

```
{() => fs}
```

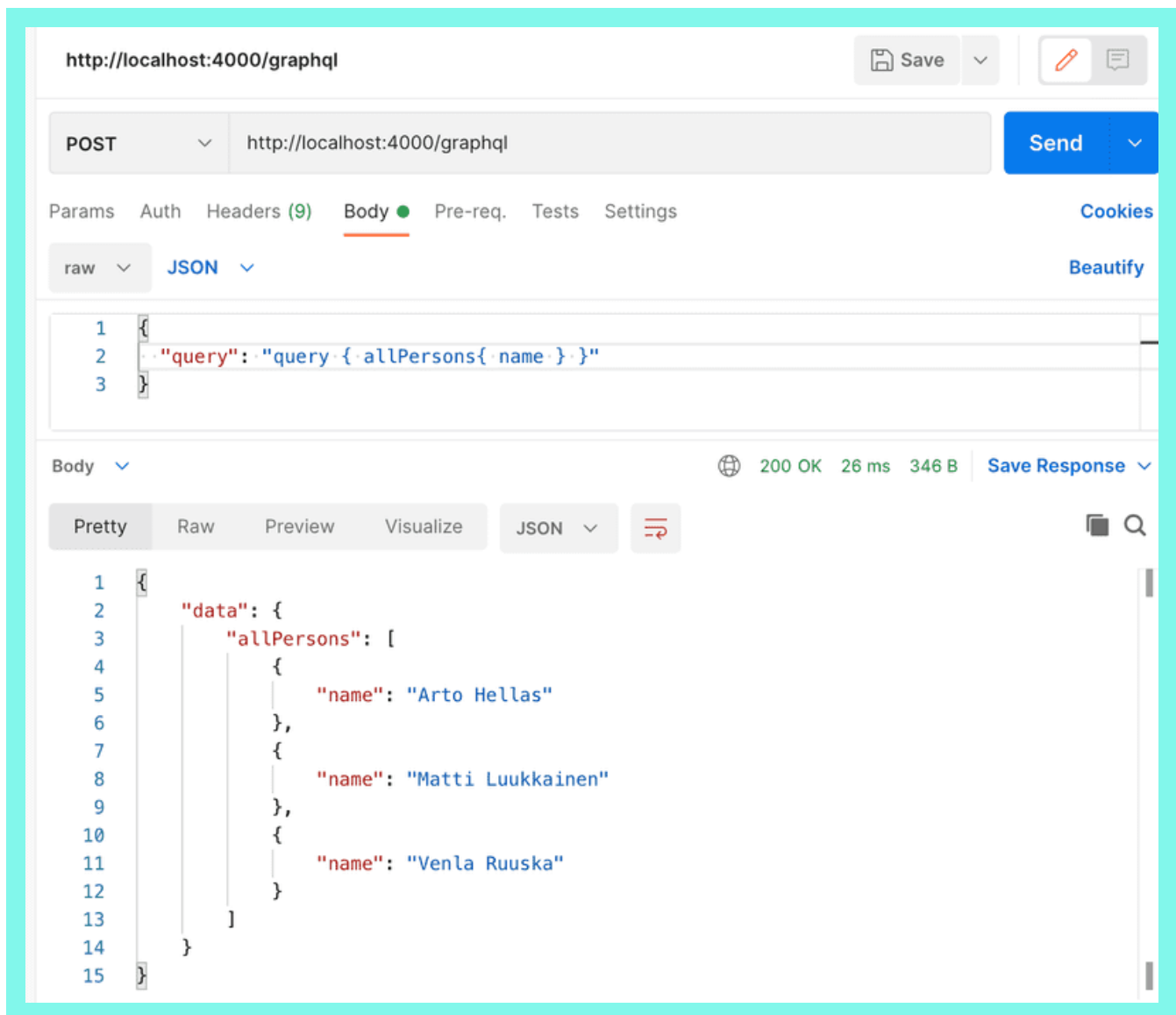


## b React and GraphQL

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:



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 us create a new React app, and can continue installing dependencies required by [Apollo client](#).

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

[copy](#)

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

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

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

const client = new ApolloClient({
  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)
  })

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

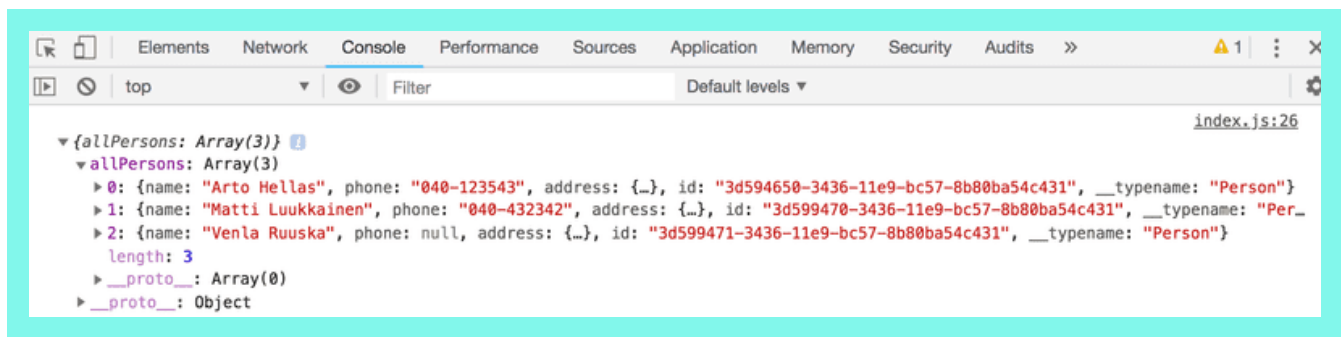
[copy](#)

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)
  })
```

[copy](#)

The server's response is printed to the console:



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

copy

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

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

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

## 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, useQuery } from '@apollo/client'

const ALL_PERSONS = gql`
query {
```

copy

```

    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>
}

```

copy

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

```

copy

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}>

```

copy

```

      {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}/>
  )
}

```

copy

## 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
  }
}

```

copy

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
  }
}

```

copy

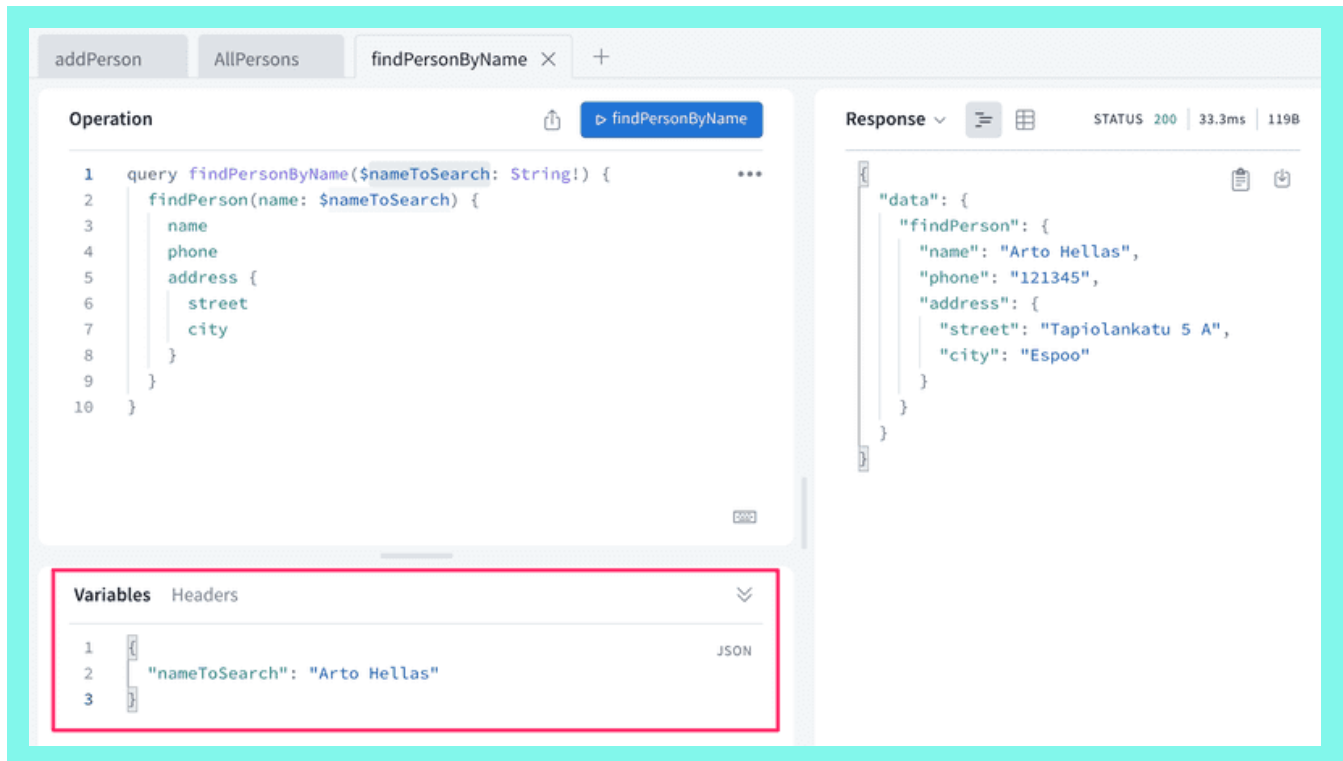
```

    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.

The solution is as follows:

```

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

const FIND_PERSON = gql`
  query findPersonByName($nameToSearch: String!) {

```

copy

```

    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](#)

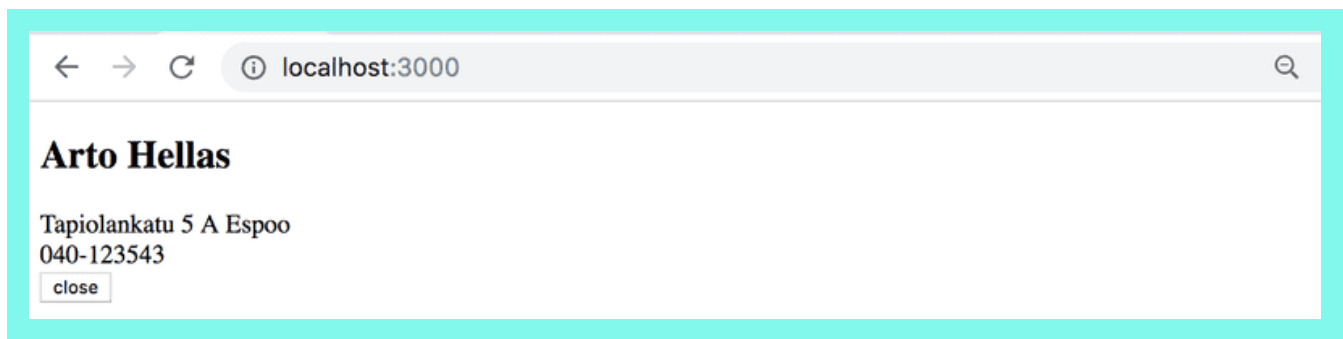
When the user is not interested in seeing the detailed info of any person, the state variable *nameToSearch* is null and the query is not executed.

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

[copy](#)

A single-person view looks like this:

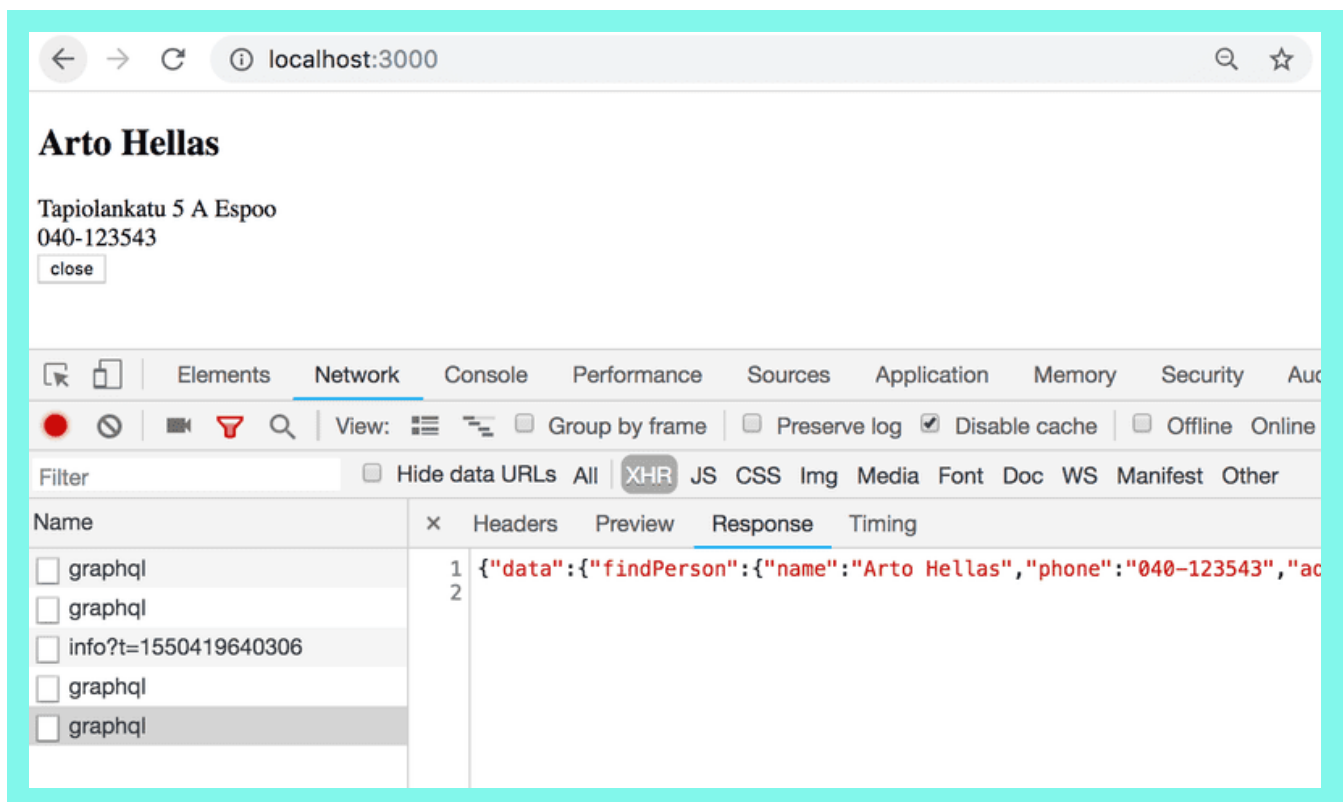


When a user wants to return to the person list, the `nameToSearch` state is set to `null`.

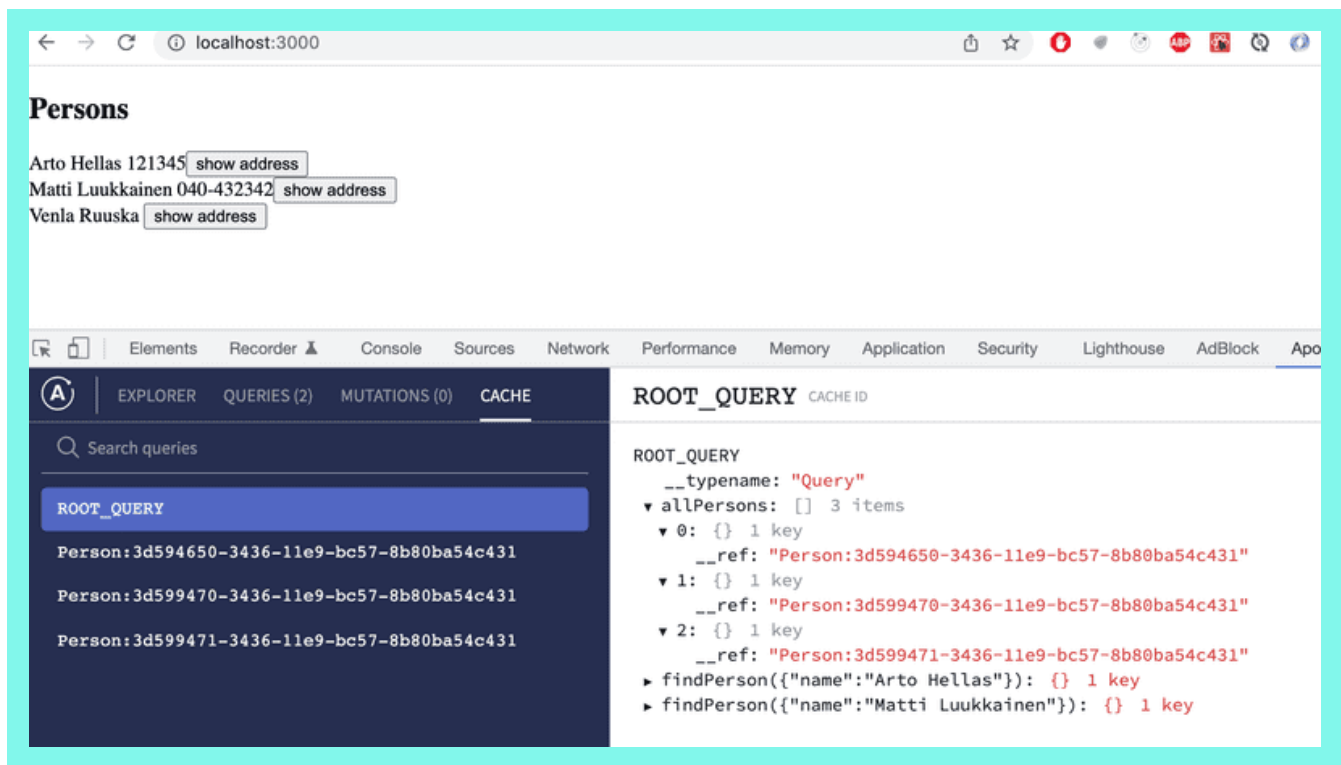
The current code of the application can be found on [GitHub](#) branch *part8-1*.

## Cache

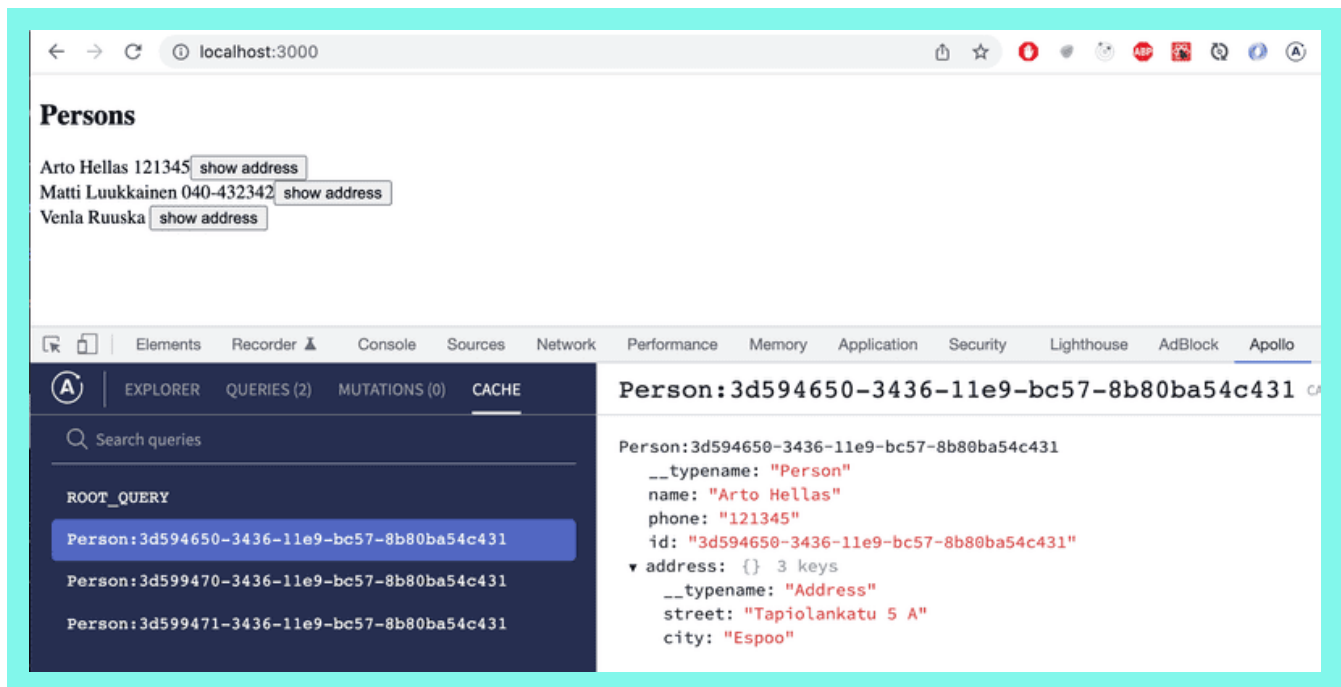
When we do multiple queries, for example with the address details of Arto Hellas, we notice something interesting: the query to the backend is done only the first time around. After this, despite the same query being done again by the code, the query is not sent to the backend.



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`
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
    }
  }
}
`
```

copy

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

const CREATE_PERSON = gql`
  // ...
`
```

copy

```
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)
```

[copy](#)

The query variables receive values when the query is made:

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

[copy](#)

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 = () => {
  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
```

[copy](#)

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, {
```

[copy](#)

```
    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, {
  refetchQueries: [ { query: ALL_PERSONS }, { query: OTHER_QUERY }, { query: ... } ] //
pass as many queries as you need
})
```

copy

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 `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) {
    // ...
  }
`
```

copy

Each component then imports the queries it needs:

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

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

copy

```
// ...
}
```

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:



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

Let's register the mutation with an error handler that 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) => {
      const messages = error.graphQLErrors.map(e => e.message).join('\n')
      setError(messages)
    }
  })

  // ...
}
```

copy



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>
  )
}
```

[copy](#)

Now the user is informed about an error with a simple notification.

Name must be unique

## Persons

Arto Hellas 040-123543 show address

Matti Luukkainen 040-432342 show address

Venla Ruuska show address

show address

name

phone

street

city

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.

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
    }
  }
`
```

copy

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

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

copy

```

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:

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*:

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.

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 a 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 application's local state to [Apollo cache](#).

### Exercises 8.8.-8.12

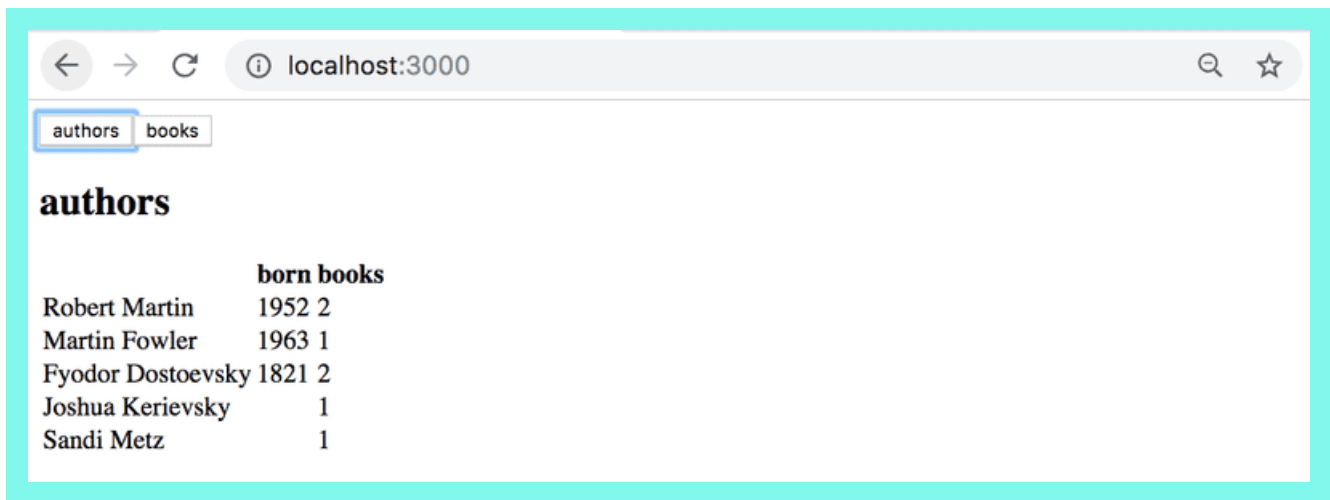
Through these exercises, we'll implement a frontend for the GraphQL library.

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 to show on a page all other 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:

### 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:



	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

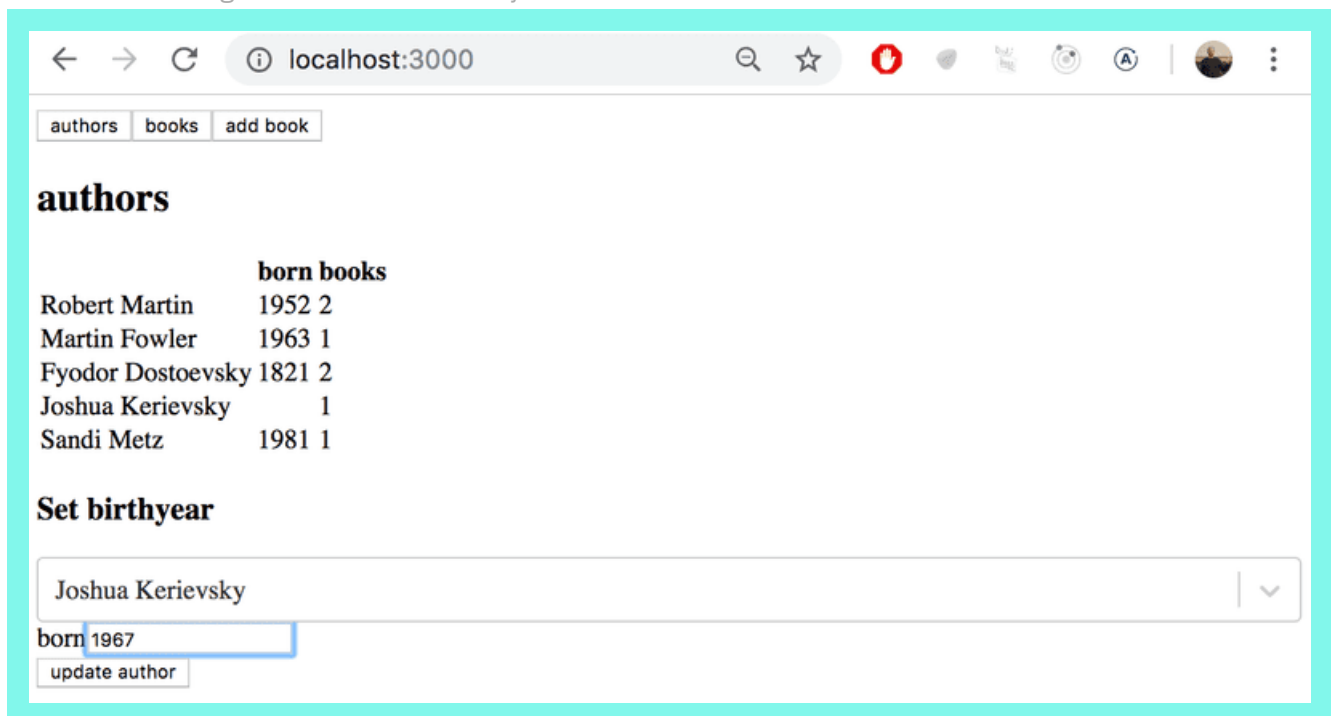
update author

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

### 8.12: Authors birth year advanced

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

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



	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

update author



[Propose changes to material](#)

Part 8a  
**Previous part**

Part 8c  
**Next part**

About course

Course contents

FAQ

Partners

Challenge



UNIVERSITY OF HELSINKI

