Tampereen yliopisto

Eeli Hernesniemi

AVOIMEN RAJAPINNAN SUUNNITTELU JA SEN HYVÄT KÄYTÄNNÖT

Kandidaatintutkielma
Informaatioteknologia ja viestintä
Tarkastajat: Terhi Kilamo
Tammikuu 2020

TIIVISTELMÄ

Eeli Hernesniemi: Avoimen rajapinnan suunnittelu ja sen hyvät käytännöt
Kandidaatintutkielma
Tampereen yliopisto
Tietotekniikan kandidaatin tutkinto-ohjelma
Tammikuu 2020

Rajapinta on sovelluksen osien tai kokonaisten sovellusten keskustelukanava ja avoimella rajapinnalla tarkoitetaan rajapintaa, joka on kaikkien käytettävissä. Internetin käyttäjämäärän kasvu ja sovelluksien arkkitehtuurien muutos pirstaleisempaan muotoon on asettanut enemmän vaatimuksia avoimille rajapinnoille. Monet internetin sovellukset ovat nykyään riippuvaisia ulkoisista avoimista rajapinnoista ja esiin on noussut maksullisia rajapintoja, jotka kilpailevat keskenään käyttäjistä. On tärkeää, että nämä rajapinnat ovat helppokäyttöisiä, nopeita ja hyvin dokumentoituja saavuttaakseen ison käyttäjämäärän.

Rajapintaa toteuttaessa tulee miettiä sen käyttökohdetta ja kohdeyleisöä. Rajapinta usein tehdään koneluettaviksi, mutta joissakin tapauksissa suositaan enemmän rajapinnan vastausnopeutta, kun taas toisissa helppoa liittymistä. On tärkeää, että rajapintaa alkaa rakentamaan oikealla teknologialla alusta asti, sillä rajapinnan vaihtaminen teknologiasta toiseen on työlästä. Käytetyin vaihtoehto tällä hetkellä on REST-arkkitehtuurimalli, jonka rinnalle on lähiaikoina noussut lisää kilpailijoita.

Työssä huomataan, että iästään riippumatta REST on vieläkin hyvä vaihtoehto uusille avoimille rajapinnoille. Vuonna 2000 syntynyt arkkitehtuurimalli ei sido käyttäjiä tiettyyn teknologiaan, joten sen ajatusmallit on helppo ottaa käyttöön vielä nykyäänkin. SOAP on ollut RESTin kilpailija jo pitkään, mutta ison lähetetyn datamäärän ja vaikealukuisen syntaksinsa takia sitä harvemmin valitaan uusille rajapinnoille.

Uutena tulokkaana on GraphQL, joka säilyttää RESTin helppokäyttöisyyden ja luettavuuden, mutta samalla pystyy vähentämään lähetetyn datan määrää ja kutsumääriä. GraphQL:n vahvuuksista isoimpana on monien olioiden ja niiden ominaisuuksien kysely yhdellä kyselylausekkeella, mikä helpottaa ja nopeuttaa rajapinnan käyttöä. Odotan, että GraphQL jatkaa kasvuaan ja saa uusia kilpailijoita tulevaisuudessa.

Avainsanat: Avoin rajapinta, Web API, JSON, REST, GraphQL, SOAP

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

API	Application Programming Interface
CRUD	Create, Read, Update, Delete
CSV	Comma Seperated Values
HTTP	Hypertext Transfer Protocol
ISO	Kansainvälinen standardointiorganisaatio
JSON	JavaScript Object Notation
QL	Query Language
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
TAU	Tampereen yliopisto (engl. Tampere University)
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WSDL	Web Services Description Language
WWW	World Wide Web
XML	Extensible Markup Language

1 JOHDANTO

WWW:n ja internetin suosion kasvu synnytti paljon nettisivuja ja -palveluita ihmisten käytettäväksi. Kehittäjät huomasivat, että tietotekniikan harrastajat alkoivat lukemaan näitä nettisivuja tietokoneohjelmien avulla, mikä kävi palvelimelle raskaaksi. Tämän välttämiseksi alettiin kehitellä avoimia rajapintoja, joista nettisivuilla oleva tieto saatiin koneystävällisemmässä muodossa ja paljon kevyemmin kuin selaimen kautta.

Nykyaikana avoimien rajapintojen suosion lisääntynyt valtavasti ja palveluiden täytyy kilpailla kehittäjien suosiosta samalla tavalla kuin nettisivut kilpailevat käyttäjistä. Etulyöntiaseman varmistamiseksi julkisen rajapinnan täytyy antaa käyttäjää kiinnostavaa tietoa, olla helppokäyttöinen, standardimuotoinen eli vähiten yllätyksiä antava, hyvin dokumentoitu sekä nopea. Täten on hyvä noudattaa hyviksi todettuja tietomalleja, suunnittelusääntöjä ja arkkitehtuurimalleja. REST-arkkitehtuurimalli on yksi vanhimmista rajapintojen suunnittelun sääntökirjoista, jonka opit pätevät edelleen, mutta sen kilpailijaksi on viime aikoina noussut GraphQL-kyselykieli.

Ohjelmoijalla on täten monia hyviä teknologioita toteuttaa uusi avoin rajanpinta. Tässä työssä käydään näistä isoimmat läpi ja pohditaan niiden eroja. Lisäksi työ käy läpi internetin teknologioita, joiden päälle jokainen rajapintatoteutus rakentuu. Luvussa 2 käydään läpi ohjelmointirajapinnan tarkoitusta ja käyttökohteita. Luvussa 3 kerrataan HTTP-protokollaa ja luvussa 4 vertaillaan REST-periaatteita sen kilpailijoihin.

2 OHJELMOINTIRAJAPINTA

Ohjelmointirajapinnaksi (API) kutsutaan väylää ohjelmiston komponenttien tai kokonaisten ohjelmien välillä. Sen tehtävänä on määritellä kuinka sen takana olevaa logiikkaa käytetään hukuttamatta käyttäjää mahdollisuuksilla. Hyvä rajapinta tarjoaa sovelluksesta tai datasta vain tarpeelliset osat muiden käytettäväksi. Rajapinta myös abstrahoi toteutuksen, sillä liika tekninen tieto rajapinnan takaa haittaa ohjelman luettavuutta ja käytettävyyttä. [1]

Rajapintoja on kaikkialla. Yksittäisestä ohjelmasta löytyy useita rajapintoja eri paketeista/kirjastoista, joita ohjelmoija käyttää tietyn toiminnallisuuden saavuttamiseksi. Kuvassa 2.1 on kuvattu osa C++ kielen standardimallikirjastosta löytyvän vector-luokan metodeista. Näiden rajapintakutsujen avulla ohjelmoija voi luoda vector-olion omassa ohjelmassaan ja käyttää sen toiminnallisuuksia, esimerkiksi tallentaa sinne luontikutsussa valittua tietotyyppiä, muuttaa kokoa, kysyä pituutta ja niin edelleen.

Capacity:

size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (public member function)
shrink_to_fit C++11	Shrink to fit (public member function)

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
data C++11	Access data (public member function)

Kuva 2.1. *Vector-luokan metodeja [2].*

Kirjaston yksi tärkeimmistä tehtävistä on toteuttaa jokin usein tarvittava toiminto ja jakaa se käyttäjille. Tällöin jokaisen käyttäjän ei tarvitse toteuttaa tätä toimintoa itse, mikä helpottaa kehitystä ja voidaan välttää mahdolliset käyttäjän toteuttamat virheet. Kirjastot ovat usein erikseen ladattavia osia ohjelmointikieleen, mutta jotkin kirjastot ovat niin usein käytettyjä, että ne asennetaan yhdessä ohjelmointikielen kanssa, esimerkiksi edellisessä kappaleessa mainittu C++-kielen standardimallikirjasto ja python-kielen datetime-kirjasto, joka toteuttaa monia aikaa ja päivämäärää koskevia ominaisuuksia.

Rajapintoja voi käyttää nopeuttamaan kehitystä. Ohjelman voi jakaa useampaan osaan,

jotka käyttävät kommunikoinnissaan ennalta sovittuja rajapintoja. Tämä säästää tarvittavan kommunikoinnin määrässä, joten eri osat voidaan tehdä jopa kielimuurien ja aikavyöhykkeiden yli. Tällä tavoin ohjelma voidaan saada konseptista markkinoille nopeammin ja halvemmalla kuin perinteisillä ohjelmistokehitystavoilla. [3]

Nämä ohjelman sisäiset rajapinnat eivät ole käytettävissä sen ulkopuolelta. Sen sijaan toiminnallisuutta voi edelleen jakaa ohjelman ulkopuolelle avoimien rajapintojen kautta. Rajapintoja voi 'julkaista' laitteella muiden ohjelmien käyttöön, mutta useimmiten rajapintoja jaetaan internetin yli käytettäviksi. Kuvassa 2.2 on muutama käyttöesimerkki OpenWeather-palvelun avoimen rajapinnan kutsusta. Tämän rajapinnan avulla käyttäjä voi kysyä senhetkistä säätietoa tietyssä kaupungissa tai ennustetta huomiselle. Käyttäjä voi lisäksi pyytää vastausta XML- tai JSON-formaatissa.

Kuva 2.2. *OpenWeather-palvelun rajapinnan esimerkkejä [4].*

Kuvassa nähdään myös 'API key' eli rajapinta-avain, joita avoimien rajapintojen kehittäjät käyttävät rajapinnan käyttäjien varmennukseen ja seurantaan[5]. Avaimen saaminen usein vaatii tunnistautumista, esimerkiksi Googlen palveluiden kautta. Rajapinnan kehittäjät voivat asettaa suurimman sallitun pyyntömäärän avainta kohti ja periä sen yli menevistä pyynnöistä lisämaksua. On olemassa myös kokonaan maksullisia julkisia rajapintoja, joissa hinnat skaalautuvat pyyntömäärän mukaan. Googlen karttasovelluksen API-avain maksaa minimissään 2 dollaria kuukaudessa, minkä avulla kehittäjä voi käyttää karttaa omassa sovelluksessaan. Pyyntömäärä on rajoitettu 1000 kutsuun, joten suurempaa maksua tarvitaan jos kehittäjä saa sovellukselleen paljon käyttäjiä. [6]

Internetin yli käytettäessä tietoliikenne, rajapintakutsut mukaan luettuna, kulkee suurimmaksi osaksi HTTP-protokollan kautta, jota tarkastelemme seuraavassa luvussa.

3 HTTP-PROTOKOLLA

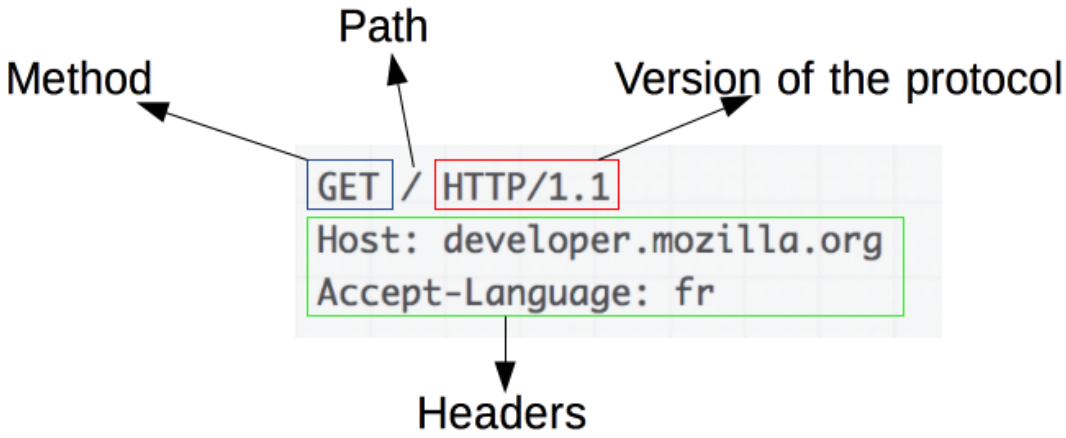
HTTP (Hypertext Transfer Protocol) on protokolla, jonka avulla voi hakea verkon takaa resursseja, esimerkiksi html-tiedostoja. Selaimet lähettävät tämän protokollan mukaisia kutsuja nettisivuja ladatessaan, joten niistä koostuu suuri osa internetliikenteestä. HTTP vaatii asiakkaan (selain) ja WWW-palvelimen välille TCP-yhteyden, jota kautta asiakas voi lähettää kutsun. [7]

HTTP sai alkunsa World Wide Web -aloitteesta vuonna 1991. Versionumerolla 0.9 varustettuna se määritteli vain 'GET'-pyynnön [8]. HTTP 1.0 julkaistiin vuonna 1996, missä määriteltiin lisäksi 'HEAD' ja 'POST' -metodit. [9] Protokollaa kehitetään edelleen ja nykyinen versio on vuonna 2015 julkaistu 2.0.

HTTP 2:n seuraajaksi on tulossa HTTP 3, toiselta nimeltään QUIC-protokolla, joka on ollut jo testikäytössä nimellä 'HTTP-over-QUIC'. QUIC käyttää TCP:n sijasta UDP-verkkotasoa, mikä lyhentää viestinnässä tapahtuvaa viivettä merkittävästi. [10]

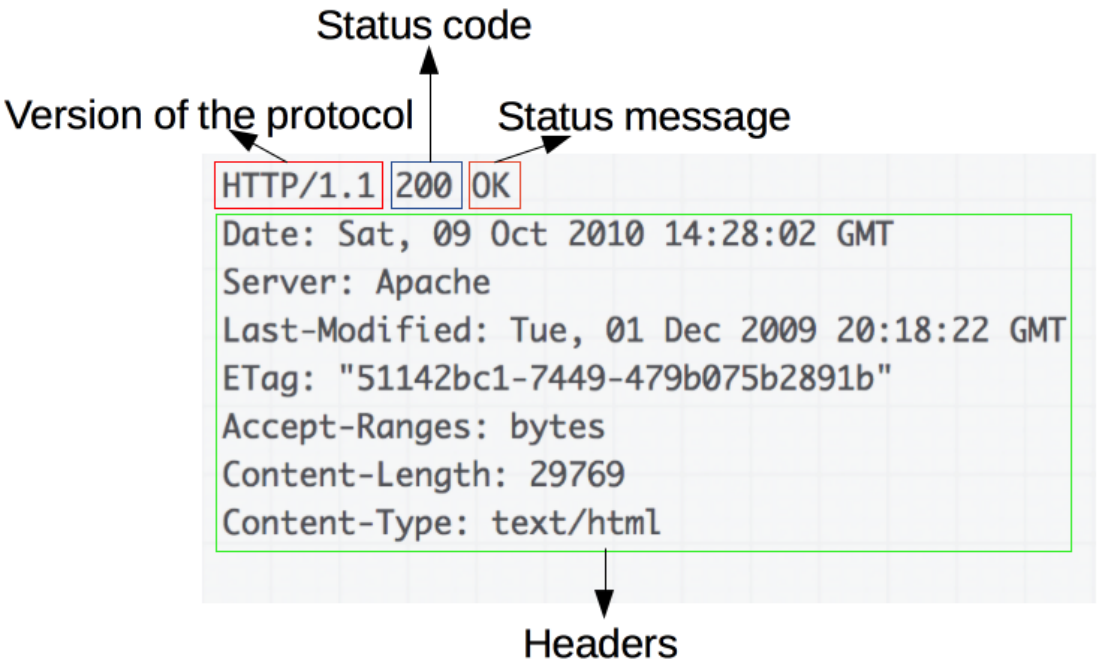
3.1 HTTP-kutsu ja -vastaus

HTTP-protokollassa tiedon välitys on jaoteltu osiin erillisiin viesteihin. Asiakas, esimerkiksi selain, voi lähettää verkkopalvelimelle kutsun, johon palvelin vastaa HTTP-otsikkotiedoilla ja mahdollisella näytettävällä sisällöllä. HTTP-kutsun tärkeä osa on käytettävä metodi, josta selviää minkälaisen operaation kutsu haluaa suorittaa. Näitä metodeita on nykyään yhdeksän, joista ensimmäiset neljä ovat CRUD-toimintoja: GET hakee resurssin, POST luo uuden resurssin, PUT päivittää olemassaolevaa resurssia ja DELETE poistaa resurssin. Vähemmän tunnettuja HTTP-metodeja ovat HEAD, joka hakee vain otsikkotiedot, CONNECT, joka avaa yhteyden palvelimelle, OPTIONS, joka kertoo mitä HTTP-metodeja kyseinen resurssi ottaa vastaan, TRACE, joka palauttaa kutsun käymän polun, sekä PATCH, joka muuttaa jotain resurssin arvoa ilman, että asiakkaan tarvitsee lähettää koko esitystä palvelimelle. [11]



Kuva 3.1. *HTTP-kutsun rakenne [7].*

HTTP-kutsu (kuvassa 3.1) alkaa käytettävällä metodilla, polulla joka on URL tai IP-osoite, sekä protokollaversiolla. Tämän jälkeen tulevat otsikkotiedot ja lopuksi mahdollinen runko (POST, PUT ja PATCH metodeilla). HTTP-vastaus (kuvassa 3.2) alkaa protokollaversiolla, statuskoodilla ja statusviestillä. Näiden jälkeen tulevat otsikkotiedot ja runko kuten HTTP-kutsussakin.



Kuva 3.2. *HTTP-vastauksen rakenne [7].*

Näistä voidaan päätellä oliko kutsu onnistunut ja jos jokin meni pieleen niin mikä. Onnistuneisiin kutsuihin vastataan koodilla 200–299, uudelleenohjaukset käyttävät koodeja 300-399, asiakkaan virheet ilmoitetaan koodeilla 400-499 ja palvelimen virheet koodeilla

500-599. Esimerkiksi hakiessa polkua, jota ei ole olemassa, saadaan vastauksena statuskoodi 404 ja statusviestiksi ’Not Found’. [12]

3.2 Resurssin lähettäminen

HTTP-protokollan mukaisessa tietoliikenteessä viestejä lähetetään edestakaisin useimmiten REST-periaatteiden mukaisesti. Tämä tarkoittaa, että internet koostuu pääosin resursseista ja niiden esityksistä. Resurssi voi olla mitä vain. Se voi olla jokin tietty tieto, tiedosto, oikea esine tai kokoelma osoittimia muihin resursseihin. Resurssi voi täten olla jotakin, mitä ei edes pysty lähettämään tietoliikenneyhteyden kautta, esimerkiksi henkilö. [13]

Resursseja itsejään ei ikinä lähetetä, sen sijaan välittyy vain resurssin esitys. Asiakkaan kysyessä palvelimelta jotain tiettyä resurssia palvelin lähettää vain esityksen resurssin senhetkisestä tilasta. Esitysmuoto on kuvattu HTTP-vastauksen otsikkotiedoissa, mutta sen voi myös joskus päätellä suoraan URL:stä. Tällä tavalla samasta resurssista voidaan lähettää montaa eri esitystä, riippuen minkälaista esitystä tarvitaan. [13, 14] Yksi esitysmuoto voisi olla resurssin upottaminen HTML-sivuun, jonka selain sitten tarjoilee käyttäjälle. Robotin pyytäessä resurssin esitystä on HTML-sivun lähettäminen turhaa, sillä se ei paranna lukunopeutta (tai mukavuutta). Tällöin voidaan karsia suuret määrät oheistietoa ja lähettää vain tärkeimmät, resurssia koskevat asiat, esimerkiksi JSON-muodossa.

Resurssin pitäisi täten olla ottamatta kantaa siihen kuinka se tulisi esittää ja pitää sisältämänsä tiedon mahdollisimman abstraktina. [13]

3.3 URI ja URL

URI, eli ’Universal Resource Identifier’ on resurssin yksilöivä nimi. Jokaisella resurssilla tulisi olla oma URI, ja yksikään URI ei saisi osoittaa enempään kuin yhteen resurssiin. URL, eli ’Universal Resource Locator’ on URI, jonka perusteella selain osaa etsiä oikean resurssin. [15] Jokainen URL on siis URI, mutta jokainen URI ei ole URL.

```
eeli.hernesniemi@          ~
$ curl -X GET https://dog.ceo/api/breed/retriever/golden/images/random
{"message":"https:\/\/images.dog.ceo\/breeds\/retriever-golden\/n02099601_281.jpg","status":"success"}
```

Kuva 3.3. *Avoimen rajapinnan käyttöä, mikä hakee käyttäjälle satunnaisia koirakuvia. (https://dog.ceo/dog-api/)*

URL on siis se osoite, jolla selain etsii DNS-palvelimelta oikean verkko-osoitteen. Iso osa rajapintasuunnittelua on näiden osotteiden suunnittelu, jotta rajapinnan käyttäminen on mahdollisimman sulavaa. Hyvää URL-suunnittelua on kuitenkin vaikea toteuttaa, jos resurssit eivät ole saaneet tarvitsemaansa huomiota. Kuvassa 3.3 haetaan HTTP:tä käyttäen satunnainen kuva kultaisesta noutajasta, minkä pystyy päätellä pitkälti URI:n loppuosasta ’api/breed/retriever-golden/images/random’.

3.4 Resurssin esitysmuodot

Avoin rajapinta vastaa asiakkaan kyselyyn datalla, mutta sille on valittava oikea esitysmuoto. Datasta saa tietoa vain, jos sen osaa tulkita. Oikea esitystapa riippuu asiakkaasta ja kutsusta; ihmissilmille kannattaa esittää esimerkiksi HTML-tiedostoja, roboteille voi tarjota jotain raaempaa.

Rajapinnoista puhuttaessa usein oletetaan muodon olevan koneluettava, jotta rajapinnan tarjoamaa dataa voi käyttää muissa ohjelmissa tarvitsematta muuntaa sitä toiseen muotoon. Suosituimmat koneluettavat muodot ovat CSV, JSON ja XML. Näistä JSON on ylivoimaisesti suosituin helppokäyttöisyytensä ja nopeutensa takia[16].

4 RAJAPINTASUUNNITTELU

Nykyajan ohjelmistokehitys on usein enemmän valmiiden palasten sovitusta yhteen rajapintojen avulla, kuin uuden palasen tekemistä[17][18]. Nämä palaset (ohjelmat, luokat, kirjastot) ovat usein eri henkilöiden kirjoittamia, ja usein ne ovat tehty eri tarkoituksiin. Näitä palasia käytetään rajapintojen kautta, mikä antaa niiden käyttäjälle mahdollisuuden säästää aikaa olla ymmärtämättä täysin jokaisen palasen sisäistä logiikkaa. Käyttäjä voi sen sijaan käyttää tästä palasesta vain tarvittavia toiminnallisuuksia ja hyvän rajapintadokumentaation myötä ymmärtää silti kokonaisuus. Nykyarkkitehtuureissa näitä muiden tekemiä palasia käytetään paljon, joten uusia rajapintoja voi tulla vastaan lähes päivittäin. [18]

Talon sisäisiä rajapintoja kehittäessä olisi toivottavaa, että tiimitoverisi tajuavat kuinka ohjelmasi toimii sulavan kehityksen takaamiseksi. Ohjelman laajuudesta riippuen tällaisia suljettuja rajapintoja kehittäessä loppukäyttäjiltä pystyy olettaa paljon enemmän kuin täysin avoimien rajapintojen käyttäjiltä. Rajapinta voi esimerkiksi seurata yhtiön omaa tyyliopasta, jonka vuoksi sieltä tulevia käytäntöjä ei tarvitse erikseen dokumentoida. Näistä syistä tiimin sisäiset ohjelmat voivat olla myös teknisempiä ja palvella tarkempaa tarkoitusta. [19] Kuten koodikin, rajapinnat voivat myös olla itsedokumentoivia hyvin suunniteltuna, jolloin erillistä dokumentaatiota tai sen ylläpitoa ei tarvita[17].

4.1 REST

REST-arkkitehtuurimalli sai alkusta Roy Thomas Fieldingin väitöskirjasta vuonna 2000, missä hän kuvasi verkkopalveluiden ja asiakasohjelmien välistä vuorovaikutusta [14]. REST, eli representational state transfer, asettaa rajoitteita ja sääntöjä tähän kommunikaatioon ja nämä täytettyä mahdollistaa tiedon jakamisen HTTP:n välityksellä. Fielding jakoi nämä säännöt kuuteen osaan:

1. Asiakkaan ja palvelimen ero
2. Yhteinen rajapinta
3. Välimuisti
4. Tilattomuus
5. Kerroksittainen järjestelmä
6. Ajettavan koodin lähettäminen

On tehtävä selvä ero asiakkaan ja palvelimen välille, joista verkkopalveluiden perusarkkitehtuuri koostuu. Asiakkaan tehtävänä on lähettää palvelimelle pyyntö, jonka palvelin käsittelee ja palauttaa vastauksen asiakkaalle. Verkkopalveluissa ja näiden koneluettavissa rajapinnoissa tämä tapahtuu yleensä HTTP-protokollan välityksellä. REST-ajatuksen mukaan asiakas ja palvelin voi hyödyntää mitä vain teknologiaa, kunhan ne käyttävät WWW:n yhtenäistä rajapintaa. [13]

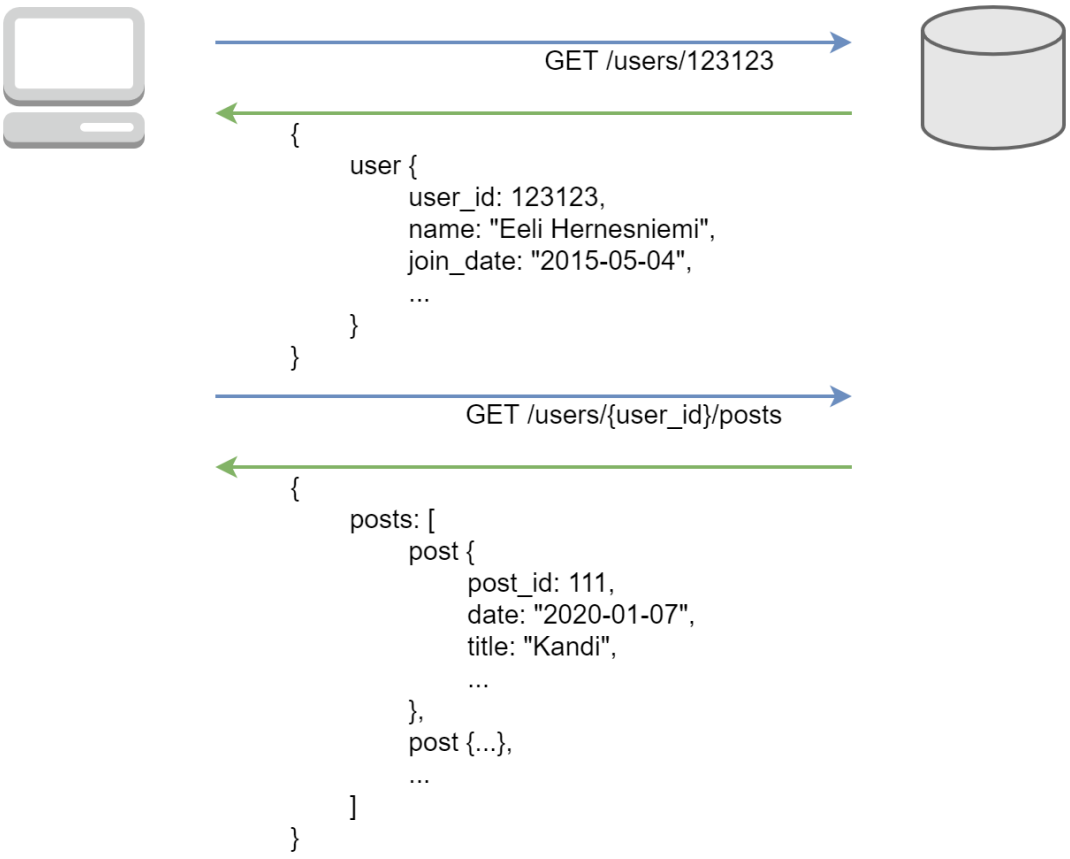
Yhtenäinen rajapinta vaatii verkkopalvelun valita eri resurssit mitä tarjota asiakkaille API:n kautta. Jokaiselle resurssille tulee valita oma uniikki URI, jota kautta saadaan kuvaus resurssista, eikä sitä itseään. Tällä tavoin sama kohde voidaan esittää eri tarkoituksiin eri muodoissa, esimerkiksi selaimelle HTML-muodossa ja automatisoidulle ohjelmalle JSON-muodossa. [13]

REST velvoittaa palvelinta ilmoittamaan onko lähetetyn vastauksen data tallennettavissa välimuistiin. Jos se on, data voidaan tallentaa ja käyttää uudelleen välttyen saman pyynnön lähettämiseltä useaan kertaan. Tämä tallennus voi tapahtua missä vain asiakas–palvelin -polun matkalla. Tämä vähentää palvelimen taakkaa ja nopeuttaa asiakkaan toimintaa. [13]

Tilattomuus tarkoittaa, ettei palvelin pidä kirjaa asiakkaiden tiloista, vaan asiakkaan pyynnössä tulisi olla kaikki tieto mitä palvelin tarvitsee pyynnön noudattamiseen. Tämä seurauksena jokaisen asiakkaan täytyy itse pitää kirjaa tilastaan. [13] Välimuistin lailla tämä on palvelimen taakkaa vähentävä rajoite, joka parantaa verkkopalvelun skaalautuvuutta ja sallii suuremmat käyttäjämäärät.

Kerroksittaisen järjestelmä mahdollistaa välipalvelimien käytön esimerkiksi välimuistin tallentamiseen tai identiteetin varmentamiseen. Kun kerroksia on monta, ei asiakas tiedä onko se suoraan kutsuttavaan palvelimeen yhteydessä vai ei. Tämä motivoi jokaista kerrosta noudattamaan tiukemmin yhtenäistä rajapintaa. [13]

RESTin viimeinen ja löysin rajoite on ns. code-on-demand, eli ajettavan koodin lähettäminen palvelimelta asiakkaalle. Tämä on näistä ainoa sääntö, joka olettaa asiakkaan ja palvelimen käyttävät samaa teknologiaa, mistä syystä sitä pidetään vaihtoehtoisena. Tätä noudattaessa palvelimella on mahdollisuus siirtää asiakkaalle ajettavia ohjelmia tai skriptejä tilapäisesti. Esimerkkeinä tästä ovat Java-, Flash-, ja JavaScript-ohjelmat. [13]



Kuva 4.1. *Esimerkki REST rajapinnasta, josta haetaan ensin käyttäjän tiedot ja toisena käyttäjän tekemät julkaisut.*

REST pilkkoo hallittavat resurssit pieniin palasiin, eikä mielellään säilytä sisäkkäisiä olioita. Data pysyy näin staattisena, mutta käyttäjälle ilmenee ongelma: ihmisluettavan tiedon rakentaminen vaatii usein monta rajapintakutsua palvelimelle (kuva 4.1) ja tietojen yhdistely täytyy toteuttaa itse. [20] Jotkut rajapintojen toteutusperiaatteet tekevät tämän yhdistelyn palvelimen puolella, kuten seuraavista esimerkeistä näemme.

4.2 SOAP

Vuonna 1999 julkaistu SOAP-protokolla ja sen implementoima SOA-arkkitehtuuri on pitkään ollut tunnetuin vaihtoehto REST-arkkitehtuurille. SOA, eli service oriented architecture, jakaa sovelluksen osiin ja jokaiselle osalle luodaan sopimus sen toiminnallisuudesta. Tämä rajapintasopimus tehdään WSDL-tiedostolla, joka puolestaan käyttää XML:n syntaksia. Tiedostossa on lueteltuna palvelun jokainen funktio ja niiden käyttämät tietotyypit. [21][22] Esimerkki tämmöisen tiedoston osasta on kuvassa 4.2.

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>
```

Kuva 4.2. *Osa WSDL-tiedostosta. (https://www.w3schools.com/xml/xml_soap.asp)*

Tämä funktioiden vahvasti määrittely tekee ohjelman rajapinnasta vaikeammin tulkittavan, mutta parantaa toimintavarmuutta. SOAP-kutsussa jokainen kenttä on nimetty, joten vahingossa väärän metodin käyttäminen on lähes mahdotonta. Kenttien vahvasta nimeämisestä johtuen SOAP-kutsut vievät enemmän kaistaa kun vastaavat REST-kutsut. SOAP myös käyttää enemmän prosessointitehoa, joten skaalautuvuus on heikompi. [23][21]

4.3 GraphQL

GraphQL on vuonna 2015 julkaistu kyselykieli (query language), joka yhdistää ominaisuuksia RESTistä ja SOAPista. Rajapinnat tulee määritellä tarkasti, kuten SOAP-protokollaa käytettäessä, minkä jälkeen tietoa voi hakea palvelimelta. Kyselyä ei tehdä suoraan resurssin URL-osoitteeseen, kuten REST-arkkitehtuuria käytettäessä, vaan muodostetaan kysely. [24] Alla on esimerkki GraphQL tietomallista eli schemasta, jossa on määritelty kolme tietotyyppiä.

```
type User {
    user_id: Int
    name: String
    email: String
}

type Item {
    item_id: Int
    name: String
    weight: Int
}
```

```
type Order {
    order_id: Int
    customer: User
    product: Item
}
```

Tietomallista voidaan hakea kaikki Item-oliot sekä niiden tunnisteet ja nimet seuraavalla kyselyllä.

```
query {
    Item {
        item_id
        name
    }
}
```

Jos palautettavat kentät jättää määrittelemättä, palautuu koko olio. Alla oleva kysely palauttaa tilauksen tuotteen nimen, sekä asiakkaan kokonaisuudessaan.

```
query {
    Order {
        product {
            name
        }
        customer
    }
}
```

Toisin kuin REST-rajapinnan päätepisteitä käyttäessä, GraphQL-kyselyllä voi valita kentät mitä palvelin palauttaa vastaksena. Tällöin käyttäjä voi pyytää rajapinnasta vain ne kentät jota sovellus tarvii ja vältytään ylihakemiselta, joka lisää sovelluksen monimutkaisuutta ja verkon yli kulkevan datan määrää. Isoilla rajapinnoilla tämä voi vähentää datan määrän jopa prosenttiin siitä mitä se olisi REST-rajapinnan kautta. [25]

4.4 Vertailu

Vaikka SOAP tarjoaa luotettavampaa tiedonsiirtoa, jää sen RESTin varjoon nopeudessa, datamäärässä ja muistinkäytössä. Luotettavuuden menetyksen REST on osittain paikannut tekemällä rajapinnasta tilattoman, jolloin kutsuja voidaan lähettää uudestaan tekemättä palvelimelle muutoksia. Näistä syistä useissa käyttötapauksissa suositaan RESTiä, mutta SOAPilla on luotettavuutensa takia vielä paikka tietoturvakriittisissä sovelluksissa. [23]

RESTiä voi käyttää usealla dokumenttityypillä, joista suosituimmat ovat XML, JSON ja CSV. XML antaa samankaltaisia etuja kuin SOAP ollessaan vahvasti nimetty tapa lähet-

tää tietoa. JSONissa otsikot ovat myös nimettyvä, mutta syntaksinsa puolesta ne ovat helposti ihmisen luettavissa ja kirjotettavissa. CSV:n otsikkotietojen vähyys tekee siitä hyvän vaihtoehdon silloin, kun halutaan lähettää suuria määriä dataa. [26]

GraphQL-kyselykielen syntaksi on monelle helpompi ymmärtää kuin peräkkäiset REST-kyselyt, joita tarvitaan sisäkkäisen datan hakemiseen palvelimelta. Luettavuuden takia on myös huomattu, että ennalta kokemattomat ja REST-rajapinnoissa kokeneet pystyvät implementoimaan GraphQL-kyselyitä nopeammin kuin REST-vastineita. GraphQL-tietomalli mahdollistaa ohjelmointityökaluille lisäominaisuuksia mitä REST rajapinnat ei tue, esimerkiksi sanojen automaattisen täydennyksen ja tyyppien tarkastuksen. [27]

5 YHTEENVETO

Internetin rakennuspalikat ovat kehittyneet vuosien saatossa, mutta samat periaatteet pätevät yhtä. Samaa voi sanoa rajapintojen suunnitteluperiaatteista. Ohjelmien pirstaloituessa monoliittisista arkkitehtuurimalleista pienimmiksi osiksi tarvitaan rajapintoja yhä enemmän. Rajapintojen suurten lukumäärän vuoksi hyväksi todetut yhteiset periaatteet ovat tärkeitä, jotta kokonaisuus pysyy hallittavana.

Roy Fieldingin keksimät REST-periaatteet ovat päivittyneet aikojen saatossa ja pätevät yhä. Nykyäänkin on toivottavaa, että uudet rajapinnat noudattavat näitä 22 vuotta vanhoja periaatteita. RESTin pitkäaikaisin kilpailija SOAP voi vieläkin hyvin, mutta monet avoimen rajapintojen kehittäjät suosivat RESTin luettavuutta SOAPin vahvan tyypityksen yli. Viimeaikoina pakkaa on sekoittanut Facebookin kehittämä GraphQL kyselykieli, joka yhdistelee hyvät puolet RESTistä ja SOAPista, ja onnistuu samalla vähentämään datan määrää huomattavasti. Odotan, että GraphQL jatkaa kasvuaan ainakin avoimien rajapintojen puolella ollessaan huomattavasti luettavampi kokonaisuus kuin REST tai SOAP.

LÄHTEET

[1] Reddy, M. *API Design for C++*. Elsevier Science, 2011. ISBN: 9780123850041. URL: https://books.google.fi/books?id=IY29LylT85wC.

[2] *Vector - C++ Reference*. 2020. URL: https://www.cplusplus.com/reference/vector/vector/ (viitattu 01. 11. 2021).

[3] Shrivastava, S. V. et al. Distributed agile software development: A review. *arXiv preprint arXiv:1006.1955* (2010).

[4] *Current weather data - OpenWeather API Reference*. 2021. URL: https://openweathermap.org/current (viitattu 07. 11. 2021).

[5] Farrell, S. API Keys to the Kingdom. *IEEE Internet Computing* 13.5 (2009), 91–93.

[6] *Pricing Plans and API Costs - Google Maps Platform*. 2022. URL: https://mapsplatform.google.com/pricing/ (viitattu 28. 03. 2022).

[7] *An overview of HTTP*. 29. syyskuuta 2020. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview (viitattu 07. 05. 2020).

[8] *The Original HTTP as defined in 1991*. 1. tammikuuta 1991. URL: https://www.w3.org/Protocols/HTTP/AsImplemented.html (viitattu 07. 05. 2020).

[9] Fielding, R. ja Berners-Lee, T. *Uniform Resource Identifier (URI): Generic Syntax*. eng.

[10] *Hypertext Transfer Protocol Version 3 (HTTP/3)*. 21. helmikuuta 2020. URL: https://tools.ietf.org/html/draft-ietf-quic-http-27 (viitattu 08. 05. 2020).

[11] *HTTP request methods. Mozilla*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods (viitattu 17. 05. 2020).

[12] *HTTP response status codes. Mozilla*. 21. toukokuuta 2020. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status (viitattu 23. 05. 2020).

[13] Massé, M. *REST API Design Rulebook*. eng. Place of publication not identified.

[14] Fielding, R. *Architectural Styles and the Design of Network-based Software Architectures*. eng.

[15] *URI Generic Syntax. Berners-Lee, et al.* 1. tammikuuta 2005. URL: https://www.ietf.org/rfc/rfc3986.txt (viitattu 23. 05. 2020).

[16] Lin, B., Chen, Y., Chen, X. ja Yu, Y. Comparison between JSON and XML in Applications Based on AJAX. *2012 International Conference on Computer Science and Service System*. IEEE. 2012, 1174–1177.

[17] Stylos, J., Faulring, A., Yang, Z. ja Myers, B. A. Improving API documentation using API usage information. *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2009, 119–126.

[18] Tulach, J. *Practical API design: Confessions of a Java framework architect*. Apress, 2008.

[19] Henning, M. API design matters. *Communications of the ACM* 52.5 (2009), 46–56.

[20] Helgason, A. F. *Performance analysis of Web Services: Comparison between RESTful & GraphQL web services*. 2017.

[21] *SOAP: Simple Object Access Protocol*. 1999. URL: https://datatracker.ietf.org/doc/html/draft-box-http-soap-00 (viitattu 28. 03. 2022).

[22] *What is SOA?* 2021. URL: https://www.ibm.com/cloud/learn/soa (viitattu 28. 03. 2022).

[23] Soni, A. ja Ranga, V. API features individualizing of web services: REST and SOAP. *International Journal of Innovative Technology and Exploring Engineering* 8.9 (2019), 664–671.

[24] *GraphQL | A query language for your API*. 2022. URL: https://graphql.org/ (viitattu 12. 05. 2022).

[25] Brito, G., Mombach, T. ja Valente, M. T. Migrating to GraphQL: A practical assessment. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2019, 140–150.

[26] *CSV vs JSON for your Data Science Projects*. 2022. URL: https://www.naukri.com/learning/articles/csv-vs-json-for-your-data-science-projects/ (viitattu 24. 01. 2023).

[27] Brito, G. ja Valente, M. T. REST vs GraphQL: A controlled experiment. *2020 IEEE international conference on software architecture (ICSA)*. IEEE. 2020, 81–91.