jamk.fi

GraphGL- ja REST-rajapintojen vertailu

Riku Laajala

Opinnäytetyö
Toukokuu 2019,
Tekniikan ja liikenteen ala
Insinööri (AMK), Tieto- ja viestintätekniikan tutkinto-ohjelma

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Laajala, Riku	Opinnäytetyö, AMK	Toukokuu 2019
	Sivumäärä 54	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
GraphGL- ja REST-rajapintojen vertailu

Tutkinto-ohjelma
Insinööri (AMK), tieto- ja viestintätekniikan tutkinto-ohjelma

Työn ohjaaja(t)
Pasi Manninen, Kari Niemi

Toimeksiantaja(t)
Meiko Oy

Tiivistelmä

Opinnäytetyön tavoitteena oli kartoittaa millaisissa käytännön tilanteissa GraphQL- sekä REST-rajapintojen hyödyt ja haitat tulevat ilmi sekä pohtia, milloin ja minkälaisissa projekteissa GraphQL-teknologiaa kannattaa hyödyntää.

Rajapinnat ovat tärkeä osa nykypäivän web-sovelluksia. Ne mahdollistavan eri sovellusten välisen kommunikoinnin sekä tiedon hakemisen useasta eri lähteestä. Rajapintoja käyttävälle osapuolelle on tärkeää, että ne ovat helppokäyttöisiä, luotettavia, nopeita sekä hyvin dokumentoituja. Rajapintojen kehittäjille on tärkeää, että rajapintaa pystytään muokkaamaan vaivattomasti sekä turvallisesti muuttuvien sovellusvaatimusten seurauksena.

REST-arkkitehtuurimallin mukaisesti toteutetut rajapinnat ovat nykypäivän web-sovelluksissa yleisin toteutustapa. REST-arkkitehtuurimallin haastajaksi on noussut Facebookin kehittämä, vuonna 2012 alkunsa saanut GraphQL-teknologia. GraphQL tarjoaa uudenlaisen tavan hakea tietoa rajapinnoista ja pyrkii tekemään niiden käyttämisestä sekä jatkokehittämisestä helpompaa verrattuna REST-rajapintoihin.

Opinnäytetyön tuloksina oli REST-arkkitehtuurimallin sekä GraphQL-teknologian esittely sekä näiden ominaisuuksien vertailu. Ominaisuuksien vertailun pohjalta kävi ilmi, että GraphQL- sekä REST-rajapinnoilla on omat vahvuutensa. Tämän takia oikean teknologian valinta projekteissa ei ole helppoa. Valinta on järkevintä tehdä käyttötilanteen, aiemman osaamisen sekä vaatimusten mukaan. GraphQL kuitenkin tarjoaa rajapinnan käyttäjälle intuitiivisemman tavan hakea tietoa. Se on hyödyllinen projekteissa, joissa rajapinnan halutaan mukautuvan useiden käyttäjien erilaisiin vaatimuksiin.

Avainsanat (asiasanat)
API, Rajapinnat, GraphQL, REST

Muut tiedot (Salassa pidettävät liitteet)

jamk.fi

Description

Author(s)	Type of publication	Date
Laajala, Riku	Bachelor's thesis	May 2019
		Language of publication: Finnish
	Number of pages 54	Permission for web publication: x

Title of publication

Comparison between GraphGL- and REST-APIs

Degree programme

Information and Communications Technology

Supervisor(s)

Pasi Manninen, Kari Niemi

Assigned by

Meiko Oy

Abstract

The thesis was assigned by Meiko Oy, a software company located in Jyväskylä. The objective was to survey and point out the practical cases where the benefits and disadvantages between GraphQL and REST APIs would be demonstrated. In addition, the aim was to speculate what kind of projects would benefit from choosing GraphQL over REST.

Web APIs are a crucial part of modern web applications. They enable the communication between different applications and allow data to be fetched from multiple different sources. For the API users, it is important that the API is easy to use, reliable, fast and well documented. For the API developers, it is important that the API is easily and safely adaptable to support the continuous change of demands.

The REST architecture style is currently the most common one to build APIs in web applications today. GraphQL, the technology originally developed by Facebook in 2012, has become the new contender to challenge the existing REST APIs. GraphQL provides a new way to fetch data from the API. It strives to make the API development and usability more enjoyable compared to REST APIs.

The thesis resulted in an introduction of the REST architecture style and GraphQL technoogy and a comparison of their features. While comparing GraphQL and REST APIs it was found out that they both have their own strengths, which is why it is impossible to say which one is the best option in any eventuality. However, GraphQL still offers a more intutive way to query the API. It is useful in projects where the API consumers have different usage requirements and needs. In the end, the choice should be made considering the current case, previous knowledge and requirements.

Keywords/tags (subjects)

API, interfaces, GraphQL, REST

Miscellaneous (Confidential information)

Sisältö

Kuviot

Käsitteet

API

Application programming interface. Ohjelmointirajapinta. Ohjelman tai web-sovelluksen tarjoama määritelmä kyseisen ohjelman tai web-sovelluksen käyttömahdollisuuksista. Rajapintojen avulla eri ohjelmat pystyvät keskustelemaan keskenään.

CRUD

Lyhenne sanoista CREATE, READ, UPDATE ja DELETE. Tiedon kanssa operoimiseen käytettävät neljä perustoiminnallisuutta. Käytetään yleensä tietokannan tai rajapinnan toiminnallisuudesta puhuttaessa.

HTTP

Tiedonsiirtoprotokolla. Selaimet sekä palvelimet pystyvät käyttämään HTTP-protokollaa keskinäiseen kommunikointiin.

JavaScript

Ohjelmointikieli. Alun perin, käytettiin ainoastaan selaimissa, nykyään sitä voidaan kuitenkin käyttää ohjelmointiin esim. palvelimilla.

JSON

Avoimen standardin tiedostomuoto mitä voidaan käyttää alustariippumattomaan tiedonvälitykseen.

Node.js

JavaScript-ohjelmointikielen avulla rakennettu, palvelimella suoritettava ajoympäristö.

ORM

Lyhenne sanoista “Object-relational Mapping”. Ohjelmointitekniikka, jonka avulla pystytään mallintamaan esimerkiksi tietokannassa olevaa tietoa olioiden kautta.

SOAP

Simple Access Object Protocol. Viestintäprotokolla mikä mahdollistaa eri järjestelmien keskinäisen kommunikaation.

SQL

Structured query language. Kyselykieli mitä käytetään tiedon hakemiseen sekä tallentamiseen relaatiotietokannoissa.

URI

Uniform Resource Identifier. Merkkijono, joka yksilöi tietyn tiedon paikan. Esimerkiksi verkkosivun osoite (URL) *https://www.jamk.fi/fi/Etusivu/* voidaan luokitella URI:ksi.

Webhook

Käyttäjän määrittelemiä, tiettyjen toimintojen seurauksena automaattisesti suoritettavia HTTP-kutsuja.

WebSocket

Protokolla mikä mahdollistaa esimerkiksi selaimen sekä web-palvelimen välisen reaaliaikaisen kommunikaation sekä tiedonvälityksen.

1 Työn lähtökohdat

1.1 Johdanto

Rajapinnat ovat kriittinen osa nykypäivän web-sovelluksia. Monet sovellukset sisältävät jo selaimella käytettävän version lisäksi esimerkiksi mobiilisovelluksen, jolla on useasti tarve päästä käsiksi samaan tietoon, jota sovelluksen selainversio käyttää. Sovellusta käyttäville tahoille on kuitenkin tärkeää, että tarvittava tieto on mahdollisimman nopeasti sekä helposti saatavilla palvelusta. On tapauksia, joissa omaan sovellukseen on tarve tuoda tai hakea toisesta täysin erillisestä järjestelmästä jotain toiminnallisuutta tai tietoa rajapinnan kautta. Yritysten liiketoiminta voi perustua esimerkiksi tietyn toiminnallisuuden tarjoamiseen oman rajapinnan kautta. Tällaisissa tapauksissa on tärkeää, että tarjottavat rajapinnat ovat helppokäyttöisiä, hyvin dokumentoituja sekä niiden kehittäminen ja ylläpitäminen on vaivatonta.

Vallitseva menetelmä rajapintojen tekemiseen pohjautuu REST-arkkitehtuurimalliin. GraphQL on kuitenkin uusi teknologia, joka pyrkii haastamaan perinteisemmät REST-rajapinnat tarjoamalla vaihtoehtoisen tavan tiedon hakemiseen. GraphQL pyrkii olemaan helppokäyttöinen teknologia, joka mahdollistaisi rajapintojen tehokkaan käyttämisen, kehittämisen sekä muokkaamisen muuttuvien sovellusten tarpeisin.

Oikean menetelmän sekä teknologian valinta omaan sovellusprojektiin ei ole helppoa. Tässä opinnäytetyössä tutkitaan GraphQL- sekä REST-rajapintojen välisiä eroja, sekä pyritään selvittämään millaisissa projekteissa niiden käyttäminen on perusteltua. Tämän työn arvo lukijalle on antaa parempi ymmärrys siitä, millaisiin projekteihin GraphQL voisi olla parempi valinta. Työn arvo tekijälle on antaa mahdollisuus oman ammatillisen osaamisen syventämiseen sekä tiedon kartuttamiseen käsiteltävien aiheiden osalta.

1.2 Toimeksiantaja

Opinnäytetyön toimeksiantajana toimi jyväskyläläinen, vuonna 2011 perustettu ohjelmistotalo Meiko Oy. Meiko Oy suunnittelee ja toteuttaa räätälöityjä ohjelmistoja. Sen tavoitteena on kehittää yritysten yksilöllisiin tarpeisiin suunnattuja

järjestelmiä, jotka kehittävät liiketoimintaa sekä tukevat työntekoa. Meiko Oy työllistää tällä hetkellä 24 työntekijää. (Me 2018.)

1.3 Tausta ja tavoitteet

Opinnäytetyön tarkoitus on pyrkiä tuottamaan ratkaisu tutkimusongelmaan. Tutkintakysymyksiä voidaan käyttää ongelman ratkaisun apuna. Kerätyn aineiston pohjalta esitetyt vastaukset pyrkivät vastaaman näihin tutkintakysymyksiin, sekä sitä kautta ratkaisemaan tutkimusongelman. (Kananen 2015, 11.)

Idea opinnäytetyön tekemiselle lähti tilaajan tarpeesta kartoittaa, millaisissa asiakasprojekteissa GraphQL:n käyttö olisi edullista. GraphQL koettiin uutena sekä mielenkiintoisena teknologiana, mitä mahdollisesti haluttiin hyödyntää näissä projekteissa. Ongelmana kuitenkin oli, että tarve sen käytölle ei kuitenkaan aina ollut selkeä.

Opinnäytetyön tavoitteena oli kartoittaa, missä käytännön tilanteissa GraphQL:n hyödyt/haitat verrattuna REST-rajapintoihin tulevat ilmi, sekä tutkia ja pohtia, milloin ja minkälaisissa projekteissa GraphQL-teknologiaa kannattaisi hyödyntää. Tärkeimmät tutkimuskysymykset, joihin haettiin vastausta, olivat seuraavat:

- Millaisissa projekteissa, sekä tapauksissa GraphQL:n valinta on kannattavaa?
- Millä tavoin GraphQL-rajapinnan kehittäminen eroaa REST-rajapinnan kehittämisestä?
- Mitä mahdollisuuksia GraphQL:n käyttämisestä verrattuna REST-arkkitehtuurimalliin on?
- Mitä uhkia GraphQL:n käyttämisestä verrattua REST-arkkitehtuurimalliin on?

Työn tutkimusotteena käytetään laadullista tutkimusta. Laadullisen tutkimuksen tarkoitus on pyrkiä tuottamaan ratkaisu tai ymmärrys tutkittavana olevaan ongelmaan (Kananen 2015, 29). Laadullista tutkimusta tukevana aineiston tietolähteinä voidaan hyödyntää esimerkiksi kirjallisuutta, raportteja tai verkkosivuja (Kananen 2019, 28). Tutkimuksen aineistopohja koostetaan pääosin aiheeseen liittyvän kirjallisuuden sekä erinäisten internet-artikkeleiden kautta. GraphQL-

teknologian käyttöön liittyen löytyy paljon ohjeita sekä materiaalia. Tämän opinnäytetyön tarkoituksena ei ole kuitenkaan toimia tarkkana ohjeena siinä, miten kyseistä teknologiaa käytetään.

2 REST-arkkitehtuurimalli

2.1 Historia

Internetin kasvaessa nopeasti 1990-luvun loppupuolella, kehittäjien ongelma oli kehittää web-sovelluksia, jotka pystyisivät tehokkaasti kommunikoimaan keskenään. SOAP-protokolla, jonka tarkoitus oli helpottaa palvelimien sekä niiden käyttäjien keskinäistä kommunikointia, julkaistiin ongelman ratkaisemiseksi. SOAP-protokolla sisälsi hyvin paljon sääntöjä, jotka monille näyttäytyivät kuitenkin ensisijaisesti hyvänä asiana. Standardisoidun protokollan ansiosta, kehittäjät pystyisivät aloittamaan työskentelyn uusissa projekteissa. (Hoffmann 2017.)

Roy Fielding ei kuitenkaan innostunut SOAP-protokollan kankeudesta. Kun SOAP-protokolla julkaistiin, Fielding työskenteli HTTP-protokollan spesifikaation kanssa ja samaan aikaan julkaisi oman määritelmänsä, jonka pohjalta web-sovelluksia pystyttäisiin rakentamaan. Fielding ja hänen kannattajansa väittivät, että REST-arkkitehtuurimalli soveltuisi paremmin web-sovelluksien rakentamisen pohjalle, koska se oli yksinkertaisempi sekä antoi SOAP-protokollaan verraten enemmän joustavuutta sitä hyödyntäville järjestelmille. (Mt.)

2.2 Yleistä

Roy Fielding julkaisi vuonna 2000 väitöskirjan, jossa hän kuvaa REST-arkkitehtuurityylin käyttöön liittyvät säännökset. REST-arkkitehtuurimallia yleisimmin käytetään HTTP-protokollan kanssa, mutta sitä ei itsessään ole sidottu mihinkään tiettyyn tiedonsiirtoprotokollaan. (Fielding 2000; What is REST n.d.)

ProgrammableWeb-sivustolla vuonna 2017 julkaistun tutkimuksen mukaan REST-arkkitehtuurimalli on yleisin heidän arkistoistaan löytyvien rajapintojen toteutusarkkitehtuurimalli 81.53%:n lukemalla (Santos 2017).

REST-arkkitehtuurimallissa palvelun tarjoama tieto abstraktoidaan eri resurssien (engl. resource) taakse. Jokaisella resurssilla on oma yksilöllinen tunniste (URI), jonka kautta tietoa voidaan hakea sekä muokata. Resursseja voidaan kuvata esimerkiksi kuviossa 1 nähtävällä tavalla. (Fielding 2000.)

```
GET /api/users
PUT /api/users/:id
POST /api/users
DELETE /api/comments/:id
```

Kuvio 1. Resurssien kuvaaminen REST-arkkitehtuurimallin avulla

HTTP-protokolla on yleisin REST-arkkitehtuurimallin kanssa käytetty tiedonsiirtoprotokolla. Tämä mahdollistaa HTTP-protokollan ominaisten metodien (esim. GET, POST, PUT, DELETE) käyttämisen tietojen hakemiseen sekä muokkaamiseen. (Santos 2017; What is REST n.d.)

Kuviossa 2 on nähtävissä esimerkki käyttäjien tiedon hakemisesta REST-rajapinnasta käyttäen *curl*-työkalua. Kuviosta 2 nähdään, että tietojen hakemiseen käytetään HTTP-protokollan GET-metodia.

```
curl -X GET http://localhost:9000/api/users
```

Kuvio 2. Käyttäjien hakeminen REST-rajapinnalta käyttäen curl-työkalua

Koska REST-arkkitehtuurimalli ei määrittele, mitä tiedonsiirtoprotokollia tai metodeja REST-rajapintojen tekemisessä pitää käyttää, on täysin mahdollista, että eri

rajapinnoissa voidaan käyttää eroavia metodeja esimerkiksi tiedon muokkaamiseen (What is REST n.d).

2.3 Rajoitteet

REST-arkkitehtuurimalli asettaa tiettyjä rajoitteita sitä hyödyntäville järjestelmille. Näiden rajoitteiden on täytyttävä mikäli järjestelmää halutaan (Fielding 2000).

Palvelin-asiakas-malli

Järjestelmän käyttöliittymä tulee eriyttää järjestelmän tietovarastosta. Tämä mahdollistaa palvelimen sekä käyttöliittymän puolella toisistaan riippumattoman kehityksen. (Mt.)

Tämä tarkoittaa, että esimerkiksi sovelluksen frontend-puolelle tehtyjen koodimuutosten ei pitäisi suoraan vaikuttaa sovelluksen taustajärjestelmän toimintaan. Tämän avulla sovelluksen eri osa-alueita työstävät tiimit voivat työstää omia ominaisuuksia ilman, että muiden työ häiriintyy.

Tilaton

Järjestelmän tulee olla tilaton. Palvelimelle lähetettyjen pyyntöjen tulee sisältää kaikki pyynnön käsittelemiseen tarvittava tieto. Pyynnön käsittelemisessä ei voida käyttää hyväksi mitään aiempaa kontekstiin liittyvää tietoa. Käytännössä tämä tarkoittaa, että järjestelmää käyttävän osapuolen (käyttäjä) on ylläpidettävä tietoa omasta tilastaan. (Mt.)

Kyseisen rajoitteen on tarkoitus parantaa järjestelmän skaalautuvuutta, luotettavuutta ja näkyvyyttä. Skaalautuvuus parantuu, koska palvelimen ei tarvitse käyttää erikseen resursseja järjestelmän tilan ylläpitämiseen pyyntöjen välillä. Luotettavuus ja näkyvyys parantuvat, koska virhetilanteissa pystytään helposti tutkimaan lähetettyjä pyyntöjä sekä niiden sisältöä. (Mt.)

Käyttäjän tekemien pyyntöjen pitää siis sisältää esimerkiksi tarvittava tieto autentikointiin liittyen, jotta palvelin pystyy määrittämään, onko käyttäjä kirjautunut vai ei.

Välimuisti

Vastauksissa olevan tiedon mukaan tulee määrittää joko implisiittisesti tai eksplisiittisesti, onko tieto mahdollista tallentaa välimuistiin myöhempää käyttöä varten. Tiedon tallentaminen välimuistiin parantaa järjestelmän suorituskykyä, koska se voi vähentää verkkokyselyiden määrää. (Mt.)

Välimuistin käyttämisellä on myös riskinsä, se voi mahdollisesti myös vähentää järjestelmän luotettavuutta. Tämä voi tapahtua, jos välimuistin sisältämä tieto eroaa huomattavasti siitä, mitä palvelimelta pyydetty tieto tulisi olemaan. (Mt.)

Yhtenäinen rajapinta

Järjestelmän tulee tarjota yhtenäinen rajapinta. Tämä mahdollistaa järjestelmän arkkitehtuurin yksinkertaistamisen sekä järjestelmän sisältämän operaatioiden näkyvyyden parantamisen. Palvelun tekninen implementaatio voidaan eriyttää tällöin tarjottavasta rajapinnasta, mikä mahdollistaa helposti palvelun jatkuvan kehityksen. (Mt.)

Rajapinnan tarjoamien resurssien nimeämiset on syytä tehdä yhteneväisesti. Ei käytetä sekaisin esimerkiksi yksikköä ja monikkoa resurssien nimissä, kuten ”/posts” ja ”/user”.

Kerrosrakenteinen järjestelmä

Kerrosrakenteinen järjestelmä mahdollistaa palvelun arkkitehtuurin koostamisen erillisistä osioista. Järjestelmän erilliset osiot operoivat vain oman osionsa rajoissa. (Mt.)

Käytännössä tämä voi tarkoittaa esimerkiksi, että järjestelmän osa, joka on vastuussa rajapinnasta palautettavasta tiedosta, ei ota kantaa siihen, millä tavalla tieto tulisi visualisoida. Tai puolestaan järjestelmän kuormantasaaja ei ota kantaa siihen, missä muodossa tieto tulee palauttaa rajapinnalta.

Tämä auttaa myös järjestelmän skaalautuvuuteen, jos järjestelmän eri osiot sijaitsevat eri palvelimilla. Tällöin voidaan skaalata koko järjestelmän sijasta ainoastaan yhtä osiota.

Valinnainen koodin lataaminen

REST-arkkitehtuurimallin viimeinen rajoitus koskee valinnaista koodin lataamista. Järjestelmän on mahdollista tarjota ladattavaksi esimerkiksi skriptejä, joiden on tarkoitus tarjota järjestelmän käyttäjälle lisää toiminnallisuutta. Kaikkien järjestelmän käyttäjien ei kuitenkaan ole mahdollista hyödyntää ominaisuuksia, jos esimerkiksi palomuuri estää näiden tiedostojen lataamisen. Tästä syystä koodin lataaminen on valinnainen rajoite REST-arkkitehtuurimallin mukaisissa järjestelmissä. (Mt.)

Yleensä rajapinta palauttaa staattista tietoa esimerkiksi JSON-formaatin muodossa. Mikäli kuitenkin käyttäjän selain sallii esimerkiksi JavaScriptin suorittamisen, voidaan tällöin rajapinnalta myös palauttaa JavaScript-koodia, jonkin tietyn ominaisuuden käyttämiseksi.

3 GraphQL

3.1 Historia

GraphQL on alkujaan Facebookin sisäisesti kehittämä projekti, joka sai alkunsa vuonna 2012. Kolme vuotta myöhemmin se muutettiin avoimen lähdekoodin projektiksi ja kehitysvastuu siirrettiin varta vasten perustetulle GraphQL Foundationille. GraphQL Foundation kuuluu pääorganisaationsa, The Linux Foundationin, alle. (GraphQL Foundation 2019.)

GraphQL itsessään on vain määritelmä kyselykielestä sekä palvelimella suoritettavasta ajonajasta (engl. runtime). GraphQL:n määritelmä sisältää kyselykielen tietorakenteille asetetut pystyvyydet sekä vaatimukset. Kyseinen määritelmä on projektin alkuajoista muuttunut, ja se myös jatkaa kehitystään tulevaisuudessa. (GraphQL 2018).

Koska GraphQL:n kyselykieli on pelkkä määritelmä, sitä ei ole sidottu mihinkään tiettyyn ohjelmointikieleen tai tietokantaan. Tämä mahdollistaa GraphQL:n käytön jo olemassa olevan ohjelmointikoodin sekä tiedon ohessa. (Introduction to GraphQL 2019.)

3.2 Kehitystarve

Vuonna 2012 Facebookin mobiilisovellukset pitivät sisällään pääosin vain näkymiä heidän mobiililaitteille suunnatulta web-sivustoltaan. Sivuston kasvaessa mobiilisovellukset kuitenkin kävivät hitaaksi ja kaatuilivat usein. Facebookin tavoitteeksi tuli tällöin mobiilisovellusten kirjoittaminen kyseisten alustojen omalla natiivilla ohjelmointikielellä. (Byron 2015.)

Tämä tavoite kuitenkin vaati huomattavan määrän ohjelmointikoodin kirjoittamista sekä palvelimen että mobiilisovelluksen puolella. Palvelimelta vaaditun tiedon eriäväisyydet eri sovellusten välillä, sekä monet edestakaiset kyselyt palvelimelle tiedon hakemiselle turhauttivat heitä. (Mt.)

Näistä syistä Facebookilla oli tarve luoda rajapinta tiedonhakuun, joka pystyisi vastaamaan kaikkeen Facebookin tarjoamaan sekä tarvittavaan tietoon. Tavoitteena oli, että sovellusten kehittäjät pystyisivät helposti opettelemaan sekä käyttämään kyseistä rajapintaa. (Mt.)

3.3 Operaatiot

1970-luvulla silloinen IBM:n työntekijä Edgar F. Cobb julkaisi lyhyen tutkielman nimeltään “A Relational Model of Data for Large Shared Databanks”. Tutkielmassa kuvattiin malli, jossa tauluja käytettiin tiedon tallentamiseen sekä muokkaamiseen. Cobbin tutkielma synnytti IBM:llä useita uusia tuotteita, joista yksi oli relaatiotietokantojen operoimisen tarkoitettu kieli nimeltään SQL. SQL-kieli mahdollistaa tietokannassa olevan datan hallinnan sekä käyttämisen yksinkertaisten komentojen SELECT, INSERT, UPDATE ja DELETE avulla. SQL-kielen avulla pystytään esimerkiksi kirjoittamaan yksittäinen kysely, joka palauttaa lopulta tietoa useasta eri taulusta. GraphQL pyrkii hyödyntämään samoja ideoita, joita käytettiin SQL-kielen rakentamiseen ja tarjoamaan helpon tavan, jolla tietoa pystytään hakemaan web-rajapinnasta. (Banks & Porcello 2018, Luku 3; Relational database n.d.)

GraphQL tukee kolmea erilaista operaatiota, joita voidaan käyttää tiedon hakemiseen sekä muokkaamiseen. Näitä ovat *query*, *mutation* sekä *subscription*.

Tiedon hakeminen GraphQL-palvelusta tapahtuu käyttämällä sille ominaista *query*-operaatiota. (Banks & Porcello 2018, luku 3.)

3.4 Kyselyt

3.4.1 Kyselyiden tekeminen

Kyselyt kirjoitetaan GraphQL:n kyselykielen avulla. Kyselykielen syntaksi muistuttaa hieman JSON-notaatiota. Koska GraphQL ei ole sidottu mihinkään tiettyyn ohjelmointikieleen, kyselyt ovat lopulta vain merkkijonoja, jotka tällöin toimivat minkä tahansa ohjelmointikielen kanssa. (Banks & Porcello 2018, luku 3.) Kuviossa 3 on näkyvissä esimerkki tiedon hakemisesta käyttäen GraphQL:n kyselykieltä.

Kysely

```
query getAllUsersWithPosts{
  users {
    email
    posts {
      title
    }
  }
}
```

Vastaus


```
"data": {
  "users": [
    {
      "email": "jordiharvey@gmail.com",
      "posts": [
        {
          "title": "Sunt expedita qui ex ducimus et enim."
        },
        {
          "title": "Esse quia sit odit."
        }
      ]
    },
    {
      "email": "esteltorp@gmail.com",
      "posts": [
        {
          "title": "Ut rerum corporis et et sit."
        },
        {
          "title": "Nulla sint consequuntur enim neque eos sit recusandae."
        },
      ]
    }
  ]
}
```


Kuvio 3. Esimerkkikysely käyttäjien sekä käyttäjän julkaisujen hakemiseen

Kuvion 3 kysely alkaa operaation tyypin määrittämisellä, joka tässä tilanteessa on *query*. Kaikille GraphQL:n operaatiolle on myös mahdollista antaa kuvaava nimi, tässä tapauksessa se on *getAllUsersWithPosts*. Operaation nimen voi halutessaan jättää pois. Operaation nimen käyttöä kuitenkin suositellaan, koska sen avulla pystytään helpommin päättelemään, mitä kyseisessä operaatiossa halutaan tehdä. Myös operaation tyypin voi halutessaan jättää pois, jolloin GraphQL-palvelin olettaa, että kyseessä on *query*-operaatio. (Introduction to GraphQL 2019, Operation name.)

Kuvion 3 kyselyssä haetaan käyttäjiä sekä käyttäjän julkaisuja. Käyttäjiltä halutaan tietää sähköpostiosoite eli *email* ja käyttäjän julkaisujen otsikko eli *title*. Edellä mainittu kysely palauttaa onnistuessaan kuvion 3 alaosassa näkyvän vastauksen.

Jos kysely onnistuu, ja tietoa voidaan palauttaa, se löytyy *data*-kentän alta. Kuviosta 3 voidaan nähdä, että saadut tiedot ovat samassa rakenteessa kuin mitä kyselyssä. Vastaus ei myöskään sisällä muuta kuin kyselyssä määritetyt kentät.

Jos kyselyä suoritettaessa tapahtuu virhe, vastauksesta löytyy kenttä nimeltään *errors*. Kenttä sisältää tällöin listan kaikista virheistä, jotka tapahtuivat kyselyä suoritettaessa. (GraphQL 2018. Luku 7.1.2.) Kuviossa 4 on näkyvissä esimerkki virheellisen kyselyn vastauksesta.


```
{
  "errors": [
    {
      "message": "Error happened while quering the grapqhl api",
      "locations": [
        {
          "line": 31,
          "column": 3
        }
      ],
      "path": [
        "users"
      ]
    }
  ],
  "data": {
    "users": null
  }
}
```


Kuvio 4. Virheellisen kyselyn vastaus

Mikäli virhe tapahtuu palvelimella jo ennen kyselyn suorittamista, ei *data*-kenttä ole mukana vastauksessa.

Esimerkeissä olevien kyselyiden vastaukset on automaattisesti serialisoitu JSON-formaattiin, joka on yleisin formaatti GraphQL:n implementaatioissa. Eri serialisointiformaatteja on kuitenkin mahdollista käyttää, kunhan ne tukevat GraphQL:n määrittelyssä asetettuja reunaehtoja. (Mts. Luku 7.2.)

3.4.2 Argumentit

GraphQL-kyselyille on myös mahdollista antaa argumentteja (Introduction to GraphQL 2019, Aguments). Argumentit mahdollistavat tarkempien tulosten näyttämisen esimerkiksi tietyn tunnisteen perusteella, näytettävien tulosten määrän rajoittamisen tai sivutuksen käyttämisen.

Kuviossa 5 on näkyvissä esimerkkikysely yksittäisen käyttäjän tietojen hakemisesta *id*-argumentin avulla.

Kysely

```
query {
  user(id: 14) {
    id
    email
    name
  }
}
```

Vastaus

```
{
  "data": {
    "user": {
      "id": "14",
      "email": "jackyrussel@gmail.com",
      "name": "Jacky Russel"
    }
  }
}
```

Kuvio 5. Yksittäisen käyttäjän tietojen hakeminen *id*-argumentin avulla

Kyselyille on myös mahdollistaa antaa useampi kuin yksi argumentti. Sisäkkäisille objekteille ja yksittäisille kentille on myös mahdollista antaa argumentteja. Parametrien antaminen yksittäisille kentille mahdollistaa esimerkiksi tiedon alustuksen palvelimella ilman ylimääräistä käsittelyä selaimen puolella. (Mts. Arguments.)

Kuviossa 6 on näkyvissä esimerkki julkaisun hakemisesta. *Date*-kentälle annetaan *format*-argumentti, joka tässä tapauksessa ilmaisee, millä tavalla päivämäärä halutaan formatoida.

Kysely

```
query {
  post(id: 2) {
    id
    date(format: unix)
  }
}
```

Vastaus

```
{
  "data": {
    "post": {
      "id": "2",
      "date": "1552286398"
    }
  }
}
```

Kuvio 6. Argumenttien antaminen yksittäiselle kentälle

Kuten kuviosta 6 voidaan todeta, argumenttien käyttäminen kenttien formatointiin antaa kyselyn tekijälle enemmän valtaa siihen, millä tavalla vastauksen sisältämä tieto halutaan formatoida.

3.4.3 Aliakset

GraphQL-kyselyn kentille syötettävät argumentit mahdollistavat erilaisten vastauksien saamisen riippuen siitä, millaisia argumentteja kentälle syötetään. Koska kyselyissä määritetyt kentät palautuvat vastauksissa täysin saman nimisinä, kuin mitä ne on kyselyssä määritetty, ei esimerkiksi kuvion 6 *date*-kenttää ole suoraan mahdollista kysyä kahteen kertaan eri argumenteilla formatoituna. GraphQL kuitenkin tarjoaa keinon nimetä minkä tahansa kentän kyselyssä uudelleen. Uudelleen nimeäminen mahdollistaa tällöin saman kentän käyttämisen esimerkiksi eri argumenteilla useampaan kertaan. (Introduction to GraphQL 2019, Aliases.)

Kuviossa 7 on nähtävissä esimerkki kyselystä, jossa *date*-kenttää kysytään kahteen kertaan, eri argumenteilla.

Kysely

```
query {
  post(id: 2) {
    id
    unixDate: date(format: unix)
    ISODate: date(format: ISO)
  }
}
```

Vastaus

```
{
  "data": {
    "post": {
      "id": "2",
      "unixDate": "1552286398",
      "ISODate": "2019-03-11T06:39:58.000Z"
    }
  }
}
```

Kuvio 7. Kysely aliaksilla

Kuten kuvion 7 vastauksesta voidaan todeta, saman kentän sisältämä tieto voidaan palauttaa erilaisessa formaatissa. Kyselyä tehdessä kenttien nimet voidaan myös itse päättää. Tämän avulla voidaan välttää tarvetta vastauksen formatointiin esimerkiksi frontendin päässä.

3.4.4 Fragmentit

Tapauksissa, joissa haetaan useita saman tyyppisiä objekteja, on yksittäisten kenttien nimien kirjoittaminen jokaiselle erilliselle objektille työlästä. GraphQL kuitenkin tarjoaa keinon, miten useasti käytettäviä kenttiä voidaan yhdistää yhden *fragment*-kentän alle. (Banks & Porcello 2018, luku 3; Introduction to GraphQL 2019, Fragments.)

Fragment-kentän määrittämisestä sekä käyttämisestä on nähtävissä esimerkki kuviossa 8.

Kysely

```
fragment postContent on Post {
  title
  content
}

query {
  firstPost: post(id: 2) {
    ...postContent
  }
  secondPost: post(id: 3) {
    ...postContent
  }
}
```

Vastaus

```
{
  "data": {
    "firstPost": {
      "title": "Et aut ullam commodi a ratione ut ipsam.",
      "content": "Optio qui aut laborum suscipit consequatur veritatis."
    },
    "secondPost": {
      "title": "Tenetur aliquid accusamus nulla enim.",
      "content": "Est impedit possimus inventore beatae dignissimos deserunt."
    }
  }
}
```

Kuvio 8. Kysely *fragment*-kenttää hyödyntäen

Fragment-kenttää alustaessa kuuluu määrittää, mille tyypille kyseinen valinta kuuluu (Banks & Porcello 2018, luku 3). Kuviossa 8 on nähtävissä, miten *postContent* (*fragment*-kenttä) on määritetty kuulumaan *Post*-tyypille. *Fragment*-kenttien käyttö on kätevää erityisesti silloin, kun kyselyllä halutaan valita useita kenttiä useiden yksittäisten objektien alta.

Jos palvelimen puolelta on kyselyn vastaukseksi määritetty palautettavan *interface*- tai *union*-tyyppi, on tällöin *inline*-fragmenttien käyttäminen välttämätöntä. *Interface*- ja *union*-tyypit voivat sisältää erilaisia kenttiä, jonka takia kenttien valinta on tehtävä aina tapauskohtaisesti. (Introduction to GraphQL 2019, Inline Fragments.)

Kuviossa 9 on näkyvissä esimerkki, missä kyselyssä hyödynnetään *inline*-fragmentteja.

Kysely

```
query GetAllUsersContent {
  user(id: 2) {
    allContent {
      __typename
      ... on Post {
        title
        content
      }
      ... on Comment {
        content
      }
    }
  }
}
```

Vastaus

```
{
  "data": {
    "user": {
      "allContent": [
        {
          "__typename": "Post",
          "title": "Quasi aut velit commodi.",
          "content": "Nisi quia alias nam cupiditate saepe est sit explicabo quia."
        },
        {
          "__typename": "Post",
          "title": "Exercitationem et perferendis minima.",
          "content": "Quia autem nesciunt iusto explicabo ullam ullam."
        },
        {
          "__typename": "Comment",
          "content": "Cum eius voluptates ea et enim ea repudiandae."
        },
      ]
    }
  }
}
```

Kuvio 9. Yhden rivin *fragment*-kysely

Kuvion 9 Kenttä *allContent* palauttaa taulukon, joka sisältää *Comment*- sekä *Post*-objektityyppejä. *Comment*-objektityyppi ei sisällä *title*-kenttää, joten yhden rivin fragmenttien avulla *title*-kenttä voidaan kuitenkin valita ainoastaan *Post*-objektityypistä.

Kuviossa 9 näkyvissä oleva *__typename*-kenttä sisältää metatietona kyseisen objektin tyypin. Kyseinen metakenttä on mahdollista lisätä jokaiseen kyselyyn. (Mts. Meta fields.)

3.4.5 Muuttujat

Kuviossa 7 on nähtävissä, miten kyselyille syötetyt argumentit ovat olleet joko kovakoodattuja numeroita tai merkkijonoja. Koska argumentit ovat kovakoodattuja suoraan kyselyyn, vaatisi niiden muuttaminen koko kyselyn uudelleenrakentamisen uusilla arvoilla. Teknisen toteutuksen tasolla tämä tarkoittaisi kyselyn serialisointia merkkijonoksi, uusien arvojen syöttämistä merkkijonoon sekä uuden merkkijonon serialisointia takaisin GraphQL-kyselyksi. Tämä tapa ei kuitenkaan ole suositeltavaa. Sen sijaan GraphQL tarjoaa keinon, jolla kyselyille voidaan antaa muuttuvia arvoja. Näitä muuttujia kutsutaan nimellä *variables*. Muuttujat pystytään tarjoamaan kyselyn mukaan erillisellä JSON-objektilla. (Banks & Porcello 2018, luku 3; Introduction to GraphQL 2019, Variables.)

Kuviossa 10 on nähtävillä esimerkki muuttujien käyttämisestä kyselyn mukana.

Kysely

```
query getPostById($postId: Int!, $dateFormat: DateFormat) {
  post(id: $postId) {
    title
    date(format: $dateFormat)
  }
}
```

Muuttujat

```
{
  "postId": 3,
  "dateFormat": "unix"
}
```

Vastaus

```
{
  "data": {
    "post": {
      "title": "Tenetur aliquid accusamus nulla enim.",
      "date": "1553449339"
    }
  }
}
```

Kuvio 10. Kysely muuttujilla

Kyselyn avulla voidaan nyt saada useita eri vastauksia, riippuen kyselyyn annettavista muuttujista.

3.4.6 Direktiivit

Kentälle tai fragmentille voidaan kyselyn mukaan lisätä ominaisuus nimeltään *directive* (direktiivi). Direktiivit mahdollistavat tiettyjen kenttien suodattamisen tai lisäämisen tiettyjen muuttujien perusteella. GraphQL:n spesifikaatiossa on määritelty pakolliseksi kaksi direktiiviä: *@include* ja *@skip*. Nämä kummatkin ottavat vastaan yhden argumentin, joka on tyypiltään *Boolean* (totuusarvo). Täysin uusia direktiivejä on kuitenkin mahdollista määritellä GraphQL:n palvelinkirjaston implementoinneissa. (Introduction to GraphQL 2019, Directives).

Kuviossa 11 on nähtävissä kysely, jossa kenttien suodattamiseen käytetään direktiivejä.

Kysely

```
query GetSinglePost($showFullContent: Boolean!, $hideUser: Boolean!){
  post(id: 1) {
    title
    content @include(if: $showFullContent)
    user @skip(if: $hideUser) {
      id
      email
    }
  }
}
```

Muuttujat

```
{
  "showFullContent": false,
  "hideUser": false
}
```

Vastaus

```
{
  "data": {
    "post": {
      "title": "Aperiam laboriosam dolor magnam et vero molestias dolores laborum.",
      "user": {
        "id": "5",
        "email": "elisaklocko@gmail.com"
      }
    }
  }
}
```

Kuvio 11. Kenttien suodattaminen direktiivien avulla

Kuviossa 11 oleva *content*-kenttä näytetään ainoastaan, jos muuttuja *showFullContent* on tosi. *User*-objekti puolestaan jätetään näyttämättä ainoastaan,

jos muuttuja *hideUser* on tosi. Näin vaadittavaa sisältöä pystytään suodattamaan dynaamisesti.

3.5 Mutaatiot

GraphQL:n *query*-operaatiota käytetään yleensä tiedon hakemiseen. Tiedon muokkaamiseen käytetään sille ominaista *mutation*-operaatiota. Aivan kuten *query*-tyyppi, myös *mutation*-tyyppi palauttaa objektin, jolta pystytään siis kysymään samoja kenttiä kuin mitä normaaleissa kyselyissä. Ainoa asia miten *mutation*-operaatio eroaa *query*-operaatiosta, on että *mutation*-operaatiota käytetään muokkaamaan tietoa jollain tavalla. (Banks & Porcello 2018, luku 3; Introduction to GraphQL 2019, Mutations.)

Uuden julkaisun luominen *mutation*-operaattoria käyttäen voidaan nähdä kuviosta 12. Siinä *mutation*-operaatiolle on annettu kuvaava nimi *CreateNewPost*. Rivillä 2 oleva nimi *addPost* puolestaan viittaa GraphQL-palvelimen schemassa määritettyyn *mutation*-operaation.

Mutaatio

```
mutation CreateNewPost($userId: Int!, $title: String!, $content: String!) {
  addPost(user: $userId, title: $title, content: $content) {
    id
    title
  }
}
```

Vastaus

```
{
  "data": {
    "addPost": {
      "id": "57",
      "title": "Test title for the mutation"
    }
  }
}
```

Kuvio 12. Uuden julkaisun luominen *mutation*-operaation avulla

Kuten kuviosta 12 voidaan todeta, myös *mutation*-operaatiota käytettässä, vastaus sisältää ainoastaan käyttäjän määrittämät kentät. Tämän vuoksi, jokaisen onnistuneen mutaation jälkeen voidaan esimerkiksi käyttöliittymässä olettaa, että vastaus tulee sisältämään vaaditut kentät.

3.6 Tilaus

Subscription on viimeisin kolmesta GraphQL:n operaatiosta. *Subscription* mahdollistaa GraphQL-palvelimella tapahtuvan tiedon muutosten kuuntelemisen reaaliaikaisesti. Tarve *subscription*-operaation luomiseen tuli Facebookin tarpeesta näyttää kuinka monta tykkäystä julkaisulla oli, ilman että käyttäjän tarvitsee päivittää sivuaan. (Banks & Porcello 2018, luku 3.)

Mahdolliset *subscription*-operaatiot tulee määrittää palvelimen schemassa, jotta niitä pystytään käyttämään (mts. luku 3). Kuviossa 13 on nähtävissä esimerkki subscription-operaatiosta, joka kuuntelee uusien julkaisujen lisäystä.

Tilaus

```
subscription {
  postAdded {
    id
    title
  }
}
```

Vastaus

```
{
  "data": {
    "postAdded": {
      "id": "65",
      "title": "New post added"
    }
  }
}
```

Kuvio 13. *Subscription*-operaation kuuntelu

Uuden julkaisun lisäämisen myötä, GraphQL-rajapinta lähettää kuvion 13 mukaisen vastauksen kaikille osapuolille, jotka sillä hetkellä kuuntelevat kyseistä operaatiota. Kuten kuviosta 13 voidaan nähdä, myös *subscription*-operaation lähettämät tiedot sisältävät ainoastaa käyttäjän itse määrittelemät kentät.

3.7 Schema

3.7.1 Yleistä

GraphQL:n tarjoamat tyypit, tyyppien sisältämät kentät, tyyppien relaatiot sekä pääoperaatiot tulee määritellä schemassa. Jotta edelliset vaatimukset olisivat

helposti ymmärrettävissä, GraphQL tarjoaa helposti luettavan kielen nimeltään ”Schema definition language” (SDL), jonka avulla on tarkoitus ilmaista kaikki schemassa saatavilla olevat tyypit sekä niiden relaatiot toisiinsa. (GraphQL 2018, Luku 3.2; Understanding schema concepts n.d.)

Oman kielen määritteleminen oli tarpeellista, koska GraphQL itsessään on vain spesifikaatio, jolloin scheman määrittelemiseen ei voitu käyttää mitään tiettyä ohjelmointikieltä ja sen syntaksia (Introduction to GraphQL 2019, Schemas and Types).

Jokaisella schemassa määritetyllä tyypillä on oltava uniikki nimi. Käyttäjän luomien tyyppien nimet, eivät saa olla samannimisiä kuin GraphQL:n sisäänrakennetut tyypit. GraphQL:n scheman määrittämisessä on mahdollista käyttää kolmea erilaista tyyppiä, jotka ovat: Objektityyppi (engl. Object type), skalaarityyppi (engl. Scalar type) sekä lueteltu tyyppi (engl. Enum) (GraphQL 2018, Luku 3.2; Introduction to GraphQL 2019, Schemas and Types).

3.7.2 Pääoperaatiotyypit

Schemassa määritellään jokaiselle tuetulle pääoperaatiolle oletusarvoinen tyyppi. Pääoperaatioita ovat *query* (kysely), *mutation* (mutaatio) ja *subscription* (tilaus). Näistä ainoastaan *query*-operaatio on pakollinen määrittää. Jos *mutation*- tai *subscription*-tyyppejä ei määritetä, kyseinen palvelu ei tällöin tue niiden käyttämistä. Jokaisen pääoperaation tyypin tulee olla objektityyppi. (GraphQL 2018, Luku 3.2.)

3.7.3 Objektityyppi

Yksi GraphQL:n keskeisimmistä tyypeistä on objektityyppi. Objektityyppiä käytetään kuvaamaan sellaista oliota ja sen tietoja, mitä GraphQL-rajapinnasta pystytään hakemaan. (Introduction to GraphQL 2019, Type system.) Tarkastellaan kuvion 14 avulla uuden *User*-objektityypin määrittämistä SDL-syntaksin avulla

```
type User {
  id: ID!
  name: String!
}
```

Kuvio 14. Esimerkki *User*-tyypin määrityksestä

Kuviossa 14 määritelty User-objektityyppi sisältää *id*- ja *name*-kentät, jotka kuvaavat *User*-tyyppiin liitettyä tietoa. Missä tahansa kyselyssä, joka liittyy *User*-tyyppiin, on nyt mahdollista hakea näiden kenttien sisältämät tiedot.

3.7.4 Skalaarityypit

Kuvion 14 sisältämä *id*-kenttä, on määritelty yhdellä GraphQL:n sisäänrakennetulla skalaarityypillä (engl. scalar), nimeltään *ID*. Tämä tyyppi esittää tässä tilanteessa jokaiselle *User*-tyypille määriteltyä yksilöivää tunnistetta. Toinen kenttä *name*, on tyypiltään *String*, joka esittää UTF-8 koodattua merkkijonoa. *ID*-tyyppi sarjallistetaan samaan tapaan kuin GraphQL:n *String*-tyyppi, erona niillä on vain se, että *ID*-tyyppiä ei ole tarkoitettu ihmisluettavaksi (engl. Human readable). (Introduction to GraphQL 2019, Scalar types.)

Muita sisäänrakennettuja skalaarityyppejä on mm. *Int* (kokonaisluku), *Float* (liukuluku) sekä *Boolean* (totuusarvo). GraphQL sisältää siis kaiken kaikkiaan 5 ennalta määritettyä skalaarityyppiä. Sisäänrakennettujen skaalarityyppien lisäksi, käyttäjällä on myös mahdollisuus luoda omia, kustomoituja skaalarityyppejä. Käyttäjän vastuulla on myöhemmin schemassa määritellä, millä tavalla kustomoitu tyyppi sarjallistetaan sekä validoidaan. (Mts. Scalar types).

3.7.5 Luetellut tyypit

Luetellut tyypit (engl. *Enumeration types*) ovat skaalarityyppejä, jotka ovat ennaltamääritetty tiettyihin sallittuihin arvoihin. Lueteltuja tyyppejä voidaan käyttää esimerkiksi argumenttien validoimiseen. (Introduction to GraphQL 2019, Enumeration types.).

Luetellun *DateFormat*-tyypin määrittäminen SDL-syntaksin avulla voidaan nähdä kuviosta 15.

```
enum DateFormat {
  ISO
  unix
}
```

Kuvio 15. Luetellun *DateFormat*-tyypin määritys

Kun schemassa jatkossa viitataan *DateFormat*-tyyppiin, voidaan olettaa sen arvon olevan joko *ISO* tai *unix*.

3.7.6 Rajapintatyyppi

Interface on abstrakti rajapintatyyppi (Introduction to GraphQL 2019, Interfaces). Se toimii samalla tavalla kuin olio-ohjelmoinnissakin käytetty rajapinta, jonka tarkoituksena on ilmaista mitkä objektin käyttämät metodit on pakko määritellä, ottamatta kuitenkaan kantaa siihen, miten ne on määritelty (Documentation n.d, Object interfaces). GraphQL:n tapauksessa, rajapintatyypin tehtävänä on määrittää kentät, jotka rajapintatyypin implementoivan objektityypin tulee määrittää (Introduction to GraphQL 2019, Interfaces).

Kuviossa 16 on nähtävissä esimerkki *Publification*-rajapintatyypin määrittämisestä.

```
interface Publification {
  id: String
  title: String
  content: String
  user: User
  date(format: DateFormat) : String
}
```

Kuvio 16. *Publification*-rajapintatyypin määritys

Kaikilla objektityypeillä, jotka nyt implementoivat *Publification*-rajapintatyypin, tulee olla määritettynä kaikki rajapintatyypin määrittämät kentät ja kenttien vastaavat tyypit. Implementoivalle objektityypille on mahdollista määrittää myös uusia kenttiä, mitä ei rajapintatyypissä ole määritelty (mts. Interfaces).

3.7.7 Liitostyypit

GraphQL:n liitostyypit (engl. *union*) ovat samankaltaisia kuin rajapintatyypit. Ne eroavat toisistaan kuitenkin siinä, että liitostyypit eivät itsessään määritä mitä kenttiä

objektityyppien tulee sisältää. Liitostyyppien luominen on mahdollista ainoastaan olemassa olevista objektityypeistä. Liitostyyppiä ei voida luoda esimerkiksi rajapintatyypistä tai toisesta liitostyypistä. (Introduction to GraphQL 2019, Union types).

3.7.8 Syötetyyppi

Kuten aiemmissa esimerkeissä on nähty, kyselyille sekä mutaatioille on ollut mahdollista antaa useampia argumentteja. Tilanteissa, joissa halutaan antaa useampia parametreja, esimerkiksi objektin luomiseen, voidaan käyttää *input*-tyyppiä. *Input*-tyyppi mahdollistaa yksittäisten argumenttien syöttämisen sijasta, tiedon antamisen yhden objektin avulla. *Input*-tyypin luominen tapahtuu samalla tavalla kuin normaalin objektityypin luominen, poikkeuksena vain avainsanan *input* käyttö. (Introduction to GraphQL 2019, Input types).

Kuviossa 17 on nähtävissä esimerkki *PostInput*-syötetyypin määrittämisestä.

```
input PostInput {
  userId: Int!
  title: String!
  content: String!
}
```

Kuvio 17. *PostInput*-syötetyypin määritys

PostInput-syötetyyppiä voidaan nyt käyttää validoinnin apuna esimerkiksi mutaatioiden määrittelyssä argumenttien tyypitykseen.

3.7.9 Validointi

GraphQL:n käyttämä tyypitys mahdollistaa kyselyiden validoinnin ja oikeanlaisten virheilmoitusten näyttämisen virheellisten operaatioiden seurauksena. Tyypityksen lisäksi, kentille on myös mahdollista määrittää muuntimia (engl. *Type modifiers*), jotka määrittelevät miten kyseiset kentät validoidaan. (Introduction to GraphQL 2019, Lists and Non-null; Introduction to GraphQL 2019, Validation).

Tarkastellaan vielä kuviossa 14 nähtävissä olevaa *User*-objektityypin määritystä. *Name*-kentän tyypiksi on määritelty *String* (merkkijono). Tyypityksen perään on vielä

lisätty *!* merkki, joka tarkoittaa että *name*-kenttä ei voi missään tilanteessa olla määrittämätön. Jos se on määrittämätön, GraphQL-palvelin palauttaa virheen. (Introduction to GraphQL 2019, Lists and Non-null.)

4 GraphQL vastaan REST

4.1 Suorituskyky

4.1.1 Tiedon hakeminen

REST-arkkitehtuurimallissa tieto on mallinnettu eri resurssien ympärille. REST-rajapinta voi tällöin sisältää useita eri resursseja (esim. "*/users*" tai "*/posts*") mitä kuvataan URIen (Uniform Resource Identifiers) avulla. Jos tietoa halutaan yhdistää eri resursseista, se vaatii aina erillisiä kutsuja useaan eri URIin. (Fielding 2000; Maldonado 2019; What is REST n.d.)

REST-rajapinnoissa palvelin päättää vastauksen rakenteen sekä sen sisältämän tiedon. Tämän seurauksena rajapintaa voi palauttaa ylimääräistä tietoa, mitä asiakas ei välttämättä tarvitse lainkaan. (Wieruch 2018.)

GraphQL-palvelussa kaikki tieto on saatavissa yhden päätepisteen kautta. On palvelua käyttävän asiakkaan vastuulla määritellä, mitä kaikkea tietoa päätepisteestä tulee palauttaa. Asiakas pystyy tällöin hakemaan kaiken tarvittavan tiedon yhdellä verkkokyselyllä. Tämä parantaa sovelluksen suorituskykyä, koska tällöin vähennetään useampien verkkokyselyiden määrää. (GraphQL 2018; Maldonado 2019.)

4.1.2 N + 1 -ongelma

Yleensä N + 1 -ongelmasta puhutaan ORM-ohjelmointitekniikan (engl. *Object-relation mapping*) kanssa. Tämä tarkoittaa esimerkiksi tietokantakyselyissä, että jokaista tehtyä kyselyä kohden joudutaan tekemään *N* määrä uusia kyselyitä kaiken tarvittavan tiedon saamiseksi (REST API – N + 1 Problem n.d.)

REST-rajapinnan kanssa N + 1 -ongelma voi ilmetä, jos kaikkea tarvittavaa tietoa ei pystytä hakemaan yhdestä resurssista. Tällöin uusia verkkokyselyitä joudutaan tekemään *N* määrä lisää kaiken tarvittavan tiedon hakemiseen. Ongelma voidaan yleensä ratkaista suunnittelemalla rajapinta paremmin. Voidaan esimerkiksi määrittää resurssi palauttamaan hieman enemmän tietoa, jolloin voidaan välttyä ylimääräisiltä kyselyiltä (mt).

GraphQL-rajapinnan kanssa voi myös esiintyä N + 1 -ongelma (Shapton, Thacker-Smith & Walkinshaw 2018). Tässä tapauksessa ongelma ei kuitenkaan ole erillisten HTTP-kyselyiden määrässä, koska GraphQL-rajapinnasta tiedot voidaan yleensä hakea yhdellä HTTP-kyselyllä. Ongelma esiintyy sen sijaan ensisijaisesti tietokannan tasolla, missä monimutkaiset kyselyt voivat tuottaa turhia kyselyitä tietokantaan (mt). Tarkastellaan ongelmaa kuvion 18 avulla, missä haetaan kaikki julkaisut, sekä niihin liittyvät kommentit.

```
query {
  posts {
    id
    title
    comments {
      id
      content
    }
  }
}
```

Kuvio 18. Kysely N + 1 -ongelman havainnollistamiseksi

Kuviossa 18 haetaan julkaisuja. Jokaisen julkaisun yhteydessä haetaan myös sille kuuluvat kommentit. Kuvion 18 kysely käytännössä tarkoittaa, että jokaista julkaisua kohden tehdään ylimääräinen kysely, jossa haetaan julkaisulle kuuluvat kommentit. Eli jos julkaisuja palautetaan 100 kappaletta, joudutaan erillisiä tietokantakyselyitä kommenttien hakemiseen tehdä myös 100 kappaletta. Tämän ongelman ratkaisemiseen (*Node.js*-alustalla) voidaan kuitenkin käyttää esimerkiksi Facebookin kehittämää *DataLoader*-kirjastoa (mt). *DataLoader*-kirjasto pystyy ryhmittämään yksittäiset kyselyt yhdeksi isommaksi eräksi, jolloin tietokantaan tehtävien kyselyiden määrää pystytään vähentämään (mt). Kuvion 18 kyselyn suorittaminen ilman *DataLoader*-kirjastoa tarkoittaa siis, että jokaisen haetun julkaisun kohdalla, tehdään

uusi tietokantakysely missä haetaan julkaisuun liittyvät kommentit. *DataLoader*-kirjaston avulla näitä erillisiä tietokantakyselyitä jokaisen julkaisun kommenttien hakemiseen ei tarvitse tehdä heti, vaan ne haetaan vasta kun kaikki haettavat kommentit ovat tiedossa. Eli tällöin kommentit haetaan jälkeenpäin yhdellä tietokantakyselyllä.

4.1.3 Välimuisti

Yksi REST-arkkitehtuurimallin määritelmässä olevissa rajoitteista liittyy välimuistin käyttöön REST-rajapinnoissa. Kyselyihin saatavissa vastauksissa pitää joko yleisesti tai erikseen määrittää, onko vastauksen sisältämä tieto mahdollista tallentaa välimuistiin. Mikäli näin on, käyttäjällä on siis mahdollisuus tallentaa kyselyn vastaus myöhempää käyttöä varten. Suorituskyky parantuu, jos välimuistissa olevaa vastausta pystytään myöhemmin hyödyntämään. (Fielding 2000.)

Jos REST-rajapintaa käytetään HTTP-protokollan kanssa, on tuki välimuistin käytölle valmiina. Tiettyjä HTTP-headereita käyttämällä, suurin osa eri resursseihin tehdyistä GET-pyynnöistä pystytään yleensä tallentamaan välimuistiin. Selaimet pystyvät esimerkiksi välimuistista hakemaan aiemmin haettua tietoa, ilman että palvelimelle joudutaan tekemään uutta pyyntöä. (Caching REST API Response N.d; HTTP caching 2019.)

Toisin kuin REST-rajapinnoissa, missä tiedot on hajautettu eri URI:en (resurssien) taakse, GraphQL-rajapinta tarjoaa vain yhden yhden URL:in minkä kautta kaikki tieto tullaan hakemaan. Tästä syystä välimuistin käyttö ei ole samalla tavalla suoraan mahdollista GraphQL:n kanssa. Tällöin tietojen tallentaminen välimuistiin on täysin GraphQL-rajapintaa käyttävät asiakkaan sekä kirjaston vastuulla. (GraphQL Best Practices n.d; Introduction to GraphQL 2019.)

Tietyt GraphQL-frontend kirjastot (*Apollo, Relay, Lokka*) sisältävät kuitenkin mahdollisuuden tallentaa kyselyt välimuistiin. Yksinkertaisemmat GraphQL-frontend-kirjastot (*graphql-request, graphql-js*) eivät kuitenkaan välttämättä sisällä tukea välimuistin käytölle.

4.1.4 Vastausten kompressointi

Jos GraphQL- sekä REST-arkkitehtuurimallin mukaisia järjestelmiä käytetään HTTP-protokollan kanssa, on silloin kummankin järjestelmän kohdalla mahdollista hyödyntää HTTP-protokollan määrittelemää tiedon kompressointia. Kompressoinnin käyttäminen vaatii tiettyjen HTTP-ylätunnusten asettamista. (Introduction to GraphQL 2019; REST Resource Representation Compression n.d.)

4.2 Turvallisuus

4.2.1 Kyselyiden rajoitus

REST-rajapinnoissa kyselyiden rajoittamiseen on eri keinoja. Kolme yleisintä keinoa ovat:

- Käyttäjäkohtaisten kyselyiden määrän rajoitus
- Palvelinkohtaisten kyselyiden määrän rajoitus
- Aluekohtaisten kyselyiden määrän rajoitus

REST-rajapinnoissa eri resursseihin kohdistuvien kyselyiden vaativuus pystytään tietämään etukäteen, jolloin kyselyiden rajoittamiseen voidaan valita oikeanlainen tapa. (Brien 2018; Sandoval 2016.)

GraphQL-rajapintaan tehtävät kyselyt ovat useasti laajempia sekä monimutkaisempia kuin mitä REST-rajapintojen tapauksessa. (Sandoval 2017). Yksi GraphQL-rajapintaan tehtävä kysely voi vaatia palvelimelta samanlaisen työmäärän kuin mitä useat REST-rajapintaan tehtävät kyselyt. Tällöin esimerkiksi yksinkertaisempi palvelinkohtaisten rajoitusten määrä ei välttämättä vielä riitä estämään järjestelmän kuormitusta. (Brien 2018).

Monimutkaiset sisäkkäiset kyselyt voivan pahimmillaan aiheuttaa palvelunestohyökkäyksen, mikäli palvelimen resurssit eivät riitä suorittamaan kyselyä (Stagno 2018). Kuviossa 19 on näkyvissä esimerkki haitallisesta sisäkkäisestä kyselystä, mikä tuottaa optimoinnin ja rajoitusten puuttuessa tietokantaan hyvin monta turhaa kyselyä.

```
query {
  posts {
    comments {
      post {
        comments {
          post {
            comments {
              post {
                comments {
                  id
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Kuvio 19. Haitallinen, sisäkkäinen kysely

Kuviossa 19 näkyvään kyselyyn voidaan kuitenkin varautua määrittämällä esimerkiksi rajoitteet sisäkkäisten kyselyiden tekemiseen tai määrittämällä kyselyille tietty sallittu suoritusaika (mt).

4.2.2 Virheenkäsittely

Virheiden käsittely REST-rajapinnoissa on melko helppoa, koska virheen tyyppi voidaan useasti tarkistaa vain HTTP-statuskoodista ja sen mukaan käsitellä sovelluksessa (Poniatowicz 2019).

GraphQL-sovelluksissa esiintyvien virheiden laatu on kuitenkin suurempi. Palvelimen, autentikoinnin, virheellisen kyselyn tai validoinnin seurauksena tapahtuvat ongelmat joudutaan käsittelemään erikseen. Palvelimella tapahtuva ongelma voi palauttaa HTTP-statuskoodin 5xx. Syntaksiltaan virheellinen kysely GraphQL-rajapinnalta palauttaa HTTP-statuskoodin 200, mutta tällöin virheet ovat mukana erillisenä kohtana palautetussa JSON-objektissa. (Mt.)

4.3 Kehittäminen

4.3.1 Järjestelmän näkyvyys

Jos REST-rajapintaa käytetään HTTP-protokollan kanssa, rajapintaan tehtäviin CRUD-operaatioihin (create, read, update, delete) käytetään yleensä HTTP-protokollan ominaisia metodeja kuten GET, POST, PUT, PATCH, DELETE. Näillä metodeilla on yleensä selvä yleinen merkitys siihen, minkä operaation ne kohdistavat tiettyyn resurssiin. Esimerkiksi GET-pyyntö tiettyyn resurssiin merkitsee sitä, että tietoa halutaan palauttaa tästä resurssista, DELETE-pyyntö resurssiin merkitsee, että sieltä halutaan poistaa jotain. (API design 2018; Wieruch 2018.) Nämä metodit yhdessä kuvaavien resurssinimien kanssa antavat yleensä selvän kuvan, mitä toimintoja rajapinta tarjoaa.

GraphQL:ssä ei käytetä HTTP-protokollan mukaisia metodeja operaatioiden nimeämiseen (Stubailo 2017). Esimerkiksi tiedon hakemiseen käytetään *query*-pääoperaatiotyyppiä sekä yksilöivää operaationimeä. Jos operaationimi ei kuvaa hyvin operaation seurauksena olevaa toimintaa, voi järjestelmän käyttäminen olla vaikeaa. Kehittäjillä on tässä vaiheessa suuri vastuu. Esimerkiksi uusien mutaatioiden nimeämisessä on syytä pyrkiä käyttämään mahdollisimman kuvaavia nimiä.

REST-rajapintojen kohdalla suurin haaste on itse resurssien nimeämisessä. On tietenkin tärkeää pyrkiä nimeämään myös rajapinnan sisäisesti käytettävät funktiot loogisesti sekä yhteneväisesti. Esimerkiksi ei ole kovin loogista käyttää yksittäisen resurssin listaukseen tarkoitettua funktiota sekä ”show”, että ”getSingle” nimillä. Näillä seikoilla ei kuitenkaan ole rajapinnan käyttäjälle merkitystä, koska ne ovat näkyviä ainoastaan itse rajapinnan kehittäjälle.

4.3.2 Dokumentointi

GraphQL:n spesifikaatiossa on määritelty millä tavalla dokumentointi voidaan tarjota tyyppimääritysten ohella. GraphQL-rajapintaa pystytään tutkimaan spesifikaatiossa määritetyn *introspection*-järjestelmän avulla. *Introspection*-järjestelmä mahdollistaa esimerkiksi rajapinnalta saatavilla olevien objektityyppien ja kyselyiden tutkimisen. Tämä mahdollistaa esimerkiksi dokumentoinnin generoimisen tai entistä paremman

GraphQL-tuen integroiduille kehitysympäristöille. *Introspection*-järjestelmän avulla voidaan myös aina olettaa, että dokumentaatio on ajan tasalla. (Farstad 2019; GraphQL 2018; Introduction to GraphQL 2019.)

Toisin kuin GraphQL:n *introspection*-järjestelmä, REST-arkkitehtuurimääritelmä ei itsessään ota mitään kantaa, miten rajapinnat tulisi dokumentoida. Tästä syystä, tarjolla onkin useita eri ratkaisuja REST-rajapintojen dokumentoimiseen. Esimerkkinä muutama:

- Swagger
- Apiary
- RAML

Eri REST-rajapintojen dokumentointi voi olla siis hyvinkin erilaisessa formaatissa ja hyvin erilaisella tasolla.

4.3.3 Versiointi

REST-rajapintoja kehitettäessä, suuriin vaatimusmuutoksiin reagoiminen yleensä tarkoittaa vanhan toiminnallisuuden muuttamista, mikä voi johtaa esimerkiksi rajapintaa hyödyntävien sovelluksien rikkoutumisena. Rikkovina muutoksina voidaan pitää tiedon palautusformaatin muuttumista tai tietyn rajapinnan osan poistumista. Rikkovien muutosten seurauksena rajapinnan versionumeroa pitää nostaa. REST-rajapintojen versiointiin on käytössä erilaisia tapoja, yleisimpinä on versionumeron määrittämien joko URL:ssa tai HTTP-headereissa. (REST API Versioning N.d.)

REST-rajapintojen versioinnissa on myös otettava huomioon aiempien versioiden toimivuus sekä useiden eri versioiden ylläpito. Eri versioiden ylläpito voi vaatia kehittäjiltä paljon aikaa sekä työtä. (Curry 2017.)

GraphQL:n tapauksessa versioinnin ongelma ei välttämättä ole yhtä suuri. GraphQL-rajapinnan käyttäjä joutuu eksplisiittisesti määrittämään kyselyssä halutut kentät, jolloin rajapintaan voidaan luoda uusia kenttiä sekä toiminnallisuutta ilman sen kummempaa vaikututusta aiempiin kyselyihin. Kenttä pystytään myös määrittämään vanhentuneeksi, jolloin esimerkiksi *introspection*-järjestelmä pystyy varoittamaan kehittäjiä kyseisen kentän käytöstä. Vanhojen kenttien uudelleennimeämistä ei

suositella, jotta yhteensopivuus vanhojen kyselyiden kanssa säilyy. Tästä syystä erillistä versiointia GraphQL-rajapinnoille ei tarvitse tehdä. Mikään ei kuitenkaan estä versioinnin lisäämistä GraphQL-rajapintoihin, sitä ei kuitenkaan suositella. (GraphQL 2018; GraphQL Best Practices 2019, Versioning; Tarvainen 2016.)

Muutosten tekeminen ja ydinlogiikan muokkaaminen voi olla GraphQL-rajapinnoissa olla helpompaa. Jokaiselle kentälle pystytään erikseen määrittämään, millä tavalla tieto halutaan palauttaa. Tämä mahdollistaa muutosten tekemisen palvelimen puolella, ilman että GraphQL-rajapinnassa näkyisi ulospäin muutoksia.

4.3.4 Työkalut

REST-arkkitehtuurimallin pohjalta tehtävien rajapintojen kehityksen helpottamiseksi löytyy jo paljon erilaisia työkaluja. Kokonaisia kirjastoja eri ohjelmointikielien päälle löytyy paljon, esimerkkinä muutama:

- Django REST framework: Python-ohjelmointikielen avulla rakennettu sovelluskehys.
- Restify: Node.js-alustan päälle rakennetty sovelluskehys REST-rajapintojen kehittämiseen.
- RESTit: Go-ohjelmointikielen avulla rakennettu pieni kirjasto REST-rajapintojen integraatiotestaukseen.
- Dingo API: Laravel- sekä Lumen-ohjelmistokehyksille rakennettu kirjasto REST-rajapintojen kehittämiseen.

(Riady 2016.)

Valmiita kirjastoja sekä sovelluskehyksiä REST-rajapintojen kehitykseen löytyy paljon. Sovelluskehykset helpottavat REST-rajapintojen kehityksessä ilmeneviä haasteita. Laravel-sovelluskehys tarjoaa esimerkiksi automaattisen koodin generoimisen kontrollereille, jotka käsittelevät sovelluksen rajapintaan tulevia pyyntöjä. Laravel-sovelluskehyksestä löytyy myös apufunktioita pyyntöjen validoimiseen sekä se tarjoaa myös paginaation tulosten sivuttamiseen. (Documentation N.d.)

PHP-ohjelmointikielelle löytyy myös kirjasto nimeltään Fractal Transformers. Kyseisen kirjaston avulla yhdestä rajapinnan URI:sta pystytään saada yhden resurssin lisäksi myös siihen liittyviä muita resursseja. (Transformers N.d.) Kuviossa 20 on

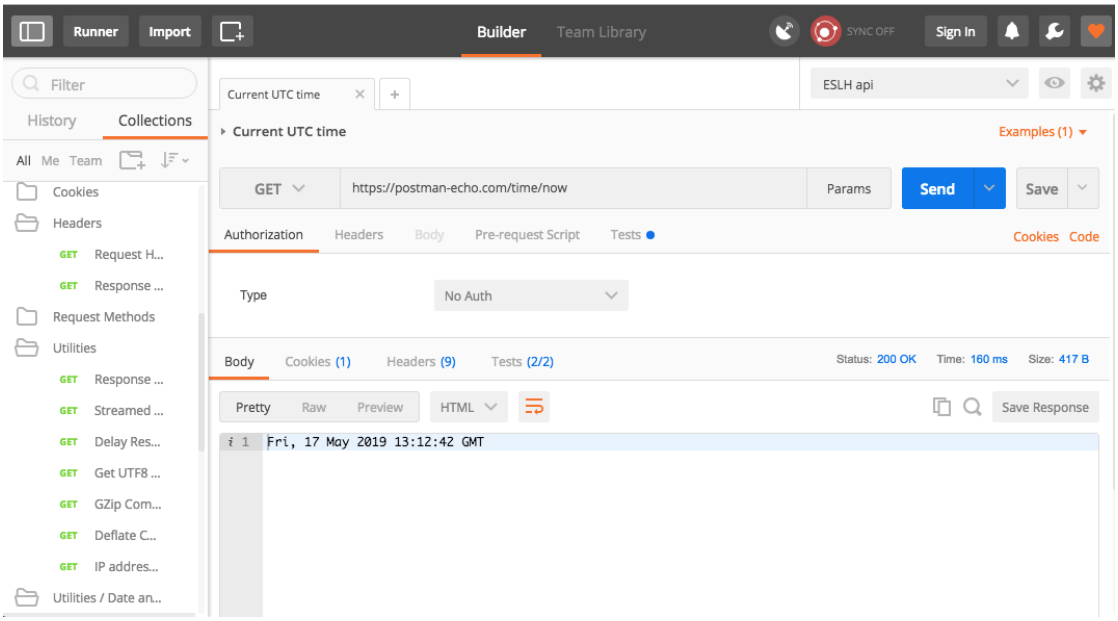
näkyvissä esimerkki, missä yhdestä URI:sta voidaan saada sekä julkaisut ja julkaisujen kommentit.

```
curl -X GET 'http://api.test/V1/posts?include=comments'
```

Kuvio 20. Usean resurssin lataaminen yhdellä HTTP-pyynnöllä

Kuvion 20 mukainen kysely voi huomattavasti vähentää tarvittavien kyselyiden määrä ja osittain ratkaista myös REST-rajapinnoissa ilmenevät N + 1 -ongelmat, koska nyt julkaisuun liittyvät kommentit voidaan myös ladata julkaisun yhteydessä. Tälläisten kyselyiden käyttämistä on kuitenkin aina pohdittava etukäteen, koska mikäli liitetylle tiedolle ei ole käyttöä, joudutaan verkon yli lataamaan vielä entistä enemmän turhaa tietoa.

REST-rajapintojen testaamiseen löytyy myös hyviä työkaluja kuten Postman sekä PAW. Kuviossa 21 on näkyvissä kuvakaappaus Postman-työkalun käyttöliittymästä.



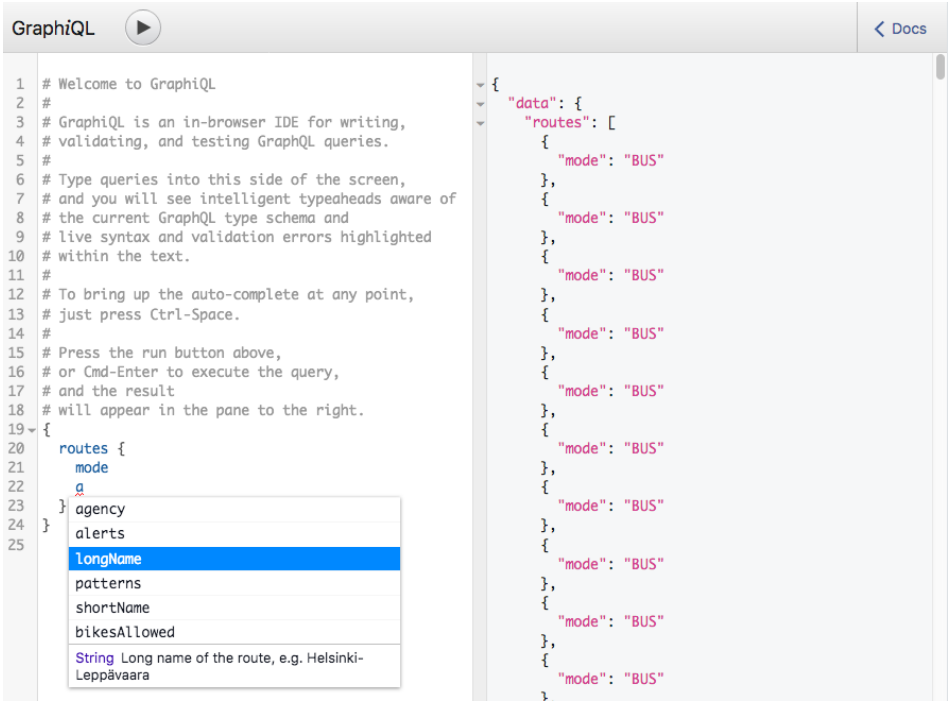
Kuvio 21. Postman-työkalu

Postman-työkalun pystytään suunnittelemaan, dokumentoimaan, testaamaan, käyttämään sekä monitoroimaan rajapintoja. Työkalun avulla tiimit pystyvät myös yhdessä osallistumaan rajapintojen kehittämiseen. (The Only Complete API Platform N.d.)

Hyvien työkalujen ansiosta kehitystiimit pystyvät nopeasti luomaan uutta toiminnallisuutta rajapintoihin. Tämä on erityisen tärkeää projekteissa, missä kehitystahdin on oltava nopeaa esimerkiksi oman tai asiakkaan liiketoiminnan takia. Mikäli työkalujen käyttäminen ei ole kehittäjille luontevaa, voi se johtaa tiimin sisäisen moraalin laskuun, mikä voi entistään hidastaa projektin valmistumista.

GraphQL:n suosion myötä se on nopeasti kerännyt ympärilleen useita erilaisia kehitystä, monitorointia ja optimointia helpottavia ohjelmistoja. Yksi yleinen työkalu GraphQL-rajapintojen tutkimiseen sekä kyselyiden ja mutaatioiden tekemiseen on GraphiQL. GraphiQL on selaimessa toimiva interaktiivinen työkalu, joka on yleensä saatavilla monen julkisen GraphQL-rajapinnan yhteydessä. (GraphQL or Bust 2018, S. 92-93.)

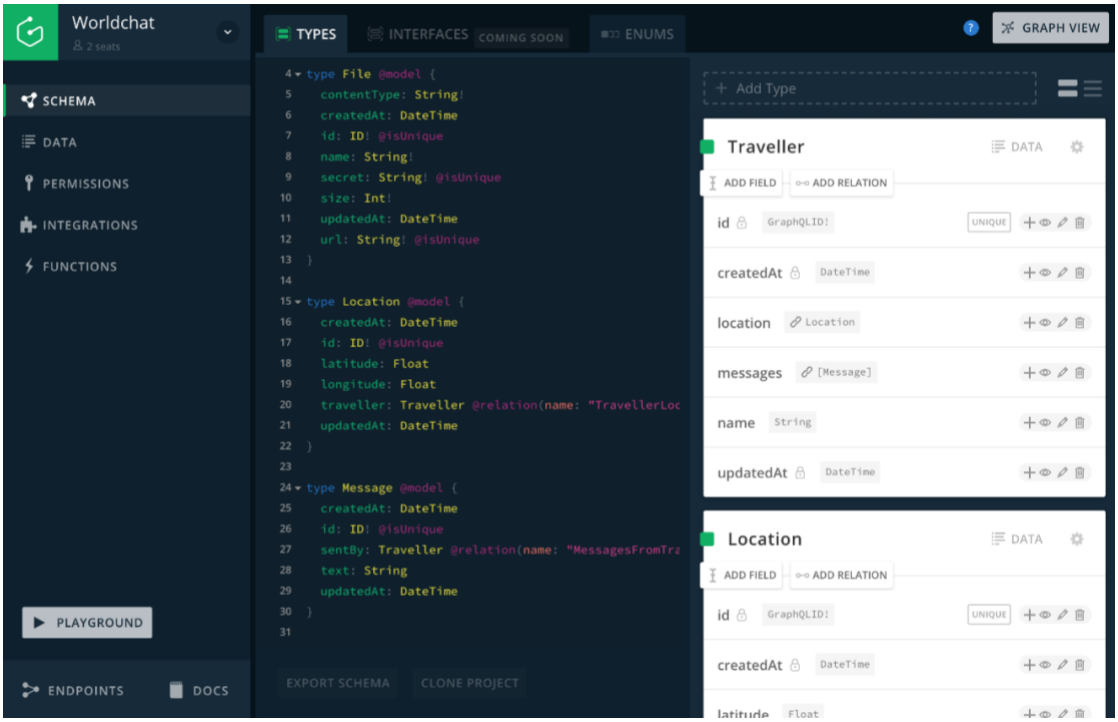
Kuviossa 22 on nähtävissä esimerkki GraphiQL työkalun käyttämisestä.



Kuvio 22. GraphiQL-työkalu

Kuviossa 22 GraphiQL-työkalua käytetään HSL:n (Helsingin Seudun Liikenne) rajapinnan tutkimiseen. Kuviosta voidaan myös nähdä miten avautuvan ”pop-up” ikkunan avulla pystyään valitsemaan tai automaattisesti täydentämään valitun kentän nimi. Uuden rajapinnan sisältämää tietoa pystytään siis tutkimaan vaivattomasti.

Graphcool tarjoaa kirjaston ja myös verkossa sijaitsevan alustan, jonka avulla pystytään helposti luomaan uusia taustajärjestelmiä tietokantoineen GraphQL-rajapintojen kehittämistä varten. (GraphQL or Bust 2018, S. 99-100; Self-Hosted GraphQL Baas n.d.) Kuviossa 23 on näkyvissä kuvakaappaus Graphcool-työkalun hallintapaneelista.

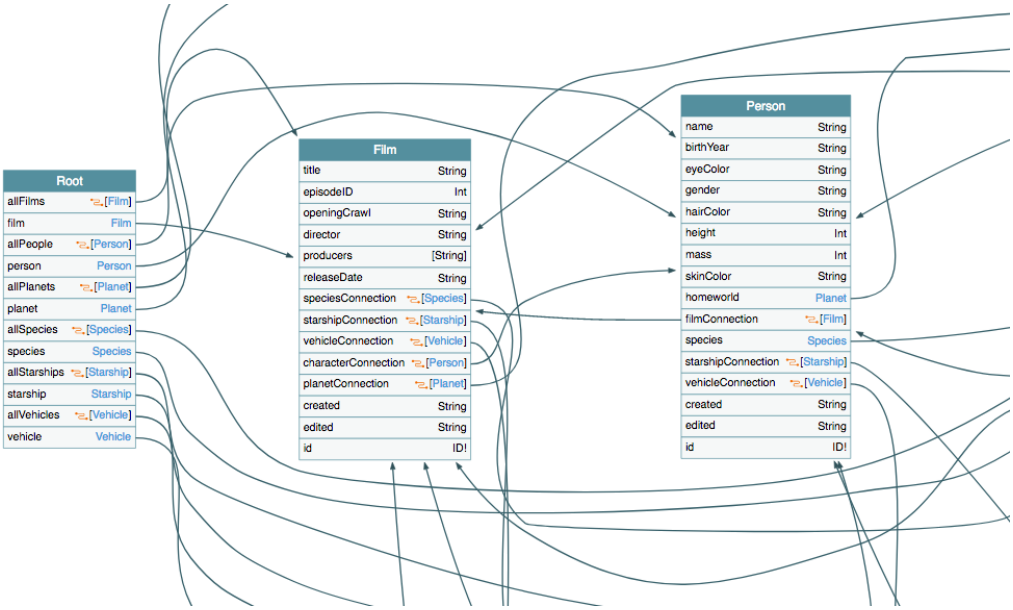


Kuvio 23. Näkymä Graphcool-työkalun tarjoamasta hallintapaneelista

Graphcool-mahdollistaa Webhookkien käyttämisen tiedon palauttamiseen eri lähteistä, jolloin se voidaan liittää esimerkiksi olemassa olevan REST-rajapinnan rinnalle. Tällöin jo olemassa olevia rajapintoja ja sovelluksia voidaan hyödyntää. (Basics n.d.)

Graphcool:in kaltaisten työkalujen ansiosta uuden GraphQL-rajapinnan pystytys voidaan hoitaa hyvinkin nopeasti, koska aikaa ei tarvitse käyttää erikseen esimerkiksi tietokantojen konfiguroimiseen.

GraphQL-rajapinnan sisältämän scheman visualisoimiseksi voidaan käyttää GraphQL Voyager-työkalua. GraphQL Voyager parsii rajapinnan tarjoamaa tietoa, sekä luo sen pohjalta visuaalisen graafin. Graafin avulla pystytään tarkastelemaan eri objektien sekä kenttien suhteita toisiinsa. Kuviossa 24 on nähtävissä pieni osa GrapQl Voyager:in laatimasta graafista.



Kuvio 24. Scheman visualisoiminen GraphQL Voyager:in avulla

Kehittäjät sekä rajapinnan käyttäjät voivat visuaalisten graafien avulla saada paljon nopeammin ymmärryksen, millä tavalla tieto on taustajärjestelmässä mallinnettu. Kuvaava graafi, yhdistettynä GraphiQL-työkalun käyttämiseen tarjoaa tehokkaan tavan tutkia uutta GraphQL-rajapintaa. Näiden työkalujen avulla kokonaisvaltaisen yleiskuvan saaminen GraphQL-rajapinnasta mielestäni hyvin helppoa.

4.3.5 Mikropalveluarkkitehtuurit

Apollo-kirjasto tarjoaa *Node.js*-ympäristössä suoritettaville GraphQL-sovelluksille ominaisuuden nimeltään *Schema stitching*. Tämän omaisuuden avulla pystytään yhdistämään useita eri GraphQL-palvelujen skeemoja yhdeksi sulautetuksi skeemaksi. Käytännössä siis useat eri GraphQL-rajapinnat voidaan yhdistää yhden kootun GraphQL-rajapinnan alle. Tämä mahdollistaa esimerkiksi laajemman monoliittisen järjestelmän pilkkomisen pienempiin mikropalveluihin. Mikropalveluja pystytään sen jälkeen kehittämään sekä julkaisemaan erillään muusta järjestelmästä. (Schema stitching n.d.)

Erilliset GraphQL-rajapintojen sekä skeemojen sijainnilla ei skeemojen yhdistämisvaiheessa ole väliä. Ne voivat sijaita samalla tai erillisellä palvelimella. Tämä mahdollistaa esimerkiksi kolmannen osapuolen tarjoamien GraphQL-rajapintojen yhdistämisen osaksi omaa GraphQL-rajapintaa. (Mt.)

REST-arkkitehtuurimallin pohjalta toteutetut mikropalvelut ovat myös suosittu valinta. Tämä luultavimmin johtuu niiden yksinkertaisesta luonteesta, koska ne eivät toimiakseen tarvi ylimääräisiä tukea intrastruktuurilta. (Williams 2015.)

Mikropalvelujen kehittämisen tapauksessa ei välttämättä ole helppo sanoa kumpaa teknologiaa kannattaisi käyttää. Mikäli mikropalveluiden on tarkoitus ratkaista tietty ongelma osana laajempaa järjestelmää, on mielestäni ongelman ratkaisemiseen parasta käyttää aina sillä hetkellä järkevintä vaihtoehtoa.

Yksi GraphQL:n etu mikropalveluarkkitehtuurissa on kuitenkin se, että sen avulla pystytään kasaamaan useista eri mikropalveluista tieto yhden ja saman päätepisteen alle. Tällöin voidaan myös yhdistää muita REST-rajapintoja osaksi isompaa GraphQL-sovellusta. *Apollo*-kirjaston tarjoama *Schema stitching* voi myös mahdollisesti tehdä järjestelmän kokonaiskuvan ymmärtämisen vaikeaksi, mikäli eri mikropalveluissa sijaitsevat skeemat tulevat vahvasti riippuvaisiksi toisistaan eivätkä kykene toimimaan itsenäisesti.

5 Tulokset

5.1 GraphQL:n käytön kannattavuus

Ideologialtaan GraphQL eroaa REST-arkkitehtuurimallista siinä, että se siirtää tiedon rakenteen määrittämisen täysin rajapintaa käyttävälle osapuolelle. REST-arkkitehtuurimalliin pohjautuvissa rajapinnoissa taas palvelin päättää, miten tieto on mallinnettu. Tämä on rajapintaa käyttävän osapuolen kannalta hyvä asia, koska tällöin rajapinnan voidaan katsoa mukautuvan käyttäjän vaatimuksiin, eikä toisinpäin. Rajapinnan mukautuminen eri osapuolten yksilöllisiin vaatimuksiin mahdollistaa, että jokainen osapuoli pystyy nyt itse optimoimaan tiedonsiirron määrän. Projekteissa ja sovelluksissa missä rajapinnan halutaan tarjoavan sisältöä esimerkiksi useille eri päätelaitteille (selain, puhelin, tablet) on GraphQL:n käytöstä tällöin hyötyä.

Projektit missä suorituskyvyn merkitys on tärkeää, GraphQL:n valitseminen on hyvin perusteltua. Turhan tiedon hakeminen rajapinnalta vähenee, koska halutut kentät pitää eksplisiittisesti määrittää kyselyyn. Ylimääräiset kyselyt myös vähenevät, koska kaikki tarvittava tieto voidaan saada yhdellä ja samalla kyselyllä. REST-rajapintojen tapauksissa palvelimille tehtävien ylimääräisten kyselyiden sekä tiedonsiirron optimointia voidaan parantaa eri kirjastojen sekä sovelluskehyksien avulla, mikäli tarvittavaa tieto ei saada haettua yhdellä kyselyllä. Nämä keinot kuitenkin yleensä vaihtelevat kirjastojen sekä sovelluskehyksien mukaan, jolloin niiden käyttäminen jokaisessa projektissa ei välttämättä ole aina mahdollista. GraphQL tarjoaa myös *DataLoader*-kirjaston kaltaisten työkalujen avulla paremman keinon optimoida tietokantaan kohdistuvien kyselyiden määrän. REST-rajapinnoissa tietokantaan kohdistuvia kyselyitä ei pystytä samalla tavalla optimoimaan, koska jokainen resurssiin kohdistettu kysely on irrallinen muista kyselyistä. REST-rajapinnoissa kyselyitä voidaan vähentää esimerkiksi niiden määrää rajoittamalla, mikä voi kuitenkin tuottaa ongelmia rajapintaa käyttävälle osapuolelle.

On kuitenkin rajatapauksia, missä esimerkiksi yksinkertaiset frontend-sovellukset, jotka tekevät HTTP-protokollan päälle tehtyihin REST-rajapintoihin hyvin yksinkertaisia kyselyitä ja luottavat vahvasti välimuistin käyttämiseen, voivat jossain määrin jopa menettää suorituskykyään, mikäli käytettävä GraphQL:n kirjasto ei tue vastausten tallentamista välimuistiin.

Jos projektin vaatimuksien mukaan mallinnettu tieto ja sen rakenne pystytään tarkasti määrittelemään projektin alkaessa ja suuria tiedon mallintamisen tai rakenteeseen liittyviä muutoksia ei projektin tulevaisuudessa ole tiedossa, voi GraphQL:n käytöstä saatavat hyödyt jäädä pieniksi. Varsinkin jos rajapinta ei tule sisältämään kovin useaa erilaista resurssia ja sitä käytetään ainoastaan yhden sovelluksen tai päätelaitteen osalta, voi GraphQL ainoastaan lisätä kompleksisuutta pohjimmiltaan yksinkertaiseen projektiin.

Mikäli projektin alkaessa ei ole täysin varmaa, millä tavalla tieto pitäisi mallintaa on GraphQL:n tarjoama rajapinnan ”versioton” kehitysmalli hieman ketterämpi vaihtoehto, kun projektin vaatimukset muuttuvat. Tämä voi projektin myöhemmässä

vaiheessa säästä aikaa verrattuna REST-rajapintoihin, missä vanhempia rajapinnan versioita joudutaan myös osittain ylläpitämään.

GraphQL:n mukana tulevan kompleksisuuden takia, on myös tärkeää ottaa huomioon millainen ymmärrys sekä tietämys projektin kehittäjillä teknologiasta on. Koska GraphQL-rajapinnan käyttäjillä on valta määrittää, millaisessa rakenteessa tieto palautetaan, on hyvin monimutkaisten sekä raskaiden kyselyiden esiintyminen mahdollista. Mikäli kehittäjät eivät osaa varautua tällaisiin tilanteisiin, on rajapinnan, sekä sen myötä koko järjestelmän ja sovelluksen toimintavarmuus vaarassa.

5.2 GraphQL- ja REST-rajapinnan kehittämisen erot

Vastuu tiedon rakenteen määrittämisellä on GraphQL-rajapinnoissa sitä käyttävällä osapuolella. Tästä syystä rajapinnan kehittäjän vastuu on lisääntynyt järjestelmän turvallisuuden ja ylläpidon osalta. Useat eri GraphQL-teknologian mukana tulevat työkalut ovat kuitenkin tehokkaita, joiden avulla kehitystä voidaan helpottaa.

REST-rajapintojen kehittämiseen löytyy sovelluskehyksiä, joiden avulla kehitystä pyritään nopeuttamaan. Mikäli kokemusta tällaisista sovelluskehyksistä löytyy, voi rajapintojen kehittäminen olla myös hyvin nopeaa. Useat REST-rajapintojen kehittämiseen tarkoitetut sovelluskehykset pyrkivät tarjoamaan keinoja millä REST-arkkitehtuurimallin määrittelemiä rajoitteita pystytään ”ohittamaan”, tarjoamalla esimerkiksi mahdollisuuden hakea useita resursseja yhdellä kyselyllä. On myös huomattava, että REST-rajapintojen suunnittelu voi rajoitteiden takia olla hankalaa, verrattuna GraphQL-rajapintojen suunniteluun, missä tieto mallinnetaan suoraan eri olioiden välisillä suhteilla.

5.3 GraphQL:n käyttämisen mahdollisuudet ja vaikeudet

Uusien kehittäjien sisäänajo projekteihin on helpompaa, mikäli kehitettävistä rajapinnoista löytyy asianmukaiset dokumentaatiot. Projektin kannalta on aina parempi, mitä nopeammin kehittäjät saadaan tekemään tuottavaa työtä.

GraphQL- sekä REST-rajapintojen kehittämiseen löytyy paljon työkaluja. GraphQL:n tarjoaman vahvasti tyypitetyn scheman ansiosta sekä *introspection*-järjestelmän

avulla monet työkalut pystyvät tarjoamaan erittäin hyvin GraphQL-rajapintoihin integroituvia kehitystyökaluja. Tulevaisuudessa on myös todennäköistä, että GraphQL-rajapintojen kehitykseen tarkoitetut työkalut parantuvat entisestään.

Suuri osa löydetyistä GraphQL-kirjastoista sekä työkaluista on kuitenkin rakennettu JavaScriptin sekä *Node.js*-teknologian avulla. Mikäli osaamista näiden teknologioiden käyttöön ei ole, voi useita laadukkaita GraphQL-rajapintojen kehittämiseen tarkoitettuja työkaluja jäädä hyödyntämättä.

5.4 Yhteenveto

GraphQL on kannattava valinta projekteihin missä rajapinnan on tärkeää mukautua useiden eri käyttäjien tiedon haun eriäviin vaatimuksiin tiedon rakenteen ja sisällön suhteen. Se tarjoaa hyvin mahdollisuuksia rajapinnan suorituskyvyn sekä tiedonsiirron määrän optimointiin. Sen käyttöönotto kuitenkin vaati aiheeseen perehtymistä, jotta järjestelmän lisääntynyt monimutkaisuus ei tuota ongelmia tulevaisuudessa.

REST-arkkitehtuurimallin pohjalta tehdyt rajapinnat ovat kannattava valinta projekteihin missä vaatimukset tiedon suhteen tiedetään etukäteen, eikä suuria muutoksia tiedon suhteen ole tulevaisuudessa tiedossa. Se on myös hyvä valinta silloin, kun rajapinnalla on esimerkiksi ainoastaan yksi käyttäjä. Tällöin tiedonsiirto sekä kyselyt pystytään optimoimaan tämän yhden käyttäjän tarpeisiin.

Lopullinen valinta GraphQL- sekä REST-rajapinnan kehittämisen välillä on kuitenkin aina tehtävä projektikohtaisesti.

6 Pohdinta

6.1 Tavoitteiden saavuttaminen

Työssä vertailtiin kahta erilaista teknologiaa rajapintojen toteuttamiseen. Alun perin yksi opinnäytetyön tavoitteista oli myös testata ja vertailla GraphQL- sekä REST-rajapintojen suorituskykyä. Tämä kuitenkin päätettiin myöhemmin jättää pois, koska

jäljellä olevien resurssien valossa sen ei koettu olevan tarpeeksi tärkeä asia, mihin aikaa olisi kannattanut käyttää.

Aiheen laajuus tuotti työn tekemisessä haasteita. Työn alussa ei oltu määritelty tarkkoja aihealueita, mitä näiden kahden eri teknologian välillä haluttiin vertailla. Mikäli työn alkaessa olisi keskitytty tiettyjen projektien vaatimuksiin, esimerkiksi selainympäristössä toimivan yhden sivun sovelluksen käyttämän rajapinnan vaatimuksiin, olisi vertailu voitu kohdistaa tarkemmin vastaamaan näitä olosuhteita. Tällöin kuitenkin työn tulokset eivät välttämättä olisi yhtä yleisesti sovellettavia.

Vertailun alla olleet teknologiat olivat mielenkiintoisia. Aiempaa kokemusta REST-rajapintojen kehittämisestä sekä käyttämisestä oli kertynyt noin kahden vuoden ohjelmistokehitystyön seurauksena. Syvällistä tietoa tai ymmärrystä REST-arkkitehtuurimallista ei kuitenkaan entuudestaan ollut. Aiempi kosketus GraphQL-teknologiaan oli tullut GraphQL Finland -tapahtuman sekä omatoimisen tutkimisen perusteella. Käytännön kokemusta GraphQL-teknologiasta ei kuitenkaan ollut.

Kuitenkin kummankin teknologian konkreettisen havainnoinnin seurauksena voidaan nostaa muutamia huomioita ylös. Sovelluksissa, missä rajapinnan ensisijainen tehtävä on palvella erillistä frontend-sovellusta, on GraphQL:n tarjoama valinnanvapaus kyselyiden sisältämän tiedon rakenteen suhteen suuri etu frontend-kehityksessä. Tämä mahdollistaa helposti rikkaiden käyttöliittymien tekemistä, missä rajapinnalta tarvittavaa tieto tarvitaan paljon. Frontend-sovellusten kehitysnopeutta ajatellen on paljon helpompaa kirjoittaa vaadittava tieto yhtenä GraphQL-kyselynä, kuin alkaa tekemään koodissa useita eri kyselyitä eri resursseihin, sekä sen jälkeen yhdistelemään saatuja tuloksia tarvittavaan muotoon.

Taustajärjestelmän kehityksen osalta vaikutti, että saman toiminnallisuuden tekeminen rajapintaan GraphQL-teknologialla oli aluksi hitaampaa. Tähän osaksi vaikuttaa myös se, että REST-rajapintojen kehittämisestä oli aikaisempaa kokemusta. Varsinkin kyselyiden optimointi *DataLoader*-kirjaston avulla vei melko paljon aikaa. Sitä oltaisiin voitu säästää, mikäli kirjaston käytön aloittamisessa, sen yhteydessä tarjottuun dokumentointiin oltaisiin tutustuttu paremmin.

Kehitysvauhti frontend-sovelluksissa vaikutti nopeutuvan GraphQL:n käytön myötä, kun taas taustajärjestelmän puolella se hieman hidastui. Kun kokemusta GraphQL:n käytön myötä kertyy enemmän, on kuitenkin hyvin mahdollista, että myös taustajärjestelmän kehitysvauhti nopeutuu entisestään.

GraphQL-rajapintojen kehitykseen tarjotut työkalut ovat hyvin tehokkaita. REST-rajapintojen kehitykseen verrattuna, GraphQL:n tarjoama tyypitys ja *introspection*-järjestelmä antavat paljon edullisemman aseman laadukkaampien työkalujen tekemisen tälle alustalle. REST-rajapintojen kehitykseen löytyvät Postman- ja PAW-työkalut ovat kuitenkin nykytilassaan myös erittäin tehokkaita apuvälineitä REST-rajapintojen kehittämiseen. Postman esimerkiksi tarjoaa mahdollisuuden muuttujien käyttämiselle URI:ssa ja HTTP-ylätunnuksissa. Postmanilla tehtyjä kyselyitä, sekä niiden palauttamia tietoja voidaan myös tallentaa muuttujiin sekä lisätä myöhemmin esimerkiksi osaksi uutta kyselyä HTTP-headereiden muodossa. Tämä mahdollistaa rajapinnan testaamisen esimerkiksi sellaisen järjestelmän kanssa, joka vaatii kirjautumista tai autentikaatiota. Vaihtoehtoisesti GraphiQL-työkalusta ei löytynyt samanlaista ominaisuutta.

GraphQL tuo mukanaan paljon ominaisuuksia mitä REST-arkkitehtuurimallin mukaan rakennetut rajapinnat joutuisivat myös itse tekemään. GraphQL:n tarjoama tyypityksen avulla rajapintaan pystytään tekemään suoraan vahva validointi tiedon suhteen. REST-rajapinnoissa tämän saman toiminnallisuuden saavuttamiseen joudutaan yleensä käyttämään jotain sovelluskehystä. GraphQL:n tarjoama tyypitys tarjoaa myös kyselyiden tekemiseen frontendin puolella hyvän kokemuksen. Schemassa määritetyt tyypit ovat nyt yhtenäiset koko sovelluksen erillisillä osa-alueilla. Toisin kuin REST-arkkitehtuurimallin pohjalta rakennetuissa rajapinnoissa, missä itse rajapinta tuo omanlaisen ”muurin” sovelluksen front- sekä backendin väliin.

Myös GraphQL:n sisäänrakennettu *Subscription*-operaattori voi poistaa tarpeen ylimääräisen kirjaston tai palvelun käyttämiselle sellaisissa tapauksissa, joissa sovelluksen halutaan kuuntelevan tiedon muutoksia reaaliaikaisesti. Hyviä esimerkkejä ovat esimerkiksi erilaiset chat-sovellukset, joissa uusien viestin

saapumisesta pitää tehdä ilmoitus käyttäjälle. Ilman *Subscription*-operaattoria, tällaisen toiminnallisuuden tekemiseen pitäisi käyttää esimerkiksi WebSocketteja.

GraphQL-rajapinnan tarjoamat virheet olivat myös joskus vaikeammin tulkittavissa verrattuna REST-rajapintaan, jossa virheet yleensä palautetaan HTTP-statuskoodin avulla. Tämä voi tuottaa esimerkiksi frontend-sovelluksissa lisää työtä. Jos palvelin, joka ajaa GraphQL-rajapintaa ei itsessään toimi oikein, on statuskoodi erilainen verrattuna esimerkiksi itse GraphQL-schemassa tai kyselyn syntaksissa olevaan virheeseen. Näiden eri tilanteiden varalle kuitenkin joudutaan tekemään virheenkäsittelyä, jotta sovelluksen käyttäjät eivät jää kärsimään mahdollisesti rikkinäisestä sovelluksesta.

Työn tietoperustassa esitellyt luvut REST-arkkitehtuurimalli sekä GraphQL ovat pituudeltaan hyvin erilaiset. Tämä johtuu siitä, että REST-arkkitehtuurimalli itsessään on melko lyhyt sekä yksinkertainen verrattuna GraphQL:n määritelmään, joka tuo mukanaan oman kyselykielen sekä tyyppijärjestelmän määrittelyn.

Työssä käytettiin paljon aikaa GraphQL:n teknisen dokumentaation lukemiseen sekä sen opetteluun. Tämä mahdollisti teknologian konkreettisen käyttämisen, jonka avulla pystyttiin vahvistamaan tietoperustassa esiintyviä havaintoja. Työ oltaisiin voitu myös toteuttaa ilman konkreettista kosketusta teknologioihin, jolloin lopputulokset olisivat pohjautuneet ainoastaan lähteistä saatuihin tietoihin. Tämä kuitenkin olisi voinut vaikuttaa saatuihin tulokseen epärealistisella tavalla, koska kaikkia teknologioiden tuomia kipupisteitä ei välttämättä pysty havainnoimaan esimerkiksi kehittäjän näkökulmasta ainoastaan dokumentaation sekä olemassa olevan tiedon perusteella.

Työn lähtökohdissa esitettyihin kysymyksiin verrattuna tulokset olivat tavoitteen mukaisia. Kaikkiin työn alussa määritettyihin tutkimuskysymyksiin saatiin vastaus.

6.2 Tulosten luotettavuus ja pätevyys

Laadullista tutkimusta tehdessä, tutkimusaineistona voidaan käyttää esimerkiksi erilaisia dokumentteja, videoita, äänitteitä sekä verkkosivuja. Kerätystä aineistosta

tutkija tekee valitun analyysimenetelmän mukaan tulkinnan sekä johtopäätöksen. (Kananen 2019, 28.)

GraphQL- sekä REST-rajapintojen rakentamiseen sekä käyttämiseen löytyy paljon erilaisia oppaita sekä kirjallisuutta. Kirjallisuutta näiden kahden teknologian väliseen syvempään vertailuun ei kuitenkaan vielä paljoa löydy, tästä syystä suurin osa vertailuun saadusta tietoperustasta on peräisin eri GraphQL- sekä REST-rajapintojen kehittäjien laatimista sähköisistä artikkeleista sekä kirjoista. Tulokset laadittiin kerätyn tutkimusaineiston havainnoinnin sekä kummankin teknologian käyttämisestä saatujen omakohtaisten kokemuksien kautta.

Työn laatua sekä luotettavuutta mitataan reliabiliteetin sekä validiteetin avulla. Reliabiliteetti ilmaisee tulosten pysyvyyttä ja validiteetti puolestaan sitä, että tutkimuksessa ollaan keskitytty tutkimaan relevantteja asioita. Tutkimuksessa kerätyn aineiston tulee olla aitoa, sekä sitä tulee olla riittävästi, jotta aineiston perusteella tehdyt tulkinnat ja johtopäätökset ovat oikeita. (Kananen 2019, 31.)

Laadullista tutkimusta tehdessä ei kuitenkaan ole tarkkoja säännöksiä siihen millainen aineistomäärä on riittävä (mts. 55).

Tutkimus oli haastava, koska teknologioiden vertailuun vaadittua aineistoa joutui hakemaan useista eri lähteistä sekä yhdistämään niistä tutkimuksen kannalta olennaiset osat osaksi työtä. Koska suurin osa aineistosta pohjautui yksittäisten kirjoittajien omakohtaisiin kokemuksiin, oli lähteiden välillä myös eriäviä mielipiteitä. Hyvien sekä laadukkaiden lähteiden etsiminen oli itselleni yksi opinnäytetyön suurimmista haasteista. Myös teknologioiden välinen kriittinen vertailu oli haasteellista, koska GraphQL:n mukana tuotu uutuudenviehätys ei saa vaikuttaa vertailun tuloksiin. Aineistoa kerättiin useista eri lähteistä, mutta niiden sisältämät, osaksi eriävät mielipiteet voivat vaikuttaa kielteisesti aineiston pohjalta tehtyihin johtopäätöksiin. Tämä voi puolestaan vähentää tutkimuksen luotettavuutta.

GraphQL itsessään on melko uusi teknologia ja sen spesifikaation sekä oheisten työkalujen kehitys jatkuu tulevaisuudessa. Tästä syystä johtopäätöksissä esiintyvät havainnot voivat vanheta tulevaisuudessa, kun mielipiteet, havainnot sekä kokemukset teknologian käyttämisestä lisääntyvät.

Työn aineiston perusteella johdetut tulokset kuitenkin tukevat suurimmalta osin aineistossa esiin nousseita huomioita. Esimerkiksi GraphQL:n osalta koetaan, että sen suorituskyky ja mahdollisuus kyselyiden optimoinnille on parempi verrattuna REST-rajapintoihin.

6.3 Tulosten yleistettävyys

Laadullinen tutkimus pyrkii yleistämisen sijasta ymmärtämään tutkittavaa ilmiötä. Tulosten siirrettävyyteen voidaan kuitenkin myötävaikuttaa kuvaamalla tarkasti lähtötilanne ja oletukset. Tällöin on siirtäjän vastuulla päätellä, voiko tutkimustuloksia hyödyntää hänen tapauksessaan. (Kananen 2015, 353.)

Tässä tutkimuksessa toimeksiantajan asettamat vaatimukset eivät rajoittuneet yhdenkään tietyn projektin reunaehtoihin. Tästä syystä työn tulokset ovat sovellettavissa yleisesti kaikkiin projekteihin, missä valinta GraphQL- sekä REST-rajapintojen välillä on aiheellinen.

Työn seurauksena syntyneistä tuloksista sekä pohdinnoista toivotaan olevan hyötyä toimeksiantajan tulevien projekteiden teknologiavalinnoissa. Toimeksiantajan lisäksi, tämän työn tuloksista toivotaan olevan hyötyä myös kaikille muille osapuolille, jotka pohtivat projektinsa teknologiavalintaa näiden kahden teknologian välillä.

6.4 Jatkokehitys

Tämän opinnäytetyön pohjalta tehtävään jatkokehitykseen voitaisiin valita vertailun pohjalta yksi osa-alue, johon keskityttäisiin syvemmin. Olisi mielenkiintoista nähdä GraphQL- sekä REST-rajapinnan kohdalla konkreettista vertailua suorituskyvyn osalta. Tällöin voitaisiin konkreettisesti nähdä sekä tutkia, millä tavalla suorituskyvyn erot vaikuttavat esimerkiksi sovelluksen käyttäjäkokemukseen. Se voisi myös antaa tarkempaa tietoa siitä, missä vaiheessa GraphQL:n tarkemmat kyselyt antavat merkittävän edun verrattuna REST-rajapintaan. Tällaiselle tiedolle olisi käyttöä, kun esimerkiksi pohditaan siirtymistä REST-rajapinnasta GraphQL-rajapintaan.

Tässä työssä tehtiin GraphQL-teknologian mukana tulleiden työkalujen tarkastelemiseen vain pieni pintaraapaisu, joten jatkokehitystä voitaisiin tehdä

tarkempaan työkalujen tarkastelemiseen. Työkalujen parantuessa voitaisiin tutkia kehittäjän näkökulmasta, kuinka paljon lisäarvoa ne voisivat tuoda GraphQL-rajapintojen kehittämiseen.

Yksi jatkokehityskohde voisi olla GraphQL- sekä Falcor-teknologioiden vertaus. Falcor on Netflixin kehittämä tietoalusta, joka pyrkii optimoimaan tiedonsiirron esimerkiksi sovelluksen front- sekä backendin välillä (What is Falcor N.d). GraphQL:n suosion myötä, Falcor on ehkä hieman jäänyt sen varjoon. Nämä kummatkin teknologiat pyrkivät optimoimaan tiedon hakemista rajapinnasta, olisi mielenkiintoista nähdä mitä yhteistä/eroa näillä on keskenään.

Lähteet

Banks A & Porcello E. 2018. Learning GraphQL. O'Reilly Media. Viitattu 23.2.2019 https://learning.oreilly.com/library/view/learning-graphql/9781492030706/.

Basics, N.d. Yleiskuvaus Graphcool-työkalun käyttämisestä. Viitattu 17.5.2019. https://www.graph.cool/docs/faq/basic-iph5dieph9.

Brien, D. 2018. GraphQL and Resource Limitations. Artikkeli GraphQL-rajapinnan rajoittamisesta. Viitattu 12.05.2019. https://medium.com/in-the-weeds/graphql-and-resource-limitations-442c3bd72358.

Byron, L. 2015. GraphQL: A data query language. Artikkeli Facebookin Code-sivustolla. Viitattu 15.2.2019. https://code.fb.com/core-data/graphql-a-data-query-language/.

Caching REST API Response. N.d. Artikkeli välimuistin käyttämisestä *REST*-rajapinnoissa. Viitattu 22.2.2019. https://restfulapi.net/caching/.

Curry, J. 2017. Introduction to API Versioning Best Practices. Artikkeli rajapintojen versionnista. Viitattu 11.05.2019. https://nordicapis.com/introduction-to-api-versioning-best-practices/.

Documentation. N.d. Dokumentaatio Laravel-sovelluskehyksen käyttämisestä. Viitattu 17.5.209. https://laravel.com/docs/5.8.

Farstad, B. 2019. Why we Killed our REST API in Favour of GraphQL. Artikkeli Crystallize-yhtiön kokemuksista GraphQL:n käyttämisestä. Viitattu 12.05.2019. https://crystallize.com/blog/why-we-killed-our-rest-api-in-favour-of-graphql.

Fielding, R. 2000. Architectural Styles and
the Design of Network-based Software Architectures. Väitöskirja, filosofian tohtori. Californian yliopisto, Irvine, Information and Computer Science. Viitattu 23.2.2019. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

GraphQL. 2018. Dokumentaatio GraphQL:n määritelmästä. Viitattu 15.2.2019. https://facebook.github.io/graphql/June2018/.

GraphQL Best Practices. 2019. GraphQL:n käyttöön liittyviä suosituksia. Viitattu 21.4.2019. https://graphql.org/learn/best-practices/.

GraphQL Foundation. 2019. GraphQL projektin kuvaus. Viitattu 15.2.2019. https://foundation.graphql.org/.

GraphQL is the better REST. N.d. Opastus fullstack-sovellusten tekemiseen GraphQL:llä. Viitattu 17.2.2019. https://www.howtographql.com/basics/1-graphql-is-the-better-rest/.

GraphQL or Bust. 2018. E-kirja GraphQL:n käyttöönottamisesta sekä sen kannattavuudesta. Viitattu 15.5.2019. https://19yw4b240vb03ws8qm25h366-wpengine.netdna-ssl.com/wp-content/uploads/GraphQL-or-Bust-2018.pdf.

Hoffmann, J. 2017. SOAP And REST At Odds. Artikkeli SOAP-protokollan sekä REST-arkkitehtuurimallin kehityksestä sekä syntymisestä. Viitattu 18.5.2019. https://thehistoryoftheweb.com/soap-rest-odds/.

HTTP caching, 2019. Opastusartikkeli välimuistin käyttämiseestä HTTP-protokollan kanssa. Viitattu 22.4.2019. https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching.

Introduction to GraphQL. 2019. Ohjeistus GraphQL:n käyttämisestä. Viitattu 15.2.2019. https://graphql.org/learn/.

Kananen, J. 2015. Opinnäytetyön kirjoittajan opas: Näin kirjoitan opinnäytetyön tai pro gradun alusta loppuun. Jyväskylä: Jyväskylän ammattikorkeakoulu.

Kananen, J. 2019. Opinnäytetyön ja pro gradun pikaopas: Avain opinnäytetyön ja pro gradun kirjoittamiseen. Jyväskylä: Jyväskylän ammattikorkeakoulu.

Maldonado, L. 2019. Why GraphQL is the future of APIs. Artikkeli GraphQL:n hyödyistä. Viitattu 12.5.2019. https://dev.to/leonardomso/why-graphql-is-the-future-of-apis-3632.

Me. 2018. Meiko Oy:n verkkosivut. Viitattu 10.1.2019. https://meiko.fi/me.

Poniatowicz, T. 2019. GraphQL vs REST - Handling errors. Artikkeli virheenkäsittelyn eroista GraphQL- sekä *REST*-rajapintojen välillä Viitattu 12.05.2019. https://blog.graphqleditor.com/graphql-vs-rest-errors/.

Relational database. N.d. Verkkoartikkeli IMB:n sivuilla. Viitattu 3.3.2019. https://www.ibm.com/ibm/history/ibm100/us/en/icons/reldb/.

REST API Versioning. N.d. Opetusartikkeli REST-rajapinnan versioinnista. Viitattu 21.4.2019. https://restfulapi.net/versioning/.

REST API – N+1 Problem. N.d. Artikkeli N + 1 -ongelmasta REST-rajapinnoissa. Viitattu 7.5.2019. https://restfulapi.net/rest-api-n-1-problem/.

REST Resource Representation Compression. N.d. Artikkeli kompressoinnin käyttämisestä *REST*-rajapinnoissa. Viitattu 11.5.209. https://restfulapi.net/rest-resource-compression/.

Riady, Y. 2016. HTTP API Development Tools. Lista REST- sekä JSON-rajapintojen kehitykseen liittyviä työkaluja. Viitattu 17.5.2019. https://github.com/yosriady/api-development-tools.

Sandoval, K. 2016. Stemming the Flood – How to Rate Limit an API. Artikkeli rajapintojen rajoittamisesta. Viitattu 12.05.2019. https://nordicapis.com/stemming-the-flood-how-to-rate-limit-an-api/.

Sandoval, K. 2017. Security Points to Consider Before Implementing GraphQL. Artikkeli turvallisuusnäkökulman huomiomisesta GraphQL-sovelluksissa. Viitattu 12.05.2019. https://nordicapis.com/security-points-to-consider-before-implementing-graphql/.

Santos, W. 2017. Which API Types and Architectural Styles are Most Used? Artikkeli *ProgrammableWeb*-sivustolla tehdyn tutkimuksen mukaan. Viitattu 19.4.2019. https://www.programmableweb.com/news/which-api-types-and-architectural-styles-are-most-used/research/2017/11/26.

Schema stitching. N.d. Dokumentaatio skeemojen yhtenäistämisestä *Apollo*-kirjaston avulla. Viitattu 12.05.2019. https://www.apollographql.com/docs/graphql-tools/schema-stitching.

Self-Hosted GraphQL BaaS. N.d. Esittely Graphcool-sovelluskehityksestä. Viitattu 17.5.2019. https://www.graph.cool/.

Shapton, L., Thacker-Smith, D. & Walkinshaw, S. 2018. Solving the N+1 Problem for GraphQL through Batching. Artikkeli N + 1 -ongelman ratkaisemista GraphQL:n kanssa. Viitattu 12.05.2019. https://engineering.shopify.com/blogs/engineering/solving-the-n-1-problem-for-graphql-through-batching.

Stagno, P. 2019. GraphQL - Security Overview and Testing Tips. Artikkeli GraphQL:n turvallisuudesta sekä vinkkejä sen testaukseen. Viitattu 12.05.2019. https://blog.doyensec.com/2018/05/17/graphql-security-overview.html.

Stubailo, S. 2017. GraphQL vs REST. Artikkeli missä vertaillaan GraphQL- sekä REST-tekniikoiden eroja. Viitattu 11.5. https://blog.apollographql.com/graphql-vs-rest-5d425123e34b.

Tarvainen, J. 2016. Versioning an API in GraphQL vs. REST. Blogijulkaisu GraphQl sekä REST rajapintojen versioinnin vertailusta Symphony.fi sivustolla. Viitattu 21.4.2019. https://symfony.fi/entry/versioning-an-api-in-graphql-vs-rest.

The Only Complete API Platform. N.d. Postman-työkalun esittely. Viitattu 17.5.2019. https://www.getpostman.com/products.

The PHP Group. N.d. Documentation. PHP ohjelmointikielen dokumentaatio. Viitattu 7.4.2019. https://www.php.net/docs.php.

Transformers. N.d. Ohjeistus Transformereiden käytöstä. Viitattu 17.5.2019. https://fractal.thephpleague.com/transformers/.

Understanding schema concepts. N.d. Kuvaus GraphQL scheman konsepteista. Viitattu 17.2.2019. https://www.apollographql.com/docs/apollo-server/essentials/schema.htm.

What is Falcor. N.d. Esittely Netflixin kehittämästä Falcor-teknologiasta. Viitattu 16.5.2019. https://netflix.github.io/falcor/starter/what-is-falcor.html.

What is REST. N.d. Opetusartikkeli REST-protokollasta. Viitattu 19.4.2019. https://restfulapi.net/.

Williams, C. 2015. Is REST Best in a Microservices Architecture? Artikkeli missä pohditaan REST-järjestelmien sopivuutta mikropalveluihin. Viitattu 18.5.2019 https://capgemini.github.io/architecture/is-rest-best-microservices/.

Wieruch, R. 2018. Why GraphQL: Advantages, Disadvantages & Alternatives. Artikkeli GraphQL:n hyödyistä, haitoista sekä vaihtoehdoista. Viitattu 11.05.2019. https://www.robinwieruch.de/why-graphql-advantages-disadvantages-alternatives/.