d Login and updating the cache

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

User log in

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

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

  // ...

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

  return (
    // ...
  )
}
```

Next we define a mutation for logging in

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

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

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

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

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

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

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

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

  return (
    <div>
      <form onSubmit={submit}>
        <div>
          username <input
            value={username}
            onChange={({ target }) => setUsername(target.value)}
          />
```

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

export default LoginForm
```

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

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

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

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

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

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

}
```

The current code of the application can be found on Github, branch *part8-6*.

Adding a token to a header

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

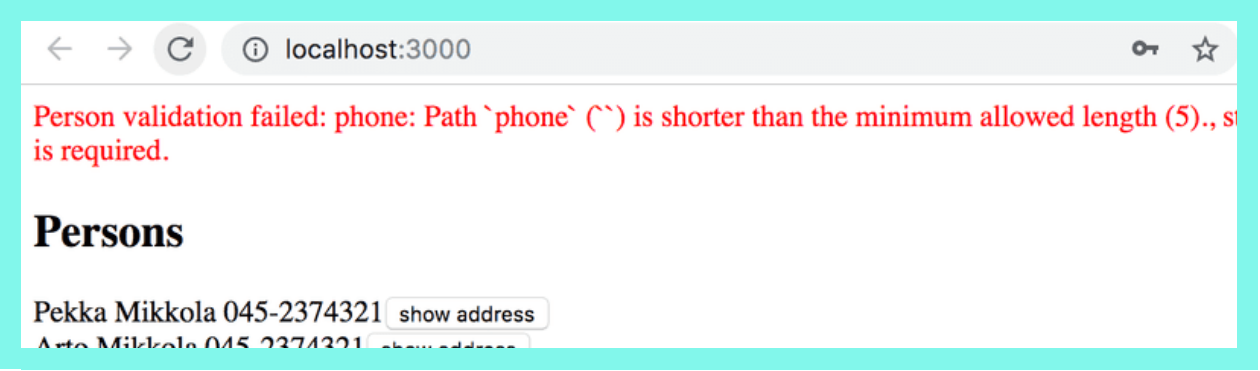
```
import { setContext } from 'apollo-link-context'

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

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

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

The link parameter given to the `client` -object defines how apollo connects to the server. Here the normal httpLink connection is modified so that the request's *authorization* header contains the token if one has been saved to the localStorage.

We also need to install the library required by this modification

```
npm install apollo-link-context
```

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



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

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

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

  // ...
  }

  // ...
}
```

Current application code can be found on Github, branch *part8-7*.

Updating cache, revisited

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

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

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

```
  })
```

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

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

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

  const [ createPerson ] = useMutation(CREATE_PERSON, {
    onError: (error) => {
      setError(error.graphQLErrors[0].message)
    },
    update: (store, response) => {
      const dataInStore = store.readQuery({ query: ALL_PERSONS })
      store.writeQuery({
        query: ALL_PERSONS,
        data: {
          ...dataInStore,
          allPersons: [ ...dataInStore.allPersons, response.data.addPerson ]
        }
      })
    }
  })

  // ..
}
```

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

The code reads the cached state of `ALL_PERSONS` query using `readQuery` function and updates the cache with `writeQuery` function adding the new person to the cached data.

Note that readQuery will throw an error if your cache does not contain all of the data necessary to fulfill the specified query. This can be solved using a try-catch block.

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

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

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

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

The current code of the application can be found on Github, branch *part8-8*.

Exercises 8.17.-8.22.

8.17 Listing books

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

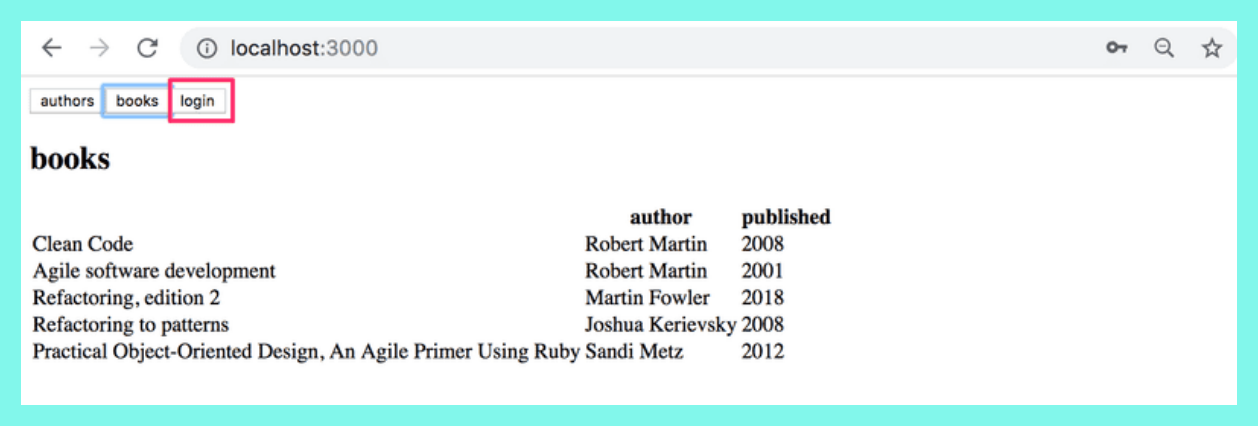
8.18 Log in

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

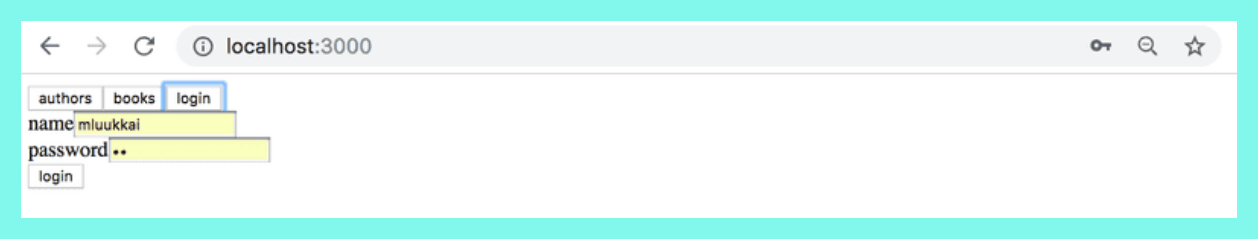
Implement login functionality and fix the mutations.

It is not necessary yet to handle validation errors.

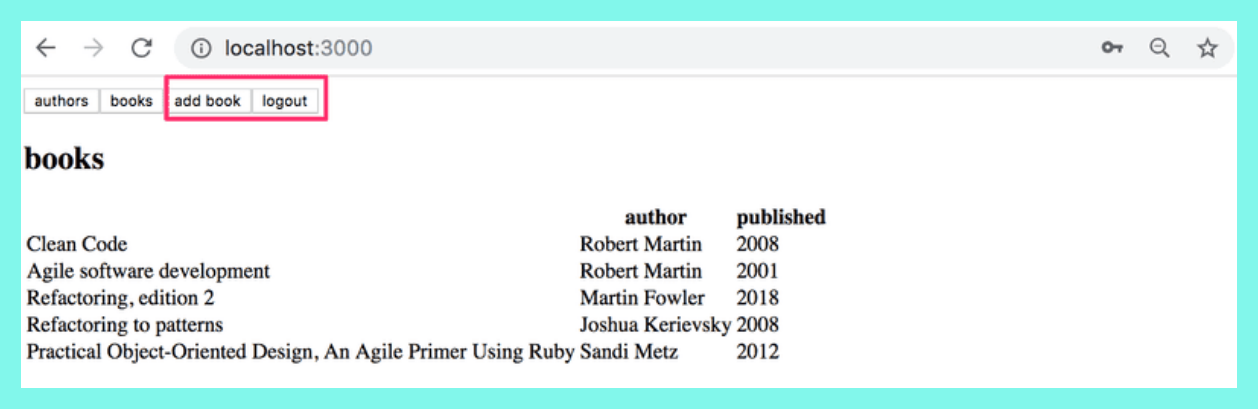
You can decide how the log in looks on the user interface. One possible solution is to make the login form into a separate view which can be accessed through a navigation menu:



The login form:



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



8.19 Books by genre, part 1

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

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

8.20 Books by genre, part 2

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

8.21 books by genre with GraphQL

In the previous exercise 8.20, the filtering could have been done using just React. To complete this exercise, you should filter the books in the recommendations page using a GraphQL query to the server. The query created in exercise 8.5 could be useful here.

This and the next exercises are quite challenging like it should be this late in the course. You

might want to complete first the easier ones in next part.

Some tips

- Instead of using *useQuery* it is probably better to do the queries with the *useLazyQuery*-hook
- It is sometimes useful to save the results of a GraphQL query to the state of a component.
- Note, that you can do GraphQL queries in a *useEffect*-hook.
- The second parameter of a *useEffect* - hook can become handy depending on your approach.

8.22 Up to date cache and book recommendations

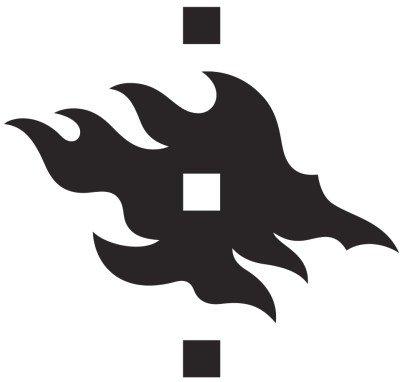
If you did the previous exercise, that is fetch the books in a genre with GraphQL, ensure somehow that the books view is kept up to date. So when a new book is added, the books view is updated at least when a genre selection button is pressed.

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

About course
Course contents
FAQ
Partners
Challenge

Propose changes to material

Previous part Part 8c
Next part Part 8e



UNIVERSITY OF HELSINKI


HOUSTON