**Period 3 GraphQL**

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

GraphQL er et query language, som giver clienten muligheden for, at spørge efter præcis den information, de har brug for, hverken mere eller mindre. Du får ikke et helt object fyldt med felter, men kun det felt, der rent faktisk er brug for.

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

GraphQL Server med en Connected Database: Her er en webserver, der implementerer grapql specifikationene. Når en query modtages af GraphQL serveren, læser den hvad query’en indeholder og fethcer den information, der efterspørges, i databasen. Det kaldes at resolve query’en. Responses konstrueres som et objekt og returneres til klienten.

GrapthQL Server integrated with the Existing System: Integration af multiple systemer bag en enkel GraphQL API. Specielt egnet til firmaer med et system bestående af mange API’er. GraphQL kan bruges til at forene disse systemer og gemme kompleksiteten væk bag et enkel GrapQL API.

Hybrid Approach with a Connected Database and integrated systems

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

I REST henter man det hele ned, f.eks. alle informationer om en bruger. Dvs., at der downloades ret meget mere, end der egentlig er brug for.

Med GraphQL kan man nøjes med lige præcis det, man skal bruge, f.eks. kun navn.

Underfetching er, når der ikke hentes nok data ned i et enkelt api request. Der skal så foretages adskillige requests.

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

Typesystemet definerer datatyper, der kan bruge i applikationen. Skemaet er en kontrakt mellem client og server.

Query er en forespørgsel efter en bestemt type data, f.eks. getOneFriend, getAllFriends osv.

Mutation er CRUD-operationer som createFriend, updateFriend og deleteFriend.

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

Skemaer bruges til at modellere dataen.

SDL er Schema Definition Language. Hvis det starter med ”type”, indeholder det en objekt-type, hvis det hedder mutation, er det en CRUD, og hvis det hedder ”query” er det en get request.

Skemaet er bygget op, så man får præcis den data retur, som man efterspørger.

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

export const resolvers = {

    Query: {

        getFriend: (\_, { id }) => {

            return Friends.findById(id);

         },

         allFriends: () => {

           return Friends.find({})

         }

    },

interface IFriends {

  allFriends: ILyndaFriend[]

}

export const ALL\_FRIENDS = gql`

query {

  allFriends

  {

    id

    firstName

    lastName

    gender

    email

    age

  }

}

`

export default function All() {

  const {loading, error, data, refetch} = useQuery<IFriends>(

    ALL\_FRIENDS,

    { fetchPolicy: "cache-and-network"

    }

    )

    if (loading) return <p>loading...</p>

    if (error) return <p>{error.toString()}</p>

  return (

    <div>

      <table className="table">

        <thead>

          <tr><th>ID</th><th>First Name</th><th>Last Name</th><th>Gender</th><th> Email</th></tr>

        </thead>

        <tbody>

          {data && data.allFriends.map(f => (

            <tr key={f.id}><td>{f.id}</td><td>{f.firstName}</td><td>{f.lastName}</td><td>{f.gender}</td><td>{f.email}</td></tr>

          ))}

        </tbody>

      </table>

      <button onClick={()=>refetch()}>Refetch</button>

    </div>

  )

}

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.

Create a friend mutation:

 Mutation: {

    createFriend: async (\_: object, { input }: { input: IFriend }) => {

      return friendFacade.addFriendV2(input)

    }

  },

Mutation: {

        createFriend: (root, { input }) => {

            const newFriend = new Friends({

                firstName: input.firstName,

                lastName: input.lastName,

                gender: input.gender,

                email: input.email,

                language: input.language,

                age: input.age,

                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 "Friend deleted"

            }

            throw new Error("Could not delete a friend with that id")

          }

        },

Create a friend with Apollo:

const ADD\_FRIEND = gql`

  mutation createFriend($friend: FriendInput){

    createFriend(input:$friend){

      firstName

      lastName

      email

      age

      gender

      id

    }

  }`

type AddFriendProps = {

  initialFriend?: ILyndaFriend

}

interface IKeyableFriend extends ILyndaFriend {

  [key: string]: any

}

const AddFriend = ({ initialFriend }: AddFriendProps) => {

  const EMPTY\_FRIEND: ILyndaFriend = { firstName: "", lastName: "", gender: "OTHER", age: -1, email: "" }

  let newFriend = initialFriend ? initialFriend : { ...EMPTY\_FRIEND }

  const [friend, setFriend] = useState({ ...newFriend })

  const [addFriend, {data}] = useMutation(

    ADD\_FRIEND,

    {

      update(cache, {data}) {

        const addedFriend = data.createFriend;

        const d: any = cache.readQuery({ query: ALL\_FRIENDS })

        if (!d) {

          return

        }

        let allFriends = d.allFriends

        cache.writeQuery({

          query: ALL\_FRIENDS,

          data: { allFriends: [...allFriends, addedFriend] }

        })

      }

    }

  )

  const handleChange = (event: any) => {

    const id = event.currentTarget.id;

    let friendToChange: IKeyableFriend = { ...friend }

    if(id==="age"){

      friendToChange[id] = Number(event.currentTarget.value);

    } else {

    friendToChange[id] = event.currentTarget.value;

    }

    setFriend({ ...friendToChange })

  }

  const handleSubmit = (event: React.FormEvent<HTMLFormElement>) => {

    event.preventDefault();

    //alert(JSON.stringify(friend))

    //Todo save friend on servers

    addFriend({

      variables: {friend:{...friend, age:Number(friend.age)}}

    })

    setFriend({ ...EMPTY\_FRIEND })

  }

  return (

    <form onSubmit={handleSubmit}>

      <label>

        FirstName<br />

        <input type="text" id="firstName" value={friend.firstName} onChange={handleChange} />

      </label>

      <br />

      <label>

        LastName <br />

        <input type="text" id="lastName" value={friend.lastName} onChange={handleChange} />

      </label>

      <br />

      <label>

        Gender &nbsp;

          <select id="gender" value={friend.gender} onChange={handleChange}>

          <option value="MALE">Male</option>

          <option value="FEMALE">Female</option>

          <option value="OTHER">Other</option>

        </select>

      </label>

      <br />

      <label>

        Age <br />

        <input type="number" id="age" value={friend.age} onChange={handleChange} />

      </label>

      <br /><br />

      <input type="submit" value="Save Friend" />

    </form>

  );

}

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

Resolver er en samling af funktioner, der genererer responses fra et GraphQL query. Resolvers kan returnere objekter eller bare felter såsom strings, booleans osv.

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

Apollo er en open source JavaScript GraphQL server. Den bruges til at specificere de typer og felter, som skal være tilgængelige i GraphQL.

Derudover er den behjælpelig med at udfylde collectionen af resolvers, der specificerer, hvordan hvert felt befolkes med data.

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

* Explain the purpose of ApolloClient and the ApolloProvider component

export const ALL\_FRIENDS = gql`

query {

  allFriends

  {

    id

    firstName

    lastName

    gender

    email

    age

  }

}

`

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

Gql eksekverer queries og mutationer i frontend.

* Explain Custom Hooks used by your Client Code

Når der benyttes useQuery. Det bruges til at sætte data på en hook, hvor der kan ændres fetchpolicy, errors osv.

* Explain and demonstrate the caching features built into Apollo Client

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

Mutation: {

        createFriend: (root, { input }) => {

            const newFriend = new Friends({

                firstName: input.firstName,

                lastName: input.lastName,

                gender: input.gender,

                email: input.email,

                language: input.language,

                age: input.age,

                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 "Friend deleted"

            }

            throw new Error("Could not delete a friend with that id")

          }

        },

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

Se vedlagte kode.