Santeri Einola

Selvitys HTTP API - jäljittelypalvelimen luomisesta

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

22.5.2023

Tiivistelmä

Tekijä:	Santeri Einola
Otsikko:	Selvitys HTTP API -jäljittelypalvelimen luomisesta
Sivumäärä:	33 sivua
Aika:	2.6.2023
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Lehtori Simo Silander
	Ohjelmistokehittäjä Janne Antila

Työn tarkoituksena oli selvittää, mitä etuja HTTP API -jäljittelypalvelimen käyttämisellä on perinteisten testikirjastojen käyttöön verrattuna, mahdollisuudet ja menetelmät mainitun jäljittelypalvelimen luomiselle sekä mahdollisuuksien mukaan toteuttaa sellaisen käyttöönotto finanssialan yrityksen web-sovelluksessa. Selvitystyön tuloksena syntyi kattava raportti siitä, miten jäljittelypalvelimen käyttäminen vertautuu perinteisempiin integraatiotestausmenetelmiin. Merkittävimpinä etuina voidaan katsoa olevan jäljittelypalvelimen hyödyntämisen realistisuus tuotantokäytössä olevien, kolmannen osapuolen palvelimien käyttöön verraten sekä tällaisten tuotantopalvelimien testauskäytöstä seuraavien haittapuolien eliminoiminen.

Koska jäljittelypalvelimen luomisen todettiin olevan mahdollista ja sen käyttöönotto sisälsi etuja perinteisempiin integraatiotestausmenetelmiin verrattuna, päätettiin sellainen toteuttaa työn kohteena olevalle sovellukselle. Toteutuksen työkaluksi valittiin WireMock, joka on työn tekohetkellä tarkoitukseen pätevin ja soveltuvin vaihtoehto. WireMockia vastaavia työkaluja ei juuri ole tarjolla, ja muutamat jokseenkin samankaltaiset työkalut sisältävät merkittäviä haittapuolia WireMockiin verrattuna.

Käyttöönoton tuloksena yrityksen sovelluksen integraatiotestaaminen kohentui monelta osin. Koska alun perin sovelluksessa kolmannen osapuolen palveluihin liittyvään integraatiotestaamiseen käytettiin joko automaatiotesteihin sisällytettyjä kirjastoja tai suoraan kolmannen osapuolen palveluja, saavutettiin WireMock-palvelimen käyttöön siirtymisen myötä merkittäviä parannuksia. Näistä tärkeimmät ovat jäljittelypalvelimen realistisuus kolmannen osapuolen tuotantopalvelimiin nähden, sen nopeus ja luotettavuus sekä kolmannen osapuolen palveluiden testikäytöstä johtuvien kulujen eliminoiminen. Lisäksi vapaasti muokattavat testikutsupäätteet sekä -data ovat WireMockin tuomia uusia ominaisuuksia.

Avainsanat: WireMock, integraatiotestaus, web-sovellus, automaatiotestaus, Docker, Kubernetes, PollyJS, Robot Framework

Abstract

Author:	Santeri Einola
Title:	Creating HTTP API Mock Server
Number of Pages:	33 pages
Date:	2 June 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Software Engineering
Supervisors:	Simo Silander, Senior Lecturer Janne Antila, Developer

---

The goal of the thesis was to find out what advantages using a HTTP API mock server has compared to using traditional test libraries, the possibilities and methods for creating such, and, if possible, to implement one to be used with the web application of a financial company. As a result of the investigation, a comprehensive report was created on how using a mock server compares to more traditional integration testing methods. The most significant advantages are the realism of using a mock server compared to the use of third-party servers in production use, and the elimination of disadvantages resulting from the testing use of such production servers.

Since the creation of the mock server was found to be possible and its implementation has advantages compared to more traditional integration testing methods, it was decided to implement one for the application in question. WireMock was chosen as the implementation tool, which is the most valid and suitable alternative for the purpose at the time of the thesis. There are almost no tools available comparable to WireMock, and the few that are somewhat similar tools have significant disadvantages compared to WireMock.

As a result of the implementation, the integration testing of the company's application improved in many respects. Since initially the application used either libraries included in automation tests or third-party services directly for integration testing related to third-party services, significant improvements were achieved when switching to the WireMock server. The most important of these are the realism of the mock server compared to third-party production servers, its speed and reliability, and the elimination of costs resulting from the test use of third-party services. In addition, freely customizable test call terminals and data are new features introduced by WireMock.

Keywords: WireMock, Software Testing, Integration Testing, Web Application, Automation Testing, Docker, Kubernetes, PollyJS, Robot Framework

Sisällysluettelo

Lyhenteet

API: *Application Programming Interface*. Rajapinta, jonka avulla kaksi tai useampi sovellusta voi välittää tietoja keskenään.

HTTP: *Hypertext Transfer Protocol*. Sovelluskerroksen protokolla hypermedia-asiakirjojen, kuten HTML:n, lähettämiseen.

JSON: *JavaScript Object Notation*. Kevyt tiedonsiirtomuoto, jota on helppo lukea ja kirjoittaa.

1 Johdanto

Tämän insinöörityön aiheena on tutkia ja selvittää mahdollisuudet ja menetelmät kaupallisen sovelluksen hyödyntämää ulkoista tuotantopalvelinta vastaavan jäljittelypalvelimen luomiselle. Tällaisen jäljittelypalvelimen avulla on mahdollista saavuttaa merkittäviä hyötyjä kolmannen osapuolen palveluihin kohdistuvien HTTP API -kutsujen testaamisessa.

Sovelluksen ja API:n välisen kommunikaation testaamisella varmistetaan sekä kutsujen toimivuus että palautuvan datan oikeanlainen käsittely. Myös sovellukseen tehtyjen muutosten myötä tapahtuva mahdollinen kommunikaation rikkoutuminen on havaittavissa automaattisilla testeillä. Tällaista testausta kutsutaan integraatiotestaamiseksi. Työn tavoitteena on tutkia ja selvittää menetelmät ja teknologiat, joilla integraatiotestaamisessa hyödynnettävä jäljittelypalvelin on mahdollista toteuttaa, ja valita näistä parhaimmat käytettäviksi toteutuksessa. Jos jäljittelypalvelimen luominen todetaan mahdolliseksi, selvitetään seuraavaksi, onko sitä mahdollista hyödyntää työn kohteena olevan finanssialan yrityksen web-sovelluksen integraatiotestaamisessa. Kyseinen sovellus on riippuvainen useista ulkoisen tarjoajan HTTP API -palveluista, ja jäljittelypalvelimen avulla voidaan parantaa sen integraatiotestausmenetelmiä.

Tuotannossa olevien kolmannen osapuolen API-palveluiden käyttäminen niihin kohdistuvien kutsujen ja palautuvan datan testaamiseen sisältää riskejä ja rajoitteita, joiden takia näitä on järkevämpää testata tuotantopalvelinten toimintaa jäljittelevillä mock-palvelimilla. Rajoitteista merkittävimmät ovat riippuvuus ulkoisen palvelun oikeanlaisesta toiminnasta sekä automaatiotestauksen hitaus ulkoista API:a kutsuttaessa. Työssä tutkitaan mahdollisuuksia jäljittelypalvelun luomiselle ja valitaan parhaat menetelmät sen toteuttamiseen.

Integraatiotestaamiseen hyödynnettävien jäljittelypalvelimien toiminnan ja ominaisuuksien ymmärtämiseen tarvitaan kattavaa tuntemusta erilaisista

teknologioista ja kehitysmenetelmistä. Nämä käydään perusteellisesti läpi ennen siirtymistä toteutusta ja käyttöönottoa käsittelevään osioon.

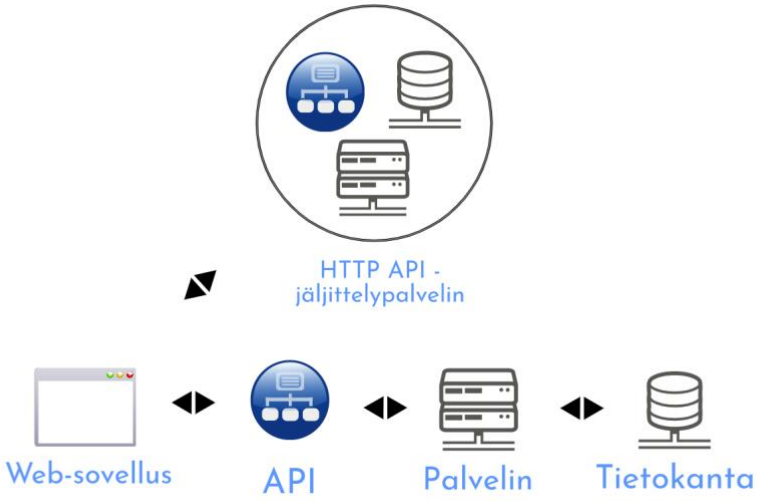
Työn tuloksena selviävät mahdollisuudet ulkoista tuotantopalvelinta jäljittelevän HTTP API:n luomiselle sekä tavat, joilla sellaista voi hyödyntää. Lisäksi jäljittelypalvelimen toteutuksen onnistuessa kohteena oleva web-sovellus saa kattavan, monipuolisen ja ulkoisista palveluista riippumattoman testipohjan niille HTTP API -kutsuille ja palautuvalle datalle, jotka liittyvät kolmannen osapuolen tarjoamiin palveluihin. Lisäksi työstä syntyvää raporttia voidaan hyödyntää muissa sovelluksissa, joissa on aikomuksena siirtyä jäljittelypalvelimen käyttöön. Raportista käyvät ilmi sen luomiseen valitun työkalun ominaisuudet, hyödyt ja haitat, soveltuvuus eri käyttötarkoituksiin sekä yksityiskohtainen kuvaus käyttöönotosta ja sen askeleista.

2 Integraatiotestaaminen

Jotta voimme ymmärtää, mistä integraatiotestaamisessa on kyse, tulee meidän ensin olla perillä muutamasta siihen liittyvästä peruskäsitteestä, sekä siitä, miksi integraatiotestaaminen on tärkeää.

2.1 Ulkoiset API:t

Monet sovellukset käyttävät palvelimilta haettavaa dataa. Esimerkiksi sivustot, joissa hyödynnetään käyttäjäkirjautumista, lähes poikkeuksetta säilövät käyttäjätietonsa palvelimelle. Jotta sovelluksen on mahdollista hakea ja hyödyntää palvelimella olevaa dataa, tarvitaan tarkoitukseen sopiva rajapinta. Tällaista rajapintaa kutsutaan yleisnimellä ’API’ (AWS).



Kuva 1. API:n arkkitehtuuri.

API:en toiminta on kuvattu yksinkertaistetusti kuvassa 1. Web-sovellus lähettää internetin kautta kutsuja rajapintaan, joka hakee kutsun mukaisen datan palvelimelta ja lähettää sen jälleen web-sovellukselle. API:a on olemassa kahdenlaisia: sisäisiä ja ulkoisia. Sisäisillä tarkoitetaan rajapintoja, jotka tarjoavat pääsyn organisaation arkaluontoiseen dataan ainoastaan sen sisällä. Ulkoiset API:t sen sijaan avaavat portin myös yrityksen ulkopuolisille tahoille. Näin ollen esimerkiksi säänseurantasovellukset voivat hyödyntää säädataa tuottavia palveluntarjoajia tietolähteinään (Tozzi 2022).

Koska ulkoiset API:t ovat käytettävissä myös sen hallinnoiman organisaation ulkopuolella on otettava huomioon muutama erityisseikka. Ensimmäinen näistä on tietoturva. Ulkoiset API:t tulee aina suunnitella niiden sisältämän arkaluontoisen datan suojaaminen edellä. Niillä on lisäksi pääsääntöisesti suuremmat käyttäjämäärät, jolloin suorituskyvyn ja käyttötietojen seuraaminen toiminnan optimoimiseksi ovat usein välttämättömiä. Ulkoiselle rajapinnalle on myös tärkeää määritellä käyttöehdot kuten esimerkiksi, mihin tietoihin kullakin asiakkaalla/sovelluksella on pääsy ja kuinka monta kutsua tietyssä aikaikkunassa rajapintaan voi tehdä.

2.2 Mitä on integraatiotestaaminen

Sovellukset ovat monimutkaisia työkaluja, joiden toiminta on altista häiriöille. Pahimmillaan yksi väärässä kohdassa oleva merkki saattaa estää koko sovellusta toimimasta. Siksi on ensiarvoisen tärkeää, että sovellusta testataan kattavasti ja mahdolliset ongelmat havaitaan ajoissa.

Yksinkertaisimmillaan sovellustestaus on sen ominaisuuksien toiminnallisuuden varmistamista manuaalisesti sitä käyttämällä. Aluksi selvitetään mahdolliset käyttötapaukset, jonka jälkeen näiden toimintaa testataan yksi kerrallaan. Tällaista testaamista kutsutaan manuaalitestaamiseksi. Manuaalitestaamisen rinnalla kulkevat automaatiotestit. Niillä tarkoitetaan ohjelmoituja testitapauksia, jotka kykenevät ajettaessa itsenäisesti tutkimaan sovellusta ja raportoimaan siitä löytyneitä ongelmia. Automaatiotestaamisella on etunsa manuaaliseen verrattuna. Näitä ovat esimerkiksi lyhyempi toteutusaika, mahdollisuus määrittää testien ajamisajankohdat ilman rajoituksia, sekä se, että kerran luotu automaatiotesti voidaan ajaa rajattomia kertoa ilman vaivaa, joka manuaalisesti sovellusta käytettäessä syntyisi (Cummings-John 2022).

Sovellustestaaminen jaetaan yleisesti kahteen pääkategoriaan, yksikkötestaukseen ja integraatiotestaukseen. Yksikkötestaamisella tarkoitetaan johonkin yksittäiseen sovelluksen osaan kohdistuvaa testaamista, ja yleensä kohteena olevan komponentin toiminta on muista sovelluksen osista riippumatonta, jolloin se on helppo eristää testaamista varten (pp_pankaj 2022).

Yleisesti ottaen sovellukset ovat monimutkaisia, ja niiden toiminta riippuu useiden komponenttien saumattomasta yhteistyöstä. Näin ollen pelkkä yksikkötestaaminen on usein riittämätöntä, ja joudutaan kirjoittamaan testejä ominaisuuksille, jotka hyödyntävät useamman sovelluksen osan yhteistoimintaa. Tätä kutsutaan integraatiotestaamiseksi. Toisin kuin yksikkötestaaminen, integraatiotestaaminen ottaa huomioon myös sivuvaikutukset, joita komponenttien välisessä viestinnässä voi ilmetä. Tässä insinöörityössä keskitytään integraatiotestaukseen, joka tapahtuu ulkoista tuotantokäytössä olevaa palvelinta imitoivan jäljittelypalvelimen avulla.

Tärkein integraatiotestaamisen tuoma etu on sen kyky havaita ongelmat, joita ei välttämättä sovelluksen osaa tai yksikön toteutusta tutkiessa huomaa. Sen avulla on mahdollista tuoda esiin komponenttien yhteistoiminnassa ilmenevät virheet ja puutteet. Joskus tällaisia virheitä voi olla vaikeita havaita tai uudelleen toteuttaa manuaalisesti sovellusta käyttämällä. (Hamilton 2022.)

2.3 ’End-to-end’-testaaminen

Sovellustestaamisen kokonaisvaltaisinta menetelmää kutsutaan ’end-to-end’-testaamiseksi (jatkossa E2E). Se suomentuu vapaasti muotoon ’päästä päähän testaaminen’, joka kuvaakin hyvin sen tarkoitusta: kyseessä on tapa, jolla mitataan sovelluksen toimivuutta käytön aloituksesta sen lopettamiseen oikeita käyttötapauksia mukaillen. Käytännössä tämä tarkoittaa sitä, että E2E-testi käy läpi jokaisen sovelluksen toiminnallisuuden testatakseen, miten sovellus kommunikoi laitteiston, nettiyhteyden, ulkoisten riippuvuuksien, tietokantojen ja toisten sovellusten kanssa. E2E-testaus toteutetaan yleensä vasta yksikkö- ja toiminnallisuustestien jälkeen. (Bose 2022.)

Kuvitteellinen E2E-tapaus, kirjan lainaaminen kirjaston järjestelmästä:

1) Kirjoitetaan lainaussovelluksen URL osoitekenttään siirtyäksemme sisäänkirjautumissivulle.

2) Kirjaudutaan sisään voimassa olevilla tunnuksilla.

3) Siirrytään lainausnäkymään. Valitaan lainattavat kirjat koriin.

4) Lähetetään ’ei saatavilla’ olevasta kirjasta saatavuuskysely.

5) Siirrytään omaan lainauskoriin. Vahvistetaan lainaus.

6) Kirjaudutaan ulos painamalla nappia ’Kirjaudu ulos’.

2.4 Sovellustestaamisen tärkeys

Suurin osa ohjelmistoalan yrityksistä käyttää hyödykseen joko manuaali- tai automaatiotestaamista tai näiden yhdistelmää. On kuitenkin mahdollista, ettei näistä kumpaakaan hyödynnetä. Tästä seuraa yleensä ongelmia, ja vaikka testittömyyden tarkoituksena on yleensä säästää kustannuksissa, saattaa tarkoitusperä kääntyä lopulta päälaelleen. Siksi onkin tärkeää ymmärtää sovellustestaamisen merkitys sekä asiat, jotka sillä voidaan saavuttaa.

Sovellustestaamisen tavoitteena on löytää mahdolliset virhetoiminnallisuudet ja ongelmat jo kehitysvaiheessa. Tämä tapahtuu arvioimalla tilanteita, joissa toiminnallisia virheitä saattaa ilmetä, ja luomalla kyseisille tilanteille niitä vastaavat testit. Näin saavutetaan tilanne, jossa lopulliseen tuotteeseen pääsee mahdollisimman vähän virhetoiminnallisuutta, jolloin asiakastyytyväisyys ja kuluttajien luottamus tuotetta kohtaan kasvaa. Lähestymistapaa, jossa testi luodaan ennen varsinaista toiminnallisuutta, kutsutaan testilähtöiseksi kehittämiseksi. (Hamilton 2022.)

Puutteellisen testaamisen tuomat ongelmat konkretisoituivat Starbucksille vuonna 2015 (Soper 2015). Tuolloin sen digitaalinen myyntialusta kaatui järjestelmän ajoittaiseen päivitykseen liittyvän virhetoiminnallisuuden takia. Yhtiö hävisi kaatumisesta johtuvan myyntikatkoksen takia miljoonia dollareita. Jälkeenpäin todettiin, että virhetoiminnallisuus olisi voitu havaita sovellustestein. Vastaavasti Nissan koki mittavia menetyksiä vuonna 2016 sen turvatyynyihin liittyvän ohjelmisto-ongelman takia.

Sovellustestaamisen suurimmat hyödyt ovat seuraavat:

- Ongelmat pystytään havaitsemaan jo kehitysvaiheessa. Mitä monimutkaisempi sovellus, sitä alttiimpi se on toiminnallisille virheille.

- Tuotteesta saadaan laadukkaampi. Testaamisen ansiosta lopulliseen tuotteeseen pääsee vähemmän virhetoiminnallisuutta, ja kuluttajat pitävät toiminnaltaan luotettavaa sovellusta laadukkaana.

- Kuluttajaluottamus kasvaa. Testaamisella parannetaan myös sovelluksen kykyä torjua kyberhyökkäyksiä ja tietomurtoja. Näin ollen myös asiakkaiden arkaluotoiset tiedot pysyvät turvassa.

- Auttaa mittaamaan datansiirtomääriä. Skaalautuvuustestauksella pystytään selvittämään esimerkiksi se, kuinka paljon samanaikaista käyttäjäliikennettä sovellus kestää.

- Säästää rahaa. Kuten yllä mainittujen esimerkkitapausten kohdalla, havaitsemattomat virheet saattavat aiheuttaa mittavia rahallisia menetyksiä. Yleisesti ottaen kehitysvaiheessa havaitut ongelmat ovat halvempia ja helpompia korjata kuin jo tuotantoon päästessään. (Korzeniewski 2016.)

2.5 Nykytilanne yrityksen sovelluksessa

Tämän insinöörityön kohteena olevassa sovelluksessa ovat testaamiseen liittyvät seikat pääosin kunnossa. Kriteerit on määritelty tarkasti: koodin automaatiotestikattavuuden on oltava kaikilta osin vähintään 80 %, jokainen uusi ominaisuus kulkee katselmoinnin jälkeen tarkan manuaalitestauksen läpi, integraatiotestaaminen on niin ikään kattavalla pohjalla ja sovelluksen toimintaa testataan myös ’end-to-end’-tyylisesti.

Kuitenkin käytössä olevat teknologiat alkavat tietyiltä osin olla jo vanhentuneita, jolloin kehittämiselle on tarvetta. Integraatiotestaamisen osalta sovelluksessa on tarkoituksena siirtyä nykyisen PollyJS-kirjaston sijaan ulkoista kolmannen osapuolen palvelimen toimintaa imitoivan jäljittelypalvelimen käyttöön. Tämä on linjassa yrityksen yleisten käytänteiden kanssa, sillä osassa rinnakkaissovelluksista on jo siirrytty käyttämään työkalua nimeltä ’WireMock’. WireMock on tämän työn kirjoitushetkellä yksi varteenotettavimmista työkaluista, joilla on mahdollista luoda kuvatun kaltainen jäljittelypalvelin.

Suurimpana erona varsinaisen jäljittelypalvelimen ja PollyJS:n välillä on niiden toteutustapa: siinä missä PollyJS on kirjasto, joka kykenee jäljittelemään

sovelluksen ja palvelimen välisten HTTP-kutsujen pyyntöjä ja vastauksia (Jason Mitchell, Offir Golan, Sophinie Som, 2022), voidaan luotu jäljittelypalvelin määritellä täysiveristä tuotantokäytössä olevaa HTTP API -palvelinta vastaavaksi. Tämän työn toteutuksessa on suunnitelmana luoda vuorokauden ympäri toimiva palvelin, jonka käyttäminen ja hyödyntäminen testauksessa vastaa palautusdatan autenttisuutta lukuun ottamatta täysin ulkoisen kolmannen osapuolen tuotantopalvelimen toimintaa. PollyJS:n kanssa joudutaan joissain tapauksissa hyödyntämään tuotantokäytössä olevaa palvelinta, joten tästä riippuvuudesta päästään eroon ja vältytään varsinaisten ulkoisten palvelinten testauskäytön tuomista ongelmista.

3 PollyJS ja Robot Framework

Tämän työn toteutuksen keskiössä on siirtymä PollyJS:n käytöstä ulkoista tuotantokäytössä olevaa palvelinta imitoivan jäljittelypalvelimen käyttöön. Lisäksi nykyisiä Robot Frameworkilla toteutettuja integraatiotestejä tullaan muokkaamaan niin, että ne käyttävät tuotannossa olevien ulkoisten palvelinten sijaan jäljittelypalvelinta. Näin ollen näihin kahteen työkaluun on syytä tutustua tarkemmin.

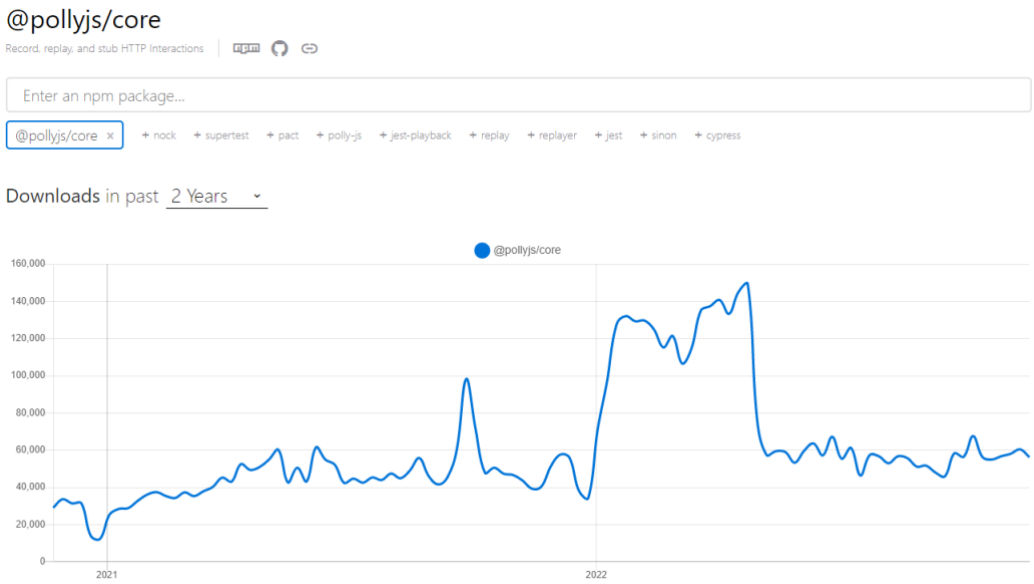
3.1 Mikä on PollyJS?

PollyJS on sovelluskehyksistä (esimerkiksi Vue, Angular ja React) riippumaton JavaScript-kirjasto, jonka pääominaisuuksia ovat HTTP-kutsujen tallentaminen ja uudelleenajaminen sekä kutsujäljitelmien luominen. Sovelluskehysriippumattomuudella tarkoitetaan sitä, että kyseessä oleva kirjasto on täysin eristetty omiin tiedostoihinsa, jolloin se toimii moitteetta minkä tahansa sovelluskehyksen kanssa. Tämä on hyödyllistä esimerkiksi suurten yrityksien sovellusten kanssa, jolloin itsenäisten tiimien käyttäessä valitsemiaan sovelluskehyksiä ei yhteensopivuusongelmia PollyJS:n suhteen synny. Suurimpana PollyJS:n etuna ovat sen tuomat mock-kutsujen ja -palautusten luontimahdollisuudet, jotka vaativat hyvin vähän konfigurointia. Mock-kutsuilla tarkoitetaan testaamista varten luotuja palvelimelle tehtäviä HTTP-kutsuja vastaavia jäljitelmiä ja mock-palautuksilla palvelimelta saapuvan datan

jäljittelemistä. PollyJS-kirjastoa on mahdollista hyödyntää useimpien Node- ja selainpohjaisten kutsu-API:en kanssa, ja mock-kutsujen hallinta on tehty helpoksi yksinkertaisen ja intuitiivisen sisäänrakennetun API:n ansiosta. (Mitchell, Golan, Som 2022.)

3.2 Miksi valita PollyJS?

Koska HTTP-kutsujen testaamiseen on olemassa lukemattomia toisistaan poikkeavia tapoja ja teknologioita, on syytä ymmärtää perusteellisesti jokaisen tuomat mahdollisuudet ja rajoitteet.



Kuva 2. PollyJS:n latausmäärät vuosina 2021–2022 (npm trends, 2022.).

Mainitulla saralla PollyJS on yksi suosituimmista työkaluista, ja vuoden 2022 alussa sen latausmäärät lähtivät jyrkkään nousuun (kuva 2). Suosion syynä voidaan pitää sen monipuolisten ominaisuuksien kokonaisuutta, jotka koostuvat seuraavista:

- testidataolioiden automaattinen ajantasaisuuden säilyttäminen tallentamalla ja ylläpitämällä oikeita palvelinvasteita

- testitapausten HTTP-kutsujen ja -palautusten tallentaminen ja uudelleen toistaminen sekä vertaaminen tulevien testiajojen kanssa mahdollisten muutosten havaitsemiseksi

- mahdollisuus hyödyntää PollyJS:n sisäänrakennettua asiakaspuolen palvelinta, joka mahdollistaa kutsujen ja palautusten muokkaamisen erilaisten sovelluksen käyttötilanteiden simuloimiseksi (esimerkiksi peruskäyttötila, sisällön lataaminen, virhetilanne). (Mitchell, Golan, Som 2022.)

3.3 PollyJS:n käyttäminen

Tutustutaan seuraavaksi PollyJS:n käyttämiseen koodiesimerkin avulla. Seuraavien esimerkkien koodit muodostavat yhden JavaScript (.js) -tiedoston, joka on selkeyden vuoksi jaettu useampaan osaan. Esimerkissä testataan kuvitteellisen kirjanlainausjärjestelmän toimintoja, kuten sisäänkirjautumista ja kirjan lainaamista, ja testin sisällä määritellään tiettyjen kutsupäätteiden (endpointien) palautteet, esimerkiksi onnistuneen kutsun statuskoodi 200. Nämä määrittelyt on toteutettu PollyJS-kirjaston avulla. Koodin kommenteissa ja testien nimissä on yleisestä tavasta poiketen käytetty suomen kieltä opastamistarkoituksessa.

PollyJS:n riippuvuudet tuodaan tiedoston käyttöön muiden JavaScript-riippuvuuksien tapaan. Adapterit, jotka mahdollistavat pyyntöjen vastaanottamisen erityyppisistä lähteistä, ja persisterit, jotka määrittelevät kutsujen ja pyyntöjen käsittelytavat, täytyy erikseen rekisteröidä Polly:n ’register’-metodilla (esimerkkikoodi 1).

```
import { Polly } from '@pollyjs/core';
import XHRAdapter from '@pollyjs/adapter-xhr';
import FetchAdapter from '@pollyjs/adapter-fetch';
import RESTPersister from '@pollyjs/persister-rest';

/*
     Adaptereiden ja persistereiden rekisteröimisen ansiosta jokainen
     tiedoston Polly-instanssi kykenee hyödyntämään niitä nimiensä
     perusteella.
*/

Polly.register(XHRAdapter);
Polly.register(FetchAdapter);
Polly.register(RESTPersister);
```

Esimerkkikoodi 1. PollyJS-esimerkki, riippuvuudet ja rekisteröinnit.

Kun riippuvuudet ja adaptereiden sekä persistereiden rekisteröimiset ovat valmiit, siirrytään Polly-instanssin luomiseen. Kyseisellä instanssilla toteutetaan kaikki PollyJS-kirjastoon liittyvä toiminnallisuus, kuten palvelinkutsut. Instanssia luodessa täytyy tarvittavat adapterit ja persisterit antaa sille attribuutteina. Polly-instanssista tuodaan palvelin (server) omaksi muuttujakseen, jotta sitä voidaan hyödyntää erikseen testeissä (esimerkkikoodi 2).

```
describe('Lainausjärjestelmän etusivu', function () {
     it('sisäänkirjautumisen tulisi olla mahdollista', async function
     () {

          /*
               Luodaan uusi polly-instanssi.

               Yhdistetään Polly fetch- ja xhr-adaptereita hyödyntäen
               selain-API:in.
               Oletusarvoisesti jokainen uudentyyppinen kutsu
               tallennetaan ja jo tallennettuja kutsuja verrataan
               uudesta testiajosta syntyviin.
          */

          const polly = new Polly('Sign In', {
               adapters: ['xhr', 'fetch'],
               persister: 'rest'
          });
          const { server } = polly;
```

Esimerkkikoodi 2. PollyJS-esimerkki, Polly-instanssin luominen.

Aiemmin mainitun ’server’-muuttujan avulla voidaan puuttua testattavalle palvelimelle tuleviin kutsuihin. Aluksi määritellään osoitepääte, johon tuleviin testikutsuihin halutaan vaikuttaa, minkä jälkeen määritellään vaikutus. Se voi

olla esimerkiksi tietyn päätteen kutsuihin vastaaminen valitulla statuskoodilla (esimerkkikoodi 3). ’server’-muuttujalla on myös toinen hyvin olennainen ominaisuus, ’passthrough’ (esimerkkikoodi 3). Kyseinen ominaisuus mahdollistaa sen, että haluttuihin osoitepäätteisiin tuleviin testikutsuihin ei puututa lainkaan, vaan kutsu menee suoraan varsinaiselle palvelimelle.

```
/* Jokaiseen '/lainaus'-endpointiin tulevaan kutsuun vastataan
statuskoodilla 200 */

server
     .get('/lainaus/"path")
     .intercept((req, res) -> res.sendStatus(200));

/* Jokainen '/palautus'-endpointiin tuleva kutsu päästetään polly:n
läpi */

server.get('/palautus').passthrough();
```

Esimerkkikoodi 3. PollyJS-esimerkki, palvelinvasteet.

Testien suorittamisen jälkeen täytyy polly-instanssin toiminta katkaista, jotta sen kautta tapahtuvat kutsut ja yhteys selain-API:in voidaan lopettaa (esimerkkikoodi 4).

```
/* pseudo-testikoodi alkaa */
await visit('/kirjaudu-sisaan');
await fillIn('käyttäjätunnus', 'late-lainaaja');
await fillIn('salasana', 'iforgor');
await book('Ihmisen lyhyt historia");
await goToCart();
/* pseudo-testikoodi päättyy */

expect(location.pathname).to.equal('/cart');

/*
     Kutsumalla 'stop'-funktiota estetään polly:n kautta tapahtuvat
     kutsut ja katkaistaan yhteys selain-API:in.
*/

await polly.stop();
```

Esimerkkikoodi 4. PollyJS-esimerkki, instanssin toiminnan pysäyttäminen.

3.4 Robot Framework

Testiautomaation toteuttamiseen on tarjolla lukuisia prosessia helpottavia kehyksiä. Tämän työn toteutuksen yhteydessä hyödynnetään Robot Framework -nimistä kehystä. Kyseessä on monipuolinen, avoimeen lähdekoodiin perustuva työkalu, joka määrittää tavan, jolla testiautomaatiota/ohjelmistorobotiikkaa kirjoitetaan. Robot Framework ei ole teknologiariippuvainen, joten sen soveltuvuus eri ohjelmistojen kanssa on huippuluokkaa.

Robot Frameworkin pääkäyttötarkoitukset ovat testiautomaatio ja RPA-automaatio. Sen vahvuuksia ovat ilmaiset käyttölisenssit ja yhteensopivuus käytännössä minkä tahansa työkalun kanssa. Monet johtavat teknologiayritykset, kuten Eficode, Reaktor ja Finnair ovatkin omaksuneet Robot Frameworkin huolehtimaan testiautomaatiopohjastaan (Robot Framework Foundation 2022). Työkalun toiminta perustuu avainsanapohjaisuudelle. Tällä tarkoitetaan sitä, että toiminnallisuudet toteutetaan vapaasti määriteltyjen avainsanojen kautta, jolloin kyseisiä avainsanoja voidaan hyödyntää useamman testikokonaisuuden kesken. Näin saavutetaan testipohjan korkea ylläpidettävyyden taso. (Czarnecki 2020.)

Tutustutaan seuraavaksi Robot Framework -testin luomiseen koodiesimerkin avulla. Tässä testisarjassa kuvataan backend API:n käyttäjänhallinnan testaamista. Käyttäjän täytyy olla sisään kirjautunut ennen toimintojen tekemistä. Lisäksi toiminnallisuutta rajoittavat eritasoiset käyttöoikeudet, joita ovat järjestelmänvalvoja, normaali käyttäjä ja vieras.

```
Run Test Suite

*** Settings ***
Documentation      Testisarja sisäänkirjautumiselle.
…
…                  Avainsanat (Keywords) tuodaan testisarjan käyttöön
                   alla määritellystä tiedostosta

Resource           keywords.resource

Suite Setup        Yhdista Palvelimeen

Test Teardown      Kirjaudu Ulos

Suite Teardown     Katkaise Palvelinyhteys
```

Esimerkkikoodi 5. Testisarjan määrittely.

Testisarjan luominen alkaa määrittelyosiosta (esimerkkikoodi 5). Se sisältää testisarjan dokumentaation, resurssitiedostojen määrittelyn ja yleiskomennot sarjan alustamiselle, yksittäisen testin purkamiselle ja sarjan purkamiselle. Mainittujen komentojen toiminnallisuus toteutetaan avainsanojen avulla.

```
*** Keywords ***
Yhdista Palvelimeen
     Connect   fe80::aede: 48ff:fe00:1122

Katkaise Palvelinyhteys
     Disconnect

Kirjaudu Sisaan
     [Arguments]    ${login}  ${password}
     Set Login Name     ${login}
     Set Password  ${password}
     Execute Login

Kirjaudu Ulos
     Execute Logout
```

Esimerkkikoodi 6. Avainsanatiedosto.

Robot Framework perustuu vapaasti määriteltyjen avainten hyödyntämiselle ja uusiokäyttämiselle. Avaimilla tarkoitetaan pieniä yksittäisiä toiminnallisuuksia, joita kutsutaan testien aikana niiden nimen (avaimen) perusteella. Ne määritellään omiin ’.robot’-tyyppisiin tiedostoihin (Esimerkkikoodi 6), joista ne tuodaan testisarjoissa käytettäviksi.

4 Jäljittelypalvelimen hyödyt ja mahdollisuudet sellaisen luomiselle

4.1 Jäljittelypalvelimen hyödyt

Ulkoisia kolmannen osapuolen HTTP API -palvelimia jäljittelevien palvelimien käyttäminen integraatiotestaamisessa on verrattain uusi käytäntö. Niillä saavutetaan kuitenkin merkittäviä hyötyjä muihin menetelmiin verrattuna. Aikaisemmin integraatiotestaamisessa jouduttiin lähes poikkeuksetta turvautumaan varsinaisten tuotantokäytössä olevien ulkoisten tarjoajien palvelinten käyttämiseen, mistä seuraa ongelmia.

Havainnollistetaan tuotantokäytössä olevien kolmannen osapuolen palvelinten ja jäljittelypalvelimen testauskäytön eroavaisuuksia. Esimerkiksi verkkovaatekaupan toimintaa testattaessa tarvitaan yleensä kolmannen osapuolen tarjoamaa maksupalvelua. Tällaisen maksupalvelun käyttämiseen sisältyy haittapuolia. Sieltä ei välttämättä löydy testikäyttöön tarkoitettuja kutsuominaisuuksia, jolloin testaamiseen joudutaan joko käyttämään oikeaa maksudataa (kuten luottokorttitietoja), jäljittelemään maksutapahtuma automaatiotestin sisällä (mikä ei vastaa todellista maksutapahtumaa) tai kokonaan ohittamaan maksutapahtuman testaaminen. Seuraavana haittapuolena on maksupalvelun kutsumisen mahdollinen hitaus tai kaatumisesta/huoltokatkoista johtuva käytön estyminen, joista johtuen testien suorittaminen saattaa estyä. Viimeisenä huomattavaa haittaa aiheuttavana seikkana ovat kolmannen osapuolen palvelun käyttämisestä aiheutuvat mahdolliset maksut.

Edellä mainitut ongelmat vaikuttavat merkittävästi sovellusten integraatiotestaamiseen. Toisaalta testaamisen todenmukaisuuden nimissä halutaan testaamisessa usein käyttää varsinaisia ulkoisia palvelimia, toisaalta siitä syntyvät ongelmat ovat sen verran merkittäviä, että on syytä pohtia vaihtoehtoisia, ei-palvelinpohjaisia menetelmiä. Nykyään kuitenkin

jäljittelypalvelinten myötä on mahdollista saavuttaa ulkoisen palvelimen testauskäytön hyödyt ilman siitä syntyviä yleisiä haittoja ja rajoitteita. Jäljittelypalvelimelle voidaan vapaasti konfiguroida halutut testikutsupäätteet ja niistä palautuva data, jäljittelypalvelinten toiminta on huomattavasti nopeampaa ja luotettavampaa kuin tuotantokäytössä olevien kolmannen osapuolten palvelimien (minkä lisäksi häiriötilanteet on mahdollista ratkaista itse) eikä jäljittelypalvelimen hyödyntämisestä aiheudu käyttömaksuja. Näistä syistä johtuen jäljittelypalvelinten hyödyntäminen integraatiotestaamisessa on työn tekohetkellä kokonaisuutena yksi varteenotettavimmista menetelmistä.

4.2 Mahdollisuudet jäljittelypalvelimen luomiselle

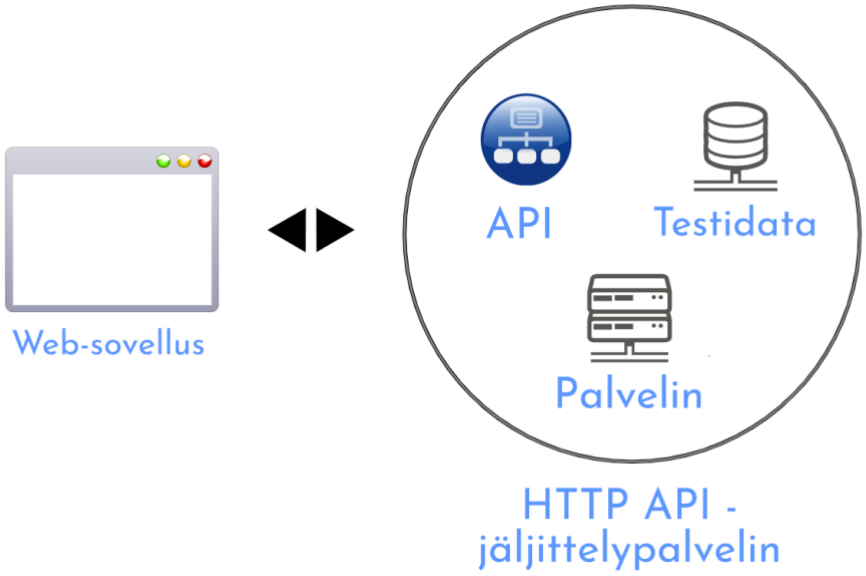
Jotta jäljittelypalvelin on kattavine testausominaisuuksineen mahdollista luoda, tarvitaan avuksi siihen tarkoitettu työkalu. Seuraavaksi edessä on selvitys siitä, mitä työkaluja jäljittelypalvelimen luomiseen on mahdollista käyttää. Aluksi on hyvä ymmärtää, millaisia työkaluja integraatiotestaamiseen on tarjolla. Nämä työkalut jakautuvat karkeasti kolmeen ryhmään.

Ensimmäisen ryhmän työkalut otetaan käyttöön testauksen kohteena olevan sovelluksen testitiedostoissa kirjastoina, ja ne kykenevät ainoastaan imitoimaan ulkoisen palvelimen toimintaa (kuva 3). Tähän ryhmään kuuluvien työkalujen suurimpana heikkoutena ovat, että niiden toiminta ja palautuva data pitää konfiguroida jokaisen testin osalta erikseen, eikä varsinaisia kutsuja ulkoiselle palvelimelle tehdä lainkaan. Tämä vastaa työn kohteena olevan yrityksen sovelluksen nykyistä integraatiotestaamisen tilannetta (käytössä PollyJS) ja sitä halutaan kehittää tuotannossa olevan ulkoisen REST API:n käyttöä vastaavaksi. Näin ollen ensimmäisen ryhmän integraatiotestaamiseen tarkoitetut työkalut (suurin osa kuuluu näihin) ovat poissuljettuja toteutuksen osalta.

Kuva 3. Ensimmäisen ryhmän integraatiotestaustyökalujen arkkitehtuuri.

Seuraavaan ryhmään kuuluvat työkalut toimivat ulkoisella palvelimella tuotantokäytössä olevien ulkoisten palvelinten tavoin (kuva 4), mutta niitä hallinnoi kyseisen työkalun omistava yritys, eikä niitä voi näin ollen tietoturvasyistä käyttää arkaluontoista tietoa käsittelevien yritysten sovelluksien kanssa. Tämän insinöörityön kohteena oleva sovellus kuuluu näihin. Esimerkkeinä toisen ryhmän työkaluista ovat mm. Mockoon ja Kong API Gateway.

Kolmanteen ryhmään kuuluvia työkaluja on työn laatimishetkellä vain yksi. Kyseessä on WireMock. Se vastaa pitkälti toisen ryhmän työkaluja, mutta merkittävänä erona on se, että WireMock-palvelinta ei hallitse työkalun omistava yritys. Tämä tekee WireMockista tietoturvallisemman. Sen avulla on mahdollista luoda oma täysiverinen HTTP API -palvelin, joka jäljittelee tuotantokäytössä olevaa kolmannen osapuolen palvelinta (kuva 4). WireMockilla on monia integraatiotestaamisen kannalta hyödyllisiä ominaisuuksia, kuten REST-kutsujen ja niiden palautusdatan konfigurointi, kutsujen tallentaminen ja mahdollisuus ohjata konfiguroimattomat kutsut WireMock-palvelimen läpi varsinaiselle jäljiteltävälle palvelimelle.



Kuva 4. Toisen/kolmannen ryhmän integraatiotestaustyökalujen arkkitehtuuri.

Koska työn tavoitteisiin kuuluu tuotantokäytössä olevien kolmannen osapuolien palvelimien testauskäytön korvaaminen omalla jäljittelypalvelimella, on WireMock työn tekohetkellä ainoa mahdollinen työkalu tähän tarkoitukseen. Tutustutaan seuraavassa luvussa tarkemmin WireMockiin.

5 WireMock

5.1 Mikä on WireMock?

Kuten aiemmin on todettu, on tuotantokäytössä olevien ulkoisten API-palveluiden käyttäminen integraatiotestaamiseen ongelmallista. Siitä seuraa mm. testien riippuvuus ulkoisen API:n toiminnasta, testiajojen suorittamisen hitautta, mahdollisia ylimääräisiä palvelumaksuja ja riippuvuus internetyhteydestä. Tämän vuoksi on järkevää käyttää varsinaisten API:en sijaan työkaluja, jotka pystyvät jäljittelemään niiden toimintaa ilman mainittuja haittoja. WireMock on yksi tällainen työkalu.

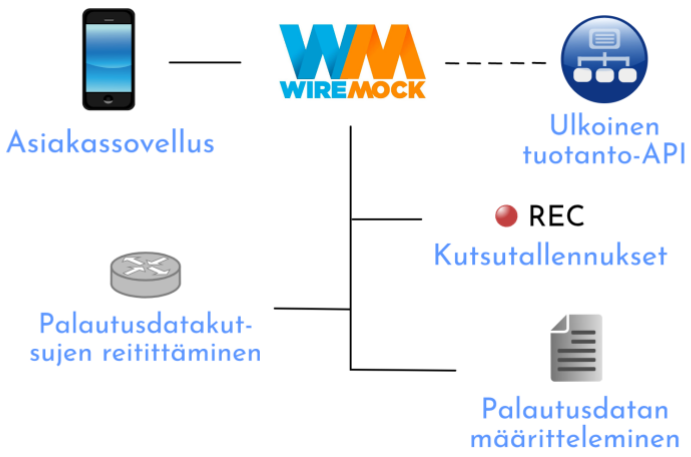
WireMock-palvelimen päätoiminnallisuudet ovat seuraavat:

- mahdollisuus konfiguroida vastaukset, jotka saadaan palvelinta kutsuttaessa

- mahdollisuus käyttää sekä kirjastona että omana palvelimenaan

- kyky tallentaa saapuvia http-kutsuja ja luoda niiden tarkastamiseen omat testinsä

- kyky tunnistaa ja ohjata mallinnetut http-kutsut oikeaan päätteeseen URL:n, kutsumetodin, kutsun otsikkotietojen ja evästeiden avulla. (Yasar 2022.)



Kuva 5. WireMock-toimintakaavio. Perustuu lähteeseen Vlad Georgescu.

WireMockin toimintaperiaate on esitetty kuvassa 5. Testausvaiheessa asiakassovellus kutsuu tuotannossa olevan oikean API:n sijaan WireMock proxy -palvelinta, johon on määritelty tuotannossa olevaa API:a vastaavat kutsuosoitteet ja niistä palautuva data. Lisäksi proxy-palvelimelle tehdyt kutsut on mahdollista tallentaa tarkastelua varten.

5.2 WireMockin soveltuvuus

Vaikka kyseessä on monipuolinen ja moniin käyttötarkoituksiin soveltuva työkalu, on silti tärkeää tunnistaa tilanteet, joissa WireMockin käyttöönotto on

mahdollista. Näin saavutetaan maksimaalinen hyöty kehittämisen kannalta ja varmistutaan siitä, että juuri oikea työkalu on käytössä. Käydään seuraavaksi läpi tällaiset tilanteet.

WireMockin käyttöönotto on aiheellista, kun kehitettävän sovelluksen käytettäväksi suunniteltua API:a ei ole vielä toteutettu. Tällaisissa tilanteissa kuitenkin usein halutaan varmistua ennen käyttöönottoa siitä, että tuleva yhteys toimii saumattomasti. Yhteyttä voidaan näin ollen testata luomalla tulevaa API:a jäljittelevä WireMock-palvelin. (Kainulainen 2018.)

Yleisin käyttösovellus liittyy integraatiotestaamiseen. Ohjelman komponentit ja osat, jotka hyödyntävät kolmannen osapuolen palvelimilta haettavaa dataa, on usein suoraviivaisinta ja turvallisinta testata WireMockin kaltaisilla työkaluilla. Ne soveltuvat useimpiin integraatiotestaustarkoituksiin sovelluksesta riippumatta ja tarjoavat joustavan ympäristön tähän tarkoitukseen. (Kainulainen 2018.)

5.3 WireMock vs. kilpailijat

Integraatiotestaamiseen ja ulkoisten palvelinten toiminnan jäljittelemiseen on tarjolla muutamia laajasti käytössä olevia työkaluja. Näin ollen kehittäjien on valittava omaan sovellukseensa sopivin vaihtoehto, eikä se ole aina yksinkertaista. WireMockia vastaavaa jäljittelypalvelintoteutusta ei kuitenkaan kilpailijoiden joukosta vielä löydy. Otetaan seuraavaksi lyhyt katsaus Mockitoon, joka on yksi suosituimmista WireMockin kaltaisista kirjastoista/työkaluista (AmiyaRanjanRout, 2022.) sekä siihen, miten se WireMockiin vertautuu.

Mockito

Sekä Mockito että WireMock ovat laaja-alaisesti käytössä olevia integraatiotestauksen työkaluja. Niiden tarjoamilla ominaisuuksilla on kuitenkin perustavanlaatuisia eroja. Siksi on tärkeä ymmärtää, mihin kumpikin parhaiten soveltuu.

Kuten luvussa 3.1 mainittiin, on WireMockilla mahdollista luoda palvelin, johon voidaan tehdä kutsuja ja josta palautuu dataa aivan kuten tuotantokäytössä olevalta palvelimelta. Suurin ero mainitun ja Mockiton välillä onkin, että jälkimmäinen ei tähän kykene. Mockitolla palautuva data lisätään suoraan koodin joukkoon, jolloin todellisia http-kutsuja ei tehdä. Tämä onkin sen suurin rajoite. Lisäksi Mockitoa käytettäessä pitää hyödyntää juuri kyseessä olevalle kielelle tarkoitettua kirjastoa. WireMockin käyttö ei ole kieliriippuvaista. Mockiton suurimmaksi hyödyksi WireMockin yli voidaan katsoa sen käyttöönoton helppous: ei tarvitse kuin tuoda kirjasto testitiedoston käyttöön ja aloittaa testien luominen. (AmiyaRanjanRout 2022.)

6 Ohjelmistokonttiarkkitehtuuri toimintaympäristönä

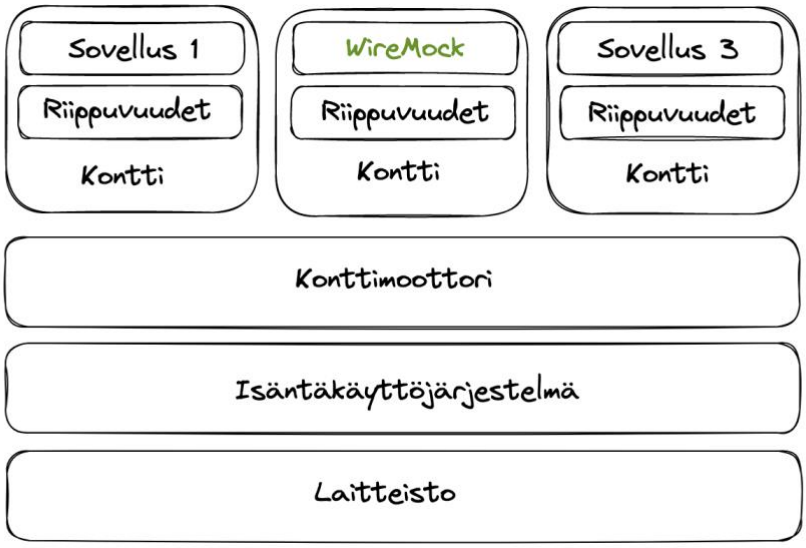
6.1 Ohjelmistokonttiarkkitehtuuri

WireMock-palvelimen toimintaympäristöksi valittiin laajasti käytössä oleva konttiarkkitehtuuritoteutus. Kyseinen toteutustapa vaatii tarkoitukseen sopivan työkalun. Tällaiset työkalut ovat ohjelmia, joilla käyttäjät voivat ajaa sovelluksiaan erillään infrastruktuurista.

Konttityökalun (konttimoottorin) toiminta perustuu siihen, että sen kautta käynnistetään niin kutsuttuja imageja, jotka ovat yhden tai useamman sovelluksen sisältäviä kokonaisuuksia. Imaget taas koostuvat ohjelmistokonteista, jotka ovat sovelluksen ja sen riippuvuuksien muodostamia kokonaisuuksia (kuva 6). Tällainen kontti voi olla esimerkiksi sääseurantasovelluksen ja sen tarvitsemien riippuvuuksien, kuten käyttöliittymäkomponentti- ja tietokannanhallintariippuvuuksien, muodostama kokonaisuus. Imagen sisältämiä kontteja voidaan ajaa samanaikaisesti, ja niiden välillä voidaan jakaa resursseja kunkin kontin senhetkisten tarpeiden mukaan. Näin ollen esimerkiksi virtuaalikonetoteutukseen verrattuna ’ylimääräisten’ resurssien käytöltä, kuten kunkin koneen tarvitsemilta kiinteiltä muisti- ja laskentatehotarpeilta, vältytään. Yksinkertainen esimerkki imagen hyödyntämisestä on tilanne, jossa sekä ohjelmistoyrityksen kehittämis-, testaus-

että tuotantopuoli käyttävät kaikki samaa palvelintoteutuksen sisältämää imagea sen sijaan, että jokaisella olisi oma, toisiinsa nähden identtinen toteutus.

Työn konttitoteutuksen työkaluksi valittiin Docker-niminen ohjelmisto, joka on laajan levinneisyytensä ja aktiivisen ylläpitonsa ansiosta luotettavaksi todettu vaihtoehto.



Kuva 6. Konttiarkkitehtuuri.

6.2 Ohjelmistokonttien hallintajärjestelmä (Kubernetes)

Ohjelmistokonttien käytön hallinta on verrattain monimutkaista, ja aiheuttaa manuaalisesti hoidettuna työtä ja resurssien käyttöä, joilta voitaisiin välttyä hyödyntämällä tarkoitukseen sopivaa ohjelmistokonttienhallintajärjestelmää, joka automatisoi hallintaa. Kubernetes on yksi monipuolisimmista tarkoitukseen sopivista ohjelmistoista, ja sitä käytetään WireMockin toteutuksessa. Kubernetes perustuu avoimeen lähdekoodiin ja sillä on ominaisuuksia, jotka tekevät siitä kilpailijoita kyvykkäämmän tietyissä käyttötapauksissa. Sen toiminta perustuu virtuaalikoneryhmiin, jotka koostuvat kontteja ajavista solmuista. Tällainen ryhmä jakaa samat laskentaresurssit koneidensa kesken.

Kubernetesin pääominaisuudet ovat seuraavat:

- Laskentasuoritusten ajoittamisella jokaisen kontin resurssitarpeet otetaan yksilöllisesti huomioon ja suoritusten ajaminen ajoitetaan näiden tarpeiden mukaan.

- Kubernetes kykenee lisäämään ja poistamaan instansseja sen mukaan, kuinka paljon laskentatehoa kulloinkin tarvitaan.

- Kontin kaatuessa Kubernetes kykenee luomaan sen automaattisesti uudelleen.

- IP-osoitteiden, DNS:n ja useiden instanssien kuormitus on mahdollista tasoittaa toisiinsa verraten.

- Päivitysten aikana uusien instanssien toimintaa tarkkaillaan ja ongelmien ilmetessä siirrytään automaattisesti käyttämään aikaisempaa versiota.

- Kubernetes kykenee hallitsemaan sovellusten konfiguraatioita ja salaisuuksia (secrets). (Haley 2019.)

6.3 Kubernetesin kommunikaatiomenetelmät

Kubernetesin solmujen välinen kommunikaatio perustuu pää- ja alasolmujen hierarkiaan. Pääsolmuja on yleensä yksi ja alasolmuja useampia. Pääsolmun tehtävänä on hallita ryhmää kolmen komponentin avulla. Nämä komponentit ovat API-palvelin, ohjain ryhmän objektien ja resurssien tarkkailun ja tilan hallintaan sekä aikatauluttaja, jolla hallitaan laskentasuoritteiden ajoittamista. Alasolmujen tehtävänä on tarjota ajoaikaa sovelluksille ja pitää yhteyttä muihin solmuihin. Alasolmujen komponentit koostuvat Kubeleteista, jotka kommunikoivat pääsolmun kanssa ja huolehtivat konttien ajon ylläpitämisestä, välityspalvelimesta, joka hallitsee verkkoliikennettä sekä Dockerista, joka jakaa ajoaikaa konteille. (Jason Haley, 2019.)

7 Toteutus ja käyttöönotto

7.1 Suunnittelu

Suunnittelun pohjana on toteuttaa finanssialan yrityksen sovellukselle palvelin, joka imitoi käytössä olevien ulkoisten palvelinten toimintaa. Tähän käytetään WireMockia. Suunnittelu alkoi määrittelemällä ympäristöt, joissa WireMock-palvelin tulisi toimimaan. Tähän on kaksi vaihtoehtoa: joko WireMock-palvelin laitettaisiin erikseen toimimaan paikallisesti käyttäjän omalla tietokoneella, tai se lisättäisiin jo olemassa olevaan pilvipohjaiseen testiympäristöön, jossa se olisi toiminnassa jatkuvasti tuotannossa olevien palvelimien tapaan. Projektin osalta päädyttiin viimeksi mainittuun ratkaisuun. Sillä on muutamia etuja paikallisen palvelimen yli, joista tärkeimmät ovat jäljittelykutsujen luomisen keskittäminen yhteen paikkaan sekä se, ettei palvelinta tarvitse pystyttää erikseen sitä käytettäessä.

7.2 WireMock-palvelimen pohja

WireMockin käyttöönotolle ja toiminnalle on olemassa useita ratkaisuja. Tässä osiossa käymme läpi ainoastaan yhden, työn toteutuksessa käytettävän Kubernetesiin ja Dockeriin pohjautuvan palvelintoteutuksen. Aloitetaan palvelimen pohjana toimivan arkiston (Git-repositoryn) luomisella.

Arkistoon luotava ensimmäinen tiedosto on Dockerin toimintaa ohjaava Dockerfile. Kyseessä on tekstidokumentti, joka sisältää Docker-imagen luomiseen liittyvien komentojen määrittelyt.

```
# komento WireMock-pohjan käyttöönottamiselle
FROM eficode.com/base/wiremock:2022-04-17

# Dockerin ympäristömuuttujat määritellään joko ARG- tai ENV-
etuliitteellä
ARG MAPPINGS_FOLDER

ENV JAVA_HOME=/usr/bin/java

# Dockerin työhakemiston määrittelelvä etuliite
WORKDIR /opt/wiremock

# Komento tiedostojen/hakemistojen kopioimiselle
COPY $MAPPINGS_FOLDER ./mappings/
```

Esimerkkikoodi 7. Dockerfile-esimerkki.

Usein WireMock-palvelinten alustana toimii ulkoinen WireMock-pohja, jolla peruskomentojen toiminnallisuus saadaan lisättyä vaivattomasti. Myös tämän työn toteutus pohjautuu vastaavan alustan käyttämiselle. Käydään seuraavaksi Dockerfilen esimerkkisisältö läpi havainnollistavan esimerkkikoodin 7 avulla. Dockerfilen FROM-etuliitteellä tuodaan WireMock-pohja palvelimen käyttöön määrittelemällä pohjan osoite komennon jälkeen. Rakentamisen (build) aikana tarvittavat ympäristömuuttujat tai niiden sijainnit määritellään ARG-etuliitteen jälkeen. Toinen ympäristömuuttujille tarkoitettu etuliite on ENV. Erona ARG:iin on se, että ARG-etuliitteen muuttujia voidaan käyttää ainoastaan rakentamisen (build) aikana, ENV:llä määriteltyjä sen sijaan vasta rakentamisen jälkeen. WORKDIR-etuliitteellä asetetaan Dockerin käyttämä työhakemisto, jossa osa komennoista (esimerkiksi RUN ja COPY) ajetaan. WORKDIR-komento luodaan ja ajetaan automaattisesti, jos sitä ei erikseen määritellä Dockerfilessä (Educative Answers Team, 2022). Esimerkin viimeisellä komennolla (COPY) kopioidaan tiedostoja tai hakemistoja sellaisenaan sijainnista toiseen. Lisäksi on olemassa vastaava komento, ADD, joka eroaa COPY:n toiminnasta lähinnä tiedostojen purkamisen osalta (Simic 2019).

Kun Dockerfile-tiedoston sisältö on kunnossa, voidaan WireMock-palvelin rakentaa Kubernetesin kautta. Emme tällöin tarvitse erikseen RUN-komentoa imagen ajamiselle.

7.3 Palautusdatan määritteleminen

Koska WireMock-palvelin toimii tuotannossa olevan vastaavan palvelimen tavoin, täytyy sieltä palautuva data erikseen määritellä. Tutustutaan seuraavaksi siihen, miten tämä tapahtuu.

Palautusdata määritellään JSON-muotoisiin tiedostoihin. Tällaisten tiedostojen muodostamaa kokonaisuutta kutsutaan englanninkielisellä nimityksellä ’mappings’. Luontevin tapa sijoittaa palautusdatan määrittelevät tiedostot onkin laittaa ne yhden mappings-kansion sisälle. Paljon palautusdatatiedostoja sisältävän toteutuksen kanssa myös niitä kategorisoivat alikansiot saattavat tulla kyseeseen.

Seuraavaksi itse palautusdataa määrittelevien tiedostojen pariin (esimerkkikoodi 8). Kuten aiemmin mainittiin, ovat palautusdatatiedostot JSON-muotoisia. Niiden sisältämät objektit sisältävät kaikki yksittäiseen kutsuun liittyvät tiedot. Objektin määrittely alkaa kutsun (”request”) tiedoista, joista tärkeimmät ovat metodi (esim. GET tai ANY) ja url, joka määrittelee kutsun osoitepäätteen. Seuraavaksi määritellään kutsun palautusdata (”response”). Palautusdatan vaadittu metasisältö koostuu statuksesta, palautuvasta datasta ja ylätunnisteista. Status on kutsuun sopiva http-statuskoodi (esim. 200 tai 401). Ylätunnisteisiin kuuluvat muun muassa palautuvan datan tyyppi (”application/json”), mahdolliset evästeet ja cachen hallinta. Itse palautuva data voi olla esimerkiksi pelkkää tekstiä, HTML-elementtejä tai vaikkapa JSON-dataa.

Kun kutsut ja niiden palautusdatat on määritelty, voidaan ne ottaa käyttöön niiden tiedostopolkuun viitaten.

```
{
     "request": {
          "method": "GET",
          "url": "/bookings/info"
     },
     "response": {
          "status": 200,
          "jsonBody": {
               "userFullName": "Yakley Reid",
               "userId": "30-123455",
               "currentlyBorrowedBooks": [
                    "Kingdom of Lies",
                    "Hammer and the Inferno",
                    "Quantum tomorrow"
               ],
               "booked": [
                    "Tower and the Crescent"
               ]
          },
          "headers": {
               "Content-Type": "application/json",
               "Set-Cookie": ["session_id=91837492837"],
               "Cache-Control": "no-cache"
          }
     }
}
```

Esimerkkikoodi 8. WireMock-palvelimen palautusdatan määrittely.

7.4 Robot Framework -testit ja niiden alustaminen

Robot Frameworkia käytetään työn kohteena olevan yrityksen laadunvarmistuksessa ’end-to-end’-testaamiseen, eli sovelluksen alusta loppuun vietyjen käyttötapausten testaamiseen. Myös näissä testeissä on mahdollista hyödyntää WireMock-palvelinta. Toisin kuin aiemmassa luvussa, Robot Framework -testien palautusdataa ei määritellä erillisissä tiedostoissa, vaan ne otetaan käyttöön testiajojen alkuvaiheessa API-kutsujen kautta ja poistetaan käytöstä testiajojen päätteeksi. Koska itse Robot-testien luominen on käsitelty luvussa 9, tutustutaan tässä ainoastaan testien alustamiseen.

```
WM_UNITS_MAPPING = {
     "request": {
          "urlPattern": "/AA100/REST.*",
          "headers": {
               "x-session-id": {
                    "equalTo": "{SESSION_ID}"
               }
          },
          "bodyPatterns": [
               {
                    "matchesJsonPath" : "$.
                    AA1000peration.parentPartyId [?(@.partyID ==
                    900000123456)]"
               }
          ],
          "method": "POST"
     },
     "response": {
          "status": 200,
          "headers": {
               "Content-Type": "application/json; charset=utf-8"
          },
          "body": "{\"AA1000perationResponse\":{\"command\": \"proident
          nostrud sit enim eaullamcoquisnoneiusmod velit eu
          occaecatvelit amet su\",\"exit…
     }
}
```

Esimerkkikoodi 9. Palautusdatan määrittelevä Python-tiedosto.

Alustaminen alkaa palautusdatan käyttöönottamisella. Tämä toteutetaan perusmuotoisen ’.robot’-tiedoston ja Robot-testin kautta. Haluttu palautusdata määritellään erilliseen python-tiedostoon (esimerkkikoodi 9) JSON-muotoisena. Tapa vastaa läheisesti luvussa 10.3 käsiteltyä.

```
Create Mock Response In Wiremock

     [Arguments] ${session id}

     Set To Dictionary ${WM_UNITS_MAPPING ['request']['headers']['x-
     session-id']} equalTo ${sessionId} &{headers} Create Dictionary
     Content-Type=application/json; charset=utf-8

     Create Http Session ${WIREMOCK SESSION} ${WIREMOCK URL} ${headers}

     ${response} Send Post Request ${WIREMOCK SESSION} uri=${WIREMOCK
     ADMIN MAPPINGS} data=${WM_UNITS_MAPPING}

     Should Be Equal As Integers ${response.status_code} 201

     ${responseBody} Get Response Json

     ${mapping id} Get From Dictionary ${responseBody} id

     [return] ${mapping id}
```

Esimerkkikoodi 10. Palautusdatan alustava testi.

Kun palautusdatan määrittely on valmis, siirrytään itse testien alustamiseen tarkoitettuun testiin (esimerkkikoodi 10). Python mahdollistaa niin sanottujen kirjastomuuttujien käytön. Ja jotta palautusdatatiedoston sisältöä voidaan hyödyntää Robot-testeissä, täytyy se sisällyttää kirjastomuuttujiin. Tämä tapahtuu ’Set to Dictionary’ -komennolla. Komennolla varmistetaan esimerkin tapauksessa myös oikea session id ja ylätunnisteet sekä määritellään kirjaston datan tyyppi (application/json). Seuraavaksi avataan http-sessio ja lähetetään palautusdatan sisältävä POST-kutsu WireMock-palvelimen osoitteeseen. Alustamistestin lopuksi tarkistetaan vielä sen onnistuminen http-statuskoodin ja palautuksen sisältämän datan vertaamisella.

```
Delete Mock Response From Wiremock
     [Arguments] ${mapping id}

     ${response} Send Delete Request ${WIREMOCK SESSION} uri=${WIREMOCK
     ADMIN MAPPINGS}/${mapping id}

     Should Be Equal As Integers ${response.status_code} 200
```

Esimerkkikoodi 11. Palautusdatan poistava testi.

Kun kaikki testit on suoritettu, täytyy palautusdatat poistaa palvelimelta. Tämä tapahtuu alustamisen tapaan omalla testillään (esimerkkikoodi 11). Toteutus on yksinkertainen: lähetetään poistettavan palautusdatan id:llä yksilöity http-kutsu poistamiseen tarkoitettuun osoitepäätteeseen ja tarkistetaan lopuksi http-statuskoodi poiston onnistumisen varmistamiseksi.

8 Yhteenveto

Työn tarkoituksena oli selvittää, mitä etuja HTTP API -jäljittelypalvelimen käyttämisellä on perinteisten testikirjastojen käyttöön verrattuna, mahdollisuudet ja menetelmät mainitun jäljittelypalvelimen luomiselle sekä mahdollisuuksien mukaan toteuttaa sellaisen käyttöönotto finanssialan yrityksen web-sovelluksessa. Selvitystyön tuloksena syntyi kattava raportti siitä, miten jäljittelypalvelimen käyttäminen vertautuu perinteisempiin integraatiotestausmenetelmiin. Merkittävimpinä etuina voidaan katsoa olevan jäljittelypalvelimen hyödyntämisen realistisuus tuotantokäytössä olevien, kolmannen osapuolen palvelimien käyttöön verraten sekä tällaisten tuotantopalvelimien testauskäytöstä seuraavien haittapuolien eliminoiminen.

Työn alkupuolella käytiin kattavasti läpi integraatiotestaamiseen ja jäljittelypalvelimen luomiseen liittyvät käsitteet ja teknologiat, joiden toiminta on ymmärrettävä jäljittelypalvelinta luodessa. Lisäksi käsiteltyjä teknologioita vertailtiin niiden potentiaalisimpiin kilpailijoihin.

Koska jäljittelypalvelimen luomisen todettiin olevan mahdollista ja sen käyttöönoton sisältävän etuja perinteisempiin integraatiotestausmenetelmiin verrattuna, päätettiin sellainen toteuttaa työn kohteena olevalle sovellukselle. Toteutuksen työkaluksi valittiin WireMock, joka on työn tekohetkellä tarkoitukseen pätevin ja soveltuvin vaihtoehto. WireMockia vastaavia työkaluja ei juuri ole tarjolla, ja muutamat jokseenkin samankaltaiset työkalut sisältävät merkittäviä haittapuolia WireMockiin verrattuna. Käyttöönoton tuloksena yrityksen sovelluksen integraatiotestaaminen kohentui monelta osin. Koska alun perin sovelluksessa kolmannen osapuolen palveluihin liittyvään integraatiotestaamiseen käytettiin joko automaatiotesteihin sisällytettyjä

kirjastoja tai suoraan kolmannen osapuolen palveluja, saavutettiin WireMock-palvelimen käyttöön siirtymisen myötä merkittäviä parannuksia. Näistä tärkeimmät ovat jäljittelypalvelimen realistisuus kolmannen osapuolen tuotantopalvelimiin nähden, sen nopeus ja luotettavuus sekä kolmannen osapuolen palveluiden testikäytöstä johtuvien kulujen eliminoiminen. Lisäksi vapaasti muokattavat testikutsupäätteet sekä -data ovat WireMockin tuomia uusia ominaisuuksia.

Jatkokehitysideana ja työn pohjalta tehtynä suosituksena WireMockin käyttöönoton mahdollisuuksia kannattaa selvittää myös työn toteutukseen liittyvien sovellusten ulkopuolelle. Sen avulla luodun jäljittelypalvelimen testikäyttömahdollisuudet ovat kattavat, ja toiminta laajalti konfiguroitavissa testattavan sovelluksen tarpeiden mukaan. WireMockin soveltuvuus on huippuluokkaa ja sillä saavutettavat edut perinteisiin integraatiotestausmenetelmiin verrattuna merkittäviä.

Lähteet

AmiyaRanjanRout 2022. WireMock vs Mockito. Verkkoaineisto. https://www.geeksforgeeks.org/wiremock-vs-mockito/. Luettu 15.01.2023.

AWS 2022. AWS - What is an API? Verkkoaineisto. https://aws.amazon.com/what-is/api/. Luettu 15.01.2023.

Baeldung 2022. Introduction to WireMock. Verkkoaineisto. https://www.baeldung.com/introduction-to-wiremock. Luettu 12.01.2023.

Bose, Shreya 2022. What is End To End Testing? Verkkoaineisto. https://www.browserstack.com/guide/end-to-end-testing. Luettu 08.01.2023.

Cummings-John, Ronald 2022. What is automation testing? Verkkoaineisto. https://www.globalapptesting.com/blog/what-is-automation-testing. Luettu 14.2.02023.

Educative Answers Team 2022. What is the WORKDIR command in Docker? Verkkoaineisto. https://www.educative.io/answers/what-is-the-workdir-command-in-docker. Luettu 19.01.2023.

Georgescu, Vlad 2022. Android and iOS App testing with Appium and WireMock. Verkkoaineisto. https://adoreme.tech/android-and-ios-app-testing-with-appium-and-wiremock-the-setup-640bd0863b2d. Luettu 21.02.2023.

Haley, Jason 2019. Demystifying containers, Docker, and Kubernetes. Verkkoaineisto. https://cloudblogs.microsoft.com/opensource/2019/07/15/how-to-get-started-containers-docker-kubernetes/. Luettu 27.01.2023.

Hamilton, Thomas 2022. Integration Testing: What is, Types with Example. Verkkoaineisto. https://www.guru99.com/integration-testing.html. Luettu 08.01.2023.

Hamilton, Thomas 2022. What is Test Driven Development (TDD)? Verkkoaineisto. https://www.guru99.com/test-driven-development.html. Luettu 08.01.2023.

Kainulainen, Petri. 2018. WireMock Tutorial: Introduction. Verkkoaineisto. https://www.petrikainulainen.net/programming/testing/wiremock-tutorial-introduction/. Luettu 02.03.2023.

Korzeniewski, Jeremy 2016. Nissan recalls 3.5 million vehicles over airbag sensor. Verkkoaineisto. https://www.autoblog.com/2016/04/30/nissan-recalls-3-million-vehicles-airbag-sensor/. Luettu 01.03.2023.

Mitchell, Jason, Golan, Offir, Som, Sophinie 2022. Record, Replay, and Stub HTTP Interactions. Verkkoaineisto. https://netflix.github.io/pollyjs/#/README. Luettu 26.03.2023.

Npm trends, 2022. Verkkoaineisto. https://npmtrends.com/@pollyjs/core. Luettu 11.03.2023.

pp_pankaj, 2022. Difference between Unit Testing and Integration Testing. Verkkoaineisto. https://www.geeksforgeeks.org/difference-between-unit-testing-and-integration-testing/. Luettu 14.02.2023.

Simic, Sofija 2019. Docker ADD vs. COPY: What are the Differences? Verkkoaineisto. https://phoenixnap.com/kb/docker-add-vs-copy. Luettu 08.02.2023.

Sirotka, Alesia 2022. What Is API and How Does API Work? Quick introduction. Verkkoaineisto. https://flatlogic.com/blog/what-is-api-and-how-api-works/. Luettu 12.01.2023.

Soper, Taylor 2015. Starbucks lost millions in sales because of a ‘system re-fresh’ computer problem. Verkkoaineisto. https://www.geekwire.com/2015/starbucks-lost-millions-in-sales-because-of-a-system-refresh-computer-problem/. Luettu 08.02.2023.

Tozzi, Chris 2022. - The management approach for internal vs. external APIs. Verkkoaineisto. https://www.techtarget.com/searchapparchitecture/tip/The-management-approach-for-internal-vs-external-APIs. Luettu 12.02.2023.

WireMock, 2022. WireMock - The flexible tool for building mock APIs. Verkkoaineisto. https://wiremock.org. Luettu 21.02.2023.

Yasar, Kinza 2022. software testing. Verkkoaineisto. https://www.techtarget.com/whatis/definition/software-testing. Luettu 11.03.2023. [Original source: https://studycrumb.com/alphabetizer]