Period 3: Learning goals

Explain shortly about GraphQL, its purpose and some of its use cases

- GraphQL er et open source-dataforespørgsel og manipulationssprog til API'er og runtime til udfyldelse af forespørgsler med eksisterende data.
- GraphQL blev udviklet af Facebook
- Bruges til at køre queryes, mutations (CRUD-operationer) på en database
- Erstatter REST api hvor der kan undgås over/under-fetching. Alt data kan hentes og man vælger selv hvad man vil se efter en query fx

```
query{
   allFriends{
     id
     firstName
     lastName
     gender
     email
     age
   }
    {id firstName lastName}
 }
```

Explain some of the Server Architectures that can be implemented with a GraphQL backend

What is meant by the terms over- and under-fetching in GraphQL, compared to REST

Over-fetching: Hente for meget data fra et REST-endpoint

Under-fetching: Hente for lidt data fra et REST-endpoint

Med graphQl vælger man selv hvilke værdier man vil have retur i en query. Derved kan man ungå at have spilddata som man alligevel ikke benytter.

Explain shortly about GraphQL’s type system and some of the benefits we get from this

- Alle graphQL services definerer hvilke typer data som kan queries fra.

```
type Character {
  name: String!
  appearsIn: [Episode!]!
}
```

Character : GraphQL object type ⇒ Det er en type med nogle felter

Name and **appersIn** : felter som et character-graphQL objekt kan indeholde

String, Int, Float, Boolean, and **ID** : Built in scalar type

! : Non-nullable af en bestemt type

Episode : Homemade type

- GraphQL er stærkt typebaseret, hvilket betyder at udvikleren på forhånd ved hvilke type data der arbejdes med / skal inputtes etc
- Alle typer der kan kaldes i GraphQL API'et skrives ned i et skema ved brug af GraphQL Schema Definition Language (SDL). Skemaet er en kontrakt imellem client og server.

Explain shortly about GraphQL Schema Definition Language, and provide examples of schemas you have defined.

- Typestærkt ⇒ Alle typer er defineret på forhånd

▼ Eksempel på et GraphQL skema

```
type Friend {
        id: ID
        firstName: String
        lastName: String
        gender: Gender
        language: String
        age: Int
        email: String
        contacts: [Contact]
    }
    type Contact {
        firstName: String
        lastName: String
    }
    enum Gender {
        MALE
        FEMALE
        OTHER
    }
    type Query {
        getFriend(id: ID): Friend
        allFriends : [Friend]!
    }
    input FriendInput {
        id: ID
        firstName: String
        lastName: String
        gender: Gender
        language: String
        age: Int
        email: String
        contacts: [ContactInput]
    }
    input ContactInput {
        firstName: String
        lastName: String
    }
    type Mutation {
        createFriend(input: FriendInput): Friend
        updateFriend(input: FriendInput): Friend
        deleteFriend(id: ID!): String
    }
```

- Et skema som indeholder Queries og mutations (til CRUD-operationer).
- Indeholder typen Friend og Contact som har sine egne felter.
- To slags input felter som benyttes til at kunne definere hvilken slags input der kan modtages i mutations ⇒ Det fremgår i parantesen i mutation-metoderne

Provide examples demonstrating data fetching with GraphQL. You should provide examples both running in a Sandbox/playground and examples executed in an Apollo Client

Se mappen "apollo-client-for-lynda-server-main" i filerne i components mappen fx: app.tsx, allFriends.tsx

Provide a number of examples demonstrating; creating, updating and deleting with Mutations. You should provide examples both running in a Sandbox/playground and examples executed in an Apollo Client.

Se mappen "apollo-client-for-lynda-server-main" i filerne i components mappen fx: app.tsx, addFriend.tsx, FindFriend.tsx med apollo eller se startkoden for execution i en apollo client.

▼ Schemas for mutations

```
input FriendInput {
        id: ID
        firstName: String
        lastName: String
        gender: Gender
        language: String
        age: Int
        email: String
        contacts: [ContactInput]
    }

type Mutation {
        createFriend(input: FriendInput): Friend
        updateFriend(input: FriendInput): Friend
        deleteFriend(id: ID!): String
    }
```

▼ Mutations

```
Mutation: {
    createFriend: (root, { input }) => {
      const newFriend = new Friends({
        firstName: input.firstName,
        lastName: input.lastName,
        gender: input.gender,
        language: input.language,
        age: input.age,
        email: input.email,
        contacts: input.contacts,
      });
      newFriend.id = newFriend._id;
      return newFriend.save();
    },
    updateFriend: (root, { input }) => {
      return Friends.findOneAndUpdate({ _id: input.id }, input, { new: true });
    },
    deleteFriend: async (root, { id }) => {
      const res = await Friends.deleteOne({ _id: id });
      if (res.deletedCount === 1) {
        return "Succesfully deleted a friend";
      }
      throw new Error("Could not delete a friend with the provided id");
    },
  },
```

Explain the Concept of a Resolver function, and provide a number of simple examples of resolvers you have implemented in a GraphQL Server.

Resolvers er funktioner som benyttes til at udføre queries, mutations etc i fra GraphQL i JS/TS - kode.

Resolvers er field-functions som medtager et parent-objekt, argumenter og execution contexts (fx at det er en query og hvilken metode den så skal udføre).

Resolvers er ansvarlige for at returnerere et resultat for et felt

▼ Resolvers.js fra GraphQL_test

```
import { Friends } from "./dbConnectors";

// resolver map
export const resolvers = {
  Query: {
    getFriend: (_, { id }) => {
      return Friends.findById(id);
    },
    allFriends: () => {
      return Friends.find({});
    },
  },
  Mutation: {
    createFriend: (root, { input }) => {
      const newFriend = new Friends({
        firstName: input.firstName,
        lastName: input.lastName,
        gender: input.gender,
        language: input.language,
        age: input.age,
        email: input.email,
        contacts: input.contacts,
      });
      newFriend.id = newFriend._id;
      return newFriend.save();
    },
    updateFriend: (root, { input }) => {
      return Friends.findOneAndUpdate({ _id: input.id }, input, { new: true });
    },
    deleteFriend: async (root, { id }) => {
      const res = await Friends.deleteOne({ _id: id });
      if (res.deletedCount === 1) {
        return "Succesfully deleted a friend";
      }
      throw new Error("Could not delete a friend with the provided id");
    },
  },
};
```

Explain the benefits we get from using a library like Apollo-client, compared to using the plain fetch-API

Apollo-client bruges til local-state-management og sætter apollo-clienten et globalt scope (i fx en REACT-router)

▼ Eksempel på react router med apollo-provider

```
<ApolloProvider client={client}>
        <div className="content">
          <Switch>
            <Route exact path="/">
              <Home />
            </Route>
            <Route path="/allFriends">
              <AllFriends />
            </Route>
            <Route path="/findFriend">
              <FindFriend />
            </Route>
            <Route path="/addFriend">
              <AddFriend />
            </Route>
          </Switch>
        </div>
 </ApolloProvider>
```

Apollo-clienten kan så tilgå GraphQL queries/etc fra alle steder i koden.

Man undgår fx at skulle lifting-state-up

Ved plain fetch-api er det mere kompliceret at holde styr på state og hvilket scope der arbejdes på.

In an Apollo-based React Component, demonstrate how to perform GraphQL Queries, including:

Explain the purpose of ApolloClient and the ApolloProvider component

Apollo-client bruges til local-state-management og sætter apollo-clienten et globalt scope (i fx en REACT-router)

▼ Eksempel på react router med apollo-provider

```
<ApolloProvider client={client}>
        <div className="content">
          <Switch>
            <Route exact path="/">
              <Home />
            </Route>
            <Route path="/allFriends">
              <AllFriends />
            </Route>
            <Route path="/findFriend">
              <FindFriend />
            </Route>
            <Route path="/addFriend">
              <AddFriend />
            </Route>

          </Switch>
        </div>
 </ApolloProvider>
```

Apollo-clienten kan så tilgå GraphQL queries/etc fra alle steder i koden.

Explain the purpose of the gql-function (imported from graphql-tag)

Gql : Query language

Gql-function bruges til at parse elementer til et query-document

▼ Eksempel på gql-function

```
import { gql } from "@apollo/client"

const ALL_FRIENDS = gql`
 query{
   allFriends{
     id
     firstName
     lastName
     gender
     email
     age
   }
 }
`
```

Explain Custom Hooks used by your Client Code

Explain and demonstrate the caching features built into Apollo Client

In an Apollo-based React Component, demonstrate how to perform GraphQL Mutations?

Demonstrate and highlight important parts of a "complete" GraphQL-app using Express and MongoDB on the server-side, and Apollo-Client on the client.