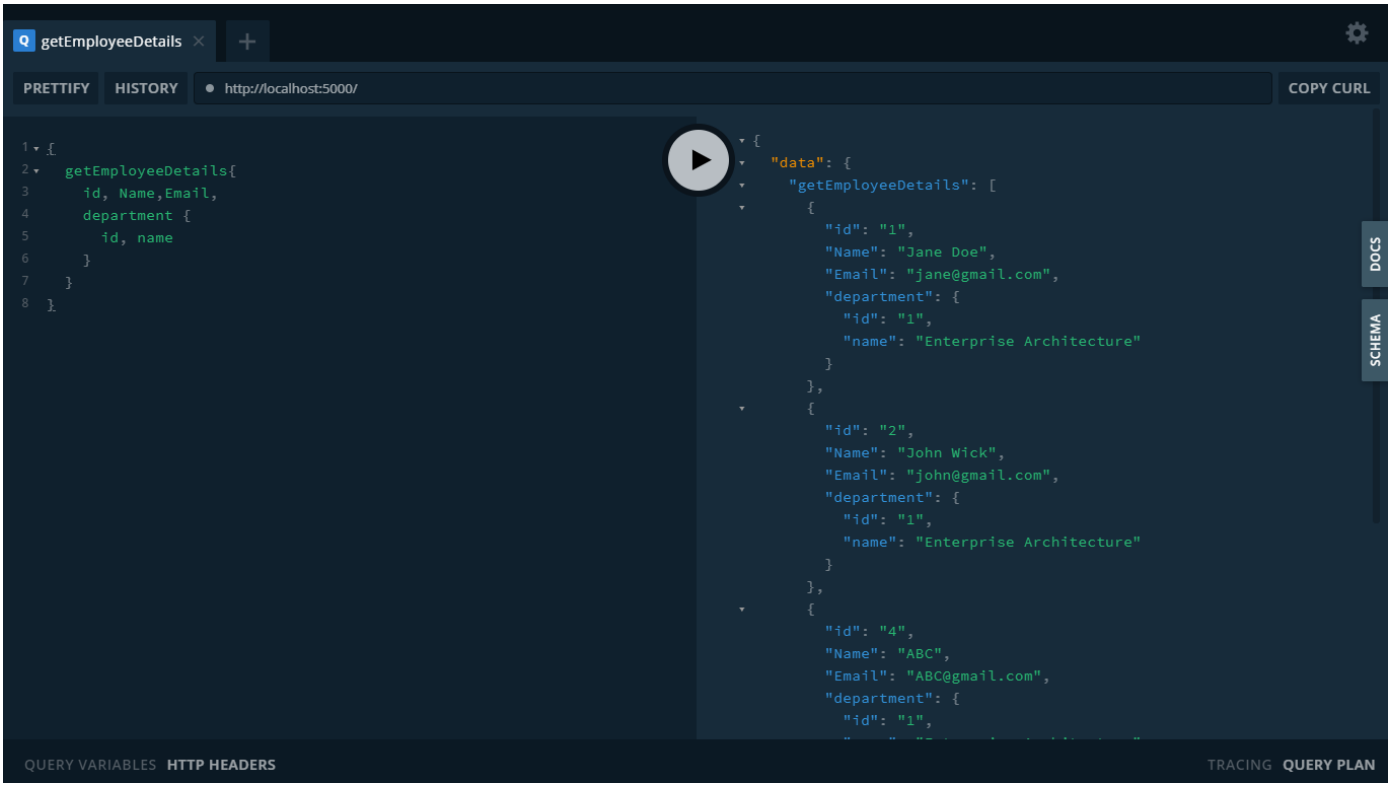
We need to declare a new resolver object for the department where we can put functions to resolve the fields of the department. Now how to return the right department from this function?

Here each resolver function receives some arguments and the argument is the parent object. In this case, since we are resolving the designation for the employee, the parent object is an emp and we want to return the associated department and add it to the

```
const Employee = {
  department: (emp) => department.findByPk(emp.Department_id)
}
```

So we are saying that return a department whose id is the same as that of department_id for the employee.

Now let us run the same query and let's check the output:



We ran the exact same query as that before. Now the department field is not null anymore, they are objects with id and name.

Similarly, you can do it for designations and manager.

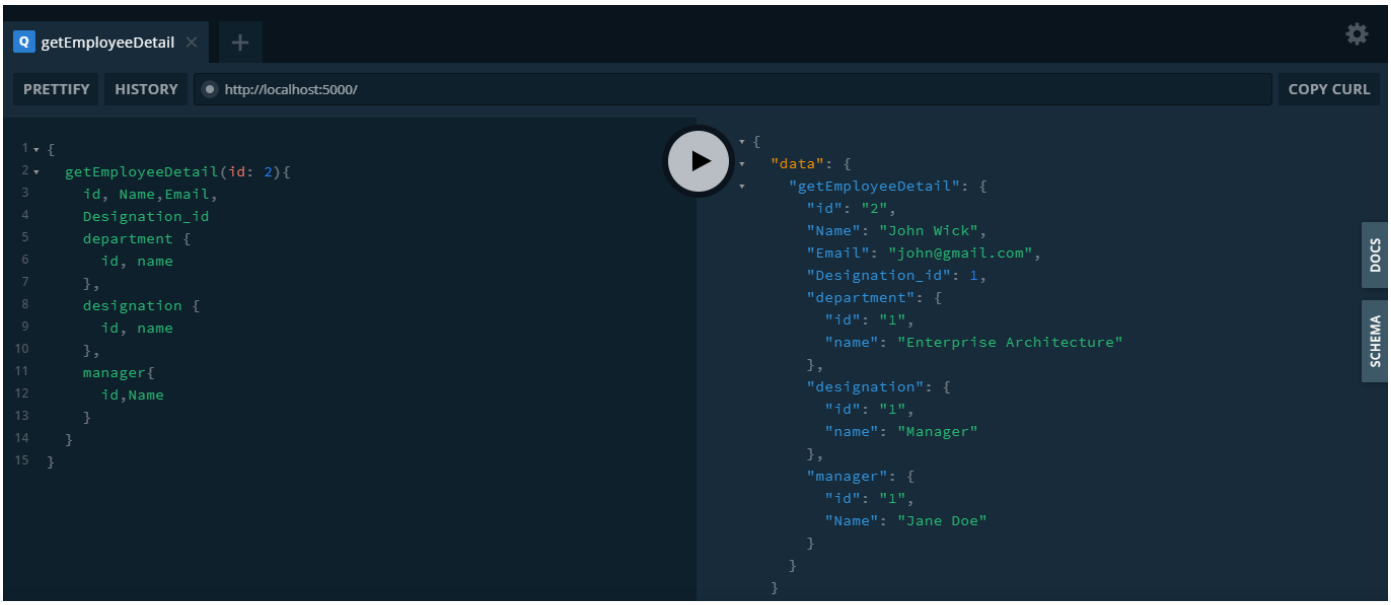
Here we are getting a list of all employees what if we want to get only specific employee details? To do that we need to add a new field to the query type, we just don't want any employee details to be returned, we want one with a specific id. In GraphQL we can pass

```
type Query {
  getEmployeeDetails: [Employee]
  getEmployeeDetail(id: ID!): Employee
}
```

Here exclamation (!) mark represents that the ID cannot be null. Now we need to write a resolver:

```
getEmployeeDetail: async (root, { id }) => {
  try {
    const emp = await employee.findByPk(id)
    return emp;
  } catch (err) {
    console.log(err);
  }
}
```

This function accepts two arguments where the root is the parent object and the second is id. Let's test this:



Here we get only a single employee object with id:2

Step 4 — Writing mutations.

In our graphql schemas, we only have queries, it is all about returning existing data, there is nothing that modifies the data.

In GraphQL the operations that modify the data are called **mutations**. They must be kept separated from the queries. We need to add a separate type called Mutation. The mutation is a root type just like the query. Inside mutation, we can find operations like create, update, and delete.

> ***Mutations*** *allow clients to manipulate data (i.e., create, update, or delete, similar to POST, PUT, or DELETE, respectively).*

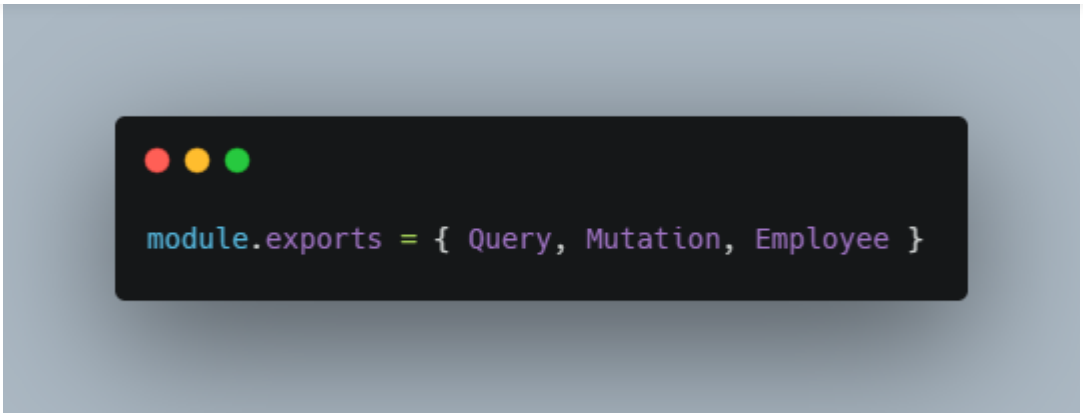
Below we have a mutation for adding a new employee, for that we need to pass the arguments for adding the employee. If you look at Employee type you have name, email, designation_id, department_id, manager_id. Here we don't have an id field because in the database we have set it to auto-increment.

```
type Mutation {
  createEmployee(
    Name: String
    Email: String
    Designation_id: Int
    Department_id: Int
    Manager_id: Int
  ): String
}
```

schema

Mutations must return a result just like queries. In our case, we'll return a String.

In resolvers we need to match the structure of schemas, so we need to define a new object



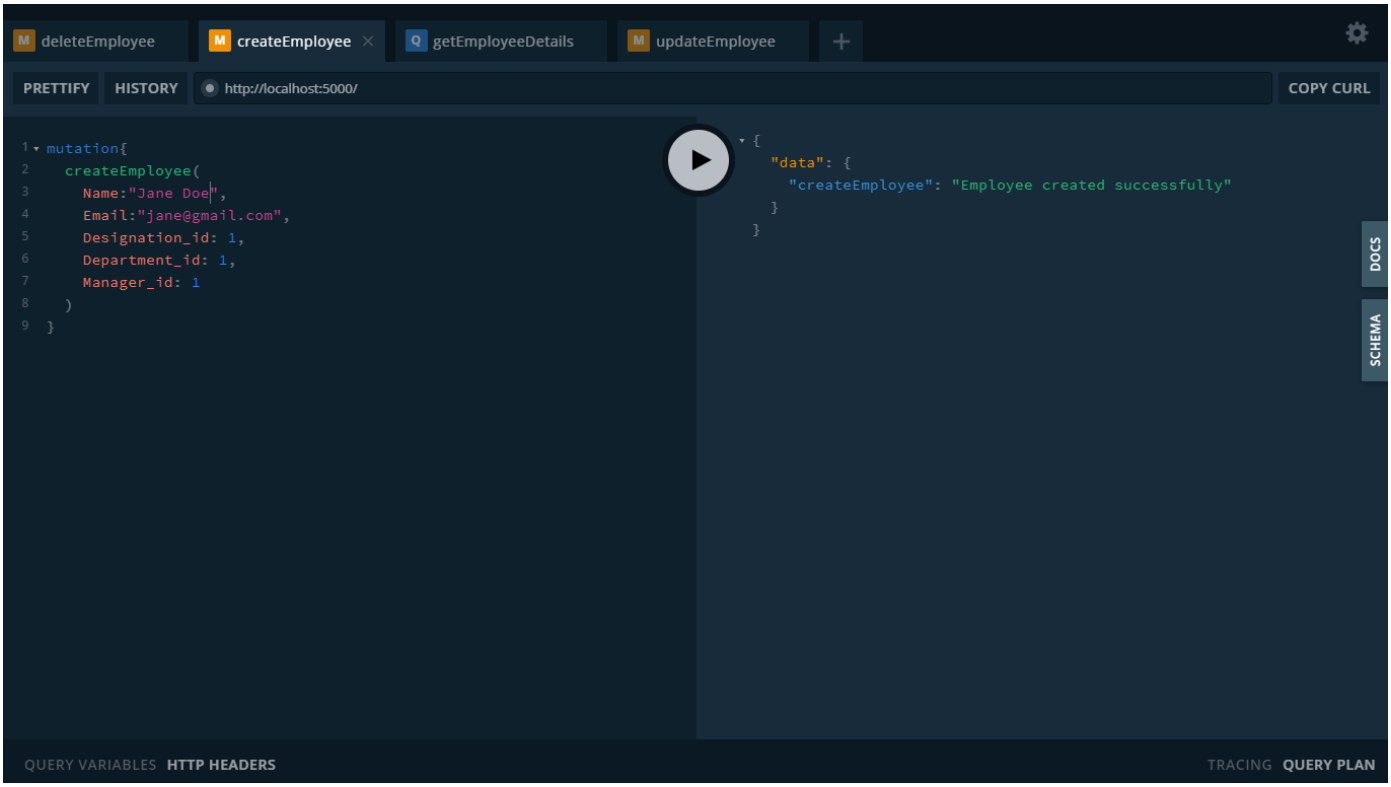
Now we are ready to create a mutation function for adding an employee.

```
const Mutation = {
  createEmployee: async (root, {
    Name,
    Email,
    Designation_id,
    Department_id,
    Manager_id
  }) => {
    try {
      await employee && employee.create({
        Name,
        Email,
        Designation_id,
        Department_id,
        Manager_id
      })
      return "Employee created successfully"
    } catch (err) {
      console.log(err)
    }
  }
}
```

mutation object to add a new employee

```
mutation{
  createEmployee(
    Name:"Jane Doe",
    Email:"jane@gmail.com",
    Designation_id: 1,
    Department_id: 1,
    Manager_id: 1
  )
}
```

The output will look like this:



Output for add employee.

Ours create statement works successfully. Similarly, we can update the employee. We need to update our schemas and resolvers.

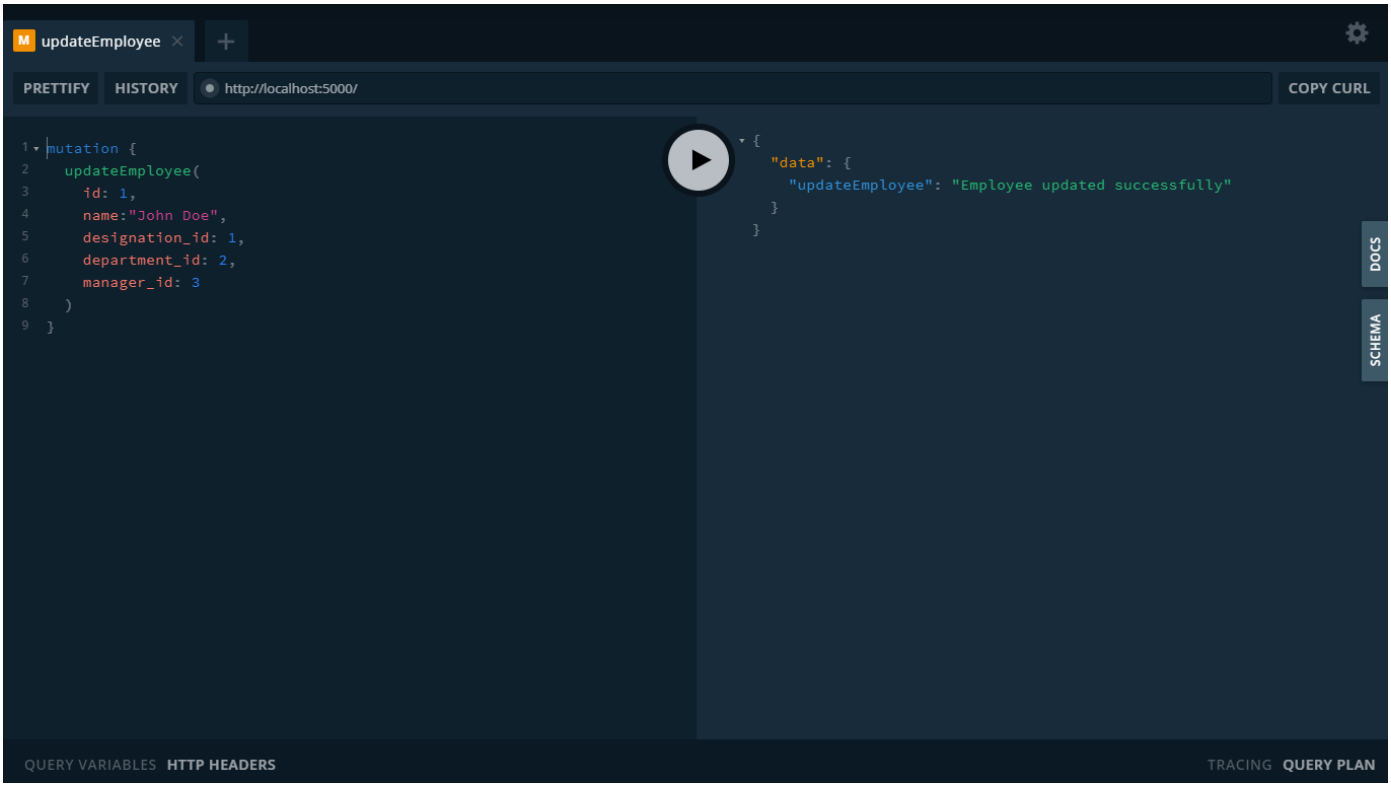
```
updateEmployee(
  Name: String
  Email: String
  Designation_id: Int
  Department_id: Int
  Manager_id: Int
): String
```

schema

```
updateEmployee: async (_, {
  Name,
  Email,
  Designation_id,
  Department_id,
  Manager_id
}) => {
  try {
    await employee && employee.update({
      Name,
      Email,
      Designation_id,
      Department_id,
      Manager_id
    }, { where: { id: id } });
    return "Employee updated successfully";
  } catch (err) {
    console.log(err)
  }
}
```

resolver function

Once we have code in place we can run the following :



Our update statement works successfully and employee with id 1 gets updated.

Now let us try to delete an employee. For this, we need to add one more function to mutations in schemas as well as resolvers.

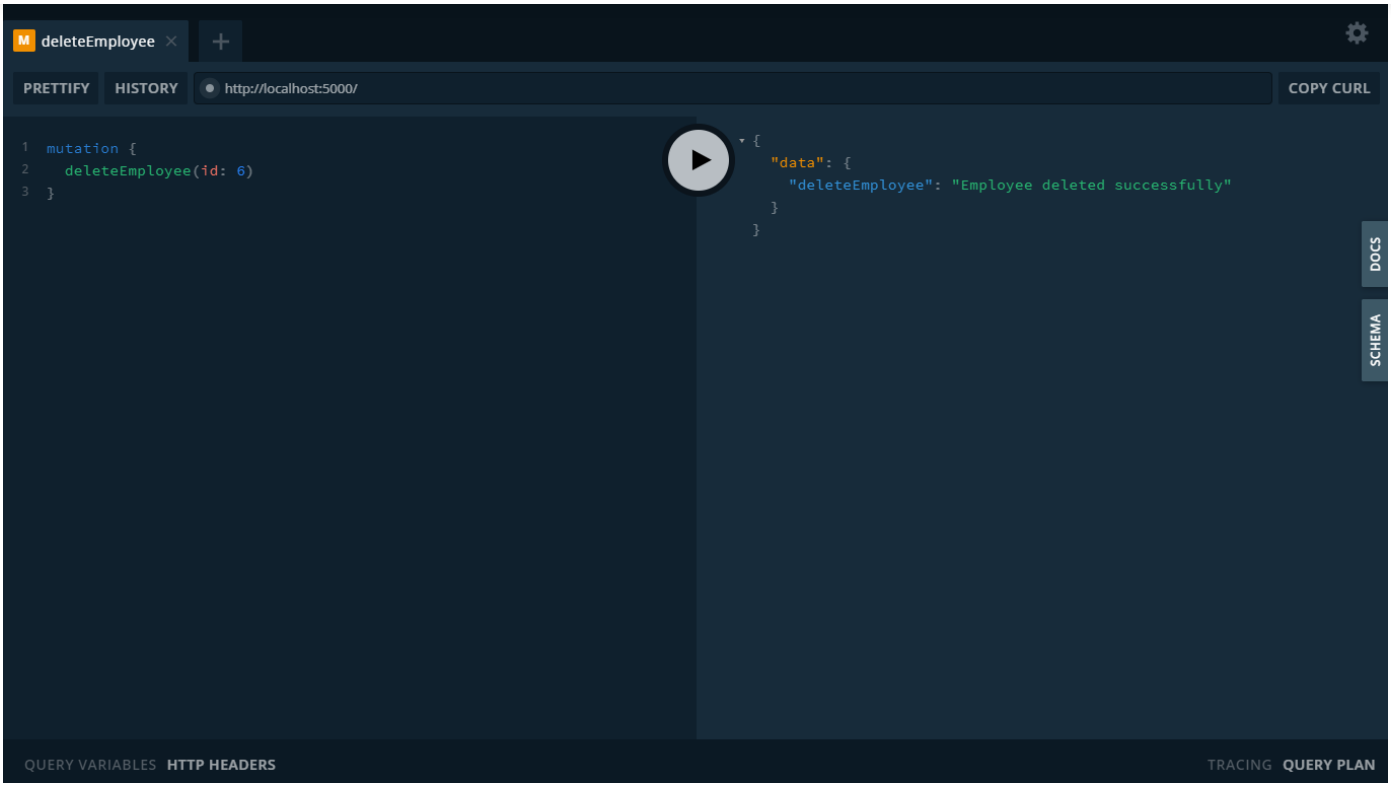
```
deleteEmployee(id: ID!) : String
```

schema

Here we are deleting an employee based on employee id and returning a string.

```
deleteEmployee: (root, { id }) => {
  employee.destroy({ where: { id: id } })
  return "Employee deleted successfully"
}
```

resolver function



The employee with a specific id is deleted successfully.

Conclusion

1. We saw how to create a GraphQL server in Node.js with Apollo Server.
2. We also saw how to integrate MySQL database with a GraphQL server using Sequelize.
3. This tutorial has covered the basics of CRUD operations.

The tutorial project is available on GitHub repo.