# Ohjelmointirajapintasimulaation toteutus

Ari Aaltonen

Haaga-Helia ammattikorkeakoulu
Amk-opinnäytetyö
2021
Tradenomin tutkinto

## Tiivistelmä

**Tekijä(t)**
Ari Aaltonen

**Tutkinto**
Tradenomi

**Opinnäytetyön nimi**
Ohjelmointirajapintasimulaation toteutus

**Sivu- ja liitesivumäärä**
24 + 2

Tässä työssä käydään läpi ohjelmointirajapintasimulaation toteutus kahdella eri ohjelmointirajapintasimulaatioratkaisulla. Työn tarkoituksena on selvittää, kuinka ulkoisia ohjelmointirajapintoja simuloidaan valituilla ratkaisuilla. Työn tietoperustassa käydään läpi työn ymmärtämiseksi vaadittuja termejä.

Tutkimusmenetelmä-kappaleessa käydään lyhyesti läpi valittuja ratkaisuja ja niiden ominaisuuksia, minkä jälkeen siirrytään pohjaratkaisun toteuttamiseen simulaatioratkaisuilla. Työssä käydään läpi vaihe vaiheelta pohjaratkaisun toteutus molemmilla simulaatioratkaisuilla, samalla avaten vaiheitten sisältöä.

Tulokset-kappaleessa käydään läpi molempien ratkaisujen simulaatiotiedostojen sisällöt vaihe vaiheelta ja vertaillaan simulaatioratkaisuja keskenään pohjaratkaisun luomassa viitekehyksessä.

Pohdinta-kappaleessa pohditaan työn tuloksia, työn yleispätevyyttä ja siinä parannettavia osuuksia, tämän lisäksi reflektoidaan tekijän omaa oppimista, työn etenemistä ja kohdattuja haasteita.

**Asiasanat**
Simulointi, Yksikkötestaus, Ohjelmointirajapinta

## Sisällys

# 1 Johdanto

Ohjelmistokehityksen projektit jakautuvat useisiin osa-alueisiin. Yleisesti on tiedossa, että ohjelmoijat ohjelmoivat, mutta on tärkeää huomioida, ettei ohjelmoiminen ole yksittäinen käsite vaan se sisältää eri alueita, joista yksi on testaus, ja siihen liittyy olennaisena osana testien kirjoittaminen. Testaus on tärkeä osa ohjelmointia, mutta ainakin Haaga-Helian tradenomi koulutuksessa se on laajalti jätetty huomiotta. Työelämässä testaus tulee vastaan luultavasti heti ensimmäisen tehtävän parissa. Testaus on tärkeä osa laadunvarmistusta, ja testaamisella koitetaan varmistaa, että ohjelmisto toimii odotetulla tavalla.

Osa ohjelmoijista pitää testaamista ja testejä niin tärkeänä, että testaamisen pohjalta on syntynyt oma tapa lähestyä ohjelmistokehitystä. Tämä lähestymistapa on nimeltään ”Test driven development” ja siinä on lyhyesti kuvaillen tarkoituksena aloittaa koodaamalla testit halutusta toiminnallisuudesta, minkä jälkeen ruvetaan kirjoittamaan ohjelmakoodia, jota muokataan siihen asti, että testit toimivat. (Unadkat 2021).

Erilaisten ohjelmointirajapintojen eli APIen (Application Programming Interface) käyttö on jatkuvasti kasvussa ohjelmistokehityksessä. (Bettendorf 2020). Ulkoisten ohjelmointirajapintojen käyttäminen voi luoda ongelmatilanteita koodin testaamisessa. Ulkoiset rajapinnat voivat olla projektiin liittymättömistä syistä saavuttamattomissa. Rajapinnat, joita projektin johto ei itse omista, voivat myös lakata kokonaan olemasta, jolloin suuria osia koodista voidaan joutua kirjoittamaan uudestaan.

Ulkoisten rajapintojen testaamisessa voidaan käyttää erilaisia API- eli ohjelmoitavan rajapinnan virtualisaatio- tai simulointi ratkaisuja. Perehdyn tässä työssä siihen, miten simulointi voidaan toteuttaa kahdella tällaisella ratkaisulla, ja vertaan ratkaisujen eroja, hyviä ja huonoja puolia.

Tutkimukseni tarkoituksena on selvittää, kuinka simuloida ulkoisia ohjelmointirajapintoja valmiilla ratkaisuilla.

Aion käyttää tutkimusmenetelmänä laadullista tutkimusta, vapaasti internetissä saatavilla olevaa aineistoa ja empiirisessä osiossa toteutettavia ratkaisuja. Olen valinnut työssäni kaksi API-simulaatioratkaisua, joilla toteutan empiirisessä osiossa molemmilla pohjaratkaisun. Sen jälkeen vertailen ratkaisujen positiivisia puolia ja mahdollisia puutteita.

Ratkaisuilla toteutettavan ohjelmointirajapintasimulaation pohjaratkaisun tuloksena tulee syntyä REST-ohjelmointirajapintasimulaatio (Representational State Transfer), joka ottaa vastaan POST-pyynnön, jossa on objekti JSON-formaatissa. (JavaScript Object Notation) Ohjelmointirajapintasimulaatio tulkitsee objektin JSON sisältöä ja vastaa JSON:ia tulkinnan pohjalta.

# 2 Tietoperusta

Tässä kappaleessa käydään läpi työhön liittyvää tietoperustaa ja termistöä.

## 2.1 Ohjelmointirajapinta

Stylos, Faulring, Yang ja Myers ovat työssään “Improving API Documentation Using API Usage Information” määritelleet API:n seuraavasti: Ohjelmointirajapinta eli API on käyttöliittymä, joka antaa sitä käyttävälle ohjelmoijalle pääsyn funktionalisuuden kirjastoon. Samassa teoksessa on myös määritelty tarkemmin API:a seuraavin lausein: Ohjelmointirajapinnat eli API:t paljastavat palveluita tai dataa, jota tietokoneohjelma tarjoaa erinäisten reittien lävitse. Nämä reitit sisältävät metodeja, objekteja ja URI:ja. (Stylos, Faulring, Yang, Myers 2009).

## 2.2 URI – (Uniform Resource Identifier)

URI on lyhennelmä Uniform Resource Identifieristä, joka viittaa jonkin resurssin sijaintiin verkossa. URI ei määrittele tapaa, millä resurssiin otetaan yhteyttä tai millä sitä haetaan. (Mealling, Denenberg 2002).

## 2.3 Yksikkötestaus

Runesonin (2006) työssä ”A survey of unit testing practices” James Whittaker on todennut yksikkötestauksen testaavan yksittäisiä ohjelmiston osia tai laajempaa kokoelmaa, joka sisältää useampia osia. Samassa työssä on sanottu yksikkötestauksen tarkoittavan pienimmän yksilöitävän testattavissa olevan moduulin testaamista.

Yksikkötestaus tarkoittaa pienimmän yksilöitävän moduulin testaamista järjestelmässä. Työssä on tehty kysely ohjelmistokehittäjille yksikkötestauksen määritelmästä, ja vastannut joukko ohjelmoijia oli päässyt sopuun seuraavasta toteamuksesta: Yksikkötestauksen pääasiallinen tarkoitus on varmistaa järjestelmän funktionalisuus. Yksikkötestaus ei harkitse mitään ekstrafunktionaalisia osa-alueita, koska se on erotettu ja se ajetaan erillään järjestelmästä. (Runeson 2006).

Yksikkötestaus ei ota huomioon muita osia järjestelmästä kuin ne, jotka testissä testattavan osan testit vaativat. (Runeson 2006). Tämä tarkoittaa omakohtaisen kokemukseni perusteella sitä, että testit voivat olla osana projektia mutta yleensä suoritetaan joko testejä tai ohjelmaa, tavallisesti testit ajetaan ennen ohjelman ajamista, ja niillä on tarkoitus kattaa mahdollisimman suuri osa ohjelman toiminnallisuudesta, jottei ohjelmaa normaalisti käytettäessä jouduttaisi virhetilanteisiin.

### 2.4 Simulointi

Breitenecker ja Troch kuvailevat simulaatiota seuraavasti:
Simulaatio on metodi ongelmanratkaisuun dynaamisissa järjestelmissä, ja se tutkii oikean järjestelmän sijaan mallia oikeasta järjestelmästä. (Breitenecker, Troch 2011)”

Hu kuvailee simulaation käyttötapaa työssään ”Simulaatio-pohjaisen ohjelmistokehityksen metodologiaa jaetuille reaaliaikaisille järjestelmille” seuraavalla tavalla: Simulaatiota käytetään yleensä kahdessa vaiheessa, analyysivaiheessa konseptin kehittämisen tukena ja testausvaiheessa. Simulaatio tarjoaa testausympäristöjä ja testitapauksia järjestelmän verifiointiin ja arviointiin. Käyttäen simulaatioteknologioita analyysivaiheessa voidaan saada tärkeää tietoa mahdollisista ongelmista tarpeeksi aikaisessa vaiheessa kehitystä, jolloin niihin voidaan vastata kustannustehokkaasti. (Hu 2004).

Simulointi omakohtaisen kokemuksen perusteella tässä tarkoittaa, että simulointiratkaisu matkii oikean järjestelmän toimintaa tarpeeksi hyvin, niin ettei oikeaa järjestelmää tarvitse kuormittaa testaamisella.

### 2.5 JSON - (JavaScript Object Notation)

ECMA:n määritelmässä "The JSON Data Interchange Syntax" kuvaillaan JSON:ia seuraavasti: JSON on tekstisyntaksi, joka kehitettiin datan välittämiseen ohjelmointikieleltä toiselle riippumatta ohjelmointikielien yhteensopivuudesta.

JSON:n syntaksi koostuu suluista, aaltosuluista, hakasuluista, puolipisteistä ja pilkuista.

```
{ "continue":"yes" }
```

Kuva 1 Kuvassa on esimerkki yksinkertaisesta JSON-objektista

JSON tarjoaa yksinkertaisen merkintätavan ilmaista kokoelmia/listoja ja nimi/arvopareja. Hyväksymällä ja käyttämällä JSON:n yksinkertaista merkintätapaa, monimutkaisia datarakenteita voidaan helposti vaihtaa eri ohjelmointikielien ja ohjelmien kesken.

```
{
    "nimi": "Matti",
    "ikä": 30,
    "kissat": [
        {
           "nimi": "Tiina"
        },
       {
             "nimi”: "Misu"
       }
    ]
}
```

Kuva 2 Kuvassa on esimerkki monimutkaisemmasta JSON-objektista

Yksi JSON:in vahvuuksista on sen yksinkertaisuus, jonka seurauksena sen syntaksin ei oleteta muuttuvan ikinä.

JSON on kevyt, tekstipohjainen, ohjelmointikielestä riippumaton syntaksi datan vaihdon formaatin määrittelyyn. (ECMA-404 2017).

### 2.6 REST – (Representational State Transfer)

REST on arkkitehtuurinen tyyli webpalveluiden luomiseen, jonka perustana toimivat standardisoidut operaatiot, esimerkiksi get, modify, create ja delete. REST-palvelun käyttäjä käyttää palvelua URI:n kautta, ja palvelu palauttaa resurssin, joka vastaa URI:a. REST ei vaadi erillistä viitekehystä vaan se toimii HTTP:llä ja Uniform Resource Identifiereilla eli URI:lla. (Widmann 2006)

RESTissä edellä mainittuja standardisoituja operaatioita käytetään GET-, POST-, PUT- ja DELETE-pyynnöillä. GET hakee sisältöä, POST luo tai päivittää sisältöä, PUT päivittää sisältöä ja DELETE poistaa sisältöä.

REST:iä noudattavia järjestelmiä kutsutaan RESTful-järjestelmiksi, ja niiden suurin tunnusmerkki on asiakkaan eli käyttäjän ja palvelimen/palvelun huolien erottaminen toisistaan. Tämä mahdollistaa modulaarisen järjestelmän luomisen, jossa palvelimen koodia voidaan vaihtaa vapaasti sen vaikuttamatta palvelun käyttäjään ja palvelun käyttäjien käyttämään koodiin voidaan tehdä muutoksia sen vaikuttamatta palvelimeen. (Booth, Haas, Newcomer, Champion, Ferns, Orchard 2004).

Toinen RESTful-järjestelmää määrittelevä tunnusmerkki on palvelimen tilattomuus, eli palvelimen ei tarvitse tietää asiakkaan tilasta mitään ja toisinpäin. Tämä aiheuttaa sen, että asiakas ja palvelin voivat ymmärtää saamansa viestit tietämättä aikaisemmista viesteistä. (Codecademy 2021)

RESTiä noudattavan järjestelmän tulee palauttaa verkkoresurssinsa tekstimuodossa ja sallia niiden lukeminen ja muokkaus tilattomalla protokollalla, mikä tarkoittaa, että palvelin ei säilytä pyynnöstä tietoja itselleen ja pyynnön tulee olla itse sisällöstään palvelimelle selvä ilman aikaisempia pyyntöjä. (Booth ym. 2004).

### 2.7 YAML – (YAML Ain't Markup Language)

YAML on datan sarjallistamiseen luotu kieli, joka on suunniteltu mahdollisimman ihmisluettavaksi ja toimimaan paremmin modernien ohjelmointikielien kanssa. YAML luotiin erityisesti toimimaan hyvin yleisien käyttötapauksien kanssa muun muassa asetustiedostoissa, lokitiedostoissa ja datan siirtämisessä ohjelmointikielien välillä. (Ben-Kiki, Evans, döt Net 2009).

JSON ja YAML molemmat tähtäävät olemaan ihmisluettavia datan siirtämisformaatteja, jokainen validi JSON tiedosto on myös validi YAML tiedosto, mutta ei toisinpäin. Tämän seurauksena YAMLia voi pitää JSONin ylijoukkona. JSON edellyttää RFC4627 internet standardin mukaisesti, että avaimet ovat uniikkeja, kun YAML taas vaatii niiden olevan ainutlaatuisia. YAML käsittelee toistuvia avain arvoja virheinä, jonka takia ainoastaan JSON tiedostoja, jotka sisältävät ainoastaan ainutlaatuisia avain arvoja ovat valideja YAML tiedostoja. (Ben-Kiki, Evans, döt Net 2009).

### 2.8 HTTP – (Hypertext Transfer Protocol)

Hypertext Transfer Protocol on sovellustason request/response eli pyyntö/vastaus protokolla, asiakas lähettää pyynnön palvelimelle lomakkeen muodossa, joka sisältää HTTP metodin, URI:n ja protokollan version, tämän lisäksi MIME-tyyppisen (Multipurpose Internet Mail Extensions) viestin, joka sisältää pyynnön muuttujat, asiakkaan tiedot ja pyyntöön on mahdollista sisällyttää dataa pyynnön rungossa. Palvelin vastaa pyyntöön HTTP status rivillä, joka sisältää viestin protokollan version ja HTTP status koodin, joka vastaa onnistunutta pyyntöä tai virhettä. Tämän lisäksi vastauksessa on MIME-tyyppinen viesti, joka sisältää palvelimen tiedot, metadataa ja mahdollisesti rungossa dataa asiakkaalle. (Fielding, Gettys, Mogul, Frystyk, Masinter, Leach & Berners-Lee 1999).

HTTP kommunikaatio tapahtuu yleensä TCP/IP yhteyden ylitse, TCP portissa 80 mutta muitakin portteja voidaan käyttää. HTTP sallii avoimen sarjan metodeja ja otsikoita, joilla osoittaa pyynnön tarkoitus. HTTP käyttää URI:a, URL:ia tai URN:ia, jolla osoitetaan mihin resurssiin metodia tulisi soveltaa. Viestit toimitetaan formaatissa, joka on samanlainen kuin jota sähköposteissa käytetään. (Fielding ym. 1999).

### 2.9 curl – (Client URL)

Curl on komentoriviltä käytettävä työkalu, jolla voi hakea tai lähettää tiedostoja käyttäen URL syntaksia. curl käyttää libcurlia, minkä takia curl tukee myös kaikkia libcurlin ominaisuuksia ja toiminallisuuksia. Libcurl on siirettävissä oleva kirjasto ominaisuuksia ja toiminallisuuksia, joka tarjoaa käyttäjälle helpon käyttöliittymän yleisien internet-protokollien käyttämiseen. (Curl s.a.).

### 2.10 Komentorivi

Komentorivillä tarkoitetaan käyttöliittymää, jota käytetään kirjottamalla komentoja komentoriville, hiiren käyttämisen sijaan. Komentorivillä käyttäjä voi suorittaa pelkästään näppäimistöä käyttäen melkein kaikki samat tehtävät kuin graafisella käyttöliittymällä hiirtä käyttäen. Komentorivillä monet tehtävät voidaan tosin suorittaa nopeammin, ja niitä voidaan automatisoida helpommin. Minkä takia tehtäviä voi suorittaa helposti etäyhteydellä tietokoneeseen pelkkää komentoriviä käyttäen. (Computer Hope 2020).

# 3 Tutkimusmenetelmä

## 3.1 API Simulator

Valitsin yhdeksi ratkaisukseni tässä tutkimuksessa API Simulator -ohjelman, jonka pääpainotus on HTTP-rajapinnoissa, joita kutsutaan myös Web API -nimellä.

API Simulaatiot ovat oikeiden rajapintojen korvikkeita testaamisessa, jotka imitoivat oikean rajapinnan vastauksen dataa ja käytöstä, oli se aito rajapinta tai vielä suunnitelmatasolla oleva konsepti. Rajapinnan simulaatio API Simulaattorilla koostuu sarjasta ”simlettejä”, API Simulaattorin simletti simuloi HTTP vastausta HTTP pyyntöön. (API Simulator 2021).

API Simulaattori tutkii pyyntöjä vertaamalla niitä pyyntöjä vastaaviin sääntöihin, tämän pohjalta se yhdistää oikeat pyynnöt oikeisiin simletteihin, jotka puolestaan generoivat simuloidun vastauksen pyynnölle. Ohjelma tarjoaa kaksi vaihtoehtoa käyttäjälle, jos pyyntö ei vastaa yhtään simlettiä, käyttäjä voi määrittää oletus simletin käytettäväksi tai ohjelma voi lähettää tunnistamattomat pyynnöt eteenpäin niiden oikeaan määränpäähän ja palauttaa vastauksena aidon datan oikeasta määränpäästä. (API Simulator 2021).

Simletti voi vastata pyyntöihin kahdella eri tavalla, joko staattisilla määritetyillä vastauksilla, joiden sisältö ei muutu eri pyyntöjen saapuessa kyseiseen rajapintaan tai dynaamisilla vastauksilla, jotka voivat reagoida pyynnön sisältöön ja muodostaa vastauksen riippuen siitä mitä sen omiin määritelmiin on tallennettu vastaukseksi esimerkiksi tiettyyn aikaan saapuviin pyyntöihin tai tietyn lähettäjän toimittamiin pyyntöihin. API Simulaattori tukee myös keinotekoista viiveen luontia pyynnön ja vastauksen välille, jäljitelläkseen vielä realistisemmin rajapintojen käyttäytymistä. (API Simulator 2021).

API-Simulator siis ottaa vastaan HTTP protokollalla tehtyjä pyyntöjä, ja vastaa niihin esimerkkidatalla myös HTTP protokollalla. API-Simulator on koittanut vähentää rajapintojen simulointiin riippuvaa ohjelmointia mahdollisimman vähän, ja se toimiikin luoden simlettejä määrittelytiedostojen pohjalta, ohjelma ei myöskään vaadi erillistä web-palvelinta pyöriäkseen vaan sen voi asettaa pyörimään paikallisesti omalla koneella. (API Simulator 2021).

### 3.2 Hoverfly

Toiseksi ratkaisuksi olen valinnut avoimeen lähdekoodiin perustuvan ratkaisun nimeltään Hoverfly. Hoverfly kykenee toiminnallisuudeltaan paljon monimutkaisempien toteutuksien simulointiin ja suurta osaa sen ominaisuuksista ei käytetä tässä työssä. Hoverfly:ta käytetään komentoriviltä tai terminaalista hoverctl nimisellä sovelluksella, joka tulee Hoverflyn asennuksen mukana. (Hoverfly 2021).

Hoverfly:ssä on useita eri käyttötiloja, käytin tässä työssä ainoastaan ”Kaappaustilaa” ja ”Simulointitilaa”, jonka takia esittelen ainoastaan näiden tilojen perusteet. Kaappaustilassa Hoverfly toimii välityspalvelimena, käyttäjän tulee ohjata pyyntönsä Hoverflyn kautta, jotta Hoverfly voi kaapata pyynnön vastauksen, Hoverfly luo pyynnön vastauksesta automaattisesti tarkan simulaation, jota voi muokata jälkeenpäin itse sen tiedostosta. Simulointitilassa Hoverfly toimii palvelimena, ja simuloi ulkoista rajapintaa, kun käyttäjä ilmoittaa sille minkä tiedoston pohjalta rajapintaa tulee simuloida. Simulaatioita voi luoda käsin tai käyttäen kaappaustilaa. (Hoverfly 2021).

Hoverflyn simulaatiot käyttävät rakenteenaan JSON:ia, ja niitä on suhteellisen helppo ihmisen ymmärtää ilman jatkuvaa dokumentaation tulkitsemista. Simulaatiot käyttävät ”pyyntö ja vastaus” pareja pyyntöjen ohjaamiseen oikean simulaation luo. (Hoverfly 2021)

Ratkaisujen suurin ero toisiinsa on se, että Hoverfly perustuu avoimeen lähdekoodiin, kun taas API Simulator on suljetun lähdekoodin ratkaisu. API Simulatorin simulaatioissa käytetään syntaksina YAML:ia (YAML Ain’t Markup Language) kun Hoverfly käyttää JSON:ia. (JavaScript Object Notation)

Molempien simulaattorien simulaatiotiedostoissa määritellään HTTP metodi, johon simulaatio vastaa. Käytin tässä työssä POST:ia simulaatioissa, POST:ia käytetään yleensä tiedon lähettämiseen, kun GET:iä käytetään tiedon hakemiseen. Simulaatioiden lopputulos tulee olemaan identtinen toiminnaltaan, toteutustapa vain vaihtelee riippuen simulaatioratkaisusta.

Rajapintojen simulaation pääasiallinen tarkoitus on vähentää testaamiseen tarvittavia riippuvuuksia ohjelmalta, jolloin ohjelmaa ja sen osia voidaan kokeilla riippumatta ulkoisten rajapintojen saatavuudesta tai muista ongelmista. Ohjelmaa voi myös käyttää paikallisesti ilman verkkoyhteyttä rajapintojen ollessa simuloituina paikallisesti, voi ohjelman kehittämistä ja testaamista silti jatkaa verkkoyhteyden ollessa toimimattomassa tilassa.

# 4 Pohjaratkaisun toteuttaminen simulaatioratkaisuilla

## 4.1 Pohjaratkaisun määrittely

Ratkaisulla toteutettava pohjaratkaisu otti vastaan toisen seuraavista JSON-objekteista osana POST pyyntöä.

```
{"continue": "yes"}
{"continue": "no"}
```

Kuva 3 Kuvassa on JSON-objekteina esimerkit lähetettävistä arvoista

Simulaatiot tulkitsivat pyyntöön sisälletyn JSON-objektin sisällön, ja palauttivat vastauksena HTTP tilakoodin 200, joka tarkoittaa pyynnön onnistuneen ja toisen seuraavista JSON-objekteista riippuen pyynnön JSON-objektin sisällöstä.

```
{"requestState": "Continuing"}
```

Kuva 4 Kuvassa on JSON-objekti vastauksesta pyynnön sisältäessä arvon ”yes”

```
{"requestState": "Done"}
```

Kuva 5 Kuvassa on JSON-objekti vastauksesta pyynnön sisältäessä arvon ”no”

Simulaatio toteutettiin molemmilla ratkaisuilla Microsoft Windows 10 ympäristössä.

## 4.2 Hoverfly toteutus

Aloitin työn toteuttamalla simulaation Hoverfly simulaatioratkaisua käyttämällä, latasin Hoverflyn tiedostot sen omalta GitHub sivulta julkaisuista. Siirsin tiedostot Windowsin system32 kansioon, jotta pystyin käyttämään niitä suoraan Windowsin komentoriviltä.

Ensimmäinen vaihe oli esimerkkisimulaation luominen, Hoverflyssa on kaappaustila, joka on tehty simulaatioiden luomisen helpottamiseksi. Kaappaustilassa ohjelma asetetaan toimimaan välityspalvelimena, joka tarkoittaa verkkoliikenteen ohjaamista sen kautta internettiin. Pyynnön kulkiessa ohjelman läpi, ohjelma kirjaa ylös pyynnön tiedot ja luo simulaation vastauksen pohjalta, tätä simulaatiota on helppo tulkita ja muokata manuaalisesti luomisen jälkeen. Hoverfly käynnistettiin kaappaustilaan syöttämällä komentoriville seuraavat komennot järjestyksessä:

```
hoverctl start
hoverctl mode capture
```

Kuva 6 Kuvassa on komennot, joilla avataan Hoverfly ja asetetaan ohjelma kaappaus tilaan

Tämän jälkeen lähetin curl pyynnön Hoverflyn läpi seuraavalla komennolla

```
curl -X POST -H "Content-Type: application/json" -d "{ \"continue\": \"yes\" }" --proxy http://localhost:8500/ http://jsonplaceholder.typicode.com/posts
```

Kuva 7 Kuvassa on curl komento, jolla lähetetään POST-pyyntö komennossa annettuun osoitteeseen käyttäen välityspalvelinta vastauksen vastaanottamiseen portissa 8500

Komennossa asetetaan tässä pyynnössä metodiksi POST. Pyynnössä määritellään ”Content-Type” otsikko vastaamaan arvoa ”application/json”, jotta määränpää osaa tulkita pyynnön sisällön oikein. Dataksi on laitettu JSON muodossa {"continue": "yes"} mutta Windows ympäristössä ”-merkit on täytynyt edeltää \-merkillä, jotta komento tulkitaan oikein. Pyynnössä toiseksi viimeiseksi on määritelty välityspalvelin eli proxy asetus, jossa pyyntö on laitettu kulkemaan ”http://localhost:8500/” läpi, jossa Hoverfly on käynnissä kaappaustilassa. Lopuksi on pyynnön määränpää, joka tässä tapauksessa on ” http://jsonplaceholder.typicode.com/posts”, sivu itse kuvaa itseään seuraavasti: ”Free Fake REST API”, osoite palauttaa takaisin pyynnön sisällön ja lisää siihen id kentän. Tämä ei vastannut simulaatiomme haluttua lopullista tulosta, jonka takia jouduin muokkaamaan vastauksen manuaalisesti simulaation luonnin jälkeen.

```
hoverctl export simulation.json
```

Kuva 8 Kuvassa on komento, jolla tallennetaan simulaatio tiedostoon

Tämä komento tallensi simulaation tiedostoon simulation.json, jonka jälkeen suljin ohjelman kaappaustilasta komennolla

```
hoverctl stop
```

Kuva 9 Kuvassa on komento, jolla suljetaan Hoverfly

Näin luotiin simulaatio Hoverflyn kaappaustilalla, tämän jälkeen muokkasin simulaation tiedostosta ylimääräisiä otsikoita pois ja vaihdoin sisällön vastaamaan haluttua lopputulosta.

Seuraavaksi Hoverfly tarvitsi käynnistää web-palvelimena, jolloin sen tila on oletuksena simulointi. Tämä onnistui komennolla

```
hoverctl start webserver
```

Kuva 10 Kuvassa on komento, jolla Hoverfly avataan web-palvelin tilassa

Käynnistyksen jälkeen ohjelmalle tarvitsi määritellä tiedosto, jota simuloida. Annoin aikaisemmin luodun simulaatiotiedoston komennolla

```
hoverctl import simulation.json
```

Kuva 11 Kuvassa on komento, jolla Hoverflylla avataan valmis simulaatiotiedosto käyttöön

Tämän jälkeen pystyin testaamaan simulaation toimivuutta muokkaamalla aikaisemmin käytettyä curl komentoa niin, että se osoitti omalle koneelleni porttiin, jossa Hoverfly oli palvelintilassa käynnissä.

```
curl -X POST -H "Content-Type: application/json" -d "{ \"continue\": \"yes\" }" http://localhost:8500/posts
```

Kuva 12 Kuvassa on curl komento, jolla testataan simulaation toimintaa arvolla ”yes”

Pyynnöstä pystyi muokkaamaan suoraan continue:lle arvon ”no”, jonka seurauksena pystyin testaamaan molemmat variaatiot pyyntöjen vastauksista.

```
curl -X POST -H "Content-Type: application/json" -d "{ \"continue\": \"no\" }" http://localhost:8500/posts
```

Kuva 13 Kuvassa on curl komento, jolla testataan simulaation toimintaa arvolla ”no”

```
C:\Users\Jozhua>curl -X POST -H "Content-Type: application/json" -d "{ \"continue\": \"yes\" }" http://localhost:8500/po
sts
{
  "requestState": "Continuing"
}
C:\Users\Jozhua>curl -X POST -H "Content-Type: application/json" -d "{ \"continue\": \"no\" }" http://localhost:8500/pos
ts
{
  "requestState": "Done"
}
```

Kuva 14 Kuvassa näkyy, että simulaatio toimii oikein molemmilla komennoilla

Komentojen palautteesta kykenin päättelemään, että simulaatiot toimivat oikein.
Lopuksi ohjelma suljettiin komennolla

```
hoverctl stop
```

Kuva 15 Kuvassa on komento, jolla Hoverfly suljetaan

### 4.3 API Simulator toteutus

Seuraavaksi siirryin toteuttamaan vastaavaa simulaatiota API Simulator nimisellä ratkaisulla.

API Simulaattorin sivuilla on lataus, joka sisältää API Simulaattorin tiedostot ja sen lisäksi suuren määrän esimerkkejä, joista katsoa mallia simulaatioita tehdessä. Huomattavaa tässä oli, että API Simulator käyttää Javaa, jonka seurauksena sen käyttö ei onnistu ilman Javan asentamista, ja JAVA_HOME ympäristömuuttujan asettamista.
API Simulaattoria käytetään myös komentoriviltä tai terminaalista, avasin API Simulaattorin kansion sisältä aktiiviseksi bin kansion, jossa sijaitsee API Simulaattorin käytettävät tiedostot apisimulator, apirecorder ja apiclient – näistä tiedostoista tarvitsee käyttää vain apisimulatoria.

Lähdin liikkeelle API Recorderia käyttämällä, sillä on kaksi toimintatilaa proxy ja router, mutta käytämme ainoastaan proxy eli välityspalvelin tilaa. API Recorder käynnistettiin komennolla

```
apirecorder start -m proxy -p 6090
```

Kuva 16 Kuvassa on komento, jolla avataan apirecorder välityspalvelin tilassa portissa 6090

-m proxy parametrilla määriteltiin ohjelma käynnistymään välityspalvelin tilassa, ja -p parametrilla määriteltiin porttinumero, jota ohjelma kuuntelee. Seuraavaksi tuli lähettää pyyntömme apirecorderin läpi komennolla

```
curl -X POST -H "Content-Type: application/json" -d "{ \"continue\": \"yes\" }" --proxy http://localhost:6090/ http://jsonplaceholder.typicode.com/posts
```

Kuva 17 Kuvassa on curl komento, jolla lähetetään POST-pyyntö komennossa annettuun osoitteeseen käyttäen välityspalvelinta vastauksen vastaanottamiseen portissa 6090

apirecorder loi pyynnöstä ja sen vastauksesta tiedoston, joka sisältää kaikki tiedot mitä lähetettiin ja vastaanotettiin, tiedostosta poimin olennaiset osaksi simulaatiotamme ja siirsin ne omaan tiedostoon.

API Simulator suosii kansiorakennetta, jossa simulaatiotiedostot tulee pistää simlets nimiseen kansioon simulaatiolle luodun kansion sisällä.

```
simulaation_kansio\simlets\simulaation_nimi.yaml
```

Kuva 18 Kuvassa on API Simulatorin käyttämä kansiorakenne ilmaistuna tiedostopolkuna

Simulaatiot tehdään API Simulaattorin omalla DSL (Domain-Specific Language) muodolla, joka perustuu YAML:iin. Simulaatioiden tekemisen apuna löytyy API Simulaattorin sivuilta API Simulator Cloud Studio:n alfa versio, sivu toimi hyvin hitaasti, mutta sivulla sijaitseva Simlet Editor auttoi simulaatiotiedoston syntaksin kanssa. Huomattavaa tässä vaiheessa oli, että simulaation kirjoittaminen oli työlästä ja jouduin tulkitsemaan API Simulaattorin asennuksen mukana tulleita esimerkkejä päästäkseni haluttuun lopputulokseen.

Simulaation testaamiseksi se tuli käynnistää API Simulaattorilla komennolla

```
apisimulator start ”tiedostopolku/simulaation_pohja_kansioon”
```

Kuva 19 Kuvassa on komento, jolla avataan API Simulator käyttäen komennossa määriteltyä simulaatiota

API Simulaattorin oletusportiksi on asetettu 6090, joten simulaatiolle lähetettävät kutsut tuli ohjata porttiin 6090, kuten tässä komennossa on tehty.

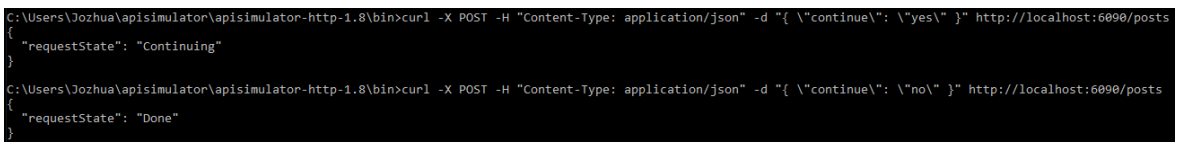
```
curl -X POST -H "Content-Type: application/json" -d "{ \"continue\": \"yes\" }" http://localhost:6090/posts
```

Kuva 20 Kuvassa on curl komento, jolla testataan simulaation toimintaa arvolla ”yes”

API Simulaattorin vastaus tuli suoraan takaisin komentoriville, testasin saman tien toisella sisällöllä.

```
curl -X POST -H "Content-Type: application/json" -d "{ \"continue\": \"no\" }" http://localhost:6090/posts
```

Kuva 21 Kuvassa on curl komento, jolla testataan simulaation toimintaa arvolla ”no”




Kuva 22 Kuvassa näkyy, että simulaatio toimii oikein molemmilla komennoilla

Vastauksista näki, että simulaatio toimii oikein, ja vastaus pyyntöihin oli käytännöllisesti katsoen identtinen Hoverfly simulaatioratkaisun vastauksiin.

# 5 Tulokset

Tutkimuksen tuloksena syntyi kaksi simulaatiotiedostoa, toinen Hoverfly:lle ja toinen API Simulaattorille. Tässä osiossa pyrin avaamaan tiedostojen sisältöä, jotta työ mahdollistaisi työn lukijan toteuttamaan tutkimuksen myös itse.

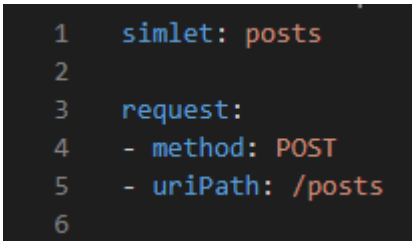
## 5.1 API Simulatorilla käytettävä simulaatiotiedosto

Liite 1 sisältää kuvan API Simulatorin simulaatiotiedostosta kokonaisuudessaan

API Simulaattorilla simulaation kansiorakenteen tuli noudattaa tiettyä rakennetta, simulaation niminen kansio, tässä tapauksessa posts, jonka sisällä tuli olla simlets kansio, johon simulaatiotiedosto simlet.yaml sijoitettiin.

| posts\simlets\simlet.yaml |
|---|

Kuva 23 Kuvassa on simulaation kansiorakenne tiedostopolkuna




Kuva 24 Kuvassa on simulaatiotiedoston pyynnön eli requestin metodin ja uriPath:in määrittely

Simulaatiotiedoston alussa määritellään simletin nimeksi ’posts’, jonka jälkeen määritellään mihin HTTP metodiin eli pyyntöön simulaatio reagoi. Tässä työssä käytimme ainoastaan POST pyyntöä, ja otimme vastaan pyynnön ainoastaan /posts osoitteessa.

```
responses:
- when:
    request:
    - where: body
      element: ".continue"
      equals: "yes"
  from: template
  status: 200
  body: `
{
  "requestState": "Continuing"
}
`
```

Kuva 25 Kuvassa on simulaatiotiedoston ensimmäisen vastauksen määrittely

Tämän jälkeen tiedostossa tulee määritellä responses osio, eli vastaukset pyyntöön koska API Simulator perustuu Request/Response pareihin. Aloitamme luomalla vastauksen pyyntöön, silloin kun pyynnön mukana on tullut datana JSON-objekti, jonka sisällä olevan kentän continue arvo on ”yes”, simulaatio vastaa HTTP status koodin 200, joka vastaa onnistunutta pyyntöä ja lähettää samalla takaisin JSON-objektin

```
{
        ”requestState”: ”Continuing”
}
```

Kuva 26 Kuvassa on JSON-objekti, jonka simulaatio lähettää takaisin arvon ollessa ”yes”

```
- when:
    request:
    - where: body
      element: ".continue"
      equals: "no"
  from: template
  status: 200
  body: `
{
  "requestState": "Done"
}
`
```

Kuva 27 Kuvassa on simulaatiotiedoston toisen vastauksen määrittely

Tämä osio on myös osa responses osaa tiedostosta, tässä mukaillaan ensimmäisen vastauksen määrittelyä muuten, tätä vastausta käytetään kun continuen arvo on ”no” ja pyyntöön vastataan JSON-objektilla

```
{
        ”requestState”: ”Done”
}
```

Kuva 28 Kuvassa on JSON-objekti, jonka simulaatio lähettää takaisin arvon ollessa ”no”

```
32  - from: template
33    template: Simula
34    status: 400
35    body: `
36  {
37      "message": "Bad request"
38  }
39  `
```

Kuva 29 Kuvassa on simulaatiotiedoston muissa tapauksissa käytettävän osan määrittely

Viimeisenä tiedostossa on pohjalla osio, jota sovelletaan pyyntöihin, jotka eivät vastaa ylempänä olevia määritelmiä. Osio palauttaa HTTP status koodin 400, joka vastaa ”Bad Request:ia” ja liittää vastaukseen JSON-objektin

```
{
        ”message”: ”Bad request
}
```

Kuva 30 Kuvassa on JSON-objekti rajapinnan vastauksesta, kun pyynnön sisältö ei vastaa kumpaakaan odotetuista muodoista

### 5.2 Hoverflylla käytettävä simulaatiotiedosto

Liite 2 sisältää kuvan Hoverflyn simulaatiotiedostosta kokonaisuudessaan


Kuva 31 Kuvassa on simulaatiotiedoston request osan pyynnön metodin, sisällön ja polun määrittely

Simulaatiotiedosto aloitetaan luomalla data JSON-objekti, jonka sisälle määritellään kaikki muut osat tiedostosta, objektin sisältö aloitetaan luomalla ”pairs” niminen JSON-objekti, jonka sisältö koostuu request/response pareista. Request objektin sisälle luodaan polku johon simulaatio reagoi, tämän jälkeen metodi ”POST”, koska tämä tiedosto on luotu Hoverflyn kaappaustilalla, mukaan on tallennettu alkuperäisen pyynnön päämäärä. Pyynnön sisällön tulee vastata JSON-objektia

```
{
        ”continue”: ”yes”
}
```

Kuva 32 Kuvassa on JSON-objekti lähetettävän pyynnön ensimmäisen vaihtoehdon sisällöstä


Kuva 33 Kuvassa on ensimmäisen request parin vastauksen eli responsen määrittely

Tässä määritellään simulaatio vastaamaan requestia vastaavaan pyyntöön HTTP status koodilla 200, joka tarkoittaa ”OK” tilaa, samalla lähetetään vastauksessa JSON-objekti muodossa

```
{
        ”requestState”: ”Continuing”
}
```

Kuva 34 Kuvassa on JSON-objekti ensimmäisen vaihtoehdon pyynnön odotetusta vastauksesta


Kuva 35 Kuvassa on ”pairs” objektin toisen request objektin määrittely

Samoin kuin ensimmäisessä, niin tässä määritellään pyynnön polku, metodi, määränpää ja sisältö, tässä pyynnön sisällön tulee vastata JSON-objektia

```
{
        ”continue”: ”no”
}
```

Kuva 36 Kuvassa on JSON-objekti lähetettävän pyynnön toisen vaihtoehdon sisällöstä


Kuva 37 Kuvassa on toisen request parin vastauksen eli responsen määrittely

Tässä jälleen määritellään request objektissa määriteltyä pyyntöä vastaavan vastauksen status koodi ja sisältö, jossa lähetetään vastauksena JSON-objekti muodossa

```
{
        ”requestState”: ”Done”
}
```

Kuva 38 Kuvassa on JSON-objekti toisen vaihtoehdon pyynnön odotetusta vastauksesta


Kuva 39 Kuvassa on tiedoston lopusta löytyvät Hoverflyn kaappaustilan automaattisesti luomat tiedot

Simulaation pohja luotiin käyttäen Hoverflyn kaappaustilaa, ja sen sisältöä muokattiin vastaamaan haluttua lopputulosta jälkikäteen, minkä takia tiedoston lopusta löytyy automaattisesti luotuja kenttiä, joista ainakin ”meta” JSON-objekti on pakollinen osa Hoverflyn simulaatiotiedostoa.

### 5.3 Simulaatioratkaisujen vertailua

Tutkimuksen perusteella vertailen simulaatioratkaisujen Hoverfly ja API Simulator hyviä ja huonoja puolia tässä työssä esitetyn ja toteutetun pohjaratkaisun viitekehyksessä.

Hoverflyssa oli positiivista sen helppo käyttö ja asennus, ladatessasi Hoverflyn, tiedostossa tulee mukana kaikki tarvittavat tiedostot, eikä ohjelmalla ole ulkoisia riippuvuuksia.

API Simulatorissa taas latauksessa tulevat tiedostot eivät riitä ohjelman pyörittämiseen, koska ratkaisuita pyöritetään Javalla, minkä seurauksena ohjelma vaatii Javan asentamisen ja JAVA_HOME ympäristömuuttujan asettamisen käyttöympäristössä.

Hoverflyn kaappaustila on todella hyvä, kaappaustila loi tarkan toimivan simulaation sen läpi lähetettyjen pyyntöjen pohjalta, muutoksia ei välttämättä tarvitse siis edes tehdä, jos rajapinta on oikeasti jo olemassa. Simulaatioita pystyi myös muokkaamaan suhteellisen helposti kaappaustilan luoman simulaation pohjalta, jonka takia myös keskeneräisiä rajapintoja voi simuloida.

API Simulatorissa oli myös vastaava toiminto, mutta se tallensi raakadatana pyynnön kaikki tiedot ja vastauksen tiedostoon, eikä luonut simulaatiota datan pohjalta suoraan.

Hoverfly on avoimen lähdekoodin ratkaisu, jonka takia sitä voi parantaa ja muokata vastaamaan omaa käyttötarvetta, myös dokumentaatiota ja ohjeita ohjelman käyttämisestä löytyi hyvin netistä.

API Simulator on suljetun lähdekoodin ratkaisu, joka saattoi olla osasyy siihen, ettei ohjelmaan löydy helposti apua, eikä sitä miten asiat ovat toteutettu voi itse katsoa lähdekoodista.

Hoverflyssa simulaation ajamiseen riittää, että ohjelma avataan simulaation kansiosta komentorivillä, jonka jälkeen simulaatio avataan ohjelmassa import komennolla.

API Simulator vaatii simulaatioiltaan tietyn kansiorakenteen toimiakseen, simulaatiot avataan kansion perusteella, pelkän simulaatiotiedoston sijaan.

Näiden kohtien ulkopuolella simulaatioratkaisut ovat melko samanlaisia, ja molemmilla päästiin haluttuun toiminnallisuuteen, joka määriteltiin pohjaratkaisun määritelmässä.

# 6 Pohdinta

Tuloksissa keskitytään enimmäkseen havaintoihin, jotka on tehty rajapintasimulaatioratkaisuiden käytössä pohjaratkaisua toteutettaessa. Simulaatioiden luonti on edelleen hyvin manuaalista, simulaatioiden luomisen mahdollinen automatisointi tuli välttämättä mieleen useasti simulaatioratkaisutiedostoja tehdessä. Suurin aika tutkimuksessa menikin kyseisten tiedostojen sisällön muokkaamiseen pohjaratkaisua vastaavaksi, tämä ei työn sisällöstä selviä millään tavalla vaan kuittasin tämän osan työstä muutamassa sivulauseessa.

Tutkimuksen tulokset jäivät hyvin suppeiksi, koska pohjaratkaisu käytti niin pientä osaa valittujen ratkaisujen ominaisuuksista. Tämän tutkimuksen pohjalta ei voi tehdä laajemmin päteviä vertailuja/kriteeristöjä simulaatioratkaisuista, vaan tulokset rajoittuvat pohjaratkaisun luomaan viitekehykseen. API Simulator on niin geneerinen nimi, että minkään muun dokumentaation löytäminen ratkaisulle sen oman dokumentaation lisäksi osoittautui hyvin hankalaksi. Suurin haaste työssä oli tuloksien määritelmä, alkuperäisesti työstä piti muodostua tulokseksi yhtenäinen ohje ratkaisujen pohjalta, mutta työtä tehdessä selvisi, ettei ole mahdollista yhdistää simulaatioratkaisujen ohjeita yhdeksi, joten tulostavoite muuttui vertailuksi. Vertailu on aika suppea, mutta mielestäni työssä on syntynyt riittävä ohjeistus, että vähemmänkin tehnyt osaisi sitä noudattamalla toistaa työn lopputuloksen.

Aiheeseen liittyvää ja sopivaa tieteellistä kirjallisuutta oli hyvin haastavaa löytää. Työssä ei käsitelty millään tasolla vaihtoehtoja ohjelmointirajapintasimulaatiolle, tämä jäi työstä kokonaan pois liian tiukan aikataulun takia. Työn aikatauluksi muotoutui Projektisuunnitelman palautus 19.3.2021 ja opinnäytetyön tuli olla valmis 12.5.2021, tämä oli täysipäiväisen työn ohessa liian tiukka aikataulu, jonka takia työ jäi suppeaksi. Työtä tehdessä opin itse alkeistason käytön valituista simulaatioratkaisuista, YAML oli tullut vain muutaman ohjelman asetustiedostossa aikaisemmin vastaan. Suurin parannuskohde olisi tutkimuksen ja sen tavoiteltujen tulosten määrittely tarkemmin. Parhaiten työssä onnistui tutkimuksen ja sen tuloksena syntyneiden tiedostojen dokumentaatio.

## Lähteet

API Simulator. Basic concepts. Luettavissa: https://apisimulator.io/docs/1.8/getting-started/basic-concepts.html. Luettu: 18.4.2021 15:18

Ben-Kiki, O. Evans, C. döt Net, I. 2009. YAML Ain't Markup Language (YAML™) Version 1.2. Luettavissa: https://yaml.org/spec/cvs/spec.pdf. Luettu: 8.5.2021.

Bettendorf, M. 2020. API Growth Continues to Skyrocket in 2020 and into 2021. Luettavissa: https://blog.postman.com/api-growth-rate/ Luettu: 8.5.2021

Booth, D. Haas, H. Newcomer, E. Champion, M. Ferns, C. Orchard, D. 2004. Web Services Architecture. Luettavissa: https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest Luettu: 8.5.2021

Breitenecker, F. Troch, I. 2011. SIMULATION SOFTWARE – DEVELOPMENT AND TRENDS. Luettavissa: https://www.eolss.net/sample-chapters/C18/E6-43-07-07.pdf. Luettu: 8.5.2021

Codecademy, "What is REST?". Luettavissa: https://www.codecademy.com/articles/what-is-rest. Luettu: 7.4.2021

Computer Hope, 2020. What is the Command Line? Luettavissa: https://www.computerhope.com/jargon/c/commandi.htm. Luettu: 8.5.2021

Curl, s.a. FAQ -- Frequently Asked Questions. Luettavissa: https://curl.se/docs/faq.html#What_is_cURL Luettu: 8.5.2021

ECMA-404, "The JSON data interchange syntax", 2nd edition, December 2017. Luettavissa: https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf. Luettu: 8.5.2021.

Fielding, R. Gettys, J. Mogul, J. Frystyk, H. Masinter, L. Leach, P. Berners-Lee, T. 1999. Hypertext Transfer Protocol -- HTTP/1.1. Luettavissa: https://www.hjp.at/doc/rfc/rfc2616.html. Luettu: 8.5.2021

Hoverfly. 2021. Key concepts. Luettavissa: https://docs.Hoverfly.io/en/latest/pages/keyconcepts/keyconcepts.html. Luettu: 22.4.2021

Hu, X. 2004. A simulation-based software development methodology for distributed real-time systems. Luettavissa: https://repository.arizona.edu/handle/10150/280514. Luettu: 8.5.2021.

Unadkat, J. 2021. What is Test Driven Development (TDD) : Approach & Benefits. Luettavissa: https://www.browserstack.com/guide/what-is-test-driven-development Luettu: 8.5.2021

Mealling, M. Denenberg, R. 2002. Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations. Luettavissa: https://www.rfc-editor.org/info/rfc3305. Luettu: 8.5.2021.

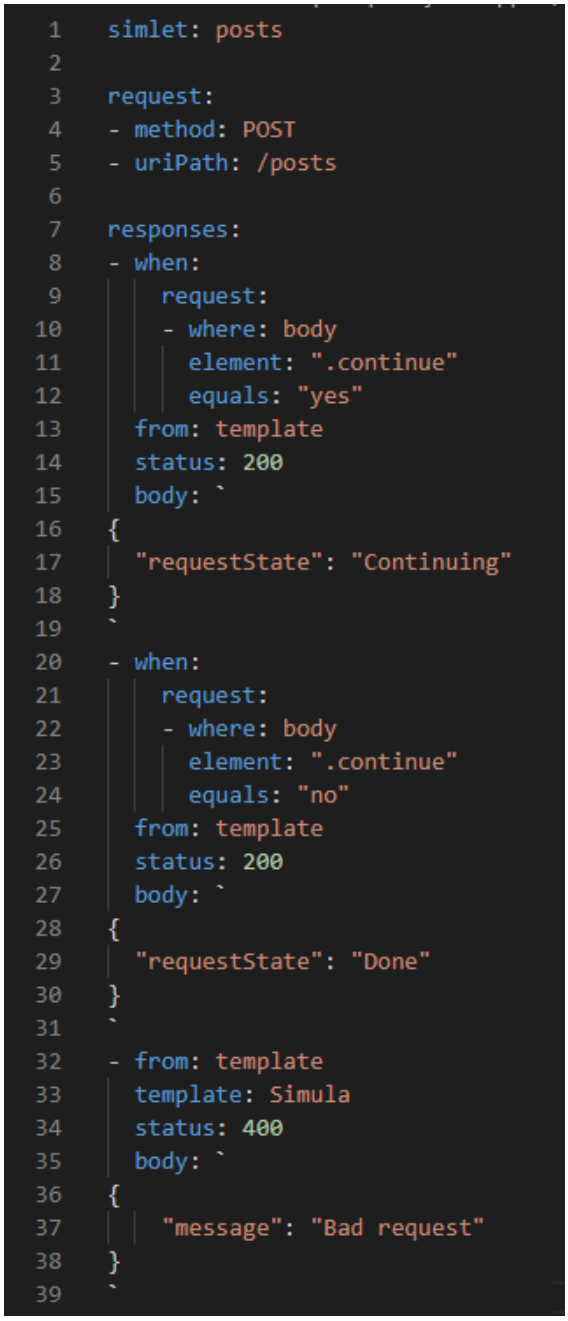
Runeson, P. "A survey of unit testing practices," in IEEE Software, vol. 23, no. 4, pp. 22-29, July-Aug. 2006, doi: 10.1109/MS.2006.91.

Stylos, J. Faulring, A. Yang, Z. and Myers, B. "Improving API documentation using API usage information," 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Corvallis, OR, 2009, pp. 119-126, doi: 10.1109/VLHCC.2009.5295283.

Widmann, A. 2006. Why Implementing Web Services using Representational State Transfer (REST) is better than SOAP. Luettavissa: https://www.cs.drexel.edu/~bmitchel/course/cs575/projFall2006/t3.pdf. Luettu: 8.5.2021.

# Liitteet

## Liite 1: API Simulator simulaatiotiedoston sisältö



```
simlet: posts

request:
- method: POST
- uriPath: /posts

responses:
- when:
    request:
    - where: body
      element: ".continue"
      equals: "yes"
  from: template
  status: 200
  body: `
{
  "requestState": "Continuing"
}
`
- when:
    request:
    - where: body
      element: ".continue"
      equals: "no"
  from: template
  status: 200
  body: `
{
  "requestState": "Done"
}
`
- from: template
  template: Simula
  status: 400
  body: `
{
    "message": "Bad request"
}
`
```

Kuvassa on API Simulator simulaatiotiedoston koko sisältö

## Liite 2: Hoverfly simulaatiotiedoston sisältö


```
{
    "data": {
        "pairs": [
            {
                "request": {
                    "path": [
                        {
                            "matcher": "exact",
                            "value": "/posts"
                        }
                    ],
                    "method": [
                        {
                            "matcher": "exact",
                            "value": "POST"
                        }
                    ],
                    "destination": [
                        {
                            "matcher": "exact",
                            "value": "jsonplaceholder.typicode.com"
                        }
                    ],
                    "scheme": [
                        {
                            "matcher": "exact",
                            "value": "http"
                        }
                    ],
                    "body": [
                        {
                            "matcher": "json",
                            "value": "{ \"continue\": \"yes\" }"
                        }
                    ]
                },
                "response": {
                    "status": 200,
                    "body": "{\n  \"requestState\": \"Continuing\" \n}",
                    "encodedBody": false,
                    "headers": {
                        "Content-Type": [
                            "application/json; charset=utf-8"
                        ]
                    },
                    "templated": false
                }
            },
            {
                "request": {
                    "path": [
                        {
                            "matcher": "exact",
                            "value": "/posts"
                        }
                    ],
                    "method": [
                        {
                            "matcher": "exact",
                            "value": "POST"
                        }
                    ],
                    "destination": [
                        {
                            "matcher": "exact",
                            "value": "jsonplaceholder.typicode.com"
                        }
                    ],
                    "scheme": [
                        {
                            "matcher": "exact",
                            "value": "http"
                        }
                    ],
                    "body": [
                        {
                            "matcher": "json",
                            "value": "{ \"continue\": \"no\" }"
                        }
                    ]
                },
                "response": {
                    "status": 200,
                    "body": "{\n  \"requestState\": \"Done\" \n}",
                    "encodedBody": false,
                    "headers": {
                        "Content-Type": [
                            "application/json; charset=utf-8"
                        ]
                    },
                    "templated": false
                }
            }
        ],
        "globalActions": {
            "delays": [],
            "delaysLogNormal": []
        }
    },
    "meta": {
        "schemaVersion": "v5.1",
        "hoverflyVersion": "v1.3.2",
        "timeExported": "2021-04-20T18:01:21+03:00"
    }
}
```


Kuvassa on Hoverfly simulaatiotiedoston koko sisältö