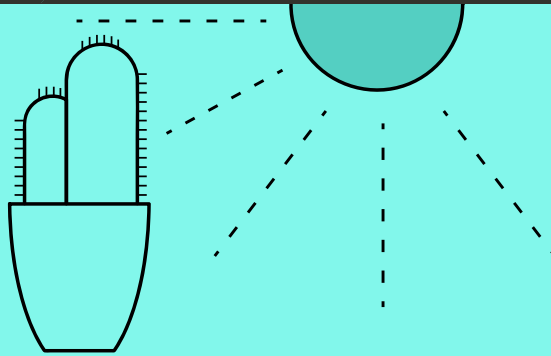


[About course](#)[Course contents](#)[FAQ](#)[Partners](#)[Challenge](#) [English](#) [Fullstack](#)[Part 8](#)[Database and user administration](#)

## c Database and user administration

a GraphQL-server  
b React and GraphQL  
c Database and user administration

-Mongoose and Apollo

-Validation

-User and log in

-Friends list

-Exercises 8.13.-8.16

We will now add user management to our application, but let's first start using a database for storing data.

### Mongoose and Apollo

Install Mongoose and dotenv:

```
npm install mongoose dotenv
```

[copy](#)

d Login and updating the cache  
e Fragments and subscriptions

We will imitate what we did in parts [3](#) and [4](#).

The person schema has been defined as follows:

```
const mongoose = require('mongoose')

const schema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
```

[copy](#)

```

    minlength: 5
  },
  phone: {
    type: String,
    minlength: 5
  },
  street: {
    type: String,
    required: true,
    minlength: 5
  },
  city: {
    type: String,
    required: true,
    minlength: 3
  },
})

module.exports = mongoose.model('Person', schema)

```

We also included a few validations. `required: true`, which makes sure that a value exists, is actually redundant: we already ensure that the fields exist with GraphQL. However, it is good to also keep validation in the database.

We can get the application to mostly work with the following changes:

```

// ...
const mongoose = require('mongoose')
mongoose.set('strictQuery', false)
const Person = require('./models/person')

require('dotenv').config ()

const MONGODB_URI = process.env.MONGODB_URI

console.log('connecting to', MONGODB_URI)

mongoose.connect(MONGODB_URI)
  .then(() => {
    console.log('connected to MongoDB')
  })
  .catch((error) => {
    console.log('error connection to MongoDB:',
error.message)
  })

const typeDefs = `
  ...
`

const resolvers = {
  Query: {
    personCount: async () =>
Person.collection.countDocuments(),
    allPersons: async (root, args) => {

```

copy

```

    // filters missing
    return Person.find ({}))
  },
  findPerson : async (root, args) => Person.findOne ({
name: args.name }),
  },
  Person: {
    address: (root) => {
      return {
        street: root.street,
        city: root.city,
      }
    },
  },
  Mutation: {
    addPerson: async (root, args) => {
      const person = new Person({ ...args })
      return person.save()
    },
    editNumber: async (root, args) => {
      const person = await Person.findOne ({ name:
args.name })
      person.phone = args.phone
      return person.save()
    },
  },
}
}

```

The changes are pretty straightforward. However, there are a few noteworthy things. As we remember, in Mongo, the identifying field of an object is called *\_id* and we previously had to parse the name of the field to *id* ourselves. Now GraphQL can do this automatically.

Another noteworthy thing is that the resolver functions now return a *promise*, when they previously returned normal objects. When a resolver returns a promise, Apollo server sends back the value which the promise resolves to.

For example, if the following resolver function is executed,

```

allPersons: async (root, args) => {
  return Person.find ({}))
},

```

copy

Apollo server waits for the promise to resolve, and returns the result. So Apollo works roughly like this:

```

allPersons: async (root, args) => {
  const result = await Person.find ({}))
  return result
}

```

copy

Let's complete the `allPersons` resolver so it takes the optional parameter `phone` into account:

```
Query: {  
  // ..  
  allPersons: async (root, args) => {  
    if (!args.phone) {  
      return Person.find ({})  
    }  
  
    return Person.find ({ phone: { $exists: args.phone ===  
'YES' } })  
  },  
},
```

copy

So if the query has not been given a parameter `phone`, all persons are returned. If the parameter has the value `YES`, the result of the query

```
Person.find ({ phone: { $exists: true } })
```

copy

is returned, so the objects in which the field `phone` has a value. If the parameter has the value `NO`, the query returns the objects in which the `phone` field has no value:

```
Person.find ({ phone: { $exists: false } })
```

copy

## Validation

As well as in GraphQL, the input is now validated using the validations defined in the mongoose schema. For handling possible validation errors in the schema, we must add an error-handling `try/catch` block to the `save` method. When we end up in the catch, we throw an exception GraphQLError with error code :

```
Mutation: {  
  addPerson: async (root, args) => {  
    const person = new Person({ ...args })  
  
    try {  
      await person.save()  
    } catch (error) {  
      throw new GraphQLError('Saving person failed', {  
        extensions: {  
          code: 'BAD_USER_INPUT',  
          invalidArgs: args.name,  
          error  
        }  
      })  
    }  
  }  
}
```

copy

```

    })
  }

  return person
},
editNumber: async (root, args) => {
  const person = await Person.findOne ({ name:
args.name })
  person.phone = args.phone

  try {
    await person.save()
  } catch (error) {
    throw new GraphQLError('Saving number failed', {
      extensions: {
        code: 'BAD_USER_INPUT',
        invalidArgs: args.name,
        error
      }
    })
  }
})
}

return person
}
}

```

We have also added the Mongoose error and the data that caused the error to the *extensions* object that is used to convey more info about the cause of the error to the caller. The frontend can then display this information to the user, who can try the operation again with a better input.

The code of the backend can be found on [Github](#), branch *part8-4*.

## User and log in

Let's add user management to our application. For simplicity's sake, let's assume that all users have the same password which is hardcoded to the system. It would be straightforward to save individual passwords for all users following the principles from [part 4](#), but because our focus is on GraphQL, we will leave out all that extra hassle this time.

The user schema is as follows:

```

const mongoose = require('mongoose')

const schema = new mongoose.Schema({
  username: {
    type: String,
    required: true,
    minlength: 3
  },
  friends: [
    {

```

copy

```

        type: mongoose.Schema.Types.ObjectId,
        ref: 'Person'
      }
    ],
  })

module.exports = mongoose.model('User', schema)

```

Every user is connected to a bunch of other persons in the system through the `friends` field. The idea is that when a user, e.g. *mluukkai*, adds a person, e.g. *Arto Hellas*, to the list, the person is added to their `friends` list. This way, logged-in users can have their own personalized view in the application.

Logging in and identifying the user are handled the same way we used in [part 4](#) when we used REST, by using tokens.

Let's extend the schema like so:

```

type User {
  username: String!
  friends: [Person!]!
  id: ID!
}

type Token {
  value: String!
}

type Query {
  // ..
  me: User
}

type Mutation {
  // ...
  createUser(
    username: String!
  ): User
  login(
    username: String!
    password: String!
  ): Token
}

```

The query `me` returns the currently logged-in user. New users are created with the `createUser` mutation, and logging in happens with the `login` mutation.

The resolvers of the mutations are as follows:

```

const jwt = require('jsonwebtoken')

```

```

Mutation: {
  // ..
  createUser: async (root, args) => {
    const user = new User({ username: args.username })

    return user.save()
      .catch(error => {
        throw new GraphQLError('Creating the user
failed', {
          extensions: {
            code: 'BAD_USER_INPUT',
            invalidArgs: args.username,
            error
          }
        })
      })
  },
  login: async (root, args) => {
    const user = await User.findOne ({ username:
args.username })

    if ( !user || args.password !== 'secret' ) {
      throw new GraphQLError('wrong credentials', {
        extensions: {
          code: 'BAD_USER_INPUT'
        }
      })
    }

    const userForToken = {
      username: user.username,
      id: user._id,
    }

    return { value: jwt.sign(userForToken,
process.env.JWT_SECRET) }
  },
}

```

The new user mutation is straightforward. The login mutation checks if the username/password pair is valid. And if it is indeed valid, it returns a jwt token familiar from [part 4](#). Note that the `JWT_SECRET` must be defined in the `.env` file.

User creation is done now as follows:

```

mutation {
  createUser (
    username: "mluukkai"
  ) {
    username
    id
  }
}

```

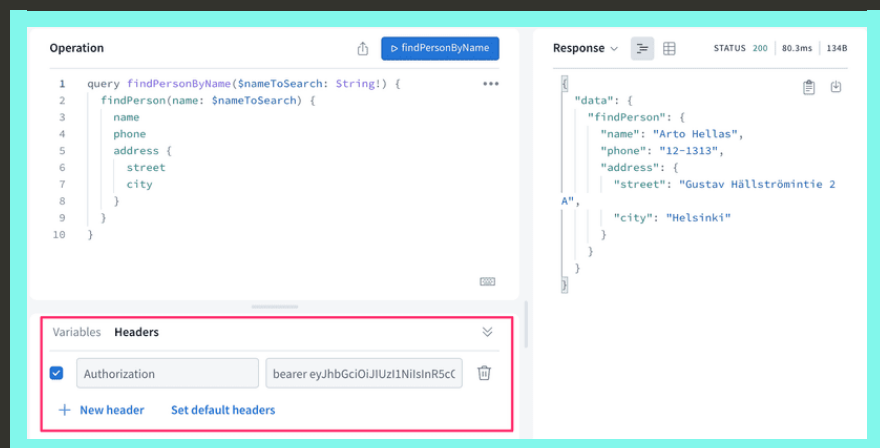
copy

The mutation for logging in looks like this:

```
mutation {
  login (
    username: "mluukkai"
    password: "secret"
  ) {
    value
  }
}
```

Just like in the previous case with REST, the idea now is that a logged-in user adds a token they receive upon login to all of their requests. And just like with REST, the token is added to GraphQL queries using the *Authorization* header.

In the Apollo Explorer, the header is added to a query like so:



Modify the startup of the backend by giving the function that handles the startup startStandaloneServer another parameter context

```

standaloneServer({
  listen: { port: 4000 },
  context: async ({ req, res }) => {
    const auth = req ? req.headers.authorization : null
    if (auth && auth.startsWith('Bearer ')) {
      const decodedToken = jwt.verify(
        auth.substring(7), process.env.JWT_SECRET
      )
      const currentUser = await User
        .findById (decodedToken.id).populate('friends')
      return { currentUser }
    }
  },
}).then(({ url }) => {
  console.log(`Server ready at ${url}`)
})

```

The object returned by context is given to all resolvers as their *third parameter*. Context is the right place to do things which are shared



by multiple resolvers, like [user identification](#).

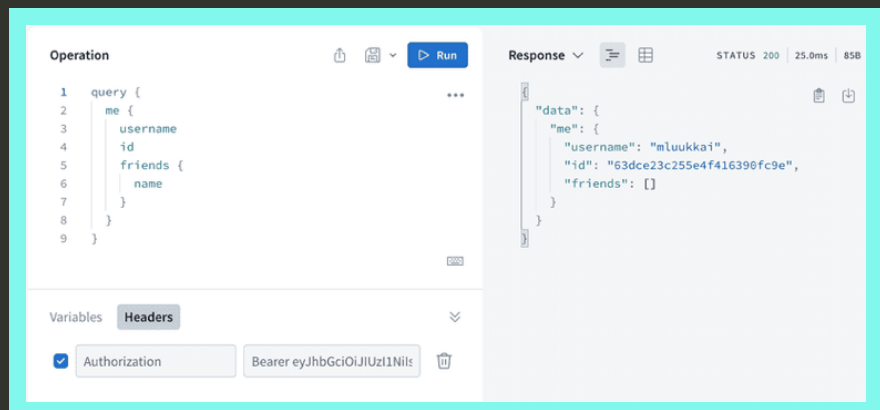
So our code sets the object corresponding to the user who made the request to the `currentUser` field of the context. If there is no user connected to the request, the value of the field is undefined.

The resolver of the `me` query is very simple: it just returns the logged-in user it receives in the `currentUser` field of the third parameter of the resolver, `context`. It's worth noting that if there is no logged-in user, i.e. there is no valid token in the header attached to the request, the query returns `null`:

```
Query: {  
  // ...  
  me: (root, args, context) => {  
    return context.currentUser  
  }  
},
```

copy

If the header has the correct value, the query returns the user information identified by the header



## Friends list

Let's complete the application's backend so that adding and editing persons requires logging in, and added persons are automatically added to the friends list of the user.

Let's first remove all persons not in anyone's friends list from the database.

`addPerson` mutation changes like so:

```
Mutation: {  
  addPerson: async (root, args, context) => {  
    const person = new Person({ ...args })  
    const currentUser = context.currentUser
```

copy

```

    if (!currentUser) {
      throw new GraphQLError('not authenticated', {
        extensions: {
          code: 'BAD_USER_INPUT',
        }
      })
    }

    try {
      await person.save()
      currentUser.friends =
        currentUser.friends.concat(person)
      await currentUser.save()
    } catch (error) {
      throw new GraphQLError('Saving user failed', {
        extensions: {
          code: 'BAD_USER_INPUT',
          invalidArgs: args.name,
          error
        }
      })
    }

    return person
  },
  //...
}

```

If a logged-in user cannot be found from the context, an `GraphQLError` with a proper message is thrown. Creating new persons is now done with `async/await` syntax, because if the operation is successful, the created person is added to the friends list of the user.

Let's also add functionality for adding an existing user to your friends list. The mutation is as follows:

```

type Mutation {
  // ...
  addAsFriend(
    name: String!
  ): User
}

```

copy

And the mutation's resolver:

```

addAsFriend: async (root, args, { currentUser }) => {
  const isFriend = (person) =>
    currentUser.friends.map(f =>
      f._id.toString()).includes(person._id.toString())

  if (!currentUser) {
    throw new GraphQLError('wrong credentials', {
      extensions: { code: 'BAD_USER_INPUT' }
    })
  }
}

```

```

    })
  }

  const person = await Person.findOne ({ name: args.name })
})
if ( !isFriend(person) ) {
  currentUser.friends =
currentUser.friends.concat(person)
}

await currentUser.save()

return currentUser
},

```

Note how the resolver *destructures* the logged-in user from the context. So instead of saving `currentUser` to a separate variable in a function

```

addAsFriend: async (root, args, context) => {
  const currentUser = context.currentUser

```

it is received straight in the parameter definition of the function:

```

addAsFriend: async (root, args, { currentUser }) => {

```

The following query now returns the user's friends list:

```

query {
  me {
    username
    friends{
      name
      phone
    }
  }
}

```

The code of the backend can be found on [Github](#) branch *part8-5*.

The following exercises are quite likely to break your frontend. Do not worry about it yet; the frontend shall be fixed and expanded in the next chapter.

### 8.13: Database, part 1

Change the library application so that it saves the data to a database. You can find the *mongoose schema* for books and authors from [here](#).

Let's change the book graphql schema a little

```
type Book {  
  title: String!  
  published: Int!  
  author: Author!  
  genres: [String!]!  
  id: ID!  
}
```

so that instead of just the author's name, the book object contains all the details of the author.

You can assume that the user will not try to add faulty books or authors, so you don't have to care about validation errors.

The following things do *not* have to work just yet:

- `allBooks` query with parameters
- `bookCount` field of an author object
- `author` field of a book
- `editAuthor` mutation

**Note:** despite the fact that author is now an *object* within a book, the schema for adding a book can remain same, only the *name* of the author is given as a parameter

```
type Mutation {  
  addBook(  
    title: String!  
    author: String!  
    published: Int!  
    genres: [String!]!  
  ): Book!  
  editAuthor(name: String!, setBornTo: Int!):  
  Author  
}
```

### 8.14: Database, part 2

Complete the program so that all queries (to get `allBooks` working with the parameter `author` and `bookCount` field of an author object is not required) and mutations work.

Regarding the *genre* parameter of the all books query, the situation is a bit more challenging. The solution is simple, but finding it can be a headache. You might benefit from [this](#).

### 8.15 Database, part 3

Complete the program so that database validation errors (e.g. book title or author name being too short) are handled sensibly. This means that they cause `GraphQLError` with a suitable error message to be thrown.

### 8.16 user and logging in

Add user management to your application. Expand the schema like so:

```
type User {  
  username: String!  
  favoriteGenre: String!  
  id: ID!  
}  
  
type Token {  
  value: String!  
}  
  
type Query {  
  // ..  
  me: User  
}  
  
type Mutation {  
  // ...  
  createUser(  
    username: String!  
    favoriteGenre: String!  
  ): User  
  login(  
    username: String!  
    password: String!  
  ): Token  
}
```

Create resolvers for query `me` and the new mutations `createUser` and `login`. Like in the course material,

you can assume all users have the same hardcoded password.

Make the mutations `addBook` and `editAuthor` possible only if the request includes a valid token.

(Don't worry about fixing the frontend for the moment.)

## Propose changes to material

Part 8b  
**Previous part**

Part 8d  
**Next part**



HOUSTON