{() => fs}

Fullstack > Part 8 > Fragments and subscriptions

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*

```
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
```

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)
  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!
}
```

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
```

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();
            },
          };
        },
      },
    ],
  })

  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()
```

```
// ...

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
```

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')
  },
},
```

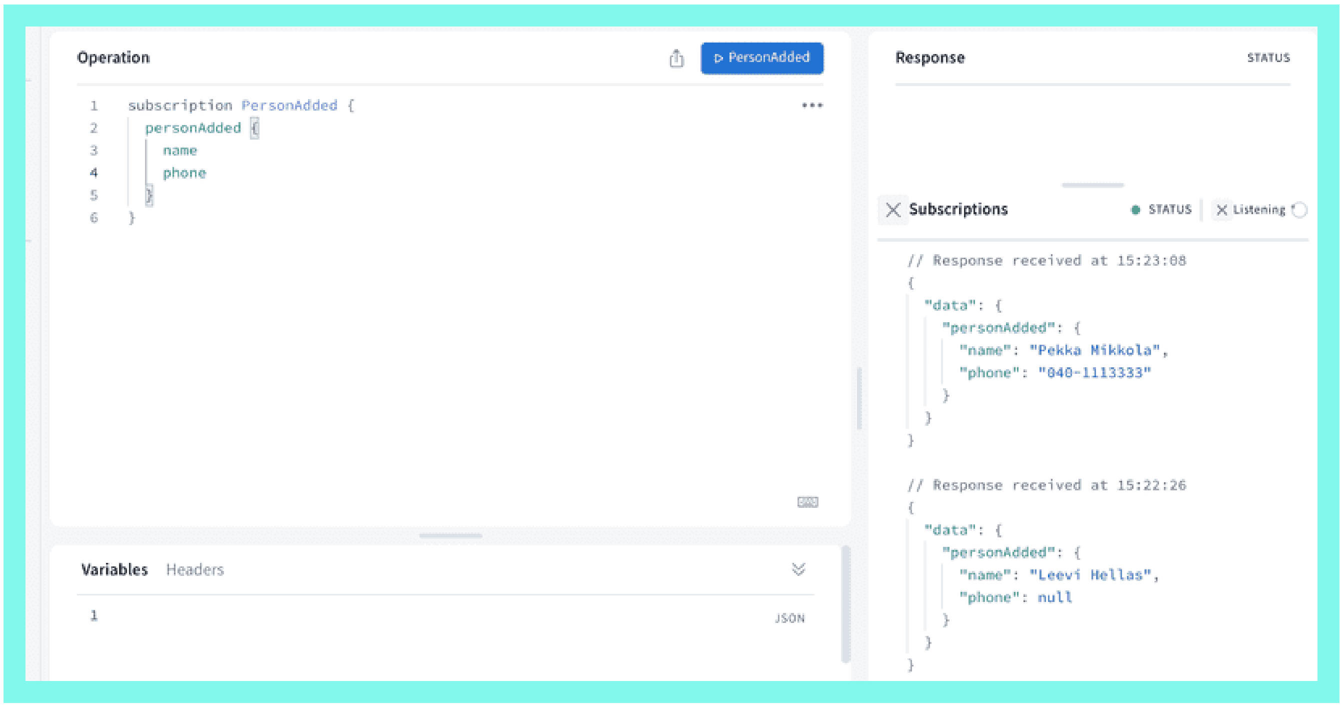
The iterator name is an arbitrary string, but to follow the convention, it is the subscription name written in capital letters.

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:

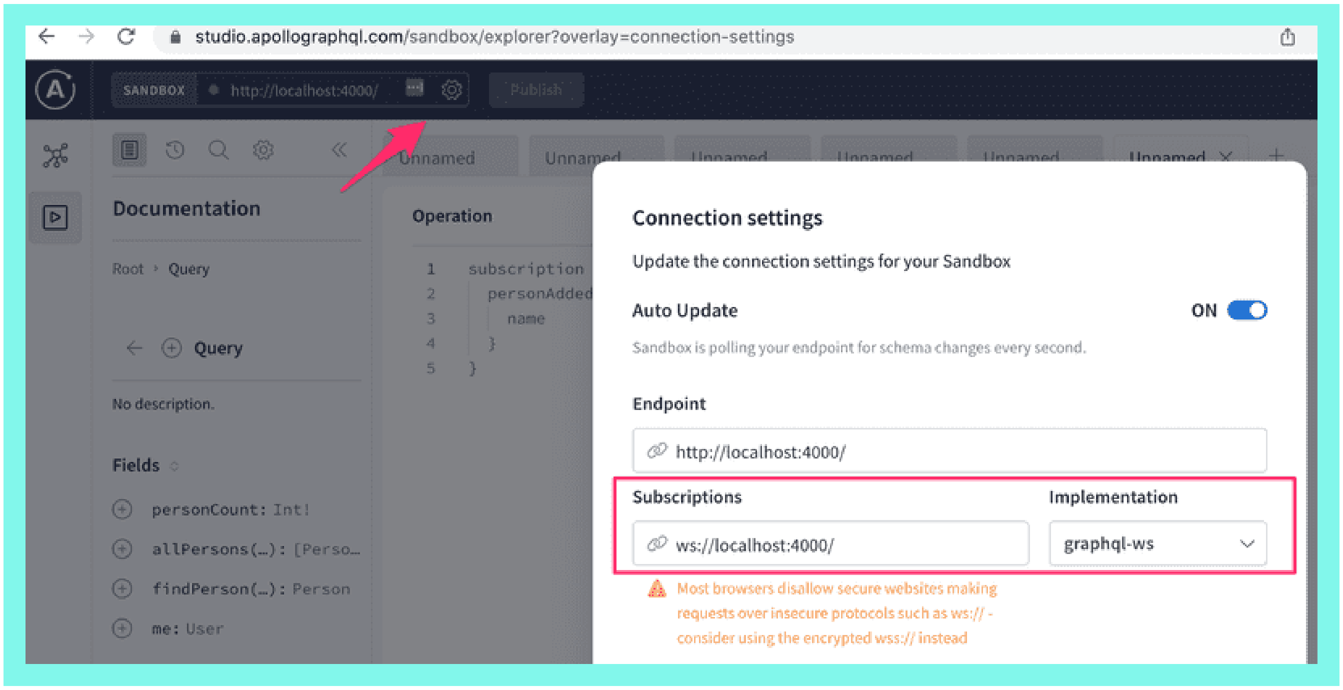


```
subscription PersonAdded {
  personAdded {
    name
    phone
  }
}
```

```
// Response received at 15:23:08
{
  "data": {
    "personAdded": {
      "name": "Pekka Mikkola",
      "phone": "040-1113333"
    }
  }
}

// Response received at 15:22:26
{
  "data": {
    "personAdded": {
      "name": "Leevi Hellas",
      "phone": null
    }
  }
}
```

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:



```
subscription
  personAdded
    name
  }
}
```

Connection settings

Update the connection settings for your Sandbox

Auto Update ON

Sandbox is polling your endpoint for schema changes every second.

Endpoint: http://localhost:4000/

Subscriptions: ws://localhost:4000/

Implementation: graphql-ws

Most browsers disallow secure websites making requests over insecure protocols such as ws:// - consider using the encrypted wss:// instead

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}
`
```

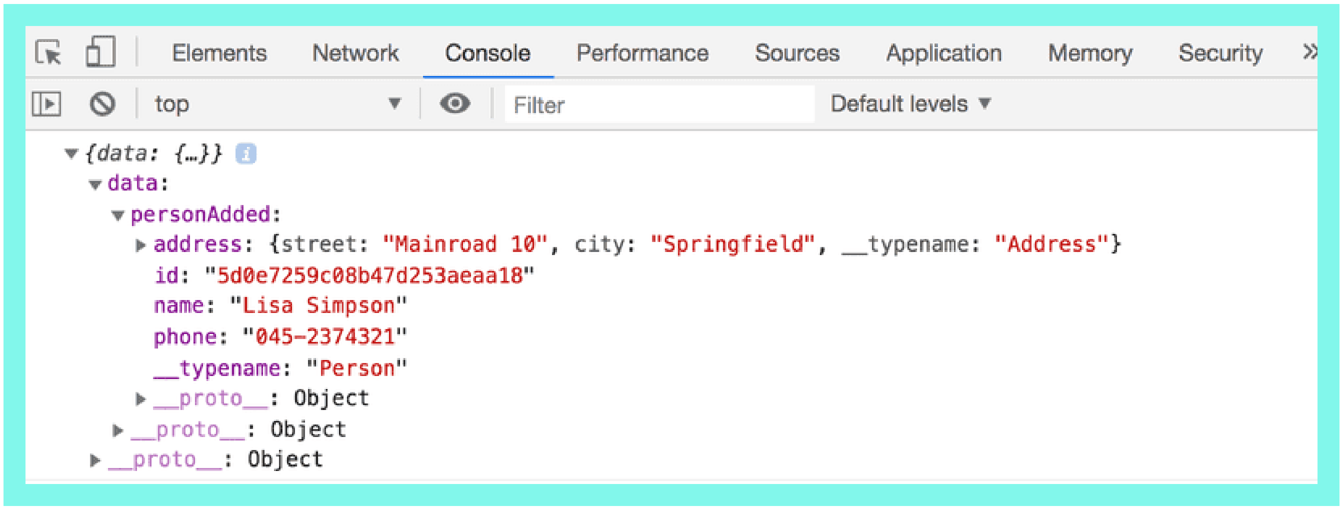
copy

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)
    }
  })

  // ...
}
```

copy

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:



```
▼{data: {…}}
  ▼data:
    ▼personAdded:
      ▶address: {street: "Mainroad 10", city: "Springfield", __typename: "Address"}
       id: "5d0e7259c08b47d253aeaa18"
       name: "Lisa Simpson"
       phone: "045-2374321"
       __typename: "Person"
      ▶__proto__: Object
    ▶__proto__: Object
  ▶__proto__: Object
```

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
```

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'
    }
  ],
})
```

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

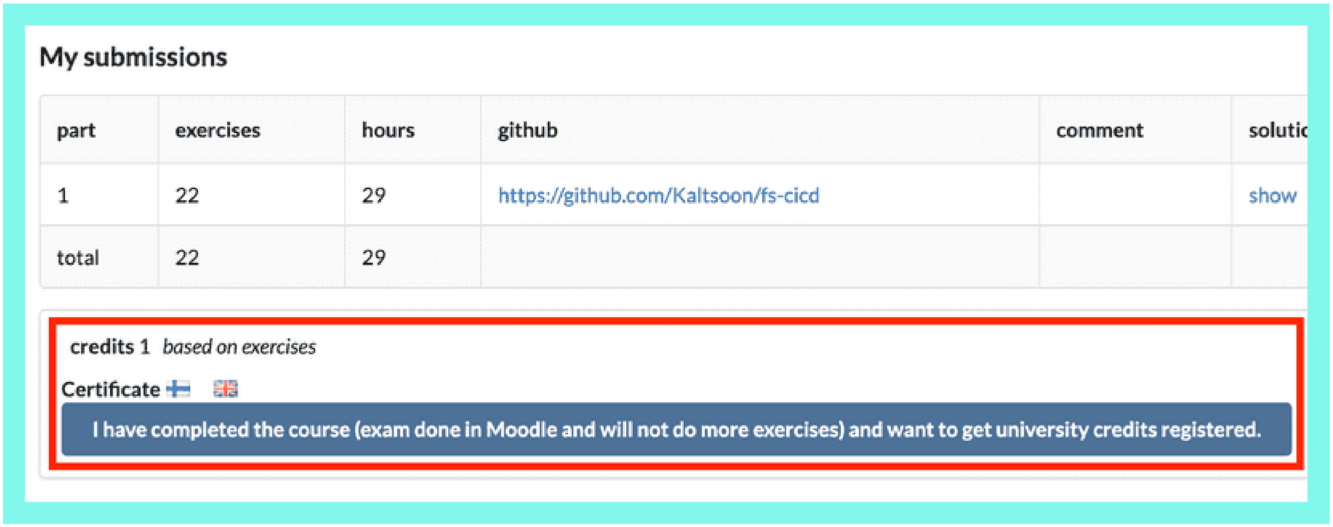
Solve the n+1 problem of the following query using any method you like.

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

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:



My submissions

part	exercises	hours	github	comment	solutic
1	22	29	https://github.com/Kaltsoon/fs-cicd		show
total	22	29			

credits 1 *based on exercises*

Certificate

I have completed the course (exam done in Moodle and will not do more exercises) and want to get university credits registered.

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