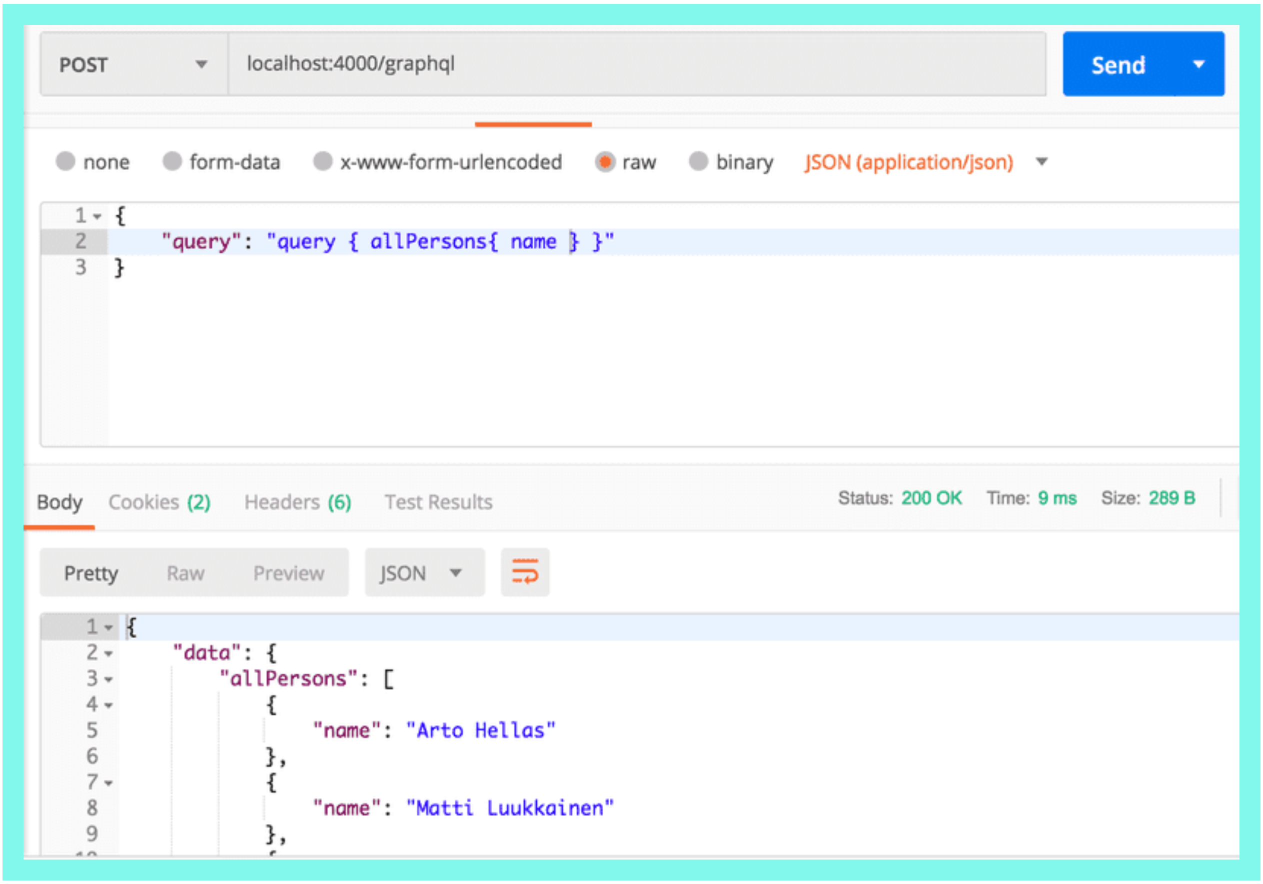
b React and GraphQL

We will next implement a React-app which 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.



```
{
    "query": "query { allPersons{ name } }"
}
```

```
{
    "data": {
        "allPersons": [
            {
                "name": "Arto Hellas"
            },
            {
                "name": "Matti Luukkainen"
            },
```

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

Create a new React-app and install the dependencies required by Apollo client.

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

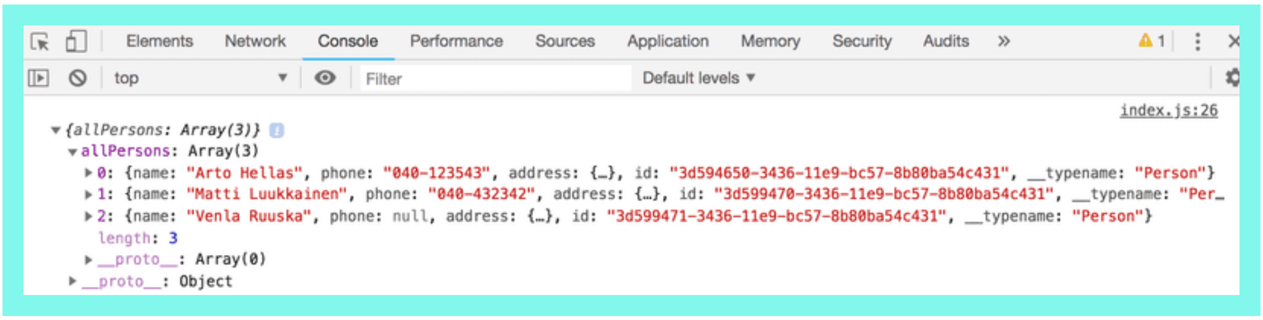
We'll start with the following code for our application.

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

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

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

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

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

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

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 servers response is printed to the console:



```
                                                                              index.js:26
▼{allPersons: Array(3)}
 ▼allPersons: Array(3)
  ▶0: {name: "Arto Hellas", phone: "040-123543", address: {…}, id: "3d594650-3436-11e9-bc57-8b80ba54c431", __typename: "Person"}
  ▶1: {name: "Matti Luukkainen", phone: "040-432342", address: {…}, id: "3d599470-3436-11e9-bc57-8b80ba54c431", __typename: "Per…
  ▶2: {name: "Venla Ruuska", phone: null, address: {…}, id: "3d599471-3436-11e9-bc57-8b80ba54c431", __typename: "Person"}
   length: 3
  ▶__proto__: Array(0)
 ▶__proto__: Object
```

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 React from 'react'
import ReactDOM from 'react-dom'
import App from './App'

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

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

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

Making queries

We are ready to implement the main view of the application, which shows a list of phone numbers.

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, which's code is as follows:

```
import React from 'react'
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 response is received, the result of the *allPersons* query can be found from 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.

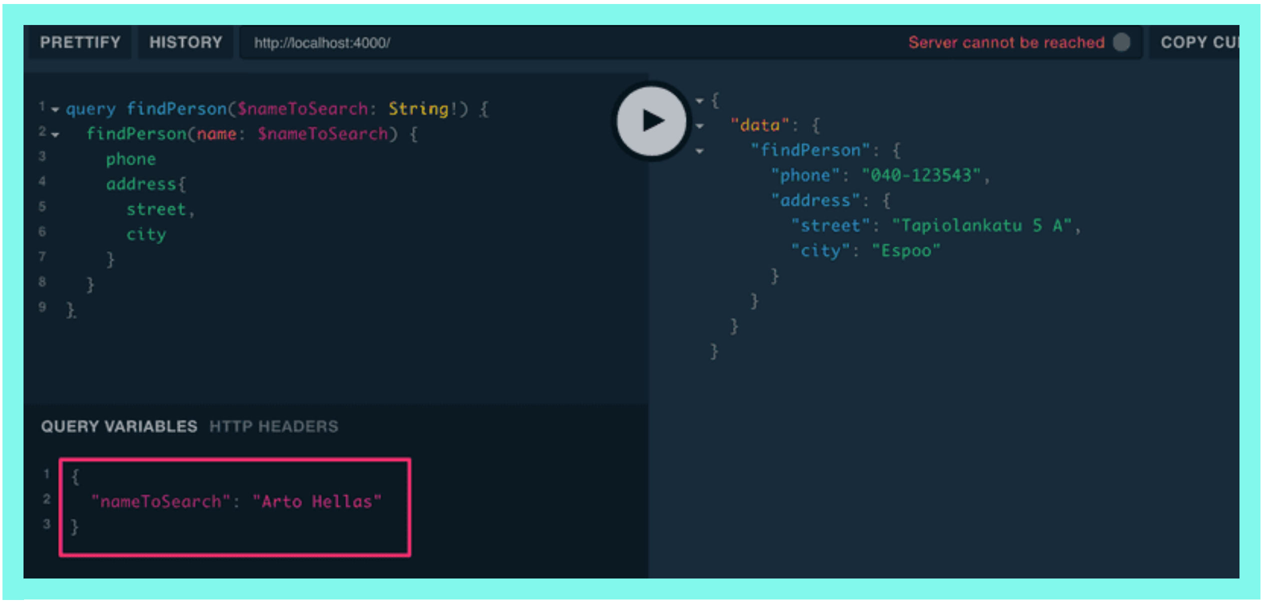
GraphQL variables are well suited for this. To be able to use variables, we must also name our queries.

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 GraphQL Playground. The parameters are given in *Query variables*:



```
query findPerson($nameToSearch: String!) {
  findPerson(name: $nameToSearch) {
    phone
    address{
      street,
      city
    }
  }
}
```

```
{
  "nameToSearch": "Arto Hellas"
}
```

```
{
  "data": {
    "findPerson": {
      "phone": "040-123543",
      "address": {
        "street": "Tapiolankatu 5 A",
        "city": "Espoo"
      }
    }
  }
}
```

The `useQuery` hook is well suited for situations where the query is done when the component is rendered. However now we 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.

For this this situation the hook-function useLazyQuery is a good choice. The *Persons* component becomes:

```
const FIND_PERSON = gql`
  query findPersonByName($nameToSearch: String!) {
    findPerson(name: $nameToSearch) {
      name
      phone
      id
      address {
        street
        city
      }
    }
  }
`

const Persons = ({ persons }) => {
  const [getPerson, result] = useLazyQuery(FIND_PERSON)
  const [person, setPerson] = useState(null)

  const showPerson = (name) => {
    getPerson({ variables: { nameToSearch: name } })
  }

  useEffect(() => {
    if (result.data) {
      setPerson(result.data.findPerson)
    }
  }, [result])

  if (person) {
    return(
      <div>
        <h2>{person.name}</h2>
        <div>{person.address.street} {person.address.city}</div>
        <div>{person.phone}</div>
        <button onClick={() => setPerson(null)}>close</button>
      </div>
    )
  }

  return (
    <div>
      <h2>Persons</h2>
      {persons.map(p =>
        <div key={p.name}>
          {p.name} {p.phone}
          <button onClick={() => showPerson(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 a person's "show address" button is clicked, its event handler `showPerson` is executed, and makes a GraphQL query to fetch the persons details:

```
const [getPerson, result] = useLazyQuery(FIND_PERSON)

// ...

const showPerson = (name) => {
  getPerson({ variables: { nameToSearch: name } })
}
```

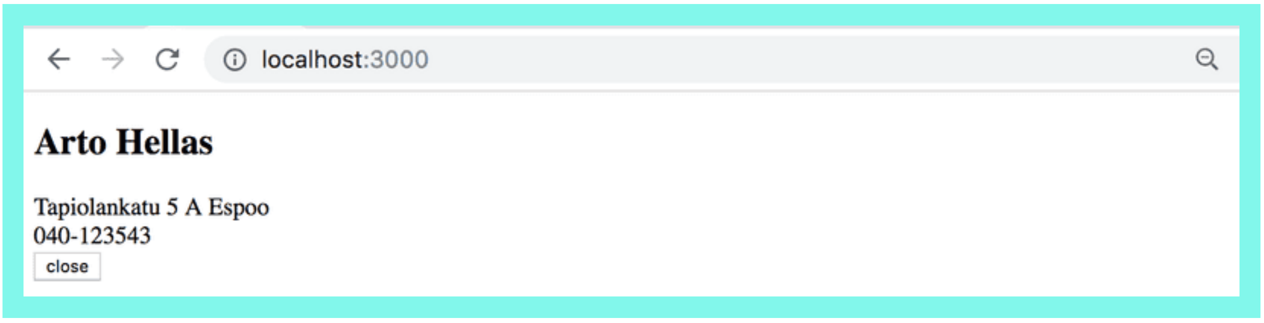
The query's `nameToSearch` variable receives a value when the query is run.

The query response is saved to the variable `result` , and its value is saved to the component's state `person` in the `useEffect` hook.

```
useEffect(() => {
  if (result.data) {
    setPerson(result.data.findPerson)
  }
}, [result])
```

The hook's second parameter is `result` , so the function given to the hook as its second parameter is executed *every time the query fetches the details of a different person*. Without handling the update in a controlled way in a hook, returning from a single person view to an all persons view would cause problems.

If the state `person` has a value, instead of showing a list of all persons, only the details of one person are shown.



Arto Hellas

Tapiolankatu 5 A Espoo
040-123543

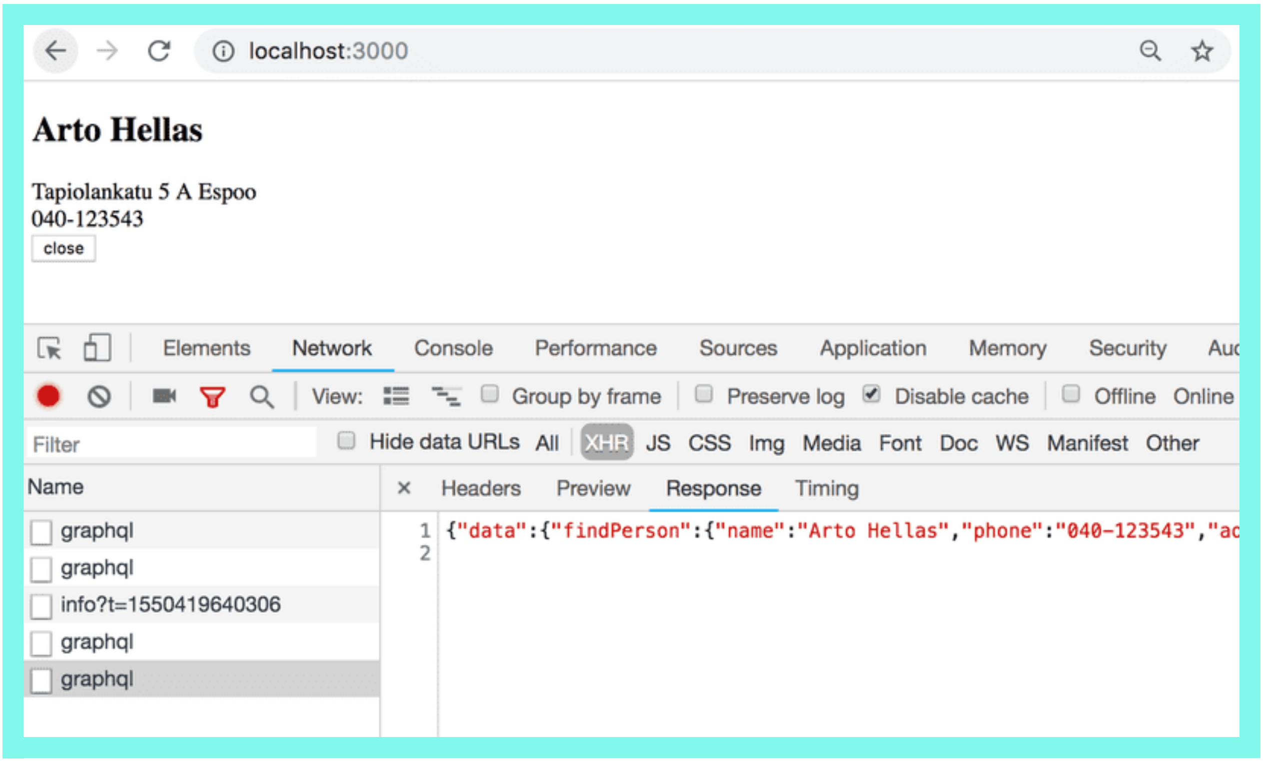
When a user wants to return to the persons list, the `person` state is set to `null` .

The solution is not the neatest possible, but it is good enough for us.

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

Cache

When we do multiple queries for example 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 of the same query being done again by the code, the query is not sent to the backend.



Arto Hellas

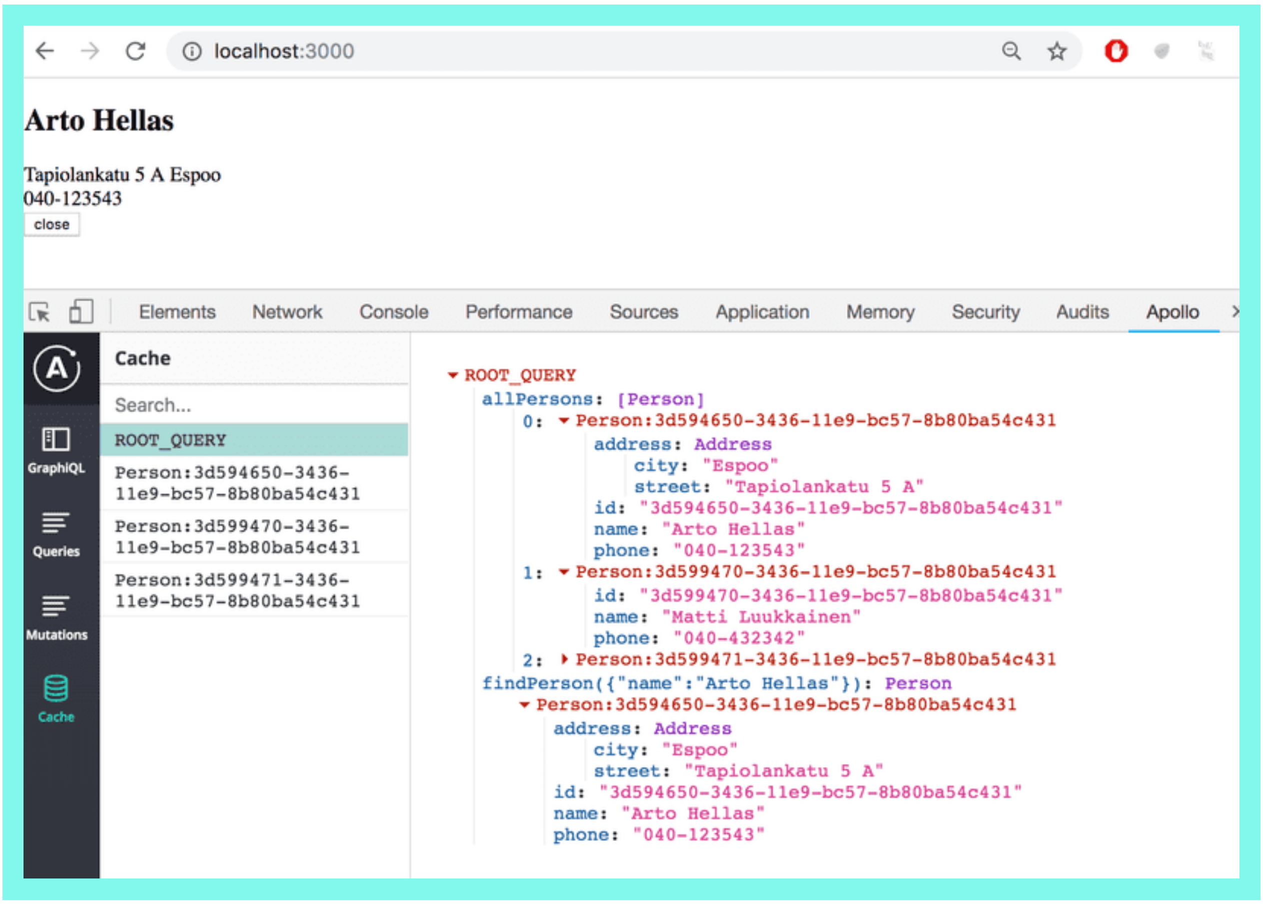
Tapiolankatu 5 A Espoo
040-123543

```
graphql
graphql
info?t=1550419640306
graphql
graphql
```

```
{"data":{"findPerson":{"name":"Arto Hellas","phone":"040-123543","ac
```

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.

It is possible to install Apollo Client devtools to Chrome to view the state of the cache.



```
ROOT_QUERY
Person:3d594650-3436-11e9-bc57-8b80ba54c431
Person:3d599470-3436-11e9-bc57-8b80ba54c431
Person:3d599471-3436-11e9-bc57-8b80ba54c431
```

```
▾ ROOT_QUERY
  allPersons: [Person]
    0: ▾ Person:3d594650-3436-11e9-bc57-8b80ba54c431
         address: Address
           city: "Espoo"
           street: "Tapiolankatu 5 A"
         id: "3d594650-3436-11e9-bc57-8b80ba54c431"
         name: "Arto Hellas"
         phone: "040-123543"
    1: ▾ Person:3d599470-3436-11e9-bc57-8b80ba54c431
         id: "3d599470-3436-11e9-bc57-8b80ba54c431"
         name: "Matti Luukkainen"
         phone: "040-432342"
    2: ▸ Person:3d599471-3436-11e9-bc57-8b80ba54c431
  findPerson({"name":"Arto Hellas"}): Person
    ▾ Person:3d594650-3436-11e9-bc57-8b80ba54c431
      address: Address
        city: "Espoo"
        street: "Tapiolankatu 5 A"
      id: "3d594650-3436-11e9-bc57-8b80ba54c431"
      name: "Arto Hellas"
      phone: "040-123543"
```

Data in the cache is organized by query. Because *Person* objects have an identifying field *id* which is type *ID*, if the same object is returned by multiple queries, Apollo is able to combine them into one. Because of this, doing *findPerson* queries for the address details of Arto Hellas has updated the address details also for the query *allPersons*.

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 React, { 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 function using the `useMutation` -hook. The hook returns an *array*, first element of which contains the function to cause the mutation.

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

The query variables receive values when the query is made:

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

New persons are added just fine, but the screen is not updated. The reason being that 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 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.

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

At the moment in our code queries and component are defined in the same place. 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) {
    // ...
  }
`
```

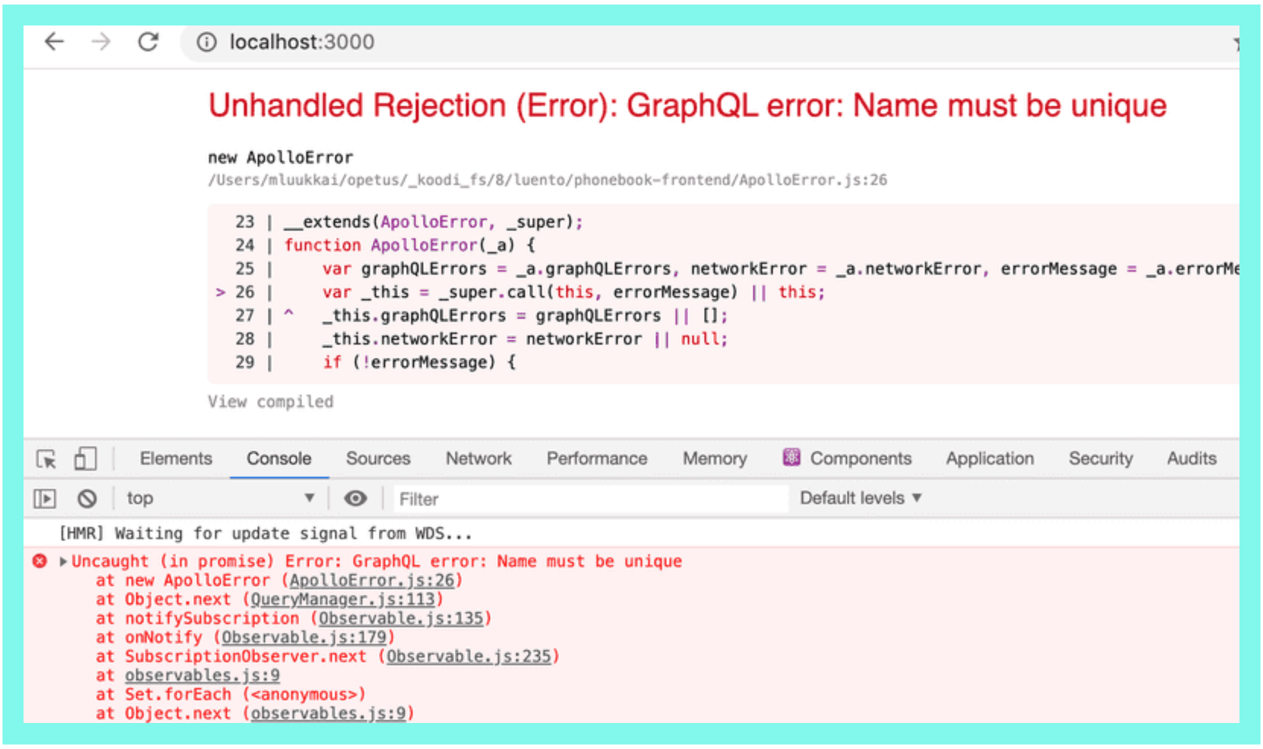
Each component then imports the queries it needs:

```
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, and the whole application breaks



Unhandled Rejection (Error): GraphQL error: Name must be unique

```
new ApolloError
/Users/mluukkai/opetus/_koodi_fs/8/luento/phonebook-frontend/ApolloError.js:26

  23 | __extends(ApolloError, _super);
  24 | function ApolloError(_a) {
  25 |     var graphQLErrors = _a.graphQLErrors, networkError = _a.networkError, errorMessage = _a.errorMe
> 26 |     var _this = _super.call(this, errorMessage) || this;
  27 | ^   _this.graphQLErrors = graphQLErrors || [];
  28 |     _this.networkError = networkError || null;
  29 |     if (!errorMessage) {

View compiled
```

```
[HMR] Waiting for update signal from WDS...
Uncaught (in promise) Error: GraphQL error: Name must be unique
    at new ApolloError (ApolloError.js:26)
    at Object.next (QueryManager.js:113)
    at notifySubscription (Observable.js:135)
    at onNotify (Observable.js:179)
    at SubscriptionObserver.next (Observable.js:235)
    at observables.js:9
    at Set.forEach (<anonymous>)
    at Object.next (observables.js:9)
```

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

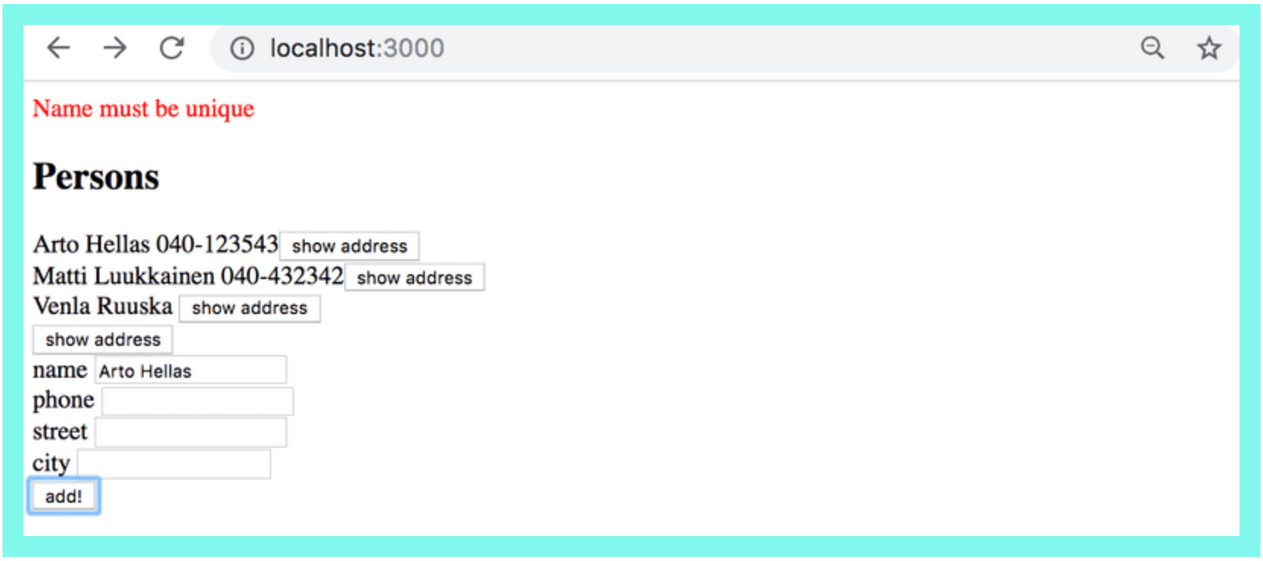
Let's register the mutation an error handler, which uses the `setError` function it receives as a parameter to set an error message:

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

  const [ createPerson ] = useMutation(CREATE_PERSON, {
    refetchQueries: [  {query: ALL_PERSONS } ],
    onError: (error) => {
      setError(error.graphQLErrors[0].message)
    }
  })

  // ...
}
```

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.



localhost:3000

Name must be unique

Persons

Arto Hellas 040-123543 show address
Matti Luukkainen 040-432342 show address
Venla Ruuska show address
show address
name Arto Hellas
phone
street
city
add!

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 solutions is almost identical to the one we used for adding new persons.

Again, the mutation requires parameters.

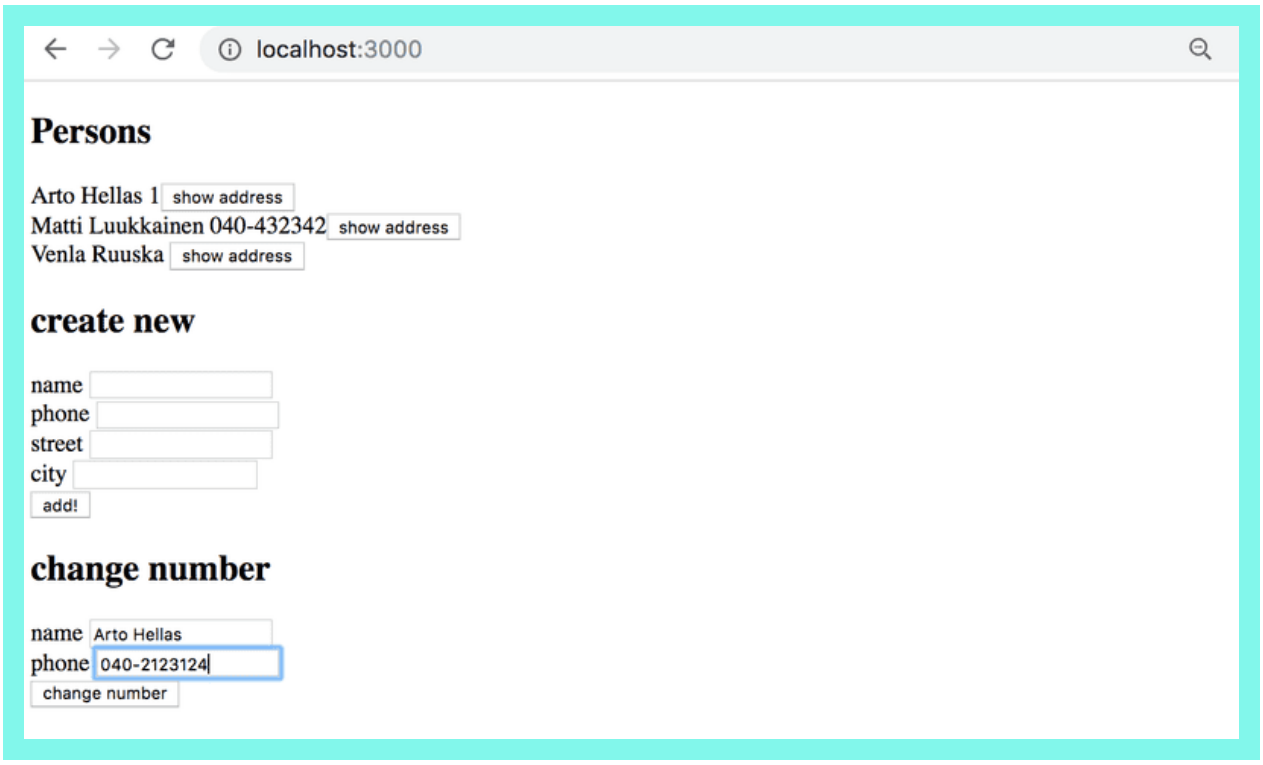
```
export const EDIT_NUMBER = gql`
  mutation editNumber($name: String!, $phone: String!) {
    editNumber(name: $name, phone: $phone)  {
      name
      phone
      address {
        street
        city
      }
      id
    }
  }
`
```

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 React, { 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:



Persons

Arto Hellas 1 show address
Matti Luukkainen 040-432342 show address
Venla Ruuska show address

create new

name
phone
street
city
add!

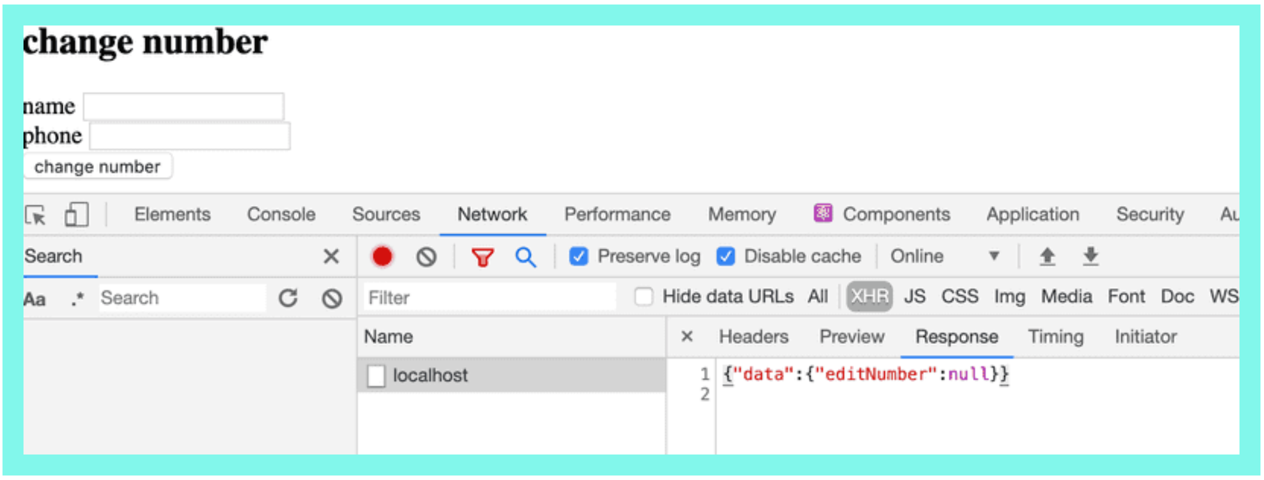
change number

name Arto Hellas
phone 040-2123124
change number

Surprisingly, when 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.

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

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*:



```
{"data":{"editNumber":null}}
```

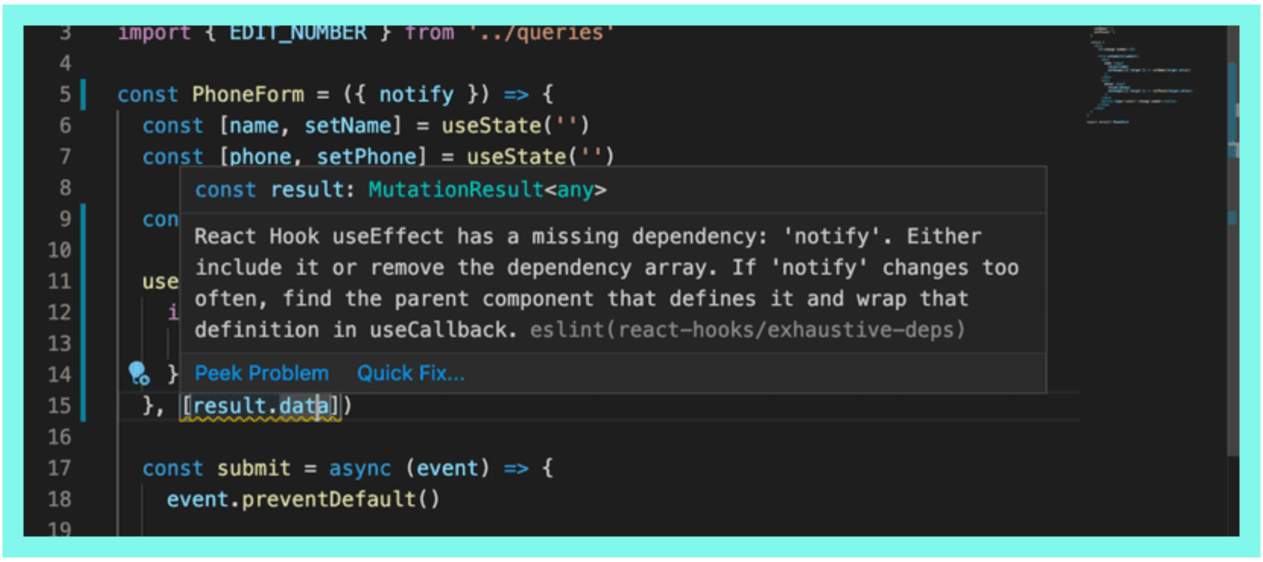
For GraphQL this is not an error, so registering an `onError` error handler is not useful.

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.

Using useEffect causes an ESLint warning:



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

const PhoneForm = ({ notify }) => {
  const [name, setName] = useState('')
  const [phone, setPhone] = useState('')
          const result: MutationResult<any>
  con
          React Hook useEffect has a missing dependency: 'notify'. Either
  use     include it or remove the dependency array. If 'notify' changes too
    i     often, find the parent component that defines it and wrap that
          definition in useCallback. eslint(react-hooks/exhaustive-deps)
    }     Peek Problem    Quick Fix...
  }, [result.data])

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

The warning is pointless, and the easiest solution is to ignore the ESLint rule on the line:

```
useEffect(() => {
  if (result.data && !result.data.editNumber) {
    setError('name not found')
  }
}, [result.data])  // eslint-disable-line
```

We could try to get rid of the warning by adding the `setError` function to useEffect's second parameter array:

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

However this solution does not work if the `notify` -function is not wrapped to a useCallback - function. If it's not, this results to an endless loop. When the `App` component is rerendered after a notification is removed, a *new version* of `notify` gets created which causes the effect function to be executed which causes a new notification and so on an so on...

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

Apollo Client and the applications state

In our example, management of the applications state has mostly become the responsibility of

Apollo Client. This is 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 applications local state to Apollo cache.

Exercises 8.8.-8.12.

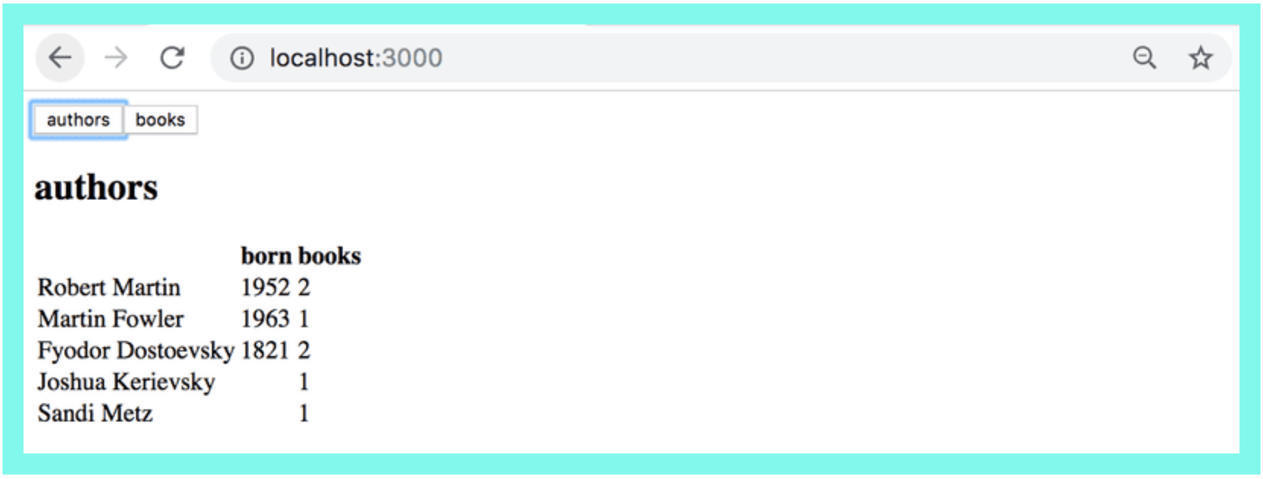
Through these exercises we'll implement a frontend for the GraphQL-library.

Take this project for a start of your application.

You can implement your application either using the render prop -components *Query* and *Mutation* of the Apollo Client, or using the hooks provided by Apollo client 3.0.

8.8: Authors view

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

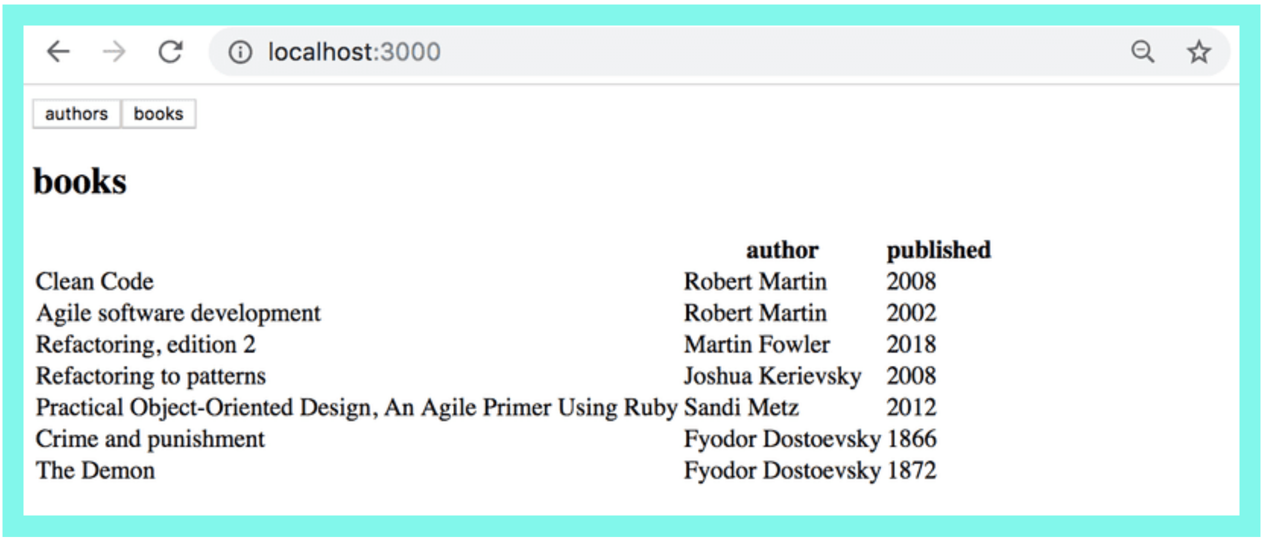


authors

	born	books
Robert Martin	1952	2
Martin Fowler	1963	1
Fyodor Dostoevsky	1821	2
Joshua Kerievsky		1
Sandi Metz		1

8.9: Books view

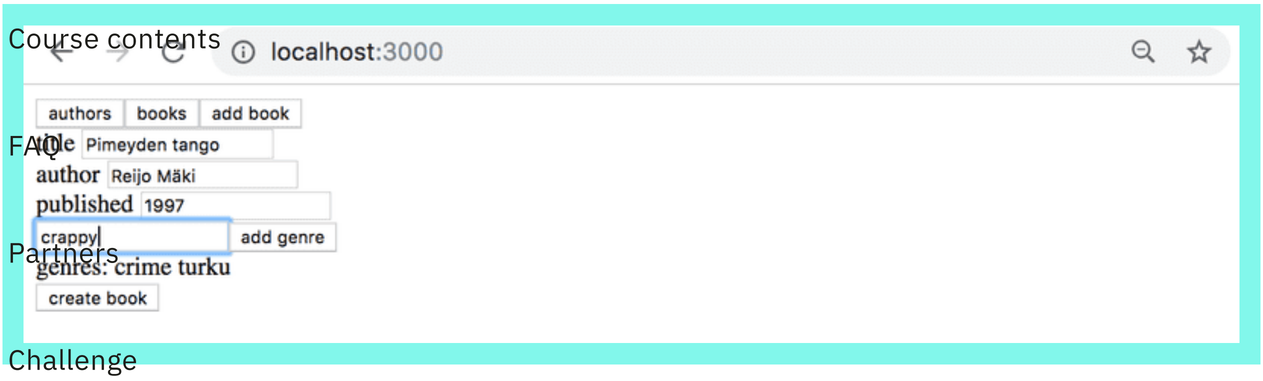
Implement a Books view to show on a page all other details of all books except their genres.



	author	published
Clean Code	Robert Martin	2008
Agile software development	Robert Martin	2002
Refactoring, edition 2	Martin Fowler	2018
Refactoring to patterns	Joshua Kerievsky	2008
Practical Object-Oriented Design, An Agile Primer Using Ruby	Sandi Metz	2012
Crime and punishment	Fyodor Dostoevsky	1866
The Demon	Fyodor Dostoevsky	1872

`8.10: Adding a book`

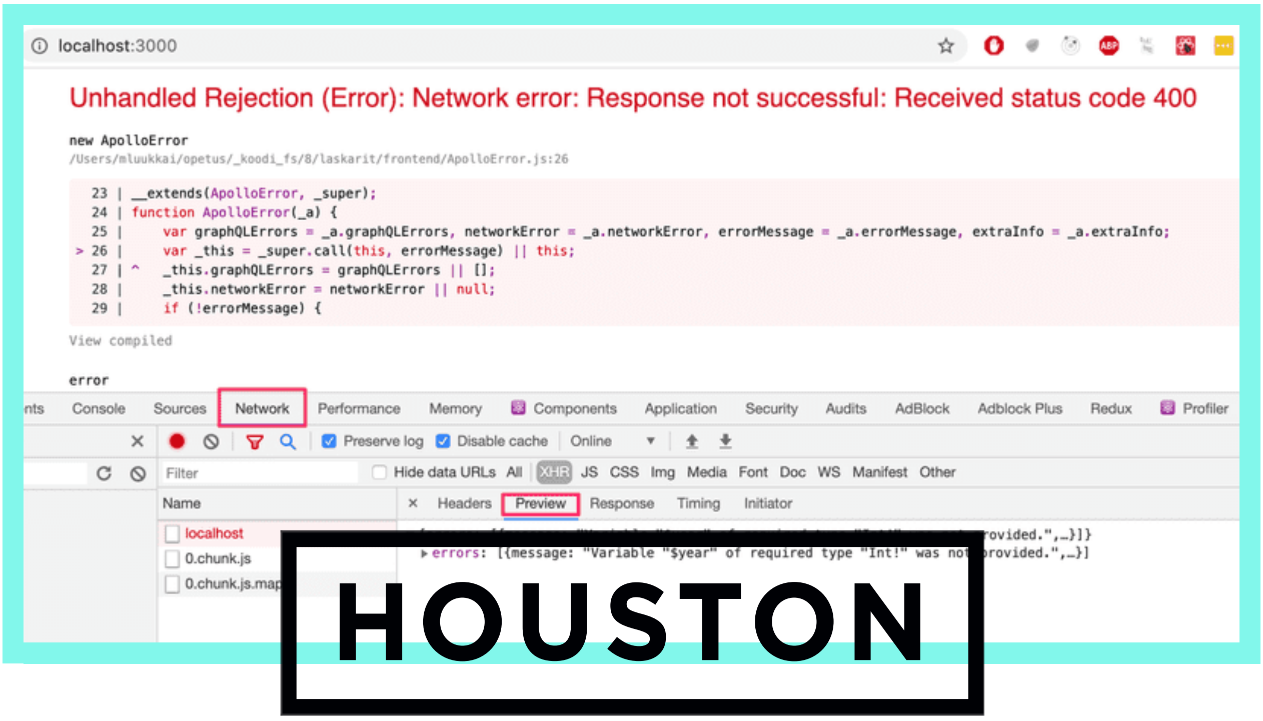
Implement a possibility to add new books to your application. The functionality can look like this:



title Pimeyden tango
author Reijo Mäki
published 1997
crappy
genres: crime turku

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 developer console what the server response is:



localhost:3000

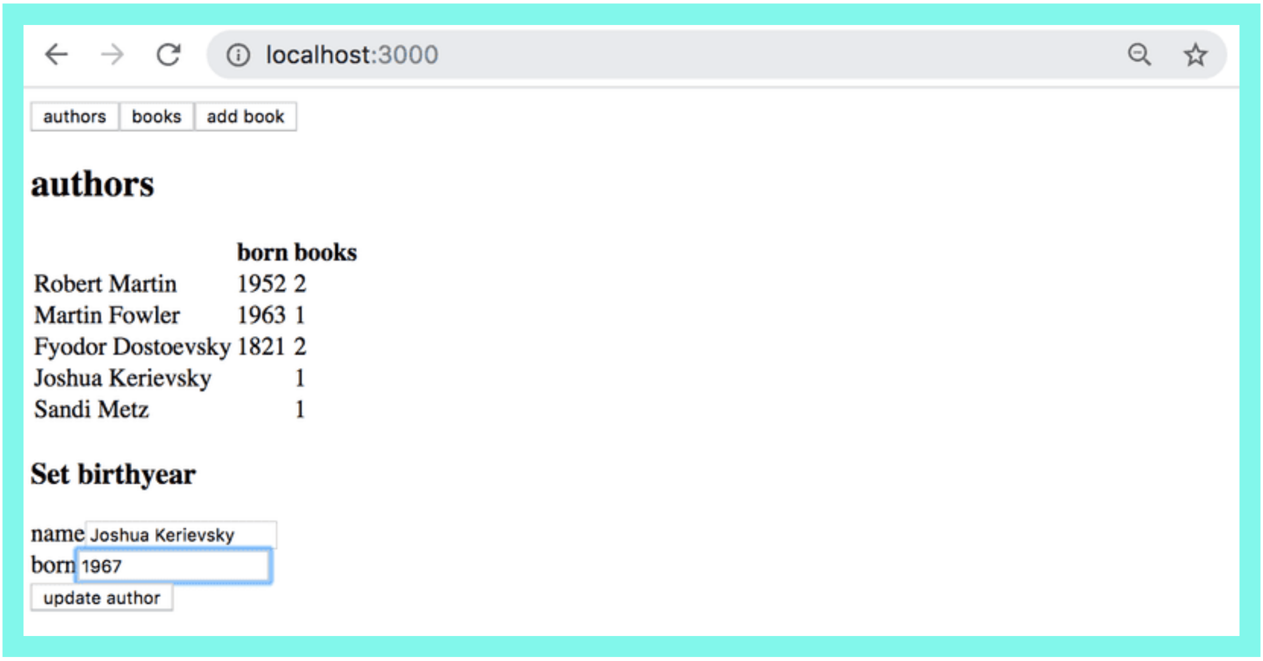
Unhandled Rejection (Error): Network error: Response not successful: Received status code 400

```
new ApolloError
/Users/mluukkai/opetus/_koodi_fs/8/laskarit/frontend/ApolloError.js:26

__extends(ApolloError, _super);
function ApolloError(_a) {
    var graphQLErrors = _a.graphQLErrors, networkError = _a.networkError, errorMessage = _a.errorMessage, extraInfo = _a.extraInfo;
    var _this = _super.call(this, errorMessage) || this;
    _this.graphQLErrors = graphQLErrors || [];
    _this.networkError = networkError || null;
    if (!errorMessage) {

View compiled

error
```

```
▸errors: [{message: "Variable "$year" of required type "Int!" was not provided.",…}]
```

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:



localhost:3000

authors

	born	books
Robert Martin	1952	2
Martin Fowler	1963	1
Fyodor Dostoevsky	1821	2
Joshua Kerievsky		1
Sandi Metz		1

Set birthyear

name	Joshua Kerievsky
born	1967

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

Propose changes to material

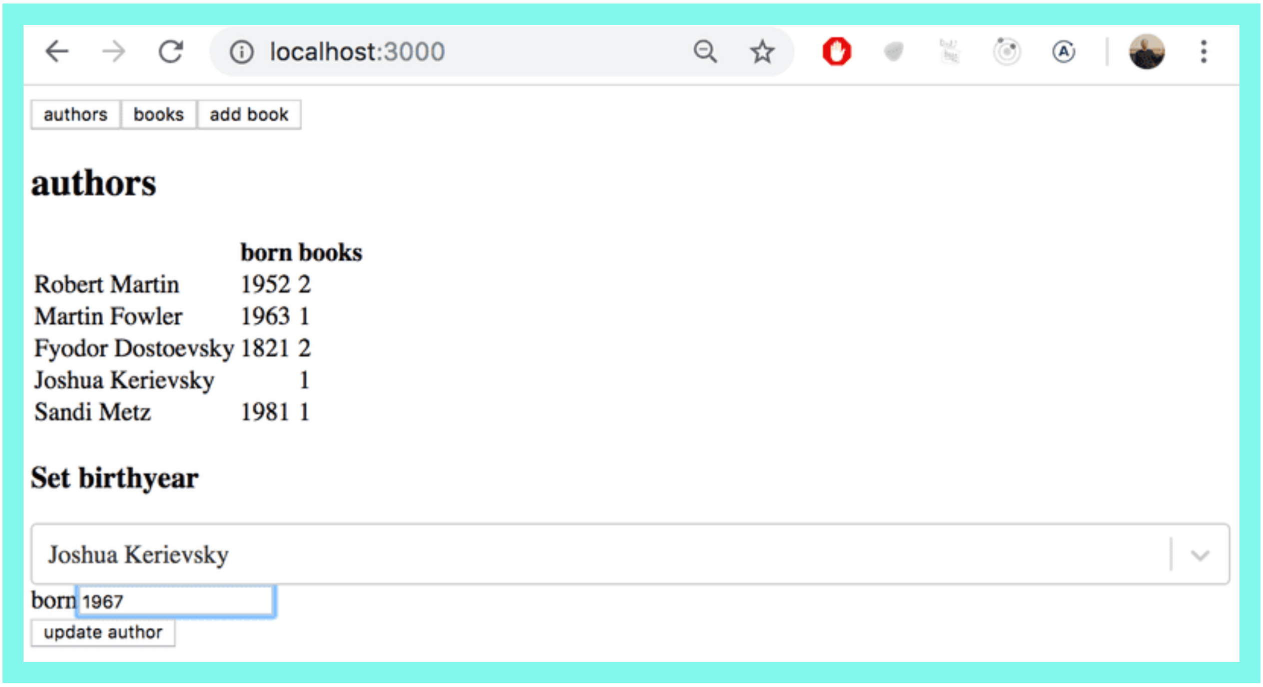
8.12: Authors birth year advanced

Part 8a
Previous part

Part 8c
Next part

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

A solution using the react-select -library looks as follows:



authors

	born	books
Robert Martin	1952	2
Martin Fowler	1963	1
Fyodor Dostoevsky	1821	2
Joshua Kerievsky		1
Sandi Metz	1981	1

Set birthyear

Joshua Kerievsky

born 1967