Tampereen yliopisto

Joona Mäkipernaa

RESTFUL- JA GRAPHQL-VERKKO-OHJELMOINTIRAJAPINTOJEN VERTAILU

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Joulukuu 2020

TIIVISTELMÄ

Joona Mäkipernaa: RESTful- ja GraphQL-verkko-ohjelmointirajapintojen vertailu
Kandidaattitutkielma
Tampereen yliopisto
Tietotekniikan tutkinto-ohjelma
Joulukuu 2020

Verkko-ohjelmointirajapintojen (web API) merkitys modernien selain- ja mobiilisovelluksien kehityksessä on kasvanut. Myös muut tietokoneohjelmat hyödyntävät verkko-ohjelmointirajapintoja enemmän ja monipuolisemmin kuin ennen. Tämä on johtanut verkko-ohjelmointirajapintojen tarjoamien palvelujen ja rajapintoihin tehtyjen kyselyjen määrän kasvamiseen sekä rajapintojen monimutkaisuuden lisääntymiseen. Tämän takia tehokkaan, helposti ylläpidettävän ja helppokäyttöisen verkko-ohjelmointirajapinnan tarjoaminen ja toteuttaminen on tärkeämpää kuin ennen.

Tässä kandidaatintutkielmassa tutustutaan kahteen moderniin verkko-ohjelmointirajapintaparadigmaan ja vertaillaan niiden etuja ja heikkouksia keskenään. RESTful- ja GraphQL-rajapinnat ovat keskenään vaihtoehtoisia tapoja verkko-ohjelmointirajapinnan toteuttamiseen. Tutkielmassa käsiteltävät RESTful-rajapinnat perustuvat REST-ohjelmistoarkkitehtuuriseen tyyliin sekä HTTP-protokollaan. GraphQL on avoimen standardin datan kysely- ja muokkauskieli ohjelmointirajapintojen toteuttamiseen sekä palvelinpuolen ajonaikainen järjestelmä kyselyjen käsittelemiseen.

RESTful-rajapinta on pitkään ollut vallitseva tapa toteuttaa verkko-ohjelmointirajapinta. GraphQL:n suosio on ollut kasvussa ja se lupaa tarjota ratkaisun eräisiin RESTful-verkko-ohjelmointirajapintoja vaivaaviin ongelmiin, kuten datan yli- ja alihakemiseen. Tutkielmassa tutustutaan molempiin rajapintaparadigmoihin ja vertaillaan niiden etuja ja heikkouksia keskenään. Tutkielma on toteutettu kirjallisuuskatsauksena.

Tutkielma koostuu kahdesta osasta. Ensimmäisessä osassa esitellään vertailtavien rajapintateknologioiden ominaisuudet ja perusteet. Toisessa osassa vertaillaan GraphQL- ja RESTful-rajapintoja keskenään. Vertailussa huomioidaan asiakkaan ja palvelimen välillä liikkuvien kyselyjen ja datan määrä sekä rajapintojen ylläpidettävyys, yhdenmukaisuus, turvallisuus ja käytettävyys. Vertailuosiossa todetaan, että GraphQL-rajapinta on todennäköisesti RESTful-rajapintaa parempi ratkaisu rajapinnan toteuttamiseen, jos palvelu on monimutkainen tai nopeasti kehittyvä. Muussa tapauksessa GraphQL ei juurikaan tarjoa hyötyjä RESTful-rajapintaan verrattuna, jolloin RESTful-rajapinta on todennäköisesti yksinkertaisuutensa ja yleisyytensä takia parempi valinta verkko-ohjelmointirajapinnan toteutustavaksi.

Avainsanat: web API, verkko-ohjelmointirajapinta, REST, RESTful, GraphQL

Sisällysluettelo

1 Johdanto 1

2 RESTful- ja GraphQL-ohjelmointirajapinnat 2

2.1 REST 2

2.1.1 RESTful-verkko-ohjelmointirajapinta 4

2.1.2 REST:n kaltaiset ohjelmointirajapinnat 7

2.2 GraphQL 8

2.2.1 Tyyppijärjestelmä 8

2.2.2 GraphQL-kysely 9

3 RESTful- ja GraphQL-rajapintojen vertailu 13

3.1 Datan yli- ja alihakeminen 13

3.2 Rajapinnan ylläpidettävyys ja versiointi 15

3.3 Välimuisti 15

3.4 Rajapintojen yhdenmukaisuus 16

3.5 Kyselyn kirjoittamiseen kuluva aika, käytettävyys ja työkalutuki 16

3.6 Autentikointi, turvallisuus ja kyselyjen kompleksisuus 17

3.7 RESTful- vai GraphQL-rajapinta 18

4 Yhteenveto 20

Lähteet 21

LYHENTEET JA MERKINNÄT

API	Application programming interface, ohjelmointirajapinta
BFF	Backends For Frontends, suunnittelumalli RESTful-rajapintojen optimointiin
HTTP	Hypertext Transfer Protocol, tiedonsiirtoprotokolla
HTTPS	Hypertext Transfer Protocol Secure, salattu HTTP-protokolla
JSON	JavaScript Object Notation, tiedostomuoto
REST	Representational State Transfer, Ohjelmistoarkkitehtuurinen tyyli
URI	Uniform Resource Identifier, Resurssin tunniste
URL	Uniform Resource Locator, URI:n alityyppi, joka muodostuu resurssin sijainnista
XML	Extensible Markup Language, tiedostomuoto

1 Johdanto

Ohjelmointirajapinta (API) on rajapinta, jonka kautta kaksi tietokoneohjelmaa ovat keskenään vuorovaikutuksessa. Ohjelmien välinen vuorovaikutus on tiedon siirtämistä tai käskyjen antamista. Verkko-ohjelmointirajapinta (web API) on ohjelmointirajapinta, joka on käytettävissä tietoverkon, yleensä internetin, välityksellä. Asiakas pääsee käsiksi palvelimen tarjoamiin toimintoihin verkko-ohjelmointirajapinnan kautta. Kommunikointi asiakas- ja palvelinohjelman välillä tapahtuu kyselyjen ja vastausten välityksellä.

Verkkosivut ovat muuttuneet staattisista palvelimen puolella renderöidyitä sivuista dynaamisemmiksi kokonaisuuksiksi, jotka latautuvat ja päivittyvät osa kerrallaan saadessaan palvelimen verkko-ohjelmointirajapinnan kautta uutta dataa. Modernit, esimerkiksi React, Vue tai Angular -ohjelmistokehyksillä toteutetut, yhden sivun sovellukset (single page application) mahdollistavat monipuolisemmat käyttöliittymät, pienemmät datansiirtomäärät sekä aiheuttavat pienemmän taakan palvelimelle kuin perinteiset palvelimen puolella renderöidyt verkkosivut. Tämä on johtanut verkko-ohjelmointirajapintojen merkityksen kasvuun modernissa verkko-ohjelmistokehityksessä.

Myös mobiililaitteiden sekä muiden internettiin kytkettyjen laiteiden lisääntyminen on kasvattanut ohjelmointirajapintojen merkitystä, sillä se on lisännyt erilaisten päätelaitteiden ja -ohjelmien monimuotoisuutta. Verkko-ohjelmointirajapinta on usein yleiskäyttöinen ja pystyy palvelemaan useita erilaisia asiakkaita. Selain- ja mobiilisovellusten lisäksi verkko-ohjelmointirajapinnat voivat palvella myös muunlaisia tietokoneohjelmia. Tietokoneohjelma voi esimerkiksi tilata automaattisesti kaupan varastoon lisää kauppatavaraa tukun verkko-ohjelmointirajapinnan kautta.

RESTful-rajapinnat ovat pitkään olleet vallitseva verkko-ohjelmointirajapintojen toteutustapa. Viime aikoina RESTful-rajapinnoille vaihtoestoisten GraphQL-rajapintojen suosio on kasvanut. Ohjelmointirajapintojen merkityksen ja kokojen sekä kyselyjen monimutkaisuuden kasvaessa on myös rajapinnan ylläpidettävyyden, käytettävyyden ja tehokkuuden merkitys kasvanut. GraphQL pyrkii olemaan etenkin näillä osa-alueilla RESTful-rajapintoja parempi ohjelmointirajapintaratkaisu.

Tässä tutkielmassa tutustutaan RESTful- sekä GraphQL-ohjelmointirajapintoihin ja vertaillaan niitä keskenään. Tutkielman tutkimusmenetelmä on kirjallisuuskatsaus, jossa perehdytään aiheeseen liittyvään kirjallisuuteen. Luvussa 2 esitellään vertailtavien rajapintojen perusteet ja niiden toimintaa. Luvussa 3 vertaillaan rajapintoja keskenään ja määritellään niiden tärkeimmät edut ja heikkoudet toisiinsa nähden. Luku 4 on tutkielman yhteenveto.

2 RESTful- ja GraphQL-ohjelmointirajapinnat

Tämä luku esittelee RESTful- sekä GraphQL-ohjelmistorajapintojen tärkeimmät ominaisuudet. Luku keskittyy rajapintojen perusperiaatteisiin, tärkeimpiin ominaisuuksiin ja rajapintojen käyttämiseen. Esittely ei ole kaikenkattava, eikä se esittele esimerkiksi ohjelmistorajapintojen kaikkia ominaisuuksia tai käytänteitä. Luku ei myöskään esittele yksityiskohtaisesti rajapintojen palvelinpuolen toteutusta. Tämän luvun tarkoituksena on antaa lukijalle peruskäsitys molemmista verkko-ohjelmointirajapinnoista.

Luvun ensimmäisessä osassa esitellään REST, joka on ohjelmistoarkkitehtuurinen tyyli hajautettujen järjestelmien toteuttamiseen. REST asettaa järjestelmälle joukon rajoitteita ja vaatimuksia, joiden tarkoituksena on taata järjestelmän tehokkuus, skaalautuvuus ja hyvä ylläpidettävyys. Tämän jälkeen tutustutaan HTTP-protokollaa hyödyntäviin REST:n mukaisiin verkko-ohjelmointirajapintoihin eli niin kutsuttuihin RESTful-rajapintoihin. RESTful-rajapinnat ovat olleet pitkään yleisin verkko-ohjelmointirajapintojen toteutustapa. Luvun toisessa osassa esitellään GraphQL. GraphQL on uudehko kyselykieli datan hakemiseen ja muokkaamiseen sekä järjestelmä GraphQL-kyselyjen käsittelemiseen. GraphQL-rajapinnat ovat vaihtoehto RESTful-rajapinnoille.

2.1 REST

REST (Representational state transfer) on ohjelmistoarkkitehtuurinen tyyli hajautetun hypermediajärjestelmän toteuttamiseen, jonka Roy Fielding määritteli vuonna 2000 *Architectural Styles and the Design of Network-based Software Architectures* -väitöskirjassaan [1]. REST asettaa järjestelmälle kuusi rajoitetta:

1. asiakas–palvelin-malli (client-server)
2. tilattomuus (stateless)
3. välimuisti (cache)
4. yhdenmukainen rajapinta (uniform interface)
5. kerrostettu järjestelmä (layered system)
6. code-on-demand,

joista code-on-demand on vapaaehtoinen rajoite. Näiden rajoitteiden tarkoituksena on taata järjestelmän skaalautuvuus, tehokkuus, kommunikoinnin näkyvyys, yleispätevyys, yksinkertaisuus, muokattavuus ja ylläpidettävyys. Kommunikoinnin näkyvyydellä tarkoitetaan sitä, että kysely on toisista kyselyistä riippumaton sekä itseään kuvaava. [1]

Asiakas-palvelin-malli vaatii asiakkaan ja palvelimen tehtävien selvää erottamista. Tämä yksinkertaistaa sekä asiakasohjelmaa että palvelinohjelmaa, sillä asiakasohjelma voi keskittyä esimerkiksi vain käyttöliittymän tarjoamiseen ja palvelin datan tallentamiseen, muokkaamiseen ja tarjoamiseen. Tämä mahdollistaa myös sen, että ohjelmia voidaan kehittää toisistaan riippumatta, sekä sen, että samalla palvelimella voi olla palveltavana erilaisia asiakasohjelmia. [1]

REST:n toinen vaatimus on tilattomuus. Tämä tarkoittaa sitä, että palvelimella ei tallenneta istuntokohtaisia tietoja asiakkaasta tai kyselyistä kyselyjen välillä. Kaikkien kyselyjen on oltava siis itsenäisiä kokonaisuuksia eli niiden on sisällettävä kaikki tarpeellinen tieto, jonka palvelin saattaa tarvita kyselyn suorittamiseen. Istunnon tilan muistaminen on asiakkaan tehtävä. Tämä vähentää palvelimen tehtäviä ja kuormitusta sekä yksinkertaistaa palvelimen toteutusta. Haittapuolena on se, että tämä saattaa lisätä asiakkaan ja palvelimen välillä liikkuvaa datamäärää. [1]

Kolmas REST:n vaatimus on se, että kaikkiin kyselyihin on merkittävä, onko kysely tallennettavissa asiakkaan tai välityspalvelimien välimuistiin. Jos kysely on tallennettu asiakkaan välimuistiin, sitä ei tarvitse hakea uudelleen. Jos kyselyn vastaus on tallennettu välityspalvelimen välimuistiin, välityspalvelin voi vastata kyselyyn palvelimen puolesta. Välimuistin hyödyntäminen nopeuttaa vastausten saamista kyselyihin sekä vähentää tarvittavaa tietoliikennettä. [1] Tämän vaatimuksen haittapuolena on se, että välimuistista huolehtivat järjestelmät saattavat olla monimutkaisia.

Yhdenmukainen rajapinta lisää järjestelmän yleispätevyyttä ja yksinkertaisuutta. Yhdenmukainen rajapinta -rajoite vaatii palvelujen toteutuksen ja rajapinnan erottamista toisistaan. Toisin sanoen asiakkaan ei tarvitse tietää, miten palvelu toimii, vaan riittää, että asiakas tietää tarjotut palvelut ja osaa käyttää niitä. Tämä rajoite helpottaa järjestelmän komponenttien eli asiakkaan, palvelimen sekä välityspalvelimien erillistä kehittämistä. REST asettaa yhtenäiselle rajapinnalle neljä alivaatimusta:

- resurssien yksilöiminen (identification of resources)
- resurssien käsittely ilmentymien kautta (manipulation of resources through representations)
- itsekuvaavat viestit (self-descriptive messages)
- HATEOAS (Hypermedia as the engine of application state). [1]

Resurssien yksilöiminen tarkoittaa sitä, että jokaisella resurssilla on jokin yksilöivä tunniste eli URI. Resurssien käsittely ilmentymien kautta tarkoittaa sitä, että asiakas ei käsittele suoraa resurssia, vaan resurssin ilmentymää, joka sisältää kaiken tarpeellisen tiedon resurssin käsittelemiseen. Tämän ansiosta on palvelimen mahdollista jakaa samaa resurssia erilaisina ilmentyminä eli toisin sanoen esimerkiksi eri tiedostomuotoina. Itsekuvaavat viestit -rajoite vaatii, että kysely sisältää kaiken tarvittavan tiedon, mitä tarvitaan kyselyn käsittelemiseen, esimerkiksi kysely sisältää viestissä käytetyn tiedostomuodon, jos käytössä on useita eri tiedostomuotoja. HATEOAS vaatii, että vastaus sisältää hyperlinkkejä, joista voidaan löytää muita resurssiin liittyviä tietoja tai palveluita. Linkit toimivat siis samaan tapaan kuin linkit verkkosivulla. Palvelimen tarjoama linkkilista on voi olla dynaaminen eli riippua resurssin tilasta. [2] HATEOAS tekee palvelusta itsedokumentoivan sekä mahdollistaa palvelun uusiin ominaisuuksiin tutustumisen palvelun käyttämisen yhteydessä.

REST:n viides vaatimus on kerrostettu järjestelmä eli asiakas ei tiedä onko se suoraan yhteydessä palvelimeen vai onko palvelimen ja asiakkaan välissä kerroksia eli välityspalvelimia. Tämän ansiosta rajapintaan ei tarvitse tehdä muutoksia, vaikka asiakkaan ja palvelimen väliin lisättäisiin esimerkiksi kuormituksen tasaus -kerros tai välityspalvelin, joka tallentaa suosittuja pyyntöjä välimuistiin parantaen tehokkuutta. Välityspalvelin voi huolehtia myös esimerkiksi turvallisuudesta, estäen esimerkiksi vaarallisia kyselyjä menemästä perille. Uuden kerroksen lisääminen aiheuttaa muutoksia korkeintaan palvelimen puolella lähimpiin kerroksiin. Kerrokset voivat myös muodostaa "putken", jonka rakenne riippuu kyselystä. [1]

Code-on-demand on REST:n vapaaehtoinen vaatimus. Se antaa asiakkaalle mahdollisuuden ladata palvelimelta suoritettavia ohjelmia, esimerkiksi JavaScript-koodia. Code-on-demand-mahdollisuus voi yksinkertaistaa asiakasohjelmien toimintaa ja parantaa järjestelmän joustavuutta. Vaatimuksen vapaaehtoisuus johtuu siitä, että vaatimus luo etujen lisäksi myös haittoja, kuten sen, että se lisää asiakkaan ja palvelimen välistä riippuvuutta oletettaessa, että kaikki asiakkaat voivat suorittaa ladatun ohjelman. [1]

2.1.1 RESTful-verkko-ohjelmointirajapinta

REST:n vaatimukset täyttävää ohjelmointirajapintaa kutsutaan RESTful- tai REST-ohjelmointirajapinnaksi. REST-periaate on protokollariippumaton, mutta lähes kaikki

RESTful-verkko-ohjelmointirajapinnat hyödyntävät HTTP-protokollaa, joten tässä tutkielmassa keskitytään HTTP-RESTful-ohjelmointirajapintoihin. RESTful-ohjelmointirajapinnan käyttäessä HTTP-protokollaa se noudattaa sekä REST:n että HTTP-protokollan vaatimuksia. Tämän yhdistelmän etuna on se, että se asettaa vaatimuksia vain palvelun ohjelmointirajapinnalle. Se ei aseta rajoituksia järjestelmän toteutustavalle. Riittää, että järjestelmän komponentit pystyvät lähettämään ja ymmärtämään HTTP-viestejä ja -vastauksia. [3]

REST itsessään ei ole standardi vaan ohjelmistoarkkitehtuurinen tyyli. HTTP-REST-yhdistelmä hyödyntää HTTP- ja URL-standardeja. Nämä standardit eivät kuitenkaan aseta tarkkoja vaatimuksia kaikille rajapinnan ominaisuuksille. HTTP-REST-käyttöliittymille on kehittynyt yleisiä käytänteitä tilanteisiin, joissa standardit eivät anna selviä vaatimuksia rajapinnalle. [2] Tässä tutkielmassa käsitellään näiden yleistenkäytänteiden ja HTTP- ja URL-standardien mukaisia ohjelmointirajapintoja. RESTful-käyttöliittymien yhdenmukaistamiseksi on luotu myös standardeja, joista merkittävin on OpenAPI-spesifikaatio [3].

RESTful-ohjelmointirajapinnat tukevat yleensä JSON- tai XML-tiedostomuotoa. RESTful-ohjelmointirajapinta voi käyttää resurssien esittämiseen mitä tahansa tiedostomuotoa, kunhan sekä asiakas että palvelin ymmärtävät sitä. [2] REST:n vaatimusten mukaisesti samalla resurssilla voi olla useita eri ilmenemismuotoja.

RESTful-ohjelmointirajapinta on resurssikeskeinen eli jokainen operaatio kohdistetaan johonkin resurssiin. Resurssien yksilöiminen ja kyselyjen kohdistaminen tehdään URL:n avulla. Resurssien muokkaamiseen käytetään pääasiassa HTTP-metodeja GET, POST, PUT, PATCH ja DELETE. Käytetty metodi riippuu siitä, millainen operaatio resurssille halutaan suorittaa. Metodien odotetaan täyttävän HTTP:n vaatimukset, esimerkiksi joillekin metodeille asetetut idempotenssi- ja resurssin muokkaamattomuus -vaatimukset. [3]

RESTful-rajapinnan URL-polut jaetaan neljään arkkityyppiin: kokoelma, varasto, dokumentti ja kontrolleri. Kokoelma ja varasto sisältävät joukon resursseja. Kokoelman ja varaston erona on se, että varasto on suoraan asiakkaan muokattavissa. Dokumentti on yksittäinen resurssi. Jos sellaista HTTP-metodia, joka kuvaisi haluttua operaatiota ei ole käytettävissä, operaatio on käytettävissä kontrollerissa. [2]

URL-polku muodostetaan hierarkkisesti: ensin palvelun osoite, jonka jälkeen annetaan haluttu resurssi. Jos resurssi on kokoelma tai varasto, URL:ssä voidaan tarkentaa haluttu kokoelmassa tai varastossa oleva resurssi tunnisteen avulla [2] kuvan 1 esimerkin 1 mukaisesti. Jos resurssille on määritelty kontrolleri, voidaan kysely kohdistaa siihen kuvan 1 esimerkin 2 mukaisesti. Kun kysely saapuu palvelimelle, se ohjataan palvelimella

HTTP-metodia ja URL-polkua vastaavalle funktiolle, joka suorittaa kyselyn ja lähettää vastauksen [3].

```
Esimerkki 1. URL: http://esimerkki.fi/api/kokoelma/2245
Esimerkki 2. URL: http://esimerkki.fi/api/kokoelma/2245/kontrolleri
Esimerkki 3. URL: http://esimerkki.fi/api/kokoelma/2245?param1=55
```

Kuva 1. Esimerkkejä URL:n muodostamisesta.

Asiakas voi välittää palvelimelle kyselyn mahdolliset parametrit palvelimelle kahdella tavalla, joko kyselyn URL:ssä tai HTTP-viestin rungossa (body). URL:ssä välitetyt parametrit erotetaan kyselyn polusta kysymysmerkillä tai kirjoitetaan osana polkua. [3] Parametrien välittäminen voi tapahtua esimerkiksi kuvan 1 esimerkin 3 mukaisesti.

Kuvassa 2 tehdään esimerkkikysely avoimeen RESTful-rajapintaan [4], joka tarjoaa tietoja valtioista. Palvelusta voidaan hakea tietoja esimerkiksi valtion väkiluvusta tai pääkaupungin nimestä. Kyselyssä määritellään HTTP-metodi, joka on kuvan 2 tapauksessa GET. URL-polku johon kysely tehdään, on *https://restcountries.eu/rest/v2/name/cuba.*

```
GET https://restcountries.eu/rest/v2/name/cuba?fields=name;population;capital
Content-Type: application/json
```

Kuva 2. Esimerkkikysely avoimeen RESTful-ohjelmointirajapintaan [4].

Kuvan 2 URL:ssä välitetään polun lisäksi parametrejä. Esimerkin ohjelmointirajapinta tarjoaa mahdollisuuden määrätä kentät, jotka palvelin täyttää vastauksessa. Jos kyselyssä ei olisi määrätty haluttuja kenttiä, palvelin olisi palauttanut vastauksessa oletuskentät. Haluttujen kenttien määritteleminen on melko yleinen ominaisuus isoissa RESTful-rajapinnoissa, etenkin jos pyydetyllä resurssilla on paljon kenttiä tai jonkun kentän koko on suuri [3]. Tämä parantaa asiakkaan ja palvelimen tehokkuutta ja pienentää siirrettävää datamäärää. URL:sta nähdään myös, että kysely käyttää HTTPS-protokollaa pelkän HTTP-protokollan sijaan. Kuvassa 3 nähdään kuvan 2 kyselyyn saatu vastaus.

```
HTTP/1.1 200
Date: Thu, 15 Oct 2020 12:28:36 GMT
Content-Type: application/json;charset=utf-8
Transfer-Encoding: chunked
Connection: close
Cache-Control: public, max-age=86400
Content-Encoding: gzip

[
  {
    "name": "Cuba",
    "capital": "Havana",
    "population": 11239004
  }
]
```

Kuva 3. Kuvan 2 kyselyyn saatu vastaus. Kuvasta on poistettu joitain otsikkokenttiä.

Kuvan 3 vastaus koostuu kahdesta osasta; otsikosta (header), joka sisältää HTTP-protokollaan liittyviä kenttiä ja rungosta (body), joka sisältää pyydetyt tiedot. Kuvan 3 ensimmäisellä rivillä nähdään HTTP-tilakoodi, joka on tässä tapauksessa *200*, joka tarkoittaa kyselyn onnistumista. HTTP-RESTful-rajapinnat käyttävät HTTP-koodeja kyselyn onnistumisen tai epäonnistumisen viestimiseen [3].

2.1.2 REST:n kaltaiset ohjelmointirajapinnat

Ohjelmointirajapintoja, jotka eivät täytä kaikkia REST:n asettamia rajoituksia kutsutaan REST:n kaltaisiksi (REST-like) ohjelmointirajapinnoiksi. REST:n kaltaiset rajapinnat yleensä rikkovat REST:n HATEOAS-rajoitetta. [5] Vertailuosiossa läpikäytävät seikat pätevät sekä RESTful- että REST:n kaltaisiin rajapintoihin, ellei toisin mainita.

Moni RESTful-rajapinta onkin todellisuudessa vain REST:n kaltainen rajapinta. HATEOAS-rajoitteen rikkomisen syynä on se, että se tekee rajapinnan toteuttamisesta työläämpää ja monimutkaisempaa. Haittapuolena HATEOAS-rajoitteen rikkomisessa on se, että se tekee rajapinnasta paljon itseään huonommin dokumentoivan. Asiakasohjelmat voisivat linkkien avulla myös käyttää rajapinnan palveluita dynaamisemmin ja monipuolisemmin. [5]

2.2 GraphQL

GraphQL on Facebookin kehittämä datan kysely- ja muokkauskieli ohjelmointirajapintojen toteuttamiseen sekä palvelinpuolen ajonaikainen järjestelmä kyselyjen käsittelemiseen [6]. GraphQL:n kehittämisen syynä oli tarve saada RESTful-ohjelmointirajapinnoille tehokkaampi ja helppokäyttöisempi vaihtoehto. Facebook aloitti GraphQL:n kehittämisen vuonna 2012. Vuonna 2015 Facebook julkaisin GraphQL-spesifikaation avoimena standardina sekä avoimen lähdekoodi GraphQL-referenssitoteutuksen JavaScript-kielellä. [7] Avoimen lähdekoodin GraphQL-palvelin toteutuksia on JavaScriptin lisäksi saatavilla myös useilla muilla suosituilla palvelinpuolen kielillä kuten C#, Java, Python, Rust ja PHP [8].

GraphQL:n perusajatuksena on siirtää palvelimen lähettämän datan rakenteen ja sisällön määritteleminen palvelimelta asiakkaalle. Asiakas määrittelee kyselyssään haluamansa kentät ja saa vastauksessa vain määrittelemänsä kentät. Tämä tarjoaa ratkaisun monia RESTful-rajapintoja vaivaavaan datan yli- tai alihakemiseen. [9]

GraphQL-kyselyt ovat hierarkkisia. GraphQL:n kentät muodostuvat nimi–arvo-pareista. Kyselyn kentät voivat sisältää kenttiä, joilla on omat kenttänsä, jotka voivat puolestaan sisältää toisia kenttiä, jos kyselyn tyyppien välille on määritelty jokin suhde. Tämän ansiosta yksi GraphQL-kysely voi hakea tietoja, joiden hakeminen olisi vaatinut RESTful-rajapinnalta monta eri kyselyä eri resursseihin. [9]

GraphQL-skeema määrittää palvelun rajapinnan muodostaen graafin. Skeema määrittelee resurssit, niiden tyypit ja resurssien väliset suhteet eli graafiteorian käsittein skeema vastaa graafia, resurssit solmuja ja resurssien väliset suhteet kaaria. Rajapinnasta riippuen kaaret voivat olla yksi- tai kaksisuuntaisia. Graafeilla voidaan mallintaa monia ilmiöitä, joten GraphQL:n avulla helppo luoda intuitiivinen rajapinta monenlaisille eri palveluille. [10]

2.2.1 Tyyppijärjestelmä

Skeema määrittää GraphQL-palvelimen rajapinnan. Skeemassa määritellään kaikki ohjelmointirajapinnan tarjoamat operaatiot ja ohjelmointirajapinnan kenttien tyypit. GraphQL-skeema on introspektiivinen eli se toimii samalla myös osana rajapinnan dokumentaationa ja mallina kyselyjen validointiin. Rajapinnan käyttäjä saa skeeman itselleen GraphQL:n introspektiokyselyjen avulla. Skeema voi sisältää kenttiä kuvailevia viestejä. [9]

GraphQL on tyypitetty kieli ja se sisältää tyyppitarkastuksen. GraphQL:ssä jokaisella kentällä on tyyppi. GraphQL sisältää kuusi eri tyyppiluokkaa: skalaari (scalar), enum, olio (object), union, interface ja input. GraphQL-kieli sisältää viisi valmista skalaarityyppiä: kokonaisluku (int), desimaaliluku (float), merkkijono (string), totuusarvo (boolean) ja ID. Samoin kaikkien muiden tyyppien tapaan, palvelin voi määrittää omia skalaarityyppejä. Enum on tyyppi, joka koostuu annetusta määrästä skalaarityyppejä. Skalaarit ja enum-kentät muodostavat graafin reunat, sillä näiden kenttien arvona ei voi olla toisia kenttiä. [11]

Olio on GraphQL-tyyppi, jonka kentillä on arvona jokin GraphQL-tyyppi eli olioiden kenttien arvona voi olla myös toinen olio, jolloin oliot muodostavat hierarkkisen rakenteen. GraphQL:ssä on kaksi abstraktia tietotyyppiä union ja interface, joilla voidaan määrittää olioiden ominaisuuksia. Input-tyypin kentät koostuvat skalaari-, enum- tai input-tyypeistä. [11]

GraphQL sisältää kaksi tyyppimääritettä, jotka ovat list ja non-null. List-määrite määrää, että kentän arvona on yhden arvon sijaan lista annetun tyyppisiä arvoja. Non-null-määrite kertoo, että kentän arvona ei voi olla GraphQL:n null-arvo. Jos non-null-määritettä ei ole annettu, minkä tahansa kentän arvona voi oletusarvoisesti olla null. Lista koostuu nollasta tai useammasta saman tyyppisestä arvosta. Listan merkitsemiseen skeemassa käytetään hakasulkeita kentän tyypin ympärillä ja non-null-määreen merkitsemiseen käytetään huutomerkkiä kentän tyypin perässä. [11]

GraphQL sisältää kolme operaatiotyyppiä: query, mutation ja subscription. Kaikki kyselyt sisältävät vähintään yhden operaation. Query on datan hakuoperaatio, joka ei muuta resurssien tilaa. Mutation on datan muokkausoperaatio, jota seuraa datan hakuoperaatio. Mutation yleensä palauttaa vastauksessa muokkauksen tuloksen. Subscription on pitkäikäinen kysely, joka pyytää palvelinta lähettämään kyselyyn vastauksen aina tietyn tapahtuman, yleensä datan muuttumisen, yhteydessä. Rajapinnan resursseihin pääsee käsiksi vain operaatioiden kautta eli toisin sanoen operaatiot toimivat sisääntulopisteenä rajapinnan GraphQL-graafiin. [11]

2.2.2 GraphQL-kysely

GraphQL-kyselyt ja -vastaukset siirretään yleensä HTTP-protokollan avulla. GraphQL ei hyödynnä HTTP-protokollaa yhtä laajasti kuin RESTful-rajapinnat. Toisin kuin RESTful-rajapinnoilla, GraphQL-rajapinnoilla on vain yksi URL-osoite. Kyselyn tekemiseen

käytetään yleisten käytänteiden mukaisesti yleensä POST-metodia ja GraphQL-kyselymerkkijono annetaan HTTP-kyselyn rungossa. Kysely voidaan tehdä myös GET-metodilla, jolloin GraphQL-kyselymerkkijono on annettu query-nimisenä URL-parametrina. GraphQL ei hyödynnä HTTP-koodeja, vaan GraphQL-kysely palauttaa aina koodin *200*, jos yhteys palvelimeen saatiin muodostettua.

Riippumatta siitä, miten kysely lähetettiin palvelimelle palvelin palauttaa vastauksen samalla tavalla, yleensä JSON-muodossa HTTP-viestin rungossa. Jos kysely aiheuttaa palvelimella virheen, esimerkiksi kysely on väärin muodostettu, palvelin palauttaa error-kentän, joka sisältää virheviestin. Kyselyn vastaus on vastauksen data-kentässä. GraphQL voi käyttää mitä tahansa tiedostomuotoa, joka pystyy kuvaamaan datan samoin JSON, mutta JSON on yleisin tiedostomuoto. [12]

GraphQL client -sovellukset ovat asiakasohjelmille luotuja moduuleja, jotka helpottavat GraphQL-rajapinnan käyttämistä. Ne hoitavat rajapinnan käyttämisen matalan tason yksityiskohdat, kuten yhteyden muodostamisen palvelimeen, kyselyjen lähettämisen sekä kyselyjen vastaanottamisen. Esimerkiksi käytettäessä GraphQL client -moduulia sille tarvitsee antaa sille vain GraphQL-kyselymerkkijono, jolloin GraphQL client hoitaa kyselyn tekemisen muut yksityiskohdat. Suosittuja GraphQL client -kirjastoja selainsovelluksille ovat muun muassa Apollo Client ja Relay. [9]

Kyselyn kentät voivat ottaa vastaan argumentteja monien ohjelmointikielten funktioiden tapaan. Skeemassa ja kyselymerkkijonossa mahdolliset argumentit merkitään kentän jälkeen olevissa sulkeissa. GraphQL:ssä argumentit ovat nimettyjä ja niiden järjestyksellä ei ole väliä. Argumentit muuttavat kentän evaluoivan resolverin toimintaa. Argumentit voivat olla joko vapaaehtoisia tai pakollisia. Pakolliset kentän argumentit merkitään huutomerkillä non-null-määreen tapaan. [11]

Kun kysely saapuu palvelimelle, tyyppijärjestelmä validoi kyselyn skeeman ja GraphQL:n sääntöjen mukaisesti. Jos kysely on validi, palvelin aloittaa kyselyn suorittamisen. Jos kysely ei ole validi palautetaan asiakkaalle vastaus, joka sisältää virheviestin. Palvelimella jokaiselle kentälle on määritelty resolver-funktio. Resolver-funktion tehtävänä on palauttaa oman kenttänsä arvo. Jos kentän arvona on toinen kenttä, resolver kutsuu seuraavan kentän resolveria ja palauttaa sen palauttaman arvon. Jos evaluoitavan kentän tyyppinä on skalaari, kutsuketju päättyy, sillä skalaari- tai enum-kentän arvona ei voi olla toista kenttää. Kun kaikki resolverit on suoritettu, palvelin muodostaa resolverien arvoista vastauksen ja palauttaa sen asiakkaalle. GraphQL tukee asynkronisia resolvereita, koska resolverit ovat useasti asynkronisia, sillä palvelimet hakevat kenttien datan usein erillisestä tietokannasta. Resolverien rinnakkainen suorittaminen on myös mahdollista, mikä parantaa kyselyn suoritusnopeutta. [11]

Kuvassa 4 on ote avoimen GraphQL-rajapinnan [13] skeemasta. Skeema on saatu tekemällä GraphQL-introspektiokysely palvelimen rajapintaan. Kuvan 4 osittaisessa skeemassa määritellään oliotyypit *Country* ja *Language*. Skeemasta nähdään kaikkien kenttien tyypit. *State* ja *Continent* ovat muita skeemassa määriteltyjä tyyppejä. Non-null-kentät on merkitty huutomerkillä eli näillä kentillä on aina jokin muu arvo kuin null-arvo. Esimerkissä *Country*n *languages*-kentän arvona on lista. *Languages*-kenttä ei voi saada null-arvoa. *Languages*-kentän lista sisältää *Language*-tyyppisiä olioita. Myöskään lista ei sisällä null-arvoja, koska listan sisältämille arvoille on annettu non-null-määre.

```
type Country {
  code: ID!
  name: String!
  native: String!
  phone: String!
  continent: Continent!
  capital: String
  currency: String
  languages: [Language!]!
  emoji: String!
  emojiU: String!
  states: [State!]!
}

type Language {
  code: ID!
  name: String
  native: String
  rtl: Boolean!
}

type Query {
  continents(filter: ContinentFilterInput): [Continent!]!
  continent(code: ID!): Continent
  countries(filter: CountryFilterInput): [Country!]!
  country(code: ID!): Country
  languages(filter: LanguageFilterInput): [Language!]!
  language(code: ID!): Language
}
```

Kuva 4. Ote avoimen GraphQL-rajapinnan [13] skeemasta.

Query-tyyppi määrittelee kaikki rajapinnan query-tyyppiset operaatiot. Mahdolliset mutation- ja subscription-tyyppiset operaatiot määriteltäisiin skeemassa samalla tavalla. Kuvasta 4 nähdään, että kaikki kyseisen rajapinnan query-operaatiot voivat ottaa vastaan parametreja. Kenttien pakolliset parametrit on merkitty skeemassa huutomerkillä.

Kuvassa 5 vasemmalla on GraphQL-kyselymerkkijono kuvan 4 palvelimelle. Kysely hakee palvelimelta tiedot valtion, jonka *code*-kentän arvona on merkkijono *FI*, pääkaupungin nimestä ja kyseisen valtion virallisista kielistä. Valtion virallisista kielistä kysely hakee kielien englanninkielisen ja kotoperäisen nimen.

```
query {
  country(code:"FI") {
    capital
    languages {
      name
      native
    }
  }
}
```

```
{
  "data": {
    "country": {
      "capital": "Helsinki",
      "languages": [
        {
          "name": "Finnish",
          "native": "Suomi"
        },
        {
          "name": "Swedish",
          "native": "Svenska"
        }
      ]
    }
  }
}
```

Kuva 5. Vasemmalla on esimerkkikyselymerkkijono, joka lähetettiin kuvan 4 GraphQL-rajapintaan [13] ja oikealla on kyselyyn saatu JSON-muotoinen vastaus.

Kuvassa 5 oikealla nähdään vasemmalla olevaan kyselyyn saatu vastaus. Kuvasta nähdään, että GraphQL-kyselyn vastauksen muoto mukailee kyselyn muotoa. Tämä helpottaa kyselyjen kirjoittamista ja tekee vastauksen tulkitsemisen ihmisille helpommaksi.

3 RESTful- ja GraphQL-rajapintojen vertailu

3.1 Datan yli- ja alihakeminen

RESTful-rajapintojen yksi suurimmista ongelmista on se, että kyselyt eivät voi usein antaa asiakkaalle juuri sitä dataa, jonka asiakas tarvitsee. Tämä oli tärkein syy GraphQL:n kehittämiseen. Tätä ongelmaa kutsutaan datan yli- tai alihakemiseksi. [7]

Datan ylihakeminen tarkoittaa, että kyselyn vastaus sisältää ylimääräistä dataa eli kenttiä, joille asiakkaalla ei ole tarvetta. RESTful-rajapinnoilla on kaksi yleistä tapaa tämän ongelman ratkaisemiseen. Yksi tapa ratkaista tämä ongelma on se, että asiakas tarkentaa kyselyssään haluamansa kentät. Tämä voidaan tehdä esimerkiksi kuten kuvan 2 kyselyssä eli halutut kentät annetaan URL:ssä parametreina. Parametrit voidaan välittää usein myös kyselyn rungossa. Iso osa RESTful-rajapinnoista ei kuitenkaan tue tätä mahdollisuutta eikä sitä ei vaadita REST:n rajoitteissa tai HTTP- tai URL-standardeissa. Ongelmana tässä tavassa on myös se, että tapaa, jolla halutut kentät välitetään palvelimelle ei ole standardoitu, joten se voi vaihdella rajapinnan mukaan, mikä tekee tämän ominaisuuden käyttämisestä vaikeampaa. [14] Rajapinnan käyttäjä ei välttämättä myöskään ole motivoitunut käyttämään tätä ominaisuutta, sillä haluttujen kenttien tarkentaminen tekee asiakasohjelman kirjoittamisesta ja ylläpitämisestä työläämpää.

Datan alihakeminen tarkoittaa sitä, että yksi kysely ei riitä joidenkin tietojen hakemiseen tai operaation suorittamiseen, vaan tarvittavien tietojen hakemiseksi on tehtävä useampi kysely. Se, että yksi kysely ei riitä voi johtua siitä, että seuraavan kyselyn sisältö riippuu edellisen kyselyn vastauksesta tai siitä, että tarvittavat tiedot eivät ole saatavilla samoista URL-poluista. Usean kyselyn tekeminen voi hidastaa asiakkaan toimintaa huomattavasti yhteyden viiveen takia varsinkin, jos kyselyjä ei voi suorittaa rinnakkain. Tiedon kokoaminen useista vastauksista tekee myös asiakasohjelmasta monimutkaisemman ja vaikeammin ylläpidettävän. Useiden kyselyjen tekeminen yhden kyselyn sijaan vaatii myös enemmän tietoliikennettä ja kuluttaa enemmän virtaa. [14] Virrankulutuksen ja tietoliikenteen vähentäminen on tärkeää etenkin mobiili- ja IoT-laitteiden kannalta.

Toinen ratkaisu, joka vähentää RESTful-rajapintojen datan ylihakemista, on luoda sellaisille kenttäkombinaatioilla, joita tarvitaan usein yhdessä, yksi URL-polku, josta saadaan vain nämä kentät. Tätä menetelmää kutsutaan *backends for frontends*- tai BFF-suunnittelumalliksi. Tämä ratkaisu voi vähentää myös datan alihakemista. Tämä ratkaisutapa vaatii, että palvelimen toteuttaja tietää nämä usein tarvitut kenttäkombinaatiot, mikä vaatii

palvelimen ja asiakasohjelmien kehittäjien välistä yhteistyötä. Tämä lähestymistapa tekee rajapinnan ylläpitämisestä työläämpää, sillä se lisää ylläpidettävien URL-polkujen ja täten niiden käsittelijöiden määrää. Uusien polkujen lisääminen RESTful-rajapintaan vaatii aikaa ja saattaa täten hidastaa myös näitä polkuja tarvitsevien asiakasohjelmien kehittämistä. [14]

Datan ylihakeminen ei vaivaa GraphQL-rajapintoja, sillä asiakas saa vastauksessaan vain ne kentät, jotka asiakas on kyselyssään pyytänyt. GraphQL vähentää myös datan alihakemista RESTful-rajapintoihin verrattuna, mutta ei kuitenkaan aina poista sitä täysin [16].

Tutkimuksessa [16], jossa vertailtiin GraphQL- ja RESTful-rajapintoihin tehtyihin kyselyihin saatuja vastauksia, todettiin, että GraphQL-kyselyjen vastaukset olivat pienempiä. Tutkimuksessa muunnettiin 22 seitsemästä eri ohjelmistoprojektista löytynyttä RESTful-kyselykokonaisuutta vastaaviksi GraphQL-kyselyiksi. Kyselykokonaisuudella tarkoitetaan sarjaa kyselyitä, joka tarvitaan tietyn toiminnon toteuttamiseksi. GraphQL-kyselykokonaisuuksien vastauksissa oli keskimäärin 94 % vähemmän kenttiä ja ne olivat keskimäärin bittimäärältään 99 % pienempiä. GraphQL-kysely tuotti vastauksessa aina korkeintaan yhtä monta kenttää kuin vastaava RESTful-rajapintaan tehty kysely. Tutkimuksen RESTful-rajapintoihin tehdyt kyselyt eivät käyttäneet vain haluttujen kenttien hakemista.

Samassa tutkimuksessa todettiin myös, että tarvittujen kyselyjen määrä oli pienempi GraphQL-rajapintoja käytettäessä kuin RESTful-rajapintoja käytettäessä. Tutkimuksessa viisi kyselykokonaisuutta koostui kahdesta tai useammasta RESTful-kyselystä. Näiden kyselykokonaisuuksien suorittaminen vaati yhteensä 12 kyselyä RESTful-rajapintaan ja 6 kyselyä vastaavaan GraphQL-rajapintaan. Tutkimuksessa oli yksi kyselykokonaisuus, jonka suorittaminen vaati yhden kyselyn RESTful-rajapintaan, mutta kaksi kyselyä GraphQL-rajapintaan. Tästä tuloksesta voidaan todeta, että GraphQL vähentää alihakemista RESTful-rajapintoihin verrattuna, mutta ei poista sitä kokonaan. [16]

Eräässä toisessa tutkimuksessa [17], jossa palvelun RESTful-rajapinta ja rajapintaan tehdyt kyselyt muutettiin GraphQL-rajapinnaksi ja GraphQL-kyselyiksi, todettiin, että rajapintojen suoritusaikojen välillä ei ollut merkityksellisiä eroja kyselyn hakiessa yksittäisen resurssin tiedot palvelimelta. Tutkimuksessa korvattiin myös kolmesta kyselystä koostuva kyselykokonaisuus yhdellä GraphQL-kyselyllä. Tällöin GraphQL-kyselyn suoritus aika 46 % lyhyempi verrattuna RESTful-kyselyjen vaatimaan kokonaisaikaan.

3.2 Rajapinnan ylläpidettävyys ja versiointi

Rajapinnan mukaan GraphQL-rajapinnan ylläpitäminen voi olla helpompaa kuin vastaavan RESTful-rajapinnan. Jos rajapinta käyttää BFF-suunnittelumallin mukaisia URL-polkuja rajapinnan optimointiin, voi se tarkoittaa sitä, että yhden muutoksen tekeminen vaatii palvelimen koodin uudelleen kirjoittamista useassa eri kohdassa. Liian suuri määrä URL-polkuja voi tehdä rajapinnasta turhan monimutkaisen, mikä heikentää rajapinnan käytettävyyttä ja ylläpidettävyyttä. GraphQL-rajapintaa muokatessa muutoksen tekeminen vaatii vain muutosta koskevien resolverien kirjoittamista tai päivittämistä. [9]

GraphQL-rajapinnan versioinnintarve on pienempi kuin vastaavan RESTful-rajapinnan versioinnintarve. Uusien kenttien lisäämistä RESTful-rajapinnan resursseihin voidaan pitää rajapinnan rikkovana muutoksena, sillä se voi muuttaa olemassa oleviin kyselyihin saatavia vastauksi. Kenttien lisääminen RESTful-rajapinnan resurssiin vaatisi täten uuden version julkaisemista ja versionumeron muuttamista. GraphQL:ää käytettäessä uusien tyyppien tai kenttien lisääminen ei aiheuta rajapintaa rikkovaa muutosta, sillä vanhat kyselyt käsittelevät vain kenttiä ja tyyppejä, jotka olivat olemassa ennen muutoksen tekemistä. Tällöin vastaukset vanhoihin kyselyihin eivät sisällä uusia kenttiä tai tyyppejä. [15] GraphQL ei kuitenkaan poista kokonaan versioinnintarvetta. Voi olla, että rajapinnan skeemaan on joskus tehtävä sellainen muutos, joka rikkoo rajapinnan. Tällaisessa tilanteessa myös GraphQL-rajapinta on versioitava.

3.3 Välimuisti

GraphQL:n suurin heikkous RESTful-rajapintoihin verrattuna lienee se, että GraphQL ei itsessään sisällä välimuistinhallintaa. Välimuistilla tässä yhteydessä tarkoitetaan vastausten tallentamista muistiin tietyn ajanjakson ajaksi, jolloin se on sieltä nopeasti saatavilla eikä palvelimen tarvitse käsitellä kyselyä, vaan vastaus kyselyyn on valmiina jonkun osapuolen muistissa.

GraphQL-spesifikaatio ei aseta mitään sääntöjä välimuistin käyttämiselle. GraphQL-rajapinnoille on kehitetty spesifikaation ulkopuolisia ratkaisuja ja työkaluja välimuistinhallintaan. Yleensä GraphQL client -ohjelma tai jotkin muu asiakaspuolen järjestelmä hoitaa asiakasohjelmien välimuistinhallinnan. Tämän ratkaisun huono puoli on se, että asiakasohjelmista tulee hieman monimutkaisempia kuin vastaavista RESTful-rajapintaa käyttävistä asiakasohjelmista, joiden välimuistinhallinta on palvelimen vastuulla. GraphQL:n välimuistinhallintalogiikka vaihtelee myös usein sovelluksen mukaan, jolloin siihen on harvoin tarjolla täysin valmiita ratkaisuja. GraphQL:n asiakaspuolen välimuistinhallinta

hyödyntää usein ID-kenttiä. Sekä GraphQL- ja RESTful-rajapintojen tapauksissa palvelimen käyttämien tietokantojen kannattaa hyödyntää välimuistia. Tällöin kyselyihin vastaaminen on nopeampaa. [9]

RESTful-rajapintojen käytettäessä välimuistin hyödyntäminen on helppoa, sillä välimuistin hyödyntäminen yksi REST:n vaatimuksista ja HTTP-protokolla tukee välimuistinhallintaa. Riittää, että palvelin asettaa sopivat arvot vastauksen välimuistinhallintaan liittyviin HTTP-otsikkokenttiin. Tällöin asiakkaat ja välityspalvelimet voivat hyödyntää HTTP-viestin otsikkokentän tietoja välimuistinhallinnassa. [2]

3.4 Rajapintojen yhdenmukaisuus

RESTful-rajapinnoilla ei ole virallista standardia, mikä johtaa epäjohdonmukaisuuksiin rajapintojen välillä. Tämä johtaa siihen, että asiakkaan on uuteen rajapintaan tutustuessaan tutustuttava myös rajapinnan käytänteisiin. Jos rajapinnan käyttäjä olettaa rajapinnan toimivan toisin kuin se todellisuudessa toimii, voi syntyä ohjelmointivirheitä. RESTful-rajapinnoille on luotu standardeja, jotka pyrkivät yhtenäistämään niitä. Näistä standardeista merkittävin on OpenAPI-spesifikaatio. [14]

GraphQL-spesifikaatio määrää tarkasti kielen syntaksin ja rajapinnan toiminnan, jolloin kaikki GraphQL-rajapinnat toimivat samoin. Tämä helpottaa uusiin rajapintoihin tutustumista. Yhdenmukaisuus rajapintojen välillä helpottaa myös erilaisten työkalujen kehittämistä. GraphQL:n tyypitysjärjestelmä ja skeema vähentävät myös yksilöistä johtuvaa yhdessä rajapinnassa olevaa epäyhdenmukaisuutta [14].

3.5 Kyselyn kirjoittamiseen kuluva aika, käytettävyys ja työkalutuki

Tutkimuksessa [18], joka vertaili GraphQL- ja RESTful-rajapintoihin tehtävien kyselyjen kirjoittamiseen kulunutta aikaa todettiin, että GraphQL-kyselyjen toteuttamiseen kului keskimäärin vähemmän aikaa. Tutkimukseen osallistui 10 perustutkintoa ja 12 jatko-opiskelijaa. Kaikilla tutkimuksen osallistujilla oli vähintään yhden vuoden pituinen ohjelmointitausta. Kukin osallistuja toteutti kahdeksan toisiaan vastaavaa kyselyparia GitHubin RESTful- ja GraphQL-rajapintoihin. Osallistujat saivat käyttää molempien rajapintojen dokumentaatiota, sekä RESTful-rajapinnan kyselyjen testaamiseen selainta ja GraphQL-kyselyjen kirjoittamiseen yleistä Graph*i*QL IDE -työkalua.

Tutkimuksessa RESTful-rajapintojen yhden kyselyn kirjoittamiseen kului keskimäärin yhdeksän minuuttia ja GraphQL-kyselyjen kirjoittamiseen kului keskimäärin kuusi minuuttia. GraphQL-kyselyjen toteuttaminen vaati keskimäärin vähemmän aikaa sekä kokeneempien, eli jatko-opiskelijoiden, joukossa että vähemmän kokeneempien osallistujien, eli perustutkinto opiskelijoiden, joukossa. GraphQL-kyselyjen toteuttaminen vaati keskimäärin vähemmän aikaa jopa sellaisilta osallistujilta, joilla oli ennen tutkimukseen osallistumista kokemusta RESTful-rajapinnoista, mutta ei GraphQL- rajapinnoista. GraphQL:n hyöty kasvoi suhteessa RESTful-rajapintaan kyselyjen ollessa monimutkaisia eli URL:ien ollessa pitkiä ja kyselyn sisältäessä useita parametrejä. [18] Tutkimustulosta voidaan pitää jokseenkin epätasapuolisena, sillä RESTful-rajapintojen testaamiseen on olemassa myös selainta parempia työkaluja, jotka voivat esimerkiksi helpottaa kyselyjen lähettämistä ja vastausten lukemista.

Kyselyjen suorittamisen jälkeen osallistujia pyydettiin osallistumaan vapaaehtoiseen kyselyyn, jossa kysyttiin mielipiteitä RESTful- ja GraphQL-kyselyjen toteuttamisesta. Osallistujien mukaan GraphQL:n etuna oli Graph*i*QL IDE -työkalu, joka auttaa GraphQL-kyselyjen kirjoittamisessa. [18] Graph*i*QL IDE tarjoaa muun muassa automaattisen täydennyksen ja näyttää käytetyn GraphQL-rajapinnan skeeman. Toinen osallistujien mainitsema GraphQL:n etu oli helpommin luettava syntaksi ja parametrien määrittelemisen helppous. RESTful-rajapinnan eduksi osallistujat mainitsivat rajapinnan paremman dokumentaation. [18] RESTful-rajapinnan dokumentaation paremmuus johtunee siitä, että kyseinen rajapinta on vanhempi kuin GraphQL-rajapinta, joten dokumentaatio on todennäköisesti kypsempi.

GraphQL:n tyyppijärjestelmä ja rajapinnan skeema mahdollistavat monipuolisten kyselyjen kirjoittamista helpottavien työkalujen luomisen. Nämä työkalut voivat tarkastaa, että kysely on validi eli GraphQL:n sääntöjen ja käytetyn rajapinnan skeeman mukainen. Ne voivat tarjota tämän lisäksi muun muassa automaattisen täydennyksen, kirjoitusvirheiden korostamisen, huomautukset pakollisten parametrien puuttumisesta ja niin edelleen. Yleisesti samankaltaisten työkalujen kehittäminen ei ole mahdollista RESTful-rajapinnoille, elleivät ne seuraa jotain standardia, joka mahdollistaa sen, kuten OpenAPI-spesifikaatiota. [18]

3.6 Autentikointi, turvallisuus ja kyselyjen kompleksisuus

GraphQL- tai RESTful-rajapintojen säännöt tai rajoitukset eivät määrää, miten käyttäjät pitäisi autentikoida. RESTful-rajapinnoille on olemassa standardoituja tapoja käyttäjien autentikointiin, kuten OAuth-protokolla. RESTful-rajapintojen autentikointi on yleensä

HTTP-pohjaista eli ne hyödyntävät pelkästään HTTP-protokollaa. [5] Kun GraphQL-kyselyjä siirretään HTTP-protokollan avulla, kuten yleensä, ne voivat käyttää samoja HTTP-pohjaisia tapoja käyttäjien autentikoitiin kuin RESTful-rajapinnat. Tällöin autentikointi voidaan suorittaa palvelimella GraphQL-tason yläpuolella HTTP-viestejä käsittelevällä tasolla. [14]

Yksi GraphQL-rajapintojen tietoturvaongelma, jota RESTful-rajapinnoilla ei ole, on se, että GraphQL ei tue piilotettuja kenttiä [16]. Kun käyttäjä saa GraphQL-rajapinnan skeeman itselleen, hän näkee kaikki skeeman tyypit ja kentät. Tämä voi paljastaa rajapinnasta enemmän, kuin rajapinnan tarjoaja haluaisi kaikille asiakkaille paljastaa. Jos halutaan pitää joitain kenttiä joiltain asiakkailta salassa, niin on luotava erilaisille asiakkaille eri rajapinnat, joilla on omat skeemansa.

Toinen GraphQL-rajapintojen turvallisuusongelma, jota RESTful-rajapinnoilla ei ole, on se, että GraphQL-kyselyjen monimutkaisuus ei ole rajoitettu. GraphQL-kysely voi sisältää useita tasoja, raskaita operaatioita tai olla todella pitkä, jolloin kyselyn suorittaminen vaatii paljon suoritusaikaa. Tallaiset kyselyt voivat tahattomasti hidastaa palvelimen toimintaa tai niitä voidaan käyttää palvelunestohyökkäyksissä. Ratkaisuna tähän ongelmaan on se, että palvelin jättää tallaiset kyselyt suorittamatta. Palvelin voi suodattaa ei-toivottuja kyselyjä esimerkiksi kyselyn koon tai arvioidun kompleksisuuden perusteella, tai palvelin voi asettaa kyselyn suorittamiselle yläaikarajan. [14] GraphQL-kyselyn kompleksisuuden arvioiminen on mahdollista ja tehokasta algoritmisesti. [19]

3.7 RESTful- vai GraphQL-rajapinta

GraphQL-rajapinta on RESTful-rajapintaa parempi valinta verkko-ohjelmointirajapinnaksi, jos rajapintaan tehtävät kyselyt olisivat RESTful-rajapintaa käytettäessä monimutkaisia eli URL-osoitteet olisivat pitkiä ja parametrien määrä olisi suuri. Tällöin GraphQL:ää käytettäessä kyselyn kirjoittaminen on helpompaa ja nopeampaa sekä kyselyn kirjoittamiseen on tarjolla enemmän työkaluja. GraphQL-rajapinta on myös parempi valinta kuin RESTful-rajapinta, jos RESTful-rajapinta sisältäisi paljon BFF-suunnittelumallin mukaisia polkuja tai RESTful-rajapinta kärsisi datan alihakemisesta. GraphQL-rajapinta on todennäköisesti yksinkertaisempi ja helpompi ylläpitää kuin paljon BFF-mallin mukaisia polkuja sisältävä RESTful-rajapinta. GraphQL vähentää datan alihakemista, mikä tekee asiakasohjelmista yksinkertaisempia ja nopeampia sekä vähentää tarvittua tietoliikennettä ja virrankulutusta. Jos rajapinta kehittyy nopeasti, GraphQL-rajapinta on RESTful-rajapintaa parempi vaihtoehto pienemmän versioinnintarpeensa ansiosta. Datan ylihakeminen ei myöskään ikinä ole ongelma GraphQL-rajapintoja käytettäessä.

RESTful-rajapinta on muissa tapauksissa GraphQL:ää parempi valinta rajapinnan tarjoamiseen, sillä GraphQL-rajapinta ei tarjoa tallaisissa tilanteissa suuria hyötyjä RESTful-rajapintaan verrattuna. RESTful-rajapintojen etuna on se, että ne ovat suositumpia kuin GraphQL-rajapinnat, jolloin useammat ihmiset osaavat hyödyntää niitä ja ne ovat vakiinnuttaneet asemansa rajapintakentällä. GraphQL on vielä uudehko teknologia, jonka käyttäminen vaatii monilta käyttäjiltä uuden opettelemista. RESTful-rajapinnat ovat myös joustavampia kuin GraphQL-rajapinnat, sillä ne tukevat suoraan lähes mitä tahansa tiedostomuotoja. RESTful-rajapinnoilla on myös ratkaisunsa datan ylihakemisen estämiseen, vaikkakin GraphQL:n lähestymistavalla on etunsa RESTful-rajapintojen ratkaisuihin verrattuna. GraphQL-rajapinnan toteuttaminen yksinkertaiselle palvelulle ei todennäköisesti ole vaivan, eli skeeman ja resolverien määrittelemisen, arvoista verrattuna RESTful-rajapinnan toteuttamisen vaivaan. Välimuistinhallinta on myös helpompaa käytettäessä RESTful-rajapintaa. Toisaalta GraphQL:n valitseminen ohjelmointirajapinnaksi yksinkertaiselle palvelulle ei aiheuta kovin suuria haittojakaan. GraphQL-rajapinnan hyödyt RESTful-rajapintaan verrattuna kasvavat rajapinnan ja kyselyjen monimutkaisuuden kasvaessa.

GraphQL- ja RESTful rajapinnat eivät ole toisiaan poissulkevia, vaan palvelu voi tarjota molemmat ohjelmointirajapinnat. Tällöin saadaan molempien rajapintojen hyödyt ja asiakas voi valita tilanteeseensa paremmin sopivan rajapinnan käyttöönsä. GraphQL-kääreen generoiminen RESTful-rajapinnalle voi onnistua jopa automaattisesti [20]. Kahden ohjelmointirajapinnan tarjoaminen ja kehittäminen kuitenkin vaatii aina enemmän työtä kuin yhden.

4 Yhteenveto

Tutkielmassa esiteltiin GraphQL- ja RESTful-verkko-ohjelmointirajapinnat ja vertailtiin niiden ominaisuuksia keskenään. Vertailuosiossa vertailtiin verkko-ohjelmointirajapinta-paradigmojen datan yli- ja alihakemista, ylläpidettävyyttä, välimuistin hyödyntämistä, rajapintojen yhdenmukaisuutta, kyselyn kirjoittamiseen kuluvaa aikaa, käytettävyyttä sekä turvallisuutta keskenään.

Vertailuosiossa todetaan, että GraphQL- ja RESTful-ohjelmointirajapinnat sopivat erilaisiin tilanteisiin. GraphQL sopii paremmin tilanteisiin, joissa kyselyt ovat monimutkaisempia ja rajapinnan on vastattava useisiin erilaisiin kyselyihin. RESTful-rajapinta sopii puolestaan paremmin tilanteisiin, joissa kyselyt ovat samankaltaisia, kyselyjen parametrien sekä resurssien erilaisten URL-polkujen määrä on pieni ja URL-polut ovat yksinkertaisia.

Kyselyjen kirjoittaminen GraphQL-rajapintaan on helpompaa ja nopeampaa kuin vastaavaan RESTful-rajapintaan etenkin, jos kyselyt ovat monimutkaisia. GraphQL vähentää myös datan alihakemista eikä datan ylihakeminen vaivaa GraphQL-rajapintoja. GraphQL-rajapinta saattaa myös olla helpommin ylläpidettävä kuin vastaava RESTful-rajapinta. Jos rajapinta ei ole nopeasti kehittyvä, datan yli- tai alihakeminen ei ole ongelma tai kyselyt eivät ole monimutkaisia, GraphQL ei juurikaan tarjoa suuria hyötyjä RESTful-rajapintaan verrattuna. Tällöin RESTful-rajapinta on suuremmasta suosiostaan ja yksinkertaisuudestaan johtuen todennäköisesti GraphQL-rajapintaa parempi valinta verkko-ohjelmointirajapinnan toteuttamiseen.

Tässä tutkielmassa rajapintojen vertailu keskittyi rajapintojen perusominaisuuksiin ja vertailussa huomioidut asiat olivat suurimmaksi osaksi asiakaskeskeisiä. Rajapintojen vertailua voisi jatkaa vertailemalla tarkemmin rajapintojen toteuttamista, rajapintojen kehittämiseen, käyttämiseen ja testaamiseen saatavilla olevia työkaluja sekä rajapintojen kehittämiseen kuluvaa aikaa.

Lähteet

[1] R. T. Fielding, *Architectural styles and the design of network-based software architectures (Ph.D.)*, 2000. University of California, Irvine. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[2] M. Masse, *REST API Design Rulebook*, 2011. O'Reilly Media, Inc.

[3] *Web API design*, 1.12.2018. Microsoft Docs. https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design (viitattu 05.10.2020)

[4] *REST Countries*, apilayer. https://restcountries.eu/ (viitattu: 16.10.2020)

[5] L. Richardson & M. Amundsen, RESTful Web APIs, 2013, O'Reilly.

[6] *Learn,* The GraphQL Foundation. https://graphql.org/learn/ (viitattu: 16.10.2020)

[7] L. Byron *GraphQL: A data query language,* 14.9.2015. Facebook engineering. https://engineering.fb.com/core-data/graphql-a-data-query-language/ (viitattu: 16.10.2020)

[8] *Code,* The GraphQL Foundation. https://graphql.org/code/ (viitattu: 16.10.2020)

[9] E. Porcello & A. Banks, *Learning GraphQL*, 2018 O'Reilly Media, Inc.

[10] *Thinking in Graphs,* The GraphQL Foundation. https://graphql.org/learn/thinking-in-graphs/ (viitattu: 20.10.2020)

[11] *GraphQL specification (draft)*. https://spec.graphql.org/draft/ (viitattu: 26.10.2020)

[12] *Serving over HTTP,* The GraphQL Foundation. https://graphql.org/learn/serving-over-http/ (viitattu: 20.10.2020)

[13] T. Blades, *Countries*. https://github.com/trevorblades/countries (retrieved: 22.10.2020)

[14] K. Goetsch*, GraphQL for Modern Commerce*, 2020. O'Reilly Media, Inc.

[15] *GraphQL Best Practices*, The GraphQL Foundation. https://graphql.org/learn/best-practices/ (retrieved: 28.10.2020)

[16] G. Brito, T. Mombach & M. T. Valente, *Migrating to GraphQL: A Practical Assessment,* 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 2019, pp. 140–150

[17] M. Vogel, S. Weber & C. Zirpins, *Experiences on migrating RESTful Web Services to GraphQL*, 15th International Conference on ServiceOriented Computing (ICSOC), pp. 28-295, 2017

[18] G. Brito & M. T. Valente*, REST vs GraphQL: A Controlled Experiment,* 2020 IEEE International Conference on Software Architecture (ICSA), Salvador, Brazilia, 2020, pp. 81–91

[19] O. Hartig & J. Pérez, *Semantics and complexity of GraphQL*, 27th World Wide Web Conference on World Wide Web (WWW), pp. 1155-1164, 2018.

[20] C. Wittern, *Generating GraphQL-Wrappers for REST(-like) APIs*, 2018, International Conference on Web Engineering (ICWE) 2018, pp. 65–83