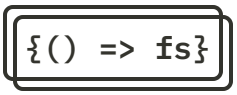


GraphQL-server

REST, familiar to us from the previous parts of the course, has long been the most prevalent way to implement the interfaces servers offer for browsers, and in general the integration between different applications on the web.

In recent years, GraphQL, developed by Facebook, has become popular for communication between web applications and servers.

The GraphQL philosophy is very different from REST. REST is *resource-based*. Every resource, for example a *user*, has its own address which identifies it, for example */users/10*. All operations done to the resource are done with HTTP requests to its URL. The action depends on the HTTP method used.

The resource-basedness of REST works well in most situations. However, it can be a bit awkward sometimes.

Let's consider the following example: our bloglist application contains some kind of social media functionality, and we would like to show a list of all the blogs that were added by users who have commented on any of the blogs we follow.

If the server implemented a REST API, we would probably have to do multiple HTTP requests from the browser before we had all the data we wanted. The requests would also return a lot of unnecessary data, and the code on the browser would probably be quite complicated.

If this was an often-used functionality, there could be a REST endpoint for it. If there were a lot of these kinds of scenarios however, it would become very laborious to implement REST endpoints for all of them.

A GraphQL server is well-suited for these kinds of situations.

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

The data described in the above scenario could be fetched with (roughly) the following query:

```
query FetchBlogsQuery {
  user(username: "mluukkai") {
    followedUsers {
      blogs {
        comments {
          user {
            blogs {
              title
            }
          }
        }
      }
    }
  }
}
```

The content of the `FetchBlogsQuery` can be roughly interpreted as: find a user named `"mluukkai"` and for each of his `followedUsers`, find all their `blogs`, and for each blog, all its `comments`, and for each `user` who wrote each comment, find their `blogs`, and return the `title` of each of them.

The server's response would be about the following JSON object:

```
{
  "data": {
    "followedUsers": [
      {
        "blogs": [
          {
            "comments": [
              {
                "user": {
                  "blogs": [
                    {
                      "title": "Goto considered harmful"
                    },
                    {
                      "title": "End to End Testing with Cypress is most enjoyable"
                    },
                    {
                      "title": "Navigating your transition to GraphQL"
                    },
                    {
```

```
                    "title": "From REST to GraphQL"
                  }
                ]
              }
            }
          ]
        }
      ]
    }
  ]
}
```

The application logic stays simple, and the code on the browser gets exactly the data it needs with a single query.

Schemas and queries

We will get to know the basics of GraphQL by implementing a GraphQL version of the phonebook application from parts 2 and 3.

In the heart of all GraphQL applications is a schema, which describes the data sent between the client and the server. The initial schema for our phonebook is as follows:

```
type Person {
  name: String!
  phone: String
  street: String!
  city: String!
  id: ID!
}

type Query {
  personCount: Int!
  allPersons: [Person!]!
  findPerson(name: String!): Person
}
```

The schema describes two types. The first type, *Person*, determines that persons have five fields. Four of the fields are type *String*, which is one of the scalar types of GraphQL. All of the String fields, except *phone*, must be given a value. This is marked by the exclamation mark on the schema. The type of the field *id* is *ID*. *ID* fields are strings, but GraphQL ensures they are unique.

The second type is a Query. Practically every GraphQL schema describes a Query, which tells what kind of queries can be made to the API.

The phonebook describes three different queries. `personCount` returns an integer, `allPersons` returns a list of *Person* objects and *findPerson* is given a string parameter and it returns a *Person* object.

Again, exclamation marks are used to mark which return values and parameters are *Non-Null*. `personCount` will, for sure, return an integer. The query `findPerson` must be given a string as a parameter. The query returns a *Person*-object or *null*. `allPersons` returns a list of *Person* objects, and the list does not contain any *null* values.

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

The simplest of the queries, `personCount`, looks as follows:

```
query {
  personCount
}
```

copy

Assuming our application has saved the information of three people, the response would look like this:

```
{
  "data": {
    "personCount": 3
  }
}
```

copy

The query fetching the information of all of the people, `allPersons`, is a bit more complicated. Because the query returns a list of *Person* objects, the query must describe *which fields* of the objects the query returns:

```
query {
  allPersons {
    name
    phone
  }
}
```

copy

The response could look like this:

```
{
  "data": {
    "allPersons": [
      {
        "name": "Arto Hellas",
        "phone": "040-123543"
      },
      {
```

copy

```
      "name": "Matti Luukkainen",
      "phone": "040-432342"
    },
    {
      "name": "Venla Ruuska",
      "phone": null
    }
  ]
 }
}
```

A query can be made to return any field described in the schema. For example, the following would also be possible:

```
query {
  allPersons{
    name
    city
    street
  }
}
```

The last example shows a query which requires a parameter, and returns the details of one person.

```
query {
  findPerson(name: "Arto Hellas") {
    phone
    city
    street
    id
  }
}
```

So, first, the parameter is described in round brackets, and then the fields of the return value object are listed in curly brackets.

The response is like this:

```
{
  "data": {
    "findPerson": {
      "phone": "040-123543",
      "city": "Espoo",
      "street": "Tapiolankatu 5 A"
      "id": "3d594650-3436-11e9-bc57-8b80ba54c431"
    }
```

```
  }
}
```

The return value was marked as nullable, so if we search for the details of an unknown

```
query {
  findPerson(name: "Joe Biden") {
    phone
  }
}
```

copy

the return value is *null*.

```
{
  "data": {
    "findPerson": null
  }
}
```

copy

As you can see, there is a direct link between a GraphQL query and the returned JSON object. One can think that the query describes what kind of data it wants as a response. The difference to REST queries is stark. With REST, the URL and the type of the request have nothing to do with the form of the returned data.

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

Despite its name, GraphQL does not actually have anything to do with databases. It does not care how the data is saved. The data a GraphQL API uses can be saved into a relational database, document database, or to other servers which a GraphQL server can access with for example REST.

Apollo Server

Let's implement a GraphQL server with today's leading library: Apollo Server.

Create a new npm project with `npm init` and install the required dependencies.

```
npm install @apollo/server graphql
```

copy

Also create a `index.js` file in your project's root directory.

The initial code is as follows:

```
const { ApolloServer } = require('@apollo/server')
const { startStandaloneServer } = require('@apollo/server/standalone')

let persons = [
  {
    name: "Arto Hellas",
    phone: "040-123543",
    street: "Tapiolankatu 5 A",
    city: "Espoo",
    id: "3d594650-3436-11e9-bc57-8b80ba54c431"
  },
  {
    name: "Matti Luukkainen",
    phone: "040-432342",
    street: "Malminkaari 10 A",
    city: "Helsinki",
    id: '3d599470-3436-11e9-bc57-8b80ba54c431'
  },
  {
    name: "Venla Ruuska",
    street: "Nallemäentie 22 C",
    city: "Helsinki",
    id: '3d599471-3436-11e9-bc57-8b80ba54c431'
  },
]

const typeDefs = `
  type Person {
    name: String!
    phone: String
    street: String!
    city: String!
    id: ID!
  }

  type Query {
    personCount: Int!
    allPersons: [Person!]!
    findPerson(name: String!): Person
  }
`

const resolvers = {
  Query: {
    personCount: () => persons.length,
    allPersons: () => persons,
    findPerson: (root, args) =>
      persons.find(p => p.name === args.name)
  }
}

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

```
})

startStandaloneServer(server, {
  listen: { port: 4000 },
}).then(({ url }) => {
  console.log(`Server ready at ${url}`)
})
```

The heart of the code is an `ApolloServer`, which is given two parameters:

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

The first parameter, `typeDefs`, contains the GraphQL schema.

The second parameter is an object, which contains the `resolvers` of the server. These are the code, which defines *how* GraphQL queries are responded to.

The code of the resolvers is the following:

```
const resolvers = {
  Query: {
    personCount: () => persons.length,
    allPersons: () => persons,
    findPerson: (root, args) =>
      persons.find(p => p.name === args.name)
  }
}
```

As you can see, the resolvers correspond to the queries described in the schema.

```
type Query {
  personCount: Int!
  allPersons: [Person!]!
  findPerson(name: String!): Person
}
```

So there is a field under *Query* for every query described in the schema.

The query

```
query {
  personCount
}
```

Has the resolver

```
() => persons.length
```

So the response to the query is the length of the array `persons`.

The query which fetches all persons

```
query {
  allPersons {
    name
  }
}
```

has a resolver which returns *all* objects from the `persons` array.

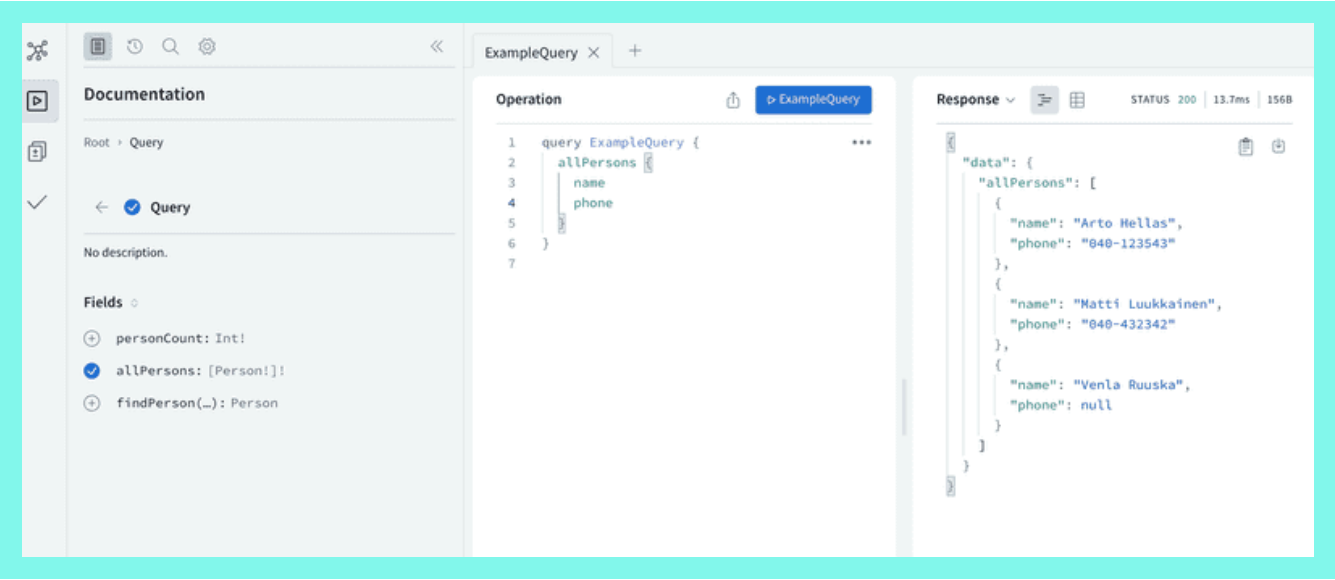
```
() => persons
```

Start the server by running `node index.js` in the terminal.

Apollo Studio Explorer

When Apollo server is run in development mode the page http://localhost:4000 has a button *Query your server* that takes us to Apollo Studio Explorer. This is very useful for a developer, and can be used to make queries to the server.

Let's try it out:



At the left side Explorer shows the API-documentation that it has automatically generated based on the schema.

Parameters of a resolver

The query fetching a single person

```
query {
  findPerson(name: "Arto Hellas") {
    phone
    city
    street
  }
}
```

has a resolver which differs from the previous ones because it is given *two parameters*:

```
(root, args) => persons.find(p => p.name === args.name)
```

The second parameter, `args`, contains the parameters of the query. The resolver then returns from the array `persons` the person whose name is the same as the value of *args.name*. The resolver does not need the first parameter `root`.

In fact, all resolver functions are given four parameters. With JavaScript, the parameters don't have to be defined if they are not needed. We will be using the first and the third parameter of a resolver later in this part.

The default resolver

When we do a query, for example

```
query {
  findPerson(name: "Arto Hellas") {
    phone
    city
    street
  }
}
```

the server knows to send back exactly the fields required by the query. How does that happen?

A GraphQL server must define resolvers for *each* field of each type in the schema. We have so far only defined resolvers for fields of the type *Query*, so for each query of the application.

Because we did not define resolvers for the fields of the type *Person*, Apollo has defined default resolvers for them. They work like the one shown below:

```
const resolvers = {
  Query: {
    personCount: () => persons.length,
    allPersons: () => persons,
    findPerson: (root, args) => persons.find(p => p.name === args.name)
  },
  Person: {
    name: (root) => root.name,
    phone: (root) => root.phone,
    street: (root) => root.street,
    city: (root) => root.city,
    id: (root) => root.id
  }
}
```

The default resolver returns the value of the corresponding field of the object. The object itself can be accessed through the first parameter of the resolver, `root`.

If the functionality of the default resolver is enough, you don't need to define your own. It is also possible to define resolvers for only some fields of a type, and let the default resolvers handle the rest.

We could for example define that the address of all persons is *Manhattan New York* by hard-coding the following to the resolvers of the street and city fields of the type *Person*:

```
Person: {
  street: (root) => "Manhattan",
```

```
  city: (root) => "New York"
}
```

Object within an object

Let's modify the schema a bit

```
type Address {
  street: String!
  city: String!
}

type Person {
  name: String!
  phone: String
  address: Address!
  id: ID!
}

type Query {
  personCount: Int!
  allPersons: [Person!]!
  findPerson(name: String!): Person
}
```

so a person now has a field with the type *Address*, which contains the street and the city.

The queries requiring the address change into

```
query {
  findPerson(name: "Arto Hellas") {
    phone
    address {
      city
      street
    }
  }
}
```

and the response is now a person object, which *contains* an address object.

```
{
  "data": {
    "findPerson": {
      "phone": "040-123543",
```

```
      "address":  {
        "city": "Espoo",
        "street": "Tapiolankatu 5 A"
      }
    }
  }
}
```

We still save the persons in the server the same way we did before.

```
let persons = [
  {
    name: "Arto Hellas",
    phone: "040-123543",
    street: "Tapiolankatu 5 A",
    city: "Espoo",
    id: "3d594650-3436-11e9-bc57-8b80ba54c431"
  },
  // ...
]
```

The person-objects saved in the server are not exactly the same as the GraphQL type *Person* objects described in the schema.

Contrary to the *Person* type, the *Address* type does not have an *id* field, because they are not saved into their own separate data structure in the server.

Because the objects saved in the array do not have an *address* field, the default resolver is not sufficient. Let's add a resolver for the *address* field of *Person* type :

```
const resolvers = {
  Query: {
    personCount: () => persons.length,
    allPersons: () => persons,
    findPerson: (root, args) =>
      persons.find(p => p.name === args.name)
  },
  Person: {
    address: (root) => {
      return {
        street: root.street,
        city: root.city
      }
    }
  }
}
```

So every time a *Person* object is returned, the fields *name*, *phone* and *id* are returned using their default resolvers, but the field *address* is formed by using a self-defined resolver. The parameter `root` of the resolver function is the person-object, so the street and the city of the address can be taken from its fields.

The current code of the application can be found on Github, branch *part8-1*.

Mutations

Let's add a functionality for adding new persons to the phonebook. In GraphQL, all operations which cause a change are done with mutations. Mutations are described in the schema as the keys of type *Mutation*.

The schema for a mutation for adding a new person looks as follows:

```
type Mutation {
  addPerson(
    name: String!
    phone: String
    street: String!
    city: String!
  ): Person
}
```

The Mutation is given the details of the person as parameters. The parameter *phone* is the only one which is nullable. The Mutation also has a return value. The return value is type *Person*, the idea being that the details of the added person are returned if the operation is successful and if not, null. Value for the field *id* is not given as a parameter. Generating an id is better left for the server.

Mutations also require a resolver:

```
const { v1: uuid } = require('uuid')

// ...

const resolvers = {
  // ...
  Mutation: {
    addPerson: (root, args) => {
      const person = { ...args, id: uuid() }
      persons = persons.concat(person)
      return person
    }
  }
}
```

The mutation adds the object given to it as a parameter `args` to the array `persons`, and returns the object it added to the array.

The *id* field is given a unique value using the `uuid` library.

A new person can be added with the following mutation

```
mutation {
  addPerson(
    name: "Pekka Mikkola"
    phone: "045-2374321"
    street: "Vilppulantie 25"
    city: "Helsinki"
  ) {
    name
    phone
    address{
      city
      street
    }
    id
  }
}
```

Note that the person is saved to the `persons` array as

```
{
  name: "Pekka Mikkola",
  phone: "045-2374321",
  street: "Vilppulantie 25",
  city: "Helsinki",
  id: "2b24e0b0-343c-11e9-8c2a-cb57c2bf804f"
}
```

But the response to the mutation is

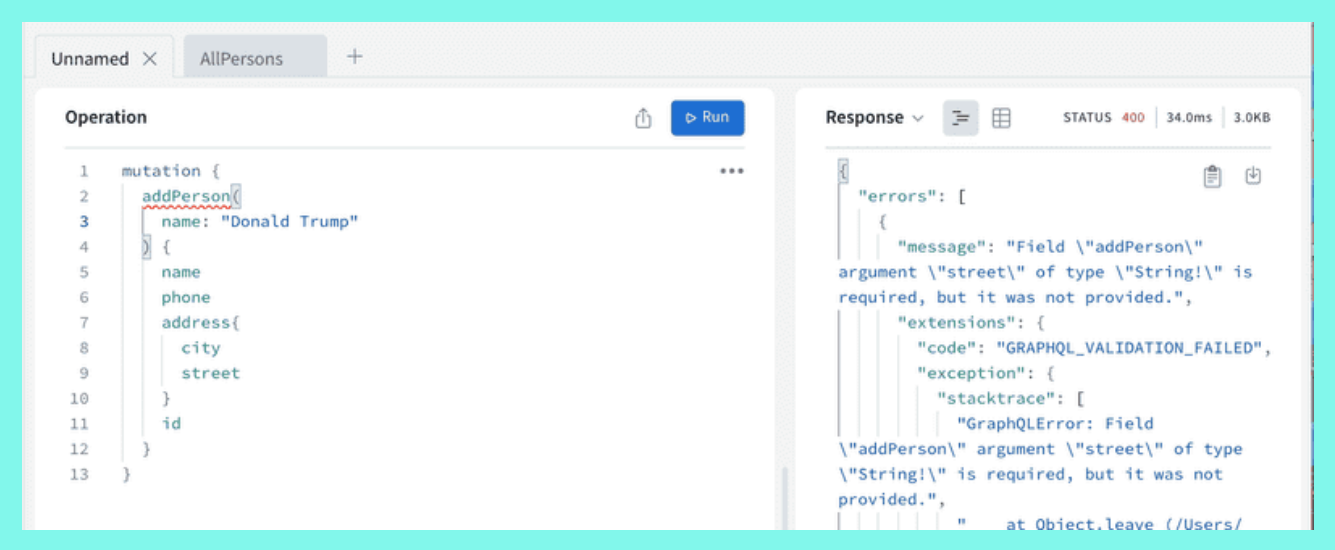
```
{
  "data": {
    "addPerson": {
      "name": "Pekka Mikkola",
      "phone": "045-2374321",
      "address": {
        "city": "Helsinki",
        "street": "Vilppulantie 25"
      },
      "id": "2b24e0b0-343c-11e9-8c2a-cb57c2bf804f"
    }
```

```
  }
}
```

So the resolver of the *address* field of the *Person* type formats the response object to the right form.

Error handling

If we try to create a new person, but the parameters do not correspond with the schema description, the server gives an error message:



So some of the error handling can be automatically done with GraphQL validation.

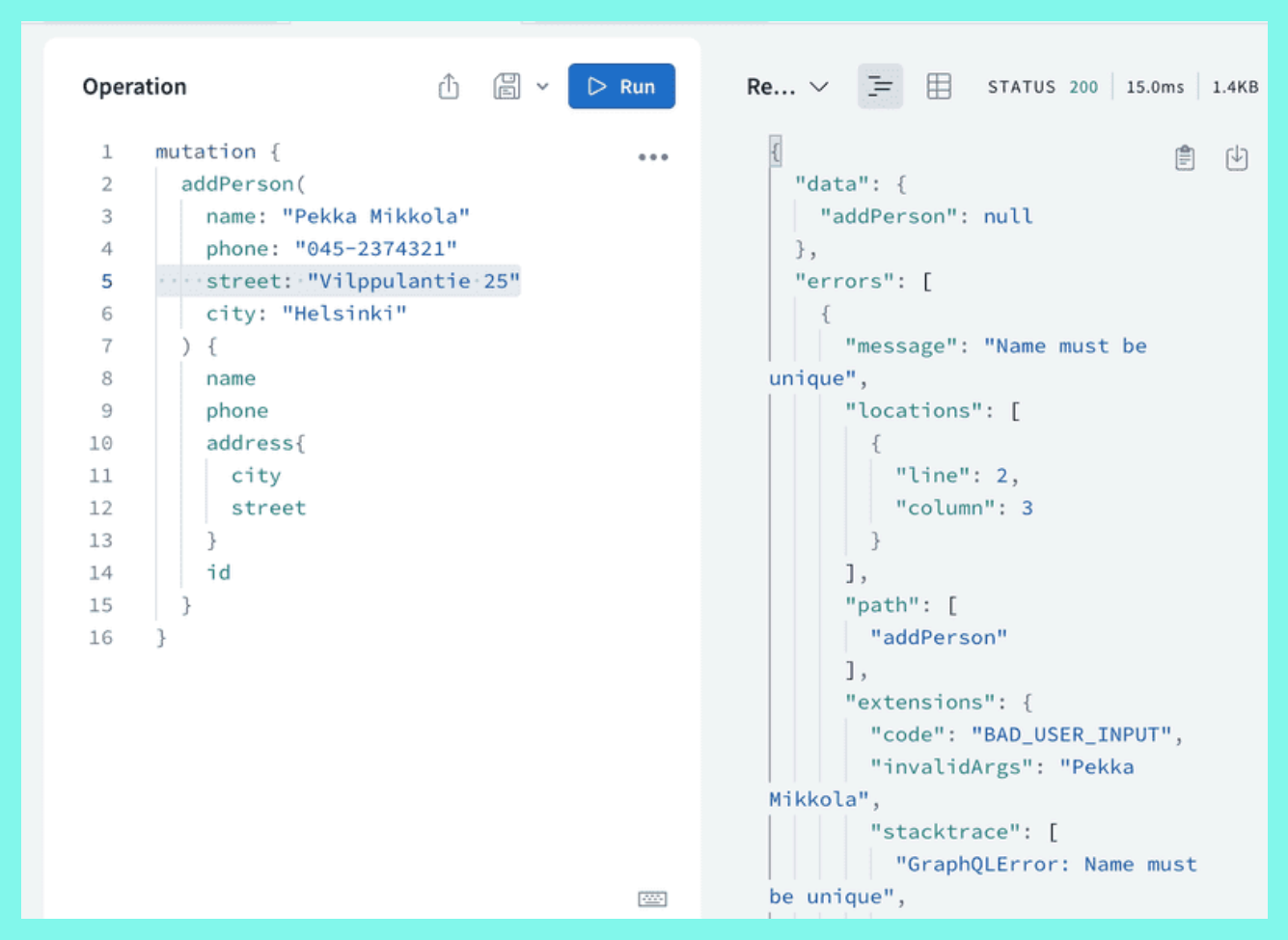
However, GraphQL cannot handle everything automatically. For example, stricter rules for data sent to a Mutation have to be added manually. An error could be handled by throwing GraphQLError with a proper error code.

Let's prevent adding the same name to the phonebook multiple times:

```
const { GraphQLError } = require('graphql')

// ...

const resolvers = {
  // ..
  Mutation: {
    addPerson: (root, args) => {
      if (persons.find(p => p.name === args.name)) {
        throw new GraphQLError('Name must be unique', {
          extensions: {
            code: 'BAD_USER_INPUT',
            invalidArgs: args.name
          }
        })
      }
```

```
      const person = { ...args, id: uuid() }
      persons = persons.concat(person)
      return person
    }
  }
}
```

So if the name to be added already exists in the phonebook, throw `GraphQLError` error.



The current code of the application can be found on GitHub, branch *part8-2*.

Enum

Let's add a possibility to filter the query returning all persons with the parameter *phone* so that it returns only persons with a phone number

```
query {
  allPersons(phone: YES) {
    name
    phone
  }
}
```

or persons without a phone number

```
query {
  allPersons(phone: NO) {
    name
  }
}
```

The schema changes like so:

```
enum YesNo {
  YES
  NO
}

type Query {
  personCount: Int!
  allPersons(phone: YesNo): [Person!]!
  findPerson(name: String!): Person
}
```

The type *YesNo* is a GraphQL enum, or an enumerable, with two possible values: *YES* or *NO*. In the query `allPersons`, the parameter `phone` has the type *YesNo*, but is nullable.

The resolver changes like so:

```
Query: {
  personCount: () => persons.length,
  allPersons: (root, args) => {
    if (!args.phone) {
      return persons
    }
    const byPhone = (person) =>
      args.phone === 'YES' ? person.phone : !person.phone
    return persons.filter(byPhone)
  },
  findPerson: (root, args) =>
    persons.find(p => p.name === args.name)
},
```

Changing a phone number

Let's add a mutation for changing the phone number of a person. The schema of this mutation looks as follows:

```
type Mutation {
  addPerson(
    name: String!
    phone: String
    street: String!
    city: String!
  ): Person
  editNumber(
    name: String!
    phone: String!
  ): Person
}
```

and is done by a resolver:

```
Mutation: {
  // ...
  editNumber: (root, args) => {
    const person = persons.find(p => p.name === args.name)
    if (!person) {
      return null
    }

    const updatedPerson = { ...person, phone: args.phone }
    persons = persons.map(p => p.name === args.name ? updatedPerson : p)
    return updatedPerson
  }
}
```

The mutation finds the person to be updated by the field *name*.

The current code of the application can be found on Github, branch *part8-3*.

More on queries

With GraphQL, it is possible to combine multiple fields of type *Query*, or "separate queries" into one query. For example, the following query returns both the amount of persons in the phonebook and their names:

```
query {
  personCount
  allPersons {
    name
  }
}
```

The response looks as follows:

```
{
  "data": {
    "personCount": 3,
    "allPersons": [
      {
        "name": "Arto Hellas"
      },
      {
        "name": "Matti Luukkainen"
      },
      {
        "name": "Venla Ruuska"
      }
    ]
  }
}
```

Combined query can also use the same query multiple times. You must however give the queries alternative names like so:

```
query {
  havePhone: allPersons(phone: YES){
    name
  }
  phoneless: allPersons(phone: NO){
    name
  }
}
```

The response looks like:

```
{
  "data": {
    "havePhone": [
      {
        "name": "Arto Hellas"
      },
      {
        "name": "Matti Luukkainen"
      }
    ],
    "phoneless": [
      {
        "name": "Venla Ruuska"
      }
```

```
    ]
  }
}
```

In some cases, it might be beneficial to name the queries. This is the case especially when the queries or mutations have parameters. We will get into parameters soon.

Exercises 8.1.-8.7

Through the exercises, we will implement a GraphQL backend for a small library. Start with this file. Remember to `npm init` and to install dependencies!

8.1: The number of books and authors

Implement queries `bookCount` and `authorCount` which return the number of books and the number of authors.

The query

```
query {
  bookCount
  authorCount
}
```

should return

```
{
  "data": {
    "bookCount": 7,
    "authorCount": 5
  }
}
```

8.2: All books

Implement query `allBooks`, which returns the details of all books.

In the end, the user should be able to do the following query:

```
query {
  allBooks {
    title
    author
```

```
    published
    genres
  }
}
```

8.3: All authors

Implement query `allAuthors`, which returns the details of all authors. The response should include a field `bookCount` containing the number of books the author has written.

For example the query

```
query {
  allAuthors {
    name
    bookCount
  }
}
```

copy

should return

```
{
  "data": {
    "allAuthors": [
      {
        "name": "Robert Martin",
        "bookCount": 2
      },
      {
        "name": "Martin Fowler",
        "bookCount": 1
      },
      {
        "name": "Fyodor Dostoevsky",
        "bookCount": 2
      },
      {
        "name": "Joshua Kerievsky",
        "bookCount": 1
      },
      {
        "name": "Sandi Metz",
        "bookCount": 1
      }
    ]
  }
}
```

copy

8.4: Books of an author

Modify the `allBooks` query so that a user can give an optional parameter *author*. The response should include only books written by that author.

For example query

```
query {
  allBooks(author: "Robert Martin") {
    title
  }
}
```

copy

should return

```
{
  "data": {
    "allBooks": [
      {
        "title": "Clean Code"
      },
      {
        "title": "Agile software development"
      }
    ]
  }
}
```

copy

8.5: Books by genre

Modify the query `allBooks` so that a user can give an optional parameter *genre*. The response should include only books of that genre.

For example query

```
query {
  allBooks(genre: "refactoring") {
    title
    author
  }
}
```

copy

should return

```
{
  "data": {
    "allBooks": [
      {
        "title": "Clean Code",
        "author": "Robert Martin"
      },
      {
        "title": "Refactoring, edition 2",
        "author": "Martin Fowler"
      },
      {
        "title": "Refactoring to patterns",
        "author": "Joshua Kerievsky"
      },
      {
        "title": "Practical Object-Oriented Design, An Agile Primer Using Ruby",
        "author": "Sandi Metz"
      }
    ]
  }
}
```

The query must work when both optional parameters are given:

```
query {
  allBooks(author: "Robert Martin", genre: "refactoring") {
    title
    author
  }
}
```

8.6: Adding a book

Implement mutation `addBook`, which can be used like this:

```
mutation {
  addBook(
    title: "NoSQL Distilled",
    author: "Martin Fowler",
    published: 2012,
    genres: ["database", "nosql"]
  ) {
    title,
    author
  }
}
```

The mutation works even if the author is not already saved to the server:

```
mutation {
  addBook(
    title: "Pimeyden tango",
    author: "Reijo Mäki",
    published: 1997,
    genres: ["crime"]
  ) {
    title,
    author
  }
}
```

If the author is not yet saved to the server, a new author is added to the system. The birth years of authors are not saved to the server yet, so the query

```
query {
  allAuthors {
    name
    born
    bookCount
  }
}
```

returns

```
{
  "data": {
    "allAuthors": [
      // ...
      {
        "name": "Reijo Mäki",
        "born": null,
        "bookCount": 1
      }
    ]
  }
}
```

8.7: Updating the birth year of an author

Implement mutation `editAuthor`, which can be used to set a birth year for an author. The mutation is used like so:

```
mutation {
  editAuthor(name: "Reijo Mäki", setBornTo: 1958) {
    name
    born
  }
}
```

If the correct author is found, the operation returns the edited author:

```
{
  "data": {
    "editAuthor": {
      "name": "Reijo Mäki",
      "born": 1958
    }
  }
}
```

If the author is not in the system, *null* is returned:

```
{
  "data": {
    "editAuthor": null
  }
}
```

Propose changes to material

Part 7
Previous part

Part 8b
Next part

About course

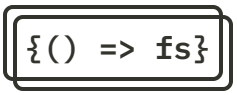
Course contents

FAQ

Partners

Challenge


HOUSTON

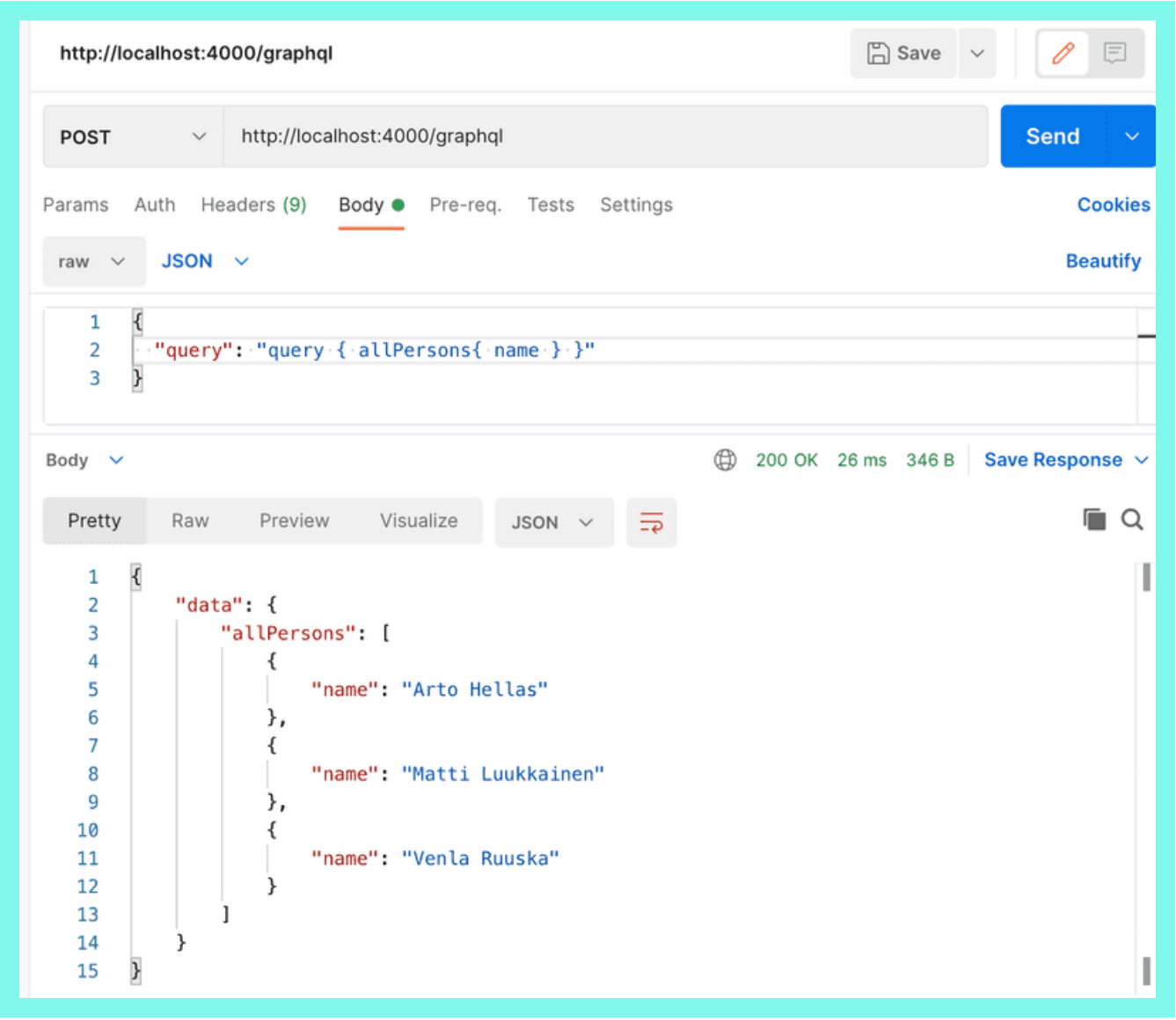


b React and GraphQL

We will next implement a React app that uses the GraphQL server we created.

The current code of the server can be found on GitHub, branch *part8-3*.

In theory, we could use GraphQL with HTTP POST requests. The following shows an example of this with Postman:



The communication works by sending HTTP POST requests to http://localhost:4000/graphql. The query itself is a string sent as the value of the key *query*.

We could take care of the communication between the React app and GraphQL by using Axios. However, most of the time, it is not very sensible to do so. It is a better idea to use a higher-order library capable of abstracting the unnecessary details of the communication.

At the moment, there are two good options: Relay by Facebook and Apollo Client, which is the client side of the same library we used in the previous section. Apollo is absolutely the most popular of the two, and we will use it in this section as well.

Apollo client

Let us create a new React app, and can continue installing dependencies required by Apollo client.

```
npm install @apollo/client graphql
```

copy

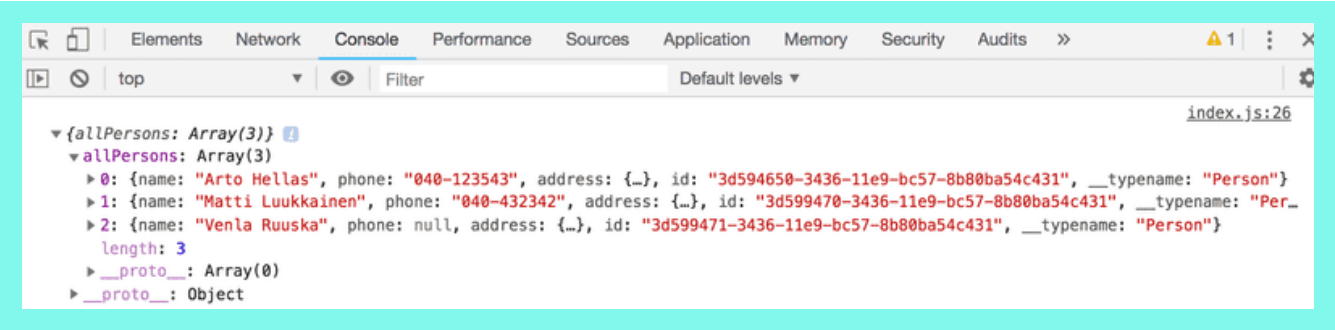
We'll start with the following code for our application:

```
import ReactDOM from 'react-dom/client'
import App from './App'

import { ApolloClient, InMemoryCache, gql } from '@apollo/client'

const client = new ApolloClient({
  uri: 'http://localhost:4000',
  cache: new InMemoryCache(),
})

const query = gql`
  query {
    allPersons  {
      name,
      phone,
      address {
        street,
        city
      }
      id
    }
  }
`

client.query({ query })
  .then((response) => {
    console.log(response.data)
  })


ReactDOM.createRoot(document.getElementById('root')).render(<App />)
```

The beginning of the code creates a new `client` object, which is then used to send a query to the server:

```
client.query({ query })
  .then((response) => {
    console.log(response.data)
  })
```

The server's response is printed to the console:



The application can communicate with a GraphQL server using the `client` object. The client can be made accessible for all components of the application by wrapping the *App* component with ApolloProvider.

```
import ReactDOM from 'react-dom/client'
import App from './App'

import {
  ApolloClient,
  ApolloProvider,
  InMemoryCache,
} from '@apollo/client'

const client = new ApolloClient({
  uri: 'http://localhost:4000',
  cache: new InMemoryCache(),
})

ReactDOM.createRoot(document.getElementById('root')).render(
  <ApolloProvider client={client}>
    <App />
  </ApolloProvider>
)
```

Making queries

We are ready to implement the main view of the application, which shows a list of person's name and phone number.

Apollo Client offers a few alternatives for making queries. Currently, the use of the hook function useQuery is the dominant practice.

The query is made by the *App* component, the code of which is as follows:

```
import { gql, useQuery } from '@apollo/client'

const ALL_PERSONS = gql`
query {
```

```
    allPersons {
      name
      phone
      id
    }
}
`

const App = () => {
  const result = useQuery(ALL_PERSONS)

  if (result.loading) {
    return <div>loading...</div>
  }

  return (
    <div>
      {result.data.allPersons.map(p => p.name).join(', ')}
    </div>
  )
}

export default App
```

When called, `useQuery` makes the query it receives as a parameter. It returns an object with multiple fields. The field *loading* is true if the query has not received a response yet. Then the following code gets rendered:

```
if (result.loading) {
  return <div>loading...</div>
}
```

When a response is received, the result of the *allPersons* query can be found in the data field, and we can render the list of names to the screen.

```
<div>
  {result.data.allPersons.map(p => p.name).join(', ')}
</div>
```

Let's separate displaying the list of persons into its own component:

```
const Persons = ({ persons }) => {
  return (
    <div>
      <h2>Persons</h2>
      {persons.map(p =>
        <div key={p.name}>
```

```
          {p.name} {p.phone}
        </div>
      )}
    </div>
  )
}
```

The `App` component still makes the query, and passes the result to the new component to be rendered:

```
const App = () => {
  const result = useQuery(ALL_PERSONS)

  if (result.loading)  {
    return <div>loading...</div>
  }

  return (
    <Persons persons={result.data.allPersons}/>
  )
}
```

Named queries and variables

Let's implement functionality for viewing the address details of a person. The *findPerson* query is well-suited for this.

The queries we did in the last chapter had the parameter hardcoded into the query:

```
query {
  findPerson(name: "Arto Hellas") {
    phone
    city
    street
    id
  }
}
```

When we do queries programmatically, we must be able to give them parameters dynamically.

GraphQL variables are well-suited for this. To be able to use variables, we must also name our queries.

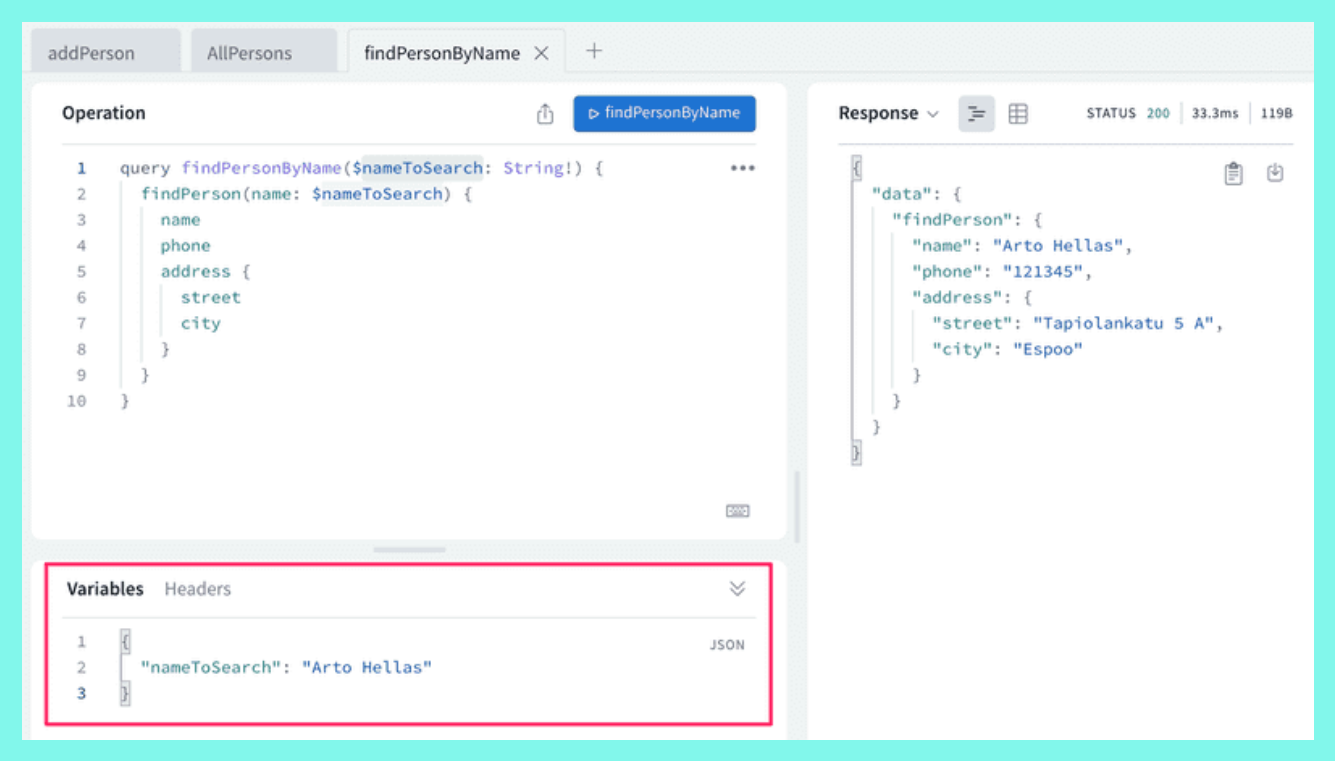
A good format for the query is this:

```
query findPersonByName($nameToSearch: String!) {
  findPerson(name: $nameToSearch) {
    name
```

```
    phone
    address {
      street
      city
    }
  }
}
```

The name of the query is *findPersonByName*, and it is given a string *$nameToSearch* as a parameter.

It is also possible to do queries with parameters with the Apollo Explorer. The parameters are given in *Variables*:



The `useQuery` hook is well-suited for situations where the query is done when the component is rendered. However, we now want to make the query only when a user wants to see the details of a specific person, so the query is done only as required.

One possibility for this kind of situations is the hook function useLazyQuery that would make it possible to define a query which is executed *when* the user wants to see the detailed information of a person.

However, in our case we can stick to `useQuery` and use the option skip, which makes it possible to do the query only if a set condition is true.

The solution is as follows:

```
import { useState } from 'react'
import { gql, useQuery } from '@apollo/client'

const FIND_PERSON = gql`
  query findPersonByName($nameToSearch: String!) {
```

```
    findPerson(name: $nameToSearch) {
      name
      phone
      id
      address {
        street
        city
      }
    }
  }
`

const Person = ({ person, onClose }) => {
  return (
    <div>
      <h2>{person.name}</h2>
      <div>
        {person.address.street} {person.address.city}
      </div>
      <div>{person.phone}</div>
      <button onClick={onClose}>close</button>
    </div>
  )
}

const Persons = ({ persons }) => {
  const [nameToSearch, setNameToSearch] = useState(null)
  const result = useQuery(FIND_PERSON, {
    variables: { nameToSearch },
    skip: !nameToSearch,
  })

  if (nameToSearch && result.data) {
    return (
      <Person
        person={result.data.findPerson}
        onClose={() => setNameToSearch(null)}
      />
    )
  }

  return (
    <div>
      <h2>Persons</h2>
      {persons.map((p) => (
        <div key={p.name}>
          {p.name} {p.phone}
          <button onClick={() => setNameToSearch(p.name)}>
            show address
          </button>
        </div>
      ))}
    </div>
  )
```

```
}

export default Persons
```

The code has changed quite a lot, and all of the changes are not completely apparent.

When the button *show address* of a person is pressed, the name of the person is set to state *nameToSearch*:

```
<button onClick={() => setNameToSearch(p.name)}>
  show address
</button>
```

This causes the component to re-render itself. On render the query *FIND_PERSON* that fetches the detailed information of a user is executed *if the variable nameToSearch* has a value:

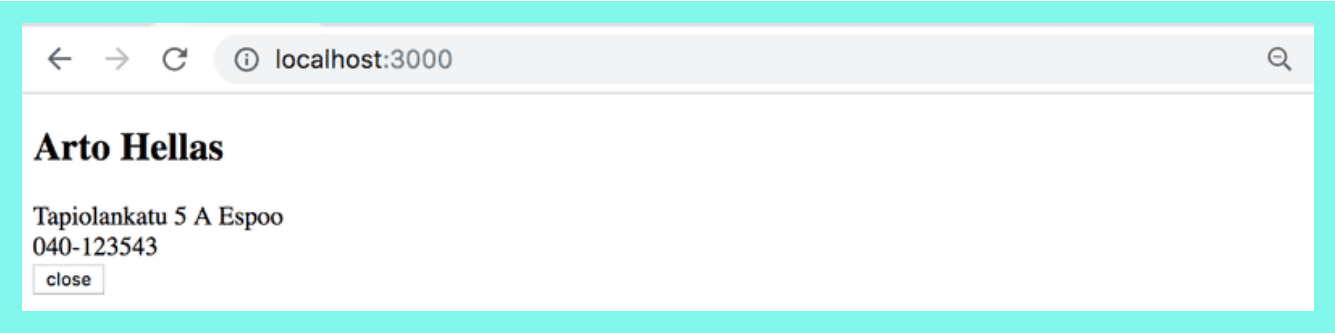
```
const result = useQuery(FIND_PERSON, {
  variables: { nameToSearch },
  skip: !nameToSearch,
})
```

When the user is not interested in seeing the detailed info of any person, the state variable *nameToSearch* is null and the query is not executed.

If the state *nameToSearch* has a value and the query result is ready, the component *Person* renders the detailed info of a person:

```
if (nameToSearch && result.data) {
  return (
    <Person
      person={result.data.findPerson}
      onClose={() => setNameToSearch(null)}
    />
  )
}
```

A single-person view looks like this:

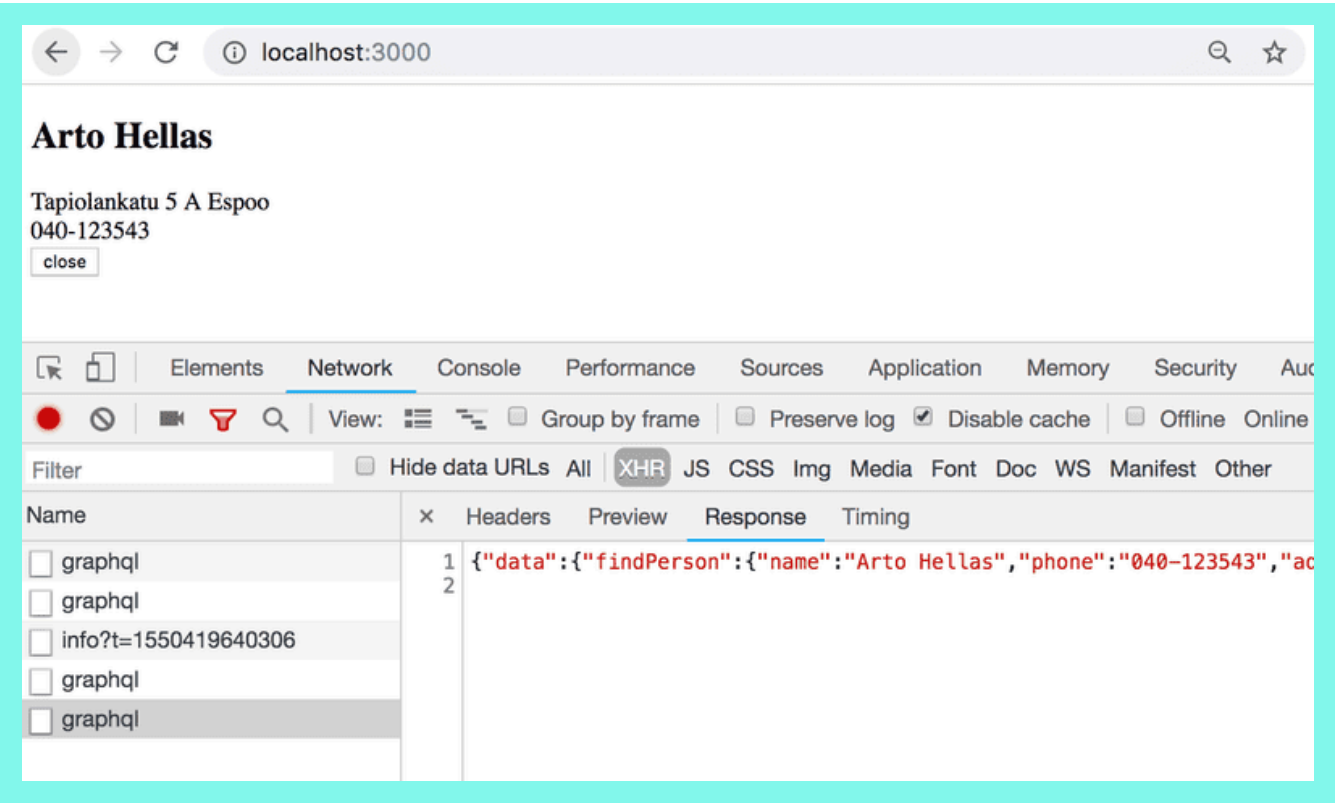


When a user wants to return to the person list, the `nameToSearch` state is set to `null`.

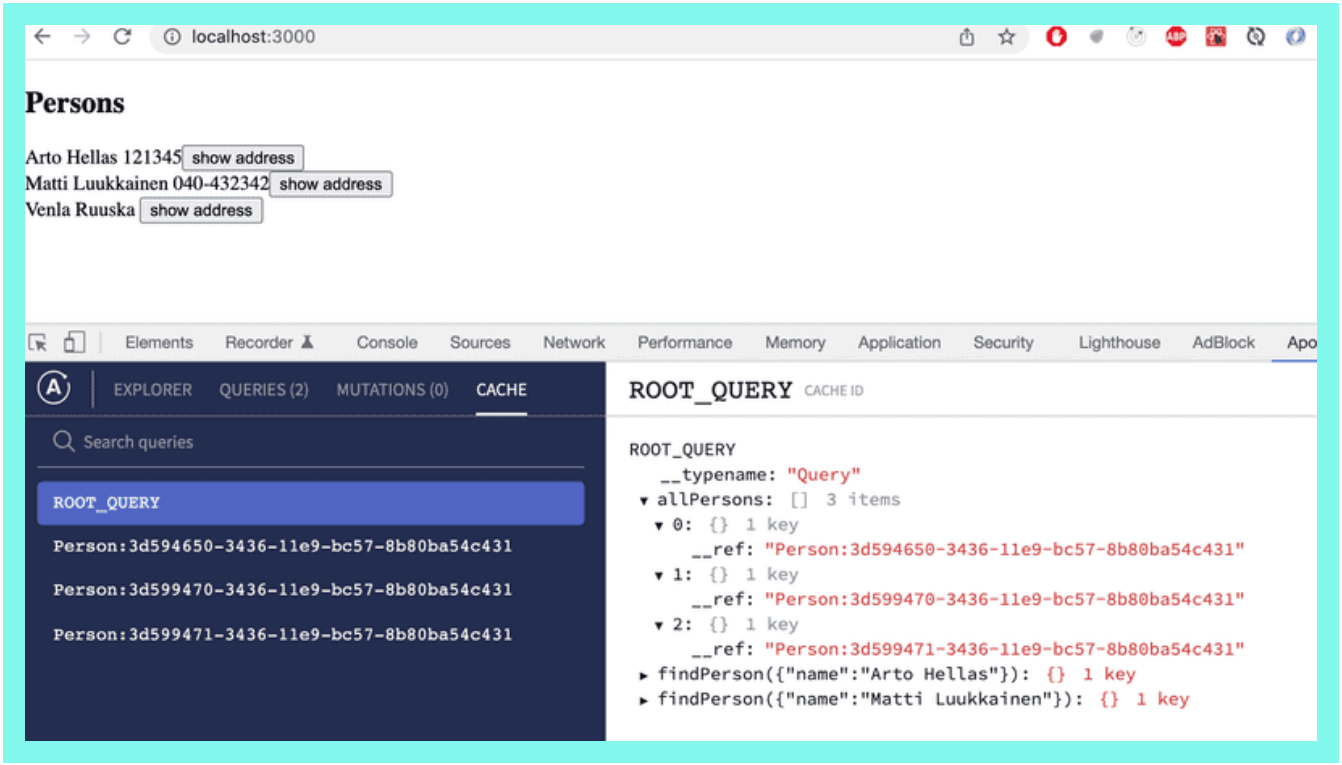
The current code of the application can be found on GitHub branch *part8-1*.

Cache

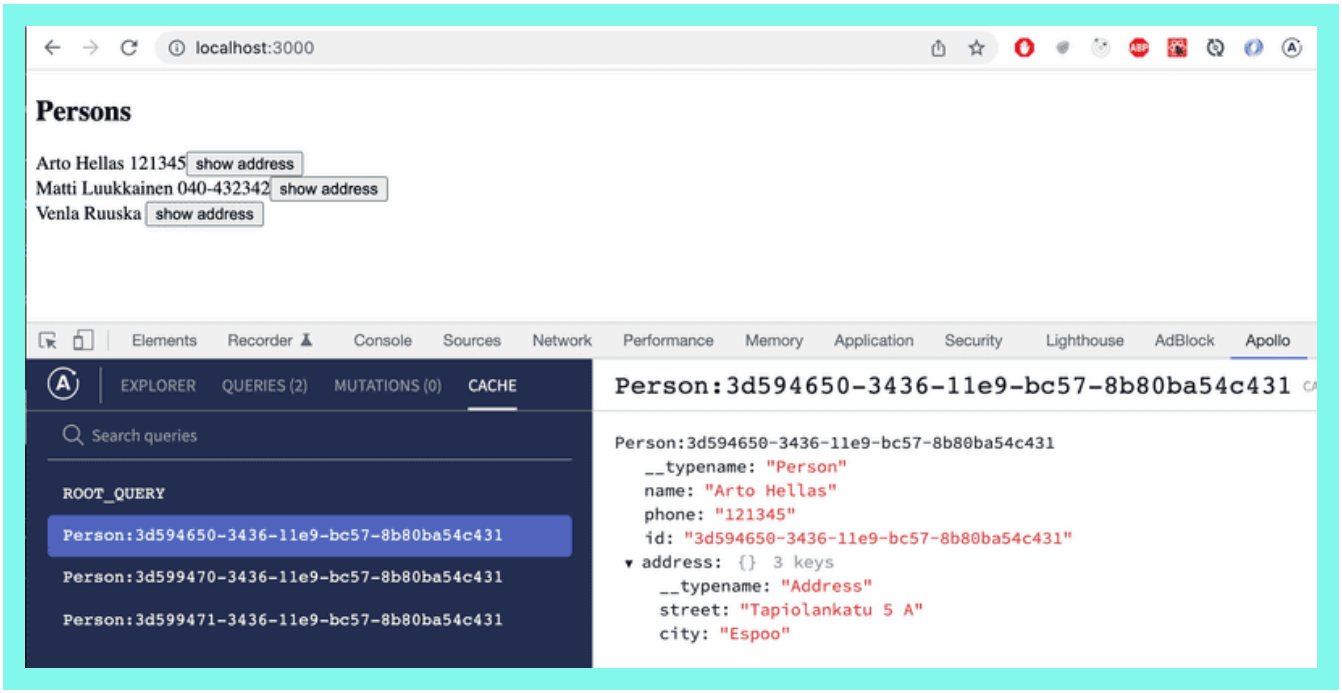
When we do multiple queries, for example with the address details of Arto Hellas, we notice something interesting: the query to the backend is done only the first time around. After this, despite the same query being done again by the code, the query is not sent to the backend.



Apollo client saves the responses of queries to cache. To optimize performance if the response to a query is already in the cache, the query is not sent to the server at all.



Cache shows the detailed info of Arto Hellas after the query *findPerson*:



Doing mutations

Let's implement functionality for adding new persons.

In the previous chapter, we hardcoded the parameters for mutations. Now, we need a version of the addPerson mutation which uses variables:

```
const CREATE_PERSON = gql`
mutation createPerson($name: String!, $street: String!, $city: String!, $phone: String) {
  addPerson(
    name: $name,
    street: $street,
    city: $city,
    phone: $phone
  ) {
    name
    phone
    id
    address {
      street
      city
    }
  }
}
`
```

The hook function `useMutation` provides the functionality for making mutations.

Let's create a new component for adding a new person to the directory:

```
import { useState } from 'react'
import { gql, useMutation } from '@apollo/client'

const CREATE_PERSON = gql`
  // ...
`

const PersonForm = () => {
  const [name, setName] = useState('')
  const [phone, setPhone] = useState('')
  const [street, setStreet] = useState('')
  const [city, setCity] = useState('')

  const [ createPerson ] = useMutation(CREATE_PERSON)

  const submit = (event) => {
    event.preventDefault()

    createPerson({  variables: { name, phone, street, city } })

    setName('')
    setPhone('')
    setStreet('')
    setCity('')
  }

  return (
    <div>
```

```
        <h2>create new</h2>
        <form onSubmit={submit}>
          <div>
            name <input value={name}
              onChange={({ target }) => setName(target.value)}
            />
          </div>
          <div>
            phone <input value={phone}
              onChange={({ target }) => setPhone(target.value)}
            />
          </div>
          <div>
            street <input value={street}
              onChange={({ target }) => setStreet(target.value)}
            />
          </div>
          <div>
            city <input value={city}
              onChange={({ target }) => setCity(target.value)}
            />
          </div>
          <button type='submit'>add!</button>
        </form>
      </div>
    )
}

export default PersonForm
```

The code of the form is straightforward and the interesting lines have been highlighted. We can define mutation functions using the `useMutation` hook. The hook returns an *array*, the first element of which contains the function to cause the mutation.

```
const [ createPerson ] = useMutation(CREATE_PERSON)
```

copy

The query variables receive values when the query is made:

```
createPerson({  variables: { name, phone, street, city } })
```

copy

New persons are added just fine, but the screen is not updated. This is because Apollo Client cannot automatically update the cache of an application, so it still contains the state from before the mutation. We could update the screen by reloading the page, as the cache is emptied when the page is reloaded. However, there must be a better way to do this.

Updating the cache

There are a few different solutions for this. One way is to make the query for all persons poll the server, or make the query repeatedly.

The change is small. Let's set the query to poll every two seconds:

```
const App = () => {
  const result = useQuery(ALL_PERSONS, {
    pollInterval: 2000
  })

  if (result.loading)  {
    return <div>loading...</div>
  }

  return (
    <div>
      <Persons persons = {result.data.allPersons}/>
      <PersonForm />
    </div>
  )
}

export default App
```

The solution is simple, and every time a user adds a new person, it appears immediately on the screens of all users.

The bad side of the solution is all the pointless web traffic.

Another easy way to keep the cache in sync is to use the `useMutation` hook's refetchQueries parameter to define that the query fetching all persons is done again whenever a new person is created.

```
const ALL_PERSONS = gql`
  query  {
    allPersons  {
      name
      phone
      id
    }
  }
`

const PersonForm = (props) => {
  // ...

  const [ createPerson ] = useMutation(CREATE_PERSON, {
```

```
    refetchQueries: [ { query: ALL_PERSONS } ]
  })
```

The pros and cons of this solution are almost opposite of the previous one. There is no extra web traffic because queries are not done just in case. However, if one user now updates the state of the server, the changes do not show to other users immediately.

If you want to do multiple queries, you can pass multiple objects inside refetchQueries. This will allow you to update different parts of your app at the same time. Here is an example:

```
    const [ createPerson ] = useMutation(CREATE_PERSON, {
    refetchQueries: [ { query: ALL_PERSONS }, { query: OTHER_QUERY }, { query: ... } ] //
pass as many queries as you need
  })
```

There are other ways to update the cache. More about those later in this part.

At the moment, queries and components are defined in the same place in our code. Let's separate the query definitions into their own file *queries.js*:

```
import { gql } from '@apollo/client'

export const ALL_PERSONS = gql`
  query {
    // ...
  }
`
export const FIND_PERSON = gql`
  query findPersonByName($nameToSearch: String!) {
    // ...
  }
`

export const CREATE_PERSON = gql`
  mutation createPerson($name: String!, $street: String!, $city: String!, $phone: String)
{
    // ...
  }
`
```

Each component then imports the queries it needs:

```
import { ALL_PERSONS } from './queries'

const App = () => {
  const result = useQuery(ALL_PERSONS)
```

```
  // ...
}
```

The current code of the application can be found on GitHub branch *part8-2*.

Handling mutation errors

Trying to create a person with invalid data causes an error:

We should handle the exception. We can register an error handler function to the mutation using the `useMutation` hook's `onError` option.

Let's register the mutation with an error handler that uses the `setError` * function it receives as a parameter to set an error message:

```
const PersonForm = ({ setError }) => {
  // ...

  const [ createPerson ] = useMutation(CREATE_PERSON, {
    refetchQueries: [  {query: ALL_PERSONS } ],
    onError: (error) => {
      const messages = error.graphQLErrors.map(e => e.message).join('\n')
      setError(messages)
    }
  })

  // ...
}
```

We can then render the error message on the screen as necessary:

```
const App = () => {
  const [errorMessage, setErrorMessage] = useState(null)

  const result = useQuery(ALL_PERSONS)

  if (result.loading)  {
    return <div>loading...</div>
  }

  const notify = (message) => {
    setErrorMessage(message)
    setTimeout(() => {
      setErrorMessage(null)
    }, 10000)
  }

  return (
    <div>
      <Notify errorMessage={errorMessage} />
      <Persons persons = {result.data.allPersons} />
      <PersonForm setError={notify} />
    </div>
  )
}

const Notify = ({errorMessage}) => {
  if ( !errorMessage ) {
    return null
  }
  return (
    <div style={{color: 'red'}}>
    {errorMessage}
    </div>
  )
}
```

Now the user is informed about an error with a simple notification.

The current code of the application can be found on GitHub branch *part8-3*.

Updating a phone number

Let's add the possibility to change the phone numbers of persons to our application. The solution is almost identical to the one we used for adding new persons.

Again, the mutation requires parameters.

```
export const EDIT_NUMBER = gql`
  mutation editNumber($name: String!, $phone: String!) {
    editNumber(name: $name, phone: $phone) {
      name
      phone
      address {
        street
        city
      }
      id
    }
  }
`
```

The *PhoneForm* component responsible for the change is straightforward. The form has fields for the person's name and new phone number, and calls the `changeNumber` function. The function is done using the `useMutation` hook. Interesting lines on the code have been highlighted.

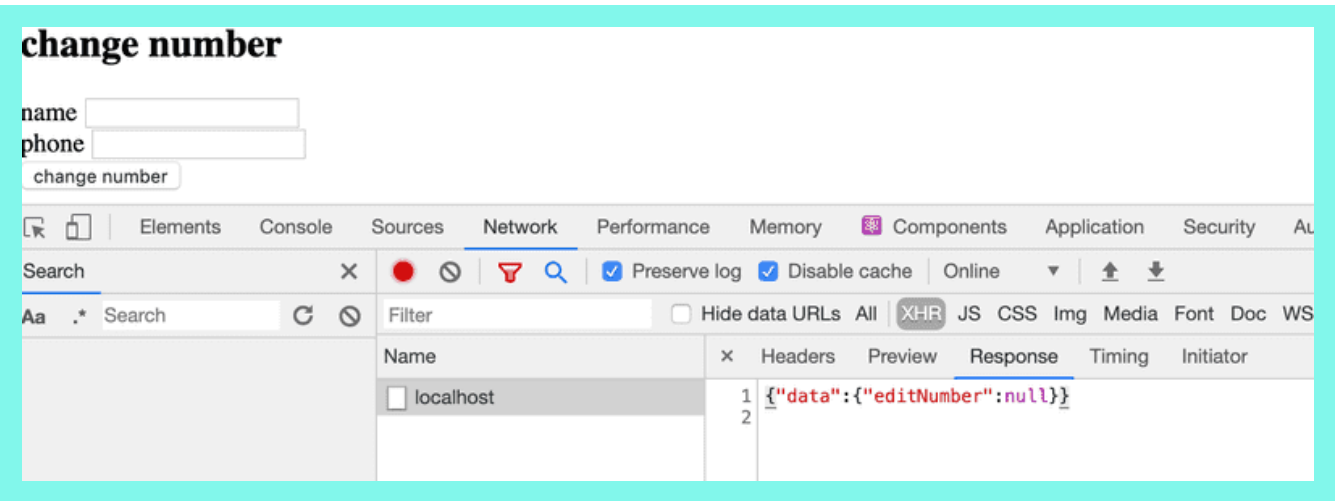
```
import { useState } from 'react'
import { useMutation } from '@apollo/client'

import { EDIT_NUMBER } from '../queries'
```

```
const PhoneForm = () => {
  const [name, setName] = useState('')
  const [phone, setPhone] = useState('')

  const [ changeNumber ] = useMutation(EDIT_NUMBER)

  const submit = (event) => {
    event.preventDefault()

    changeNumber({ variables: { name, phone } })

    setName('')
    setPhone('')
  }

  return (
    <div>
      <h2>change number</h2>

      <form onSubmit={submit}>
        <div>
          name <input
            value={name}
            onChange={({ target }) => setName(target.value)}
          />
        </div>
        <div>
          phone <input
            value={phone}
            onChange={({ target }) => setPhone(target.value)}
          />
        </div>
        <button type='submit'>change number</button>
      </form>
    </div>
  )
}

export default PhoneForm
```

It looks bleak, but it works:

Surprisingly, when a person's number is changed, the new number automatically appears on the list of persons rendered by the *Persons* component. This happens because each person has an identifying field of type *ID*, so the person's details saved to the cache update automatically when they are changed with the mutation.

Our application still has one small flaw. If we try to change the phone number for a name which does not exist, nothing seems to happen. This happens because if a person with the given name cannot be found, the mutation response is *null*:



For GraphQL, this is not an error, so registering an `onError` error handler is not useful.

We can use the `result` field returned by the `useMutation` hook as its second parameter to generate an error message.

```
const PhoneForm = ({ setError }) => {
  const [name, setName] = useState('')
  const [phone, setPhone] = useState('')

  const [ changeNumber, result ] = useMutation(EDIT_NUMBER)

  const submit = (event) => {
    // ...
  }

  useEffect(() => {
    if (result.data && result.data.editNumber === null) {
      setError('person not found')
    }
  }, [result.data])

  // ...
}
```

If a person cannot be found, or the `result.data.editNumber` is `null`, the component uses the callback function it received as props to set a suitable error message. We want to set the error message only when the result of the mutation `result.data` changes, so we use the useEffect hook to control setting the error message.

The current code of the application can be found on GitHub branch *part8-4*.

Apollo Client and the applications state

In our example, management of the applications state has mostly become the responsibility of Apollo Client. This is a quite typical solution for GraphQL applications. Our example uses the state of the React components only to manage the state of a form and to show error notifications. As a result, it could be that there are no justifiable reasons to use Redux to manage application state when using GraphQL.

When necessary, Apollo enables saving the application's local state to Apollo cache.

Exercises 8.8.-8.12

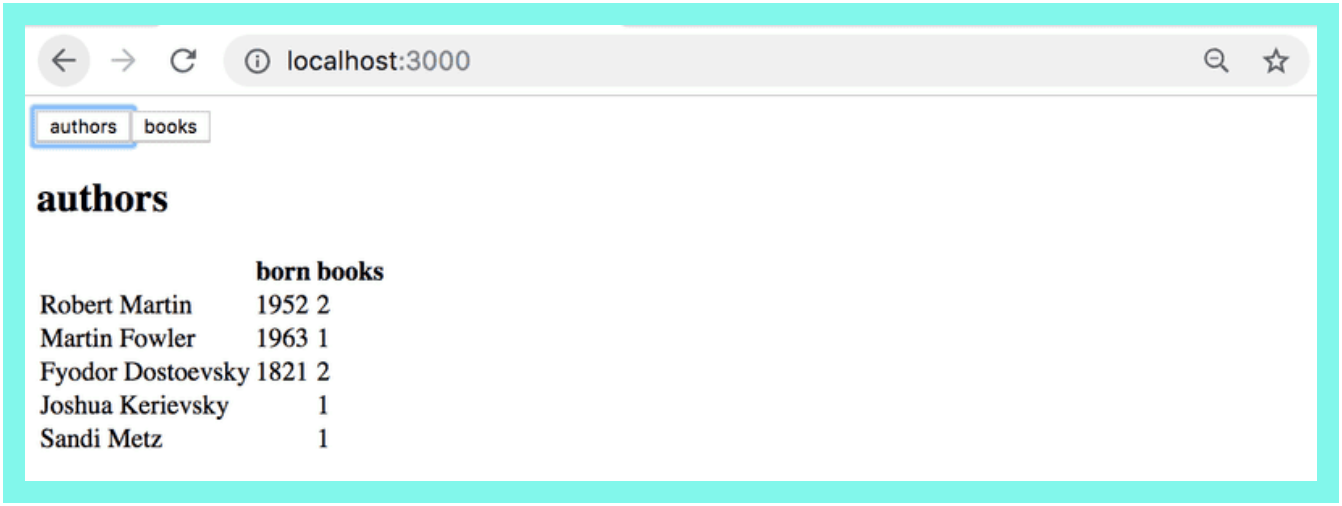
Through these exercises, we'll implement a frontend for the GraphQL library.

Take this project as a start for your application.

Note if you want, you can also use React router to implement the application's navigation!

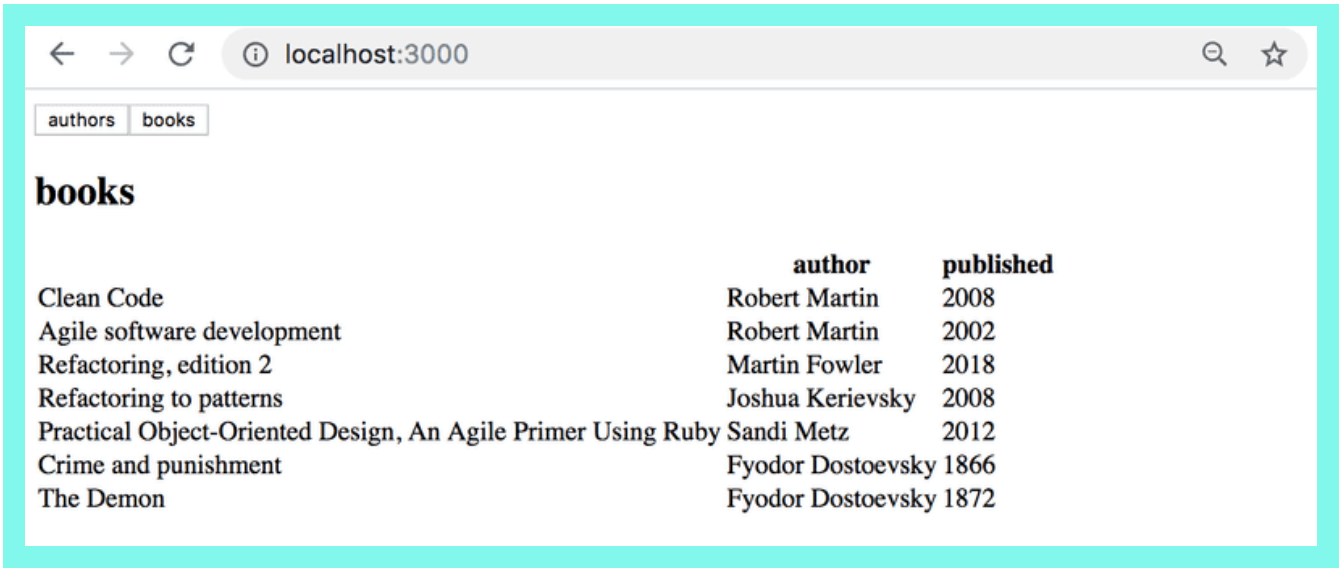
8.8: Authors view

Implement an Authors view to show the details of all authors on a page as follows:



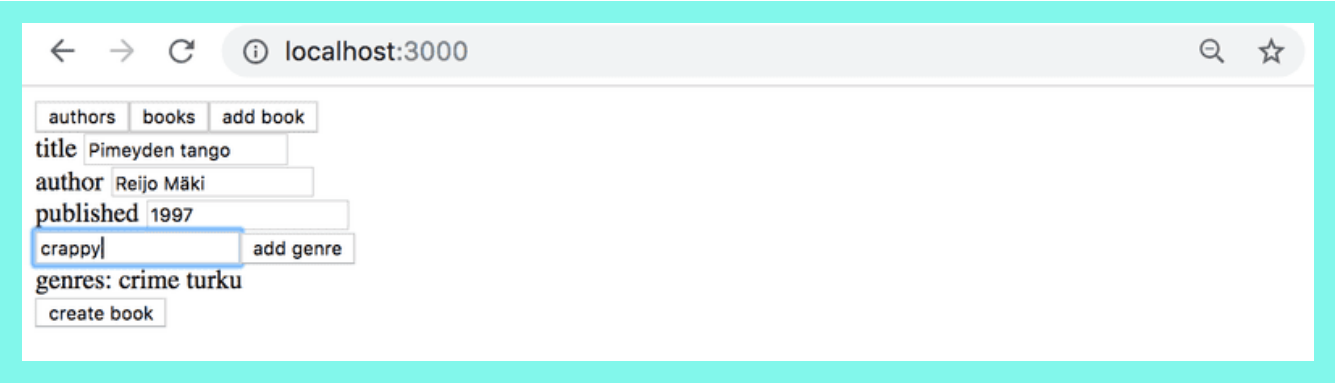
8.9: Books view

Implement a Books view to show on a page all other details of all books except their genres.



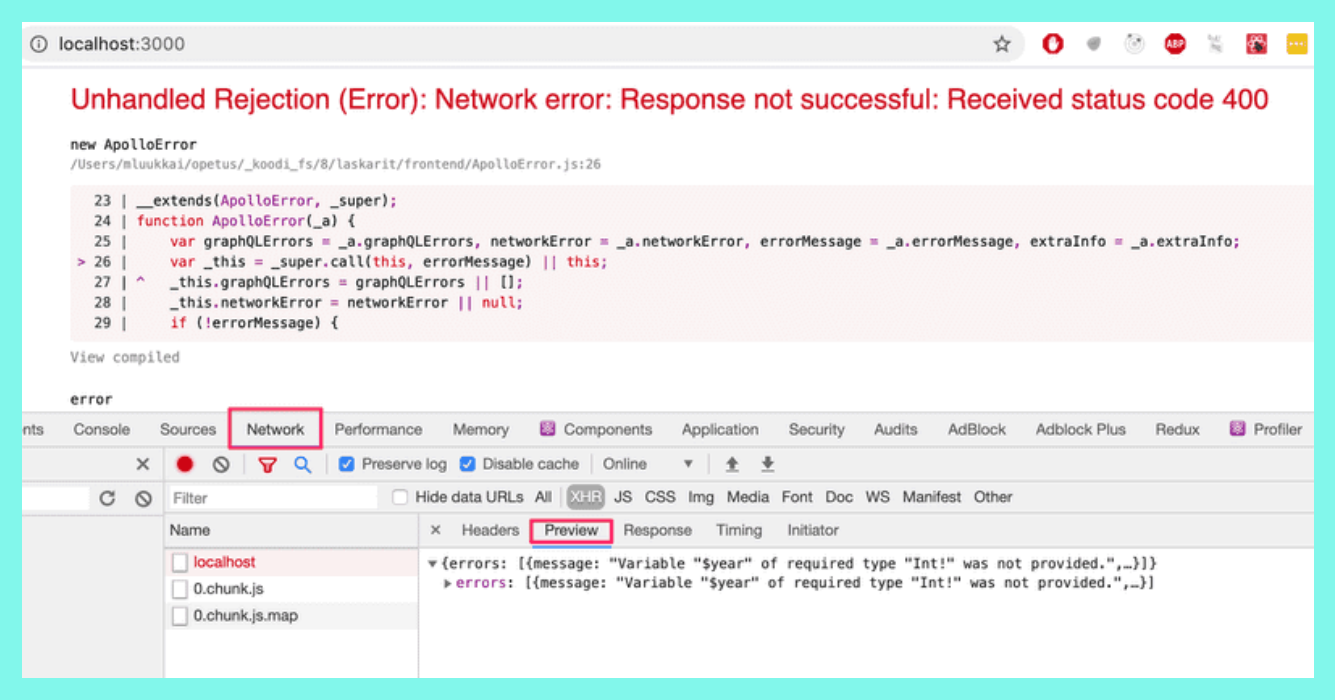
8.10: Adding a book

Implement a possibility to add new books to your application. The functionality can look like this:



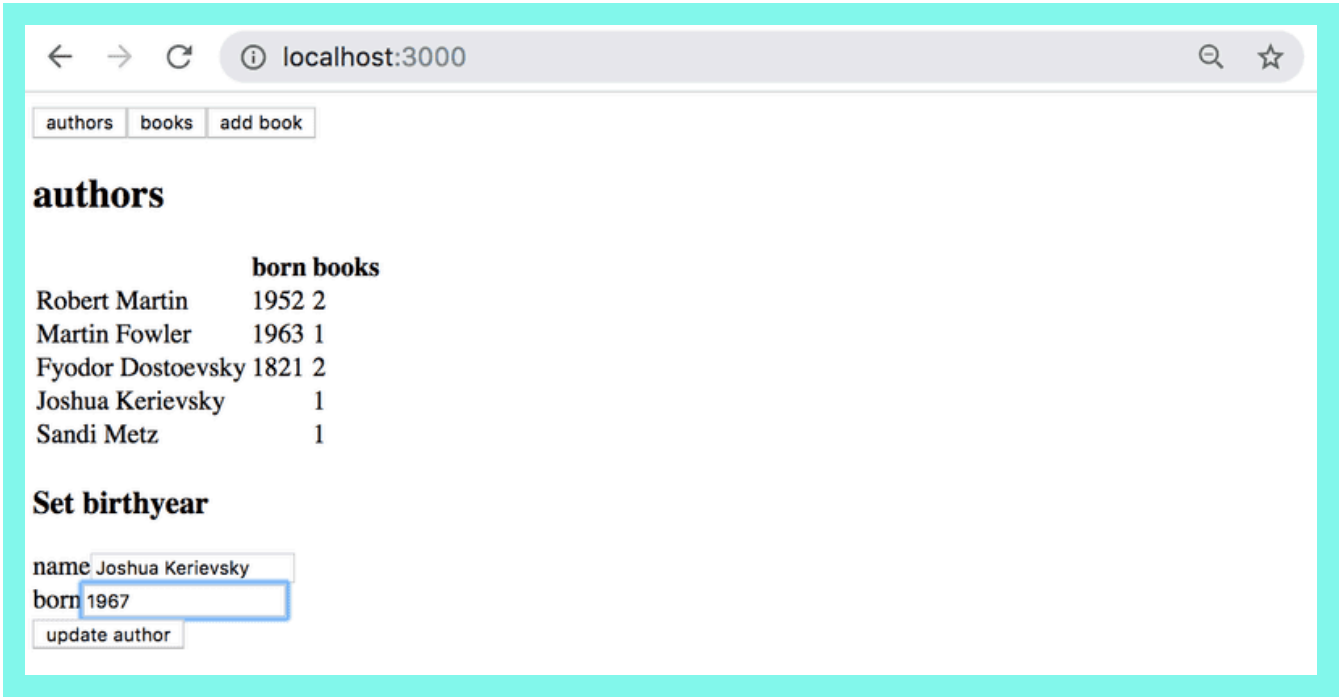
Make sure that the Authors and Books views are kept up to date after a new book is added.

In case of problems when making queries or mutations, check from the developer console what the server response is:



8.11: Authors birth year

Implement a possibility to set authors birth year. You can create a new view for setting the birth year, or place it on the Authors view:

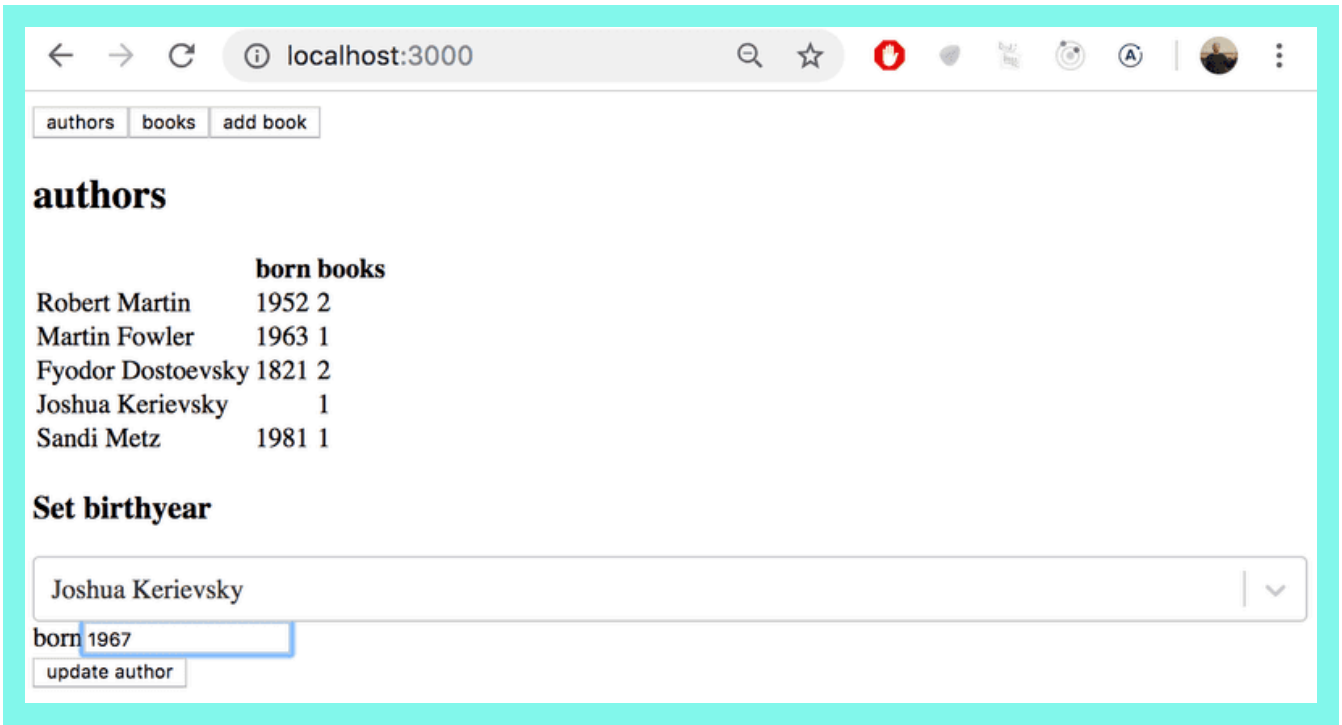


Make sure that the Authors view is kept up to date after setting a birth year.

8.12: Authors birth year advanced

Change the birth year form so that a birth year can be set only for an existing author. Use select tag, react select, or some other mechanism.

A solution using the react select library looks as follows:



Propose changes to material

Part 8a
Previous part

Part 8c
Next part

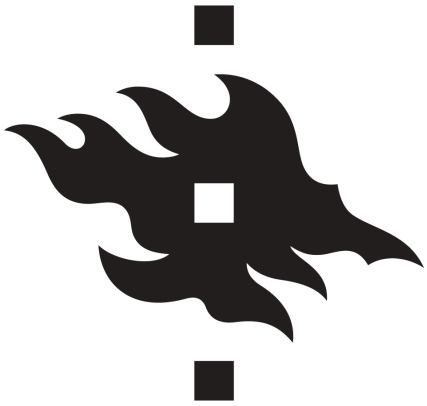
About course

Course contents

FAQ

Partners

Challenge



UNIVERSITY OF HELSINKI


HOUSTON

{() => fs}

Fullstack > Part 8 > Database and user administration

c Database and user administration

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

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,
    minlength: 5
  },
  phone: {
    type: String,
    minlength: 5
  },
  street: {
```

copy

```
    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:

copy

```
// ...
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 = gql`
  ...
`

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

copy

```
    }

    return Person.find({ phone: { $exists: args.phone === 'YES' } })
  },
},
```

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 a 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
        }
      })
    }

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

    try {
```

copy

```
        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: [
    {
      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
  }
}
```

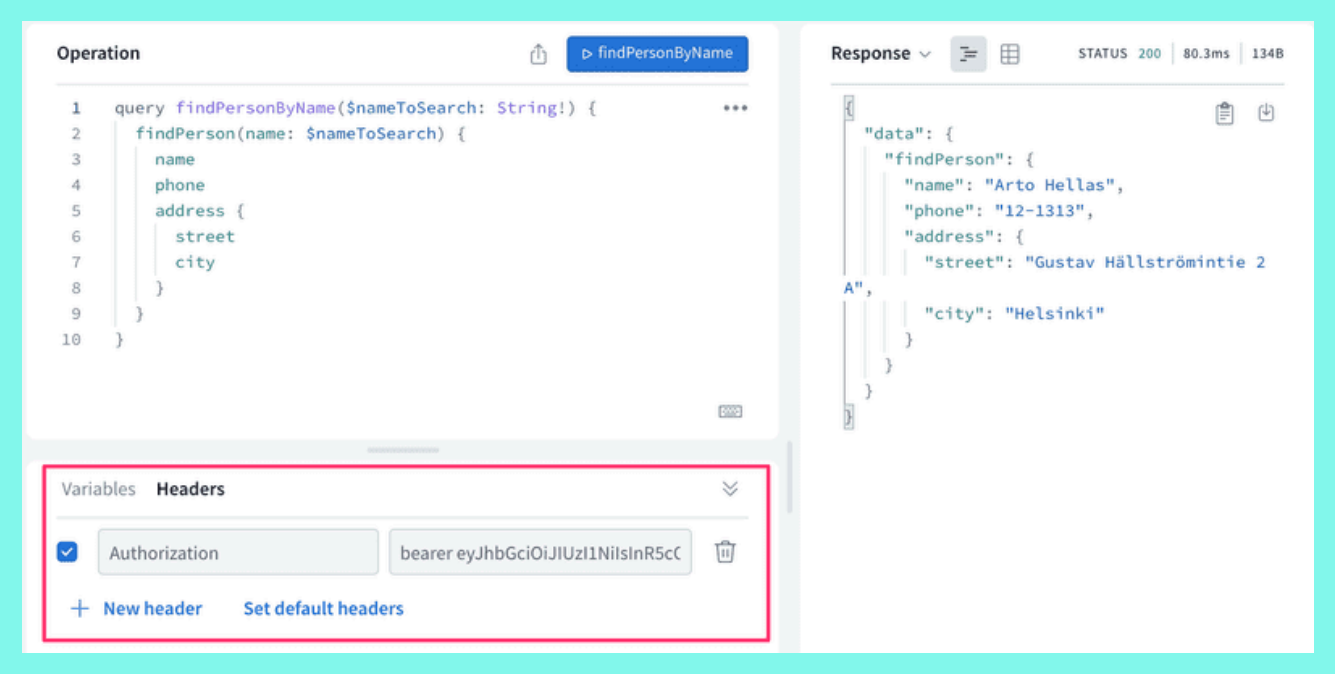
The mutation for logging in looks like this:

```
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

```
startStandaloneServer(server, {
  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}`)
})
```

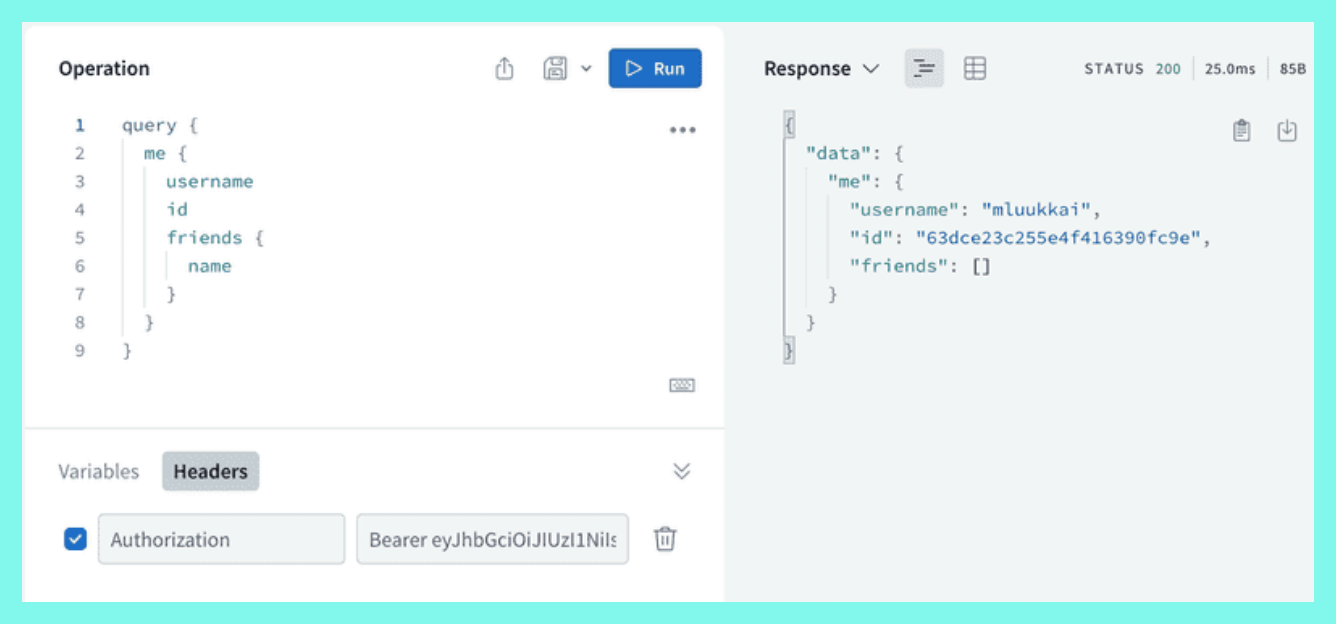
The object returned by context is given to all resolvers as their *third parameter*. Context is the right place to do things which are shared by multiple resolvers, like user identification.

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
  }
},
```

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

      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
}
```

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
```

copy

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

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

copy

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

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

copy

The code of the backend can be found on Github branch *part8-5*.

Exercises 8.13.-8.16

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

About course

Course contents

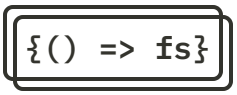
FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON



d Login and updating the cache

The frontend of our application shows the phone directory just fine with the updated server. However, if we want to add new persons, we have to add login functionality to the frontend.

User login

Let's add the variable `token` to the application's state. When a user is logged in, it will contain a user token. If `token` is undefined, we render the *LoginForm* component responsible for user login. The component receives an error handler and the `setToken` function as parameters:

```
const App = () => {
  const [token, setToken] = useState(null)

  // ...

  if (!token) {
    return (
      <div>
        <Notify errorMessage={errorMessage} />
        <h2>Login</h2>
        <LoginForm
          setToken={setToken}
          setError={notify}
        />
      </div>
    )
  }

  return (
```

```
    // ...
  )
}
```

Next, we define a mutation for logging in:

```
export const LOGIN = gql`
  mutation login($username: String!, $password: String!) {
    login(username: $username, password: $password)  {
      value
    }
  }
`
```

The `LoginForm` component works pretty much just like all the other components doing mutations that we have previously created. Interesting lines in the code have been highlighted:

```
import { useState, useEffect } from 'react'
import { useMutation } from '@apollo/client'
import { LOGIN } from '../queries'

const LoginForm = ({ setError, setToken }) => {
  const [username, setUsername] = useState('')
  const [password, setPassword] = useState('')

  const [ login, result ] = useMutation(LOGIN, {
    onError: (error) => {
      setError(error.graphQLErrors[0].message)
    }
  })

  useEffect(() => {
    if ( result.data ) {
      const token = result.data.login.value
      setToken(token)
      localStorage.setItem('phonenumbers-user-token', token)
    }
  }, [result.data])

  const submit = async (event) => {
    event.preventDefault()

    login({ variables: { username, password } })
  }

  return (
    <div>
      <form onSubmit={submit}>
        <div>
          username <input
```

```
            value={username}
            onChange={({ target }) => setUsername(target.value)}
          />
        </div>
        <div>
          password <input
            type='password'
            value={password}
            onChange={({ target }) => setPassword(target.value)}
          />
        </div>
        <button type='submit'>login</button>
      </form>
    </div>
  )
}

export default LoginForm
```

We are using an effect hook to save the token's value to the state of the `App` component and the local storage after the server has responded to the mutation. Use of the effect hook is necessary to avoid an endless rendering loop.

Let's also add a button which enables a logged-in user to log out. The button's onClick handler sets the `token` state to null, removes the token from local storage and resets the cache of the Apollo client. The last step is important, because some queries might have fetched data to cache, which only logged-in users should have access to.

We can reset the cache using the resetStore method of an Apollo `client` object. The client can be accessed with the useApolloClient hook:

```
const App = () => {
  const [token, setToken] = useState(null)
  const [errorMessage, setErrorMessage] = useState(null)
  const result = useQuery(ALL_PERSONS)
  const client = useApolloClient()

  if (result.loading)  {
    return <div>loading...</div>
  }

  const logout = () => {
    setToken(null)
    localStorage.clear()
    client.resetStore()
  }

  if (!token) {
    return (
      <>
        <Notify errorMessage={errorMessage} />
        <LoginForm setToken={setToken} setError={notify} />
```

```
      </>
    )
  }

  return (
    <>
      <Notify errorMessage={errorMessage} />
      <button onClick={logout}>logout</button>
      <Persons persons={result.data.allPersons} />
      <PersonForm setError={notify} />
      <PhoneForm setError={notify} />
    </>
  )
}
```

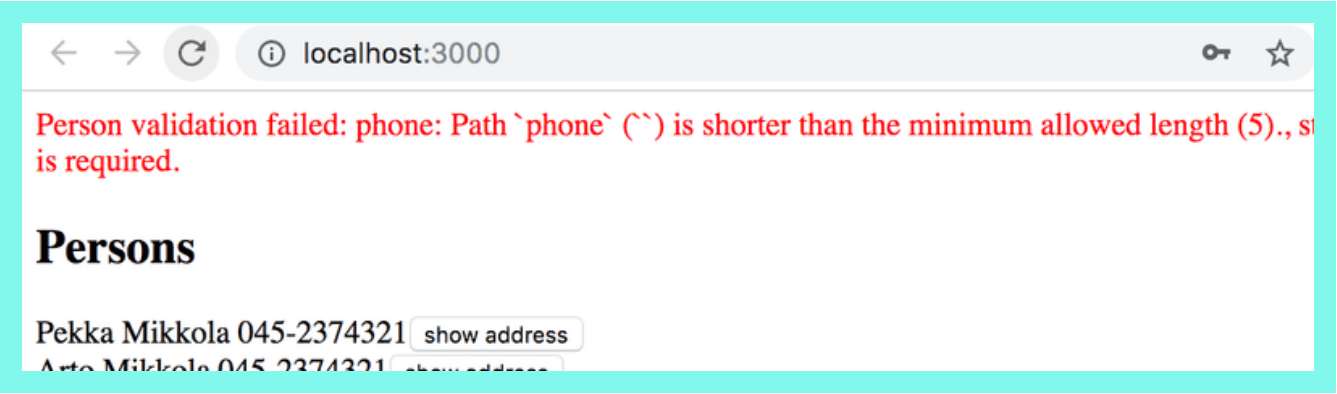
Adding a token to a header

After the backend changes, creating new persons requires that a valid user token is sent with the request. In order to send the token, we have to change the way we define the `ApolloClient` object in *index.js* a little.

```
import { ApolloClient, InMemoryCache, ApolloProvider, createHttpLink } from
'@apollo/client'
import { setContext } from '@apollo/client/link/context'

const authLink = setContext((_, { headers }) => {
  const token = localStorage.getItem('phonenumbers-user-token')
  return {
    headers: {
      ...headers,
      authorization: token ? `Bearer ${token}` : null,
    }
  }
})

const httpLink = createHttpLink({
  uri: 'http://localhost:4000',
})

const client = new ApolloClient({
  cache: new InMemoryCache(),
  link: authLink.concat(httpLink)
})
```

The field `uri` that was previously used when creating the `client` object has been replaced by the field `link`, which defines in a more complicated case how Apollo is connected to the server. The server url is now wrapped using the function createHttpLink into a suitable httpLink object. The link is modified by the context defined by the authLink object so that a possible token in localStorage is set to header *authorization* for each request to the server.

Creating new persons and changing numbers works again. There is however one remaining problem. If we try to add a person without a phone number, it is not possible.



Validation fails, because frontend sends an empty string as the value of `phone`.

Let's change the function creating new persons so that it sets `phone` to `undefined` if user has not given a value.

```
const PersonForm = ({ setError }) => {
  // ...
  const submit = async (event) => {
    event.preventDefault()
    createPerson({
      variables: {
        name, street, city,
        phone: phone.length > 0 ? phone : undefined
      }
    })

  // ...
  }

  // ...
}
```

Updating cache, revisited

We have to update the cache of the Apollo client on creating new persons. We can update it using the mutation's `refetchQueries` option to define that the `ALL_PERSONS` query is done again.

```
const PersonForm = ({ setError }) => {
  // ...

  const [ createPerson ] = useMutation(CREATE_PERSON, {
    refetchQueries: [  {query: ALL_PERSONS} ],
    onError: (error) => {
      const messages = error.graphQLErrors.map(e => e.message).join('\n')
```

```
      setError(messages)
    }
  })
```

This approach is pretty good, the drawback being that the query is always rerun with any updates.

It is possible to optimize the solution by handling updating the cache ourselves. This is done by defining a suitable `update` callback for the mutation, which Apollo runs after the mutation:

```
const PersonForm = ({ setError }) => {
  // ...

  const [ createPerson ] = useMutation(CREATE_PERSON, {
    onError: (error) => {
      const messages = error.graphQLErrors.map(e => e.message).join('\n')
      setError(messages)
    },
    update: (cache, response) => {
      cache.updateQuery({ query: ALL_PERSONS }, ({ allPersons }) => {
        return {
          allPersons: allPersons.concat(response.data.addPerson),
        }
      })
    },
  })

  // ..
}
```

The callback function is given a reference to the cache and the data returned by the mutation as parameters. For example, in our case, this would be the created person.

Using the function `updateQuery` the code updates the query ALLPERSONS in the cache by adding the new person to the cached data.

In some situations, the only sensible way to keep the cache up to date is using the `update` callback.

When necessary, it is possible to disable cache for the whole application or single queries by setting the field managing the use of cache, `fetchPolicy` as `no-cache`.

Be diligent with the cache. Old data in the cache can cause hard-to-find bugs. As we know, keeping the cache up to date is very challenging. According to a coder proverb:

> *There are only two hard things in Computer Science: cache invalidation and naming things.* Read more here.

The current code of the application can be found on Github, branch *part8-5*.

Exercises 8.17.-8.22

8.17 Listing books

After the backend changes, the list of books does not work anymore. Fix it.

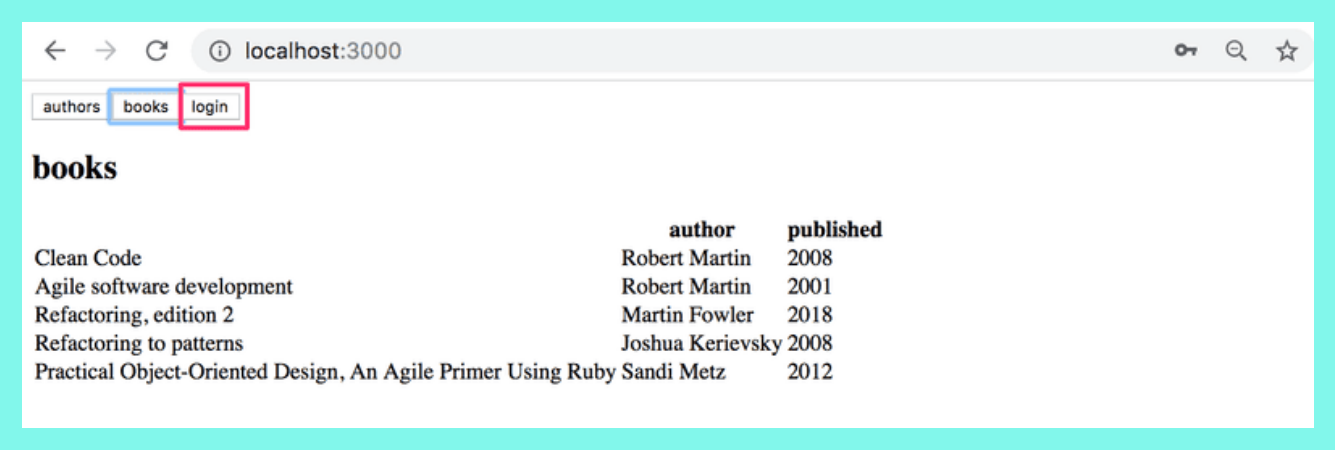
8.18 Log in

Adding new books and changing the birth year of an author do not work because they require a user to be logged in.

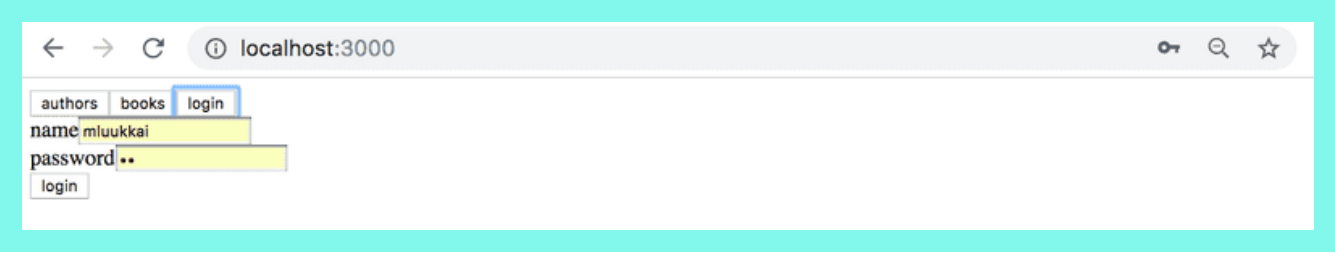
Implement login functionality and fix the mutations.

It is not necessary yet to handle validation errors.

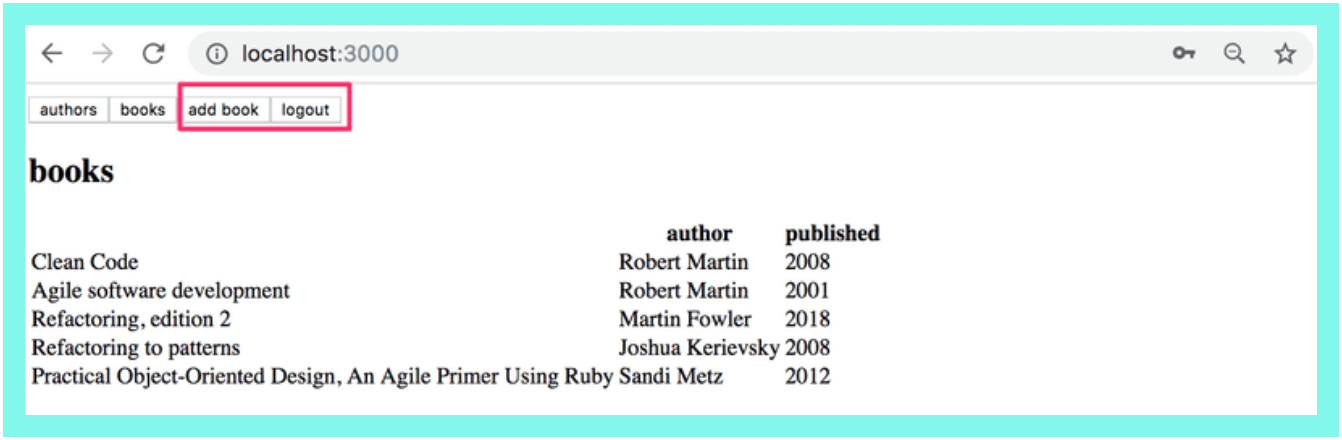
You can decide how the login looks on the user interface. One possible solution is to make the login form into a separate view which can be accessed through a navigation menu:



The login form:



When a user is logged in, the navigation changes to show the functionalities which can only be done by a logged-in user:



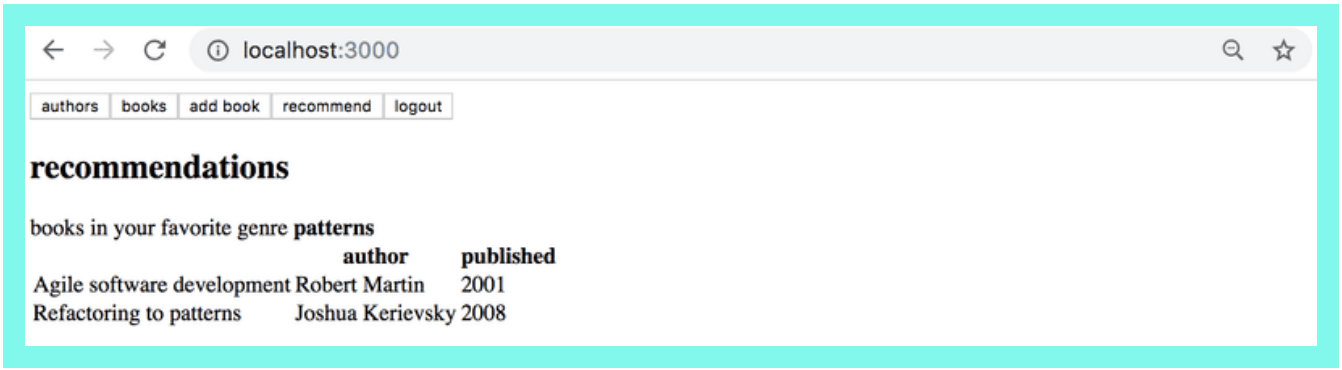
8.19 Books by genre, part 1

Complete your application to filter the book list by genre. Your solution might look something like this:

In this exercise, the filtering can be done using just React.

8.20 Books by genre, part 2

Implement a view which shows all the books based on the logged-in user's favourite genre.



8.21 books by genre with GraphQL

In the previous two exercises, the filtering could have been done using just React. To complete this exercise, you should redo the filtering the books based on a selected genre (that was done in exercise

8.19) using a GraphQL query to the server. If you already did so then you do not have to do anything.

This and the next exercises are quite **challenging** like it should be this late in the course. You might want to complete first the easier ones in the next part.

8.22 Up-to-date cache and book recommendations

If you did the previous exercise, that is, fetch the books in a genre with GraphQL, ensure somehow that the books view is kept up to date. So when a new book is added, the books view is updated **at least** when a genre selection button is pressed.

When new genre selection is not done, the view does not have to be updated.

Propose changes to material

Part 8c
Previous part

Part 8e
Next part

About course

Course contents

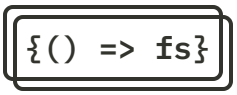
FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON



e Fragments and subscriptions

We are approaching the end of this part. Let's finish by having a look at a few more details about GraphQL.

Fragments

It is pretty common in GraphQL that multiple queries return similar results. For example, the query for the details of a person

```
query {
  findPerson(name: "Pekka Mikkola") {
    name
    phone
    address{
      street
      city
    }
  }
}
```

and the query for all persons

```
query {
  allPersons {
    name
    phone
```

```
    address{
      street
      city
    }
  }
}
```

both return persons. When choosing the fields to return, both queries have to define exactly the same fields.

These kinds of situations can be simplified with the use of fragments. Let's declare a fragment for selecting all fields of a person:

```
fragment PersonDetails on Person {
  name
  phone
  address {
    street
    city
  }
}
```

With the fragment, we can do the queries in a compact form:

```
query {
  allPersons {
    ...PersonDetails
  }
}

query {
  findPerson(name: "Pekka Mikkola") {
    ...PersonDetails
  }
}
```

The fragments ***are not*** defined in the GraphQL schema, but in the client. The fragments must be declared when the client uses them for queries.

In principle, we could declare the fragment with each query like so:

```
export const FIND_PERSON = gql`
  query findPersonByName($nameToSearch: String!) {
    findPerson(name: $nameToSearch) {
      ...PersonDetails
    }
  }
```

```
  fragment PersonDetails on Person {
    name
    phone
    address {
      street
      city
    }
  }
`
```

However, it is much better to declare the fragment once and save it to a variable.

```
const PERSON_DETAILS = gql`
  fragment PersonDetails on Person {
    id
    name
    phone
    address {
      street
      city
    }
  }
`
```

Declared like this, the fragment can be placed to any query or mutation using a dollar sign and curly braces:

```
export const FIND_PERSON = gql`
  query findPersonByName($nameToSearch: String!) {
    findPerson(name: $nameToSearch) {
      ...PersonDetails
    }
  }
  ${PERSON_DETAILS}
`
```

Subscriptions

Along with query and mutation types, GraphQL offers a third operation type: subscriptions. With subscriptions, clients can *subscribe* to updates about changes in the server.

Subscriptions are radically different from anything we have seen in this course so far. Until now, all interaction between browser and server was due to a React application in the browser making HTTP requests to the server. GraphQL queries and mutations have also been done this way. With subscriptions, the situation is the opposite. After an application has made a subscription, it starts to listen to the server. When changes occur on the server, it sends a notification to all of its *subscribers*.

Technically speaking, the HTTP protocol is not well-suited for communication from the server to the browser. So, under the hood, Apollo uses WebSockets for server subscriber communication.

Refactoring the backend

Since version 3.0 Apollo Server does not support subscriptions out of the box, we need to do some changes before we set up subscriptions. Let us also clean the app structure a bit.

Let's start by extracting the schema definition to the file *schema.js*

```
const typeDefs = `
  type User {
    username: String!
    friends: [Person!]!
    id: ID!
  }

  type Token {
    value: String!
  }

  type Address {
    street: String!
    city: String!
  }

  type Person {
    name: String!
    phone: String
    address: Address!
    id: ID!
  }

  enum YesNo {
    YES
    NO
  }

  type Query {
    personCount: Int!
    allPersons(phone: YesNo): [Person!]!
    findPerson(name: String!): Person
    me: User
  }

  type Mutation {
    addPerson(
      name: String!
      phone: String
      street: String!
      city: String!
```

```
    ): Person
    editNumber(name: String!, phone: String!): Person
    createUser(username: String!): User
    login(username: String!, password: String!): Token
    addAsFriend(name: String!): User
  }
`

module.exports = typeDefs
```

The resolvers definition is moved to the file *resolvers.js*

copy

```
const { GraphQLError } = require('graphql')
const jwt = require('jsonwebtoken')
const Person = require('./models/person')
const User = require('./models/user')

const resolvers = {
  Query: {
    personCount: async () => Person.collection.countDocuments(),
    allPersons: async (root, args, context) => {
      if (!args.phone) {
        return Person.find({})
      }

      return Person.find({ phone: { $exists: args.phone === 'YES'  }})
    },
    findPerson: async (root, args) => Person.findOne({ name: args.name }),
    me: (root, args, context) => {
      return context.currentUser
    }
  },
  Person: {
    address: ({ street, city }) => {
      return {
        street,
        city,
      }
    },
  },
  Mutation: {
    addPerson: async (root, args, context) => {
      const person = new Person({ ...args })
      const currentUser = context.currentUser

      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
  },
  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('Editing number failed', {
        extensions: {
          code: 'BAD_USER_INPUT',
          invalidArgs: args.name,
          error
        }
      })
    }

    return person
  },
  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) }
    },
    addAsFriend: async (root, args, { currentUser }) => {
      const nonFriendAlready = (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 ( nonFriendAlready(person) ) {
        currentUser.friends = currentUser.friends.concat(person)
      }

      await currentUser.save()

      return currentUser
    },
  }
}

module.exports = resolvers
```

So far, we have started the application with the easy-to-use function startStandaloneServer, thanks to which the application has not had to be configured that much:

```
const { startStandaloneServer } = require('@apollo/server/standalone')

// ...

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

startStandaloneServer(server, {
  listen: { port: 4000 },
  context: async ({ req, res }) => {
    /// ...
  },
}).then(({ url }) => {
```

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

Unfortunately, startStandaloneServer does not allow adding subscriptions to the application, so let's switch to the more robust `expressMiddleware` function. As the name of the function already suggests, it is an Express middleware, which means that Express must also be configured for the application, with the GraphQL server acting as middleware.

Let us install Express

```
npm install express cors
```

copy

and the file *index.js* changes to:

```
const { ApolloServer } = require('@apollo/server')
const { expressMiddleware } = require('@apollo/server/express4')
const { ApolloServerPluginDrainHttpServer } =
require('@apollo/server/plugin/drainHttpServer')
const { makeExecutableSchema } = require('@graphql-tools/schema')
const express = require('express')
const cors = require('cors')
const http = require('http')

const jwt = require('jsonwebtoken')

const mongoose = require('mongoose')

const User = require('./models/user')

const typeDefs = require('./schema')
const resolvers = require('./resolvers')

const MONGODB_URI = 'mongodb+srv://databaseurlhere'

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)
  })

// setup is now within a function
const start = async () => {
  const app = express()
  const httpServer = http.createServer(app)
```

copy

```
  const server = new ApolloServer({
    schema: makeExecutableSchema({ typeDefs, resolvers }),
    plugins: [ApolloServerPluginDrainHttpServer({ httpServer })],
  })

  await server.start()

  app.use(
    '/',
    cors(),
    express.json(),
    expressMiddleware(server, {
      context: async ({ req }) => {
        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 }
        }
      },
    }),
  )

  const PORT = 4000

  httpServer.listen(PORT, () =>
    console.log(`Server is now running on http://localhost:${PORT}`)
  )
}

start()
```

There are several changes to the code. ApolloServerPluginDrainHttpServer has now been added to the configuration of the GraphQL server according to the recommendations of the documentation:

> *We highly recommend using this plugin to ensure your server shuts down gracefully.*

The GraphQL server in the `server` variable is now connected to listen to the root of the server, i.e. to the `/` route, using the `expressMiddleware` object. Information about the logged-in user is set in the context using the function we defined earlier. Since it is an Express server, the middlewares express-json and cors are also needed so that the data included in the requests is correctly parsed and so that CORS problems do not appear.

Since the GraphQL server must be started before the Express application can start listening to the specified port, the entire initialization has had to be placed in an *async function*, which allows waiting for the GraphQL server to start.

The backend code can be found on GitHub, branch *part8-6*.

Subscriptions on the server

Let's implement subscriptions for subscribing for notifications about new persons added.

The schema changes like so:

```
type Subscription {
  personAdded: Person!
}
```

copy

So when a new person is added, all of its details are sent to all subscribers.

First, we have to install two packages for adding subscriptions to GraphQL and a Node.js WebSocket library:

```
npm install graphql-ws ws @graphql-tools/schema
```

copy

The file *index.js* is changed to:

```
const { WebSocketServer } = require('ws')
const { useServer } = require('graphql-ws/lib/use/ws')

// ...

const start = async () => {
  const app = express()
  const httpServer = http.createServer(app)

  const wsServer = new WebSocketServer({
    server: httpServer,
    path: '/',
  })

  const schema = makeExecutableSchema({ typeDefs, resolvers })
  const serverCleanup = useServer({ schema }, wsServer)

  const server = new ApolloServer({
    schema,
    plugins: [
      ApolloServerPluginDrainHttpServer({ httpServer }),
      {
        async serverWillStart() {
          return {
            async drainServer() {
              await serverCleanup.dispose();
            },
          };
```

copy

```
        },
      },
    ],
  })

  await server.start()

  app.use(
    '/',
    cors(),
    express.json(),
    expressMiddleware(server, {
      context: async ({ req }) => {
        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 }
        }
      },
    }),
  )

  const PORT = 4000

  httpServer.listen(PORT, () =>
    console.log(`Server is now running on http://localhost:${PORT}`)
  )
}

start()
```

When queries and mutations are used, GraphQL uses the HTTP protocol in the communication. In case of subscriptions, the communication between client and server happens with WebSockets.

The above code registers a WebSocketServer object to listen the WebSocket connections, besides the usual HTTP connections that the server listens to. The second part of the definition registers a function that closes the WebSocket connection on server shutdown. If you're interested in more details about configurations, Apollo's documentation explains in relative detail what each line of code does.

WebSockets are a perfect match for communication in the case of GraphQL subscriptions since when WebSockets are used, also the server can initiate the communication.

The subscription `personAdded` needs a resolver. The `addPerson` resolver also has to be modified so that it sends a notification to subscribers.

The required changes are as follows:

```
const { PubSub } = require('graphql-subscriptions')
const pubsub = new PubSub()
```

copy

```
// ...

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

      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
          }
        })
      }

      pubsub.publish('PERSON_ADDED', { personAdded: person })

      return person
    },
  },
  Subscription: {
    personAdded: {
      subscribe: () => pubsub.asyncIterator('PERSON_ADDED')
    },
  },
}
```

The following library needs to be installed:

```
npm install graphql-subscriptions
```

copy

With subscriptions, the communication happens using the publish-subscribe principle utilizing the object PubSub.

There are only a few lines of code added, but quite a lot is happening under the hood. The resolver of the `personAdded` subscription registers and saves info about all the clients that do the subscription. The clients are saved to an "iterator object" called *PERSON_ADDED* thanks to the following code:

```
Subscription: {
  personAdded: {
    subscribe: () => pubsub.asyncIterator('PERSON_ADDED')
  },
},
```

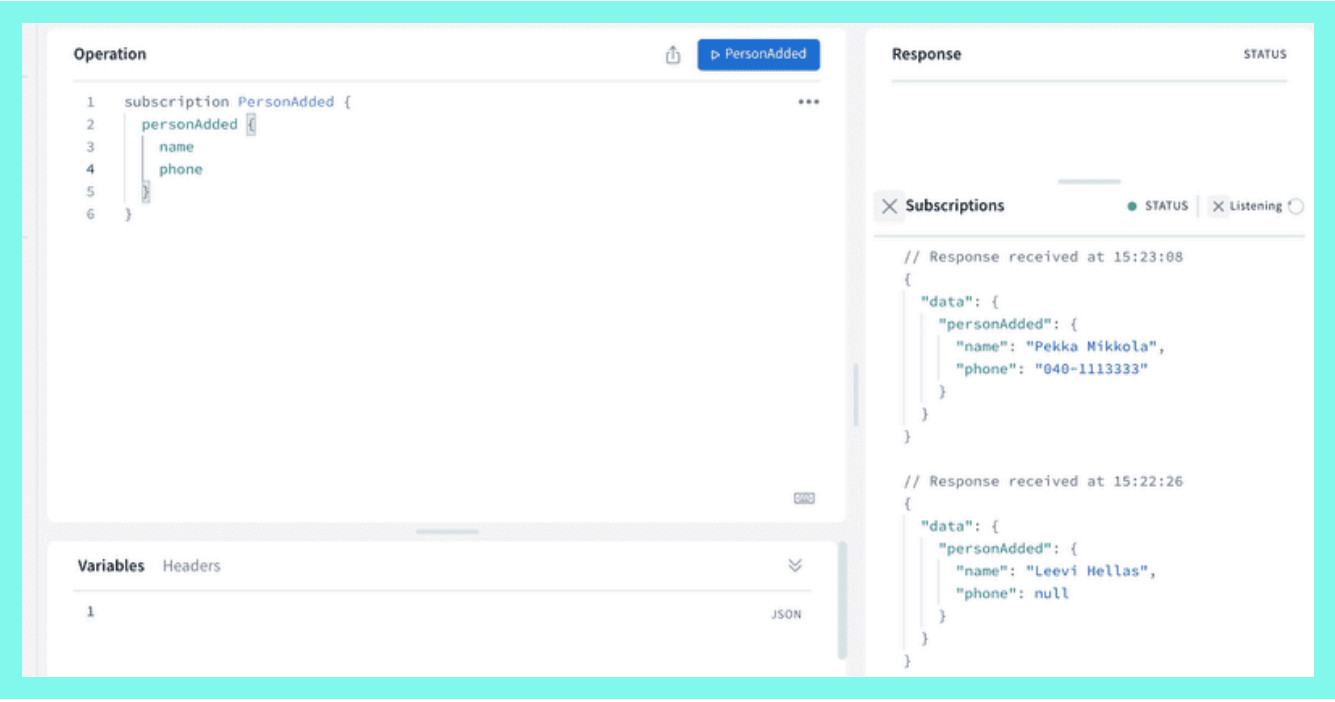
The iterator name is an arbitrary string, but to follow the convention, it is the subscription name written in capital letters.

Adding a new person *publishes* a notification about the operation to all subscribers with PubSub's method `publish`:

```
pubsub.publish('PERSON_ADDED', { personAdded: person })
```

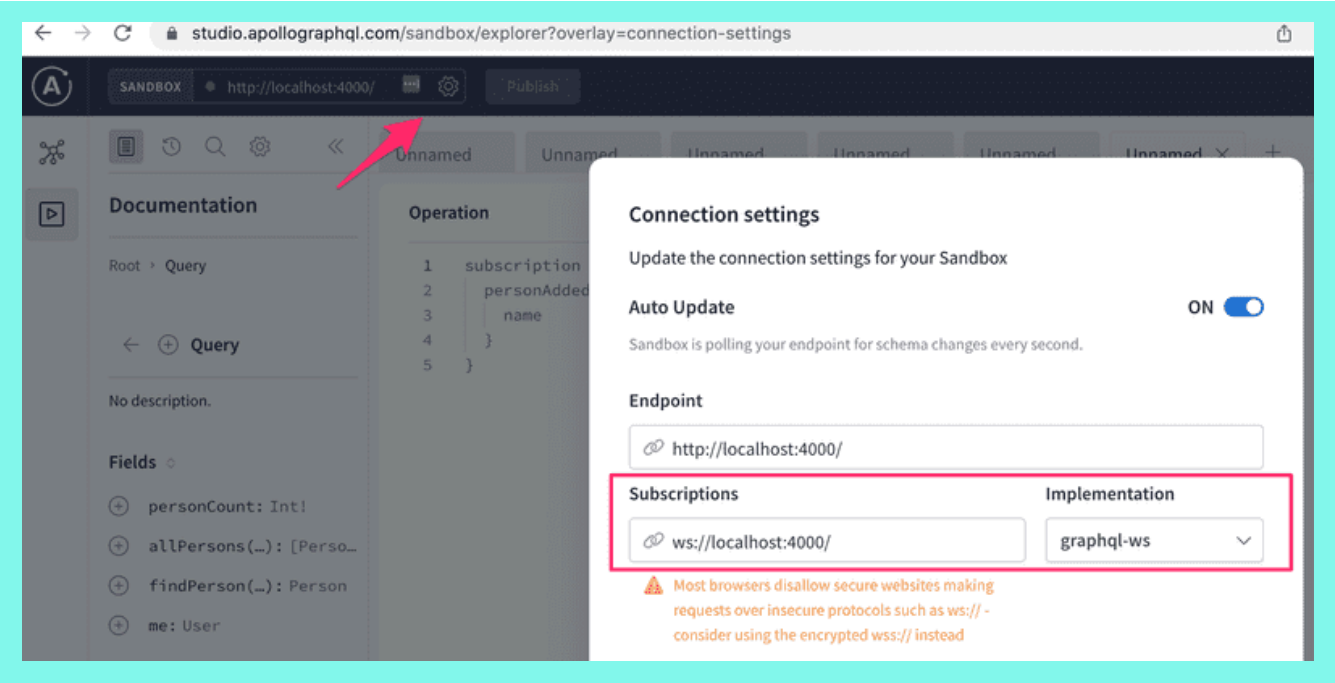
Execution of this line sends a WebSocket message about the added person to all the clients registered in the iterator *PERSON_ADDED*.

It's possible to test the subscriptions with the Apollo Explorer like this:



When the blue button *PersonAdded* is pressed, Explorer starts to wait for a new person to be added. On addition (that you need to do from another browser window), the info of the added person appears on the right side of the Explorer.

If the subscription does not work, check that you have the correct connection settings:



The backend code can be found on GitHub, branch *part8-7*.

Implementing subscriptions involves a lot of configurations. You will be able to cope with the few exercises of this course without worrying much about the details. If you are planning to use subscriptions in an production use application, you should definitely read Apollo's documentation on subscriptions carefully.

Subscriptions on the client

In order to use subscriptions in our React application, we have to do some changes, especially to its configuration. The configuration in *main.jsx* has to be modified like so:

```
import {
  ApolloClient, InMemoryCache, ApolloProvider, createHttpLink,
  split
} from '@apollo/client'
import { setContext } from '@apollo/client/link/context'

import { getMainDefinition } from '@apollo/client/utilities'
import { GraphQLWsLink } from '@apollo/client/link/subscriptions'
import { createClient } from 'graphql-ws'

const authLink = setContext((_, { headers }) => {
  const token = localStorage.getItem('phonenumbers-user-token')
  return {
    headers: {
      ...headers,
      authorization: token ? `Bearer ${token}` : null,
    }
```

```
  }
})

const httpLink = createHttpLink({ uri: 'http://localhost:4000' })

const wsLink = new GraphQLWsLink(
  createClient({ url: 'ws://localhost:4000' })
)

const splitLink = split(
  ({ query }) => {
    const definition = getMainDefinition(query)
    return (
      definition.kind === 'OperationDefinition' &&
      definition.operation === 'subscription'
    )
  },
  wsLink,
  authLink.concat(httpLink)
)

const client = new ApolloClient({
  cache: new InMemoryCache(),
  link: splitLink
})

ReactDOM.createRoot(document.getElementById('root')).render(
  <ApolloProvider client={client}>
    <App />
  </ApolloProvider>
)
```

For this to work, we have to install a dependency:

```
npm install graphql-ws
```

copy

The new configuration is due to the fact that the application must have an HTTP connection as well as a WebSocket connection to the GraphQL server.

```
const httpLink = createHttpLink({ uri: 'http://localhost:4000' })

const wsLink = new GraphQLWsLink(
  createClient({
    url: 'ws://localhost:4000',
  })
)
```

copy

The subscriptions are done using the `useSubscription` hook function.

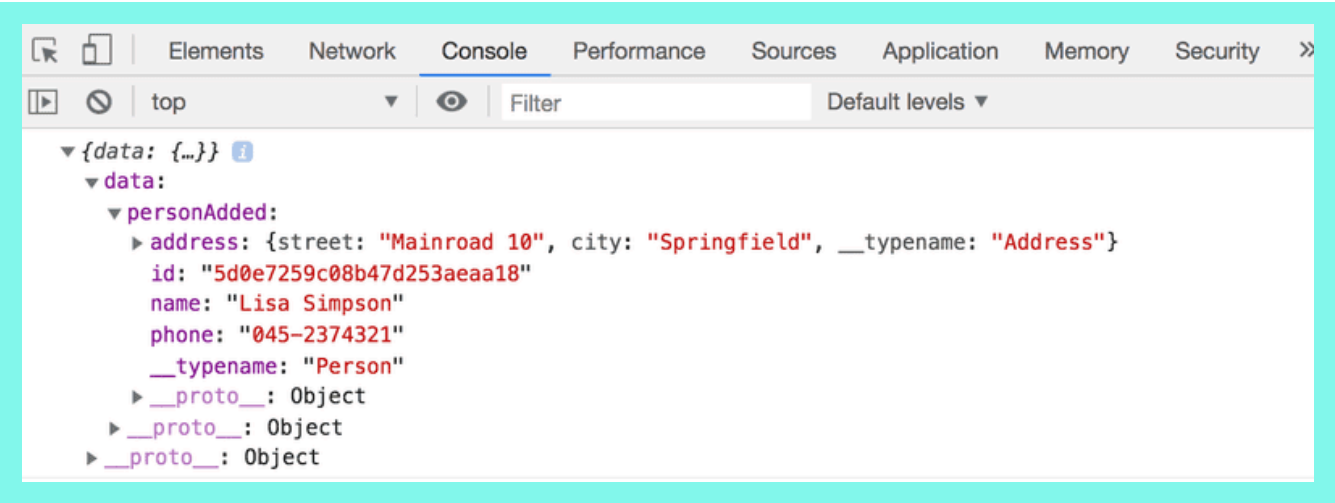
Let's make the following changes to the code. Add the code defining the subscription to the file *queries.js*:

```
export const PERSON_ADDED = gql`
  subscription {
    personAdded {
      ...PersonDetails
    }
  }
  ${PERSON_DETAILS}
`
```

and do the subscription in the App component:

```
import { useQuery, useMutation, useSubscription } from '@apollo/client'


const App = () => {
  // ...

  useSubscription(PERSON_ADDED, {
    onData: ({ data }) => {
      console.log(data)
    }
  })

  // ...
}
```

When a new person is now added to the phonebook, no matter where it's done, the details of the new person are printed to the client's console:



When a new person is added, the server sends a notification to the client, and the callback function defined in the `onData` attribute is called and given the details of the new person as parameters.

Let's extend our solution so that when the details of a new person are received, the person is added to the Apollo cache, so it is rendered to the screen immediately.

```
const App = () => {
  // ...

  useSubscription(PERSON_ADDED, {
    onData: ({ data, client }) => {
      const addedPerson = data.data.personAdded
      notify(`${addedPerson.name} added`)

      client.cache.updateQuery({ query: ALL_PERSONS }, ({ allPersons }) => {
        return {
          allPersons: allPersons.concat(addedPerson),
        }
      })
    }
  })

  // ...
}
```

Our solution has a small problem: a person is added to the cache and also rendered twice since the component `PersonForm` is adding it to the cache as well.

Let us now fix the problem by ensuring that a person is not added twice in the cache:

```
// function that takes care of manipulating cache
export const updateCache = (cache, query, addedPerson) => {
  // helper that is used to eliminate saving same person twice
  const uniqByName = (a) => {
    let seen = new Set()
    return a.filter((item) => {
      let k = item.name
      return seen.has(k) ? false : seen.add(k)
    })
  }

  cache.updateQuery(query, ({ allPersons }) => {
    return {
      allPersons: uniqByName(allPersons.concat(addedPerson)),
    }
  })
}

const App = () => {
  const result = useQuery(ALL_PERSONS)
```

```
  const [errorMessage, setErrorMessage] = useState(null)
  const [token, setToken] = useState(null)
  const client = useApolloClient()

  useSubscription(PERSON_ADDED, {
    onData: ({ data, client }) => {
      const addedPerson = data.data.personAdded
      notify(`${addedPerson.name} added`)
      updateCache(client.cache, { query: ALL_PERSONS }, addedPerson)
    },
  })

  // ...
}
```

The function `updateCache` can also be used in `PersonForm` for the cache update:

```
import { updateCache } from '../App'

const PersonForm = ({ setError }) => {
  // ...

  const [createPerson] = useMutation(CREATE_PERSON, {
    onError: (error) => {
      setError(error.graphQLErrors[0].message)
    },
    update: (cache, response) => {
      updateCache(cache, { query: ALL_PERSONS }, response.data.addPerson)
    },
  })

  // ..
}
```

The final code of the client can be found on GitHub, branch *part8-6*.

n+1 problem

First of all, you'll need to enable a debugging option via `mongoose` in your backend project directory, by adding a line of code as shown below:

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

```
mongoose.set('debug', true);
```

Let's add some things to the backend. Let's modify the schema so that a *Person* type has a `friendOf` field, which tells whose friends list the person is on.

```
type Person {
  name: String!
  phone: String
  address: Address!
  friendOf: [User!]!
  id: ID!
}
```

The application should support the following query:

```
query {
  findPerson(name: "Leevi Hellas") {
    friendOf {
      username
    }
  }
}
```

Because `friendOf` is not a field of *Person* objects on the database, we have to create a resolver for it, which can solve this issue. Let's first create a resolver that returns an empty list:

```
Person: {
  address: (root) => {
    return {
      street: root.street,
      city: root.city
    }
  },
  friendOf: (root) => {
    // return list of users
    return [
    ]
  }
},
```

The parameter `root` is the person object for which a friends list is being created, so we search from all `User` objects the ones which have root._id in their friends list:

```
  Person: {
    // ...
    friendOf: async (root) => {
      const friends = await User.find({
        friends: {
          $in: [root._id]
        }
      })

      return friends
    }
  },
```

Now the application works.

We can immediately do even more complicated queries. It is possible for example to find the friends of all users:

```
query {
  allPersons {
    name
    friendOf {
      username
    }
  }
}
```

There is however one issue with our solution: it does an unreasonable amount of queries to the database. If we log every query to the database, just like this for example,

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

// ..

},

// ..

friendOf: async (root) => {
  const friends = await User.find({ friends: { $in: [root._id] } })
  console.log("User.find")
```

```
  return friends
},
```

and considering we have 5 persons saved, and we query `allPersons` without `phone` as argument, we see an absurd amount of queries like below.

```
Person.find
User.find
User.find
User.find
User.find
User.find
```

copy

So even though we primarily do one query for all persons, every person causes one more query in their resolver.

This is a manifestation of the famous n+1 problem, which appears every once in a while in different contexts, and sometimes sneaks up on developers without them noticing.

The right solution for the n+1 problem depends on the situation. Often, it requires using some kind of a join query instead of multiple separate queries.

In our situation, the easiest solution would be to save whose friends list they are on each `Person` object:

```
const schema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    minlength: 5
  },
  phone: {
    type: String,
    minlength: 5
  },
  street: {
    type: String,
    required: true,
    minlength: 5
  },
  city: {
    type: String,
    required: true,
    minlength: 5
  },
  friendOf: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'User'
    }
```

copy

```
  ],
})
```

Then we could do a "join query", or populate the `friendOf` fields of persons when we fetch the `Person` objects:

```
Query: {
  allPersons: (root, args) => {
    console.log('Person.find')
    if (!args.phone) {
      return Person.find({}).populate('friendOf')
    }

    return Person.find({ phone: { $exists: args.phone === 'YES' } })
      .populate('friendOf')
  },
  // ...
}
```

After the change, we would not need a separate resolver for the `friendOf` field.

The allPersons query *does not cause* an n+1 problem, if we only fetch the name and the phone number:

```
query {
  allPersons {
    name
    phone
  }
}
```

If we modify `allPersons` to do a join query because it sometimes causes an n+1 problem, it becomes heavier when we don't need the information on related persons. By using the fourth parameter of resolver functions, we could optimize the query even further. The fourth parameter can be used to inspect the query itself, so we could do the join query only in cases with a predicted threat of n+1 problems. However, we should not jump into this level of optimization before we are sure it's worth it.

In the words of Donald Knuth:

> *Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time:* ***premature optimization is the root of all evil.***

GraphQL Foundation's DataLoader library offers a good solution for the n+1 problem among other issues. More about using DataLoader with Apollo server here and here.

Epilogue

The application we created in this part is not optimally structured: we did some cleanups but much would still need to be done. Examples for better structuring of GraphQL applications can be found on the internet. For example, for the server here and the client here.

GraphQL is already a pretty old technology, having been used by Facebook since 2012, so we can see it as "battle-tested" already. Since Facebook published GraphQL in 2015, it has slowly gotten more and more attention, and might in the near future threaten the dominance of REST. The death of REST has also already been predicted. Even though that will not happen quite yet, GraphQL is absolutely worth learning.

Exercises 8.23.-8.26

8.23: Subscriptions - server

Do a backend implementation for subscription `bookAdded`, which returns the details of all new books to its subscribers.

8.24: Subscriptions - client, part 1

Start using subscriptions in the client, and subscribe to `bookAdded`. When new books are added, notify the user. Any method works. For example, you can use the window.alert function.

8.25: Subscriptions - client, part 2

Keep the application's book view updated when the server notifies about new books (you can ignore the author view!). You can test your implementation by opening the app in two browser tabs and adding a new book in one tab. Adding the new book should update the view in both tabs.

8.26: n+1

Solve the n+1 problem of the following query using any method you like.

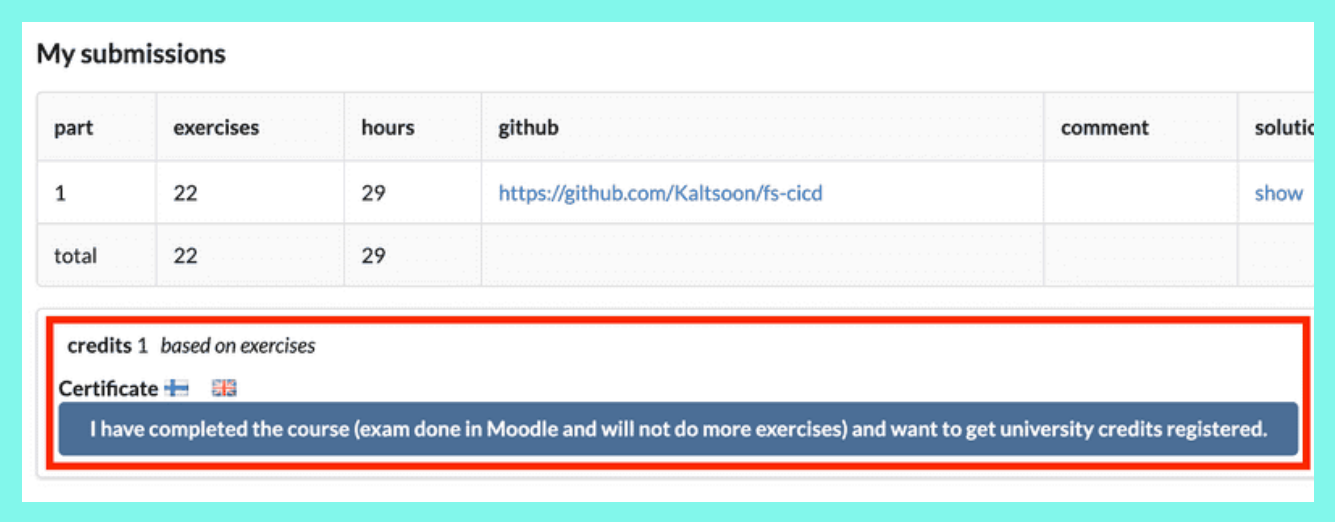
```
query {
  allAuthors {
    name
    bookCount
  }
}
```

copy

Submitting exercises and getting the credits

Exercises of this part are submitted via the submissions system just like in the previous parts, but unlike previous parts, the submission goes to different "course instance". Remember that you have to finish at least 22 exercises to pass this part!

Once you have completed the exercises and want to get the credits, let us know through the exercise submission system that you have completed the course:



Note that you need a registration to the corresponding course part for getting the credits registered, see here for more information.

You can download the certificate for completing this part by clicking one of the flag icons. The flag icon corresponds to the certificate's language.

Propose changes to material

Part 8d
Previous part

Part 9
Next part

About course

Course contents

FAQ

Partners

Challenge


HOUSTON