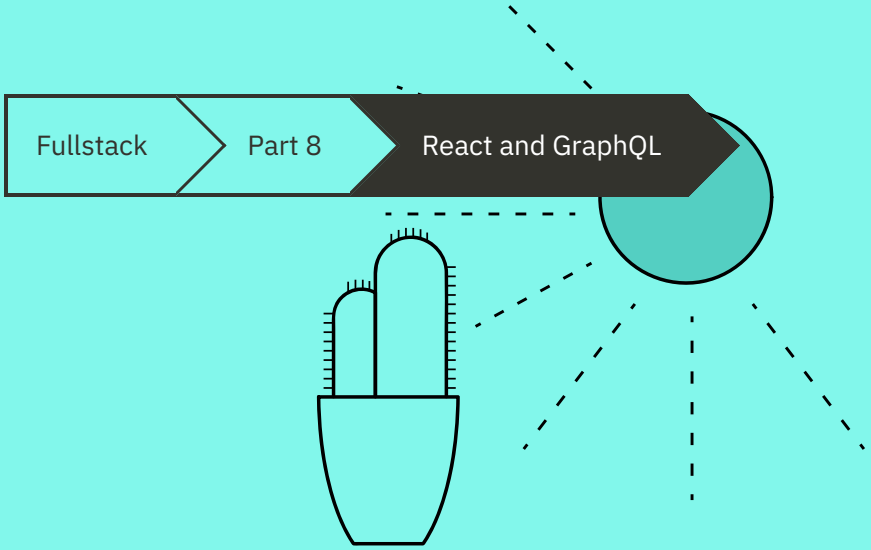


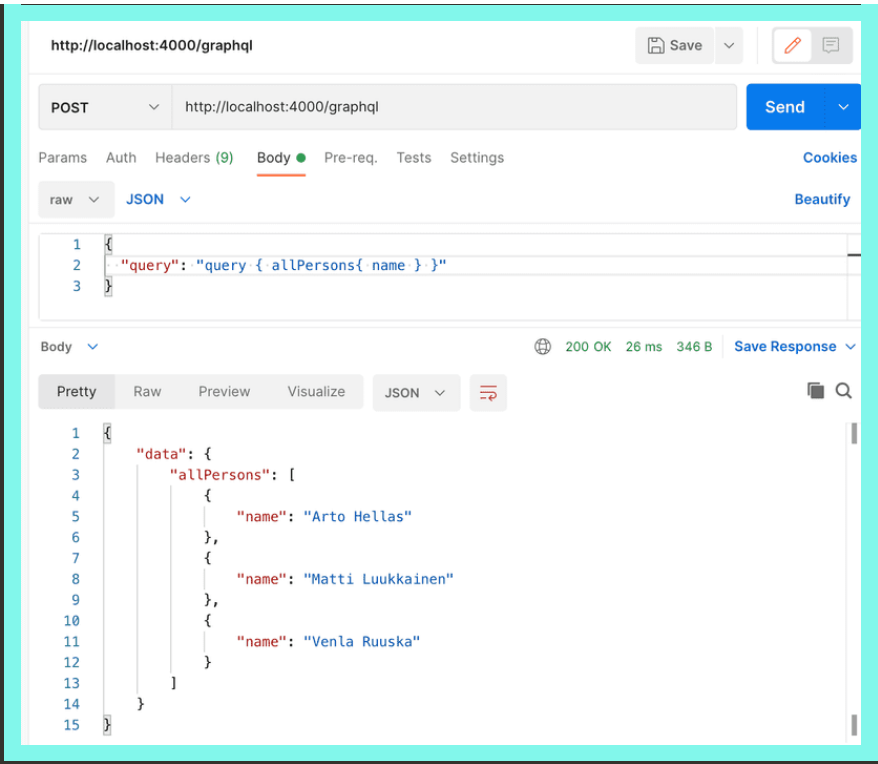
b React and GraphQL

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:

e Fragments and
subscriptions



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's create a new React app and install the necessary dependencies for Apollo client.

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

copy

Replace the default contents of the file *main.jsx* with the following program skeleton:

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import App from './App.jsx'
```

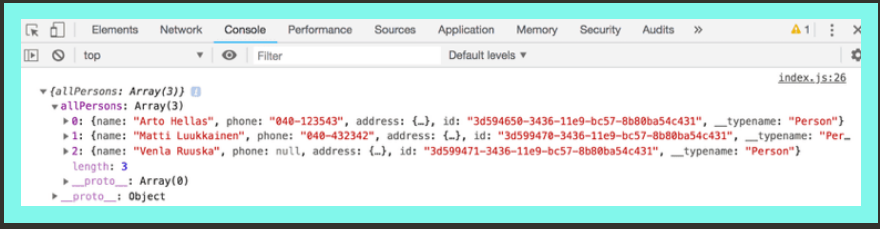
copy

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

const client = new ApolloClient({
  link: new HttpLink({
    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)
})

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

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) => {          copy
  console.log(response.data)
})
```

The server's response is printed to the console:



A `gql` tag is added before the template literal that forms the query, imported from the @apollo/client package:

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

// ...

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

Thanks to the tag, VS Code's GraphQL extension and other tooling recognize the definition as GraphQL, enabling features like syntax highlighting in the editor. On the server side, we achieved the same by adding a type-indicating comment before the template literal, because the @apollo/server library used on the server does not include a corresponding `gql` tag.

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 { StrictMode } from 'react'                    copy
import { createRoot } from 'react-dom/client'
import App from './App.jsx'

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

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

// ...

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

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 } from '@apollo/client'                    copy
import { useQuery } from '@apollo/client/react'

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

Separate the display of persons into its own component in the file *src/components/Persons.jsx*:

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

export default Persons
```

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

```
import { gql } from '@apollo/client'                     copy
import { useQuery } from '@apollo/client/react'
import Persons from './components/Persons'

// ...

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

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

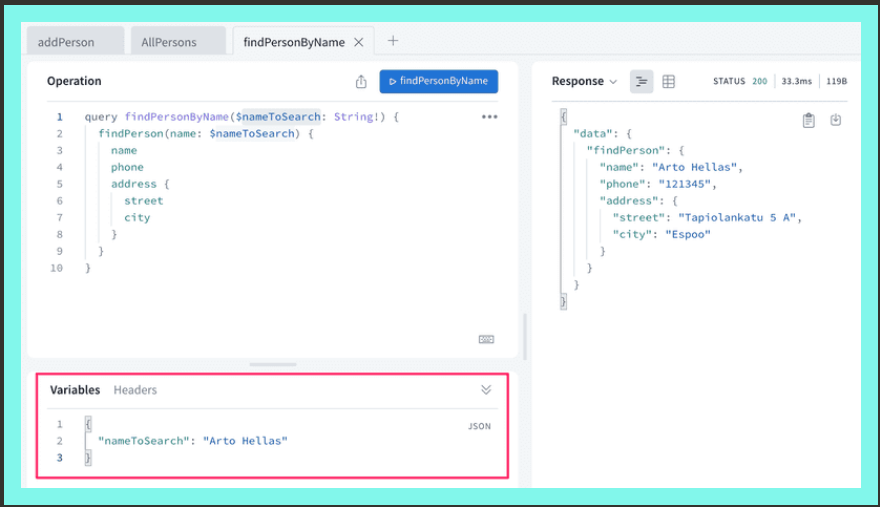
GraphQL variables are well-suited for this. To be able to use variables, we must also name our queries.

A good format for the query is this:

```
query findPersonByName ($nameToSearch: String!) {          copy
  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.

After the changes, the file *Persons.jsx* looks as follows:

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

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>
```

copy

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,
})
```

copy

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.

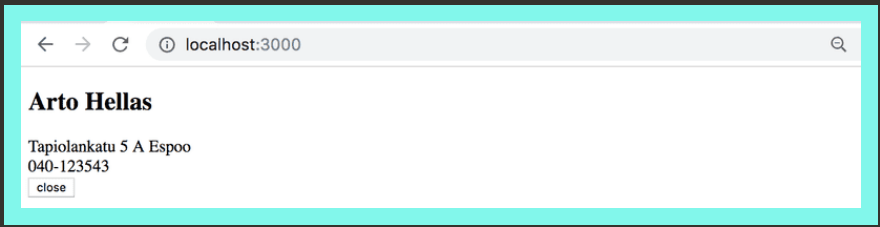
If the state *nameToSearch* has a value and the query result is ready, the component *Person* renders the detailed info of a person:

```
if (nameToSearch && result.data) {
  return (
```

copy

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

A single-person view looks like this:

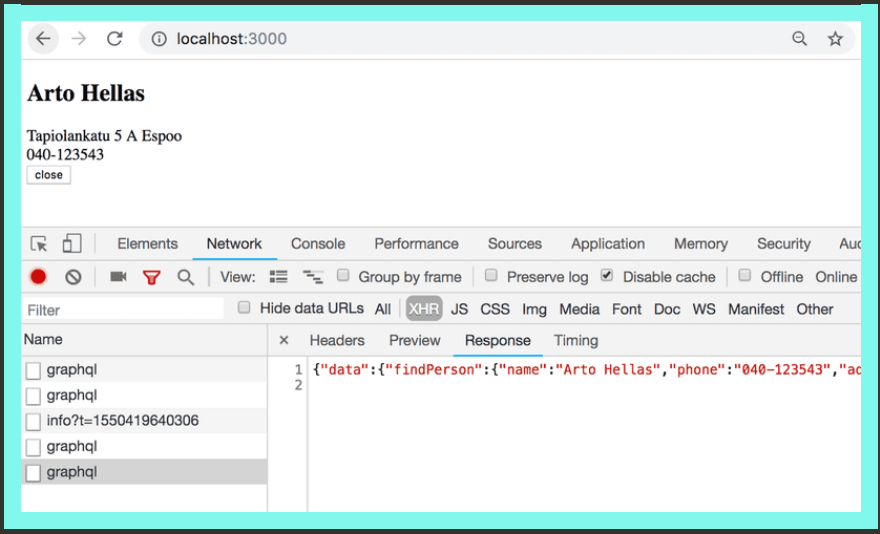


When a user wants to return to the person list, the `nameToSearch` state is set to `null`.

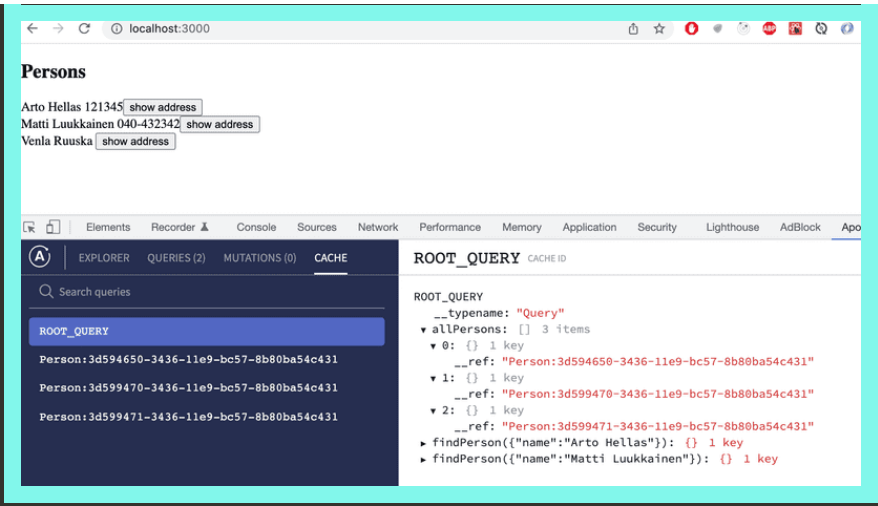
The current code of the application can be found on GitHub branch *part8-1*.

Cache

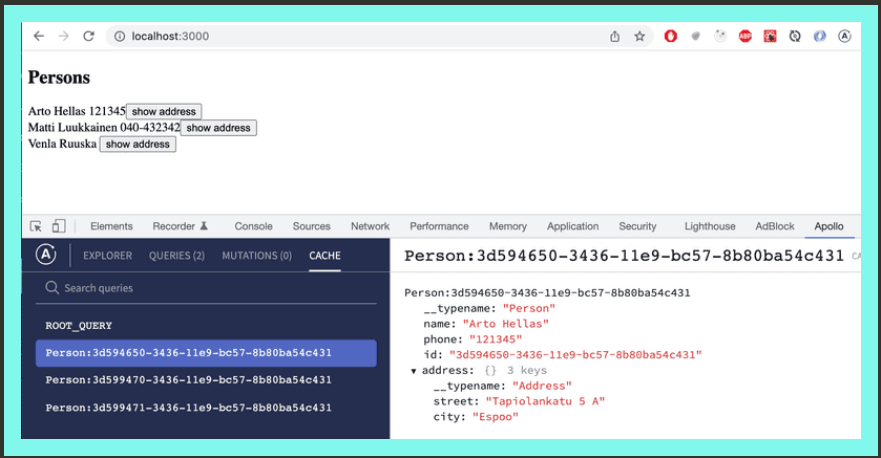
When we do multiple queries, for example with the address details of Arto Hellas, we notice something interesting: the query to the backend is done only the first time around. After this, despite the same query being done again by the code, the query is not sent to the backend.



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`                                    copy
  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.

Create a new component *PersonForm* for adding a new person to the application. The contents of the file *src/components/PersonForm.jsx* are as follows:

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

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
      }
    }
  }
`

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)
```

The query variables receive values when the query is made:

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

Enable the *PersonForm* component in the file *App.jsx*:

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

```
import PersonForm from './components/PersonForm'
import Persons from './components/Persons'

// ...

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

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

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

export default App
```

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 = () => {                                    copy
  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 downside of polling is, of course, the unnecessary network traffic it causes. In addition, the page may start to flicker, since the component is re-rendered with each query update and `result.loading` is true for a brief moment—so a *loading...* text flashes on the screen for an instant.

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.

```
// ...                                                    copy

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


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

  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, {   copy
  refetchQueries: [
    { query: ALL_PERSONS },
    { query: OTHER_QUERY },
```

```
    { query: ANOTHER_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 *src/queries.js*:

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

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

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

export 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
      }
    }
  }
`
```

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

If we try to create an invalid person, for example by using a name that already exists in the application, nothing happens. The person is not added to the application, but we also do not receive any error message.

Earlier, we defined a check on the server that prevents adding another person with the same name and throws an error in such a situation. However, the error is not yet handled in the frontend. Using the `onError` option of the `useMutation` hook, it is possible to register an error handler function for mutations.

Let's register an error handler for the mutation. The *PersonForm* component receives a `setError` function as a prop, which is used to set a message indicating the error:

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

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

  // ...
}
```

Create a separate component for the notification in the file *scr/components/Notify.jsx*:

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

```
      {errorMessage}
    </div>
  )
}

export default Notify
```

The component receives a possible error message as a prop. If an error message is set, it is rendered on the screen.

Render the *Notify* component that displays the error message in the file *App.jsx*:

```
import Notify from './components/Notify'

// ...

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>
  )
}
```

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.

The mutation again requires the use of variables. Add the following query to the file *queries.js*:

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

Create a new component *PhoneForm* in the file *src/components/PhoneForm.jsx* for updating a phone number. The component adds a form to the application where you can enter a new phone number for a selected person. The interesting parts of the code are highlighted:

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

The *PhoneForm* component is straightforward: it asks for the person's name and a new phone number via a form. When the form is submitted, it calls the `changeNumber` function that handles the update, created with the `useMutation` hook.

Enable the new component in the file *App.jsx*:

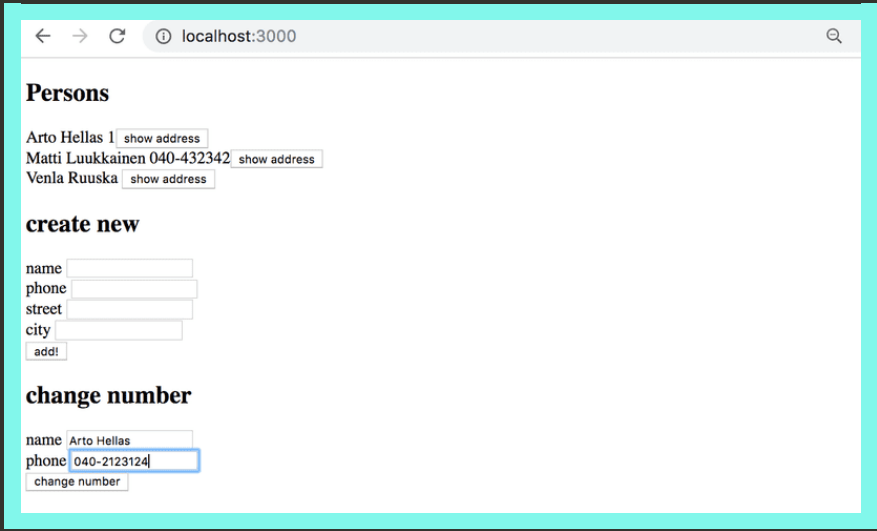
```
import PhoneForm from './components/PhoneForm'
```



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

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

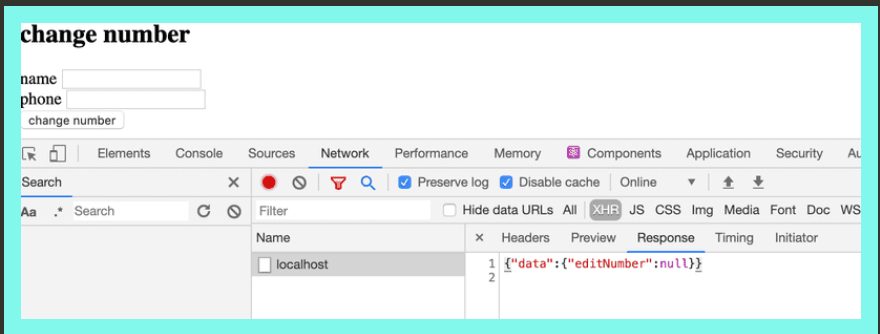
```
  )
}
```

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*:



Since this isn't considered an error state from GraphQL's point of view, registering an `onError` error handler wouldn't be useful in this situation. However, we can add an `onCompleted` callback to the `useMutation` hook, where we can generate a potential error message:

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

```
  const [changeNumber] = useMutation(EDIT_NUMBER, {
    onCompleted: (data) => {
      if (!data.editNumber) {
        setError('person not found')
      }
    }
  })

  // ...
}
```

The `onCompleted` callback function is always executed when the mutation has been successfully completed. If the person wasn't found—that is, if the query result `data.editNumber` is `null`—the component uses the `setError` callback function it received via props to set an appropriate 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 quite a 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

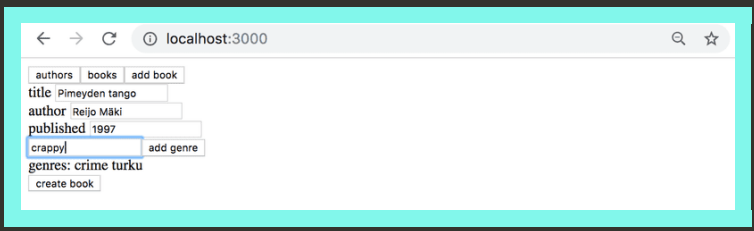
Implement an Authors view to show the details of all authors on a page as follows:

8.9: Books view

Implement a Books view that shows the details of all books except their genres.

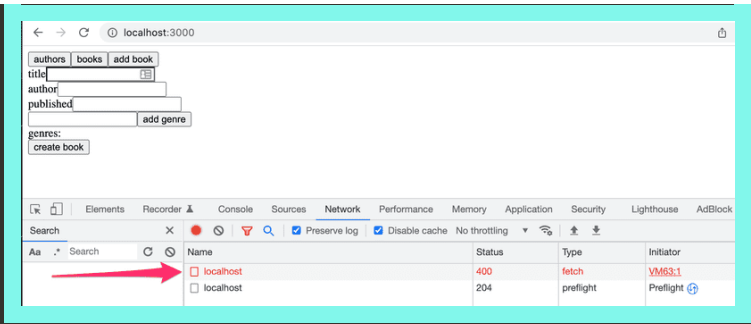
8.10: Adding a book

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:



The Chrome extension Apollo Client Devtools can be very helpful in diagnosing the situation.

8.11: Authors birth year

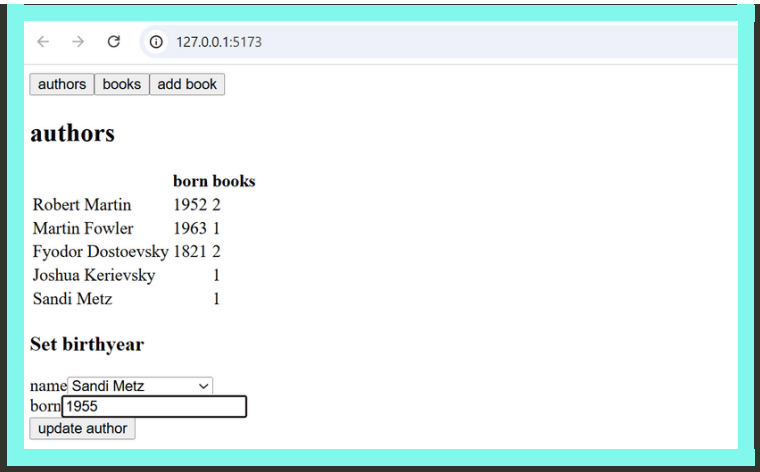
Implement a possibility to set authors birth year. You can create a new view for setting the birth year, or place it on the Authors view:

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

8.12: Authors birth year advanced

Make the birth year form such that the birth year can be set via a dropdown only for an existing author. You can use, for example, the select element or a separate library like react-select.

The solution looks as follows using a *select* element:



Propose changes to material

Part 8a
Previous part

Part 8c
Next part