

Never miss an article about web development, JavaScript and self-growth.

TAKE PART


SUBSCRIBE

GraphQL Server Tutorial with Apollo Server and Express

 Follow (<https://www.facebook.com/rwieruch>)  Follow  (https://twitter.com/intent/follow?screen_name=rwieruch)  Follow  (<https://github.com/rwieruch>)

AUGUST 22, 2018

 (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

 (<https://twitter.com/intent/tweet?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20tutorial%20by%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

This tutorial is part 4 of 4 in this series.

 (<http://twitter.com/robertmiller>)

Since GraphQL is only a query language, no one defines the transport layer or data format. GraphQL itself isn't opinionated about it. However, most often GraphQL is used as alternative to the popular REST <https://www.apolloserver.com/tutorial%2f> <https://www.robinwieruch.de%2fgraphql-by%40wieruch%23ReactJs%2f> communication over HTTP with JSON <https://www.robinwieruch.de%2f>

In the following, you are going to implement the server-side of such architecture by using GraphQL and Apollo Server. Whereas GraphQL is only the query language which got implemented as reference implementation in JavaScript by Facebook, Apollo Server builds up on top of it to simplify building GraphQL servers in JavaScript.

In the end, you will have a fully working GraphQL server boilerplate project which implements authentication, authorization, a data access layer with a database, domain specific entities such as users and messages (it could be the beginning of a chat application), different pagination strategies, and real-time abilities due to subscriptions. You can find a working solution of it (plus a working client-side application in React) in this GitHub repository: Full-stack Apollo with React and Express Boilerplate Project (<https://github.com/rwieruch/fullstack-apollo-react-express-boilerplate-project>). It can be your perfect starter project to realize your own idea.

While building this application with me in the following sections, I recommend to verify your implementations with the built-in GraphQL client application (e.g. GraphQL Playground). Once you have your database setup done, you can verify your stored data over there as well. In addition, if you feel comfortable with it, you can implement a client application (in React or something else) which consumes the GraphQL API of this server. So let's get started!

Table of Contents

- Apollo Server Setup with Express
- Type Definitions
- Resolvers
- Type Relationships
- Queries and Mutations
- GraphQL Schema Stitching with Apollo Server
 - Technical Separation
 - Domain Separation
- PostgreSQL with Sequelize for a GraphQL Server
- Connecting Resolvers and Database
- Validation and Errors
- Authentication

- Registration (Sign Up) with GraphQL

f ([https://www.facebook.com/sharer](https://www.facebook.com/sharer.php?u=https%3A%2F%2Fwww.robinwieruch.de%2Fgraphql-apollo-server-tutorial%2F)

- Securing Passwords with Bcrypt

[/share?text=GraphQL%20Server%20Tutorial](https://www.facebook.com/sharer.php?u=https%3A%2F%2Fwww.robinwieruch.de%2Fgraphql-apollo-server-tutorial%2F)

t (<https://twitter.com/rwieruch/status/1048444444444444444>)

w ([https://www.robinwieruch.de/graphql-](https://www.robinwieruch.de/graphql-apollo-server-tutorial/)

- Authorization with GraphQL and Apollo Server

by %40rwieruch %23ReactJs'

%2Fwww.robinwieruch.de%2F

- GraphQL Authorization on a Resolver Level
- Permission-based GraphQL Authorization
- Role-based GraphQL Authorization
- Setting Headers in GraphQL Playground
- Pagination in GraphQL with Apollo Server
 - Offset/Limit Pagination with Apollo Server and GraphQL
 - Cursor-based Pagination with Apollo Server and GraphQL
 - Cursor-based Pagination: Page Info, Connections and Hashes
- GraphQL Subscriptions
 - Apollo Server Subscription Setup
 - Subscribing and Publishing with PubSub
- Testing a GraphQL Server
 - GraphQL Server E2E Test Setup
 - Testing User Scenarios with E2E Tests
- Batching and Caching in GraphQL with Data Loader
- GraphQL Server + PostgreSQL Deployment to Heroku

Apollo Server Setup with Express

Apollo Server can be used with several popular libraries for Node.js: Express, Koa, Hapi. Apollo itself is kept library agnostic, so it is possible to connect it with a lot of third-party libraries in client but also server applications. In this application, you will use Express (<https://expressjs.com/>), because it is the most popular and commonly used middleware library for Node.js. So let's install these two dependencies to the *package.json* file and *node_modules* folder:

```
npm install apollo-server apollo-server-express --save
```

As you can see by the library names, you can use any other middleware solution (e.g. Koa, Hapi) for complementing your standalone Apollo Server. Apart from these libraries for Apollo Server, you will need the core libraries for Express and GraphQL:

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

Now every library is in place to get started with the source code in the *src/index.js* file. First, you have to import the necessary parts for getting started with Apollo Server in Express.

[f \(https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f\)](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)
[t \(https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2f\)](https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2f)

```
import express from 'express';
import { ApolloServer } from 'apollo-server-express';
```

And second, you can use both imports for initializing your Apollo Server with Express:

```
import express from 'express';
import { ApolloServer } from 'apollo-server-express';

const app = express();

const schema = ...
const resolvers = ...


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

server.applyMiddleware({ app, path: '/graphql' });

app.listen({ port: 8000 }, () => {
  console.log('Apollo Server on http://localhost:8000/graphql (http://localhost:8000/graphql)')
});
```

By using Apollo Server's `applyMiddleware()` method, you can opt-in any middleware (here Express). Furthermore, you can specify the path for your GraphQL API endpoint. Apart from this, you can see how the Express application gets initialized. The only thing missing is the definition for the schema and resolvers for creating the Apollo Server instance. Let's implement them first and learn about them afterward:

 (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

 (<https://twitter.com/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%23ReactJs%2fwww.robinwieruch.de%2f>)

```

import express from 'express';
import { ApolloServer, gql } from 'apollo-server-express';

const app = express();

const schema = gql`
  type Query {
    me: User
  }

  type User {
    username: String!
  }
`;

const resolvers = {
  Query: {
    me: () => {
      return {
        username: 'Robin Wieruch',
      };
    },
  },
};

...

```

The **GraphQL schema** provided to the Apollo Server is all the available data for reading (and writing) data via GraphQL. It can happen from any client who consumes the GraphQL API. The schema consists of **type definitions**, starting with a mandatory top level **Query type** for reading data, and then followed by **fields** and **nested fields**. In the schema from the Apollo Server setup, you have defined a `me` field which is of the **object type** `User`. In this case, a `User` type has only a `username` field, which is a **scalar type**. There are various scalar types in the GraphQL specification for defining strings (`String`), booleans (`Boolean`), integers (`Int`) and more. At some point, basically the whole schema has to end at its leaf nodes with scalar types in order to resolve everything properly. Being equipped with your JavaScript knowledge, think about it as analogy to a JavaScript object, which has JavaScript objects or arrays inside, but at some point it has to have primitives such as strings, booleans or integers.

```

const data = {
  me: {
    username: 'Robin Wieruch',
  },
};

```

[f \(https://www.facebook.com/sharer](https://www.facebook.com/sharer.php?u=https%3A%2F%2Fwww.robinwieruch.de%2Fgraphql-apollo-server-tutorial%2F)

[🐦 \(http](https://twitter.com/rwieruch)

The counterpart of the GraphQL schema for setting up a Apollo Server are the **resolvers** which are used to return data for your fields from the schema. The data source doesn't matter, because the data can be hardcoded (as it is at the moment), can come from a database, or from another REST API endpoint. You will learn about potential data sources later. For now, it only matters that the resolvers

are agnostic from where the data comes from. That's why GraphQL shouldn't be mistaken for a database query language. Resolvers are only functions which resolve data for your GraphQL fields in the schema. In the previous example, only a user object with the username "Robin Wieruch" gets resolved from the `me` field.

Your GraphQL API with Apollo Server and Express should be working now. On the command line, you can always start your application with the `npm start` script for verifying that it is working for you after you have changed something. But how do you verify it without having a client application yet? Apollo Server comes with a so called GraphQL Playground as built-in client for consuming your GraphQL API. After you have started your application on the command line, you should find GraphQL Playground by using your GraphQL API endpoint in the browser. Try to open `http://localhost:8000/graphql` (`http://localhost:8000/graphql`) and see if GraphQL Playground opens for you. In the application, you can define your first GraphQL query and see the result for it:

```
{
  me {
    username
  }
}
```

The result for the query should be this or your defined sample data:

```
{
  "data": {
    "me": {
      "username": "Robin Wieruch"
    }
  }
}
```

In the following sections, I might not mention all the time the GraphQL Playground, but it is up to you to verify your GraphQL API with it after you have changed something or implemented a new feature. You should always experiment it for to explore your own API.

Optionally you can also add CORS (<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>) to your Express middleware. First, install CORS on the command line:

```
npm install cors --save
```

And second, use it in your Express middleware:

```
%2fwww.robinwieruch.de%2fgraphql-
apollo-server-tutorial%2f)
```

Twitter (<https://twitter.com/rwieruch>)

/share?text=GraphQL%20Server%20Tutorial

by %40rwieruch %23ReactJs'

%2fwww.robinwieruch.de%2f

```
import cors from 'cors';
import express from 'express';
import { ApolloServer, gql } from 'apollo-server-express';

const app = express();

app.use(cors());

...
```

CORS is needed when performing HTTP request from another domain than your server domain to your server. Otherwise you may run into cross-origin resource sharing errors for your GraphQL server.

Exercises:

- read more about GraphQL (<https://graphql.org/learn>)
- play around with the schema and the resolver
 - add more fields to the user type
 - fulfil the requirements in the resolver
 - query your fields in the GraphQL Playground
- read more about Apollo Server Standalone (<https://www.apollographql.com/docs/apollo-server/v2/getting-started.html>)
- read more about Apollo Server in Express Setup (<https://www.apollographql.com/docs/apollo-server/v2/essentials/server.html>)

Apollo Server: Type Definitions

This section is all about GraphQL type definitions and how they are used to define the overall GraphQL schema. A GraphQL schema is defined by its types, the relationships between the types, and their structure. Therefore GraphQL uses a **Schema Definition Language (SDL)**. However, the schema doesn't define where the data comes from. This responsibility is outside of the SDL and as you have witnessed is performed by resolvers. When you have used Apollo Server previously, you have used a User object type within the schema and defined a resolver which returned a user for the corresponding `me` field.

In the previous implementation, perhaps you have noticed the exclamation point for the `username` field in the User object type. It means that the `username` is a **non-nullable** field. So whenever a field of type User with a `username` is returned from the GraphQL schema, the user has to have a `username`. It cannot be undefined or null. However, there isn't an exclamation point for the `me` field of the `me` field can be `null`. This is because the `me` field can be `null` if the user is not logged in. For this particular scenario, there shouldn't be always a user returned for the `me` field, because after

all, a server has to know *who is me* before it can give any response. Later you will implement an authentication mechanism (sign up, sign in, sign out) with your GraphQL server. Only when a user (most likely you) is authenticated with the server, the `me` field is populated with a user object (maybe your account details). Otherwise it stays null.

As you can see, while you define your GraphQL type definitions, you have to make conscious decisions about the types, their relationships, their structure and their (non-null) fields. Now let's extend the schema by extending or adding more type definitions to it. What about querying any user with a GraphQL client?

```
const schema = gql`
  type Query {
    me: User
    user(id: ID!): User
  }

  type User {
    username: String!
  }
`;
```

That's when **GraphQL arguments** come into play. They can be used to make more fine-grained queries, because you can provide them to the GraphQL query. See how the arguments can be used on a per field level by using parentheses. And also for the arguments you have to define the type. In this case, it is a non-nullable identifier to retrieve the correct user from a data source eventually. Furthermore, the query returns the User type, but it can be null, because sometimes a user entity might not be found in the data source when providing a non identifiable `id` for it. Now you can see how two queries already share the same GraphQL type and thus when adding fields to the User type, a client can use these fields for both queries when querying an implicit User object type.

There is already one more field which could be added to the User type. It is the `id` field, because after all a user should have an `id` when it is already possible to query a user by its `id`.

```
const schema = gql`
  type Query {
    me: User
    user(id: ID!): User
  }
```

```
  type User {
    id: ID!
    username: String!
```

[f \(https://www.facebook.com/sharer](https://www.facebook.com/sharer)

[t \(http](https://twitter.com/sharer)

[/sharer.php?u=https%3a%2f](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f)

[/share?text=GraphQL%20Server%20Tutorial](https://twitter.com/sharer/share?text=GraphQL%20Server%20Tutorial)

[%2fwww.robinwieruch.de%2fgraphql-](https://www.robinwieruch.de/graphql-apollo-server-tutorial/)

[by %40rwieruch %23ReactJs'](https://www.robinwieruch.de/graphql-apollo-server-tutorial/)

[apollo-server-tutorial%2f\)](https://www.robinwieruch.de/graphql-apollo-server-tutorial/)

[%2fwww.robinwieruch.de%2f](https://www.robinwieruch.de/graphql-apollo-server-tutorial/)

You may be wondering about the ID scalar type at this point. The ID denotes an identifier which can

be used internally for advanced features such as caching or refetching later on. It is a superior string scalar type.

So what's missing for the new GraphQL query is the resolver. In the first step, you can add it to your map of resolvers with sample data again:

```
const resolvers = {
  Query: {
    me: () => {
      return {
        username: 'Robin Wieruch',
      };
    },
    user: () => {
      return {
        username: 'Dave Davids',
      };
    },
  },
};
```

Second, you may want to make use of the incoming `id` argument from the GraphQL query to make your decision of which user you are going to return. All the arguments can be found in the second argument in the resolver function's signature:

```
const resolvers = {
  Query: {
    me: () => {
      return {
        username: 'Robin Wieruch',
      };
    },
    user: (parent, args) => {
      return {
        username: 'Dave Davids',
      };
    },
  },
};
```

You may have noticed the first argument called `parent` as well. For now, you don't need to worry about it. Later, it will be showcased where it could be used potentially in your resolvers. Now, to make the example more realistic, you can extract a map of sample users and return a user based on the `id` which is used as a key in the extracted map:

🔗 (<https://www.facebook.com/sharer>

🐦 (<https://twitter.com/sharer>

[/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

[/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20%2fwww.robinwieruch.de%2f](https://twitter.com/sharer/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20%2fwww.robinwieruch.de%2f)

```

let users = {
  1: {
    id: '1',
    username: 'Robin Wieruch',
  },
  2: {
    id: '2',
    username: 'Dave Davids',
  },
};

const me = users[1];

const resolvers = {
  Query: {
    user: (parent, { id }) => {
      return users[id];
    },
    me: () => {
      return me;
    },
  },
};

```

Now try out your queries, even when using only this combined GraphQL query, in GraphQL Playground:

```

{
  user(id: "2") {
    username
  }
  me {
    username
  }
}

```


It should return this result:

```

{
  "data": {
    "user": {
      "username": "Dave Davids"
    },
    "me": {
      "username": "Robin Wieruch"
    }
  }
}

```

 (<https://www.facebook.com/sharer>

 (<https://twitter.com/sharer>

[/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

[/share?text=GraphQL%20Server%20Tutorial](https://twitter.com/sharer/share?text=GraphQL%20Server%20Tutorial%20by%20Robin%20Wieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

least but not least, what about querying a list of users? It would be your third query. First, you would only need to add the query again to the schema:

```
const schema = gql`
  type Query {
    users: [User!]
    user(id: ID!): User
    me: User
  }

  type User {
    id: ID!
    username: String!
  }
`;
```

In this case, the `users` field would return a list of users of type `User` which is denoted with the square brackets. Within the list no user is allowed to be null, but the list itself can be null in case there are no users in the first place (otherwise it could be also `[User!]!`). Once you add a new query to your schema, you are obligated to define it in your resolvers within the `Query` object:

```
const resolvers = {
  Query: {
    users: () => {
      return Object.values(users);
    },
    user: (parent, { id }) => {
      return users[id];
    },
    me: () => {
      return me;
    },
  },
};
```

Now you have already three queries which can be used in your GraphQL client (e.g. GraphQL Playground) applications. All of them operate on the same `User` type to fulfil the data requirements in the resolvers. Thus each query has to have a matching resolver. Did you notice that all queries are grouped under your one unique yet mandatory `Query` type? Basically the `Query` type lists all your available GraphQL queries which are exposed to your clients as your GraphQL API for reading data. In a later section, you will read about the `Mutation` type for grouping your GraphQL API for writing data.

Exercises:

- read more about the GraphQL schema with Apollo Server (<https://www.apollographql.com/docs/apollo-server/v2/essentials/schema.html>)
- read more about the GraphQL mindset: Thinking in Graphs (<https://graphql.github.io/learn/thinking-in-graphs/>)
- apollo-server-tutorial (https://blog.apollographql.com/apollo-server-tutorial/) <https://www.robinwieruch.de/graphql-apollo-server-tutorial/>

graphql-2254f84c4ed7)

Apollo Server: Resolvers


This section continuous with the GraphQL schema in Apollo Server, but transitions more to the resolver side of the subject. In your GraphQL type definitions you have defined types, their relations and their structure. But there is nothing about how to get the data. That's where the GraphQL resolvers come into play.

In JavaScript, the resolvers are grouped in a JavaScript object which is often called **resolver map**. As you have noticed, each top level query in your Query type has to have a resolver. But that's not everything to it. Let's see how you can resolve things on a per field level.

```
const resolvers = {
  Query: {
    users: () => {
      return Object.values(users);
    },
    user: (parent, { id }) => {
      return users[id];
    },
    me: () => {
      return me;
    },
  },
  User: {
    username: () => 'Hans',
  },
};
```

Once you start your application again and query for a list of users, every user should have the identical username.

 (https://www.facebook.com/sharer
/sharer.php?u=https%3a%2f
%2fwww.robinwieruch.de%2fgraphql-
apollo-server-tutorial%2f)

 (http
/share?text=GraphQL%20Server%20Tutorial
by %40rwieruch %23ReactJs'
%2fwww.robinwieruch.de%2f

```
// query
{
  users {
    username
    id
  }
}

// query result
{
  "data": {
    "users": [
      {
        "username": "Hans",
        "id": "1"
      },
      {
        "username": "Hans",
        "id": "2"
      }
    ]
  }
}
```

The GraphQL resolvers can operate fine-granular on a per field level. As you have noticed, you can override the username of every User type by resolving a `username` field. Otherwise the default `username` property of the user entity is taken for it. Generally this applies to every field. Either you decide specifically what the field should return in a resolver function or GraphQL tries to fallback for the field by retrieving the property automatically from the JavaScript entity.

Let's evolve this a bit more by diving into the function signature of resolver functions. Previously you have seen that the second argument of the resolver function is the incoming arguments of a query. That's how you were able to retrieve the `id` argument for the user from the Query. What about the first argument though? It's called the parent or root argument and always returns the previously resolved field. Let's check this for the new username resolver function.

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/rwieruch/status/1038441234567890123>)

```
const resolvers = {
  Query: {
    users: () => {
      return Object.values(users);
    },
    user: (parent, { id }) => {
      return users[id];
    },
    me: () => {
      return me;
    },
  },
  User: {
    username: parent => {
      return parent.username;
    }
  },
};
```

When you query again your list of users in a running application, all usernames should be alright again. That's because GraphQL firstly resolves all users in the `users` resolver and then goes through the `User's username` resolver for each user. Each user is accessible as the first argument in the resolver function and thus can be used to access further properties on the entity. You can rename your parent argument to make it more explicit:

```
const resolvers = {
  Query: {
    ...
  },
  User: {
    username: user => {
      return user.username;
    }
  },
};
```

In this case, the `username` resolver function is redundant, because it only mimics the default behavior of a GraphQL resolver. If you would leave it out, the user's username would still be resolved with its correct property. However, this fine-grained control over the resolved fields opens up powerful possibilities. It gives you the flexibility to add your own data mapping without worrying about your data sources behind the GraphQL layer. For instance, what about exposing the full username of a user which is only a combination of its first and last name by using template literals?

f (<https://www.facebook.com/sharer>

t (<https://twitter.com/sharer>

[/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

[/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs'%2fwww.robinwieruch.de%2f](https://twitter.com/sharer/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs'%2fwww.robinwieruch.de%2f)

```
const resolvers = {
  ...

  User: {
    username: user => `${user.firstname} ${user.lastname}`,
  },
};
```

For now, we are going to leave out the `username` resolver, because it mimics only the default behavior when using Apollo Server. These are called **default resolvers**, because they work for you under the hood without you having to define them explicitly.

Next, what about the other arguments in the function signature of a GraphQL resolver?


```
(parent, args, context, info) => { ... }
```

The `context` argument is the third argument in the resolver function which is used to inject dependencies from the outside to the resolver function. For instance, let's say the signed in user is known to the outside world of your GraphQL layer (because a request to your GraphQL server is made and the authenticated user is retrieved from somewhere else). You may want to inject this signed in user to your resolvers to do something with it. Let's do it with the `me` user for the `me` field. Remove the declaration of the `me` user (`let me = ...`) and pass it instead in the context object when Apollo Server gets initialized:

```
const server = new ApolloServer({
  typeDefs: schema,
  resolvers,
  context: {
    me: users[1],
  },
});
```

Next you can access it in the resolver's function signature as third argument which gets destructured into the `me` property from the context object.

 (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

 (<https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)


```
const resolvers = {
  Query: {
    users: () => {
      return Object.values(users);
    },
    user: (parent, { id }) => {
      return users[id];
    },
    me: (parent, args, { me }) => {
      return me;
    },
  },
};
```

The context should be the same for all resolvers now. Every resolver who needs to access the context, or in this case the `me` user, can do so by using the third argument of the resolver function.

The fourth argument in a resolver function, the `info` argument, isn't used very often, because it only gives you internal information about the GraphQL request. It can be used for debugging, error handling, and advanced monitoring and tracking. You don't need to worry about it for now.

On the side, a couple of words about the a resolver's return values: As you have witnessed, a resolver can return arrays, objects and scalar types, but it has to be defined in the matching type definitions too. For instance, the type definition has to define an array or non-nullable field in order to have the resolvers working appropriately. What about JavaScript promises? Often you will make a request to a data source (database, RESTful API) in a resolver and thus return a JavaScript promise in the resolver. GraphQL can deal with it and waits for the promise being resolved. Only then the result is mapped to the type definitions. That's why you don't need to worry about asynchronous requests to your data source later on.

Exercises:

- read more about GraphQL resolvers in Apollo (<https://www.apollographql.com/docs/apollo-server/v2/essentials/data.html>)

Apollo Server: Type Relationships

In the previous sections, you have started to evolve your GraphQL schema by defining queries, mutations, and type definitions. In this section, let's add a second GraphQL type called `Message` and see how it behaves in addition to your `User` type. In this application, a user can have messages. Basic level queries and the new `Message` type to your GraphQL schema:

<https://www.robinwieruch.de/graphql-apollo-server-tutorial/>

<https://www.robinwieruch.de/graphql-apollo-server-tutorial/>

```
const schema = gql`
  type Query {
    users: [User!]
    user(id: ID!): User
    me: User

    messages: [Message!]!
    message(id: ID!): Message!
  }


  type User {
    id: ID!
    username: String!
  }


  type Message {
    id: ID!
    text: String!
  }
`;
```

And second, you have to add two resolvers for Apollo Server to match the two new top level queries:

```
let messages = {
  1: {
    id: '1',
    text: 'Hello World',
  },
  2: {
    id: '2',
    text: 'By World',
  },
};

const resolvers = {
  Query: {
    users: () => {
      return Object.values(users);
    },
    user: (parent, { id }) => {
      return users[id];
    },
    me: (parent, args, { me }) => {
      return me;
    },
    messages: () => {
      return Object.values(messages);
    },
    message: (parent, { id }) => {
      return messages[id];
    },
  },
};
```

 (<https://www.facebook.com/sharer>

 (<https://twitter.com/sharer>

[/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f) [/share?text=GraphQL%20Server%20Tutorial](https://twitter.com/sharer/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

[%2fwww.robinwieruch.de%2fgraphql-](https://www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f) [by %40rwieruch %23ReactJs'](https://www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

Once you run your application again, your new GraphQL queries should work in GraphQL playground. They are pretty similar to your previous user queries and thus it doesn't add any excitement to your

code. But what about adding relationships to both GraphQL types now? Historically, coming from the REST world, it was common to add an identifier to each entity in order to resolve its relationship later on.

```
const schema = gql`
  type Query {
    users: [User!]
    user(id: ID!): User
    me: User

    messages: [Message!]!
    message(id: ID!): Message!
  }

  type User {
    id: ID!
    username: String!
  }

  type Message {
    id: ID!
    text: String!
    userId: ID!
  }
`;
```

However, now you are using GraphQL and you can make use of it. Instead of using an identifier and resolving the entities later with multiple waterfall requests, what about using the user entity within the message entity directly? Let's try it:

```
const schema = gql`
  ...

  type Message {
    id: ID!
    text: String!
    user: User!
  }
`;
```

Since a message doesn't have a user entity in your model (here sample data), the default resolver doesn't work. You need to set up an explicit resolver for it.

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/rwieruch>)

/share?text=GraphQL%20Server%20Tutorial by %40rwieruch %23ReactJs%2fwww.robinwieruch.de%2f


```
const resolvers = {
  Query: {
    users: () => {
      return Object.values(users);
    },
    user: (parent, { id }) => {
      return users[id];
    },
    me: (parent, args, { me }) => {
      return me;
    },
    messages: () => {
      return Object.values(messages);
    },
    message: (parent, { id }) => {
      return messages[id];
    },
  },
  Message: {
    user: () => {
      return me;
    },
  },
};
```

In this case, every message is simply written by the authenticated user (here the `me` user). If you query the following about messages, you will get this result:

```
// query
{
  message(id: "1") {
    id
    text
    user {
      id
      username
    }
  }
}

// query result
{
  "data": {
    "message": {
      "id": "1",
      "text": "Hello World",
      "user": {
        "id": "1",
        "username": "Robin Wieruch"
      }
    }
  }
}
```

 ([https://www.facebook.com/sharer](https://www.facebook.com/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

 ([https://twitter.com/share](https://twitter.com/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

[/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-](https://www.facebook.com/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f) [/share?text=GraphQL%20Server%20Tutorial](https://twitter.com/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

[%2fwww.robinwieruch.de%2fgraphql-](https://www.facebook.com/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f) [by %40rwieruch %23ReactJs'](https://twitter.com/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

Let's mimic the behavior more like in a real world application. Your sample data has to have some kind of [%2fwww.robinwieruch.de%2f](https://www.robinwieruch.de/graphql-apollo-server-tutorial/)

of keys to reference entities to each other. Thus a message, in your database for instance, could have a `userId` property.

```
let messages = {
  1: {
    id: '1',
    text: 'Hello World',
    userId: '1',
  },
  2: {
    id: '2',
    text: 'By World',
    userId: '2',
  },
};
```

Now, the parent argument in your resolver function can be used to get a message's `userId` which then can be used to retrieve the appropriate user.

```
const resolvers = {
  ...
  Message: {
    user: message => {
      return users[message.userId];
    },
  },
};
```

Now every message has its own dedicated user. The last steps were crucial for understanding GraphQL. Even though you have default resolver functions or this fine-grained control over the fields by defining your own resolver functions, it is up to you to retrieve the data from a data source (here sample data, but later maybe a database). The developer has to make sure that every field can be resolved. GraphQL enables you in the end to group those fields into one GraphQL query regardless of the data source.

Let's recap this implementation detail again with another relationship: a user has messages. In this case, the relationships goes in the other direction.

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

```
let users = {
  1: {
    id: '1',
    username: 'Robin Wieruch',
    messageIds: [1],
  },
  2: {
    id: '2',
    username: 'Dave Davids',
    messageIds: [2],
  },
};
```

This sample data could come from any of your data sources. The important part is that it has a key(s) which defines a relationship to another entity. All of this is independent from GraphQL, so let's define this relationship from users to their messages in GraphQL.

```
const schema = gql`
  type Query {
    users: [User!]
    user(id: ID!): User
    me: User

    messages: [Message!]!
    message(id: ID!): Message!
  }

  type User {
    id: ID!
    username: String!
    messages: [Message!]
  }

  type Message {
    id: ID!
    text: String!
    user: User!
  }
`;
```

Since a user entity hasn't messages but message identifiers instead, you can write again a custom resolver for it. In this case, the resolver retrieves all messages from the user from the list of sample messages.

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/rwieruch/status/1041111111111111111>)

```
const resolvers = {
  ...

  User: {
    messages: user => {
      return Object.values(messages).filter(
        message => message.userId === user.id,
      );
    },
  },

  Message: {
    user: message => {
      return users[message.userId];
    },
  },
};
```

This section has shown you how to expose relationships in your GraphQL schema. But GraphQL doesn't apply any magic for you. If the default resolvers don't work for you, you have to define your own custom resolvers on a per field level for resolving the data from different data sources.


Exercises:

- query a list of users with their messages
- query a list of messages their user
- read more about the GraphQL schema (<https://graphql.github.io/learn/schema/>)

Apollo Server: Queries and Mutations

So far, you have only defined queries in your GraphQL schema by using two related GraphQL types for reading data. These should work for you in GraphQL Playground, because you have given them equivalent resolvers. So what about GraphQL mutations for writing data? In the following, you will create two mutations: create and delete a message. Let's start with creating a message as the currently signed in user (the `me` user).

 (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

 (<https://twitter.com/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%23ReactJs%2fwww.robinwieruch.de%2f>)

```
const schema = gql`
  type Query {
    users: [User!]
    user(id: ID!): User
    me: User

    messages: [Message!]!
    message(id: ID!): Message!
  }

  type Mutation {
    createMessage(text: String!): Message!
  }

  ...
`;
```

As you can see, apart from the Query type there exists also a Mutation (and Subscription) type. There you can group all your GraphQL operations for writing data instead of reading data. In this case, the `createMessage` mutation accepts a non-nullable `text` input as argument and returns the created message. Again, you have to implement the resolver as counterpart for the mutation the same as you have done it with the queries before. It happens in the mutation part of the resolver map:


```
const resolvers = {
  Query: {
    ...
  },


  Mutation: {
    createMessage: (parent, { text }, { me }) => {
      const message = {
        text,
        userId: me.id,
      };

      return message;
    },
  },

  ...
};
```

The mutation's resolver has access to the text in its arguments (second argument). Furthermore, it has access to the currently signed in user in the context argument (third argument) which can be used to associate the created message with the user. The parent argument isn't used. The one thing missing for making the message complete is an identifier. For making sure that a unique identifier is used, you can install this neat library on the command line:

 (<https://www.facebook.com/sharer>

 (<https://twitter.com/sharer>

[/sharer.php?u=https%3a%2f](https://www.facebook.com/sharer)

[/share?text=GraphQL%20Server%20Tutorial](https://twitter.com/sharer)

[%2fwww.robinwieruch.de%2fgraphql-](https://www.robinwieruch.de/graphql-apollo-server-tutorial)

[by %40rwieruch %23ReactJs'](https://www.robinwieruch.de/graphql-apollo-server-tutorial)

[apollo-server-tutorial%2f\)](https://www.robinwieruch.de/graphql-apollo-server-tutorial)

[%2fwww.robinwieruch.de%2f](https://www.robinwieruch.de/graphql-apollo-server-tutorial)

And import it to your file:

```
import uuidv4 from 'uuid/v4';
```

Now you can give your message a unique identifier:


```
const resolvers = {
  Query: {
    ...
  },

  Mutation: {
    createMessage: (parent, { text }, { me }) => {
      const id = uuidv4();
      const message = {
        id,
        text,
        userId: me.id,
      };

      return message;
    },
  },
  ...
};
```

So far, the mutation is only creating a message object and returns it to the API. However, most mutations have side-effects, because they are writing data to your data source or they do something else. Most often it will be a write operation to your database, but in this case, you only need to update your `users` and `messages` variables: The list of available messages needs to be updated and the user's reference list of `messageIds` needs to have the new message `id` as well.

 (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

 (<https://twitter.com/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

```

const resolvers = {
  Query: {
    ...
  },

  Mutation: {
    createMessage: (parent, { text }, { me }) => {
      const id = uuidv4();
      const message = {
        id,
        text,
        userId: me.id,
      };

      messages[id] = message;
      users[me.id].messageIds.push(id);

      return message;
    },
  },

  ...
};

```

The last part you have added would be essentially your writing operation to a data source (e.g. database). In this case, you have only updated the sample data.

That's it for the first mutation. You can try it right now in GraphQL Playground. Next, you are going to implement the mutation for deleting a message.

```

const schema = gql`
  type Query {
    users: [User!]
    user(id: ID!): User
    me: User

    messages: [Message!]!
    message(id: ID!): Message!
  }

  type Mutation {
    createMessage(text: String!): Message!
    deleteMessage(id: ID!): Boolean!
  }

  ...
`;

```

The `Boolean!` denotes whether the deletion was successful or not. (https://www.facebook.com/sharer

Apart from this, the mutation takes an identifier as input for identifying the message. The counterpart

of the GraphQL schema implementation is a resolver:

by %40rwieruch %23ReactJs'

%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f) %2fwww.robinwieruch.de%2f

```

const resolvers = {
  Query: {
    ...
  },

  Mutation: {
    ...

    deleteMessage: (parent, { id }) => {
      const { [id]: message, ...otherMessages } = messages;

      if (!message) {
        return false;
      }

      messages = otherMessages;

      return true;
    },
  },
  ...
};



```

The resolver finds the message by id from the object of messages by using a destructuring. If there is no message, the resolver returns false. If there is a message, the remaining messages without the deleted message are the updated version of the messages object. Then the resolver returns true. Otherwise, if no message is found, the resolver returns false. Basically that's it for implementing mutations in GraphQL and Apollo Server. It isn't much different from GraphQL queries except for the side-effect of writing data.

There is only one GraphQL operation missing for making the messages features complete. Whereas it is possible to read message(s), create a message and delete a message, the only piece missing is updating a message. This GraphQL mutation isn't important for the next sections, but you can try to implement it yourself as exercise.

Exercises:

- create a message in GraphQL Playground with a mutation
 - afterward, query all messages
 - afterward, query the me user with messages
- delete a message in GraphQL Playground with a mutation
 - afterward, query all messages
 - afterward, query the me user with messages

 (https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)
  (https://twitter.com/rwieruch/status/1040714071407140714)

• implement a updateMessage mutation for completing all CRUD operations for a message in GraphQL

• read more about GraphQL queries and mutations (https://graphql.github.io/learn/queries/) by @rwieruch

• ReactJS tutorial (https://www.robinwieruch.de/reactjs-tutorial)

GraphQL Schema Stitching with Apollo Server

Schema stitching is a powerful feature in GraphQL. Basically it's all about merging multiple GraphQL schemas into one schema which may be consumed in your GraphQL client application. As for now, you only have one schema in your application. So what's the motivation behind multiple schemas and schema stitching? For instance, take a GraphQL schema in one project which you may want to modularize based on domains (e.g. user, message). There you may end up with two schemas whereas each schema matches one type (e.g. User type, Message type). Now you would want to merge both GraphQL schemas for making the entire GraphQL schema accessible with your GraphQL server's API. That's one of the basic motivations behind schema stitching.


But you can take this one step further: You may end up with microservices or third-party platforms which expose their dedicated GraphQL APIs which then can be used for merging them into one GraphQL schema with schema stitching as single source of truth. Then again a client can consume the entire schema which is composed out of multiple domain-driven microservices.

In our case, let's start with a separation by technical concerns for the GraphQL schema and resolvers. Afterward, you will apply the separation by domains which are in this case users and messages.

Technical Separation

Let's take the GraphQL schema from the application where you have a User type and Message type. In the same step, you want to split out the resolvers to their dedicated place as well. In the end, the `src/index.js` file, where the schema and resolvers are needed for the Apollo Server instantiation, should only import both things. It becomes three things when outsourcing the data, in this case the sample data (now called models), too.

 (https://www.facebook.com/sharer
/sharer.php?u=https%3a%2f
%2fwww.robinwieruch.de%2fgraphql-
apollo-server-tutorial%2f)

 (http
/share?text=GraphQL%20Server%20Tutorial
by %40rwieruch %23ReactJs%
%2fwww.robinwieruch.de%2f)

```

import cors from 'cors';
import express from 'express';
import { ApolloServer } from 'apollo-server-express';

import schema from './schema';
import resolvers from './resolvers';
import models from './models';

const app = express();

app.use(cors());

const server = new ApolloServer({
  typeDefs: schema,
  resolvers,
  context: {
    models,
    me: models.users[1],
  },
});

server.applyMiddleware({ app, path: '/graphql' });

app.listen({ port: 8000 }, () => {
  console.log('Apollo Server on http://localhost:8000/graphql (http://localhost:8000/graphql)');
});

```

As improvement, the models are passed to the resolver function's as context. The models are your data access layer. It can be sample data (which is the case), a database, or a third-party API. It's always good to pass those things in from the outside for keeping the resolver functions pure. Then you don't need to import the models in each resolver file. So now in this case, the models are the sample data and moved to the *src/models/index.js* file:

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/rwieruch/status/1022222222222222222>)

```

let users = {
  1: {
    id: '1',
    username: 'Robin Wieruch',
    messageIds: [1],
  },
  2: {
    id: '2',
    username: 'Dave Davids',
    messageIds: [2],
  },
};

let messages = {
  1: {
    id: '1',
    text: 'Hello World',
    userId: '1',
  },
  2: {
    id: '2',
    text: 'By World',
    userId: '2',
  },
};

export default {
  users,
  messages,
};

```

Since you have passed the models to your Apollo Server context, they are accessible in each resolver. In the next step, let's move the resolvers to the *src/resolvers/index.js* file. In the same step, adjust the resolver's function signature by adding the models when they are needed to read/write users or messages.

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

```

import uuidv4 from 'uuid/v4';

export default {
  Query: {
    users: (parent, args, { models }) => {
      return Object.values(models.users);
    },
    user: (parent, { id }, { models }) => {
      return models.users[id];
    },
    me: (parent, args, { me }) => {
      return me;
    },
    messages: (parent, args, { models }) => {
      return Object.values(models.messages);
    },
    message: (parent, { id }, { models }) => {
      return models.messages[id];
    },
  },

  Mutation: {
    createMessage: (parent, { text }, { me, models }) => {
      const id = uuidv4();
      const message = {
        id,
        text,
        userId: me.id,
      };

      models.messages[id] = message;
      models.users[me.id].messageIds.push(id);

      return message;
    },

    deleteMessage: (parent, { id }, { models }) => {
      const { [id]: message, ...otherMessages } = models.messages;

      if (!message) {
        return false;
      }

      models.messages = otherMessages;


      return true;
    },
  },

  User: {
    messages: (user, args, { models }) => {
      return Object.values(models.messages).filter(
        message => message.userId === user.id,
      );
    },
  },
};

Message: {
  user: (parent, args, { models }) => {
    return models.users[message.userId];
  },
};

```

 (<https://www.facebook.com/sharer>

 (<https://twitter.com/sharer>

[/sharer.php?u=https%3a%2f](#)

[/share?text=GraphQL%20Server%20Tutorial](#)

<https://www.robinwieruch.de/graphql-apollo-server-tutorial/>

by [%40rwieruch %23ReactJs](#)

[%2fwww.robinwieruch.de%](https://www.robinwieruch.de/)

```
  },
};
```

The resolvers receive all sample data as models in the context argument rather than operating directly on the sample data as before. As mentioned, it keeps the resolver functions pure. Later on, you will have an easier time to test your resolver functions in isolation too. Last but not least, move your schema's type definitions in the `src/schema/index.js` file:

```
import { gql } from 'apollo-server-express';

export default gql`
  type Query {
    users: [User!]
    user(id: ID!): User
    me: User

    messages: [Message!]!
    message(id: ID!): Message!
  }

  type Mutation {
    createMessage(text: String!): Message!
    deleteMessage(id: ID!): Boolean!
  }

  type User {
    id: ID!
    username: String!
    messages: [Message!]
  }

  type Message {
    id: ID!
    text: String!
    user: User!
  }
`;
```

Even though your application should work again and the technical separation is complete, the separation by domains, where schema stitching is needed, isn't done yet. So far, you have only outsourced the schema, resolvers and data (models) from your Apollo Server instantiation file. Everything is separated by technical concerns now. Furthermore, you made the small improvement for passing the models through the context rather than importing them in resolver file(s).

Domain Separation

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/rwieruch/status/1041111111111111111>)

In the next step, you are going to modularize the GraphQL schema by domains (user and message). First, split out the user related entity in its own schema definition file called `src/schema/user.js`:
by [www.robinwieruch.de](https://www.robinwieruch.de/graphql-apollo-server-tutorial/) %2fgraphql-apollo-server-tutorial%2f
%2fwww.robinwieruch.de%2f


```
import { gql } from 'apollo-server-express';

export default gql`
  extend type Query {
    users: [User!]
    user(id: ID!): User
    me: User
  }

  type User {
    id: ID!
    username: String!
    messages: [Message!]
  }
`;
```

The same applies for the message schema definition in *src/schema/message.js*:

```
import { gql } from 'apollo-server-express';

export default gql`
  extend type Query {
    messages: [Message!]!
    message(id: ID!): Message!
  }

  extend type Mutation {
    createMessage(text: String!): Message!
    deleteMessage(id: ID!): Boolean!
  }

  type Message {
    id: ID!
    text: String!
    user: User!
  }
`;
```

Notice how each file only describes its own entity with a type and its relations. A relation can be a type from a different file (e.g. a Message type still has the relation to a User type even though the User type is defined somewhere else). Furthermore, note the `extend` statement on the Query and Mutation types. Since you have more than one of those types now, you need to extend the types. Finally you have to define shared base types for them in the *src/schema/index.js*:

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/rwieruch>)

/share?text=GraphQL%20Server%20Tutorial
by %40rwieruch %23ReactJs'
%2fwww.robinwieruch.de%2f

```
import { gql } from 'apollo-server-express';

import userSchema from './user';
import messageSchema from './message';

const linkSchema = gql`
  type Query {
    _: Boolean
  }

  type Mutation {
    _: Boolean
  }


  type Subscription {
    _: Boolean
  }
`;


export default [linkSchema, userSchema, messageSchema];
```

In this file, both schemas are merged together with the help of a utility `linkSchema`. The `linkSchema` defines all types which are shared within the schemas. It already defines a `Subscription` type for GraphQL subscriptions which may be implemented later on. As a workaround, there is an empty `underscore` field with a `Boolean` type in the merging utility schema, because there is no official way of doing this merging of schemas yet. Basically the utility schema only defines the shared base types which are extended with the `extend` statement in the other domain specific schemas.

Once you run the application again, it should work again, but this time with a stitched schema instead of one global schema. What's missing are the domain separated resolver maps. Let's start with the user domain again in the `src/resolvers/user.js`:

```
export default {
  Query: {
    users: (parent, args, { models }) => {
      ...
    },
    user: (parent, { id }, { models }) => {
      ...
    },
    me: (parent, args, { me }) => {
      ...
    },
  },
  User: {
    messages: (user, args, { models }) => {
```

 (<https://www.facebook.com/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

 (<https://twitter.com/sharer?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

[/share?text=GraphQL%20Server%20Tutorial%20by%40rwieruch%23ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f](https://www.robinwieruch.de/graphql-apollo-server-tutorial/)

Followed by the message resolvers in the *src/resolvers/message.js* file:

```
import uuidv4 from 'uuid/v4';

export default {
  Query: {
    messages: (parent, args, { models }) => {
      ...
    },
    message: (parent, { id }, { models }) => {
      ...
    },
  },
  Mutation: {
    createMessage: (parent, { text }, { me, models }) => {
      ...
    },
    deleteMessage: (parent, { id }, { models }) => {
      ...
    },
  },
  Message: {
    user: (message, args, { models }) => {
      ...
    },
  },
};
```

Since the Apollo Server accepts a list of resolver maps too, you can import all of your resolver maps in your *src/resolver/index.js* file and export them as a list of resolver maps again:

```
import userResolvers from '../resolvers/user';
import messageResolvers from '../resolvers/message';

export default [userResolvers, messageResolvers];
```

Then the Apollo Server can take the resolver list to be instantiated. Start your application again and verify that everything is working for you.

In the last section, you have extracted schema and resolvers from your main file and separated both by domains. Moreover the sample data is placed in a *src/models* folder where it can be easily migrated to a database driven approach in the future. Your folder structure should look similar to this one now.

[f \(https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f\)](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)
[t \(https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%40rwieruch%23ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f\)](https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%20robinwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

```
* src/
  * models/
    * index.js
  * resolvers/
    * index.js
    * user.js
    * message.js
  * schema/
    * index.js
    * user.js
    * message.js
  * index.js
```

It's a great starting point for a GraphQL server application with Node.js. Basically from here on you could develop your own application on top of it, because the last implementations gave you a universally usable GraphQL boilerplate project. However, if you are curious connecting your GraphQL server to a database, doing authentication and authorization, and implementing powerful features such as pagination in GraphQL, stay with me on this journey.

Exercises:

- read more about schema stitching with Apollo Server (<https://www.apollographql.com/docs/graphql-tools/schema-stitching.html>)
- schema stitching is only a part of **schema delegation**
 - read more about schema delegation (<https://www.apollographql.com/docs/graphql-tools/schema-delegation.html>)
 - get familiar with the motivation behind **remote schemas** and **schema transforms**

PostgreSQL with Sequelize for a GraphQL Server

In order to end up with a full-stack GraphQL application eventually, you need to introduce a sophisticated data source at some point. Whereas the sample data from before is only fluctuant data, a database can give you persistent data. In this section, you are going to set up PostgreSQL with Sequelize (ORM (https://en.wikipedia.org/wiki/Object-relational_mapping)) for Apollo Server. PostgreSQL (<https://www.postgresql.org/>) is a SQL database whereas an alternative would be the popular NoSQL database called MongoDB (<https://www.mongodb.com/>) (with Mongoose as ORM). The choice of tech is always opinionated. You could choose MongoDB or any other SQL/NoSQL solution over PostgreSQL, but for the sake of this application, let's stick to PostgreSQL.

f (<https://www.facebook.com/sharer>

t (<https://www.facebook.com/sharer>

Before you can use PostgreSQL in your application, you need to install it for your machine first. If you haven't installed it, head over to this setup guide (<https://www.robinwieruch.de/postgres-express-2fwww-robinwieruch-de-2fgraphql-by-40wieruch-23ReactJs-apollo-server-tutorial-2f>) for installing it, setting up your first database, creating an administrative database user, and learning the essentials of GraphQL. These are the things you should know before you start the tutorial.

going through the instructions:

- having running installation of PostgreSQL
- having a database super user with username and password
- having a database for this application which you have created with `createdb` or `CREATE DATABASE`

Furthermore, you should be able to run and stop your database with the following commands:

- `pg_ctl -D /usr/local/var/postgres start`
- `pg_ctl -D /usr/local/var/postgres stop`

Last but not least, when using the `psql` command on the command line, you should be able to connect to your database on a command line level. There you can list all your databases or execute SQL statements against your database. You should find a couple of these operations in the PostgreSQL setup guide, but also this section will show some of them. However, in the following sections, you should perform these on your own the same way as you are doing GraphQL operations with GraphQL Playground. The `psql` command line interface and GraphQL Playground are your perfect tools for trying your application manually.

What's needed for your GraphQL server once you have installed PostgreSQL on your local machine? First, you need to install PostgreSQL for Node.js (<https://github.com/brianc/node-postgres>) and Sequelize (ORM) (<https://github.com/sequelize/sequelize>) for your project. Whereas you don't necessarily need to look into the PostgreSQL for Node.js documentation, I highly recommend to have the Sequelize documentation open on the side for your project. You might find yourself always looking up certain things when connecting your GraphQL layer (resolvers) with your data access layer (Sequelize) in the following implementations for this applications.

```
npm install pg sequelize --save
```

Now you can create your models for your the user and message domains. Models are usually your data access layer in applications. In this case, you are going to setup your models with Sequelize in order to make read and write operations to your PostgreSQL database. Afterward, the models can be used in your GraphQL resolvers by passing them through the context object to each resolver. Now, the essential steps we are doing together are:

f (<https://www.facebook.com/sharer>)

t (<https://twitter.com/sharer>)

- creating a model for the user domain
- creating a model for the message domain
- connecting the application to a database

/share?text=GraphQL%20Server%20Tutorial

%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)
 • providing super user's username and password
 • connecting models for database usage

by %40rwieruch %23ReactJs'
 %2fwww.robinwieruch.de%2f

- synchronizing the database once application starts

First, implement the *src/models/user.js* model:

```
const user = (sequelize, DataTypes) => {
  const User = sequelize.define('user', {
    username: {
      type: DataTypes.STRING,
    },
  });

  User.associate = models => {
    User.hasMany(models.Message, { onDelete: 'CASCADE' });
  };

  return User;
};

export default user;
```

Second, implement the *src/models/message.js* model:

```
const message = (sequelize, DataTypes) => {
  const Message = sequelize.define('message', {
    text: {
      type: DataTypes.STRING,
    },
  });

  Message.associate = models => {
    Message.belongsTo(models.User);
  };

  return Message;
};

export default message;
```

Basically both models define the shapes of their entities. For instance, the message model has a database column with the name text of type string. You can add multiple database columns horizontally to your model. In the end, all columns of a model make up a table row in the database and each row reflect a database entry (e.g. message, user). The database table name is defined a first argument in the Sequelize model definition. The message domain has the table “message”.

Furthermore, you can define relationships between entities with Sequelize by using associations. In

this case, a message entity belongs to one user and a user has many messages. That's everything you need for setting up a minimal database with two domains. However, since this is mostly about GraphQL on the server-side and not too much about databases, you should read up more database subjects on your own to fully grasp the topic.

apollo-server-tutorial%2f)

%2fwww.robinwieruch.de%2f

Next, you need to connect to your database from within your application in the `src/models/index.js` file. Essentially everything that is needed are the database name, a database super user and the user's password. You may also want to define a database dialect, because Sequelize supports other databases too.

```
import Sequelize from 'sequelize';

const sequelize = new Sequelize(
  process.env.DATABASE,
  process.env.DATABASE_USER,
  process.env.DATABASE_PASSWORD,
  {
    dialect: 'postgres',
  },
);

export { sequelize };
```

In the same file, you can physically associate all your models with each other for exposing them afterward to your application as data access layer (models) to your database.

```
import Sequelize from 'sequelize';

const sequelize = new Sequelize(
  process.env.DATABASE,
  process.env.DATABASE_USER,
  process.env.DATABASE_PASSWORD,
  {
    dialect: 'postgres',
  },
);


const models = {
  User: sequelize.import('./user'),
  Message: sequelize.import('./message'),
};

Object.keys(models).forEach(key => {
  if ('associate' in models[key]) {
    models[key].associate(models);
  }
});

export { sequelize };

export default models;
```

 ([https://www.facebook.com/sharer](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

 ([https://twitter.com/share](https://twitter.com/share?text=GraphQL%20Server%20Tutorial%20for%20newbies%20on%20Node.js%20for%20GraphQL%20by%40rwsimpson%20ReactJs%20www.robinwieruch.de%2f)

As you can see, the database credentials (database name, database super-user name, database user name and password) are defined as environment variables. In your `.env` file, you can simply add the credentials as key value pairs. For instance, my defaults for local development are the following:

```

DATABASE=postgres
DATABASE_USER=postgres
DATABASE_PASSWORD=postgres

```

You should have set up everything for the environment variables in the very beginning of implementing this application. Otherwise, you can also leave the credentials in the source code for now.

Last but not least, the database needs to be migrated/synchronized once your Node.js application starts. You can do it the following way in your `src/index.js` file:

```

import express from 'express';
import { ApolloServer } from 'apollo-server-express';

import schema from './schema';
import resolvers from './resolvers';
import models, { sequelize } from './models';

...

sequelize.sync().then(async () => {
  app.listen({ port: 8000 }, () => {
    console.log('Apollo Server on http://localhost:8000/graphql (http://localhost:8000/graphql)');
  });
});

```

That's it for the database setup for your GraphQL server. In the next steps, you are going to replace the business logic in your resolvers, because there Sequelize is used to access the database instead of using sample data from before. As for now, the application should not work, because the resolvers don't use the new data access layer.

Exercises:

- get comfortable with databases
 - try the `psql` command line interface for accessing your database
 - check the Sequelize API by reading through their documentation
 - read up database jargon which is unknown to you but mentioned here

Connecting Resolvers and Database

f (<https://www.facebook.com/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>) **t** (<https://twitter.com/rwieruch>)

<https://www.robinwieruch.de/graphql-apollo-server-tutorial/> /share?text=GraphQL%20Server%20Tutorial by %40rwieruch %23ReactJs%23ApolloServer

In the previous section, you have done everything getting your PostgreSQL database up and running and connecting it to your GraphQL server on startup. Now, instead of using the old sample data, you are going to use your new database layer (models) in your GraphQL resolvers for reading and

writing data from/to your database. You are going to implement the following things in this section:

- use the new models in your GraphQL resolvers
- seed your database with data when your application starts
- add a user model method for retrieving a user by username
- learn the essentials about `psql` for the command line

Let's start by refactoring your GraphQL resolvers. You already have passed the models via Apollo Server's context object to each GraphQL resolver. Whereas you have used sample data before, you have to use the Sequelize API now. In the `src/resolvers/user.js` file, change the following lines of code for using the Sequelize API:

```
export default {
  Query: {
    users: async (parent, args, { models }) => {
      return await models.User.findAll();
    },
    user: async (parent, { id }, { models }) => {
      return await models.User.findById(id);
    },
    me: async (parent, args, { models, me }) => {
      return await models.User.findById(me.id);
    },
  },
  User: {
    messages: async (user, args, { models }) => {
      return await models.Message.findAll({
        where: {
          userId: user.id,
        },
      });
    },
  },
};
```

As you can see, the `findAll()` and `findById()` are commonly used Sequelize methods. They are a common case for database operations after all. However, finding all messages for a specific user is a more specific use case. Here you have used the `where` clause for narrowing down messages by the `userId` entry in the database. At this point, it makes totally sense reading up the Sequelize API documentation if anything is unclear, because you are accessing a database instead of using sample data which adds another layer of complexity to your application's architecture.

Finally, open the `src/resolvers/message.js` file and perform adjustments for using the Sequelize API as well:

<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f> <https://twitter.com/sharer?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2f>

```

export default {
  Query: {
    messages: async (parent, args, { models }) => {
      return await models.Message.findAll();
    },
    message: async (parent, { id }, { models }) => {
      return await models.Message.findById(id);
    },
  },
  Mutation: {
    createMessage: async (parent, { text }, { me, models }) => {
      return await models.Message.create({
        text,
        userId: me.id,
      });
    },
    deleteMessage: async (parent, { id }, { models }) => {
      return await models.Message.destroy({ where: { id } });
    },
  },
  Message: {
    user: async (message, args, { models }) => {
      return await models.User.findById(message.userId);
    },
  },
};

```

Apart from the previously used `findById()` and `findAll()` methods, you are creating and destroying (deleting) a message in the mutations too. Before you had to generate your own identifier for the message, but now Sequelize takes care of adding a unique identifier to your message once it got created in the database.

There was one crucial change in the two files: `async/await` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function). Sequelize is a JavaScript promise based ORM and thus when operating on a database it always returns a JavaScript promise which gets resolved eventually. That's where `async/await` can be used as much more readable version for asynchronous requests in JavaScript. Furthermore, you have learned about the returned results of GraphQL resolvers in Apollo Server in a previous section. A result can be a JavaScript promise as well, because the resolvers would be waiting for its actual result. So in this case, you could also get rid of the `async/await` statements and your resolvers would still work. However, sometimes it is better to be more explicit, especially when adding more business logic within the resolver's function body later on, so we will keep the statements for now.

f (<https://www.facebook.com/sharer>

t (<https://twitter.com/sharer>

So what about seeding the database with sample data when your application starts with `npm start`? Once your database synchronizes before your server listens, you can create manually two user records with messages in your database. The following code for the `src/index.js` file shows you how to perform these operations with `async/await` again. Your users will have a username and an association

of messages .

```
...

const eraseDatabaseOnSync = true;

sequelize.sync({ force: eraseDatabaseOnSync }).then(async () => {
  if (eraseDatabaseOnSync) {
    createUsersWithMessages();
  }

  app.listen({ port: 8000 }, () => {
    console.log('Apollo Server on http://localhost:8000/graphql (http://localhost:8000/graphql)');
  });
});

const createUsersWithMessages = async () => {
  await models.User.create(
    {
      username: 'rwieruch',
      messages: [
        {
          text: 'Published the Road to learn React',
        },
      ],
    },
    {
      include: [models.Message],
    },
  );

  await models.User.create(
    {
      username: 'ddavids',
      messages: [
        {
          text: 'Happy to release ...',
        },
        {
          text: 'Published a complete ...',
        },
      ],
    },
    {
      include: [models.Message],
    },
  );
};
```

Furthermore, the `force` flag in your `sequelize sync()` method can be used to seed the database on every application startup again. So every time you will start with a sample data seeded database. You can either remove the flag or set it to `false` if you want to keep your accumulated database changes over time. After all, the flag should be removed for your production database at some point.

Last but not least, we have to take care about the `me` user. Before you have simply used one of the users from your sample data. Now, you can use one of the users from your database. That's a great opportunity for writing a custom method for your user model in the `src/models/user.js` file:

```
const user = (sequelize, DataTypes) => {
  const User = sequelize.define('user', {
    username: {
      type: DataTypes.STRING,
    },
  });

  User.associate = models => {
    User.hasMany(models.Message, { onDelete: 'CASCADE' });
  };

  User.findByLogin = async login => {
    let user = await User.findOne({
      where: { username: login },
    });

    if (!user) {
      user = await User.findOne({
        where: { email: login },
      });
    }

    return user;
  };

  return User;
};

export default user;
```

The `findByLogin()` method on your user model retrieves a user either by `username` or `email` entry. You don't have a `email` entry on your user yet, but will add it eventually when this application implements an authentication mechanism. However, the `login` argument is used for both `username` and `email` for retrieving the user from the database. Maybe you can already imagine how it can be used to sign in to an application either with username or email.

Now you have introduced your first custom method on a database model. It is always worth considering where to put this business logic. When giving your model these access methods, you may end up with a concept called *fat models* in programming. An alternative would be writing separate services (e.g. functions, classes) for these data access layer functionalities.

f (<https://www.facebook.com/sharer>

t (<https://twitter.com/rwieruch>

The new model method can be used for retrieving the `me` user from the database. Then you can put it into the context object when the Apollo-Server is instantiated in the `src/index.js` file.

[%2fwww.robinwieruch.de%2fgraphql-](https://www.robinwieruch.de/graphql-apollo-server-tutorial/)

[by %40rwieruch %23ReactJs'](https://www.robinwieruch.de/graphql-apollo-server-tutorial/)

[apollo-server-tutorial%2f\)](https://www.robinwieruch.de/graphql-apollo-server-tutorial/)

[%2fwww.robinwieruch.de%2f](https://www.robinwieruch.de/graphql-apollo-server-tutorial/)

```
const server = new ApolloServer({
  typeDefs: schema,
  resolvers,
  context: {
    models,
    me: models.User.findByLogin('rwieruch'),
  },
});
```

However, this cannot work yet, because 1) the user is read asynchronously from the database and thus `me` would be a JavaScript promise rather than the actual user and 2) you may want to retrieve the `me` user on a per request basis from the database. Otherwise, the `me` user would always stay the same after the Apollo Server got created. Therefore, you can use a function which returns the context object rather than an object for the context in Apollo Server. This function can make use of the `async/await` statements then. Furthermore, the function is invoked every time a request is hitting your GraphQL API and thus the `me` user is retrieved from the database with every request.


```
const server = new ApolloServer({
  typeDefs: schema,
  resolvers,
  context: async () => ({
    models,
    me: await models.User.findByLogin('rwieruch'),
  }),
});
```

You should be able to start your application again. Try out your different GraphQL queries and mutations in GraphQL Playground and verify that everything is working for you. If there are any errors regarding the database, make sure that it is properly connected to your application and that the database is running on the command line too.

Since you have introduced a database now, GraphQL Playground is not the only manual testing tool anymore. Whereas GraphQL Playground can be used to test your GraphQL API, you may want to use the `psql` command line interface to query your database manually. For instance, you may want to check how your user or message records look like in the database or whether a message is located in the database after it has been created with a GraphQL mutation. So let's see briefly how you can do it yourself. First, connect to your database on the command line:

```
psql mydatabasename
```

 ([https://www.facebook.com/sharer](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

 ([https://twitter.com/share](https://twitter.com/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%23ReactJs%20%2fwww.robinwieruch.de%2f)

And second, try the following SQL statements. It's the perfect opportunity to learn more about SQL itself: <https://www.robinwieruch.de/graphql-apollo-server-tutorial%2f> by <https://www.robinwieruch.de%2f>

```
SELECT * from users;
SELECT text from messages;
```

Which should lead to:

```
mydatabase=# SELECT * from users;
 id | username |          createdAt          |          updatedAt
-----+-----+-----+-----
  1 | rwieruch | 2018-08-21 21:15:38.758+08 | 2018-08-21 21:15:38.758+08
  2 | ddavids  | 2018-08-21 21:15:38.786+08 | 2018-08-21 21:15:38.786+08
(2 rows)

mydatabase=# SELECT text from messages;
      text
-----
Published the Road to learn React
Happy to release ...
Published a complete ...
(3 rows)
```

So every time you are doing GraphQL mutations, it is valuable to check your database records with the `psql` command line interface too. As mentioned, it is a great way to learn about SQL (<https://en.wikipedia.org/wiki/SQL>) itself which is normally abstracted away by using a ORM such as Sequelize.

In this section, you have used a PostgreSQL database as data source for your GraphQL server whereas Sequelize is the glue between your database and your GraphQL resolvers. However, this was only one possible solution here. Since GraphQL is data source agnostic, it is up to you to opt-in any data source to your resolvers. It can be another database (e.g. MongoDB, Neo4j, Redis), multiple databases or a (third-party) REST/GraphQL API endpoint. GraphQL only makes sure that all fields are validated, executed and resolved when there is an incoming query or mutation regardless of the data source.

Exercises:

- play around with `psql` and the seeding of your database
- play around with GraphQL playground and query data which comes from a database now
- remove (and add) the `async/await` statements in your resolvers and see how they still work
 - read more about GraphQL execution (<https://graphql.github.io/learn/execution/>)

f (<https://www.facebook.com/sharer>

t (<https://twitter.com/sharer>

[/sharer.php?u=https%3A%2F%2Fwww.robinwieruch.de%2Fgraphql-apollo-server-tutorial%2F](https://www.facebook.com/sharer/sharer.php?u=https%3A%2F%2Fwww.robinwieruch.de%2Fgraphql-apollo-server-tutorial%2F) [/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20Validation%20error%20and%20edge%20case%20handling%20are%20not%20often%20verbalized%20in%20programming%20Often%20only%20the%20happy%20path%20is%20showcased%20to%20everyone%20This%20section%20should%20give%20you%20some%20insights%20into%20these%20topics](https://twitter.com/sharer/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20Validation%20error%20and%20edge%20case%20handling%20are%20not%20often%20verbalized%20in%20programming%20Often%20only%20the%20happy%20path%20is%20showcased%20to%20everyone%20This%20section%20should%20give%20you%20some%20insights%20into%20these%20topics)

[%2Fwww.robinwieruch.de%2Fgraphql-](https://www.robinwieruch.de/graphql-apollo-server-tutorial/)

[by %40rwieruch %23ReactJs'](https://www.robinwieruch.de/graphql-apollo-server-tutorial/)

Validation, error and edge case handling are not often verbalized in programming. Often only the

happy path is showcased to everyone. This section should give you some insights into these topics

when using Apollo Server and GraphQL. The great thing about GraphQL: You are in charge what you want to return from your GraphQL resolvers. Therefore, it isn't too difficult inserting business logic into your resolvers, for instance, before they are reading from your database.

```
export default {
  Query: {
    users: async (parent, args, { models }) => {
      return await models.User.findAll();
    },
    user: async (parent, { id }, { models }) => {
      return await models.User.findById(id);
    },
    me: async (parent, args, { models, me }) => {
      if (!me) {
        return null;
      }

      return await models.User.findById(me.id);
    },
  },
  ...
};
```

In general, it may be a good idea keeping the resolvers surface slim but adding business logic services on the side. Then it is always simple to reason about the resolvers. However, in this application we will keep the business logic in the resolvers for having everything at one place without scattering all the logic across the entire application.

Let's start with the validation which will lead to error handling eventually. GraphQL itself isn't directly concerned about validation. But it acts in between your tech stacks which are concerned about validation: your client application (e.g. showing some validation messages) and your database (e.g. validation of entities before writing to the database). Let's add some basic validation rules to your database models. It is up to you to extend these rules in the future. This section should only give you an introduction to the topic, because otherwise it becomes too verbose to cover all the edge cases in this application. First, add validation to your user model in the *src/models/user.js* file:

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

```
const user = (sequelize, DataTypes) => {
  const User = sequelize.define('user', {
    username: {
      type: DataTypes.STRING,
      unique: true,
      allowNull: false,
      validate: {
        notEmpty: true,
      },
    },
  });

  ...

  return User;
};

export default user;
```

And second, add the following validation rules to your message model in the *src/models/message.js* file:

```
const message = (sequelize, DataTypes) => {
  const Message = sequelize.define('message', {
    text: {
      type: DataTypes.STRING,
      validate: { notEmpty: true },
    },
  });

  Message.associate = models => {
    Message.belongsTo(models.User);
  };

  return Message;
};

export default message;
```

Now, try to create a message with an empty text in GraphQL Playground. It shouldn't work because you require a non-empty text for your message in the database. The same applies to your user entities which need a unique username now. But can GraphQL and Apollo Server handle these cases already for you? Let's try to create a message with an empty text. You should see a similar input and output:

f ([https://www.facebook.com/sharer](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)
[/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f))

t (<https://twitter.com/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)


```
// mutation
mutation {
  createMessage(text: "") {
    id
  }
}

// mutation error result
{
  "data": null,
  "errors": [
    {
      "message": "Validation error: Validation notEmpty on text failed",
      "locations": [],
      "path": [
        "createMessage"
      ],
      "extensions": { ... }
    }
  ]
}
```


It seems like Apollo Server's resolvers make sure to transform JavaScript errors (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error) to valid GraphQL output. It would be already possible to use this common error format in your client application. You wouldn't have to add any additional error handling to your resolvers.

In case you want to add custom error handling to your resolver, you always can add the commonly try/catch block statements for async/await:

```
export default {
  Query: {
    ...
  },

  Mutation: {
    createMessage: async (parent, { text }, { me, models }) => {
      try {
        return await models.Message.create({
          text,
          userId: me.id,
        });
      } catch (error) {
        throw new Error(error);
      }
    },
    ...
  }
}
```

 (<https://www.facebook.com/sharer>

 (<https://twitter.com/rwieruch>)

[/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f](#) [/share?text=GraphQL%20Server%20Tutorial](#)

[%2fwww.robinwieruch.de%2fgraphql-](https://www.robinwieruch.de/graphql-apollo-server-tutorial/)

[by %40rwieruch %23ReactJs'](#)

[apollo-server-tutorial%2f](#) [%2fwww.robinwieruch.de%2f](#)

The error output for GraphQL should stay the same in the GraphQL Playground, because you have

used the same error object to generate the Error instance. However, you could use your custom message here with `throw new Error('My error message.');`

Another way of adjusting your error message would be in the database model definition. For instance, each validation rule can have a custom validation message. You can define these messages in the Sequelize model:

```
const message = (sequelize, DataTypes) => {
  const Message = sequelize.define('message', {
    text: {
      type: DataTypes.STRING,
      validate: {
        notEmpty: {
          args: true,
          msg: 'A message has to have a text.',
        },
      },
    },
  });

  Message.associate = models => {
    Message.belongsTo(models.User);
  };


  return Message;
};

export default message;
```

This would automatically lead to the following error(s) when attempting to create a message with an empty text. Again, it could be used straight forward in your client application, because the error format itself stays the same:

```
{
  "data": null,
  "errors": [
    {
      "message": "SequelizeValidationError: Validation error: A message has to have a text.",
      "locations": [],
      "path": [
        "createMessage"
      ],
      "extensions": { ... }
    }
  ]
}
```

 (<https://www.facebook.com/sharer>

 (<https://twitter.com/rwieruch>)

[/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial/](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial/)

[/share?text=GraphQL%20Server%20Tutorial](https://twitter.com/rwieruch/status/1038444444444444444)

That's the one great benefit when using Apollo Server for GraphQL: The error handling comes often for free, because an error is either from the database, a custom JavaScript error or another third party gets transformed into a valid GraphQL error result. On the client side, you will need to include the

the error result's shape, because it comes in a common GraphQL error format whereas the data object is null but the errors are captured in an array. If you want to change your custom error, you have seen how you can do it on a resolver per resolver basis. But what about the global error formatting? Apollo Server comes with a solution for it:

```
const server = new ApolloServer({
  typeDefs: schema,
  resolvers,
  formatError: error => {
    // remove the internal sequelize error message
    // leave only the important validation error
    const message = error.message
      .replace('SequelizeValidationError: ', '')
      .replace('Validation error: ', '');

    return {
      ...error,
      message,
    };
  },
  context: async () => ({
    models,
    me: await models.User.findByLogin('rwieruch'),
  }),
});
```

Basically these are the essentials about validation and error handling with GraphQL in Apollo Server. The validation can happen on a database (model) level or on a business logic level (resolvers). It can happen on a directive level too (see exercises). If there is an error, GraphQL and Apollo Server will format it into an appropriate format which is well known for GraphQL clients. If you want to format your errors, you can do it globally in Apollo Server as well. This section has shown you the basics about error handling and validation for GraphQL servers. You can read up the more material from the exercises to dive deeper into the topic.

Exercises:

- add further validation rules to your database models
 - read more about validation in the Sequelize documentation
- read more about Error Handling with Apollo Server (<https://www.apollographql.com/docs/apollo-server/v2/features/errors.html>)
 - get to know the different custom errors in Apollo Server
- read more about GraphQL field level validation with custom directives
 - (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fblog.apollographql.com/graphql-validation-using-directives-4908fd5c1055>)
 - read more about custom schema directives (<https://www.apollographql.com/docs/apollo-server/v2/features/custom-schema-directives>)

Twitter (<https://twitter.com/rwieruch>)

Facebook (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fblog.apollographql.com/graphql-validation-using-directives-4908fd5c1055>)

LinkedIn (<https://www.linkedin.com/sharing/share-offsite/?url=https%3a%2f%2fblog.apollographql.com/graphql-validation-using-directives-4908fd5c1055>)

StumbleUpon (<https://www.stumbleupon.com/sharer.php?u=https%3a%2f%2fblog.apollographql.com/graphql-validation-using-directives-4908fd5c1055>)

Reddit (<https://www.reddit.com/sharer.php?u=https%3a%2f%2fblog.apollographql.com/graphql-validation-using-directives-4908fd5c1055>)


by %40rwieruch %23ReactJs%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

Apollo Server: Authentication

Authentication in GraphQL is a popular topic, because there is no opinionated way of doing it, but many people need it for their applications. GraphQL itself isn't opinionated about authentication since it is only a query language. If you want to have authentication in GraphQL, the implementation with GraphQL mutations is up to you. In this section, we will go through a minimalistic approach on how to add authentication to your GraphQL server. Afterward, it should be possible to register (sign up) and login (sign in) a user to your application. The previously used `me` user will be the authenticated user then.

In preparation for the authentication mechanism with GraphQL, you have to extend the user model in the `src/models/user.js` file. The user has to have a email address (as unique identifier) and a password. Both, email address and username (another unique identifier) can be used to sign in a user to the application. That's why both properties were also used for the user's `findByLogin()` method.

 (https://www.facebook.com/sharer
/sharer.php?u=https%3a%2f
%2fwww.robinwieruch.de%2fgraphql-
apollo-server-tutorial%2f)

 (http
/share?text=GraphQL%20Server%20Tutorial
by %40rwieruch %23ReactJs'
%2fwww.robinwieruch.de%2f

```

...

const user = (sequelize, DataTypes) => {
  const User = sequelize.define('user', {
    username: {
      type: DataTypes.STRING,
      unique: true,
      allowNull: false,
      validate: {
        notEmpty: true,
      },
    },
    email: {
      type: DataTypes.STRING,
      unique: true,
      allowNull: false,
      validate: {
        notEmpty: true,
        isEmpty: true,
      },
    },
    password: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        notEmpty: true,
        len: [7, 42],
      },
    },
  });

  ...

  return User;
};


export default user;

```

As you can see, the two new entries for the user model have their own validation rules as you have done them in the previous section. For instance, the password of a user should be between 7 and 42 characters and the email should have a valid email format. If any of these validations fails during the user creation, it generates a JavaScript error, transforms and transfers the error with GraphQL, and thus can be used in the client application. The registration form in the client application could display the validation error then.

You may want to add the email, but not the password, to your GraphQL user schema in the `src/schema/user.js` file too:

 (<https://www.facebook.com/sharer>

 (<https://twitter.com>

[/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f) [/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2f](https://twitter.com/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2f)

```
import { gql } from 'apollo-server-express';

export default gql`
  ...

  type User {
    id: ID!
    username: String!
    email: String!
    messages: [Message!]
  }
`;

```

Next, don't forget to add the new properties to your seed data in the *src/index.js* file:

```
const createUsersWithMessages = async () => {
  await models.User.create(
    {
      username: 'rwieruch',
      email: 'hello@robin.com',
      password: 'rwieruch',
      messages: [ ... ],
    },
    {
      include: [models.Message],
    },
  );

  await models.User.create(
    {
      username: 'ddavids',
      email: 'hello@david.com',
      password: 'ddavids',
      messages: [ ... ],
    },
    {
      include: [models.Message],
    },
  );
};

```

That's it for the data migration of your database in order to get started with the GraphQL authentication.

Registration (Sign Up) with GraphQL

Now, let's get into the implementation details for the GraphQL authentication. In the following, you are going to implement two GraphQL mutations: one to register a user and one to login a user to the application. Let's start with the sign up mutation in the GraphQL schema in the *src/schema/user.js* file:

<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f> by <https://www.robinwieruch.de%2f>

```
import { gql } from 'apollo-server-express';

export default gql`
  extend type Query {
    users: [User!]
    user(id: ID!): User
    me: User
  }

  extend type Mutation {
    signUp(
      username: String!
      email: String!
      password: String!
    ): Token!
  }

  type Token {
    token: String!
  }

  type User {
    id: ID!
    username: String!
    messages: [Message!]
  }
`;

```

The `signUp` mutation takes three non-nullable arguments: `username`, `email` and `password`. All of these things are used to create a user in the database eventually. When a user signs in to the application (after a successful sign up), the user should be able to take the username or email address combined with the password for the login. So what about the return type of the `signUp` mutation? Since we are going to use a token based authentication with GraphQL, it is sufficient to return a token which is nothing more than a string. However, in order to distinguish the token in the GraphQL schema, it has its own GraphQL type. You will learn more about the token in the following, because the token is all about the authentication mechanism for this application.

First, you can add the counterpart for your new mutation in the GraphQL schema as resolver function. In your `src/resolvers/user.js` file, add the following resolver function which creates a user in the database and returns an object with the token value as string.

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

```

const createToken = async (user) => {
  ...
};

export default {
  Query: {
    ...
  },

  Mutation: {
    signUp: async (
      parent,
      { username, email, password },
      { models },
    ) => {
      const user = await models.User.create({
        username,
        email,
        password,
      });

      return { token: createToken(user) };
    },
  },
  ...
};

```


Basically that's the necessary GraphQL framework around a token based registration. You have created the necessary GraphQL mutation and resolver for it, which creates a user in the database (based on certain validations) and its incoming resolver arguments, and finally creates a token for the registered user. The latter will be explained in the following sections. But for now, all you have set up is sufficient to create (register, sign up) a new user with a GraphQL mutation.

Securing Passwords with Bcrypt

There is one major security flaw in this code: the password of a user is stored in plain text in the database. You should never do this, because when your database gets hacked or some other third-party gets access to it, you made it very easy for the attacker to get all the passwords in plain text. That's why you can use something like bcrypt (<https://github.com/kelektiv/node.bcrypt.js>) for hashing your passwords. First, install it on the command line:

```
npm install bcrypt --save
```

 (<https://www.facebook.com/sharer>

 (<https://twitter.com/sharer>

Now, it is possible to hash the password with bcrypt in the user's resolver function when it gets created on a 'signUp' mutation. However, there is an alternative way of doing it with Sequelize. In your `models` folder, you can create a `hashPassword` function which is executed by `sequelize`. Ready to try it? <https://www.robinwieruch.de/graphql-apollo-server-tutorial/>


```
const user = (sequelize, DataTypes) => {
  const User = sequelize.define('user', {
    ...
  });

  ...

  User.beforeCreate(user => {
    ...
  });

  return User;
};

export default user;
```

In this hook function, you can add the functionalities to alter your user entity's properties before they reach the database. So let's do it for the hashed password by using bcrypt.

```
import bcrypt from 'bcrypt';

const user = (sequelize, DataTypes) => {
  const User = sequelize.define('user', {
    ...
  });

  ...

  User.beforeCreate(async user => {
    user.password = await user.generatePasswordHash();
  });

  User.prototype.generatePasswordHash = async function() {
    const saltRounds = 10;
    return await bcrypt.hash(this.password, saltRounds);
  };

  return User;
};

export default user;
```

The `bcrypt hash()` method takes a string (here the user's password) and an integer called salt rounds. What are salt rounds? Basically each salt round makes it more costly to hash the password. In return, it makes it more costly for attackers to decrypt the hash value. A common value for salt rounds nowadays ranged from 10 to 12, because if you would increase the number of salt rounds, you may run into performance issue yourself (and not only the attacker).

The `generatePasswordHash()` function is added to the user entity as a prototype method. That's why it is possible to execute the function as method on each user instance and thus you have

the user itself available within the method as `this`. You could have used another approach of doing it, for instance as function which takes the user instance with its password as argument (which would be my personal preference), but using JavaScript's prototypal inheritance is sometimes a good task to keep this knowledge in your tool chain as a developer. As for now, every time a user is created in the database, the password is hashed with `bcrypt` before it gets stored.

Token based Authentication in GraphQL

Now what about the token based authentication? So far, there is only a placeholder in your application for creating the token which should be returned on a sign up (and sign in) mutation.

Essentially the token is only important for a user who signs in to your application. A signed in user can be identified with this token and thus is able to read and write data from the database. Since a registration (sign up) will automatically lead to a login (sign in), the token is generated already in the registration phase and not only in the login phase. You will see later how the token is generated for a login as well.

Let's get into the implementation details for the token based authentication in GraphQL. Regardless of GraphQL, you are going to use a JSON web token (JWT) (<https://jwt.io/>) to identify your user. This approach is not only used in GraphQL applications. The definition for a JWT from the official website says: *JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties*. In other words, a JWT is a secure way to handle the communication between two parties (e.g. a client and a server application). If you haven't done anything security related before, don't worry too much about it. The following section will guide you through the process and in the end, the token is nothing else than a secured JavaScript object with user information.

In order to create JWT in this application, you are going to use the popular `jsonwebtoken` (<https://github.com/auth0/node-jsonwebtoken>) node package, so you can simply install it on the command line:

```
npm install jsonwebtoken --save
```

Now, you can import it in your `src/resolvers/user.js` file and use it for creating the token:

f (<https://www.facebook.com/sharer>) **t** (<https://twitter.com/rwieruch>)
/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f /share?text=GraphQL%20Server%20Tutorial
by %40rwieruch %23ReactJs%2fwww.robinwieruch.de%2f

```
import jwt from 'jsonwebtoken';

const createToken = async user => {
  const { id, email, username } = user;
  return await jwt.sign({ id, email, username });
};

...
```

The first argument to “sign” a token can be any user information except for sensible data such as passwords, because the token ends up on the client side of your application stack too. Signing a token means putting data into it (as you did) and securing it (which you haven’t done yet). In order to secure your token, you need to pass in a secret (**any** long string) which is **only available to you and your server**. No third-party should have access to it, because it is used to encode (sign) and decode your token. For instance, you can add the secret to your environment variables in the `.env` file:

```
DATABASE=postgres
DATABASE_USER=postgres
DATABASE_PASSWORD=postgres


SECRET=wr3r23fwfwefwekwsself.2456342.dawqdg
```

Then, in the `src/index.js` file, you can pass the secret via Apollo Server’s context to all resolver functions:

```
const server = new ApolloServer({
  typeDefs: schema,
  resolvers,
  ...
  context: async () => ({
    models,
    me: await models.User.findByLogin('rwieruch'),
    secret: process.env.SECRET,
  }),
});
```

Next, you can use it in your `signUp` resolver function by passing it to the token creation. The `sign` method of JWT takes care of the rest. In addition, you can pass in a third argument for setting an expiration date for a token. In this case, the token is only valid for 30 minutes. Afterward, a user would

need to sign in again.

 (http

<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f> [/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2f](https://twitter.com/sharer?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2f)

```
import jwt from 'jsonwebtoken';

const createToken = async (user, secret, expiresIn) => {
  const { id, email, username } = user;
  return await jwt.sign({ id, email, username }, secret, {
    expiresIn,
  });
};

export default {
  Query: {
    ...
  },

  Mutation: {
    signUp: async (
      parent,
      { username, email, password },
      { models, secret },
    ) => {
      const user = await models.User.create({
        username,
        email,
        password,
      });

      return { token: createToken(user, secret, '30m') };
    },
  },
  ...
};
```

Now you have secured your information in the token as well. If you would want to decode it, in order to access the secured data (the first argument of the `sign` method), you would need the secret again. Furthermore, the token is only valid for 30 minutes.

That's it for the registration: you are creating a user and return a valid token which can be used from the client application to authenticate the user again. The server can decode the token, which comes with every request, and allows the user to access the more sensible data of the application. You can try out the registration with GraphQL Playground yourself, which should create a user in the database and return a token for it. Furthermore, you can check your database with `psql` whether the user got created and whether the user has a hashed password instead of a plain text password.

Login (Sign In) with GraphQL

Before we can use the token on a per request basis, let's implement the second mutation for completing the authentication mechanism: the `signIn` mutation (or login mutation). Again, first add the GraphQL mutation to your user's schema in the `src/schema/user.js` file:

apollo-server-tutorial%2f)

%2fwww.robinwieruch.de%2f

```
import { gql } from 'apollo-server-express';

export default gql`
  ...

  extend type Mutation {
    signUp(
      username: String!
      email: String!
      password: String!
    ): Token!


    signIn(login: String!, password: String!): Token!
  }

  type Token {
    token: String!
  }

  ...
`;
```

Second, add the resolver counterpart to your `src/resolvers/user.js` file:

 (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

 (<https://twitter.com/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

```

import jwt from 'jsonwebtoken';
import { AuthenticationError, UserInputError } from 'apollo-server';

...

export default {
  Query: {
    ...
  },

  Mutation: {
    signUp: async (...) => {
      ...
    },

    signIn: async (
      parent,
      { login, password },
      { models, secret },
    ) => {
      const user = await models.User.findByLogin(login);

      if (!user) {
        throw new UserInputError(
          'No user found with this login credentials.',
        );
      }

      const isValid = await user.validatePassword(password);

      if (!isValid) {
        throw new AuthenticationError('Invalid password.');
      }

      return { token: createToken(user, secret, '30m') };
    },
  },
  ...
};

```

Let's go through the new resolver function for the login step by step. As arguments the resolver has access to the input arguments from the GraphQL mutation (login, password) and the context (models, secret). When a user tries to sign in to your application, the login, which can be either the unique username or unique email, is taken to retrieve a user from the database. If there is no user, the application should throw a error which can be used in the client application to show an error for the login form. If there is an user, the user's password is validated. You will see this method on the user model in a moment. If the password is not valid, the application throws an error again for the client application. If the password is valid, the `signIn` mutation returns a token again (identical to the `signUp` mutation). So after all, the client application either performs a successful login or shows an error message for either the invalid credentials, because no user by username or email is found in the database or the invalid password. In addition, you can see specific `AuthenticationError` and `UserInputError` used over generic JavaScript Error classes.

Last but not least, what about the `validatePassword()` method on the user instance? Let's implement it in the `src/models/user.js` file, because that's where all the model methods for the user can be implemented (same as the `findByLogin()` method):

```
import bcrypt from 'bcrypt';

const user = (sequelize, DataTypes) => {
  ...

  User.findByLogin = async login => {
    let user = await User.findOne({
      where: { username: login },
    });

    if (!user) {
      user = await User.findOne({
        where: { email: login },
      });
    }

    return user;
  };

  User.beforeCreate(async user => {
    user.password = await user.generatePasswordHash();
  });

  User.prototype.generatePasswordHash = async function() {
    const saltRounds = 10;
    return await bcrypt.hash(this.password, saltRounds);
  };

  User.prototype.validatePassword = async function(password) {
    return await bcrypt.compare(password, this.password);
  };

  return User;
};

export default user;
```

Again, it's a JavaScript prototypal inheritance for making a method available on the user instance. In this method, the user (`this`) and its password can be compared with the incoming password (coming from the GraphQL mutation) by using `bcrypt`, because the password on the user is the hashed password and the incoming password is the plain text password. After all, `bcrypt` will tell you whether the password is correct or not when a user signs in.

Now you have set up everything for the registration (sign up) and login (sign in) for your GraphQL server application. You have a `hash` function to hash (share on Facebook (https://www.facebook.com/sharer.php?u=https://www.robinwieruch.de/graphql-apollo-server-tutorial)) and a `validatePassword` function and you have used JWT to encrypt user data by a secret to a token. Then the token is returned on every sign up and sign in. Then the client application can save the token (e.g. local storage of the browser) and send it along with every request to the server.

GraphQL query and mutation as authorization.

So what should you do with the token once a user is authenticated with your application after a successful registration or login? That's what the next section will teach you about authorization in GraphQL on the server-side.

Exercises:

- register (sign up) a new user with GraphQL Playground
- check your users and their hashed passwords in the database with `psql`
- read more about JSON web tokens (JWT) (<https://jwt.io/>)
- login (sign in) a user with GraphQL Playground
 - copy and paste the token to the interactive token decoding on the JWT website (conclusion: the information itself isn't secure, that's why you shouldn't put a password in the token)

Authorization with GraphQL and Apollo Server

In the last section, you have set up all GraphQL mutations to enable the authentication with the server. You can register a new user with bcrypt hashed passwords and you are able to login with your user's credentials. Both authentication related GraphQL mutations return a token (JWT) for you which secures non-sensible user information with a secret. So what can be done once you have obtained the token in your client application?

The token, whether obtained on registration or login, is returned to the client application after a successful GraphQL `signIn` or `signUp` mutation. The client application has to make sure to store the token somewhere (e.g. the browser's session storage (<https://www.robinwieruch.de/local-storage-react>)). Then, every time a request is made to the GraphQL server, the token has to be attached to the HTTP header of the HTTP request. Then the GraphQL server can validate the HTTP header, verify its authenticity, and perform the actual request (e.g. GraphQL operation). If the token is not valid (anymore), the GraphQL server has to return an error for the GraphQL client. If the client still has a token locally stored, it should remove the token from the storage and redirect the user to the login page.

So how to perform the server part of the equation? Let's do it in the `src/index.js` file by adding a global authorization which verifies the incoming token before the request hits the GraphQL resolvers.

<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f> [https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%40rwieruch%23ReactJs%20www.robinwieruch.de%2f](https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2f)


```

import jwt from 'jsonwebtoken';
import {
  ApolloServer,
  AuthenticationError,
} from 'apollo-server-express';
...

const getMe = async req => {
  const token = req.headers['x-token'];

  if (token) {
    try {
      return await jwt.verify(token, process.env.SECRET);
    } catch (e) {
      throw new AuthenticationError(
        'Your session expired. Sign in again.',
      );
    }
  }
};

const server = new ApolloServer({
  typeDefs: schema,
  resolvers,
  ...
  context: async ({ req }) => {
    const me = await getMe(req);

    return {
      models,
      me,
      secret: process.env.SECRET,
    };
  },
});
...

```

So what's happening in this general authorization on the server-side? Basically you are injecting the `me` user, which is the authenticated user from the token, with every request to your Apollo Server's context. The `me` user is the user which you encode in the token in your `createToken()` function. It's not a user from the database anymore which is good, because you can spare the request to the database.

In the `getMe()` function, you extract the HTTP header for the authorization, which is called “x-token” in this case (common way of naming it), from the incoming HTTP request. That's because the GraphQL client application will send the token obtained from the registration or login with every other request as a HTTP header along with the actual payload of the HTTP request (e.g. GraphQL operation). Then, in the function, it can be checked whether there is such a HTTP header or not. If not, the function simply continues with the request, but the `me` user will be undefined then. If there is a token, the function verifies the token with its secret and retrieves the user information (which was stored when you created the token) from the token. If the verification fails, because there was a

header but the token was invalid or expired, the GraphQL server throws a specific Apollo Server Error. If the verification succeeds, the function continues as well, but this time with the `me` user defined.

So the only case when the function returns an error is when the client application sends a HTTP header with a token which is invalid or expired. Otherwise, the function waves the request through, because then on a resolver level it must be checked whether a user is allowed to perform certain operations or not. For instance, a non authenticated user (`me` user is undefined) may be able to retrieve a list of messages, but not to create a new message. So the application is protected at least against invalid and expired tokens now.



Basically that's it for the most high level authentication and authorization for your GraphQL server application. You are able to authenticate with your GraphQL server from a GraphQL client application with the `signUp` and `signIn` GraphQL mutations and your GraphQL server only allows valid and non expired tokens which are coming with every other GraphQL operation from a GraphQL client application. But what about a more fine-grained authorization for specific GraphQL queries and mutations?

GraphQL Authorization on a Resolver Level

As you have noticed, a GraphQL HTTP request comes through the `getMe()` function even though if it has no HTTP header for a token. It's a good default behavior, because you want to be able to register a new user or login as a user to the application without having a token yet. Moreover, you may want to query messages or users without being authenticated with the application. That's why it's okay to wave through the request even though there is no authorization token. Only when the token is invalid or expired, there will be an error.

However, certain GraphQL operations should have more fine-grained authorization too. For instance, creating a message should only be possible when you are authenticated as a user. Otherwise, who should be the creator of the message in the first place? So let's see how the `createMessage` GraphQL mutation can be protected (also called: guarded) on a GraphQL resolver level.

The naive approach of protecting the GraphQL operation would be to guard it with an if-else statement in the `src/resolvers/message.js` file:

 (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)  (<https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

```
import { ForbiddenError } from 'apollo-server';

export default {
  Query: {
    ...
  },

  Mutation: {
    createMessage: async (parent, { text }, { me, models }) => {
      if (!me) {
        throw new ForbiddenError('Not authenticated as user.');
      }

      return await models.Message.create({
        text,
        userId: me.id,
      });
    },
    ...
  },
  ...
};
```

You can imagine how this becomes repetitive and error prone when using this approach for all the GraphQL operations which are only accessible as authenticated user. You would mix lots of authorization logic into your resolver functions. So how to introduce an authorization abstraction layer for protecting those GraphQL operations? The solutions to it are so called **combined resolvers** or a **resolver middleware**. Without having to implement these things ourselves, let's install a node package for the former solution:

```
npm install graphql-resolvers --save
```

Let's implement a protecting resolver function with this package in a new `src/resolvers/authorization.js` file. It should only check whether there is a `me` user or not.

```
import { ForbiddenError } from 'apollo-server';
import { skip } from 'graphql-resolvers';

export const isAuthenticated = (parent, args, { me }) =>
  me ? skip : new ForbiddenError('Not authenticated as user.');
```

Basically the new `isAuthenticated()` resolver function acts as some kind of middleware. It either continues with the next resolver (`skip`) or does something else (throw an error). In this case `True` is returned when the `me` user is not available. Since it is a resolver function itself, it has the same arguments as a normal resolver.

Now, this guarding resolver can be used when creating a message in the *src/resolvers/message.js* file. You need to import it along with the `combineResolvers()` from the newly installed node package. Then the new resolver can be used to protect the actual resolver by combining them.

```
import { combineResolvers } from 'graphql-resolvers';

import { isAuthenticated } from './authorization';

export default {
  Query: {
    ...
  },

  Mutation: {
    createMessage: combineResolvers(
      isAuthenticated,
      async (parent, { text }, { models, me }) => {
        return await models.Message.create({
          text,
          userId: me.id,
        });
      },
    ),
    ...
  },
  ...
};
```

Now the `isAuthenticated()` resolver function always runs before the actual resolver that creates the message associated with the authenticated user in the database. The resolvers get chained after each other. The great thing about it: You can reuse the protecting resolver function wherever you need it. It only adds a small footprint to your actual resolvers and you can change it at one place in your *src/resolvers/authorization.js* file.

Permission-based GraphQL Authorization

The previous resolver only checks whether a user is authenticated or not. It is only applicable on a higher level. What about more specific use cases such as permissions? Let's add another protecting resolver which is more specific than the previous one in the *src/resolvers/authorization.js* file:

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

🐦 (<https://twitter.com/rwieruch>)

/share?text=GraphQL%20Server%20Tutorial by %40rwieruch %23ReactJs%2fwww.robinwieruch.de%2f

```

...

export const isMessageOwner = async (
  parent,
  { id },
  { models, me },
) => {
  const message = await models.Message.findById(id, { raw: true });

  if (message.userId !== me.id) {
    throw new ForbiddenError('Not authenticated as owner.');
```

This resolver checks whether the authenticated user is a message owner. It's the perfect check before deleting a message if only the message creator should be able to delete it. So the guarding resolver retrieves the message by id, checks the message's associated user with the authenticated user, and either throws an error or continues with the next resolver. Let's protect an actual resolver with this fine-grained authorization permission resolver in the `src/resolvers/message.js` file:

```

import { combineResolvers } from 'graphql-resolvers';

import { isAuthenticated, isMessageOwner } from '../authorization';

export default {
  Query: {
    ...
  },

  Mutation: {
    ...

    deleteMessage: combineResolvers(
      isMessageOwner,
      async (parent, { id }, { models }) => {
        return await models.Message.destroy({ where: { id } });
      },
    ),
  },

  ...
};
```

The `deleteMessage` resolver is protected with an authorization resolver now. Only a message owner, if a message owner is allowed to delete a message. But what about if the user isn't authenticated in the first place? Therefore you can stack your protecting resolvers onto each other:

<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>

by %40rwieruch %23ReactJs%2fwww.robinwieruch.de%2f

```
import { combineResolvers } from 'graphql-resolvers';

import { isAuthenticated, isMessageOwner } from './authorization';

export default {
  Query: {
    ...
  },

  Mutation: {
    ...

    deleteMessage: combineResolvers(
      isAuthenticated,
      isMessageOwner,
      async (parent, { id }, { models }) => {
        return await models.Message.destroy({ where: { id } });
      },
    ),
  },
  ...
};
```

In this case, first it is checked whether the user is authenticated and second whether the user is the owner of the message before deleting it.

Another spin on this would have been using the `isAuthenticated` resolver directly in the `isMessageOwner` resolver, then you would never have to deal with this in the actual resolver for deleting a message. But personally I find it more explicit doing it the other way than hiding this knowledge within the authorization resolver. You will find it the other way around in the role-based authorization section.

The second combined resolver is one way of doing permission checks, because it checks whether the user has the permission to delete the message. There are different approaches to achieve those kind of permission checks and this is only one way of doing it. In other cases, the message itself may have a boolean flag (with respect to the context which is the authenticated user) whether it is possible to delete it.

Role-based GraphQL Authorization

Previously you went from a high-level authorization to a more fine-grained authorization with permission based resolver protection. Another way of doing authorization are roles. In the following, you will implement a new GraphQL mutation which needs to have role-based authorization, because it has the ability to delete a user. So who should be able to delete a user? Maybe only a user with an admin role. So let's implement the new GraphQL mutation first, followed by the role-based authorization. You will start in the `resolvers/user.js` file with a resolver function which deletes a user in the database by identifier.

```

...

export default {
  Query: {
    ...
  },

  Mutation: {
    ...

    deleteUser: async (parent, { id }, { models }) => {
      return await models.User.destroy({
        where: { id },
      });
    },
  },
  ...
};

```

Every time you implement a new GraphQL operation, you have to do it in your resolvers and schema. So let's add the new mutation in your GraphQL schema in the `src/schema/user.js` file as well. It should only return a boolean which tells you whether the deletion was successful or not:

```

import { gql } from 'apollo-server-express';

export default gql`
  extend type Query {
    ...
  }

  extend type Mutation {
    signUp(
      username: String!
      email: String!
      password: String!
    ): Token!


    signIn(login: String!, password: String!): Token!
    deleteUser(id: ID!): Boolean!
  }

  ...
`;

```

Your new GraphQL mutation should work already. But every one is able to execute it. Now, before implementing the role-based protection for it, you have to introduce the actual roles for the user

entries! Therefore, add a `role` entry to your user's entity in the `src/models/user.js` file:

 (http

<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f> /share?text=GraphQL%20Server%20Tutorial by %40rwieruch %23ReactJs%2fwww.robinwieruch.de%2f

```

...

const user = (sequelize, DataTypes) => {
  const User = sequelize.define('user', {
    ...
    password: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        notEmpty: true,
        len: [7, 42],
      },
    },
    role: {
      type: DataTypes.STRING,
    },
  });

  ...

  return User;
};

export default user;

```

You may want to add the role to your GraphQL user schema in the *src/schema/user.js* file too:

```

import { gql } from 'apollo-server-express';

export default gql`
  ...

  type User {
    id: ID!
    username: String!
    email: String!
    role: String
    messages: [Message!]
  }
`;

```

Since you already have seed data in your *src/index.js* file for two users, you can give one of the two users a role, in this case an admin role, which can be checked later when deleting a user.

f ([https://www.facebook.com/sharer](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)
[/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f))

t (<https://twitter.com/rwieruch>)
[/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f](https://twitter.com/rwieruch)


```

...

const createUsersWithMessages = async () => {
  await models.User.create(
    {
      username: 'rwieruch',
      email: 'hello@robin.com',
      password: 'rwieruch',
      role: 'ADMIN',
      messages: [
        {
          text: 'Published the Road to learn React',
        },
      ],
    },
    {
      include: [models.Message],
    },
  );

  ...
};

```

Because you are never retrieving the actual `me` user from the database in the `src/index.js` file, but only used the user from the token, you have to add the role information of the user to the token when it gets created in the `src/resolvers/user.js` file:

```

const createToken = async (user, secret, expiresIn) => {
  const { id, email, username, role } = user;
  return await jwt.sign({ id, email, username, role }, secret, {
    expiresIn,
  });
};

```

Now you have introduced a new GraphQL mutation for deleting a user and roles for users. One of your users should be an admin user too. In the next steps, you are going to protect the new GraphQL mutation with a role-based authorization. Therefore, create a new guarding resolver in your `src/resolvers/authorization.js` file:

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/rwieruch>)

/share?text=GraphQL%20Server%20Tutorial
by %40rwieruch %23ReactJs%2fwww.robinwieruch.de%2f

```

import { ForbiddenError } from 'apollo-server';
import { combineResolvers, skip } from 'graphql-resolvers';

export const isAuthenticated = (parent, args, { me }) =>
  me ? skip : new ForbiddenError('Not authenticated as user.');
```

```

export const isAdmin = combineResolvers(
  isAuthenticated,
  (parent, args, { me: { role } }) =>
    role === 'ADMIN'
      ? skip
      : new ForbiddenError('Not authorized as admin.'),
);
```

```

export const isMessageOwner = async (
  parent,
  { id },
  { models, me },
) => {
  const message = await models.Message.findById(id, { raw: true });

  if (message.userId !== me.id) {
    throw new ForbiddenError('Not authenticated as owner.');
```

```

  }

  return skip;
};
```

The new resolver checks whether the authenticated user has the `ADMIN` role or not. If it hasn't the role, the resolver returns an error. If it has the role, the next resolver will be called. In addition, in contrast to the `isMessageOwner` resolver, the `isAdmin` resolver already is a combined resolver which makes use of the `isAuthenticated` resolver. Then you don't need to worry about this check in your actual resolver, which you are going to protect in the next step:

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

```

import jwt from 'jsonwebtoken';
import { combineResolvers } from 'graphql-resolvers';
import { AuthenticationError, UserInputError } from 'apollo-server';

import { isAdmin } from './authorization';

...

export default {
  Query: {
    ...
  },

  Mutation: {
    ...

    deleteUser: combineResolvers(
      isAdmin,
      async (parent, { id }, { models }) => {
        return await models.User.destroy({
          where: { id },
        });
      },
    ),
  },
  ...
};

```

That's it for the role-based authorization in GraphQL with Apollo Server. In this case, the role is only a string which needs to be checked. In a more elaborated role-based architecture, you may want to change the role from a string to an array of multiple roles. Then you don't have to make an equal check anymore but instead check whether the array includes the desired role. That's one way of doing a more sophisticated role-based authorization setup.

Setting Headers in GraphQL Playground

In the previous sections, you have learned how to setup authorization for your GraphQL application. But how to verify that it is working? You only need to head over to your GraphQL Playground and run through the different scenarios. Let's do it together for the user deletion scenario, but all the remaining scenarios should be verified on your own (exercise).

Before you can delete a user, you need to sign in to the application first. Let's execute a `signIn` mutation in GraphQL Playground but with a non admin user. You can repeat the walkthrough another time with an admin user afterward.

f (<https://www.facebook.com/sharer>

t (<https://twitter.com>

[/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

[/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f](https://twitter.com/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

```
mutation {
  signIn(login: "ddavids", password: "ddavids") {
    token
  }
}
```

In your GraphQL Playground result, you should get the token after the login. For the next GraphQL operations, the token needs to be set in the HTTP header. GraphQL Playground has a panel to add HTTP headers. Since your application is checking for a x-token, you need to set the token as one:

```
{
  "x-token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MiwiZW1haWwiOiJoZWxsbyBkYXZpZC5jb20iLCJ1c2VybmFtZSI6ImRkYXZpZHMlLCJpYXQiOi0jE1MzQ5MjM4NDcsImV4cCI6MTUzNDkyNTY0N30.ViGU6UUY-XWpWDJGfXqES2J1lEr-Uye8XDQ79lAvByE"
}
```

In your case, the token should be different. Since the token is set as HTTP header now, you should be able to delete a user. Let's try it with the following GraphQL mutation in GraphQL Playground. The HTTP header with the token will be send along with the GraphQL operation:

```
mutation {
  deleteUser(id: "2")
}
```

Instead of seeing a successful request, you should see the following GraphQL error after executing the GraphQL mutation for deleting a user. That's because you haven't logged in as a user with an admin role.

```
{
  "data": null,
  "errors": [
    {
      "message": "Not authorized as admin.",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "deleteUser"
      ],
      "extensions": {

```

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/sharer?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2f>)

/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2f

The shown way for doing authorization in GraphQL was only one approach of doing it. If you want to be even more fine-grained than resolver level authorization, checkout **directive-based authorization** or **field level authorization** in GraphQL. Another way would be to apply authorization on a data access level with the help of your ORM (here Sequelize). In the end, it comes down to your requirements and your application on which level of your application you want to introduce authorization.

- read more about GraphQL authorization (<https://graphql.github.io/learn/authorization/>)
- play through the different authorization scenarios with GraphQL Playground
- find out more about field level authorization with Apollo Server and GraphQL
- find out more about data access level authorization with Apollo Server and GraphQL

<https://www.robinwieruch.de/pagination-in-graphql-with-apollo-server/>

Sooner or later you will run into a feature called pagination when developing applications with lists of items. For instance, the messages in your application for a user, when thinking about a chat application, can become a very long list with lots of messages. When a client application requests the messages of a user in order to display them, you don't want to retrieve all messages at once from the server application (and database), because it will become a performance bottleneck.

So what can be done to solve this problem? The feature to solve this is called pagination and allows you to split up a list of items into multiple lists (pages). Such page is usually (depends on the pagination strategy) defined with a limit (how many items) and an offset (index in the list, starting point for the limit). That way, you can request one page of items (100 list items) and later, when a user scrolls through the items in the client application and wants to see more items, request another page of items (next 100 list items with an offset of the first 100 list items). Doesn't sound too complicated, does it?

Let's implement pagination in GraphQL with two approaches in the following sections. The first approach will be the most naive approach (**offset/limit-based pagination**) to implement it. Afterward, you will see an advanced approach (**cursor-based pagination**) which is only one way of doing a more sophisticated pagination implementation.

Offset/Limit Pagination with Apollo Server and GraphQL

The offset/limit-based pagination isn't too difficult to implement. As mentioned previously, the limit states how many items you want to retrieve from the entire list and the offset states where to begin in the entire list. By using different offsets, you can shift through the entire list of items and retrieve only a sublist (page) of it with the limit. Therefore, the message schema in the `src/schema/message.js` file has to consider these two new arguments:

```
import { gql } from 'apollo-server-express';

export default gql`
  extend type Query {
    messages(offset: Int, limit: Int): [Message!]!
    message(id: ID!): Message!
  }

  extend type Mutation {
    createMessage(text: String!): Message!
    deleteMessage(id: ID!): Boolean!
  }
`;
```

f (<https://www.facebook.com/sharer>

🐦 (<https://twitter.com/sharer>

/share.php?u=https%3a%2f

/share?text=GraphQL%20Server%20Tutorial

%2fwww.robinwieruch.de%2fgraphql-

by %40rwieruch %23ReactJs'

`; apollo-server-tutorial%2f)

%2fwww.robinwieruch.de%2f

Then you can adjust the resolver in the `src/resolvers/message.js` file to deal with these two new arguments:

```
...

export default {
  Query: {
    messages: async (
      parent,
      { offset = 0, limit = 100 },
      { models },
    ) => {
      return await models.Message.findAll({
        offset,
        limit,
      });
    },
    message: async (parent, { id }, { models }) => {
      return await models.Message.findById(id);
    },
  },
  Mutation: {
    ...
  },
  ...
};
```

Fortunately your ORM (Sequelize) gives you everything you need to make it happen with the in-house offset and limit functionality. That's it for the offset- and limit-based pagination feature itself. You can try it in GraphQL Playground yourself by adjusting the limit and offset.

```
query {
  messages(offset: 1, limit: 2){
    text
  }
}
```

Even though this approach is simpler to implement and understand, it comes with a few disadvantages. For instance, when your offset becomes very long, the database query takes longer. This can lead to a bad performance on the client-side while waiting for the next page of data. In addition, the offset/limit pagination cannot handle the edge case of deleted items in between. For instance, imagine you query the first page of data. Then someone deletes one item in the first page. When requesting the next page of data, the offset would be wrong, because it would miss one item due to the deletion. You can not easily overcome this problem with offset- and limit-based pagination. That's why there exists another more sophisticated approach for pagination: cursor-based pagination. <https://www.robinwieruch.de/graphql-apollo-server-tutorial/> by %40rwieruch %29ReactJs %2fwww.robinwieruch.de%2f

Cursor-based Pagination with Apollo Server and GraphQL

In cursor-based pagination, you give your offset an identifier (called cursor) rather than just counting the items as in the previous approach. This cursor can be used to say: “give me a limit of X items from cursor Y”. So what should be the cursor to identify an item in the list? A common approach is using dates (e.g. creation date of an entity in the database). In our case, each message already has a `createdAt` date which is assigned to the entity when it written to the database. So let's extend the `src/schema/message.js` which this field for a message. Afterward, you should be able to query this field in GraphQL Playground too:

```
import { gql } from 'apollo-server-express';


export default gql`
  extend type Query {
    messages(cursor: String, limit: Int): [Message!]!
    message(id: ID!): Message!
  }

  extend type Mutation {
    createMessage(text: String!): Message!
    deleteMessage(id: ID!): Boolean!
  }

  type Message {
    id: ID!
    text: String!
    createdAt: String!
    user: User!
  }
`;
```

Next, in order to test the cursor-based pagination based on the creation date of an entity in a more robust way, you need to adjust your seed data in the `src/index.js` file. At the moment, all seed data is created at once which applies to the messages as well. However, it would be beneficial to have each message created in one second intervals because the creation date should differ for each message to see the effect of the pagination based on this date:

 (https://www.facebook.com/sharer
 /sharer.php?u=https%3a%2f
 %2fwww.robinwieruch.de%2fgraphql-
 apollo-server-tutorial%2f)

 (http
 /share?text=GraphQL%20Server%20Tutorial
 by %40rwieruch %23ReactJs%
 %2fwww.robinwieruch.de%2f)


```

...

sequelize.sync({ force: eraseDatabaseOnSync }).then(async () => {
  if (eraseDatabaseOnSync) {
    createUsersWithMessages(new Date());
  }


  app.listen({ port: 8000 }, () => {
    console.log('Apollo Server on http://localhost:8000/graphql (http://localhost:8000/graphql)');
  });
});

const createUsersWithMessages = async date => {
  await models.User.create(
    {
      username: 'rwieruch',
      email: 'hello@robin.com',
      password: 'rwieruch',
      role: 'ADMIN',
      messages: [
        {
          text: 'Published the Road to learn React',
          createdAt: date.setSeconds(date.getSeconds() + 1),
        },
      ],
    },
    {
      include: [models.Message],
    },
  );

  await models.User.create(
    {
      username: 'ddavids',
      email: 'hello@david.com',
      password: 'ddavids',
      messages: [
        {
          text: 'Happy to release ...',
          createdAt: date.setSeconds(date.getSeconds() + 1),
        },
        {
          text: 'Published a complete ...',
          createdAt: date.setSeconds(date.getSeconds() + 1),
        },
      ],
    },
    {
      include: [models.Message],
    },
  );
};

```

 (https://www.facebook.com/sharer

 (http

That's it for the cursor which will be the creation date of each message. Now, let's advance the previous pagination to cursor-based pagination in the `src/schema/message.js` file. You only need to exchange the offset with the cursor. So instead of having an offset which can only be matched implicitly, we can use a cursor which is unique once an item is deleted from the database.

position within the list, because the creation dates of the messages will not change.

```
import { gql } from 'apollo-server-express';

export default gql`
  extend type Query {
    messages(cursor: String, limit: Int): [Message!]!
    message(id: ID!): Message!
  }

  extend type Mutation {
    createMessage(text: String!): Message!
    deleteMessage(id: ID!): Boolean!
  }

  type Message {
    id: ID!
    text: String!
    createdAt: String!
    user: User!
  }
`;
```


As you have adjusted the schema for the messages, you need to reflect the change in your `src/resolvers/message.js` file too:

```
import Sequelize from 'sequelize';

...

export default {
  Query: {
    messages: async (parent, { cursor, limit = 100 }, { models }) => {
      return await models.Message.findAll({
        limit,
        where: {
          createdAt: {
            [Sequelize.Op.lt]: cursor,
          },
        },
      });
    },
    message: async (parent, { id }, { models }) => {
      return await models.Message.findById(id);
    },
  },
  Mutation: {
```

 (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

 (<https://twitter.com/rwieruch>)

[/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f](https://www.robinwieruch.de/graphql-apollo-server-tutorial/)

Instead of using the offset, you are using the cursor which is the `createdAt` property of a message. When using Sequelize (but also any other ORM), it is possible to add a clause to find all items in a list by a starting property (here `createdAt`) with less than (`lt`) or greater than (`gt`, which is not used here) values for this property. In this case, when providing a date as a cursor, the where clause finds all messages **before** this date, because there is the `lt` Sequelize operator.

There are two more improvements for making it more robust:

```
...

export default {
  Query: {
    messages: async (parent, { cursor, limit = 100 }, { models }) => {
      return await models.Message.findAll({
        order: [['createdAt', 'DESC']],
        limit,
        where: cursor
          ? {
              createdAt: {
                [Sequelize.Op.lt]: cursor,
              },
            }
          : null,
      });
    },
    message: async (parent, { id }, { models }) => {
      return await models.Message.findById(id);
    },
  },
  Mutation: {
    ...
  },
  ...
};
```

First, the list should be ordered by `createdAt` date, because otherwise the cursor wouldn't help you any way. However, when the list is ordered, you can be sure that requesting the first page of messages without a cursor will lead into the most recent messages. When requesting the next page with the cursor of the last message's creation date from the previous page, you should get the next page of messages ordered by creation date. That's how you can move page by page through the list of messages.

Second, the ternary operator for the cursor makes sure that the cursor isn't needed for the first page request. As mentioned, the first page only retrieves the most recent messages in the list. Then you can use the creation date of the last message as cursor for the next page of messages.

Moreover, you could extract the where clause from the database query.

```

...

export default {
  Query: {
    messages: async (parent, { cursor, limit = 100 }, { models }) => {
      const cursorOptions = cursor
        ? {
            where: {
              createdAt: {
                [Sequelize.Op.lt]: cursor,
              },
            },
          }
        : {};

      return await models.Message.findAll({
        order: [['createdAt', 'DESC']],
        limit,
        ...cursorOptions,
      });
    },
    message: async (parent, { id }, { models }) => {
      return await models.Message.findById(id);
    },
  },
  Mutation: {
    ...
  },
  ...
};

```

Now, you should give it a shot yourself in GraphQL Playground. For instance, make the first request to request the most recent messages:

```

query {
  messages(limit: 2) {
    text
    createdAt
  }
}

```

Which may lead to something like this (be careful, dates should be different from your dates):

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/rwieruch/status/1041111111111111111>)

```
{
  "data": {
    "messages": [
      {
        "text": "Published a complete ...",
        "createdAt": "Wed Aug 22 2018 11:43:44 GMT+0200 (CEST)"
      },
      {
        "text": "Happy to release ...",
        "createdAt": "Wed Aug 22 2018 11:43:43 GMT+0200 (CEST)"
      }
    ]
  }
}
```




Now you can use the `createdAt` date from the last page to request the next page of messages with a `cursor`:

```
query {
  messages(limit: 2, cursor: "Wed Aug 22 2018 11:43:43 GMT+0200 (CEST)") {
    text
    createdAt
  }
}
```

It should give you the last message from the seed data, but not more even though the limit is set to 2, because there are only 3 messages in the database and you already have retrieved 2 of them in the previous page:

```
{
  "data": {
    "messages": [
      {
        "text": "Published the Road to learn React",
        "createdAt": "Wed Aug 22 2018 11:43:42 GMT+0200 (CEST)"
      }
    ]
  }
}
```

That's the basic implementation of a cursor-based pagination when using the creation date of an item as a stable identifier. Using the creation date is a common approach for this, but there may be alternatives to explore for you as well.

Cursor-based Pagination: Page Info, Connections and Hashes  (https://www.facebook.com/share/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)  (https://twitter.com/rwieruch/status/1031111111111111111)  (https://www.linkedin.com/share?text=GraphQL%20Server%20Tutorial%20by%20Robin%20Wieruch%20ReactJs%20tutorial%20with%20Apollo%20Server%20and%20Express%20-%20Robin%20Wieruch%20-%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

In this last section about pagination in GraphQL, you will advance the cursor-based pagination with a few improvements. As for how, you always have to query all creation dates of the messages in order to use the creation date of the last message for the next page as cursor. So what if you could have the

creation date of the last message as meta information? That's where GraphQL connections come into play which add only a structural change to your list fields in GraphQL. Let's use such a GraphQL connection in the *src/schema/message.js* file:

```
import { gql } from 'apollo-server-express';

export default gql`
  extend type Query {
    messages(cursor: String, limit: Int): MessageConnection!
    message(id: ID!): Message!
  }

  extend type Mutation {
    createMessage(text: String!): Message!
    deleteMessage(id: ID!): Boolean!
  }

  type MessageConnection {
    edges: [Message!]!
    pageInfo: PageInfo!
  }

  type PageInfo {
    endCursor: String!
  }

  type Message {
    id: ID!
    text: String!
    createdAt: String!
    user: User!
  }
`;
```

Basically you introduce an intermediate layer which holds meta information, which is done with the *PageInfo* type here, and has the list of items in an *edges* field. In the intermediate layer, you can introduce the new information such as an *endCursor* (*createdAt* of the last message in the list). Then you don't need to query every *createdAt* date of every message anymore but only the *endCursor* . Let's see how this looks like in the *src/resolvers/message.js* file:

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

```

...

export default {
  Query: {
    messages: async (parent, { cursor, limit = 100 }, { models }) => {
      const cursorOptions = cursor
        ? {
            where: {
              createdAt: {
                [Sequelize.Op.lt]: cursor,
              },
            },
          }
        : {};

      const messages = await models.Message.findAll({
        order: [['createdAt', 'DESC']],
        limit,
        ...cursorOptions,
      });

      return {
        edges: messages,
        pageInfo: {
          endCursor: messages[messages.length - 1].createdAt,
        },
      };
    },
    message: async (parent, { id }, { models }) => {
      return await models.Message.findById(id);
    },
  },
  Mutation: {
    ...
  },
  ...
};


```


You only give your returned result a new structure with the intermediate `edges` and `pageInfo` fields. The `pageInfo` then has the cursor of the last message in the list. Now you should be able to query the first page the following way:

```

query {
  messages(limit: 2) {
    edges {
      text
    }
    pageInfo {
      endCursor
    }
  }
}

```

 (<https://www.facebook.com/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

 (<https://twitter.com/sharer?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

Whereas the result may look like the following:

```

{
  "data": {
    "messages": {
      "edges": [
        {
          "text": "Published a complete ..."
        },
        {
          "text": "Happy to release ..."
        }
      ],
      "pageInfo": {
        "endCursor": "Wed Aug 22 2018 12:27:24 GMT+0200 (CEST)"
      }
    }
  }
}

```

And you can use the last cursor to query the next page:

```

query {
  messages(limit: 2, cursor: "Wed Aug 22 2018 12:27:24 GMT+0200 (CEST)") {
    edges {
      text
    }
    pageInfo {
      endCursor
    }
  }
}

```

Which will again only return you the remaining last message in the list. Now you are not required anymore to query the creation date of every message but instead only query the one necessary cursor the last message. In the end, the client application doesn't have to know the implementation details of having to use the cursor of the last message. It only needs to use the `endCursor` now.

Since you already have introduced the intermediate GraphQL connection layer, you can add another beneficial information there. Sometimes a GraphQL client needs to know whether there are more pages of a list to query, because every list is finite. So let's add this information to the schema for the message's connection in the `src/schema/message.js` file:

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)


```
import { gql } from 'apollo-server-express';

export default gql`
  extend type Query {
    messages(cursor: String, limit: Int): MessageConnection!
    message(id: ID!): Message!
  }

  extend type Mutation {
    createMessage(text: String!): Message!
    deleteMessage(id: ID!): Boolean!
  }

  type MessageConnection {
    edges: [Message!]!
    pageInfo: PageInfo!
  }

  type PageInfo {
    hasNextPage: Boolean!
    endCursor: String!
  }

  ...
`;
```

In the resolver in the `src/resolvers/message.js` file, you can find out about this information with the following implementation:

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

```

...

export default {
  Query: {
    messages: async (parent, { cursor, limit = 100 }, { models }) => {
      ...


      const messages = await models.Message.findAll({
        order: [['createdAt', 'DESC']],
        limit: limit + 1,
        ...cursorOptions,
      });

      const hasNextPage = messages.length > limit;
      const edges = hasNextPage ? messages.slice(0, -1) : messages;

      return {
        edges,
        pageInfo: {
          hasNextPage,
          endCursor: edges[edges.length - 1].createdAt,
        },
      };
    },
    message: async (parent, { id }, { models }) => {
      return await models.Message.findById(id);
    },
  },
  Mutation: {
    ...
  },
  ...
};

```

You only retrieve one more message than defined in the limit. If the list of messages is longer than the limit, then there is a next page. Otherwise there is no next page. Furthermore, you return only the limited messages or all messages in case there is no next page anymore. Now you should be able to include the `hasNextPage` field in the `pageInfo` field. If you query messages with a limit of 2 and no cursor, you should get `true` for the `hasNextPage` field (in case you are using the seed data). Otherwise, if query messages with a limit of more than 2 and no cursor, you should get `false` for the `hasNextPage` field. Then your GraphQL client application knows that the list has reached its end.

The last improvements have given your GraphQL client application a more straight forward GraphQL API. The client doesn't need to know about the cursor being the last creation date of a message in a list. It only uses the `endCursor` as `cursor` argument for the next page. However, the cursor is still a `freed (https://www.facebook.com/sharer` confusion on the GraphQL client side. The client should  `care about the format or the actual value of the cursor. So let's mask the cursor with a hash function` (https://www.facebook.com/sharer/share?text=GraphQL%20Server%20Tutorial%20www.robinwieruch.de%20graphql-apollo-server-tutorial%20by%40rwieruch%23ReactJs%20www.robinwieruch.de%20

```

...

const toCursorHash = string => Buffer.from(string).toString('base64');

const fromCursorHash = string =>
  Buffer.from(string, 'base64').toString('ascii');

export default {
  Query: {
    messages: async (parent, { cursor, limit = 100 }, { models }) => {
      const cursorOptions = cursor
        ? {
            where: {
              createdAt: {
                [Sequelize.Op.lt]: fromCursorHash(cursor),
              },
            },
          }
        : {};

      ...

      return {
        edges,
        pageInfo: {
          hasNextPage,
          endCursor: toCursorHash(
            edges[edges.length - 1].createdAt.toString(),
          ),
        },
      };
    },
    message: async (parent, { id }, { models }) => {
      return await models.Message.findById(id);
    },
  },
  Mutation: {
    ...
  },
  ...
};

```

The returned cursor as meta information is hashed by the new utility function. Don't forget to stringify the date before hashing it. Then the GraphQL client (try it yourself with your GraphQL Playground) receives a hashed `endCursor` field. The hashed value can be used as cursor to query the next page. In the resolver then, the incoming cursor is reverse hashed to convert it to the actual date which is used for the database query.

f (<https://www.facebook.com/sharer>

The hashing of the cursor is a common approach for cursor-based pagination, because it hides the implementation details from the client. The (GraphQL) client application only needs to use the hash value as cursor to query the next paginated page.

<https://www.robinwieruch.de/graphql-server-tutorial/>

✖ (<https://www.robinwieruch.de/graphql-server-tutorial/>

<https://www.robinwieruch.de/graphql-server-tutorial/>

by [@rwieruch](https://www.robinwieruch.de/graphql-server-tutorial/) ReactJs'

Exercises - server-tutorial%2f)

%2fwww.robinwieruch.de%2f

- read more about GraphQL pagination (<https://graphql.github.io/learn/pagination/>)

GraphQL Subscriptions

So far, you have used GraphQL to read and write data with queries and mutations. These are the two essential GraphQL operations to get a GraphQL server up and running for CRUD operations. Next you will learn about GraphQL Subscriptions for real-time communication between GraphQL client and server.

In the following you are going to implement a real-time communication for created messages. If one user creates a message, another user should get this message in a GraphQL client application as real-time update. To start, we need to add the Subscription root level type to the *src/schema* */message.js* schema:

```
import { gql } from 'apollo-server-express';

export default gql`
  extend type Query {
    ...
  }

  extend type Mutation {
    ...
  }

  ...

  type Message {
    id: ID!
    text: String!
    createdAt: String!
    user: User!
  }

  extend type Subscription {
    messageCreated: MessageCreated!
  }

  type MessageCreated {
    message: Message!
  }
`;
```

As a native GraphQL consumer, a subscription works similar to a GraphQL query. The only difference is that a subscription is a long-lived operation (vs. a query is a short-lived operation). A subscription from a GraphQL client could look like the following for the previously

<https://www.facebook.com/sharer/sharer.php?u=https%3A%2F%2Fwww.robinwieruch.de%2Fgraphql-apollo-server-tutorial%2F> <https://twitter.com/sharer?text=GraphQL%20Server%20Tutorial%20by%20RobinWieruch%23ReactJs%2Fwww.robinwieruch.de%2F>

implemented schema:

```
subscription {
  messageCreated {
    message {
      id
      text
      createdAt
      user {
        id
        username
      }
    }
  }
}
```

Now the implementation of this particular subscription. In the first part, you will setup the subscription architecture for your application. Afterward, you will add the implementation details for the created message subscription. While you have to do the former only once, the latter will be a repetitive doing when adding more GraphQL subscriptions to your application.

Apollo Server Subscription Setup

Because we are using Express as middleware, you need to expose the subscriptions with an advanced HTTP server setup in the `src/index.js` file:

```
import http from 'http';

...

server.applyMiddleware({ app, path: '/graphql' });


const httpServer = http.createServer(app);
server.installSubscriptionHandlers(httpServer);

const eraseDatabaseOnSync = true;

sequelize.sync({ force: eraseDatabaseOnSync }).then(async () => {
  if (eraseDatabaseOnSync) {
    createUsersWithMessages(new Date());
  }

  httpServer.listen({ port: 8000 }, () => {
    console.log('Apollo Server on http://localhost:8000/graphql (http://localhost:8000/graphql)');
  });
});
```

 (<https://www.facebook.com/sharer>

 (<https://twitter.com/sharer>

[/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f) /share?text=GraphQL%20Server%20Tutorial

%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f by %40rwieruch %23ReactJs'

[apollo-server-tutorial%2f](https://www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

%2fwww.robinwieruch.de%2f

Regarding the context which is passed to the resolvers, you can distinguish, in the same file, between

HTTP requests (GraphQL mutations and queries) and subscriptions. Whereas the HTTP requests come with a `req` (and `res`) object, the subscription comes with a `connection` object. Thus, you can pass at least the `models` as data access layer for the subscription's context. You will not need more for now.

```
...

const server = new ApolloServer({
  typeDefs: schema,
  resolvers,
  ...
  context: async ({ req, connection }) => {
    if (connection) {
      return {
        models,
      };
    }

    if (req) {
      const me = await getMe(req);

      return {
        models,
        me,
        secret: process.env.SECRET,
      };
    }
  },
});

...
```

Last but not least, for the sake of completing the overarching subscription setup, you have to use one of the available PubSub engines (<https://www.apollographql.com/docs/apollo-server/v2/features/subscriptions.html#PubSub-Implementations>) for publishing and subscribing to events. Apollo Server comes with its own default, but you can check the referenced link for other options. In a new `src/subscription/index.js` file, add the following implementation:

```
import { PubSub } from 'apollo-server';

export default new PubSub();
```

This `PubSub` instance is your API which enables subscriptions in your application. The overarching setup for subscriptions is done now.

Subscribing and Publishing with PubSub

Twitter (http

Let's implement the specific subscription for the message creation. It should be possible for another GraphQL client to listen to message creations. For instance, in a chat application it should be possible to see a message of someone else in real-time. Therefore, extend the previous `src/subscription`

/index.js file with the following implementation:

```
import { PubSub } from 'apollo-server';

import * as MESSAGE_EVENTS from './message';

export const EVENTS = {
  MESSAGE: MESSAGE_EVENTS,
};

export default new PubSub();
```

And add your first event in a new *src/subscription/message.js* file which is already used in the previous file:

```
export const CREATED = 'CREATED';
```

This folder structure already enables you to separate your events on a domain level. By exporting all events with their domains, you can simply import all events somewhere else and make use of the domain specific events there.

The only piece missing is using this event and the PubSub instance in your message resolver. In the beginning of this section, you have already added the new subscription to the message schema. Now you have to implement the counterpart in the *src/resolvers/message.js* file again:

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

```

...

import pubsub, { EVENTS } from '../subscription';

...

export default {
  Query: {
    ...
  },

  Mutation: {
    ...
  },

  Message: {
    ...
  },

  Subscription: {
    messageCreated: {
      subscribe: () => pubsub.asyncIterator(EVENTS.MESSAGE.CREATED),
    },
  },
};

```

The subscribe's function signature has access to the same arguments as the other resolver functions. So if you would need to access the models from the context, you could do it here. But it isn't necessary for this particular implementation.

Basically that's the subscription as resolver which fulfils the requirement of being the counterpart of the subscription in the message schema. However, since it uses a publisher-subscriber mechanism (PubSub) for events, you have only implemented the subscribing part but not the publishing part. It is possible for a GraphQL client to listen to changes, but there are no changes published yet. The best place for publishing a newly created message is in the same file when actually creating a message:

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)


```

...

import pubsub, { EVENTS } from '../subscription';

...

export default {
  Query: {
    ...
  },

  Mutation: {
    createMessage: combineResolvers(
      isAuthenticated,
      async (parent, { text }, { models, me }) => {
        const message = await models.Message.create({
          text,
          userId: me.id,
        });

        pubsub.publish(EVENTS.MESSAGE.CREATED, {
          messageCreated: { message },
        });

        return message;
      },
    ),
    ...
  },

  Message: {
    ...
  },

  Subscription: {
    messageCreated: {
      subscribe: () => pubsub.asyncIterator(EVENTS.MESSAGE.CREATED),
    },
  },
};

```

That's it. You have implemented your first subscription in GraphQL with Apollo Server and PubSub. In order to test it, you need to create a new message. Therefore you need to login a user too, because otherwise you are not authorized to create a message. My recommendation would be to step through the following GraphQL operations in two tabs in GraphQL Playground. In the first tab, execute the subscription:

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/rwieruch/status/1041111111111111111>)

```
subscription {
  messageCreated {
    message {
      id
      text
      createdAt
      user {
        id
        username
      }
    }
  }
}
```

You should see some indicator that the tab is listening to changes now. In the second tab, login a user:

```
mutation {
  signIn(login: "rwieruch", password: "rwieruch") {
    token
  }
}
```

Grab the token from the result and paste it to the HTTP headers panel in the same tab (your token should differ from mine):


```
{
  "x-token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwiZW1haWwiOiJoZWxsbyB2Jpb20iLCJ1c2VybmFtZSI6InJ3YWVydWNoIiwicm9sZSI6IkpETU10IiwiaWF0IjoxNTM0OTQ3NTYyLCJleHAiOiJlMzQ5NDkzNjJ9.mg4M6SfYPJkGf_Z2Zr7ztGNbDRDLksRWdhhDvTbmWbQ"
}
```

Then create a message in the second tab:

```
mutation {
  createMessage(text: "Does my subscription work?") {
    text
  }
}
```

Afterward, check your first tab again. It should show the created message:

 (https://www.facebook.com/sharer
/sharer.php?u=https%3a%2f
%2fwww.robinwieruch.de%2fgraphql-
apollo-server-tutorial%2f)

 (http
/share?text=GraphQL%20Server%20Tutorial
by %40rwieruch %23ReactJs%
%2fwww.robinwieruch.de%2f)

```

{
  "data": {
    "messageCreated": {
      "message": {
        "id": "4",
        "text": "Does my subscription work?",
        "createdAt": "Wed Aug 22 2018 16:22:41 GMT+0200 (CEST)",
        "user": {
          "id": "1",
          "username": "rwieruch"
        }
      }
    }
  }
}

```

Congratulations. You have implemented GraphQL subscriptions! It's not so easy to wrap your head around them, but once you get used to them and you know what you want to achieve, you can use them to create real-time GraphQL applications.

Exercises:

- read more about Subscriptions with Apollo Server (<https://www.apollographql.com/docs/apollo-server/v2/features/subscriptions.html>)
- watch a talk about GraphQL Subscriptions (<http://youtu.be/bn8qsi8jVew>)

Testing a GraphQL Server

Often one aspect when learning something new in the world of programming is missing: testing. In this section, I want to show you how to end-to-end (E2E) test your GraphQL server. While unit and integration tests are the fundamental pillars of the popular testing pyramid and will cover all standalone functionalities of your application, E2E tests cover user scenarios for your entire application. For instance, an E2E test will cover whether a user is able to sign up for your application or whether a user can delete another user when not being an admin. You don't need to write as many E2E tests as unit or integration tests, because they cover larger yet more complex user scenarios and not only small functionalities. In addition, E2E tests cover every technical corner of your application (e.g. GraphQL API, business logic, database).

GraphQL Server E2E Test Setup

Everything you need for testing your application is Mocha and Chai. Whereas Mocha is a test runner which organizes and executes your tests from a command line, Chai gives you all the functionalities to make assertions (e.g. "Expect X to be equal to Y").

```
npm install mocha chai --save-dev
```

Furthermore, you have to install a library, in this case `axios` (<https://github.com/axios/axios>), for making requests to your GraphQL API. For instance, when testing the sign up of a user, you can send a GraphQL mutation to your GraphQL API which should create a user in the database and return the created user.

```
npm install axios --save-dev
```

In order to get Mocha up and running as your test runner, you have to use it from your npm scripts in your `package.json` file. The pattern used here matches all test files with the suffix `.spec.js` within the `src/` folder.

```
{
  ...
  "scripts": {
    "start": "nodemon --exec babel-node src/index.js",
    "test": "mocha --require babel-core/register 'src/**/*.spec.js'"
  },
  ...
}
```

That should be sufficient to run your first test. Add a `src/tests/user.spec.js` to your application and write your first test over there:

```
import { expect } from 'chai';

describe('users', () => {
  it('user is user', () => {
    expect('user').to.eql('user');
  });
});
```

The test should be executable by running `npm test` on the command line. Even though the test itself doesn't test any logic of your application, it verifies that Mocha, Chai and your new npm script for testing should be working.

Before you can start to write your end-to-end tests for the GraphQL server, there needs to be another topic addressed: the database. Since the tests will run against the actual GraphQL server, you wouldn't want to test against your production database, but some test database instead. So let's add a `src/tests/user.spec.js` to your application and write your first test over there:

apollo-server-tutorial%2f) %2fwww.robinwieruch.de%2f

Now you are ready to write your tests against an actual running test sever (`npm run test-server`) which uses a consistently seeded test database. Last but not least, if you want to use `async/await` in your test environment too, you have to adjust your `.babelrc` file to:

```
{
  "presets": [
    [
      "env", {
        "targets": {
          "node": "current"
        }
      ]
    ], "stage-2"
  ]
}
```

Now you should be ready to go to write tests with asynchronous business logic with `async/await` as well.

Testing User Scenarios with E2E Tests

For every E2E test, you will make an actual request with `axios` to the API of the running GraphQL test server. For instance, testing your `user` GraphQL query would look like the following in the `src/tests/user.spec.js` file:

```
import { expect } from 'chai';

describe('users', () => {
  describe('user(id: String!): User', () => {
    it('returns a user when user can be found', async () => {
      const expectedResult = {
        data: {
          user: {
            id: '1',
            username: 'rwieruch',
            email: 'hello@robin.com',
            role: 'ADMIN',
          },
        },
      };

      const result = await userApi.user({ id: '1' });

      expect(result.data).to.eql(expectedResult);
    });
  });
});
```

f (<https://www.facebook.com/sharer>

t (<https://twitter.com/rwieruch>

[/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

[/share?text=GraphQL%20Server%20Tutorial](https://twitter.com/rwieruch/status/1021111111111111111)

<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>

<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>

written from or to the database. In between sits all the business logic such as authentication, authorization and pagination. As you can see, one such request goes through the whole GraphQL server stack from API to database. It's an end-to-end test and doesn't test an isolated unit (unit test) or a smaller composition of units (integration test).

The only piece missing for making the test itself so straight forward as it is right now is the `userApi` function. It's not implemented in the test, but in another `src/tests/api.js` file for being reusable for other tests and for keeping the tests more lightweight. In this file, you will find all your functions which can be used to run requests against your GraphQL test server.

```
import axios from 'axios';

const API_URL = 'http://localhost:8000/graphql (http://localhost:8000/graphql)';

export const user = async variables =>
  axios.post(API_URL, {
    query: `
      query ($id: ID!) {
        user(id: $id) {
          id
          username
          email
          role
        }
      }
    `,
    variables,
  });
```

After all, you can use basic HTTP for performing GraphQL operations across the network layer. It only needs a payload which is the query/mutation and the variables. Apart from that, the URL of the GraphQL server has to be known. That's everything needed to make a request to your GraphQL server. Now you only need to import the user API in your actual test file:

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<http://twitter.com/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

```
import { expect } from 'chai';

import * as userApi from './api';

describe('users', () => {
  describe('user(id: String!): User', () => {
    it('returns a user when user can be found', async () => {
      const expectedResult = {
        ...
      };

      const result = await userApi.user({ id: '1' });

      expect(result.data).to.eql(expectedResult);
    });
  });
});
```

In order to execute your test(s) now, you have to run your GraphQL test server in one command line tab with `npm run test-server` and execute your tests in another command line tab with `npm test`. The output should be similar to the following.


```
users
  user(id: ID!): User
    ✓ returns a user when user can be found (69ms)

1 passing (123ms)
```

If your output is erroneous, add console logs to your tests and application to figure out what went wrong. Another option taking the query from the axios request and putting it into GraphQL playground. Maybe there you will get an error which helps you to get things right.

That's it for your first E2E test against a GraphQL server. Let's add another one which uses the same API. Only then you see how useful it is to extract the API layer as reusable functions. In your `src/tests/user.spec.js` file add another test:

 (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

 (<https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)


```

import { expect } from 'chai';

import * as userApi from './api';

describe('users', () => {
  describe('user(id: ID!): User', () => {
    it('returns a user when user can be found', async () => {
      const expectedResult = {
        ...
      };

      const result = await userApi.user({ id: '1' });

      expect(result.data).to.eql(expectedResult);
    });

    it('returns null when user cannot be found', async () => {
      const expectedResult = {
        data: {
          user: null,
        },
      };

      const result = await userApi.user({ id: '42' });

      expect(result.data).to.eql(expectedResult);
    });
  });
});

```

It is valuable to not only test the happy path, but also other edge cases. In this case, the not so happy path didn't return an error but simply null for the user.

Let's add another test which checks that a non admin users isn't able to delete a user due to authorization. Here you will implement a whole user scenario from login to user deletion. Therefore, implement the sign in and delete user API in the *src/tests/api.js* file:

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

```

...

export const signIn = async variables =>
  await axios.post(API_URL, {
    query: `
      mutation ($login: String!, $password: String!) {
        signIn(login: $login, password: $password) {
          token
        }
      }
    `,
    variables,
  });

export const deleteUser = async (variables, token) =>
  axios.post(
    API_URL,
    {
      query: `
        mutation ($id: ID!) {
          deleteUser(id: $id)
        }
      `,
      variables,
    },
    {
      headers: {
        'x-token': token,
      },
    },
  );

```

The `deleteUser` mutation needs the token from the `signIn` mutation's result. Next, you can test the whole scenario by executing both APIs in your orchestrated way in a new E2E test:

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t ([https://twitter.com/share?text=GraphQL%20Server%20Tutorial%20by%40rwieruch%20ReactJs%2fwww.robinwieruch.de%2f](https://twitter.com/share?text=GraphQL%20Server%20Tutorial%20by%20%40rwieruch%20ReactJs%2fwww.robinwieruch.de%2f))

```

import { expect } from 'chai';

import * as userApi from './api';

describe('users', () => {
  describe('user(id: ID!): User', () => {
    ...
  });

  describe('deleteUser(id: String!): Boolean!', () => {
    it('returns an error because only admins can delete a user', async () => {
      const {
        data: {
          data: {
            signIn: { token },
          },
        },
      } = await userApi.signIn({
        login: 'ddavids',
        password: 'ddavids',
      });

      const {
        data: { errors },
      } = await userApi.deleteUser({ id: '1' }, token);

      expect(errors[0].message).to.eql('Not authorized as admin.');
```

First, you are using the `signIn` mutation to login a user to the application. The login is fulfilled once a token is returned. The token can then be used for every other GraphQL operation. In this case, it is used for the `deleteUser` mutation. However, in the end the mutation should fail, because the logged in user is not an admin user. You can try the same scenario on your own with an admin user instead for testing the happy path by reusing the APIs.

```

users
  user(id: String!): User
    ✓ returns a user when user can be found (81ms)
    ✓ returns null when user cannot be found
  deleteUser(id: String!): Boolean!
    ✓ returns an error because only admins can delete a user (109ms)

3 passing (276ms)
```

All the previous E2E tests cover scenarios for your user domain; going through the GraphQL API over

business logic to the database access. However, there is plenty of room for additions. What about testing the other user domain specific scenarios such as a user sign up (registration), providing a wrong password on sign in (login), or requesting one and another page of paginated messages for the message domain?

Furthermore, this section only covered E2E tests. By having Chai and Mocha at your disposal, you can also add smaller unit and integration tests for your different application layers (e.g. resolvers). If you need a library to spy, stub or mock something, I recommend Sinon (<https://sinonjs.org>) as complementary testing library for such cases. However it goes from here, there are no excuses anymore to not test your GraphQL server.

Exercises:



- implement tests for the message domain similar to the user domain
- write more fine-granular unit/integration tests for both domains
- read more about GraphQL and HTTP (<https://graphql.github.io/learn/serving-over-http/>)
- read more about Mocking with Apollo Server (<https://www.apollographql.com/docs/apollo-server/v2/features/mocking.html>)

Batching and Caching in GraphQL with Data Loader

The section is all about improving the requests to your database. Even though only one request (e.g. a GraphQL query) hits your GraphQL API, you might end up with multiple database reads (and writes) to resolve all your fields in your resolvers. Let's see this problem in action by performing the following GraphQL query in GraphQL Playground:

```
query {
  messages {
    edges {
      user {
        username
      }
    }
  }
}
```

Keep the query open, because you will use this query in this section as case study for the following improvements. Your query result should be similar to the following:

 (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)  (<https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

```
{
  "data": {
    "messages": {
      "edges": [
        {
          "user": {
            "username": "ddavids"
          }
        },
        {
          "user": {
            "username": "ddavids"
          }
        },
        {
          "user": {
            "username": "rwieruch"
          }
        }
      ]
    }
  }
}
```

And on your command line for your running GraphQL server, you can see that 4 requests were made to the database:

```
Executing (default): SELECT "id", "text", "createdAt", "updatedAt", "userId" FROM "messages" AS "message" ORDER BY "message"."createdAt" DESC LIMIT 101;
```

```
Executing (default): SELECT "id", "username", "email", "password", "role", "createdAt", "updatedAt" FROM "users" AS "user" WHERE "user"."id" = 2;
```

```
Executing (default): SELECT "id", "username", "email", "password", "role", "createdAt", "updatedAt" FROM "users" AS "user" WHERE "user"."id" = 2;
```

```
Executing (default): SELECT "id", "username", "email", "password", "role", "createdAt", "updatedAt" FROM "users" AS "user" WHERE "user"."id" = 1;
```

There is one request made for the list of messages and three requests for each individual user of a message. That's the nature of GraphQL, because even though you can nest your GraphQL relationships and thus your query structure, there are still database requests to be made eventually. Check your resolvers for the user of a message in your `src/resolvers/message.js` file to see where this is happening. At some point, you may run into performance bottlenecks when nesting a GraphQL query (or mutation) too deep, because a lot of things need to be retrieved from your database.

In the following, you are going to optimize these database accesses with batching. It's a strategy not only used for a GraphQL server and its database, but users in other programming environments too. Compare again the query result in GraphQL Playground and your database output on the command line. There are two improvements which can be made with batching by having two strategies in place.



First, one author of a message is retrieved twice from the database which is redundant. Even though there are multiple messages, the author of some of these messages can be the same person. Imagine this problem on a larger scale for 100 messages between two authors in a chat application. There would be one request for the 100 messages and 100 requests for the 100 authors of each message which would lead to overall 101 database accesses. What if duplicated authors would be retrieved only once? Then it would only need one request for the 100 messages and 2 requests for the authors which leads to only 3 database accesses. Since you know all the identifiers of the authors, all identifiers can be batched to a set (no identifier is repeated) of identifiers. So in your case of the two authors a list of [2, 2, 1] identifiers would become a set of [2, 1] identifiers.

Second, every author is read from the database individually even though the list is purged from its duplications. What if you could read all authors with only one database request? It should be possible, because at the time of the GraphQL API request happening and having all messages at your disposal, you know how all identifiers of the authors which you would have to read from the database effectively. This would decrease your database accesses from 3 to 2, because now you would only request the list of 100 messages and its 2 authors in two request.

The same two principals can be applied to your 4 database accesses which could be decreased to 2 database accesses. On a smaller scale it might haven't that much of a performance impact, but as you have seen for the 100 messages with the 2 authors, it would reduce your database accesses from 101 to 2. That's where Facebook's open source dataloader (<https://github.com/facebook/dataloader>) comes into play. You can install it via npm on the command line:

```
npm install dataloader
```

No in your `src/index.js` file you can import and make use of it:

 (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)  (<https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

```

import DataLoader from 'dataloader';

...

const batchUsers = async (keys, models) => {
  const users = await models.User.findAll({
    where: {
      id: {
        $in: keys,
      },
    },
  });

  return keys.map(key => users.find(user => user.id === key));
};

const server = new ApolloServer({
  typeDefs: schema,
  resolvers,
  ...
  context: async ({ req, connection }) => {
    if (connection) {
      ...
    }

    if (req) {
      const me = await getMe(req);

      return {
        models,
        me,
        secret: process.env.SECRET,
        loaders: {
          user: new DataLoader(keys => batchUsers(keys, models)),
        },
      };
    }
  },
});
...

```

Identical to the models, the loaders, which will act as abstraction on top of the models, can be passed as context to the resolvers. You will see in the next steps how the loader, in this case the user loader, is used instead of the models directly. But what about the function as argument for the DataLoader instantiation?

The function gives you access in its arguments to a list of keys. These keys are your set of identifiers (purged from any duplication, strategy 1) which can be used to retrieve something from a database. That's why keys (identifiers) and models (data access layer) are passed to the `batchUsers()` function. The function then takes the keys to retrieve the entities via the model from the database. At the end of the function, the keys are mapped to the retrieved entities (here users) in order as the retrieved entities (here users). Otherwise, it would have been possible to return the users in any order.

That's it for the setup of the loader which can be seen as improved abstraction on top of the model. Now, since you are passing the loader for the batched user retrieval as context to the resolvers, you can make use of it in the `src/resolvers/message.js` file:

Even though the `load()` function takes each identifier individually, it will batch all these identifiers to one set of identifiers (strategy 1) and request all users at once (strategy 2). Try it yourself by executing the same GraphQL query in GraphQL Playground again. The GraphQL query result should stay the same, but in your command line output for the GraphQL server you should only see two and not four requests being made to the database:

That's it for the batching improvement. Instead of fetching each (duplicated) user on its own, you fetch them all at once in one batched request with the dataloader package.

Now let's get into caching, which is a difficult topic in software engineering. Apart from batching, the dataloader package gives you the option to cache your request too. It doesn't work right now. Try to

execute the same GraphQL query twice and you should see the database accesses twice on your command line.

```
Executing (default): SELECT "id", "text", "createdAt", "updatedAt", "userId" FROM "messages" AS "message" ORDER BY "message"."createdAt" DESC LIMIT 101;
Executing (default): SELECT "id", "username", "email", "password", "role", "createdAt", "updatedAt" FROM "users" AS "user" WHERE "user"."id" IN (2, 1);

Executing (default): SELECT "id", "text", "createdAt", "updatedAt", "userId" FROM "messages" AS "message" ORDER BY "message"."createdAt" DESC LIMIT 101;
Executing (default): SELECT "id", "username", "email", "password", "role", "createdAt", "updatedAt" FROM "users" AS "user" WHERE "user"."id" IN (2, 1);
```

That's happening because for every request a new instance of a dataloader is created within the GraphQL context. If you would move the dataloader instantiation outside, you would get the caching benefit of dataloader for free:

```
...

const userLoader = new DataLoader(keys => batchUsers(keys, models));

const server = new ApolloServer({
  typeDefs: schema,
  resolvers,
  ...
  context: async ({ req, connection }) => {
    if (connection) {
      ...
    }


    if (req) {
      const me = await getMe(req);

      return {
        models,
        me,
        secret: process.env.SECRET,
        loaders: {
          user: userLoader,
        },
      };
    }
  },
});

...

```

 (https://www.facebook.com/sharer

 (http

Try to execute the same GraphQL query twice again. This time you should see only one time the database access (for the places where the loader is used) and the second time it should be cached.

%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

by %40rwieruch %23ReactJs%2fwww.robinwieruch.de%2f

```

Executing (default): SELECT "id", "text", "createdAt", "updatedAt", "userId" FROM "messages" AS
S "message" ORDER BY "message"."createdAt" DESC LIMIT 101;
Executing (default): SELECT "id", "username", "email", "password", "role", "createdAt", "update
dAt" FROM "users" AS "user" WHERE "user"."id" IN (2, 1);

Executing (default): SELECT "id", "text", "createdAt", "updatedAt", "userId" FROM "messages" AS
"message" ORDER BY "message"."createdAt" DESC LIMIT 101;

```

In this case, the users of the messages are not read from the database twice. Only the messages, because they are not using a dataloader yet. That's how you can achieve caching in GraphQL with dataloader. However, personally I would be careful with it, because choosing a caching strategy isn't simple. For instance, what if a cached user has been updated in between? Your GraphQL client application would still query the cached user.

It's difficult to find the right timing for invalidating the cache. That's why it's recommended to perform the dataloader instantiation with every incoming GraphQL request. Even though you are loosing the benefit of caching over multiple GraphQL requests then, you are still using the cache for every database access within the one incoming GraphQL request. The dataloader package says it the following way: *"DataLoader caching does not replace Redis, Memcache, or any other shared application-level cache. DataLoader is first and foremost a data loading mechanism, and its cache only serves the purpose of not repeatedly loading the same data in the context of a single request to your Application."* So if you want to get into real caching on database level, you should give `{{* a_blank "Redis" "https://redis.io/" (https://redis.io/) *}}` a shot.

In the end, you can outsource the loaders similar to your models into a different folder/file structure. In a new `src/loaders/user.js` file you can put the batching for the individual users:

```

export const batchUsers = async (keys, models) => {
  const users = await models.User.findAll({
    where: {
      id: {
        $in: keys,
      },
    },
  });

  return keys.map(key => users.find(user => user.id === key));
};

```

And in a new `src/loaders/index.js` file export all the functions:

f (<https://www.facebook.com/sharer>

t (<https://twitter.com/sharer>

`import * as user from './user';`

`/share?text=GraphQL%20Server%20Tutorial`

`%2fwww.robinwieruch.de%2fgraphql-`

`by %40rwieruch %23ReactJs'`

`apollo-server-tutorial%2f)`

`%2fwww.robinwieruch.de%2f`

Finally you can import it in your `src/index.js` file again and make use of it:

```
...
import DataLoader from 'dataloader';

...
import loaders from './loaders';

...

const server = new ApolloServer({
  typeDefs: schema,
  resolvers,
  ...
  context: async ({ req, connection }) => {
    if (connection) {
      ...
    }

    if (req) {
      const me = await getMe(req);

      return {
        models,
        me,
        secret: process.env.SECRET,
        user: new DataLoader(keys =>
          loaders.user.batchUsers(keys, models),
        ),
      };
    }
  },
});

...

```

Last but not least, don't forget to add the loader to your subscriptions too, in case you make use of them over there:

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/rwieruch/share?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%20ReactJs%20www.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

```

...

const server = new ApolloServer({
  typeDefs: schema,
  resolvers,
  ...
  context: async ({ req, connection }) => {
    if (connection) {
      return {
        models,
        user: new DataLoader(keys =>
          loaders.user.batchUsers(keys, models),
        ),
      };
    }

    if (req) {
      ...
    }
  },
});

...

```

That's it. Feel free to add more loaders, maybe also for the message domain, on your own. They give you a great abstraction on top of your models to enable batching and request-based caching.

Exercises:

- read more about GraphQL and DataLoader (<https://www.apollographql.com/docs/graphql-tools/connectors.html#dataloader>)
- read more about GraphQL Best Practices (<https://graphql.github.io/learn/best-practices/>)

GraphQL Server + PostgreSQL Deployment to Heroku

Eventually you want to deploy your GraphQL server somewhere that it can be reached online by others or that it can be used in production. In this section, you will learn how to deploy your GraphQL server to Heroku which is a platform as a service to host applications. The best thing about it: You can host your PostgreSQL there as well.

The following section will guide you through the process on the command line. If you want to take the visual route, you can checkout this GraphQL server on Heroku deployment tutorial

(<https://www.apollographql.com/docs/apollo-server/deployment/heroku.html>) which, however, doesn't include the PostgreSQL database deployment.

Initially you need to complete three requirements for using Heroku: by <https://www.robinwieruch.de/graphql-apollo-server-tutorial/>

- install git for your command line and push your project to GitHub (<https://www.robinwieruch.de/git-essential-commands/>)
- create an account for Heroku (<https://www.heroku.com/>)
- install the Heroku CLI (<https://devcenter.heroku.com/articles/heroku-cli>) for accessing Heroku's features on the command line

On the command line verify your Heroku installation with `heroku version`. If there is a valid installation, sign in to your recently created Heroku account with `heroku login` on the command line. That's it for the general Heroku setup. You should be able to use Heroku for hosting any of your applications now.

Now, in your project's folder, you can create a new Heroku application from the command line. It's up to you to give your application any name:

```
heroku create graphql-server-node-js
```

Afterward, you can also install the PostgreSQL add-on for Heroku on the command line for your project:

```
heroku addons:create heroku-postgresql:hobby-dev
```

It uses the hobby tier (<https://devcenter.heroku.com/articles/heroku-postgres-plans#hobby-tier>) which can be upgraded any time but shouldn't cost you anything for the start. The output for the PostgreSQL add-on installation should be similar to:

```
Creating heroku-postgresql:hobby-dev on ● graphql-server-node-js... free
Database has been created and is available
! This database is empty. If upgrading, you can transfer
! data from another database with pg:copy
Created postgresql-perpendicular-34121 as DATABASE_URL
Use heroku addons:docs heroku-postgresql to view documentation
```

As pointed out by the command line output, you can always check the Heroku PostgreSQL documentation (<https://devcenter.heroku.com/articles/heroku-postgresql>) for more in depth instructions for your database setup. Depending on the plan you have chosen, your database can take a couple of minutes to become available.

f (<https://www.facebook.com/sharer>)

t (<https://twitter.com/sharer>)

Now, everything should be set up from a command line perspective to take your application online. By having the PostgreSQL add-on installed, you should have gotten a database URL too. You can find it with `heroku config`. Now, let's step into your GraphQL server's code to make a couple of adjustments for production.

In your `src/models/index.js`, you need to decide between development (coding, testing) and production (live) build. Because you have a new environment variable for your database URL now, you can use this to make the decision:

```
import Sequelize from 'sequelize';

let sequelize;
if (process.env.DATABASE_URL) {
  sequelize = new Sequelize(process.env.DATABASE_URL, {
    dialect: 'postgres',
  });
} else {
  sequelize = new Sequelize(
    process.env.TEST_DATABASE || process.env.DATABASE,
    process.env.DATABASE_USER,
    process.env.DATABASE_PASSWORD,
    {
      dialect: 'postgres',
    },
  );
}

...
```

If you check your `.env` file, you will see that the `DATABASE_URL` environment variable isn't there. But you should see that it is set as Heroku environment variable with `heroku config:get DATABASE_URL`. Once your application is live on Heroku, your environment variables are merged with Heroku's environment variables. That's why the `DATABASE_URL` isn't applied for your local development environment.

Another environment variable which is used in your `src/index.js` file is the `SECRET` for your authentication strategy. If you haven't included your `.env` file in your project's version control (see `.gitignore`), you need to set the `SECRET` for your production code in Heroku on the command line too:

```
heroku config:set SECRET mysecret.
```

Another thing which needs consideration is the application's port which is specified in the `src/index.js` file. In case Heroku adds its own `PORT` environment variable, you should use the port from an environment variable as fallback.

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/rwieruch/status/1041111111111111111>)

```
...
const port = process.env.PORT || 8000;

sequelize.sync({ force: isTest }).then(async () => {
  if (isTest) {
    createUsersWithMessages(new Date());
  }

  httpServer.listen({ port }, () => {
    console.log(`Apollo Server on http://localhost (http://localhost:${port}/graphql`);
  });
});

...
```

Last but not least, you can decide whether you want to start with a seeded database, or an empty database on Heroku PostgreSQL. If it should be seeded, you can add an extra flag to the seeding:

```
...

const isTest = !!process.env.TEST_DATABASE;
const isProduction = !!process.env.DATABASE_URL;
const port = process.env.PORT || 8000;



sequelize.sync({ force: isTest || isProduction }).then(async () => {
  if (isTest || isProduction) {
    createUsersWithMessages(new Date());
  }

  httpServer.listen({ port }, () => {
    console.log(`Apollo Server on http://localhost (http://localhost:${port}/graphql`);
  });
});

...
```

Don't forget to remove the flag afterward, otherwise the database is purged and seeded with every deployment. That's it for the code adjustments. Depending on development or production, you are choosing the correct database, you seed (or seed not) your database and you choose an appropriate port for your GraphQL server. Now let's take it online to Heroku.

Before pushing your application to Heroku, you need to push all your recent changes with git on the command line to your GitHub repository (git add, git commit, git push). Afterward, you can push all the changes to your Heroku remote repository too, because you have created a Heroku application

If everything went successful, you can open the application with  (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-server-by-rwieruch%23ReactJs>) or  (<https://t.me/share?text=GraphQL%20Server%20Tutorial>). Don't forget to add the /graphql suffix to your URL in the browser to open up GraphQL Playground.

Depending on your seeding strategy, your database should be empty or should have seeded data. In %2fapollo-server-tutorial%2f, www.robinwieruch.de/

case of the former, you need first to register a user and create messages with this user via GraphQL mutations. Otherwise, if your database is seeded, you can start to request a list of messages with a GraphQL query.

Congratulations, your application should be live now. Not only your GraphQL server is running on Heroku, but also your PostgreSQL database. Follow the exercises to learn more about Heroku.

Exercises:

- create sample data in your production database with GraphQL Playground
- get familiar with the Heroku Dashboard (<https://dashboard.heroku.com/apps>)
 - find your application's logs
 - find your application's environment variables
- access your PostgreSQL database on Heroku with `heroku pg:psql`

. . .

Over the last sections, you have built a sophisticated GraphQL server boilerplate project with Express and Apollo Server. Even though GraphQL isn't opinionated about various things, you should have learned about topics such as authentication, authorization, database access, and pagination now. Most of the things were more straight forward because of using Apollo Server over the GraphQL reference implementation in JavaScript. That's okay, because many people are using Apollo Server nowadays for building their GraphQL servers. You can use this application as starter project to realize your own ideas now. Moreover, you can find the whole starter project with a GraphQL client built in React in this GitHub repository (<https://github.com/rwieruch/fullstack-apollo-react-express-boilerplate-project>). My recommendation would be to continue implementing more features for the project to make your own ideas happen or to implement a GraphQL client application with React (or anything else) for it.

Read more: A complete React with Apollo and GraphQL Tutorial (<https://www.robinwieruch.de/react-graphql-apollo-tutorial/>)

I would like to hear your thoughts :-)

Find me on  Twitter (<https://twitter.com/rwieruch>)  (<https://www.facebook.com/rwieruch>) and  GitHub (<https://github.com/rwieruch>)  (<mailto:rwieruch@gmail.com>)

Did the article help you? Share it with your friends on social media  , support me on  Patreon (<https://www.patreon.com/rwieruch>), become a React developer with my books  <https://www.robinwieruch.de/graphql-apollo-server-tutorial/>

The Road to learn React



Build a Hacker News App along the way. No setup configuration. No tooling. No Redux. Plain React in 190+ pages of learning material.

Learn React like **33.000+ readers**.

GET THE BOOK FOR FREE > ([HTTPS://WWW.GETREVUE.CO/PROFILE/RWIERUCH](https://www.getrevue.co/profile/rwieruch))

Or get it from Amazon (<https://amzn.to/2LHjxRB>)

f (<https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f>)

t (<https://twitter.com/sharer?text=GraphQL%20Server%20Tutorial%20by%20rwieruch%23ReactJs%2fwww.robinwieruch.de%2f>)

46 Comments

Robin Wieruch

1 Login ▾

❤ Recommend 2

🐦 Tweet

f Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS (?)



Name



Tymly • 2 months ago

Wow! This is an incredible piece of work. Whole articles devoted to cursor pagination aren't as clear as this. And to cover so many facets of GraphQL... brilliant. We got our first GraphQL API, with subscriptions firing from PostgreSQL updates, working in a day because of this blog. Like others have said, this is the missing manual which should have been in the GraphQL box. Thank you!

Tim

2 ^ | ▾ • Reply • Share ›



Robin Wieruch Mod ➔ Tymly • 2 months ago

Hi Tymly, thank you so much for this comment. It made my day! <3 Are you able to tweet a smaller piece of it on Twitter? I would love to retweet it, but most important I would love to have it as testimonial for the upcoming book: <https://roadtoreact.com/cou...>

^ | ▾ • Reply • Share ›



Tymly ➔ Robin Wieruch • 2 months ago

We already did! ;-)



Testimonial is no problem let us know if you need anything further! (tymly@wmfs.net)
Kind regards

Tim

^ | ▾ • Reply • Share ›

f (https://www.facebook.com/sharer

🐦 (http

/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f) /share?text=GraphQL%20Server%20Tutorial

%2fwww.robinwieruch.de%2fgraphql-

by %40rwieruch %23ReactJs'

apollo-server-tutorial%2f)

%2fwww.robinwieruch.de%2f



Maxamed Daahir Maxamed • 2 months ago

Never miss an article about web development, JavaScript and self-growth.

TAKE PART

- ✓ Join 18.000+ Developers
- ✓ Learn Web Development with JavaScript
- ✓ Tips and Tricks
- ✓ Access Tutorials, eBooks and Courses
- ✓ Personal Development as a Software Engineer

SUBSCRIBE

RECENT POSTS



([https://www.robinwieruch.de](https://www.robinwieruch.de/react-warning-cant-call-set-state/)

/react-

warning-

cant-call-

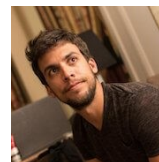
f ([https://www.facebook.com/sharer](https://www.facebook.com/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f)

on-an-
/sharer.php?u=https%3a%2f

%2fwww.robinwieruch.de%2fgraphql-
component/)

PREVENT REACT SETSTATE ON UNMOUNTED COMPONENT

ABOUT ME



Hi, I'm Robin - a German and Remote/Berlin based consulting software engineer. Find out more about me

([https://www.robinwieruch.de](https://www.robinwieruch.de/about)

/about), work with me

(<https://www.robinwieruch.de> **Twitter** (<https://www.robinwieruch.de>

/share?text=GraphQL%20Server%20Tutorial

my social media profiles below
by %40wieruch %23ReactJs'

%2fwww.robinwieruch.de%2

([HTTPS://WWW.ROBINWIERUCH.DE/REACT-WARNING-CANT-CALL-SETSTATE-ON-AN-UNMOUNTED-COMPONENT/](https://www.robinwieruch.de/react-warning-cant-call-setstate-on-an-unmounted-component/))



(<https://www.robinwieruch.de/react-state-array-add-update-remove/>)

HOW TO MANAGE REACT STATE WITH ARRAYS ([HTTPS://WWW.ROBINWIERUCH.DE/REACT-STATE-ARRAY-ADD-UPDATE-REMOVE/](https://www.robinwieruch.de/react-state-array-add-update-remove/))



(<https://www.robinwieruch.de/create-react-app-css-modules/>)

CREATE REACT APP WITH CSS MODULES ([HTTPS://WWW.ROBINWIERUCH.DE/CREATE-REACT-APP-CSS-MODULES/](https://www.robinwieruch.de/create-react-app-css-modules/))



© Robin Wieruch

 (<https://www.twitter.com/rwieruch>)

 (<https://www.github.com/rwieruch>)

 (<https://www.linkedin.com/in/robin-wieruch-971933a6>)

 (<https://www.producthunt.com/@rwieruch>)


 (<https://www.patreon.com/rwieruch>)  (<https://www.paypal.me/rwieruch>)

 (<https://www.facebook.com/rwieruch>)

 (<https://www.instagram.com/rwieruch/>)

[Legal Notice \(/legal\)](#)

 (<https://www.facebook.com/sharer>

 (<https://twitter.com/rwieruch>)

[/sharer.php?u=https%3a%2f%2fwww.robinwieruch.de%2fgraphql-apollo-server-tutorial%2f](https://www.robinwieruch.de/graphql-apollo-server-tutorial/) [/share?text=GraphQL%20Server%20Tutorial by %40rwieruch %23ReactJs%2fwww.robinwieruch.de%2f](https://twitter.com/rwieruch)