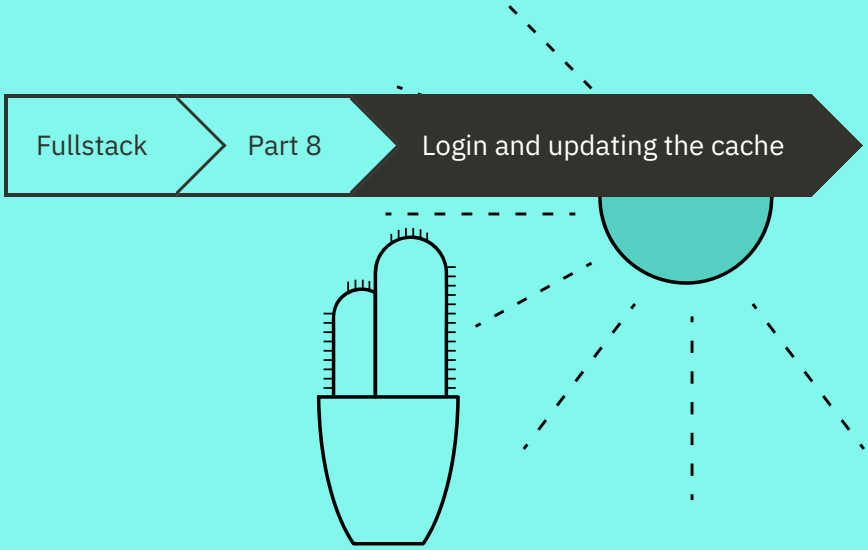


d Login and updating the cache

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 login

Let's add the variable `token` to the application's state. When a user is logged in, it will contain a user token. 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 = () => {                                    copy
  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`                                    copy
  mutation login($username: String!, $password: String!)
{
    login(username: $username, password: $password)  {
      value
    }
  }
`
```

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

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

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

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

  const submit = async (event) => {
    event.preventDefault()
    const result = await login({ variables: { username,
password } })
    if (result.data) {
      const token = result.data.login.value
      setToken(token)
      localStorage.setItem('phonenumbers-user-token',
token)
    }
  }
```

```
  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
```

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 = () => {                                    copy
  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()
  }

  if (!token) {
    return (
      <>
```

```
        <Notify errorMessage={errorMessage} />
        <LoginForm setToken={setToken} setError={notify}
/>
      </>
    )
  }

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

Adding a token to a header

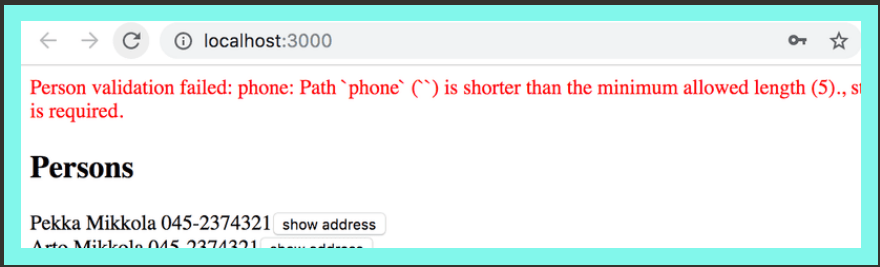
After the backend changes, creating new persons requires that a valid user token is sent with the request. In order to send the token, we have to change the way we define the `ApolloClient` object in *main.jsx* a little.

```
import { ApolloClient, InMemoryCache, HttpLink, } from
'@apollo/client'
import { ApolloProvider } from '@apollo/client/react'
import { SetContextLink } from
'@apollo/client/link/context'

const authLink = new SetContextLink((preContext) => {
  const token = localStorage.getItem('phonenumbers-user-
token')
  return {
    headers: {
      ...preContext.headers,
      authorization: token ? `Bearer ${token}` : '',
    },
  }
})

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

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

The field `uri` that was previously used when creating the `client` object has been replaced by the field `link`, which defines in a more complicated case how Apollo is connected to the server. The server url is now wrapped using the function createHttpLink into a

suitable httpLink object. The link is modified by the context defined by the authLink object so that a possible token in localStorage is set to header *authorization* for each request to the server.

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 `undefined` if user has not given a value.

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

  // ...
  }

  // ...
}
```

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 }) => {                    copy
  // ...

  const [ createPerson ] = useMutation(CREATE_PERSON, {
    refetchQueries: [  {query: ALL_PERSONS} ],
    onError: (error) => {
      const messages = error.graphQLErrors.map(e =>
e.message).join('\n')
```

```
      setError(messages)
    }
  })
```

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 }) => {                    copy
  // ...

  const [ createPerson ] = useMutation(CREATE_PERSON, {
    onError: (error) => {
      const messages = error.graphQLErrors.map(e =>
e.message).join('\n')
      setError(messages)
    },
    update: (cache, response) => {
      cache.updateQuery({ query: ALL_PERSONS }, ({
allPersons }) => {
        return {
          allPersons:
allPersons.concat(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.

Using the function updateQuery the code updates the query ALLPERSONS in the cache by adding the new person to the cached data.

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 the 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 *part8-5*.

Exercises 8.17.-8.22

8.17 Listing books

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

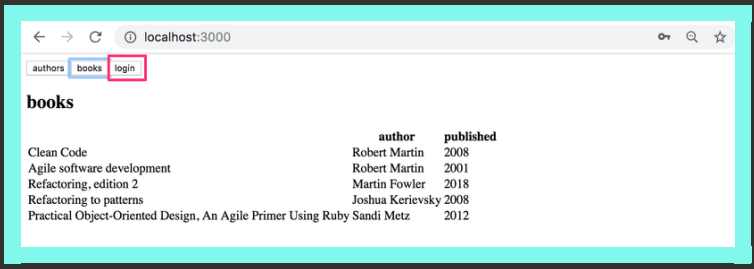
8.18 Log in

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

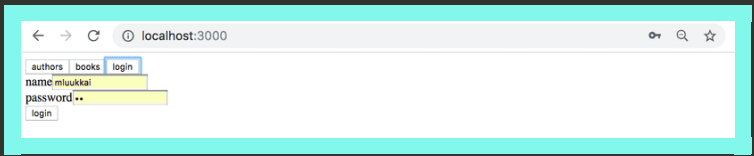
Implement login functionality and fix the mutations.

It is not necessary yet to handle validation errors.

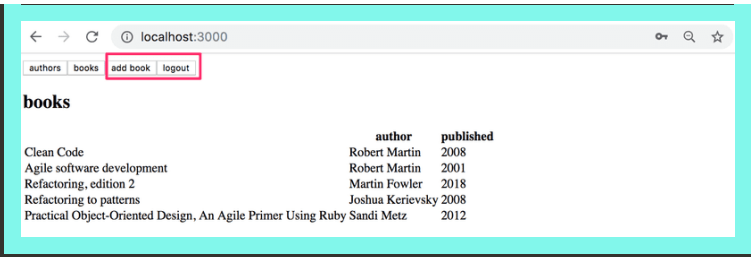
You can decide how the login looks on the user interface. One possible solution is to make the login form into a separate view which can be accessed through a navigation menu:



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:



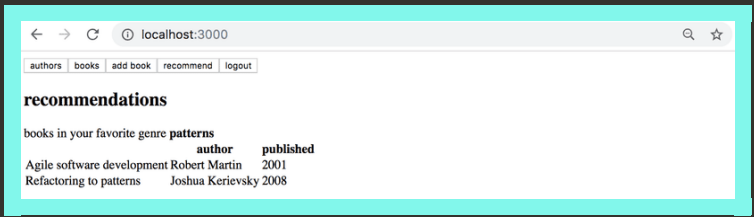
8.19 Books by genre, part 1

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 two exercises, the filtering could have been done using just React. To complete this exercise, you should redo the filtering of the books based on a selected genre (that was done in exercise 8.19) using a GraphQL query to the server. If you already did so then you do not have to do anything.

This and the next exercise are quite **challenging**, like they should be this late in the course. It may help you to complete the easier exercises in the next part before doing 8.21 and 8.22.

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.

Propose changes to material

Part 8c
Previous part

Part 8e
Next part