HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD

www.heig-vd.ch

Fullstack GraphQL

Building a GraphQL server

Daily Menu

- A minimal GraphQL server with Apollo
- Integrate Apollo server into an existing express app
- Building a demo project
  - Create a schema
  - Implement resolvers
  - Add mutations

What do I need to get started ?

You need to install two packages:

- Graphql – The JavaScript reference implementation for GraphQL
- Apollo server – A library based on the reference implementation that makes it easier to build GraphQL servers.

Create a Node.js project, then run the following command to install necessary packages:

```
$ yarn add apollo-server graphql
```

> Note: `yarn` is another package manager like `npm` that we will use for all future examples.

GraphQL hello world

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

const typeDefs = gql`
  type Query {
    message: String
  }
`;

const resolvers = {
  Query: {
    message: () => 'hello world'
  }
};

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

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

View Source

https://graphql-hello.glitch.me

The schema (Type definition)

Type definitions define the **shape** of your data and specify which ways the data can be fetched from the GraphQL server.

```
const typeDefs = gql`
  type Query {
    message: String
  }
`;
```

Resolvers

Resolvers define the technique for fetching the types in the schema.

```
const resolvers = {
  Query: {
    message: () => {
      return 'hello world';
    },
  },
};
```

ApolloServer

In the most basic sense, the ApolloServer can be started by passing type definitions (typeDefs) and the resolvers responsible for fetching the data for those types.

```
const server = new ApolloServer({
  typeDefs,
  resolvers,

  // Default configuration in development
  // introspection: true,
  // playground: true,
});
```

`playground` and `introspection` are enabled in development mode to let you easily test your server and integrate it with your code editor.

Check the ApolloServer API reference to see the parameters you can pass to the constructor.

Running the server

This `listen` method launches a web-server. Existing apps can utilize middleware options, this is exactly what we'll discuss next.

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

Integrating with an existing app

In the previous example we created a **standalone** GraphQL Server that only understand GraphQL queries.

In practice, you often need a server that does more — serving **HTML pages**, exposing a **REST** and **GraphQL** API.

In order to do that, you can use `ApolloServer` as a middleware to your app. If you use `express`, you'll have to install `apollo-server-express` instead of `apollo-server`.

```
$ yarn add apollo-server-express
```

Integrating with express

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

const app = express();

// Integrate apollo as a middleware
const server = new ApolloServer({...});
server.applyMiddleware({ app });

// Serve any static files
app.use(express.static(path.join(__dirname, 'public')));

// A REST endpoint
app.get('/api/hello', () => {
  res.send('hello from REST');
})

// Serve a single page app
app.get('/*', function(req, res) {
  res.sendFile(path.join(__dirname, 'public', 'index.html'));
});

// launch our express app
app.listen(5000);
```

Let's build a demo project

A simple app-store to review students projects

Steps

Today we'll only implement the GraphQL API. The steps are:

1. Defining a schema
2. Creating an in-memory database
3. Implement resolvers for queries
4. Implement resolvers for mutations

Defining a schema

First we need to define root queries. They are the entry points into our graph.

A user might use `applications` to get the list of all apps and `application(id: ID!)` to get details for a single app.

```
type Query {
  applications: [Application]
  application(id: ID!)
}
```

Type definitions – Application

```
type Application {
  id: ID!
  name: String
  description: String
  authors: [User]
  reviews: [Review]
  score: Int!
}
```

- In this type, we define relationship between `Application`, `User` and `Review`.
- An application can have multiple authors of type `User`
- An application also has multiple reviews made by other users.

> Note: Keep in mind that this schema doesn't represent how data are structured in our database. You **design a schema for human** developer and front-end designers. It must be designed in the most logical way.

Type definitions – User

```
type User {
  id: ID!
  name: String!
  reviews: [Review]
}
```

- To make things simple we only store the name of the user.
- It can also be useful to have all the reviews a single user has given.

Type definitions – Review

```
type Review {
  id: ID!
  stars: Int!
  author: User!
  comment: String
  application: Application
}
```

- A review has basic information: The number of stars and a comment
- We also need to reference the user who has given the review and the application that was reviewed.

> Note: If we were designing this schema for a database, we would have used ID references. Because **we design a flexible schema** for human, we use types (`User` , `Application`).

The final schema

```
// index.js
const typeDefs = gql`
  type Application {
    id: ID!
    name: String
    description: String
    authors: [User]
    reviews: [Review]
    score: Int!
  }

  type User {
    id: ID!
    name: String!
    reviews: [Review]
  }

  type Review {
    id: ID!
    stars: Int!
    author: User!
    comment: String
    application: Application
  }

  type Query {
    applications: [Application]
    application(id: ID!): Application
  }
`;
```

In-memory database

To keep things simple we'll store `apps`, `users` and `reviews` as javascript variables in our server. Our database might look like this:

```
// database.js
module.exports = {
  apps: [{
    id: "1",
    name: 'Retis',
    description: 'A social network to share samples of code between developers',
    authors: ["1", "2", "3"],
    tags: ['social', 'code'],
  }],
  reviews: [{
    id: "1",
    author: "5",
    comment: 'Nice app! But I cannot logout. Please help :(',
    applicationId: "1",
    stars: "3",
  }],
  users: [{
    id: "1",
    name: 'P-S. Rochat',
  }],
}
```

Resolvers

In order to respond to queries, a schema needs to have resolve functions for all fields. Each resolver represents a single field, and can be used to fetch data from any source(s) you may have.

```
const resolvers = {
  Query: {
    applications: () => {},
    application: () => {},
  },

  Application: {
    id: () => {},
    name: () => {},
    description: () => {},
    authors: () => {},
    reviews: () => {},
    score: () => {},
  },
};
```

```
type Query {
  applications: [Application]
  application(id: ID!): Application
}

type Application {
  id: ID!
  name: String
  description: String
  authors: [User]
  reviews: [Review]
  score: Int!
}
```

Resolver type signature

All resolver functions takes four arguments and may return a `Promise` or an `Object`. Here is the full function signature:

```
fieldName: (parent, args, context, info) => data
```

- **`parent`** – An object that contains the result returned from the resolver on the parent type
- **`args`** – An object that contains the arguments passed to the field. Ex: `{ query { application(id: "5") }}`
- **`context`** – An object shared by all resolvers in a GraphQL operation. We use the context to contain per-request state such as authentication information and access our data sources.
- **`info`** – Information about the execution state of the operation which should only be used in advanced cases

Resolvers basic usage

Here we use our in-memory database to retrieve data from resolvers. In practice you may fetch data from your database or from a REST API.

```
const { apps } = require('./database');

const resolvers = {
  Query: {
    applications: () => {
      return apps
    },
    application: (parent, args) => {
      return apps.find(app => app.id === args.id);
    },
  },
  ...
};
```

For the `application` resolver, we used `args` to access the query arguments and return a single app with the specified `id`.

Default resolvers and parent object

```
const { apps, users } = require('./database');

const resolvers = {
  Query: {
    applications: () => apps,
  },
  Application: {
    name: (parent) => parent.name,
    authors: (parent) {
      const userIds = parent.authors
      return users.filter(user => userIds.includes(user.id));
    }
  }
  ...
};
```

The `parent` parameter passed into our resolver function references an application. This parent object comes from the parent resolver, in this case it comes from `applications`.

- `name(parent)` - This resolver is not needed because it acts exactly like the **default resolver**.
- `authors(parent)` – This resolver is required in order to return a list of `User` instead of a list of IDs.

Let's practice with resolvers

Implement missing resolvers to make our app work correctly:
https://glitch.com/edit/#!/heig-labs

Mutations schema

We define mutations in the schema like we did for queries. Mutations generally takes arguments that can be either scalar types or input types.

```
input ReviewInput {
  stars: Int,
  comment: String!
  authorId: ID,
  applicationId: ID,
}

type Query {...}

type Mutation {
  addReview(data: ReviewInput!): Review
}
```

- We defined `ReviewInput` as an `input` type to create reviews.
- We defined `addReview` a query that takes a single argument and returns the created element.

Mutation resolvers

As we did for query resolvers, we use the second parameter to access arguments.

```
const resolvers =  {
  Query: {...}
  Mutation: {
    addReview: (_, { data }) => {
      const review = {
        id: String(reviews.length),
        stars: data.stars,
        comment: data.comment,
        author: data.authorId,
        applicationId: data.applicationId,
      };
      reviews.push(review);
      return review;
    },
  },
}
```

In general mutations always returns the updated/created element in order to let the client app update its local state correctly.

Sending Mutation

Here is the query you could use to create a review (from your client app or the GraphQL playground). The response of this query is an object with the fields: `id`, `stars` and `comment`.

```
mutation AddReview($data: ReviewInput!) {
  addReview(data: $data) {
    id
    stars
    comment
  }
}
```

The variable `$data` defined by the query should match the type `ReviewInput`.

```
{
  "data": {
    "stars": 5,
    "authorId": "1",
    "applicationId": "1",
    "comment": "Just amazing this app changed my life!"
  }
}
```

Defining the context

Apollo server let define a function called with the current request that creates the context shared across all resolvers.

Here is an example of how you could use the `context` for **authentication**.

```
const server = new ApolloServer({
 typeDefs,
 resolvers,
 context: ({ req }) => {
   // get the user token from the headers
   const token = req.headers.authorization || '';

   // try to retrieve a user with the token
   const user = getUser(token);

   // add the user to the context
   return { user };
 },
});
```

https://www.apollographql.com/docs/apollo-server/v2/features/authentication.html#context

Using the context

When the context is defined, a user may or may not exist. You can access the `context` from any resolver to do **authorization**.

```
const resolvers = {
  Query:{
    protectedResource: (parent, args, context) {
      if (!context.user) {
        // Permission denied (return null or throw an Error)
        return null
      }
      // All right! Access and return the protected resource
    }
  }
}
```

Resources

We have seen core concepts of ApolloServer and GraphQL - Schemas, Resolvers, Queries and Mutations.

The next step is to start implementing your app and use the following resources to learn more:

- https://graphql.org/learn/ – Learn the GraphQL language in general. How to define types, queries and mutations.
- https://www.apollographql.com/docs/apollo-server/ - Learn more features and best practices when building a GraphQL server with Apollo.
- Apollo GraphQL - A VSCode extension that provides rich editor support for GraphQL client and server development
- https://github.com/heig-vd-tweb/heig-labs - A web app for reviewing students projects (work in progress)