Fragments and subscriptions

We are approaching the end of the course. Let's finish by having a look at a few more details of 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:

```
const ALL_PERSONS = gql`
  {
    allPersons  {
      ...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:

```
const ALL_PERSONS = gql`
  {
    allPersons  {
      ...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 the server has been 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.

Subscriptions on the server

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

There are not many changes to the server. The schema changes like so:

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

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

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('apollo-server')
const pubsub = new PubSub()

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

      if (!currentUser) {
        throw new AuthenticationError("not authenticated")
      }

      try {
        await person.save()
        currentUser.friends = currentUser.friends.concat(person)
        await currentUser.save()
      } catch (error) {
        throw new UserInputError(error.message, {
          invalidArgs: args,
        })
      }

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

      return person
    },
```

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

With subscriptions, the communication happens using the publish-subscribe principle utilizing an object using a PubSub interface. Adding a new person *publishes* a notification about the operation to all subscribers with PubSub's method `publish`.

`personAdded` subscriptions resolver registers all of the subscribers by returning them a suitable iterator object.

Let's do the following changes to the code which starts the server

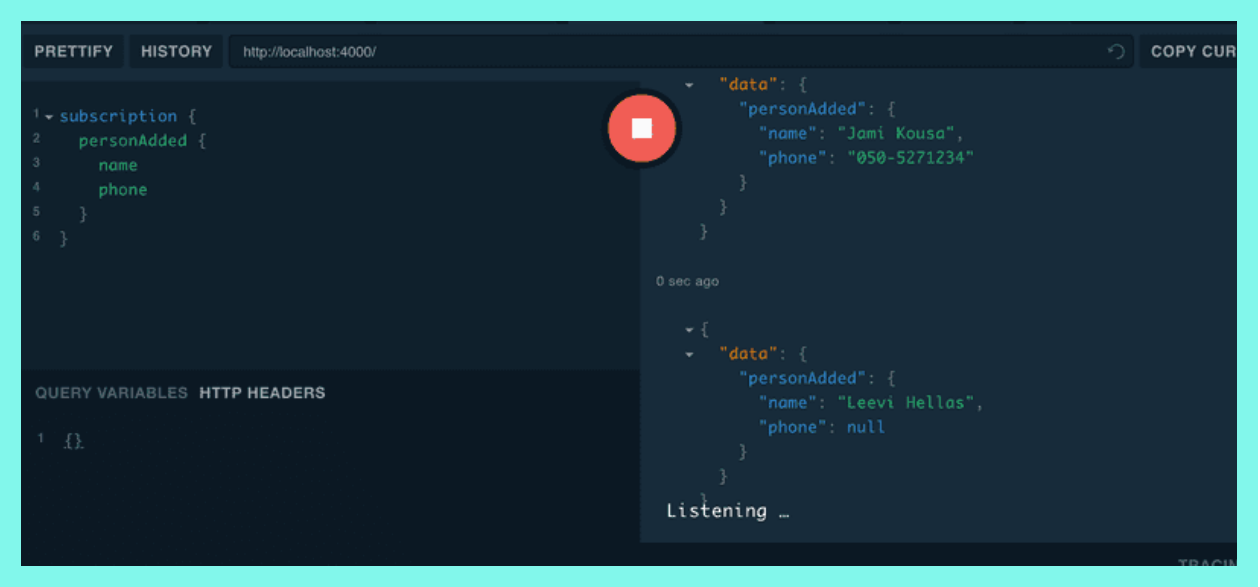
```
// ...

server.listen().then(({ url, subscriptionsUrl }) => {
  console.log(`Server ready at ${url}`)
  console.log(`Subscriptions ready at ${subscriptionsUrl}`)
})
```

We see, that the server listens for subscriptions in the address `ws://localhost:4000/graphql`

```
Server ready at http://localhost:4000/
Subscriptions ready at ws://localhost:4000/graphql
```

No other changes to the server are needed.

It's possible to test the subscriptions with the GraphQL playground like this:



When you press "play" on a subscription, the playground waits for notifications from the subscription.

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

Subscriptions on the client

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

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

import { getMainDefinition } from '@apollo/client/utilities'
import { WebSocketLink } from '@apollo/client/link/ws'

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

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

const wsLink = new WebSocketLink({
  uri: `ws://localhost:4000/graphql`,
```

```
  options: {
    reconnect: true
  }
})

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

For this to work, we have to install some dependencies:

```
npm install @apollo/client subscriptions-transport-ws
```

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.

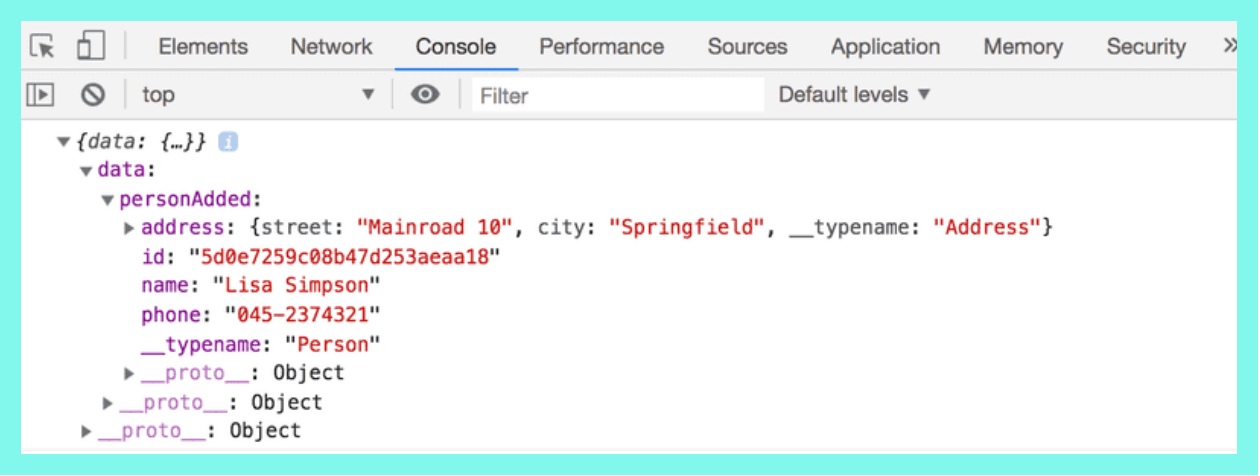
```
const wsLink = new WebSocketLink({
  uri: `ws://localhost:4000/graphql`,
  options: { reconnect: true }
})

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

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

Let's modify the code like so:

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

${PERSON_DETAILS}
`


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

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

  useSubscription(PERSON_ADDED, {
    onSubscriptionData: ({ subscriptionData }) => {
      console.log(subscriptionData)
    }
  })

  // ...
}
```

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 `onSubscriptionData` 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.

However, we have to keep in mind that when our application creates a new person, it should not be added to the cache twice:

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

  const updateCacheWith = (addedPerson) => {
    const includedIn = (set, object) =>
      set.map(p => p.id).includes(object.id)

    const dataInStore = client.readQuery({ query: ALL_PERSONS })
    if (!includedIn(dataInStore.allPersons, addedPerson)) {
      client.writeQuery({
        query: ALL_PERSONS,
        data: { allPersons : dataInStore.allPersons.concat(addedPerson) }
      })
    }
  }

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

  // ...
}
```

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

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

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

  // ..
}
```

The final code of the client can be found on Github, branch *part8-9*.

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, { useNewUrlParser: true, useUnifiedTopology: true, useFindAndModify: fals
  .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 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,

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

and we have 5 persons saved, we see an absurd amount of queries.

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

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.

Good solution for 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 on each `Person` -object:

```
const schema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    unique: 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'
    }
  ],
})
```

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 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 predicted threat for n+1 problem. 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.*

Facebook'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: the schema, queries and the mutations should at least be moved outside of the application code. 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 view updated when the server notifies about new books. 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
  }
}
```

This was the last exercise for this part of the course and it's time to push your code to GitHub and mark all of your finished exercises to the exercise submission system.

About course
Course contents
FAQ
Partners
Challenge

Propose changes to material

Previous pa Part 8 Part 9 Next part


HOUSTON