Haaga-Helia
ammattikorkeakoulu Oy

REST vs. Falcor - Rajapintamallien evoluutio

Marion Karlsson

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2019

Haaga-Helia ammattikorkeakoulu Oy

Tiivistelmä

17.3.2019

Tekijä(t)

Marion Karlsson

Koulutusohjelma

Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön otsikko

REST vs. Falcor - Rajapintamallien evoluutio

Sivu- ja liitesivumäärä

59 + 0

Opinnäytetyön otsikko englanniksi

REST vs. Falcor – Revolution of API's

Rajapinnat mahdollistavat eri ohjelmistojen ja teknologien välisen kommunikoinnin, sekä toimivat nykypäivänä avainasemassa tiedonvälityksessä. Rajapintojen rakentaminen on tullut isoksi osaksi ohjelmistokehitystä ja erilaisia teknologioita rajapintojen kehitykseen on syntynyt useita.

Opinnäytetyö vertailee kahta toisistaan poikkeavaa rajapintamallia. Tutkimuksen kohteena olivat REST-arkkitehtuuri ja Falcor-kirjasto. Molemmilla teknologioilla rakennettiin toiminnoiltaan samanlainen rajapinta ja kehitystyö dokumentoitiin.

Rajapintojen lähdekoodi kirjoitettiin JavaScript kielellä NodeJS-ympäristössä, Express-kehystä hyödyntäen.

Kehitystyön tarkoituksena oli tarkastella rajapinnan rakentamisten prosessia näillä kahdella teknologialla ohjelmistokehittäjän näkökulmasta. Tutkimuksessa selvitettiin, kumpi rajapintamalli vastaa paremmin ohjelmistokehittäjien tarpeisiin teknisten ominaisuuksien, dokumentoinnin ja yhteisön osalta.

Tutkimustuloksissa päädyttiin johtopäätökseen siitä, että tutkitut rajapintamallit olivat toiminnoiltaan ja käyttötarkoituksiltaan täysin erilaiset. Molemmissa malleissa oli omat vahvuutensa ja heikkoutensa, joita vertailtiin työn lopussa.

Asiasanat

nodejs, ohjelmistokehitys, falcor, rajapinta, api, rest

Sisällys

1 Sanasto

Express	Framework eli kehys NodeJS-ympäristöön, jonka avulla voidaan helposti kehittää web-sovelluksia ja rajapintoja HTTP-protokollalla.
Hypermedia	Hypermedialla tarkoitetaan kokonaisuutta, joka voi sisältää erilaista mediaa, kuten tekstiä, ääntää, kuvaa ja videota.
JSON	JSON (JavaScript Object Notation) on standardisoitu tapa esittää dataa avainarvo pareilla.
MongoDB	NoSQL tietokanta, joka säilöö JSON-tyyppisiä dokumentteja.
NodeJS	Ympäristö JavaScript-ohjelmointikielelle, jonka avulla kieltä voidaan suorittaa selaimen ulkopuolella.
Palvelin	Palvelimella tarkoitetaan tietokonetta ja siinä suoritettavaa palvelinohjelmistoa, jonka tehtävänä on tarjota palveluja ohjelmille ja sovelluksille verkkoyhteyden välityksellä.
Rajapinta	Rajapinta, yleisesti tunnetulta nimeltään API (Application Programming Interface) tarkoittaa määritelmää siitä, miten ohjelmat kommunikoivat toistensa kanssa.
Protokolla	Protokollalla tarkoitetaan käytäntöä tai standardia, jolla määritellään laitteiden ja ohjelmistojen väliset yhteydet. Molemmilla osapuolilla on selkeä käsitys siitä, minkälaisia viestejä toiselta voi vastaanottaa tai mitä toiselle voi lähettää.
URI	URI (Uniform Resource Indentifier) on tekstiä, joka toimii yksittäisen resurssin tunnisteena.
WWW	WWW (World Wide Web) on laaja verkosto resursseja, joihin on mahdollista päästä käsiksi URI-osoitteiden avulla HTTP-protokollan yli.

2 Johdanto

Rajapinnat ovat kaikkialla. Ne helpottavat ja yksikertaistavat ohjelmistokehittäjien työtä uusien ja olemassa olevien palveluiden kehittämisessä, antamalla tehokkaan keinon jakaa ja käsitellä tietoa. Hyvin suunnitellun rajapinnan tärkeimpiä ominaisuuksia ovat selkeä käytettävyys, johdonmukaisuus, sekä sen kyky pystyä vastaamaan loppukäyttäjien tarpeita.

On syntynyt useita toisistaan poikkeavia tapoja rakentaa rajapintoja. Tarkoitus on rajata tutkimus käsittelemään näistä kahta, REST- ja Falcor-mallia. REST on yksi käytetyimmistä ja vakiintuneimmista rajapintamalleista, kun taas Falcor on yksi uusimmista tavoista rakentaa rajapintoja. Tavoitteena on selvittää, kumpi näistä arkkitehtuureista vastaa parhaiten loppukäyttäjien, eli ohjelmistokehittäjien tarpeita.

Falcorin lisäksi markkinoille on tullut muitakin uusia rajapintateknologioita. Yksi niistä on Facebookin kehittämä GraphQL, joka on kerännyt ympärilleen laajan, uskollisen yhteisön. Tästä teknologiasta löytyy jo kuitenkin useita tutkimuksia, joten en uskonut, että olisi mahdollista tuottaa uutta tutkimusta, joka tuottaisi lisäarvoa ohjelmistokehittäjille. Falcor on taas teknologia, jota ei ole vielä tutkittu, joka taas tekee siitä hyvän tutkimuksen kohteen.

2.1 Tavoitteet

Projektin tavoitteena on rakentaa kaksi toiminnoiltaan täysin samanlaista rajapintaa, toinen REST-mallilla ja toinen Falcor-mallilla.

Tarkoituksena on vertailla näiden kahden rajapinnan kehitystyötä ohjelmistokehittäjän näkökulmasta. Tärkein kysymys on, kuinka helppoa asiakkaan, eli ohjelmistokehittäjän on rakentaa rajapinta tutkitulla teknologialla. Ovatko rajapintojen dokumentaatiot selkeitä ja helposti saatavilla? Onko rajapintaa helppo dokumentoida, jolloin sitä hyödyntävät loppukäyttäjät ovat tietoisia rajapinnan tarjoamista resursseista ja ominaisuuksista?

Näihin kysymyksiin haetaan vastauksia tutkimalla rajapintamalleista tarjolla olevia teknisiä dokumentaatioita, tutkimuksia ja artikkeleita. Niistä rakennetaan kuva rajapinnan rakenteesta ja toiminnallisuuksista, joiden pohjalta tullaan kehittämään toimiva rajapinta. Rajapinnan kehitystyö dokumentoidaan, jonka avulla haetaan vastauksia esitettyihin tutkimuskysymyksiin.

2.2 Tehtävän rajaus

Tehtävänä on vertailla ainoastaan rajapintojen kehittämistä, joten projektissa ei rakenneta käyttöliittymää datan käsittelylle ja visualisoinnille.

Projektissa ei myöskään käsitellä käytetyn pohjaprojektin tietokanta- ja autentikaatioratkaisuja, vaan keskitytään rajapinnan toiminnallisuuksiin. Tietokantaratkaisuilla tarkoitetaan kyselylausekkeita, joilla dataa lähetetään ja pyydetään käytettävästä tietokannasta. Autentikaatioratkaisuilla tarkoitetaan käyttäjän salasanan suojausta.

2.3 Ympäristö

2.3.1 Toteutusympäristö

Molemmat rajapinnat rakennetaan JavaScriptillä Node.js ympäristössä, Express-kehystä hyödyntäen.

Tietokantana toimii MongoDB.

Koodi kirjoitetaan JetBrains:in WebStorm-editorilla, joka on tarkoitettu JavaScript-ohjelmointikielellä kehittämiseen.

2.3.2 Rajapinnan toiminnot

Rajapintojen tarkoituksena on toimia palveluna, jossa voi selailla erilaisia tietokone- ja konsolipelejä. Käyttäjien on myös mahdollista rekisteröityä käyttäjiksi ja tallentaa suosikkipelejään omiin listoihinsa. Käyttäjä voi tarkastella tallentamiaan pelejä, sekä tarpeen tullen myös poistaa niitä.

Molemmat rajapinnat käyttävät samaa pohjaa, johon on jo kehitetty autentikaatio ja tarvittavat luokat ja metodit käytettäville tietokantakyselyille. Ainoa puuttuva osa on rajapinta, jonka avulla käyttäjä pääsee käsiksi kannan tietoihin.


```
_id: ObjectId("5c81309ee998663368cbf279")
› games: Array
username: "marisanity"
email: "marion.karlsson@gmail.com"
password: "$2b$10$yOBcPpsvsYFQ9TfmYUtcVebNJl6uWZdFro6zrSbPgLWMS29nEYPX6"
__v: 47
```


Kuva 1. Yksittäinen käyttäjä-dokumentti tietokannasta

Kuvassa yksi näkyy esimerkki kantaan tallennetusta käyttäjästä.

- _id on käyttäjän henkilökohtainen, uniikki tunniste
- games-lista on lista, johon käyttäjän valitsemat pelit tallennetaan
- username on käyttäjän uniikki kutsumanimi palvelussa
- email on käyttäjän sähköpostiosoite
- password on käyttäjän salasana suojatussa muodossa

Tietokannassa on valmiina 65 300 peliä, jotka on haettu GiantBomb-sivuston tietokannasta ja tallennettu lokaaliin tietokantaan, jossa rajapintojen kehitystyö tapahtuu.

Kuva 2. Yksittäinen peli-dokumentti tietokannasta

Kuvassa kaksi näkyy esimerkki kannassa esiintyvästä pelistä. Osa kentistä sisältävät tekstin ”null”, eli ovat tyhjiä, koska kaikista kannan peleistä ei ole kaikkia tietoja vielä saatavilla.

- _id on pelin henkilökohtainen, uniikki tunniste
- name on pelin englanninkielinen nimi
- aliases on lista, johon on laitettu pelin toissijaiset, tunnetut nimet.
- date_added on aika, milloin peli on lisätty GiantBomb-palveluun
- deck on lyhyt kuvaus pelistä
- description on pitkä kuvaus pelistä
- image on objekti, jonka sisällä on kuvia pelistä
- platforms on lista, jossa on kerrottu mille alustoille peli on julkaistu
- original_release_date on aika, jolloin peli on julkaistu

2.3.3 Rajapinnan vaatimukset

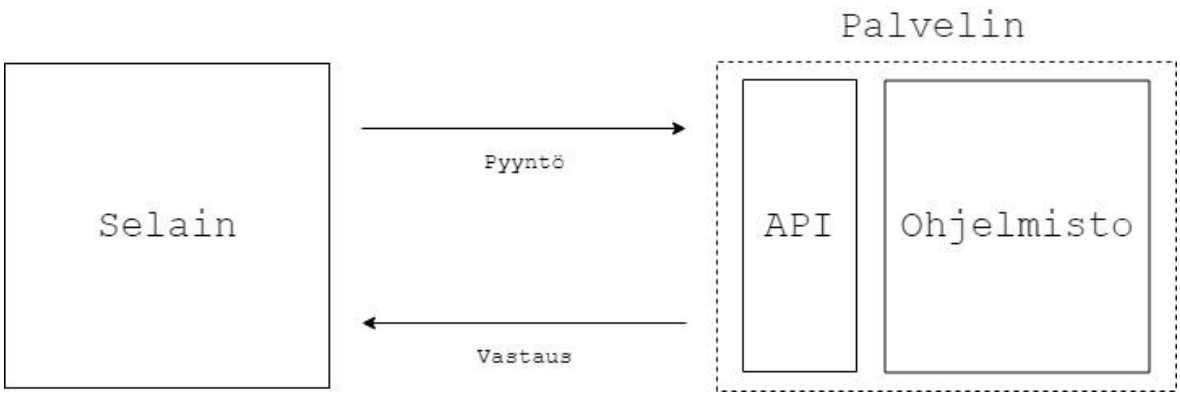
Rajapinnan vaatimuksena on, että käyttäjä pystyy

- Selaamaan pelejä kirjautumatta palveluun
- Rekisteröitymään palveluun
- Kirjautumaan palveluun
- Tallentamaan haluamiaan pelejä omaan listaansa
- Poistamaan pelejä omasta listastaan

Uloskirjautuminen on jätetty pois vaatimuksista, sillä tämä toiminto kuuluu käyttöliittymälle, joka poistaa välimuististaan token-tunnisteen. Palvelimen ei kuulu olla tietoinen käyttäjän session tilasta.

3 REST

Vuonna 2000 tutkija Roy Fielding esitteli väitöskirjassaan REST-arkkitehtuurityylin (Representational State Transfer), jota olisi tarkoitus soveltaa hajautettujen hypermedia-järjestelmien kehittämisessä. Näillä tarkoitetaan järjestelmiä, jotka tarjoavat HTTP-protokollan yli palveluita muille tietokoneille tai palvelimille, joita kutsutaan myös verkko- tai sovelluspalveluiksi. Tässä tilanteessa asiakas, kuten verkkoselain tekee pyyntöjä HTTP-protokollan yli REST-rajapintaan, joka vastaanottaa pyynnöt ja antaa ne palvelimella sijaitsevalle ohjelmistolle käsiteltäviksi (Kuva 3). (Fielding 2000, 76-78.)



Kuva 3. HTTP-pyyntö ja vastaus asiakkaalta palvelimelle

REST-arkkitehtuurimalli koostuu ohjeistuksista, joiden puitteissa rajapinnan tulisi toimia. Sen lisäksi, että malli suosittelee nojaamaan vahvasti HTTP-protokollan ominaisuuksiin, pyytää se esimerkiksi pitämään huolen siitä, että resursseihin osoittavat URI:t ovat yksiselitteisiä. Myös asiakkaan ja palvelimen välisen istunnon tulee olla tilaton. (Pivotal 2019)

Tiedonvälityksessä käytetyn datan formaattiin REST ei ota kantaa. Arkkitehtuurimalli on myös kieliriippumaton, joten REST-rajapinnan rakentaminen onnistuu kaikilla ohjelmistokielillä. (Rouse 2017)

3.1 HTTP-Protokolla

HTTP-protokolla on kokoelma sääntöjä, kuinka tietoa siirretään verkossa selainten ja WWW-palvelimien välillä. Lyhenne HTTP tulee sanoista Hypertext Transfer Protocol, joka tarkoittaa hypertekstidokumenttien eli HTML-tiedostojen siirtoon tarkoitettua protokollaa. Nimestään huolimatta protokollalla siirretään kaikenlaista dataa ja lisätietoja lähetettävästä datan tyypistä annetaan tiedonsiirron vastauksen tai pyynnön osoitteessa ja otsikoissa. (Korpela 2009)

3.1.1 Otsikot

Otsikot (headers) välittävät lisätietoja pyynnöistä ja vastauksista selaimelta palvelimelle ja takaisin. (Fielding, Gettys, Mogul, Frystyk, Masinter, Leach, Berners-Lee 1999, 99-144.)

```
GET /hello.htm HTTP/1.1
        User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
        Host: www.example.com
        Accept-Language: en-us
        Accept-Encoding: gzip, deflate
        Connection: Keep-Alive
```

Kuva 4.

Taulukko 1. Pyynnön otsakkeet kuvasta neljä

User-Agent	Kertoo lisätietoja verkkoselaimesta
Host	Pyyntö kertoo mihin osoitteeseen sitä kohdennetaan
Accept-Language	Kertoo millä kielellä/kielillä mahdollisen vastauksen sisältö on
Accept-Encoding	Kertoo missä muodossa mahdollisen vastauksen sisältö hyväksytään
Connection	Tarkentaa mitä pyynnön yhteydeltä odotetaan

```
HTTP/1.1 200 OK
    Date: Mon, 27 Jul 2009 12:28:53 GMT
    Server: Apache/2.2.14 (Win32)
    Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
    Content-Length: 88
    Content-Type: text/html
    Connection: Closed
```

Kuva 5.

Taulukko 2. Vastaukset otsakkeet kuvasta viisi

Date	Kertoo vastauksen muodostuksen ajankohdan
Server	Kertoo lisätietoja palvelimen ohjelmistosta, josta vastaus lähetettiin
Last-Modified	Kertoo milloin vastauksen sisältöä on viimeksi muutettu
Content-Length	Kertoo vastauksessa lähetetyn viestin koon
Content-Type	Kertoo vastauksessa lähetetyn viestin formaatin

3.1.2 HTTP metodit

HTTP-protokolla määrittelee pyyntöihin liitettäviä metodeita, joilla on tarkoitus

viestiä palvelimelle, mitä toimenpiteitä suoritetaan resurssia kutsuttaessa. Näitä kutsutaan myös HTTP-verbeiksi. (MDN web docs 2018.)

Taulukko 3. Yleisimmät HTTP metodit

GET	Resurssin haku palvelimelta
POST	Resurssin lähettäminen palvelimelle
DELETE	Resurssin poisto palvelimelta
PUT	Resurssin muokkaus palvelimella

3.2 Asiakas-palvelin malli

Asiakas-palvelin mallissa palvelimen tehtävänä on tarjota
palveluja asiakkaalle, kuten verkkoselaimelle ja vastata tämän pyyntöihin hyväksyvästi tai kieltävästi. Mallin ajatuksena on parantaa vikasietoisuutta ja skaalautuvuutta yksinkertaistamalla palvelimen rakennetta niin, että se ei ole vastuussa käyttöliittymän toiminnoista, joista asiakaskomponentti ottaa vastuun. Tämä mahdollistaa myös molempien komponenttien itsenäisen kehityksen. (Fielding 2000, 78.)

3.3 Tilattomuus

Asiakkaan ja palvelimen yhteyden on aina oltava tilaton. Asiakkaan on siis välitettävä resurssia pyydettäessä kaikki tarvittava tieto palvelimelle, sillä palvelimelle ei tallenneta mitään asiakkaan tietoja, joita voisi hyödyntää kyselyä suorittaessa. Asiakkaan tehtävänä on siis ylläpitää session tilaa. (Fielding 2000, 78-79.)

3.4 Välimuisti

Välimuistin hyödyntäminen mahdollistaa verkon tehokkaan käytön ja nopeuttaa palvelimen ja asiakkaan välistä kommunikaatioita. REST-rajapinnasta tulleet vastaukset kertovat asiakkaalle saako niiden mukana tullutta tietoa tallentaa välimuistiin. Kun vastauksen mukana tulee lupa tiedon tallennukseen, asiakkaan välimuistille annetaan oikeus tallentaa ja käyttää viestin mukana tullutta tietoa tarvittaessa uudestaan. (Fielding 2000, 79-81.)

3.5 Kyselyt

REST-rajapintaan tehtävät kyselyt ovat HTTP-protokollan yli tehtäviä pyyntöjä palvelimelle, jotka koostuvat useasta eri osasta. Näitä voidaan kutsua myös HTTP-pyynnöiksi. (Virender 2017)

```
GET /users HTTP/1.1
Host: api.example.com
```

Kuva 6. GET-pyyntö

Pyyntöön sisältyy vähintään resurssin osoite, eli URI, HTTP-metodi ja Host-otsake, jossa kerrotaan palvelimen osoitetiedot. Kuvassa kuusi nähdään yksinkertainen pyyntö, jossa haetaan käyttäjiä palvelimelta. GET-metodilla kerrotaan palvelimelle, että tarkoituksena on hakea dataa palvelimelta. Sen vieressä on osoite */users*, jossa resurssit sijaitsevat. Host-otsake kertoo palvelimen osoitteen. Kyselyn resurssin osoite on siis kokonaisuudessaan api.example.com/users. (MDN web docs, 2019)

```
POST /users
Host: api.example.com
Content-Type: application/json

{"user_id": 1, "email": "test.user@example.com", "password": "secret"}
```

Kuva 7. POST-pyyntö

Jos kyseessä on POST-, DELETE- tai PUT-pyyntö, niin mukana on usein myös HTTP-body, eli palvelimelle lähetettävä resurssi (Kuva 7). (Sturgeon 2015, 25-26)

3.5.1 Kyselyparametrit ja osoitemuuttujat

Kyselyissä voidaan lähettää erilaisia muuttujia, joilla tarkennetaan kyselyn ehtoja. Myös HTTP-bodyn mukana voidaan myös lähettää tarkentavia tietoja. GET-pyynnössä se ei taas ole mahdollista, mutta kyselyparametrejä on mahdollista käyttää kaikissa HTTP-pyynnöissä. (MDN web docs, 2019)

```
GET /users?page=1&pageSize=25 HTTP/1.1
Host: api.example.com
```

Kuva 8. GET-pyyntö kyselyparametrilla

Kuvassa kahdeksan on käytetty kyselyparametreja, jotka kulkevat resurssiin osoittavan URI:in mukana. Kysymysmerkillä erotetaan osoite ja kyselyparametrit toisistaan. Jos parametreja on useita, niiden välille asetetaan et-merkki. Tässä esimerkissä kyselyparametrejä käytetään sivutuksen apuna. Palvelimelle lähetetään parametri *page*, jonka arvo on 1 ja parametri *pageSize*, jonka arvo on 25. Kyselyparametreja suositellaan käytettävän filtte-

röinnissä, kun osoitemuuttujia taas kannustetaan käytettävän päätepisteissä, kun halutaan hakea resursseja yksilöidyllä tunnisteella. (Sturgeon 2015, 25.)

```
GET /users/matti HTTP/1.1
Host: api.example.com
```

Kuva 9. GET-pyyntö osoitemuuttujalla

Kuvassa yhdeksän käytetään osoitemuuttujaa kyselyssä, jossa haetaan käyttäjää käyttäjänimen ”matti” perusteella. Toisin kuin kyselyparametrissa, osoitemuuttujassa ei tarvitse tarkentaa lähetettävän arvon parametrin nimeä suoraan osoitteessa. (Medium 2017)

3.6 Vastaukset

Kun asiakas tekee onnistuneen kyselyn palvelimelle, sieltä lähetetään vastaus. Vastauksessa kerrotaan käyttäjälle, onnistuiko kysely ja jos kyselyssä on pyydetty resursseja, tulevat ne vastauksen mukana pyydetyssä muodossa. (Fielding, Gettys, Mogul, Frystyk, Masinter, Leach, Berners-Lee 1999, 39.)

```
HTTP/1.1 200 OK
Server: nginx
Content-Type: application/json
Connection: close
Cache-Control: no-cache, private
X-Powered-By: PHP/5.5.5-1+debphp.org~quantal+2
Date: Fri, 22 Nov 2018 16:37:57 GMT
Transfer-Encoding: Indentity

{
    "id":"5687",
    "name":"Theron Weissnat",
    "email":"theron.weissnat@example.com",
    "isActive":true
}
```

Kuva 10. Vastaus HTTP-pyyntöön

Kuvassa 10 näkyy esimerkki palvelimelta tulleesta vastauksesta. Ensimmäisellä rivillä kerrotaan onnistuneesta kyselystä, jonka jälkeen näkyy otsakkeita, jotka kertovat lisätietoja palvelimesta, yhteydestä ja lähetetystä resurssista. Alimmaisena, aaltosulkujen sisällä näkyy HTTP-body, eli mukana tullut resurssi, joka sisältää dataa käyttäjästä JSON-muodossa. (Strugeon 2015, 27-30.)

3.7 Resurssit ja päätepisteet

Päätepisteillä (endpoints) tarkoitetaan osoitetta, jossa resurssi sijaitsee. Päätepiste voi esimerkiksi olla https://example.com/users, mutta siihen voidaan yksinkertaisemmin viitata /users, sillä palvelimelle osoittava reitti, eli domain pysyy muuttumattomana. (Sturgeon 2015, 13.)

Rajapintaa suunniteltaessa on mietittävä, mitä toimintoja siltä vaaditaan. Niistä kertoo asiakas, oli se ohjelmistokehittäjä, ohjelmiston tilaaja tai loppukäyttäjä.
Vaatimusten perusteella muodostetaan lista resurssiin osoittavista päätepisteistä, jotka jakautuvat CRUD-toimenpiteisiin (create, read, update, delete). Päätepisteitä muodostaessa on tärkeää pitää rakenne yksinkertaisena ja nimeäminen selkeänä. Jokaiselle resurssille on myös hyvä rakentaa oma kontrollerinsa, jossa käsitellään vain siihen liittyviä kyselyitä. (Sturgeon 2015, 13-24)

Demonstroidaan yhtä resurssia, tässä tilanteessa käyttäjää koskevia päätepisteitä käytännössä. Käytetään esimerkkinä palvelua, jossa käyttäjät voivat selata elokuvia ja tallentaa suosikkejaan henkilökohtaisiin listoihin.

Taulukko 4. Datan haku

GET /users	Haetaan kaikki käyttäjät.
GET /users/X	Haetaan vain yksi käyttäjä X. X voi olla id, käyttäjänimi tai mikä vain kehittäjän määrittelemä tunniste.
GET /users/X/favorites	Haetaan tietyn käyttäjän listan kaikki elokuvat
GET /users/X/favorites/Y	Haetaan tietyn käyttäjän listalta tietty elokuva

Taulukko 5. Datan poistaminen

DELETE /users/X	Poistetaan yksi käyttäjä
DELETE /users	Poistetaan kaikki käyttäjät. Kyseenalainen kysely, jota harvemmin suositellaan kehittämään rajapintaan.
DELETE /users/X/favorites	Poistetaan tietyn käyttäjän listan kaikki elokuvat
DELETE /users/X/favorites/Y	Poistetaan tietyn käyttäjän tietty elokuva listalta

```
POST /users HTTP/1.1
Host: example.com
Content-Type: application/json

{"username": "user", "email:" "new.user@example.com", "password": "secret"}
```

Kuva 11. POST-pyyntö

Kuvassa 11 näkyy pyyntö uuden käyttäjän lisäyksestä. Aaltosulkujen ympäröimä teksti on JSON-formaatissa oleva resurssi, joka kulkee HTTP-bodyssa palvelimelle.

```
PUT /users/X HTTP/1.1
Host: example.com
Content-Type: application/json
Authorization: secret

{"email:" "new.user@example.com"}
```

Kuva 12. PUT-pyyntö

Kuvassa 12 näkyy pyyntö, jonka tarkoituksena on muokata olemassa olevan resurssin, eli käyttäjän X tietoja.

3.8 HTTP-statuskoodit ja virheviestit

Kyselyn mennessä läpi rajapinta kertoo onnistumisesta ja jos käsittely keskeytyy, kerrotaan virheestä. Ongelmien sattuessa rajapinnan on pystyttävä ilmoittamaan, johtuiko keskeytyminen asiakkaasta vai oliko kyse ohjelmiston tai palvelimen viasta. (Kapadnis 2018)

Statuskoodeja on useita erilaisia ja niistä ilmoitetaan numeroilla, jotka liikkuvat välillä 200-507. Jokaisella koodilla on oma viestinsä ja määritelmänsä. Numerolla kaksi alkava koodi kertoo onnistumisesta, kolmella alkavalla kerrotaan uudelleenohjauksesta, neljällä alkavalla asiakkaan virheestä ja viidellä alkava kertoo palvelussa ilmenneestä virheestä. (Kapadnis 2018)

4 REST-rajapinnan rakentaminen

4.1 Rajapinnan suunnittelu

REST-rajapintaa suunnitellessa on mietittävä ensin, mitä vaatimuksia rajapinnalle on asetettu ja sen avulla määriteltävä resursseihin johtavat päätepisteet.

4.1.1 Päätepisteet

Koska vaatimukset ovat jo selvillä, on mahdollista siirtyä suoraan päätepisteiden suunnitteluun. Vaatimukset käytiin läpi johdannossa.

Resursseja on vain kaksi, joten rajapinta saadaan pidettyä yksinkertaisena. Uloskirjautumiselle ei rakenneta omaa päätepistettään rajapintaan, sillä noudatamme REST:in periaatteita, jolloin palvelin ei huolehdi käyttäjän tilasta, vaan vastuu on asiakkaalla, eli tässä tapauksessa käyttöliittymällä.

Taulukko 6. Käyttäjät-resurssin suunnittelu

Käyttäjät		
Toiminto	Päätepiste	Selite
POST	/users	Luodaan uusi käyttäjä
GET	/users/{username}	Haetaan yhden käyttäjän tiedot
POST	/users/{username}/games	Lisätään yksi peli yhden käyttäjän listaan
DELETE	/users/{username}/games/{gameId}	Poistetaan yhden käyttäjän listalta yksi peli

Vaikka sisäänkirjautuminen liittyy selkeästi käyttäjään, se voidaan laittaa omaksi resurssikseen. Jos rajapintaa laajennetaan ja jatkokehitetään, on mahdollista lisätä erilaisia tapoja tunnistautumiseen, jolloin kirjautumisesta kasvaa liian iso kokonaisuus käyttäjäresurssin alle.

Taulukko 7. Autentikaatio-resurssin suunnittelu

Autentikaatio		
Toiminto	Päätepiste	Selite
POST	/auth	Kirjaudutaan sisään järjestelmään

Taulukko 8. Pelit-resurssin suunnittelu

Pelit		
Toiminto	Päätepiste	Selite
GET	/games	Haetaan kaikki pelit
GET	/games/{id}	Haetaan yhden pelit tiedot

4.2 Resurssien luonti

Seuraavaksi ohjelmistoon rakennetaan varsinainen rajapinta. Peli-, käyttäjä- ja autentikaatiokutsut erotetaan omiin luokkiinsa, joka tekee koodista helppolukuista ja selkeää.

4.2.1 Pelit

Luodaan games.js tiedosto, johon asetetaan kaikki pelejä koskevat pyynnöt.

```
const express = require('express');
const gameRepository = require('./../db/repositories/gameRepository');
const router = express.Router();
```

Kuva 13. Riippuvuudet luokassa games.js

Kuvassa 13 näkyy tiedostoon liitetyt riippuvuudet. Express-kirjastosta saadaan avuksi router-luokka, jonka metodien avulla voidaan helposti määrittää kuinka ohjelma reagoi asiakkaalta tiettyyn päätepisteeseen tulleeseen pyyntöön.

Toinen riippuvuuksista on gameRepository, joka sisältää useita erilaisia metodeja, joilla otetaan yhteys tietokannan pelit-kokoelmaan. Tästä luokkaa käyttämällä pelejä on mahdollista hakea, lisätä ja pyytää tarpeen tullen.

```
router.get('/', async (req, res) => {
    const pageNumber = parseInt(req.query.pageNumber);
    const pageSize = parseInt(req.query.pageSize);

    const games = await gameRepository.get(pageNumber, pageSize);

    if (games) return res.status(200).send(games);
    return res.status(401).send("No results");
});
```

Kuva 14. GET-metodi pelien hakemiseen

Router-luokan get-metodia käyttäen luodaan vastaanotto pyynnölle /games, jolla halutaan hakea tietokannasta kaikki pelit (Kuva 14). Http-kyselyn saapuessa kyseiseen päätepisteeseen, metodi vastaanottaa argumentteina pyyntö- (req) ja vastaus-objektit (res), jotka sisältävät vastaanotetun pyynnön sisällön ja takaisin lähetettävän vastauksen.

Koska tietokannassa on pelejä useita kymmeniä tuhansia, kaikkia ei ole järkevää lähettää kerralla takaisin asiakkaalle. Pyyntö olisi raskas, sillä dataa on niin paljon, että sen lähettäminen takaisin kestäisi useita sekunteja, jopa minuutteja.

Kun dataa on runsaasti, käyttöliittymässä data on useasti jaetttu monelle eri sivulle. Tätä hyödynnetään myös rajapinnassa. Asiakkaalta vastaanotetaan kyselyparametreina pageSize, joka kertoo, kuinka paljon dataa näytetään asiakkaalle per sivu. Parametri pageNumber kertoo miltä sivulta dataa halutaan. Näin pyynnöissä lähetettävät datamäärät ja pyyntöjen nopeudet pysyvät kohtuullisina.

Seuraavaksi kutsutaan gameRepository-luokan get-metodia, joka hakee tietokannasta pelit pyydetyn sivunumeron ja sivumäärän avulla. Metodista palautettu vastaus tallennetaan games-muuttujaan. Jos muuttuja ei ole tyhjä, palautetaan se sisältöineen takaisin asiakkaalle mukanaan statuskoodi 200, joka kertoo pyynnön onnistumisesta. Jos muuttuja on tyhjä, palautetaan asiakkaalle vain statuskoodi 401 ja virheviesti, jotka molemmat kertovat pyynnön epäonnistumisesta.

```
router.get('/:id', async (req, res) => {
    const {id} = req.params;

    const game = gameRepository.findById(id);

    if (game) return res.status(200).send(game);
    return res.status(401).send("No results");
});
```

Kuva 15. GET-metodi pelien hakemiseen id:n perusteella

Kuvassa 15 on muodostettu vastaanotto pyynnölle /games/{id}, jolla haetaan pelit id-parametrin perusteella. Router:in avulla muodostettu vastaanotto on muuten samanlainen kuin edellisessä pyynnössä, mutta koska kyseessä on GET-pyyntö asiakkaalta, käytetään Router-luokan get-metodia.

Kyselyn pyynnön osoitteesta irrotetaan osoitemuuttuja, joka on nimetty palvelimella id:ksi ja tallennetaan se samannimiseen muuttujaan.

Tämän jälkeen kutsutaan gameRepository-luokan metodia, jolla voidaan hakea pelejä id:n perusteella ja tallennetaan vastaus game-muuttujaan.

Tämän jälkeen muuttujan sisältö tarkistetaan samalla tavalla kuin kaikkia pelejä hakiessa ja lähetetään lopputulos asiakkaalle.

4.2.2 Käyttäjät ja autentikaatio

Käyttäjiä koskevia pyyntöjä varten luodaan luokka users.js.

```
const auth = require('./../middleware/auth');
const _ = require('lodash');
const authUtils = require('./../utils/auth');
const express = require('express');
const userRepository = require('./../db/repositories/userRepository');
const router = express.Router();
const {validate} = require('./models/user');
```

Kuva 16. Riippuvuudet users-luokassa

Kuvassa 16 näkyvät users-luokan riippuvuudet. Tähän on tullut uusina riippuvuuksina auth-luokka, jossa autentikoidaan käyttäjät salasanoja käyttäen. Lodash on kolmannen osapuolen kirjasto, joka tarjoaa hyödyllisiä metodeita yksinkertaistamaan koodia. AuthUtils-luokka sisältää metodeita, joiden avulla suojataan salasana ennen kantaan tallentamista. UserRepository-luokan metodeita käyttäen saadaan haettua, lisättyä ja muokattua tietokannan käyttäjiä. Validate-muutujaan /models/users-luokasta tallennetun metodin avulla validoidaan pyynnöllä mukana saapunut, asiakkaan lähettämä HTTP-bodyssa sijaitseva data. Luokassa on käytetty kolmannen osapuolen Joi-kirjastoa.

```
router.post('/', async (req, res) => {
    const {error} = validate(req.body);
    if (error) return res.status(400).send(error.details[0].message);

    let user = await userRepository.findByEmail(req.body.email);
    if(user) return res.status(400).send('User already registered');

    user = await userRepository.findByUsername(req.body.username);
    if(user) return res.status(400).send('Username taken');

    req.body.password = await authUtils.hashPassword(req.body.password);

    const created = await userRepository.create(req.body);
    res.send(_.pick(created, ['_id', 'username', 'email']));
});
```

Kuva 17. Käyttäjän rekisteröityminen

Koska kuvassa 17 on luotu päätepiste /users käyttäjien rekisteröitymiseen, käytetään siinä router-luokan post-metodia, koska tarkoituksena on vastaanottaa POST-pyyntö asiakkaalta.

Tämän jälkeen validoidaan validate-metodilla käyttäjän lähettämä käyttäjä, joka irrotetaan http-bodysta. Siinä varmistetaan, että palvelimelle lähetetyssä JSON-viestissä on uuden käyttäjän sähköposti, käyttäjätunnus ja tuleva salasana. Jos tarvittavia tietoja puuttuu, lähetetään virhekoodi takaisin asiakkaalle.

Jos tarvittavat tiedot on vastaanotettu asiakkaalta, tarkistetaan löytyykö jo käyttäjää samalla sähköpostiosoitteella tai käyttäjänimellä. Jos löytyy, virhekoodi ja viesti lähetetään asiakkaalle. Jos kumpaakaan ei löydy, suojataan käyttäjän salasana ja tallennetaan se asiakkaalta tulleen alkuperäisen salasanan päälle.

Tämän jälkeen kutsutaan userRepository-luokan metodia vastaanotetulla käyttäjällä ja yritetään tallentaa käyttäjä kantaan ja vastaanotetaan muuttujaan metodin vastaus. Jos käyttäjä tallentui onnistuneesti, palautetaan käyttäjälle onnistumisesta viestivä statuskoodi ja tallennettu käyttäjä, tietenkin ilman salasanaa.

```
router.get('/:username', async (req, res) => {
    const {username} = req.params;

    const user = await userRepository.findByUsername(username);

    if (user) return res.status(200).send(user);
    return res.status(401).send("No results");
});
```

Kuva 18. Käyttäjän haku

Kuvassa 18 on luotu vastaanotto pyynnölle /users/{username}, jossa haetaan käyttäjä käyttäjänimen perusteella. Pyynnön osoitteesta irrotetaan osoitemuuttuja username, joka tallennetaan samannimiseen muuttujaan.

Tämän jälkeen muuttujalla haetaan userRepository-luokan metodin avulla kannasta käyttäjää samannimisellä tunnuksella. Jos käyttäjä löytyy, lähetetään se asiakkaalle onnistumisesta kertovan statuskoodin kanssa. Jos käyttäjää ei löydy, ilmoitetaan epäonnistumisesta.

```
const jwt = require('jsonwebtoken');
const config = require('config');
```

Kuva 19. Riippuvuudet middleware/auth-luokassa

Ennen kuin rakennetaan vastaanottoa päätepisteisiin, jossa käyttäjän on oltava kirjautunut sisään, on luotava väliohjelmisto (middleware), eli uusi luokka, jonka avulla tunnistetaan käyttäjä. Ensin lisätään riippuvuudeksi kolmannen osapuolen jsonwebtoken-kirjasto, jonka avulla saadaan luotua jokaiselle kirjautuneelle käyttäjälle uniikki tunniste, johon sisällytetään kirjautumisaika ja tarvittavat käyttäjän tiedot.

Config-riippuvuus on myös kolmannen osapuolen riippuvuus, jonka avulla saadaan helposti käsiteltyä ja haettua koneisiin tai palvelimiin tallennettuja muuttujia. Niihin tallennetaan usein arkaluonteisia tietoja, kuten erilaisia salasanoja. (Kuva 19)

```
function isTokenValid(timestamp) {
    const difference = Date.now() - new Date(timestamp*1000);

    return difference < 180 * 60 * 1000;
}
```

Kuva 20. Aikaleiman tarkistus

Kuvassa 20 on luotu metodi, jolla voidaan tarkistaa aikaleimasta, onko käyttäjä kirjautunut sisään yli vai alle kolme tuntia sitten. Jos kirjautuminen on tapahtunut alle kolme tuntia sitten, palautetaan arvo tosi ja jos näin ei ole, palautetaan arvo epätosi.

```
module.exports = function (req, res, next) {
    const token = req.header('x-auth-token');
    if(!token) return res.status(401).send();

    try {
        req.user = jwt.verify(token, config.get('jwtPrivateKey'));

        const isValid = isTokenValid(req.user.iat);
        if (!isValid) return res.status(401).send();

        next();
    } catch (ex) {
        res.status(401).send();
    }
};
```

Kuva 21. Tunnisteen validointi

Seuraavaksi luodaan metodi (Kuva 21), joka ottaa vastaan samat parametrit, kuin aikaisemmin luodut router-luokkaa käyttävät metodit, sekä next-argumentin.

Metodissa oletetaan, että asiakas on lähettänyt pyynnössä x-auth-token nimisessä otsakkeessa yksilöllisen tunnisteen, jolla voidaan tarkistaa käyttäjän kirjautumisen tiedot. Jos tunnistetta ei löydy, palautetaan virhekoodi asiakkaalle. Jos tunniste löytyy, jatketaan tunnisteen validointia.

Luodaan req-objektiin uusi muuttuja user, jonka sisään tallennetaan jwt-verify metodin vastaus. Metodi tarkistaa onko yksilöllinen kirjautumistunniste validi. Jos on, se purkaa siitä salauksen ja palauttaa siitä löytyneet arvot, kirjautusmisajankohdan ja käyttäjän yksilöllisen id-tunnisteen.
Tämän jälkeen tarkistetaan aikaisemmin luodulla isTokenValid-metodilla (Kuva X), onko käyttäjän istunto vielä voimassa. Jos on, käytetään metodin saamaa next-argumenttia, joka kertoo ohjelmalle siirtyä seuraavaan metodiin, johon se on ketjutettu. Jos ei, käyttöliittymälle lähetetään virhekoodi, jossa kerrotaan autentikaatioon liittyvästä ongelmasta.

Jos joku metodeista keskeytyy try-palikan sisällä, lähetetään asiakkaalle suoraan catch-blokin sisällä sijaitseva virhekoodi.

```
router.post('/:username/games', auth, async (req, res) => {
    let user = await userRepository.findById(req.user._id);

    user = await userRepository.insertGame(user, req.body.gameId);
    return res.status(200).send(user.games);
});
```

Kuva 22. Pelin lisääminen käyttäjän listaan

Kuvassa 22 on toteutus päätepisteelle /users/{username}/games, jossa sisään kirjautunut käyttäjä lisää listaansa pelin. Vaikka osoitteeseen on sisällytetty osoitemuuttujassa käyttäjän käyttäjänimi, sitä ei käytetä käyttäjän tunnistamiseen. Router-luokan post-metodiin on sisällytetty päätepisteen osoitteen, pyyntö- ja vastausobjektien lisäksi auth-välikappale, joka lisättiin aikaisemmin riippuvuudeksi. Tämän metodin avulla tarkistetaan, onko käyttäjällä voimassa oleva istunto (Kuva 22).

Jos käyttäjän istunto on voimassa, otetaan auth-metodista palautetusta user-muuttujasta käyttäjän id ja haetaan sillä userRepository-luokan metodin avulla olemassa olevaa käyttäjää uuteen user-muuttujaan.

Kun olemassa oleva käyttäjä on löydetty, käytetään userRepository-luokan metodia. Sillä lisätään käyttäjän listaan peli antamalla sille argumenteiksi user-muuttuja ja asiakkaalta tulleen pyynnön mukana saapunut pelin id.

Tämän jälkeen asiakkaalle palautetaan lista tämän kaikista peleistä, onnistumisesta kertovan statuskoodin kanssa.

```
router.delete('/:username/games/:gameId', auth, async (req, res) => {
    const {gameId} = req.params;
    let user = await userRepository.findById(req.user._id);

    const updatedUser = await userRepository.removeGame(user, gameId);
    return res.status(200).send(updatedUser.games);
});
```

Kuva 23. Pelin poistaminen käyttäjän listasta

Päätepisteessä /users/{username}/games/{gameId} otetaan vastaan DELETE-pyyntöjä asiakkaalta, joilla halutaan poistaa tiettyjä, yksittäisiä pelejä käyttäjän listalta. Käyttäjän täytyy olla kirjautunut sisään ja tarkistaminen tapahtuu auth-metodissa, ennen kuin siirrytään käsittelemään pyyntöä kuvan 23 esiintyvän metodin sisään.
Kun autentikaatio on suoritettu onnistuneesti, tallennetaan osoitemuuttujasta pelin tunniste gameId-muuttujaan.

Tämän jälkeen haetaan olemassa oleva käyttäjä user-objektin avulla, joka saatiin autentikaation seurauksena. Kun käyttäjä löytyy, niin kutsutaan userRepository-luokasta metodia argumenteilla user ja gameId, jolla saadaan poistettua pyydetty peli käyttäjän listalta.

Tämän jälkeen käyttäjän lista peleistä palautetaan takaisin asiakkaalle onnistumisesta kertovan statuskoodin kanssa.

4.2.3 Kirjautuminen

Kirjautumispyynnölle luodaan uusi luokka auth.js.

```
const express = require('express');
const router = express.Router();
const _ = require('lodash');
const Joi = require('joi');
const userRepository = require('./../db/repositories/userRepository');
const auth = require('./../utils/auth');
```

Kuva 24. Riippuvuudet auth-luokassa

Luokalle annetaan uusina riippuvuuksina joi-kirjasto, joka on sama, jota käytettiin users-luokkaan liitetyn validate-luokan metodeissa, joissa validoitiin asiakkaalta lähetettyjä tietoja. (Kuva 24)

```
router.post('/', async (req, res) => {
    const {error} = validate(req.body);
    if (error) return res.status(400).send(error.details[0].message);

    let user = await userRepository.findByEmail(req.body.email);
    if(!user) return res.status(400).send('Invalid email or password');

    const validPassword = await auth.validatePassword(req.body.password, user.password);
    if(!validPassword) return res.status(400).send('Invalid email or password');

    const token = user.generateAuthToken();
    res.status(200).send(token);
});
```

Kuva 25. Käyttäjän kirjautuminen

Kuvassa 25 luodaan vastaanotto POST-pyynnölle päätepisteeseen /auth. Ensin asiakkaan lähettämä pyyntö käyttäjän lisäämiseksi validoidaan validate-metodilla, jossa on käytetty apuna joi-kirjastoa ja siitä palautettu vastaus tallennetaan error-nimiseen muuttujaan. Metodi käy läpi asiakkaan lähettämän JSON-viestin ja tarkistaa, että sen mukana on lähetetty käyttäjän sähköposti ja salasana. Jos tietoja puuttuu, asiakkaalle lähetetään ilmoitus pyynnön epäonnistumisesta tarkentavan virheviestin kanssa.

Jos ongelmia ei ilmennyt, haetaan userRepository-luokan metodin avulla tietokannasta käyttäjää, jonka sähköpostiosoite täsmää kyselyn mukana lähetettyä sähköpostiosoitetta.

Jos käyttäjää ei löydy, lähetetään virhekoodi asiakkaalle tarkentavan virheviestin kanssa. Kun taas käyttäjä löytyy, siirrytään tarkistamaan käyttäjän salasanaa.

Salasanan tarkistus tehdään kutsuen utils-kansiossa sijaitsevan auth-luokan validatePassword-metodia käyttäen. Jos salasana on virheellinen, lähetetään asiakkaalle virheestä ilmoittava statuskoodi ja viesti.

Jos salasana täsmää, jatketaan tokenin generointiin. Tokenin tarkoituksena on olla käyttöliittymän keino ylläpitää istunnon tilaa. Koska palvelimelle ei tule tallentaa käyttäjän tilan tietoja, asiakkaalle lähetetään yksilöivä token, jota asiakkaan tulee lähettää aina palvelimelle, kun halutaan ottaa yhteyttä päätepisteisiin, jossa vaaditaan tunnistautunutta käyttäjää. Kun token on luotu, lähetetään se asiakkaalle onnistumisesta kertovan statuskoodin kanssa.

5 Falcor

Falcor on Netflixin kehittämä, vuonna 2015 avoimeksi lähdekoodiksi julkaistu JavaScript-kirjasto rajapintojen rakentamiseen. Kirjasto on käytössä kaikissa Netflixin käyttöliittymissä. (Falcor 2015)

Koska Falcor-rajapintateknologiasta löytyy vain muutama artikkeli ja blogipostaus ja aihetta ei ole vielä tutkittu, on tämä tietoperusta perustettu Netflixin tuottamaan viralliseen tekniseen dokumentaatioon ja sivustolla esiintyvien teknisten asiantuntijoiden videoluentoihin.

Vielä vuonna 2011 Netflixillä oli käytössään REST-tyyppinen rajapinta. Ongelmana kuitenkin oli, että perinteinen rajapintamalli ei ollut suunniteltu ja optimoitu web-applikaatioille, jotka tarvitsevat pieniä osia useista eri resursseista. Perinteistä HTTP-protokollaa käyttäen on mahdollista kysyä vain yhtä resurssia pyynnöllä, joten applikaatioiden tuli ketjuttaa useita kyselyjä, jotta tarvittava data saatiin kerättyä. (Falcor 2015)

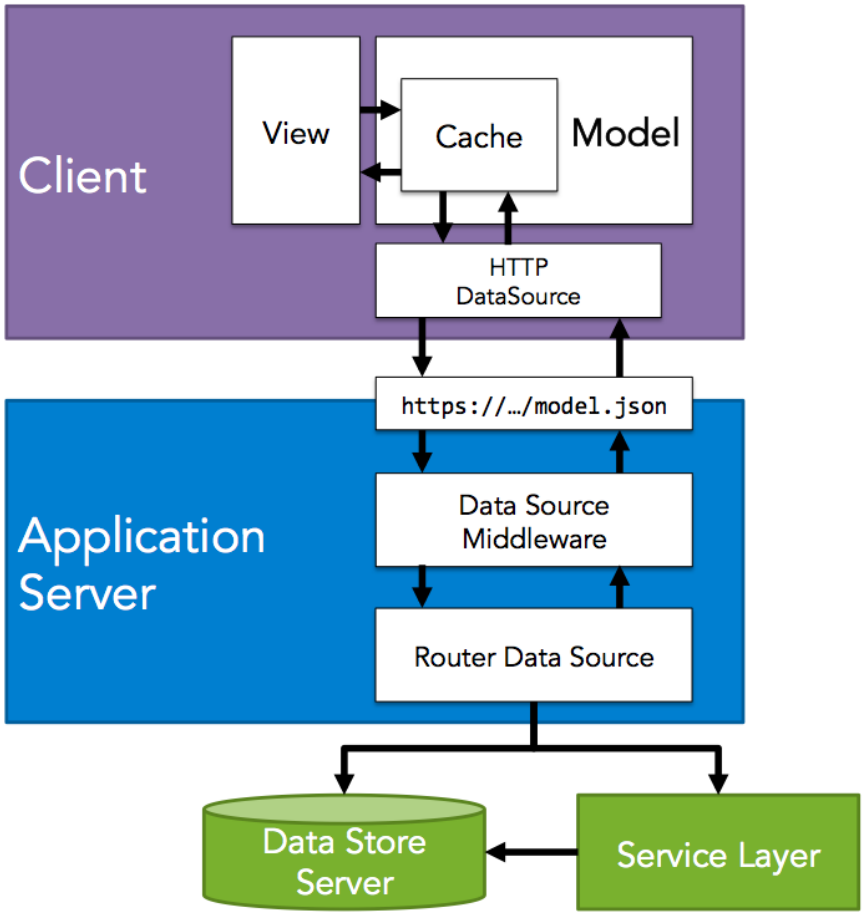
Toisen ongelman aiheutti JSON-objektin tapa näyttää dataa hierarkkisesti, vaikka data olisikin verkostoinutta. Toisin sanoen, datalla ei ole aina tiukkaa vanhempi-lapsi suhdetta ja riippuvuuksia saattaa olla useita, jolloin samaa dataa saattaa esiintyä rakenteissa useampaan kertaan. (Falcor 2015)

Kuva 26. JSON-objekti

Kuvassa on 26 näkyy tapa esittää dataa JSON-muodossa. Tässä tilanteessa molemmat kategoriat ovat samaa tyyppiä, joten merkintä esiintyy rakenteessa kahdesti, kummankin kategorian alla. Falcor mahdollistaa sen, että dataa on mahdollista käsitellä verkostona, ilman duplikaatteja ja silti esittää se normaalissa JSON-muodossa. (Falcor 2015)

Falcorin tarkoitus on myös tehdä datan pyytämisestä tehokkaampaa. Sen sijaan, että rajapinnasta pyydetään yhtä tai montaa kokonaista JSON-objektia, Falcor rakentaa yhden virtuaalisen JSON-mallin, joka tarjoaa vain sen datan, mitä pyydetään. Normaalisti jos rajapinnasta halutaan kategorian nimi, pyydetään koko JSON-objekti ja näytetään käyttöliittymässä siitä tarvittava osuus. Falcorin kanssa on taas mahdollista pyytää käyttöliittymään vain tarvittavat tiedot. (Falcor 2015)

Falcor mahdollistaa myös virtuaalisen JSON-mallin asettamisen välimuistiin. Kun dataa kysytään, Falcor konsultoi dataa ensin välimuistista ja jos sitä ei ole saatavilla, tekee se pyynnön rajapintaan vain siitä osasta, mikä puuttuu. (Falcor 2015)



Kuva 27. Falcor-arkkitehtuuri (Falcor 2015a)

Kuvassa 27 kuvataan Falcorin toimintaa applikaatiossa, jossa data sijaitsee erillisellä palvelimella. Asiakas, eli tässä tapauksessa käyttöliittymä, käyttää HTTP Datasource-rajapintaa pyytäessään JSON Graph-dataa ohjelmistopalvelimelta. Datan käsittelee ja muuntaa käyttöliittymässä sijaitseva Model-objekti, joka toimii välittäjänä käyttöliittymän näkymän ja DataSourcen välissä ja tallentaa vastaanottamaansa dataa välimuistiin, sekä muuntaa dataa JSON-muotoon. (Falcor 2015)

DataSource-rajapinnan tehtävänä on välittää JSON Graph-muotoista dataa Model-objektille. Router on taas implementaatio DataSource-rajapinnasta, joka sijaitsee lähes aina palvelimen päässä. Sen tehtävänä on osoittaa DataSource-rajapinnalta saapuvia pyyntöjä oikeisiin datalähteisiin. (Falcor 2015)

5.1 Model

Falcor esittelee Model-objektin, jota tulisi käyttää applikaatioiden käyttöliittymissä välikappaleina näkymän ja Datasource-rajapinnan välillä. Objekti käyttää DataSource-rajapintaa, joka hakee dataa erillisestä tietovarastosta JSON-graph muodossa. Model-objektin tehtävänä on muuntaa se JSON-muotoon, jotta se voidaan esittää käyttöliittymässä. (Falcor 2015)

Model-objekti tarjoaa myös kolme muuta tärkeää toimintoa, joista yksi on vastaanotetun datan tallentaminen välimuistiin latenssin kitkemiseksi. Jos samaa dataa kysytään uudestaan, Model-objekti tarkistaa ensin löytyykö pyydettävä data välimuistista. Jos välimuisti ei sisällä tarvittavaa tietoa, tehdään uusi pyyntö palvelimelle. (Falcor 2015)

5.1.1 Datan pyytäminen ja vastaanotto

Model-objektille implementoidaan DataSource-rajapinta asiakkaan puolelta, jolloin pystytään olemaan yhteydessä ulkoiseen datalähteeseen, esimerkiksi toisella palvelimella sijaitseviin JSON-objekteihin. (Falcor 2015)

```
var model = new falcor.Model({source: new falcor.HttpDataSource('/model.json')});
```

Kuva 28. Model-objektin alustus (Falcor 2015b)

Model-objekti alustetaan liittämällä se DataSource-rajapintaan, joka on yhteydessä ulkoiseen datalähteeseen kohteessa ’/model.json’ (Kuva 28).

```
var falcor = require('falcor');
var model = new falcor.Model({
    cache: {
        todos: [
            {
                name: 'get milk from corner store',
                done: false
            },
            {
                name: 'withdraw money from ATM',
                done: true
            }
        ]
    }});
```

Kuva 29. Model-objektin alustus välimuistilla (Falcor 2015b)

Jos Model-objektilla ei ole mahdollisuutta tai tarvetta olla yhteydessä Datasource-rajapintaan, voidaan myös alustaa sen välimuisti valmiilla JSON-objektilla (Kuva 29).

Dokumentaatioissa suositellaan aloittamaan työskentely Model-objektin kanssa täyttämällä sen välimuisti keksityllä datalla ja kehityksen edetessä korvata se DataSource-rajapinnalla, joka hakee datan palvelimelta. Kun objekti on alustettu, voit pyytää siltä arvoja antamalla sille osoitteita arvoihin (paths) jotka viittaavat välimuistissa tai palvelimella sijaitsevaan dataan. (Falcor 2015)

```
model.getValue('todos[0].name').then(function(value) {
    console.log(value);
});
```

Kuva 30. Datan hakeminen getValue-metodilla (Falcor 2015c)

Metodi model.getValue palauttaa lupauksen, johon päästään käsiksi .then() tai await metodeilla. Esimerkissä on käytetty then-metodia ja tulostettu vastaus selaimen konsoliin. Vastauksena tulostuu ensimmäinen todo-listan parametrin name arvo ”get milk from the corner store” (Kuva 30).

Jos kuvan data olisi haettu lokaalin data sijaan DataSource-rajapinnasta, olisi se tallennettu Model-objektin välimuistiin, missä nykyinen, keksitty data sijaitsee. Jos tämän jälkeen tehtäisiin sama datan hakupyyntö uudestaan, data olisikin haettu välimuistista, eikä palvelimelta. (Falcor 2015)

Model-objektilla ei kuitenkaan ole mahdollista pyytää vain objektia tai listaa. Model-objekti pakottaa datan pyytäjät, eli kehittäjät kertomaan tarkasti mitä arvoja tarvitaan ja kuinka paljon. Eli on mahdollista pyytää yhden tai useamman objektin sisältö, mutta Model-objektille on kerrottava tarkka määrä, sekä kaikki tarvittavat parametrit. (Falcor 2015)

Esimerkiksi kuvan 29 datasta ei voida pyytää kaikkia merkintöjä pyynnöllä ’todos’, vaan listan vastaanottamiseksi tarvitaan ”todos[{from: 0, to: 1}][’name’, ’done’]”. Model-objektille siis kerrotaan, että halutaan todo-objektit listan kohdasta 0 ja 1 ja niistä molemmista parametrit ’name’ ja ’done’, jolloin saadaan vastaukseksi koko todos-listan sisältö.

“Models force developers to be explicit about which value types they would like to retrieve in order to maximize the likelihood that server requests will have stable performance over time.” (Falcor 2015).

Dokumentaatiosta otetussa viittauksessa on tiivistettynä se, miksi datan pyytäminen on kirjaimellisen tarkkaa. Tarkoituksena on, että Falcoria käyttäen datan virtaus olisi mahdollisimman tasaista. Vaikka objektin tai listan koko olisi aluksi pieni, on kuitenkin mahdollisuus, että sisäkkäisten merkintöjen ja parametrien määrä kasvaa muutamista kappaleista kymmeniin tuhansiin. Näin haku, jossa aikaisemmin olisi kestänyt millisekunteja, saattaisi venähtää sekunteihin. Tarkka määrittely voi siis pakottaa kehittäjän rakentamaan esimerkiksi käyttöliittymälle sivutuksen datan määrän kasvaessa, joka lataa lisää dataa sitä mukaan, kun käyttäjä rullaa sivua alaspäin tai vaihtaa sivunumeroa. Näin käyttöliittymän suorituskyky pysyy optimaalisena.

5.1.2 Datan muokkaaminen Set-pyynnöillä

Model-objektilla voi myös datan pyytämisen lisäksi lähettää muokattuja arvoja DataSource-rajapintaan. Nämä Set-operaatiot kirjoittavat muutokset välimuistiin ja sen jälkeen DataSource-rajapintaan, joka lähettää muutokset palvelimelle. Vastauksena saadaan kaikki osoitteet arvoineen, joita muokataan. Jos pyyntö epäonnistuu, palautetaan vanhat osoitteet arvoineen. (Falcor 2015)

Arvojen muuttaminen Set-operaatiolla onnistuu kolmella eri tavalla.

```
var dataSource = new falcor.HttpDataSource("/model.json");
var model = new falcor.Model({
    source: dataSource
});

// prints "true"
model.setValue(["todos", 0,"done"], true).then(function(done){
    console.log(done);
});
```

Kuva 31. Datan asettaminen setValue-metodilla (Falcor 2015d)

Kuvassa 31 asetetaan arvo setValue-metodilla, jolla voidaan muokata yhtä arvoa kerrallaan. Tässä tapauksessa todos-listan ensimmäisen objektin done-parametrin arvo muutetaan epätodesta todeksi.

```
var dataSource = new falcor.HttpDataSource("/model.json");
var model = new falcor.Model({
    source: dataSource
});

model.
    set(falcor.pathValue(["todos", 0, "done"], true), falcor.pathValue(["todos", 1, "done"], true))
    then(function(response){
        console.log(JSON.stringify(response, null, 4));
    });
```

Kuva 32. Datan asettaminen set-metodilla (Falcor 2015d)

Kuvassa 32 set-metodilla muokataan useita arvoja muuttamalla ne yksittäisiksi pathValue-objekteiksi, joissa on arvoon osoittava reitti ja annettava arvo. Metodi falcor.pathValue([“todos”, 0, “done”]) näyttää tulostettuna seuraavanlaiselta: {path:["todos",0,"done"], value:true}. Metodi siis ainoastaan muodostaa objektin joka sisältää “path” ja “value” parametrit. Metodia ei ole pakko käyttää, vaan set-metodiin voidaan suoraan asettaa objekti, joka sisältää path ja value-arvo. Set-metodilla voidaan myös muuttaa yksittäisiä arvoja. (Falcor 2015)

```
var taskCompletionValuesJSONEnvelope = {
    json: {
        todos: {
            0: {
                done:true
            },
            1: {
                done:true
            }
        }
    }
};
```

Kuva 33. Data JSON Graph Envelopessa (Falcor 2015d)

Kuvassa 33 on esimerkki JSON Graph Envelope-objektista. Se sisältää JSON dataa joka on käärittyy objektin sisään.

```
model.
    set(taskCompletionValuesJSONEnvelope).
    then(function(response){
        console.log(JSON.stringify(response));
    });
```

Kuva 34. Datan asettaminen JSON Graph Envelope-objektilla set-metodissa. (Falcor 2015d)

Kuvassa 34 on kolmas tapa lähettää set-pyyntöjä palvelimelle. Tässä tapauksessa set-metodille annetaan argumenttina kuvassa 33 esitelty JSON Graph Envelope-tyyppinen objekti, jossa voidaan lähettää yksi tai useampi arvo muokattavaksi.

5.1.3 Datan muokkaaminen Call-pyynnöillä

Arvojen hakeminen ja muokkaaminen get- ja set-pyynnöillä ovat esimerkkejä idempotenttisista funktioista. Tällä tarkoitetaan sitä, että suoritettava operaatio tuottaa saman tuloksen, vaikka se suoritetaan yhden tai useamman kerran, eikä aiheuta ylimääräisiä sivuvaikutuksia. Tämän takia Model-objekti kykenee turvallisesti tekemään useita peräkkäisiä pyyntöjä samaan arvoon suoraan välimuistista, ilman että sen tarvitsee palata hakemaan tietoa DataSource-rajapinnasta. (Falcor 2015)

Jos käyttäjä haluaa tehdä operaatioita, jotka eivät ole idempotenttisia, kuten datan lisääminen ja poistaminen, tulee käyttää call-metodia. Kun pyyntö suoritetaan, model-objekti lähettää sen suoraan DataSource-rajapinnalle, eikä kierrätä sitä välimuistin kautta. Call-metodi ottaa neljä argumenttia, joista ensimmäinen on callPath. Tämä osoittaa palvelimella sijaitsevaan funktioon, jota kutsutaan. Toisena vastaanotetaan argumenttilista, joka annetaan kutsuttavalle funktiolle. Kolmantena, vapaaehtoisena argumenttina on refPath, jota käytetään usein silloin, kun kutsuttu funktio luo uuden objektin ja palauttaa siihen viittaukset. Neljäs vapaaehtoinen argumentti on thisPath, joka palauttaa kaikki muutokset, jotka on tehty this-objektiin. (Falcor 2015)

```
model.
    call(
        ["todos", "add"],
        ["pick up some eggs"],
        [["name"], ["done"]]).
    then(
        function(jsonEnvelope) {
            console.log(JSON.stringify(jsonEnvelope));
        },
        function(error) {
            console.error(error);
        });
```

Kuva 35. Datan asettaminen call-pyynnöllä (Falcor 2015e)

Kuvassa 35 näkyy esimerkki call-pyynnöstä. Kutsun ensimmäinen arvo [“todos”, “add”] on callPath-argumentti, joka kertoo, mihin routeen pyyntö suunnataan. Toisena on argumenttilista, jonka arvona on “pick up some eggs”. Kolmantena on refPath-argumentti [“name”, “done”] joka kertoo mitä palautetaan takaisin vastauksessa. Lopputuloksena palautetaan uusi todos-objekti, jonka name-parametrina on “pick up some eggs” ja sen done-parametrin arvo, joka asetetaan luultavasti palvelimella automaattisesti epätosiksi.

5.1.4 Batching

Falcor pyrkii tekemään datan pyytämisestä tehokkaampaa esittelemällä tavan niputtaa useita pyyntöjä ja lähettää ne kerralla DataSource-rajapintaan.

```
var log = console.log.bind(console);
var httpDataSource = new falcor.HttpDataSource("/model.json");
var model = new falcor.Model({ source: httpDataSource });
var batchModel = model.batch();

batchModel.getValue("todos[0].name").then(log);
batchModel.getValue("todos[1].name").then(log);
batchModel.getValue("todos[2].name").then(log);
```

Kuva 36. Batch-metodi (Falcor 2015f)

Kuvassa 36 on kolme erillistä get-kutsua, joissa pyydetään kolmesta eri todos-objektista nimi-parametria. Normaalisti nämä lähetettäisiin yksittäisinä kutsuina DataSource-rajapinnalle, mutta rivillä neljä on luotu batchModel-objekti, jonka avulla kaikki kolme get-pyyntöä yhdistetään yhdeksi get-pyynnöksi. Luotu batchModel-objekti yksinkertaistaa lähtevän pyynnön seuraavanlaiseen muotoon:

httpDataSource.get([["todos", { from: 0, to: 2 }, "name"]]);

5.2 Paths

Falcor käyttää polkuja (path), joiden avulla model-objektille kerrotaan mitä arvoja haetaan tai muutetaan. Polku on peräkkäinen sarja avaimia, joka evaluoi JSON-objektin sen juuresta lähtien ja osoittaa tiettyyn kohtaan JSON-objektissa. (Falcor 2015)

```
var falcor = require('falcor');
var model = new falcor.Model({
  cache: {
    todos: [
      {
        name: 'get milk from corner store',
        done: false
      }
    ]
  }
});

// prints 'get milk from corner store'
var name = await model.getValue("todos[0].name");
```

Kuva 37. Polku (Falcor 2015g)

Kuvassa 37 on esimerkki polun käytöstä. Model-objektille on annettu polku “todos[0].name”. Evaluointi aloitetaan JSON-objektin juuresta, eli cache-parametrin jälkeen tulevista aaltosulkeista, josta JSON alkaa. Sen jälkeen polkua seurataan systemaattisesti

alaspäin, kunnes se saavuttaa annetun kohdan. Tässä tapauksessa polku osoittaa todos-listan ensimmäisen objektin parametriin “name”.

Model-objekti pystyy vastaanottamaan polkuja kahdella eri tavalla, avainlistana ja syntaksi-merkkijonona. (Falcor 2015)

5.2.1 Polut syntaksi-merkkijonoina

Model-objektit tukevat JavaScript-tyylisiä polkuja, joita kutsutaan tässä kontekstissa syntaksi-merkkijonoiksi (path syntax strings). Nämä syntaksi-merkkijonot jäsennetään heti avainlista-poluiksi, joka kustantaa resursseja.

```
"todos[0].name" -> ["todos", 0, "name"]
```

Kuva 38. Polku syntaksi-merkkijonona (Falcor 2015g)

Kuvassa 38 näkyy esimerkkinä syntaksi-merkkijono ja kuinka se käännetään avainlistan muotoon.

5.2.2 Polut avainlistoina

Syntaksi-merkkijonoa tehokkaampi tapa on esittää polku avainlistan muodossa. Tämä johtuu siitä, että kaikki polut jotka esitetään syntaksi-merkkijonoina käännetään kuitenkin avainlistan muotoon. Model-objektit tuottavat aina polut avainlistan muodossa. (Falcor 2015)

Polku voidaan esittää listana 0...n avaimia. Seuraavat tietotyypit lasketaan valideiksi avaintyypeiksi:

- Merkkijonot
- Booleanit (true, false eli tosi tai epätosi)
- Numerot
- Null (havainnollistaa puuttuvaa arvoa)

```
["todos", 0, "name"]
["todos", 5, true]
["todos", 9, null]
```

Kuva 39. Polkujen avaintyypit (Falcor 2015g)

Kuvassa 39 demonstroidaan kolmea erilaista avainpolkua. Ero syntaksi-merkkijonoon näkyy listan rakenteessa, joka ei ole muodoltaan pelkkä merkkijono, vaan siinä esiintyy eri datatyyppejä.

5.3 JSON Graph

Falcor esittelee JSON Graph konvention, jolla esitetään verkkomaista dataa JSON-objektin muodossa. Falcoria käyttävät applikaatiot esittävät kaiken datansa yhdessä JSON Graph objektissa.

JSON Graph on validia JSON:ia, joten mikä tahansa JSON-parsija pystyy käsittelemään sitä. Se eroaa kuitenkin perinteisestä mallista esittelemällä uusia primitiiviä datatyyppejä, jonka avulla JSON-muodossa voidaan esittää verkkomaista dataa helposti ja yhdenmukaisesti.

JSON Graph dataa on mahdollista manipuloida sellaisenaan, mutta sitä suositellaan käytettävän työkalujen kanssa jotka tukevat formaatin kanssa työskentelyä. Luontevinta on ajatella Falcoria pakettina, joka koostuu erilaisista protokollista ja työkaluista JSON Graph datan siirtoon, tallentamiseen ja hakemiseen.

JSON Graph-formaatin tarkoituksena on ratkaista JSON-formaatin ongelmat, joita ovat duplikaatti- ja vanhentuneen datan esiintyminen. Tämän Falcor tekee esittelemällä JSON-formaatin primitiivisten tyyppien lisäksi kolme uutta tyyppiä.

5.3.1 Reference

Reference on yksi Falcorin esittelemistä, uusista primitiivisistä tyypeistä. Se on JSON-objekti joka sisältää $type-avaimen jonka arvona on ref, viitaten sanaan reference. Toisena avaimena on path, jonka arvoksi annetaan polku.

```
{
    todosById: {
        "44": {
            name: "get milk from corner store",
            done: false,
            prerequisites: [{ $type: "ref", value: ["todosById", 54] }]
        },
        "54": {
            name: "withdraw money from ATM",
            done: false,
            prerequisites: []
        }
    },
    todos: [
        { $type: "ref", value: ["todosById", 44] },
        { $type: "ref", value: ["todosById", 54] }
    ]
};
```

Kuva 40. Reference-tyyppi (Falcor 2015h)

Refence-tyypin objekti osoittaa aina toisen kohteen JSON-objektiin. Kuvassa 40 näkyy kahdesta eri kohteesta dataa JSON-graph objektissa. Siinä todos-listan sisällä on kaksi reference-objektia, jotka viittaavat todosById-kohteessa sijaitseviin JSON-objekteihin.

Reference on kuin linkki toiseen kohteeseen, jonka avulla vältytään duplikaattidatalta ja pystytään luomaan verkkomaista dataa. (Falcor 2015)

Jos käyttäjä kutsuu model-objektin avulla data polulla “todos[0]” käyttöliittymään ei tulostukkaan {$type: “ref”, value: [“todosById”, 44]}, vaan Falcor osaa käyttää reference-objektin viittausta ja tulostaa objektin todosById-kohteesta {name: “get mil from corner store”, done: false, prerequisites: [{$type: “ref”, value: [“todosById”, 54]}]}. (Falcor 2015)

5.3.2 Atom

Atom on toinen uusista primitiivisistä tyypeistä, joka sisältää avain-arvo parin $type: ”atom” ja avaimen path, joka sisältää JSON-arvon. Tämän avulla voidaan kiertää rajoitus, jonka mukaan Falcorilla ei voi hakea kokonaisia objekteja tai listoja. Atom-objekteja ei kuitenkaan voi muokata. Jos siihen halutaan muutoksia, täytyy se korvata uudella Atom-objektilla. (Falcor 2015)

```
var falcor = require('falcor');
var model = new falcor.Model({cache: {
    titlesById: {
        "44": {
            name: "Die Hard",
            subtitles: { $type: "atom", value: ['en', 'fr'] }
        }
    }
}});

var atom = await model.getValue(['titlesById', 44, 'subtitles']);

// returns:
// ['en', 'fr']
```

Kuva 41. Atom-tyyppi (Falcor 2015h)

Kuvassa 41 on esimerkki siitä, miltä primitiivinen atom-tyyppi joka sisältää listan näyttää, kun sitä kutsutaan model-objektilla. Vaikka kyseessä on lista, model-objektille ei tarvitse tarkentaa listan sisältämien tietojen paikkaa, vaan pelkän avaimen arvon antaminen riittää tulostamaan listan kaikki tiedot.

5.3.3 Error

Error Falcorin kolmas lisäys primitiivisiin tyyppeihin. Siitä löytyy $type: ”error” avain-arvopari ja value-avain, joka sisältää tiedot tapahtuneesta virheestä. (Falcor 2015)

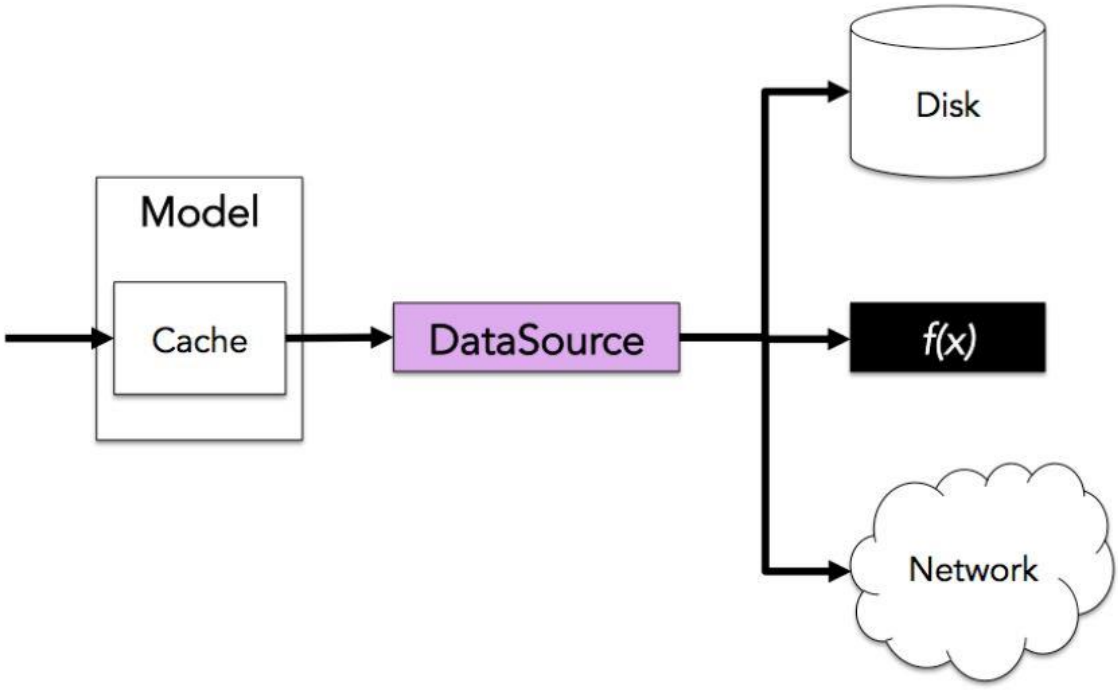
```
var router = new Router([
    {
        route: "user.name",
        get: function(pathSet) {
            return userService.getUser().
                then(
                    function(user) {
                        return { path: ['user', 'name'], value: user.name };
                    },
                    function(error) {
                        return { path: ['user'], value: { $type: "error", value: error.message } };
                    });
        }
    }
]);
```

Kuva 42. Error-tyyppi (Falcor 2015h)

Error-objekti täsmennetään siellä, mistä data noudetaan, eli usein router-objektissa. Kuvassa 42 on esimerkki siitä, miten error-objektia on käytetty. Jos tietokantahaku keskeytyy, asetetaan sieltä vastaanotettu virheviesti value-avaimen arvoksi.

5.4 Data Source

DataSource on rajapinta, joka vastaanottaa ja lähettää JSON Graph dataa model-objektille. Jokainen DataSource-rajapinta käsittelee vaih yhtä JSON Graph-mallia. (Falcor 2015)



Kuva 43. Datasource-rajapinta (Falcor 2015i)

DataSource-rajapinnat toteuttavat kolmea eri JSON Graph-operaatiota, joita ovat get-, set ja call. Model-objekti suorittaa näitä operaatioita saadakseen arvoja DataSource-rajapinnasta. DataSource-rajapinta pystyy taas vastaanottamaan JSON Graph-dataa mistä vain, oli se laitteen muistista tai ulkoiselta palvelimelta (Kuva 43).

DataSource-rajapinta pystyy vastaanottamaan polkuja ainoastaan avainlistan muodossa. Jos dataa halutaan hakea merkkijono-syntaksilla, pitää välissä käyttää model-objektia. (Falcor 2015)

Demostroidaan DataSource-rajapinnan toimintaa get-metodilla muuttamalla model-objekti DataSource-rajapinnaksi. Tämä onnistuu suorittamalla sen yhteydessä asDataSource-metodin. (Falcor 2015)

```
var response = await dataSource.get([
    ["todos", 0, ["name", "done"]],
    ["todos", 0, "prerequisites", { from: 0, to: 1 }, ["name", "done"]]
]);

// eventually returns...
// {
//     jsonGraph: {
//         todos: {
//             0: { $type: "ref", value: ["todosById", 44] },
//         },
//         todosById: {
//             44: {
//                 name: 'get milk from corner store',
//                 done: false,
//                 prerequisites: [
//                     { $type: "ref", value: ['todosById', 54] },
//                     { $type: "ref", value: ['todosById', 97] }
//                 ]
//             },
//             "54": { name: 'withdraw money from ATM', done: false },
//             "97": { name: 'pick car up from shop', done: false }
//         }
//     }
// }
```

Kuva 44. Datan haku DataSource-rajapinnasta (Falcor 2015j)

Kuvassa 44 kutsutaan dataa suoraan DataSource-rajapinnasta. Kommentoituna harmaalla tekstillä näkyy vastaus, jonka DataSource-rajapinta palauttaa. Kun välissä ei ole model-objektia suorittamassa käännöstä JSON-muotoon, tulee data raakana JSON Graph muodossa.

Jos DataSource-rajapintaa halutaan käyttää suoraan set() ja call() metodien kanssa, täytyy data antaa JSON Graph-muodossa, toisin kuin model-objektissa, jolle voidaan syöttää JSON-muotoista dataa. (Falcor 2015)

5.5 Router

Router on DataSource-rajapinnan implementaatio, joka sijaitsee lähes aina palvelimen päässä. Sen tehtävänä on osoittaa kyselyt oikeaan datalähteeseen, eli JSON Graph-objektiin polkujen perusteella. (Falcor 2015)

Router palvelee samaa tarkoitusta kuin REST-rajapinnan päätepisteet. Asiakas hakee dataa vain tarvittaessa, mahdollistaen ohjelmistopalvelimen toimimisen tilattomana. Rou-

terissa on kuitenkin eroja, joista isoimman erot koskevat resursseihin osoittavia osoitteita ja tapaan noutaa dataa. (Falcor 2015)

Kun REST-rajapinnassa käytetään URI-osoitteita, Falcor toimii JSON-polkujen avulla. Falcor Routerin yksittäinen Route kykenee myös vastaanottamaan erilaisia polkuja, kun taas REST-rajapinnan päätepiste toimii vain yhdellä osoitteella. (Falcor 2015)

Routerin avulla vältytään myös edestakaisilta matkoilta asiakkaalta palvelimelle. Jos REST-rajapinnasta halutaan vastaanottaa kolme erilaista todos-listan objektia id:n avulla, siihen tarvittaisiin kolme GET-pyyntöä osoitteeseen /todos/{id}. Routerilla sama lopputulos saadaan yhdellä pyynnöllä. (Falcor 2015)

5.5.1 Route-objekti

Yhdessä Router-luokassa voi olla useita eri Route-objekteja. Kun DataSource-rajapinnasta kutsutaan Router-luokan instanssia, se tarkastelee mihin sisältämäänsä Route-objektiin pyynnön polku täsmää. Kun vastaava Route-objekti löytyy, sen sisältämä funktio suoritetaan. (Falcor 2015)

5.5.2 Route-käsittelijä

Route-käsittelijä (Route Handler) on vastuussa PathValue:n luomisesta jokaisesta polusta, johon se täsmää get-, set- tai call-operaatioissa. PathValue on objekti, joka sisältää polku ja arvo avainparit. (Falcor 2015)

```
{
    route: 'user.["name", "surname"]',
    get: function(pathSet) {
        // pathSet is ["user", ["name"]] or ["user", ["surname"]] or ["user", ["name", "surname"]]
        if (this.userId == null) {
            throw new Error("not authorized");
        }
        return userService.
            get(this.userId).
            then(function(user) {
                // pathSet[1] is ["name"] or ["surname"] or ["name", "surname"]
                return pathSet[1].map(function(key) {
                    return { path: ["user", key], value: user[key] };
                });
            });
    }
}
```

Kuva 45. Get-tyyppinen route-objekti (Falcor 2015k)

Kuvassa 45 on esimerkki yhdestä Route-objektin get-tyyppisestä käsittelijästä. Kun Model-objekti tekee get-kutsun DataSource-rajapinnasta polulla [“user”, [“name”, “surname”]], täsmää sen polku kuvan Route-objektiin. Tämän jälkeen suoritetaan Route-objektin käsittelijän sisältämä funktio, joka vastaa pyynnön operaatiota. Route-objekti voi sisältää kolme erilaista käsittelijää, eli get-, set- ja call-operaatiot. Käsittelijän sisältämä funktio suoritetaan ja model-objektille palautetaan yksi tai useampi PathValue. (Falcor 2015)

```
[
    { path: ["user","name"], value: "Anupa" },
    { path: ["user","surname"], value: "Husain" }
]
```

Kuva 46. Route-objektista palautettu vastaus (Falcor 2015k)

Kuvassa 46 on esimerkki siitä, mitä kuvan 45 Route-käsittelijästä palautetaan. Kaksi PathValue-objektia, jotka sisältävät käyttäjän nimen ja sukunimen, sekä niihin osoittavat polut. Kun kaikki tarvittavat route-objektit on käyty läpi, Router-instanssi palauttaa JSON-Graph objektin, joka sisältää kaikista eri poluista palautetut arvot.

5.5.3 Route pattern

Route pattern on syntaksi, jonka avulla voi sisällyttää useita erilaisia arvoja yhden polun sisään. Tämä mahdollistaa Falcorin tarjoaman ominaisuuden, jossa yhteen ”päätepisteeseen” eli Routeen voi lähettää pyyntöjä eri polkurakenteilla. (Falcor 2015)

Route pattern voi sisältää kolmea erilaista tyyppiä, jotka ovat integers, keys ja ranges. Nimensä mukaan integers voi sisältää kokonaislukuja, keys merkkijonoja ja ranges sisältää tietoa arvojoukosta kokonaislukujen muodossa. (Falcor 2015)

```
titlesById[{integers}].name
```

Kuva 47. Integers-tyyppinen route pattern (Falcor 2015k)

Kuvassa 47 on esimerkki integers-tyyppisestä route-patternista. Se vastaa siis kaikkia polkuja, joiden polussa esiintyy ”titlesById”-arvo, 1..n kokonaislukua ja ”name”.

```
["titlesById", [234,223,555,111,112,113],"name"]
```

Kuva 48. Polku sisältäen listan integer-arvoja (Falcor 2015k)

Kuvassa 48 on esimerkki polusta, joka täsmää kuvan 47 route-patternin kanssa. Jos kyseessä olisi keys-tyyppinen route-pattern, polussa saisi olla.n merkkiä.

```
['genreList', [0, 1, 5, 6, 7, 9], 'name']
```

Kuva 49. Polku sisältäen listan ranges-arvoja (Falcor 2015k)

Kuvassa 49 on esimerkki polusta, joka täsmää ranges-tyyppisen route-patternin kanssa. Polku näyttää samalta kuin integers-tyyppisen route patternin kanssa täsmäävä polku, mutta Falcor muuttaa pyynnön ulkomuotoa, kun se lähetetään palvelimelle.

```
["genreList", [{from:0,to:1}, {from:5,to:7}, {from:9,to:9}], "name"]
```

Kuva 50. Palvelimelle lähetetty polku ranges-arvoilla (Falcor 2015k)

Kuvassa 50 näkyy kuvan 49 polku sellaisena, miten se lähetetään route-käsittelijälle. Luku n ja siitä seuraava luku muodostavat yhden arvojoukon. Jos lukuja on pariton määrä, jonon viimeinen luku muodostaa yksin arvojoukon.

6 Falcor-rajapinnan rakentaminen

6.1 Rajapinnan suunnittelu

Falcor-rajapinnan kehittäminen alkaa rakenteen suunnittelulla. Vaatimukset ovat samat kuin REST-rajapintaa rakentaessa, joten niitä ei tarvitse määrittää erikseen.

Helpointa olisi, jos kaikki rajapintakutsut saataisiin sisällytettyä yhteen Router-luokkaan ja sitä mukaa yhteen JSON Graph-objektiin. Tämä ei kuitenkaan onnistu, sillä osa datasta on autentikaation takana ja osa ei. Autentikaatio toteutetaan ennen Router-luokkaan pääsyä, joten luokkia on oltava ainakin kaksi kappaletta. Tämä tarkoittaa kahta JSON Graph-objektia. Koska verkkomaisen datan mahdollistavan ref-tyypin edellytys on, että data löytyy samasta JSON Graph objektista, täytyy rajapinta suunnitella niin, että referenssejä hyödyntävät kutsut asetetaan samaan Router-luokkaan sen datan kanssa, johon niillä on yhteys.

6.2 Router-luokat

Aloitetaan Router-luokan suunnittelusta, joka ei vaadi autentikaatiota. Siihen voidaan siis laittaa kaikki Route-objektit, joihin voidaan päästää kirjautumaton käyttäjä.

Taulukko 9. Router-luokan suunnittelu ilman autentikaatiota

Router-luokka ilman autentikaatiota		
Toiminto	Polku	Selite
Get	gamesById[{keys:ids}][{keys:fields}]	Haetaan yksi tai useampi peli id:n avulla. Haussa kerrotaan myös mitä parametrejä pyynnön mukana vastaanotetaan.
Get	games[{integers:pagination}]	Haetaan pelit sivunumeron ja sivulla esiintyvien pelien määrän perusteella. Viittaa gamesById-routeen.
Get	games[{keys:username}][{integers:indices}]	Haetaan yhden käyttäjän tallentamat pelit käyttäjänimen perusteella. Kerrotaan myös näytettävien pelien määrä. Viittaa gamesById-routeen.
Call	register['email', 'username', 'password']	Luodaan uusi käyttäjä. Kerro-

		taan sähköposti, käyttäjänimi ja asetettava salasana.
Call	login['email', 'password']	Kirjataan käyttäjä sisään. Kerrotaan sähköposti ja salasana.

Taulukko 10. Router-luokan suunnittelu autentikaatiolla

Router-luokka autentikaatiolla		
Toiminto	Polku	Selite
Call	user.games.remove	Poistetaan käyttäjältä peli
Call	user.games.insert	Lisätään käyttäjälle peli

6.3 Router-luokkien luonti

6.3.1 Router-luokka ilman autentikaatiota

Luodaan games.js luokka ja lisätään sinne tarvittavat riippuvuudet.

```
const gameRepository = require('./../db/repositories/gameRepository');
const userRepository = require('./../db/repositories/userRepository');

const Router = require('falcor-router');
const falcor = require('falcor');
const $ref = falcor.Model.ref;
```

Kuva 51. Riippuvuudet games-luokassa

Kuvassa 51 esitellään kaikki luokkaan lisätyt riippuvuudet. Vaikka Falcor luetaan yhdeksi kirjastoksi, mutta se on kuitenkin jaettu paketinhallinnassa kolmeen eri osaan. Projektiin on ladattu ne kaikki ja tässä luokassa tarvitaan niistä kahta Router-luokan ja Route-objektien luomiseen. Näitä ovat falcor-router ja falcor. Luokassa on myös alustettu $ref-muuttuja falcor-kirjastosta, joka on Falcorin esittelemä uusi primitiivinen datatyyppi. Muuttujaa käytetään reference-objektien luomisessa.

Sisäisiä riippuvuuksia ovat gameRepository ja userRepository. Tietokantayhteyksistä huolehtivat luokat ovat toiminnoiltaan täysin samanlaisia kuin rakennetussa REST-rajapinnassa.

```
module.exports = Router.createClass([

]);
```

Kuva 52. Router-luokan luonti

Luodaan falcor-router pakettia käyttäen Router-luokka (Kuva 52). Luokan sisään asetetaan tarvittavat Route-objektit.

```
{
    route: "gamesById[{keys:ids}][{keys:fields}]",
    get: function (pathSet) {
        const results = [];
        return gameRepository.findManyById(pathSet.ids).then(games => {
            if(games === null) return {path: ['gamesById'], value: "Games not found"};
            games.forEach(game => {
                let gameRecord = game;

                if (gameRecord === null) {
                    results.push({
                        path: ['gamesById', String(gameRecord['_id'])],
                        value: "Game not found"
                    });
                } else {
                        pathSet.fields.forEach(key => {
                        if (gameRecord[key] === undefined) {
                            results.push({
                                path: ['gamesById', String(gameRecord['_id']), key],
                                value: null
                            });
                        } else {
                            results.push({
                                path: ['gamesById', String(gameRecord['_id']), key],
                                value: String(gameRecord[key])
                            });
                        }
                    });
                }
            });
            return results;
        });
    }
}
```

Kuva 53. Pelin/pelien hakeminen id:n ja parametrien avulla

Kuvassa 53 näkyy ensimmäinen route-objekti, joka on luotu polulla gamesById[{keys:ids}][{keys:fields}]. Route-objektissa on get-tyyppinen käsittelijä, joka vastaanottaa polkua täsmäävät get-kutsut ja suorittaa sisältämänsä funktion.

Ensin gameRepository-luokasta haetaan kaikki pelit saatujen id-arvojen avulla. Se on tehty käyttämällä Falcorin pathSet parametria, jolla päästään käsiksi route-objektiin saapuneeseen polkuun. Polusta irrotetaan ids-lista, joka laitetaan suoraan tietokantaluokan metodin argumentiksi.

Seuraavaksi tietokantakyselyn tulokset käydään läpi. Jos tuloksia ei ole, palautetaan DataSource-rajapinnalle PathValue-objekti virheviestin kanssa. Jos lista ei ole tyhjä, käydään tulokset läpi yksitellen. Ensin tarkistetaan, onko listan yksittäinen peli olemassa. Jos ei ole, luodaan PathValue-objekti, joka sisältää virheviestin ja lisää sen aiemmin luotuun results-listaan. Jos peli on olemassa, luodaan PathValue-objekti jokaisesta pelin parametristä, jota asiakas on pyytänyt polun fields-listassa ja lisätään ne results-listaan. Tätä toistetaan, kunnes kaikki tietokannasta löydetyt pelit on käyty läpi. Lopulta lista palautetaan DataSource-rajapinnalle.

```
{
    route: "games[{integers:pagination}]",
    get: function (pathSet) {

        let page = pathSet.pagination[0];
        let size = pathSet.pagination.length;

        const results = [];

        return gameRepository.get(page, size).then(games => {
            if(games === null) return {path: ['games'], value: "Games not found"};
            games.forEach((game, i) => {
                let gameRecord = game;
                results.push({
                    path: ['games', pathSet.pagination[i]],
                    value: $ref(["gamesById", String(gameRecord._id)])
                });
            });
            return results;
        });
    }
}
```

Kuva 54. Pelien hakeminen sivutuksen avulla

Kuvassa 54 on luotu Route-objekti get-tyyppisellä käsittelijällä, joka hakee pelejä sivutuksen avulla. Se sisältää get-tyyppisen Route-käsittelijän, jonka funktiossa irrotetaan ensin pathSet parametrista sivutukset tiedot erillisiin muuttujiin.

Tämän jälkeen muuttujien avulla haetaan gameRepository-luokasta pelit. Jos tuloksia ei ole, palautetaan virheestä kertova PathValue-objekti. Jos tuloksia on, käydään ne läpi ja luodaan jokaisesta pelistä pathValue-objekti, joka sisältää viittauksen kyseisen pelin id:llä gamesById-routeen. Lopulta lista palautetaan DataSource-rajapinnalle.

```
{
    route: "games[{keys:username}][{integers:indices}]",
    get: async function (pathSet) {
        const results = [];

        return await userRepository.findByUsername(pathSet.username).then(user => {
            if (user === null || user.games.length === 0) {
                console.log(pathSet);
                return ({
                    path: ['user'],
                    value: "No results"
                });
            } else {
                user.games.forEach((game, i) => {
                    results.push({
                        path: ['games', pathSet.username[0], i],
                        value: $ref(["gamesById", String(game)])
                    });
                });
                return results;
            }
        })
    }
}
```

Kuva 55. Käyttäjän lisäämien pelien hakeminen

Kuvan 55 Route-objekti sisältää get-tyyppisen route-käsittelijän, joka hakee tietyn käyttäjän pelit käyttäjänimen ja indeksien avulla.

Ensin userRepository-luokasta haetaan käyttäjä irrottamalla pathSet muuttujasta käyttäjänimi. Jos käyttäjää ei löydy, palautetaan käyttöliittymälle PathValue-objekti joka kertoo virheestä. Jos käyttäjä löytyy, käydään läpi tämän pelit ja luodaan jokaisesta PathValue-objekti, joka sisältää viittauksen kyseisen pelin id:llä gamesById-routeen. Jokainen PathValue-objekti lisätään listaan ja palautetaan DataSource-rajapinnalle.

```
{
    route: "register",
    call: async function (callPath, args) {
        const email = args[0];
        const username = args[1];
        const password = args[2];

        let user = await userRepository.findByUsername(username);
        if(user) return {
            path: ['register'],
            value: 'username taken'
        };

        user = await userRepository.findByEmail(email);
        if(user) return {
            path: ['register'],
            value: 'email already registered'
        };

        const hashedPassword = await auth.hashPassword(password);

        const newUser = {
            username: username,
            email: email,
            password: hashedPassword
        };

        await userRepository.create(newUser);

        return {
            path: ['register'],
            value: 'registration successful'
        };
    }
}
```

Kuva 56. Käyttäjän rekisteröityminen

Kuvassa 56 route-objekti sisältää call-tyyppisen route-käsittelijän, jonka tehtävänä on vastaanottaa polun lisäksi pyynnön mukana lähetetyt argumentit. Argumentit irrotetaan muuttujiin email, username ja password (sähköposti, käyttäjätunnus ja salasana). Tämän jälkeen kannasta tarkistetaan, löytyykö sieltä olemassa olevaa käyttäjää samalla käyttäjätunnuksella tai sähköpostilla. Jos löytyy, palautetaan käyttäjälle pathValue-objekti virheviestin kanssa.

Jos käyttäjätunnus ja sähköposti ovat uniikkeja, suojataan käyttäjän salasana ja luodaan uusi käyttäjä-objekti, joka tallennetaan kantaan. Lopuksi lähetetään pathValue-objekti, jossa kerrotaan operaation onnistumisesta.

```
{
    route: "login",
    call: async function (callPath, args) {
        const email = args[0];
        const password = args[1];

        let user = await userRepository.findByEmail(email);
        if(!user) {
            return {
                path: ['login'],
                value: 'login unsuccessful'
            };
        }

        const validPassword = await auth.validatePassword(password, user.password);
        console.log(validPassword);
        if(!validPassword) {
            return {
                path: ['login'],
                value: 'login unsuccessful'
            };
        }

        const token = user.generateAuthToken();
        return {
            path: ['login'],
            value: token
        };
    }
}
```

Kuva 57. Käyttäjän kirjautuminen

Kuvassa 57 on call-tyyppinen route-objekti, jossa käsitellään käyttäjän sisäänkirjautuminen. Vastaanotetusta argumentista irrotetaan asiakkaalta lähetetty sähköposti ja salasana. Tämän jälkeen haetaan käyttäjä sähköpostin perusteella. Jos käyttäjää ei löydy, palautetaan virheviesti asiakkaalle.

Jos käytttäjä löytyy, validoidaan lähetetty salasana. Jos salasana ei täsmää, lähetetään virheviesti. Jos salasana täsmää, generoidaan käyttäjälle yksilöllinen token-tunniste ja lähetetään se asiakkaalle.

6.3.2 Router-luokka autentikaatiolla

Luodaan users.js luokka ja lisätään sinne tarvittavat riippuvuudet.

```
const userRepository = require('./../db/repositories/userRepository');
const Router = require('falcor-router');
```

Kuva 58. Riippuvuudet users-luokassa

Kuvassa 58 näkyy luokkaan lisätyt riippuvuudet, joista ensimmäinen on sisäinen riippuvuus userRepository ja toinen on ulkoinen riippvuus falcor-router.

```
let UserRouterBase = Router.createClass([
    // routes here
]);

let UserRouter = function (req, res) {
    UserRouterBase.call(this);
    //used later to pass token for this component
    this.req = req;
    this.res = res;
};

UserRouter.prototype = Object.create(UserRouterBase.prototype);

module.exports = function (req, res) {
    return new UserRouter(req, res);
};
```

Kuva 59. Router-luokan luonti

Autentikaatiolla varustettu Router-luokka vaatii jo hieman enemmän konfigurointia, kuin aikaisemmin luotu games-luokka. Kun asiakkaalta lähetetään pyyntö autentikoituun polkuun, kulkee se ensin välikappaleen läpi, joka tarkistaa asiakkaan tokenin voimassaolon. Jos autentikaatio onnistuu, pyyntö pääsee kuvan router-luokkaan, joka vastaanottaa request-objektin ja asettaa sen muuttujaan, jotta sitä voidaan hyödyntää luokan toiminnoissa. Välikappale, joka pitää huolen autentikaatiosta, lisää tunnistetun käyttäjän id:n request-muuttujaan.

Kuva 60. Pelin lisäys käyttäjälle

Kuvassa 60 on luotu router-luokkaan call-tyyppinen route-objekti, jonka tehtävä on vastaanottaa pelin id, ja lisätä se käyttäjän listaan. Luokan request-muuttujasta irrotetaan käyttäjän id ja haetaan sillä tietokannasta kokonainen käyttäjä-objekti. Jos käyttäjää ei löydy, lähetetään asiakkaalle pathValue-objekti, joka kertoo virheestä.

Jos käyttäjä löytyy, lisätään tämän listaan pyydetty peli, joka on tullut pyynnön mukana. Tämän jälkeen lähetään takasin pathValue-objekti, joka kertoo onnistuneesta operaatiosta.

Kuva 61. Pelin poistaminen käyttäjältä

Kuvassa 61 on luotu call-tyyppinen route-objekti, joka käsittelee vastaanotetun pelin poistamisen käyttäjän listalta. Ensin tietokannasta haetaan olemassa olevaa käyttäjää request-objektista irrotetulla id:llä. Jos käyttäjää ei löydy tai käyttäjän listassa ei ole yhtäkään peliä, palautetaan asiakkaalle virheviesti. Jos käyttäjä löytyy ja listassa on vähintään yksi peli, kutsutaan tietokantaluokan funktiota, jolla poistetaan peli pyynnön argumenttina tulleen pelin id:tä käyttäen. Tämän jälkeen lähetetään asiakkaalle viesti onnistuneesta operaatiosta.

7 Yhteenveto

REST- ja Falcor-rajapintoja rakentaessa kävi nopeasti selväksi, että vertailtavat ovat teknisesti täysin toisistaan poikkeavia ratkaisuja. REST-rajapinta on iso kokoelma suosituksia siitä, kuinka http-protokollaa tulisi hyödyntää rajapintaa rakentaessa. Falcor on taas kirjasto, jonka tarjoamien työkalujen avulla rakennetaan rajapinta, joka muoto ja rakenne on täysin kirjaston sanelema. Tämä löytö teki näiden kahden mallin vertailusta haastavaa, mutta koska molemmat tutkimuksen kohteet ovat rajapintojen rakentamiseen tarkoitettuja työkaluja, oli mahdollista vertailla kehitystyön sujuvuutta ja haasteita.

7.1 Kehitystyön tulokset

REST-arkkitehtuurimalli antaa kehittäjälle huomattavasti enemmän vapautta kehittää haluamansa rajapinta. Yhteen lopputulokseen on monta eri ratkaisua ja http-protokollan ominaisuuksia voi soveltaa monin eri tavoin. Tämä kuitenkin on kaksiteräinen miekka, joka saattaa myös aiheuttaa sen, että kehitetty rajapinta on etääntynyt kauas REST:in periaatteista ja on jo syytä miettiä, onko ratkaisu enää arkkitehtuurin mukainen.

REST-arkkitehtuurimallista löytyy kuitenkin runsaasti tutkimuksia ja aihetta käsitteleviä artikkeleita. Nämä antavat kehittäjälle mahdollisuuden kerätä vakaan tietoperustan, jonka avulla tehdä päätös rajapinnan soveltuvuudesta omaan projektiin. Runsas materiaali ja REST:in ympärille kerääntynyt laaja kehittäjäyhteisö auttavat myös kehitystyössä ilmenneisiin ongelmiin. Jos vastausta ei löydy aineistoa tutkimalla, on perusteltua olettaa, että asiasta on jo keskusteltu erinäisillä keskustelufoorumeilla.

Kun aloitin tutkimuksen REST-rajapinnasta, lähteiden puute ei koitunut ongelmaksi. Aiheesta oli kirjoitettu useita kirjoja, tutkimuksia ja artikkeleita, joka teki tutkimustyöstä helppoa. Monet alan isot toimijat käyttävät REST-rajapintaa palveluissaan, joten kriittistä ja ylistävää palautetta tuotantoympäristössä pyörivistä ratkaisuista löytyi myös runsaasti.

Falcorin kanssa tutkimusten, artikkeleiden ja kehitysyhteisön puute koituivat isoksi ongelmaksi rajapintaa rakentaessa. Tämä yllätti, sillä Netflixin dokumentointi kirjastosta oli selkeä ja kattava. Se antoi selkeän kuvan siitä, mitä tarkoitusta varten kirjasto on kehitetty, mitä ongelmia se ratkaisee, sekä kuinka sen tarjoamia työkaluja käytetään. Dokumentaatio kuvaili tarkasti, kuinka kirjaston eri komponentteja käyttäen rakennetaan rajapinta tyhjästä.

Erityisesti kokemusten ja esimerkkiprojektien puute aiheuttivat ongelmia kehitystyössä. Vaikka dokumentointi tuntui tietoperustaa kirjottaessa selkeältä, itse kehitystyötä tehdessä kysymyksiä ja ongelmia alkoi kasaantua runsaasti.

7.1.1 Falcor

Falcorin rajapinnan suunnitteluvaiheessa koin haasteelliseksi päätepisteiden konsistenttisen suunnittelun. Vaikka dokumentoinnissa käytiin läpi rajapinnan rakentamista, polkujen rakenteelle ei oltu täsmennetty noudatettavia konventioita. Niiden suunnittelu jää täysin kehittäjän vastuulle, joka saattaa aiheuttaa epävarmuutta siitä, ovatko resursseihin osoittavat polut selkeitä ja käyttäjäystävällisiä.

Käyttäjäystävällisyyttä hankaloitti myös standardisoinnin puute virheiden käsittelyssä. Netflixin dokumentaatioissa puhuttiin JSON Graphin error-tyypistä, jolla voidaan lähettää virheviestejä takaisin asiakkaalle, mutta niiden sisältö on kuitenkin kehittäjän vastuulla. Tämä tarkoittaa sitä, että virheviestien sisältö on dokumentoitava ja kuvattava, jotta asiakas ymmärtää mistä ongelmasta on kyse.

Toinen epävarmuutta aiheuttava tekijä on Falcorin sisältämät riippuvuudet muihin kirjastoihin. Falcor ladataan NodeJS-ympäristöihin npm-paketinhallinnalla ja se oli riippuvuuslistansa mukaan riippuvainen useasta eri ulkopuolisesta kirjastosta. Tämän tyyppisissä ratkaisuissa on aina riskinä, että joku kolmannen osapuolen kirjasto, jota paketti käyttää, hajoaa ja aiheuttaa ongelmia projekteihin, joissa riippuvuutta käyttävä kirjasto on käytössä.

Npm-paketinhallinta ilmoitti myös Falcor-kirjastoa ladattaessa sen sisältävän useita heikkouksia. Näihin tarjotaan aina komentoa, joilla on mahdollista yrittää korjata tilanne. Tämä ei kuitenkaan ollut mahdollista tämän kirjaston osalta. Se ei antanut vaikutelmaa siitä, että kirjastoa pidetään ajan tasalla ja eikä antanut syytä luottaa siihen, että kirjasto pysyy vakaana pidempiäkin aikoja. Tilanne aiheutti myös epäilyksen kirjaston tietoturvallisuudesta.

Yhteisön ja käytännön esimerkkien puute hankaloitti myös kehitystyössä ilmenneiden ongelmatilanteiden selvittämistä. Koska teknologia on suhteellisen nuori ja tuntematon, oli haastavaa löytää ohjeita ilmenneiden ongelmien selvittämiseksi. Esimerkkiprojekteja löytyi vain muutama ja keskusteluja ongelmatilanteista oli käyty suhteellisen vähän. Jouduin lopulta turvautumaan Stack Overflow-foorumiin, jossa kävin esittämässä kaksi kysymystä liittyen ongelmiin, joita olin kohdannut kehitystyön aikana. Molempiin, eri aikoihin kysyttyi-

hin kysymyksiini vastasi yksi ja sama henkilö muutaman päivän viiveellä. Tutkin hieman asiaa ja huomasin, että henkilö oli yksi niistä muutamista, jotka olivat vastanneet muihin foorumilla esitettyihin kysymyksiin. Toinen aktiivinen vastaaja oli Falcorin kehitystyöstä vastannut ohjelmistokehittäjä. Tämä tuntui summaavan aika lailla kehitysyhteisön nykyisen tilan, joka vaikutti erittäin vaatimattomalta.

Vaikka Falcor tuntui sisältävän paljon haasteita, löytyi rajapintateknologiasta myös paljon hyvää. Vaikka päätepisteiden suunnittelu tuntui haastavalta, oli selvää, että polun ominaisuus tukea useita erilaisia kutsuja yksinkertaistaa rajapinnan rakennetta. Rajapinnan kehittäjälle on aina helpotus, jos yhden resurssin pyynnön eri variaatiot voi sisällyttää yhteen päätepisteeseen.

Myös rajapinnan tiukka määrite siitä, että pyynnön ohessa on tarkennettava kappalemäärän ja parametrin tarkkuudella mitä rajapinnasta halutaan vastaanottaa, optimoi rajapintaa tehokkaammaksi. Tämä ominaisuus pienentää riskiä vastausten datamäärän koon paisumiseen. Kehittäjän on tieten tahdoin käytettävä JSON Graphin tarjoamaa atom-tyyppiä, jotta voidaan kiertää rajapinnan rajoitus listojen ja objektien lisäämisestä vastaukseen.

Kuitenkin yksi positiivisimmista ominaisuuksista mitä Falcor-malli tarjosi, oli virtuaalinen JSON Graph-malli, joka eliminoi tarpeen esittää duplikaattidataa, sekä yksinkertaisti ja optimoi tiedonhakua. Tämä ominaisuus näkyi erityisesti rajapinnan kehityksessä, jolloin luotiin julkista routeria, jota kautta pystyi muun muassa hakemaan pelejä sivutuksen ja id:n perusteella. Takaisin ei palautettu kuin viittaus routeen jonka avulla pystyttiin hakemaan tarkemmat tiedot peleistä, ilman että tehtäisiin uutta pyyntöä päätepisteeseen, josta pelit saadaan id:n perusteella.

7.1.2 REST

REST-rajapintaa kehittäessä haasteellisinta on pitää arkkitehtuurimallin noudattamisesta kiinni. Koska rajapintamalli ei sido kehittäjää tiukasti kiinni sääntöihin tai määrittelyihin, on helppo unohtaa tai tietoisesti jättää pois ominaisuuksia, jotka tekevät rajapinnasta juuri REST-mallin mukaisen.

REST voi aiheuttaa myös helposti hankaluuksia päätepisteiden osalta. Jos resursseja on rajapinnassa runsaasti, voi päätepisteiden määrä kasvaa hallitsemattomasti. Toisin kuin Falcor, joka mahdollistaa resursseihin johtavien polkujen joustavan käytön, REST-mallissa osoitteet pyritään pitämään selkeinä ja yksinkertaisina, eikä se anna mahdollisuutta pyytää poikkeavaa vastausta olemassa olevaa osoitetta muuttamalla. Vertailukel-

poisena esimerkkinä voi pitää tutkimuksen yhteydessä rakennetun rajapinnan pyyntöä, jossa pelit haetaan id:n perusteella. REST-rajapintaan osoitteella oli mahdollista pyytää yhden pelin tietoja, kun Falcor-malli tuki yhden tai useamman id:n lähettämistä polun mukana.

Usein REST-rajapintaa kritisoidaan myös raskaudesta. Tämä johtuu usein siitä, että rajapinnasta voidaan palauttaa kokonaisia listoja ja objekteja, joiden koko voi kasvaa hallitsemattoman suureksi, jos rajapintaa ei ole suunniteltu huolellisesti. Toisin kuin Falcor-malli, joka vaatii käyttäjää täsmentämään pyydettävien resurssien määrän, eikä suoraan tue objektien ja listojen palauttamista asiakkaalle, REST-rajapinta ei rajoita vastausten rakennetta tai kokoa. Silloin rajapinnan optimointi jää täysin kehittäjän vastuulle. Itse en kuitenkaan pidä tätä ongelmana, sillä REST-rajapintamalli tukee erittäin hyvin rajapinnan optimointia. Yksinkertaiset ominaisuudet, kuten kyselyparametrit, antavat hyvän tavan rajata asiakkaalle lähettävän datan määrää. Tämä onnistui helposti myös rakennetussa rajapinnassa, jossa kaikkien pelien haku tapahtui sivutuksen avulla.

REST-rajapintamallin parhaimmaksi ominaisuudeksi valikoituu mielestäni laaja, vakiintunut ja kokenut yhteisö. Koska malli ei ole uusinta laatuaan, useat kehittäjät ovat jakaneet kokemuksiaan rajapintamallin käytöstä niin kehitys- ja tuotantoympäristöissä, isoissa ja pienissä projekteissa. Jos kehittäjä kohtaa rajapintamallin kanssa ongelmia, on lähes varmaa, että vastaus löytyy kehittäjäfoorumeilta, kirjallisuudesta tai erinäisistä tutkimuksista.

Rajapintamallin vahvuuksiin kuuluu myös http-protokollaan pohjautuvat virhekoodit. Ohjelmistokehittäjän on helppo viestiä asiakkaalle ilmenneistä ongelmista kielellä, joka on vakiintunut yleiseen käyttöön. Virhekoodit on suunniteltu selkeiksi ja yksinkertaisiksi, joiden tarkoitus on kertoa käyttäjälle yksiselitteisesti ilmennyt ongelma. Kehittäjä kykenee ilmoittamaan suurpiirteisesti tarvittaessa vain kolmella eri virhekoodilla kaikista ongelman aiheuttajan alkuperistä.

7.2 Rajapinnan valinta

Ennen rajapintamallista tehtävää päätöstä on tärkeää kartoittaa kehitettävän rajapinnan käyttötarkoitus. Jos rajapinnan dataa on tarkoitus hyödyntää mobiililaitteilla tai muilla kevyillä alustoilla, Falcor-rajapintamalli tarjoaa vaihtoehdon tehokkaan, optimoidun rajapinnan kehittämiseen. On kuitenkin hyvä pitää mielessä teknologian nuori ikä, varsinkin jos kehitettävä projekti on laaja. Ei ole epätodennäköistä, että kirjastoa hyödyntäessä voi kohdata ongelmia, joihin ei löydy valmista tai edes suuntaa antavaa vastausta ulkopuoli-

sesta lähteestä. Myös kirjaston riippuvuuksien tila on hyvä käydä läpi ja tiedostaa mahdolliset tietoturvariskit.

REST-rajapinta on varma valinta, jossa on hyviä, sekä huonoja puolia. Rajapinnan optimointi ja käytettävyys on täysin kehittäjän vastuulla, mutta huolellisella suunnittelulla rajapintoja voidaan kehittää myös alustoille, jotka vaativat kevyitä ja optimoituja ratkaisuja.

8 Pohdinta

Opinnäytetyössä vertailtiin rajapinnan kehitystyön sujuvuutta REST-arkkitehtuurimallin ja Falcor-kirjaston välillä. Tutkimuksen aikana selvitettiin, kumpi vertailluista tarjoaa paremmat puitteet sujuvaan kehitykseen, tukien ohjelmistokehittäjän matkaa dokumentaation, teknisten ominaisuuksien ja yhteisön osalta.

8.1 Haasteet

Suunnitteluvaihe paljastui opinnäytetyön isoimmaksi kulmakiveksi, erityisesti työn rajaus. Tämä tuli ilmi ikävästi jo työn ollessa puoliksi valmis. Alkuperäisessä suunnitelmassa oli tarkoituksena käydä läpi myös rajapinnan testauksen työkaluja, sekä muita kolmannen osapuolen kirjastoja, jotka tukevat tutkittujen arkkitehtuurien avulla kehittämistä. En aikaisemmin ollut joutunut tilanteeseen, jossa tutkimuksen aiheita olisi karsittava, joten tämä toimi opettavana ja silmiä avaavana kokemuksena.

Toisena, isona haasteena oli toisen kehitettävän teknologian Falcorin, tutkimusten ja yhteisön puute. Kun rakensin teknologialla rajapintaa opinnäytetyötä varten, kohtasin työt pysäyttävän ongelman, johon ei löytynyt vastausta dokumentaatiosta tai foorumeilta. Esitinkin asiasta itse kysymyksen ohjelmistokehitykseen liittyvällä keskustelupalstalla, mutta jouduin jo valmistautumaan ajatukseen, että en saa rajapintaa valmiiksi. Vaikka sainkin muutamaa päivää myöhemmin vastauksen kysymykseeni, tilanne sai minut näkemään opinnäytetyön tarkoituksen uudesta näkökulmasta. Ymmärsin, että suunniteltua lopputulosta tärkeämpää on matka, jota pitkin kuljetaan maaliin asti. Jos kehitys olisi keskeytynyt, olisin keskittynyt enemmän kehitystyön haasteisiin ja siihen, miten sitä olisi voinut parantaa.

8.2 Jatkokehitys

Tutkimusta olisi hyvä jatkokehittää vertailemalla rajapinnoille tarjolla olevia, kolmannen osapuolen työkaluja esimerkiksi testaamisen osalta. Testaus ovat kuitenkin iso osa rajapinnan kehitystyötä, jota ilman lopullista tuontantoonpanoa ei tapahtuisi. Myös rajapintojen hyödyntämisen tutkiminen, esimerkiksi käyttöliittymän puolelta, olisi tärkeää tutkimustietoa rajapinnan kehittäjille.

9 Lähteet

Falcor 2015. A JavaScript library for efficient data fetching. Luettavissa: https://netflix.github.io/falcor/. Luettu: 20.3.2019.

Falcor 2015a. Falcor-malli. Luettavissa: https://netflix.github.io/falcor/images/falcor-end-to-end.png. Luettu: 20.3.2019.

Falcor 2015b. Model-objektin alustus. Luettavissa: https://netflix.github.io/falcor/documentation/model.html#how-the-model-works. Luettu: 20.3.2019.

Falcor 2015c. Get-pyynnöt. Luettavissa: https://netflix.github.io/falcor/documentation/model.html#retrieving-data-from-the-model. Luettu: 20.3.2019.

Falcor 2015d. Set-pyynnöt. Luettavissa: https://netflix.github.io/falcor/documentation/model.html#setting-values-using-the-model. Luettu: 20.3.2019.

Falcor 2015e. Call-pyynnöt. Luettavissa: https://netflix.github.io/falcor/documentation/model.html#calling-functions. Luettu: 20.3.2019.

Falcor 2015f. Batch-metodi. Luettavissa: https://netflix.github.io/falcor/documentation/model.html#batching-outgoing-requests. Luettu: 20.3.2019.

Falcor 2015g. Polut. Luettavissa: https://netflix.github.io/falcor/documentation/paths.html#paths Luettu: 20.3.2019.

Falcor 2015h. Primitiiviset datatyypit. Luettavissa: https://netflix.github.io/falcor/documentation/jsongraph.html#new-primitive-value-types. Luettu: 20.3.2019.

Falcor 2015i. Datasource. Luettavissa: https://netflix.github.io/falcor/documentation/datasources.html#data-sources. Luettu: 20.3.2019.

Falcor 2015j. Datasource-operaatiot. Luettavissa: https://netflix.github.io/falcor/documentation/datasources.html#datasource-operations. Luettu: 20.3.2019.

Falcor 2015k. Router-luokan luominen. Luettavissa: https://netflix.github.io/falcor/documentation/router.html#creating-a-router-class. Luettu: 20.3.2019.

Kapadnis, J. 2018. REST: Good Practices for API Design. Medium. Luettavissa: https://medium.com/hashmapinc/rest-good-practices-for-api-design-881439796dc9. Luettu: 2.2.2019

Korpela, J. 2009. Web-julkaisemisen opas. Luettavissa: http://jkorpela.fi/webjulk/. Luettu: 2.2.2019.

MDN web docs 2018. HTTP request methods. Luettavissa: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods. Luettu: 1.2.2019

MDN web docs 2019. HTTP Messages. Luettavissa: https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages. Luettu: 1.2.2019

Medium 2017. When Should You Use Path Variable and Query Parameter? Luettavissa: https://medium.com/@fullsour/when-should-you-use-path-variable-and-query-parameter-a346790e8a6d. Luettu: 17.3.2019.

Pivotal Software 2019. Understanding REST. Luettavissa: https://spring.io/understanding/REST. Luettu: 4.2.2019

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. 1999. Hypertext Transfer Protocol -- HTTP/1.1. The Internet Society.

R. Fielding. 2000. Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine. Roy Fielding.

Rouse, M. 2017. REST (REpresentational State Transfer). TechTarget. Luettavissa: https://searchmicroservices.techtarget.com/definition/REST-representational-state-transfer. Luettu: 3.2.2019

Sturgeon, P. 2015. Build APIs You Won’t Hate: Everyone and Their Dog Wants an API, So You Should Probably Learn How to Build Them. Philip J. Sturgeon.

Virender, S. 2017. HTTP-Request. Luettavissa: https://www.toolsqa.com/client-server/http-request/. Luettu: 12.3.2019