Tampereen ammattikorkeakoulu

Robot Framework -testaus Kubernetes-ympäristössä

Joonas Köppä

OPINNÄYTETYÖ
Tammikuu 2019

Tietotekniikka
Ohjelmistotekniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikka
Ohjelmistotekniikka

KÖPPÄ JOONAS
Robot Framework -testaus Kubernetes-ympäristössä

Opinnäytetyö 63 sivua, joista liitteitä 8 sivua
Tammikuu 2019

Tässä opinnäytetyössä toteutettiin Liaison Technologies Oy:n toukokuussa 2018 toimeksiantama soveltuvuusselvitys Robot Framework -työkalun käyttämisestä mikropalveluihin perustuvan ohjelmiston integraatiotestauksessa. Työ suunniteltiin ja toteutettiin yrityksen Tampereen toimipisteen laadunvarmistusosastolla yhteistyössä aikaisempien testien ja testikirjastojen kehittäjien kanssa. Päämotiivi soveltuvuusselvitykselle oli uusien integraatiotestien toteuttamisen helpottaminen ja täten vähemmän teknisen henkilökunnan valjastaminen mukaan testien luomisprosessiin.

Soveltuvuusselvityksessä oli kolme päätavoitetta. Ensin piti selvittää, miten yrityksen aikaisempaa Java-ohjelmointikielellä toteutettua integraatiotestausta varten luotua apukirjastoa voitaisiin käyttää hyväksi Robot Framework -testeissä. Sitten tuli luoda uusi Robot Framework -testi, jossa apukirjaston toiminnallisuudet on abstraktioitu selkolukuisten avainsanakutsujen taakse. Sitä varten tuli toteuttaa esimerkkitestiksi tiedostopalvelimella tiedostoa prosessoivan ohjelmiston testaaminen. Lopuksi työssä kehitetty testi tuli saattaa ajettavaksi Kubernetes-ympäristössä niin, että varmistuttaisiin mahdollisuudesta ajaa tulevaisuudessa kehitettävät testit osana jatkuvan integraation testiautomaatioprosessia.

Ratkaisu Java-apukirjaston käyttöön Robot Framework -testeissä löytyi frameworkin tukemista ulkoisista avainsanoja sisältävistä kirjastopalvelimista. Opinnäytetyössä toteutettiin Java-ohjelmointikielellä avainsanakirjasto, johon Robot Framework -testiajot voivat ottaa yhteyden. Java-apukirjasto asetettiin avainsanakirjaston riippuvaisuudeksi, jolloin siinä saatiin valjastettua käyttöön kaikki apukirjaston toiminnallisuudet. Kirjaston lisäksi luotiin kahdeksanosainen Robot Framework -testitiedosto, jossa kutsutaan kirjastopalvelimen toiminnallisuuksia varsinaiset funktiokutsut taakseen abstraktioivien avainsanojen avulla. Testi ajettiin ensin onnistuneesti läpi paikallisesti, jonka jälkeen se säiliöitiin Docker-työkalulla ja sisällytettiin osaksi Kubernetes-määrittelytiedostoa. Luotu tiedosto lähetettiin Kuberneteksen komentorivityökalulla yrityksen Kubernetes-palvelimelle, jossa testiajo suoritettiin vielä kerran läpi onnistuneesti.

Kaikki opinnäytetyön päätavoitteet saavutettiin onnistuneesti, ja soveltuvuusselvityksen lopputuloksena yrityksessä päätettiin jatkaa Robot Framework -työkalujen ja testimenetelmien kehittämistä eteenpäin. Tulevaisuudessa yrityksessä on tarkoitus siirtyä toteuttamaan kasvavassa määrin integraatiotestausta yksinomaan Robot Frameworkin avulla. Yrityksen vähemmän teknistä henkilökuntaa on jo opastettu Robot Frameworkin käyttöön ja heiltä saatu palaute uutta testaustapaa koskien on ollut erittäin positiivista.

Asiasanat: docker, gradle, integraatiotestaus, java, kubernetes, robot framework

ABSTRACT

Tampere University of Applied Sciences
ICT Engineering
Software Engineering

KÖPPÄ JOONAS
Robot Framework Testing in Kubernetes Environment

Bachelor's thesis 63 pages, appendices 8 pages
January 2019

The basis of this thesis was to conduct a proof-of-concept work about using the Robot Framework acceptance testing tool to run integration tests against software that consists of microservices. It was ordered by Liaison Technologies Oy in May 2018 and all the research and development involved with the project took place in their quality assurance department. The main motive for the thesis was to figure out an easier way for less technical employees to create new integration tests, as in the past creating new Java-based tests proved quite complex at times.

There were three main goals to achieve to prove a working concept: first, it was necessary to figure out how we could use the company's earlier Java-based utility libraries within Robot Framework testing. Then we had to create a Robot Framework test case that hides the functionality of the utility libraries behind reusable and higher-level abstract keyword calls. And finally, the working test case had to be packed into a runnable container that could then be deployed into a Kubernetes server where the test would be automatically run to completion.

The solution to include the company's already existing Java libraries in Robot Framework testing was found in the framework's capability to support external remote-type keyword libraries. Thus, a new keyword library was developed with the Java programming language that natively supports other Java libraries as dependencies. After that we could easily serve any existing Java functionality to Robot Framework test runs by adding our new library as a remote library within the Robot Framework test files. After that, we created an eight-part Robot Framework test suite that utilizes our new keyword library's functionality via simple keyword calls to conduct the proof-of-concept test case as described in the test specifications. The tests were first run locally, and then containerized to be run once more in the company's Kubernetes environment. All the goals were successfully achieved and the feedback from the company's less technical test developers has also been highly positive regarding this method of testing. As a result, the company has decided to furthermore continue the development of Robot Framework–related tooling and testing.

Key words: docker, gradle, integration testing, java, kubernetes, robot framework

SISÄLLYS

ERITYISSANASTO

Abstraktiokerros	Toiminnallisuuden tai sisällön peittäminen yleistävämmän esityksen taakse
Android	Googlen kehittämä mobiilikäyttöjärjestelmä.
Apiserver	Kubernetes-palvelimella sijaitseva rajapinta Kubernetes-toteutuksien konfigurointiin ja ajamiseen
Avainsana	Robot Framework -työkalun testitiedostoissa käytettävä funktiokutsu.
Avainsanakirjasto	Koodikirjasto, johon on toteutettu toiminnallisuus Robot Framework -testien avainsanoja varten.
Avoin lähdekoodi	Lähdekoodi, joka on kokonaisuudessaan kenen tahansa käytettävissä ja muokattavissa ilman maksullisia lisenssejä.
Base-image	Docker-työkalulla luotavien imagejen pohja. Esimerkiksi tietty Linux-distribuutio, tai aiemmin luotu kustomoitu image.
Black-box-testaus	Testityyli, jossa testattavan kohteen oikea toiminnallisuus varmistetaan ottamatta kantaa kohteen sisäisiin toimintoihin. Kohteelle suoritetaan pyyntö ja sitten varmistetaan siitä seurannut lopputulos.
Buildata	Suorittaa ohjelmiston build-vaihe, jossa koodi käännetään ajettavaan muotoon riippuvaisuuksineen.
Built-in-kirjasto	Robot Framework -työkalun sisälle valmiiksi toteutettu kirjasto, joka sisältää yleisiä perustoiminnallisuuksia.
Debugata	Etsiä virheitä koodista.

DevOps	Ohjelmistokehitysmetodologia, jonka tavoitteena on saattaa ohjelmistokehittäjien toteuttamat päivitykset mahdollisimman nopeasti tuotantotasolle.
Docker	Käyttöjärjestelmätason virtualisointia ja ohjelmistojen säiliöimistä varten toteutettu työkalu.
Docker-rekisteri	Docker-työkalulla toteutettujen imagetiedostojen säilöntärekisteri.
Forkata	Luoda versionhallintajärjestelmässä jo olemassa olevasta ohjelmistosta omaa kehitystä varten uusi haara.
Framework	Valikoima työkaluja ja toiminnallisuuksia tietyn tyyppisten ongelmien ratkaisuun.
GitHub	Suosittu Git-versionhallintajärjestelmään perustuva graafisen käyttöliittymän ja koodisäilön tarjoava palvelu osoitteessa http://www.github.com/.
Gradle	Työkalu ohjelmistojen riippuvaisuuksien hallintaan, buildaamiseen ja ajamiseen.
Groovy	Ohjelmointikieli, jota käytetään muun muassa Gradle-työkalun skripteissä.
Hyväksymistestaus	Ohjelmistotestaamisen alalaji, jonka tavoitteena on testata kohteen soveltuvuus sille asetettujen vaatimusten perusteella esimerkiksi black box -tyyppisellä testaamisella.
Image	Tiedosto, joka sisältää kaikki tarvittavat tiedot tietyn ympäristön ajamiseen konfiguraatioineen, ohjelmistoineen ja tiedostoineen

Importata	Ottaa kooditiedostossa käyttöön ulkoisessa kirjastossa toteutettu toiminnallisuus.
Instanssi	Luokkatiedoston perusteella luotu objekti, jolle varataan muistista oma alue ja joka on riippumaton mahdollisesti muista samasta luokasta luoduista instansseista.
Integraatiotestaus	Testaamisen alalaji, jossa varmistetaan, että päivitetty yksittäinen komponentti ei riko toiminnallisuutta sitä käyttävästä kokonaisuudesta.
JAR-julkaisupaketti	Paketti, joka sisältää Java-ohjelmointikielellä toteutettuja kooditiedostoja.
Java	Suosittu alustariippumaton ohjelmointikieli.
Jenkins	Ohjelmistojen build- ja deployment-vaiheiden automatisointia varten kehitetty työkalu.
Job	Kubernetes-ohjelmistossa luotava objekti, jonka onnistuneeksi tilaksi lasketaan onnistuneesti loppuun asti suoritus (vrt. Deployment, jossa onnistunut tila on se, kun objekti on jatkuvasti saatavilla).
JSON	Tiedostotyyppi, jossa avain-arvo-parit esitetään helposti luettavassa ja ohjelmallisesti manipuloitavassa muodossa.
Keyword-driven testing	Testaustapa, jossa testin eri vaiheiden komennot on piilotettu varsinaiset funktiokutsut taakseen abstraktioivien avainsanakutsujen taakse.
Konteksti	Tässä työssä: Kuberneteksen *config*-tiedostossa määritelty tieto käytettävästä Kubernetes-klusterista ja käyttäjästä.
kubectl	Kuberneteksen hallintaa varten luotu komentorivityökalu.

Kubelet-kontrolleri	Jokaisesta Kubernetes-noodista löytyvä komponentti, joka pitää huolen siitä, että kaikki kyseisen noodin objektit ovat terveitä ja toiminnallisia.
Kubernetes	Ohjelmisto säiliöityjen ympäristöjen ja ohjelmistojen ajon orkestrointiin, skaalaukseen ja muuhun hallintaan.
Laitteistoriippumaton	Ohjelmisto, toiminnallisuus, tai asia, joka toimii samalla tavalla riippumatta ajoympäristöstä, eli usein käyttöjärjestelmästä.
Liitännäinen	Ohjelmiston päälle asennettava plug-in, eli lisätoiminnallisuutta tarjoava ohjelmistokomponentti.
Linux-distribuutio	Käyttöjärjestelmä, joka perustuu Linuxin kerneliin ja sen päälle luotuihin työkaluihin. Eri distribuutioita on olemassa useita satoja, ja ne kaikki tarjoavat käyttäjälleen hieman erilaiset työkalut.
Master-noodi	Kuberneteksessä pakollinen isäntänoodi, joka sisältää kaiken perusnoodin toiminnallisuuden lisäksi apiserverin, jolla kontrolloidaan ulkopuolisten käskyjen avulla klusterin master- ja slave-noodeja.
Maven	Apachen kehittämä projektien buildaamiseen ja hallintaan tarkoitettu työkalu.
MFT	*Managed File Transfer*, tiedostonsiirtotapa, jossa jokin välikäsi siirtää tiedostoa kahden tekijän välillä.
Mikropalvelu	Yksi ohjelmiston komponenteista, joka on erikseen ajettavissa ja päivitettävissä. Tarjoaa usein input-output-tyyppisen rajapinnan ohjelmiston käytettäväksi.

Olio-ohjelmointi	Luokkatiedostoihin ja niistä luotuihin instansseihin perustuva ohjelmointimalli.
pip	Pythonin pakettienhallintatyökalu.
Podi	Yksi Kuberneteksen lukuisista objekteista. Yksi podi sisältää yhden tai useamman säiliön, jotka on konfiguraatioineen määritelty podin Kubernetes-määrittelytiedostossa.
Proxy	Välityspalvelin, jonka kautta esim. Kuberneteksessä palvelun käyttäjä saa yhteyden Podien sisältöihin.
Python	Korkean tason tulkattava ohjelmointikieli, joka perustuu pienen ytimen päälle asennettaviin lisäkkeisiin.
Replikoida	Kuberneteksessä: monistaa jokin objekti, esimerkiksi podi, niin, että jos yksi podi kaatuu tai vioittuu, identtinen palvelu on edelleen saatavilla käyttäjälle.
Repositorio	Versionhallintajärjestelmässä oleva säilö koodia varten.
Riippuvaisuus	Ohjelmistokoodissa viittaus ulkoiseen koodiin, tai resurssiin, joka tulee liittää suoritettavaan koodiin sen toiminnallisuuden mahdollistamiseksi.
Robot Framework	Ohjelmistotestaukseen suunniteltu Python-liitännäinen, joka perustuu avainsanojen taakse abstraktioitujen toiminnallisuuksien suorittamiseen.
Rolling update	Päivitystyyli, jossa saman palvelun replikoidut kopiot päivitetään yksitellen vastaamaan uusinta versiota, varmistaen täten palvelun jatkuva saatavuus käyttäjälle.
Root-käyttäjä	Linux-ympäristöissä käyttäjätili, jolla on oikeudet suorittaa mitä tahansa käskyjä ja toimintoja järjestelmässä.

SFTP	Tiedostonsiirtoprotokolla, joka käyttää suojattua SSH-yhteyttä tiedostojen hallintaan.
Skaalaus	Palvelun kapasiteetin kasvattaminen suorituskykyä lisäämällä, tai palvelua monistamalla.
Slave-noodi	Kuberneteksessä master-noodin lisäksi tarvittaessa luotava noodi, joka voi sisältää Kubernetes-objekteja ja niitä hallinnoivan kubelet kontrollerin.
Soveltuvuusselvitys	Työ, jossa selvitetään jonkin uuden, tai erilaisen lähestymistavan, tai työkalun soveltuvuus olemassa olevan ongelman ratkaisemiseksi.
Standardikirjasto	Ohjelmointikielen sisäänrakennettu kirjasto, joka tarjoaa kaikki useimmiten käytettävät toiminnallisuudet.
Super-kutsu	Ohjelmakoodissa toiminto, joka kutsuu periytetyn luokan isäntäluokan sisäistä metodia.
Synkroninen	Tapahtumaketju, jossa edellisen tapahtuman loppuun asti suorittaminen varmistetaan ennen seuraavan tapahtuman aloittamista.
Syntaksi	Ohjelmointi- ja merkkauskielien koodin ja funktiokutsujen rakenne.
Säiliö	Suljettu ympäristö, johon on pakattu kaikki tarpeellinen tietyn ohjelmiston ajamiseen riippuvaisuuksineen ja apuohjelmistoineen.
Task	Gradle-työkalulla suoritettava työvaihe. Esimerkiksi ohjelmistoa buildatessa yksikkötestejä varten voidaan luoda buildketjuun oma task.

TestNG	Java-ohjelmointikielellä toteutettujen ohjelmistojen testausta varten kehitetty testiframework.
Web-palvelin	Ohjelmisto, jonka tarkoitus on tarjota sisältöä tai toiminnallisuutta Internetin-välityksellä toisille koneille, tai ohjelmistoille.
Versionhallinta	Järjestelmä, jossa tiedostojen historiatiedot säilytetään ja muutokset dokumentoidaan. Esimerkkinä GitLab ja GitHub.
Virtuaalikone	Tietokonejärjestelmän sisällä emuloitava eristetty järjestelmä, joka matkii oikean tietokoneen toimintaa. Esimerkiksi Linux-ympäristön ajaminen Windows-käyttöjärjestelmässä.
Virtualisointi	Ympäristön emuloiminen esimerkiksi virtuaalikoneen tai Docker-säiliöiden avulla.
XML	Helposti luettava merkintäkieli datan siirtoon ja säilytykseen.
YAML	Merkintäkieli, joka on suosittu erityisesti konfiguraatiotiedostoissa. Kubernetes-määrittely- ja konfiguraatiotiedostot on toteutettu YAML-merkintäkielellä.
Yksikkötestaus	Ohjelmistotestaamisen alalaji, jossa ohjelmiston yksittäisten funktioiden toiminnallisuus testataan antamalla niille syötteitä ja varmistamalla palautettu arvo sekä muut kutsutut funktiot.

1 JOHDANTO

Tietotekniikka-alalla on viime vuosina ollut kiihtyvänä trendinä, jopa vaatimuksena, pystyä toimittamaan uusia toiminnallisuuksia mahdollisimman nopeasti ideatasolta aina asiakkaan käyttöön asti. Kilpailun kasvaessa alalla ensimmäisenä asiakkaiden tarpeisiin vastaavat yritykset ovat automaattisesti etulyöntiasemassa ja menestyvät paremmin (McKinsey & Company, 2015). Yksi tärkeimmistä tekijöistä uusien toimintojen nopeassa toimituksessa on tuotantoketjun automatisointi. Usein toistuvat, ohjelmoitavissa olevat välivaiheet kannattaa aina pyrkiä automatisoimaan, sillä pitkällä aikajänteellä suuretkin työpanostukset voivat maksaa itsensä moninkertaisesti takaisin. Tällaisia välivaiheita ovat esimerkiksi yksikkötestaus, integraatiotestaus, hyväksymistestaus, versionhallinta ja tuotteen julkaisu aina kehitysympäristöstä tuotantoympäristöön asti.

Tämä opinnäytetyö toteutetaan Liaison Technologies Oy -nimiselle yritykselle ja sen tavoite on selvittää, miten heidän ohjelmistonsa integraatiotestaus voitaisiin toteuttaa automatisoidusti Robot Framework -työkalun avulla Kubernetes-ympäristössä. Tämän lisäksi pohditaan, mitä etuja ja haittoja tällä lähestymistavalla saavutetaan ohjelmiston testaamisessa aiemmin käytettyyn TestNG-frameworkiin verrattuna. Työ toteutetaan askeleittain aina testin suunnittelusta sen lopulliseen ajoon Kuberneteksessä selvittäen samalla parhaat toimintatavat tarvittavien työkalujen dokumentaatioita vahvasti hyväksi käyttäen. Valitsin tämän opinnäytetyöaiheen, koska aloitin työni kyseisen yrityksen Tampereen toimipisteellä testiautomaatio- ja DevOps -harjoittelijana, ja koska testiautomaation kehitys on keskeinen osa hyvää DevOps-toimintamallia ja vahvistaa osaamistani juuri tällä osa-alueella.

Opinnäytetyön rakenne koostuu sekä teoriasta, että käytännön esimerkeistä, esimerkiksi koodin ja konfiguraatioiden syntaksin selventämiseksi. Jokaisen työkalun teoria ja soveltuvuus työn tavoitteiden saavuttamiseksi selvitetään opinnäytetyön alussa, jonka jälkeen seuraavissa kappaleissa ratkaistaan niitä käyttäen askeleittain koko työn tavoite. Tärkeimpinä lähteinä työssä ovat käytettävien työkalujen dokumentaatiot, jotka ovatkin hyvin kattavat, sillä ne kaikki perustuvat avoimeen lähdekoodiin. Työhön liitetään myös sen aikana luodut koodi- ja konfiguraatiotiedostot, joihin tekstissä voidaan sitten myöhemmin viitata.

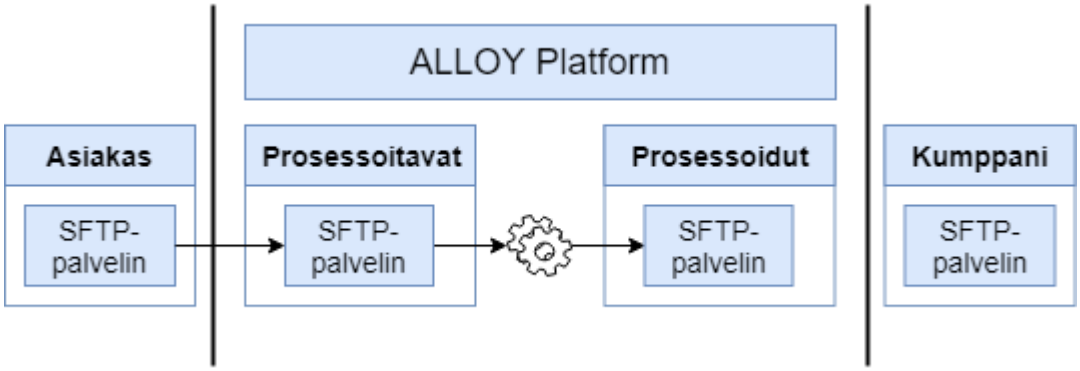
2 RATKAISTAVA ONGELMA

2.1 Lähtötilanne

Yrityksen laadunvarmistusosastolla on toteutettu osa ohjelmistojen integraatiotesteistä Javaan perustuvalla TestNG-frameworkilla. Uusien testien luonnin helpottamiseksi yritys on päättänyt toteuttaa tämän opinnäytetyön avulla soveltuvuusselvityksen Robot Framework -työkalusta osana testien kehitystä. Robot Framework -testeissä voidaan käyttää vahvasti hyväksi vanhoja testejä varten toteutettuja Java-apukirjastoja, joten siirtymä käyttämään Robot Frameworkia uusien testien luomiseen tulevaisuudessa olisi vaivaton. Robot Frameworkilla luodun esimerkkitestin ympärille tulee lopuksi rakentaa tarvittavat toiminnallisuudet testin ajamiseksi Kubernetes-ympäristössä.

2.2 Testattava ohjelmisto

Yksi yrityksen kehittämistä ohjelmistoista on informaation hallintaan ja integraatioon suunnattu alusta nimeltä ALLOY. Se tarjoaa yrityksen asiakkaille mahdollisuuden turvalliseen datan käsittelyyn, säilytykseen ja välittämiseen aina asiakkaan yhteistyökumppaneille asti, mikäli tarpeellista. Yksi ALLOY:n päätoiminnoista on MFT, eli *Managed File Transfer*. Siinä ALLOY hakee asiakkaan SFTP-palvelimelta tiedostoja, käsittelee ne, ja siirtää ne lopuksi asiakkaan yhteistyökumppanin SFTP-palvelimelle. Soveltuvuusselvityksen kohteeksi valittiin nimenomaan tämän toiminnallisuuden testaaminen Robot Frameworkin avulla. MFT-palvelun prosessikaavio on esitetty kuviossa 1. Tiedostojen mahdollinen prosessointi tapahtuu ALLOY-ohjelmiston sisällä prosessoitavien ja prosessoitujen tiedostojen kansioiden välillä.

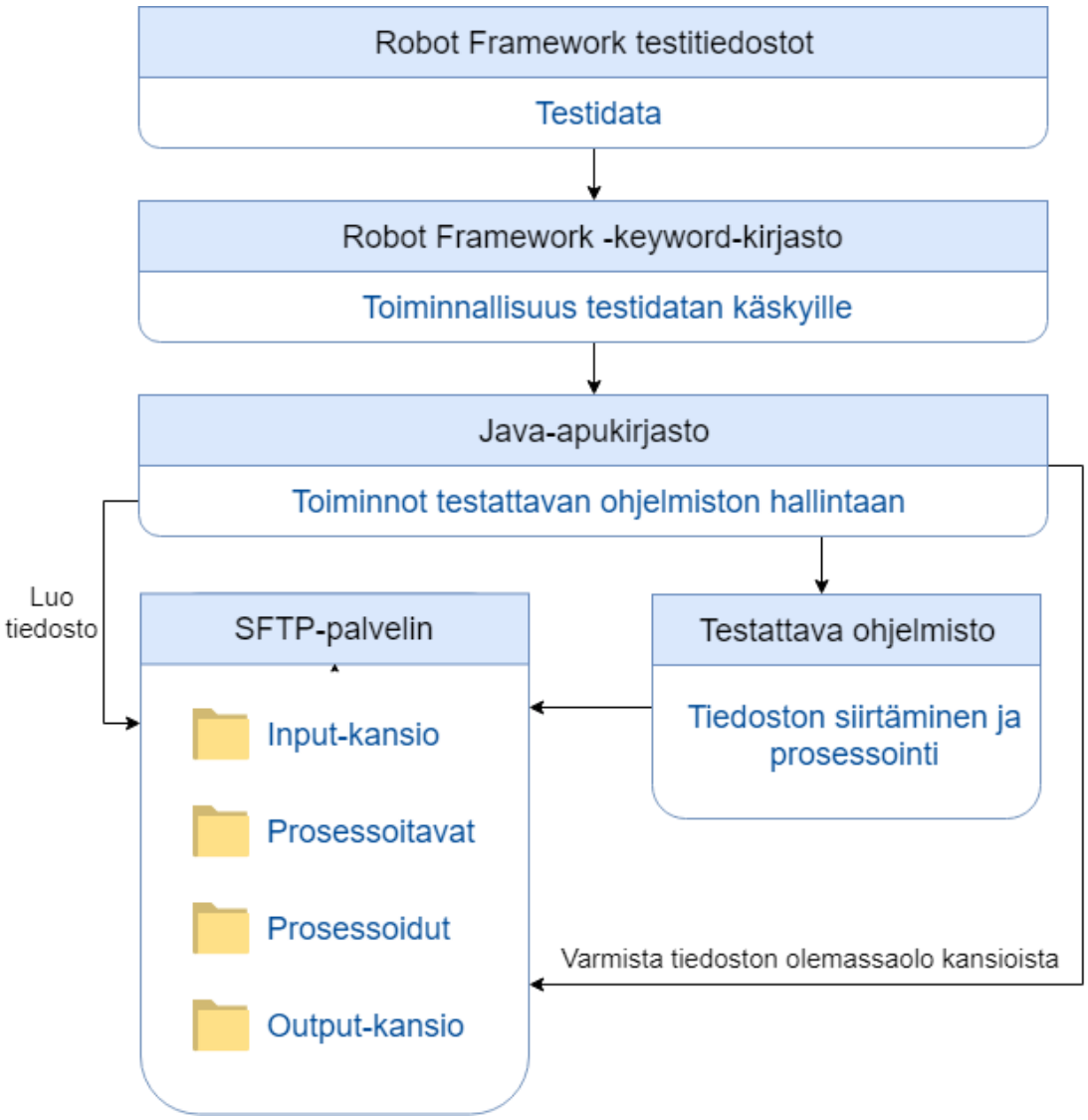


KUVIO 1. ALLOY ja *Managed File Transfer* -prosessi.

2.3 Testin vaatimukset

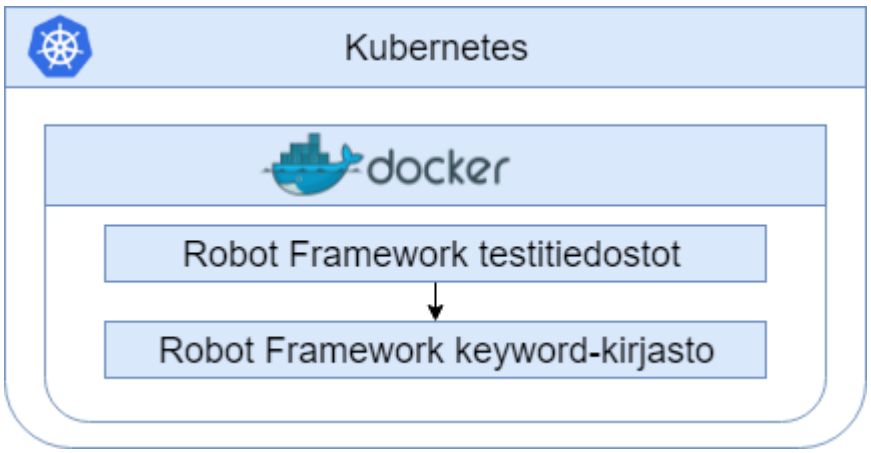
Soveltuvuusselvityksen kohteeksi valitaan edellä esitellyn ALLOY-ohjelmiston MFT-toiminnallisuuden testaus. Koska varsinaista asiakasta ei testitilanteessa ole olemassa, toteutetaan asiakkaan ja tämän kumppanin SFTP-kansiot ALLOY:n itsensä sisälle omiin kansioihinsa. Niihin viitataan myöhemmin nimillä *input*-kansio (asiakas) ja *output*-kansio (kumppani). Testi tulee aloittaa siirtämällä *input*-kansioon testiä varten luotava mielivaltainen datatiedosto. Sen jälkeen ohjelmistoa kutsutaan hakemaan tiedostot *input*-kansiosta prosessoitavien tiedostojen kansioon. Sitten ohjelmistosta käynnistetään prosessointitoiminto, joka siirtää tiedoston prosessoitavista tiedostoista prosessoitujen tiedostojen kansioon. Lopuksi prosessoitu tiedosto siirretään *output*-kansioon, eli todellisessa käyttötilanteessa asiakkaan yhteistyökumppanin kansioon, josta ulkopuolinen tekijä voisi sen halutessaan noutaa. Tiedoston prosessoinnin sisällöllä ei tässä testissä ole väliä, vaan testin tarkoituksena on nimenomaan varmistaa tiedoston oikea liikkuminen kansiosta toiseen ohjelmiston eri toimintoja, kuten prosessointia, aktivoimalla.

Varsinainen testitiedosto tulee toteuttaa ajettavaksi Robot Framework -työkalulla, ja testitiedoston toiminnallisuudet tulee sisällyttää Java-ohjelmointikielellä toteutettavaan uuteen Robot Framework -avainsanakirjastoon. Testin ympäristönä toimii itse testitiedostojen ja kirjastojen lisäksi ALLOY-ohjelmisto SFTP-palvelimineen. Robot Framework -testin tulee suorittaa käskyjä, jotka sille toteutettavan avainsanakirjaston (keyword-kirjasto) sekä yrityksen oman Java-apukirjaston avulla manipuloivat tiedostoa palvelimella ohjelmiston kautta. Testiympäristön rakenne ja testin osien väliset suhteet on esitelty kuviossa 2.



KUVIO 2. Robot Framework -soveltuvuusselvityksen testiympäristö

Kun testi on lopulta saatu ajettua läpi Robot Frameworkilla paikallisesti, tulee testiajo vielä siirtää suoritettavaksi Kubernetes-ympäristöön. Testiajo tulee säiliöidä Docker-työkalun avulla tehtyyn säiliöön, joka sitten ajetaan säiliöiden orkestrointia varten kehitetyllä Kubernetes-työkalulla yrityksen omalla Kubernetes-palvelimella. Kuberneteksen, Docker-säiliön, Robot Framework -testin ja sitä varten toteutettavan avainsanakirjaston väliset suhteet on esitetty kuviossa 3.



KUVIO 3. Kubernetes, Docker ja Robot Framework -testaus.

2.4 Käytettävät työkalut

Soveltuvuusselvityksen mukaisen testaamisen suorittamiseksi työssä tulee käyttää seuraavia työkaluja:

- Robot Framework
- Java
- Gradle
- Docker
- Kubernetes

Robot Framework -testiä varten toteutettava avainsanakirjasto tulee toteuttaa Java-ohjelmointikielellä käyttäen samalla projektinhallinnassa Gradle-työkalua. Itse ajettava testi toteutetaan tekstimuodossa millä tahansa tekstieditorilla ja se ajetaan lopuksi Python-työkalun Robot Framework -liitännäisellä. Kun testi on saatu valmiiksi, se säiliöidään Docker-työkalulla, jonka jälkeen toteutetaan Kubernetes-ohjelmistolla testisäiliöiden luonnin ja tuhoamisen orkestrointi.

3 TYÖKALUT JA OHJELMISTOT

3.1 Robot Framework

Robot Framework on Python-koodikielellä toteutettu työkalu ohjelmistojen ja verkkosivustojen hyväksymistestaukseen *keyword-driven testing* -menetelmällä. Se perustuu Pekka Klärckin tekemään diplomityöhön (P. Laukkanen, 2006) ja sen ensimmäinen versio julkaistiin Nokia Networksin toimesta samana vuonna. Myöhemmin projekti julkaistiin avoimena lähdekoodina GitHubissa, jossa sitä on forkattu jo lähes tuhat kertaa. Pekka on itsekin edelleen aktiivisesti mukana kehitystyössä, ja Robot Frameworkin ympärille on rakennettu *Robot Framework foundation* -niminen yhteisö varmistamaan kehityksen jatkuminen. Robot Frameworkia kohtaan on osoitettu viime vuosina paljon kiinnostuneisuutta sen monipuolisuuden ja sillä luotujen testien helppolukuisuuden vuoksi.

Robot Frameworkilla toteutetut testitiedostot (*test suite*) ovat tekstipohjaisia kooditiedostoja, jotka ajetaan Python-tulkin läpi käyttäen Pythonin Robot Framework -liitännäistä. Tiedostot voivat sisältää yhden tai useamman eri testivaiheen (*test case*), jotka puolestaan sisältävät kokoelman avainsanoja (*keyword*). Avainsanoilla kutsutaan testitiedostoon importatuista kirjastoista toimintoja, joilla joko suoritetaan toiminnallisuuksia, tai varmistetaan testattavan sovelluksen tai ympäristön tiloja ja muuttujien arvoja. Lopulta näiden toimintojen perusteella päätetään, onnistuiko vai epäonnistuiko testi.

Robot Framework sisältää suuren valikoiman valmiita avainsanoja hyväksymistestausta varten, mutta myös omien avainsanojen luonti on mahdollista, usein jopa tarpeellista. Valmiit avainsanat on toteutettu itse frameworkin tapaan Python-ohjelmointikielellä, mutta uusia avainsanoja voidaan käyttäjän niin halutessa toteuttaa lähes millä tahansa ohjelmointikielellä, kuten esimerkiksi Javalla (Robot Framework, 2018). Tällöin vaatimuksena kuitenkin on, että kirjasto toteutetaan aktiivisena etäkirjastopalvelimena pelkkien kooditiedostojen sijaan. Robot Framework valittiin työkaluksi tähän opinnäytetyöhön johtuen siitä, että sen avulla halutaan mahdollistaa uusien testien luominen myös vähemmän tekniselle henkilökunnalle. Kun kaikki tarvittavat avainsanat on toteutettu kirjastoihin, itse testien kirjoittaminen on huomattavasti Java-pohjaisten testien kirjoittamista yksinkertaisempaa ja suoraviivaisempaa.

3.2 Java

Java on Sun Microsystemsin vuonna 1995 julkaisema ohjelmointikieli, jota käytetään nykyään jo miljardeissa laitteissa aina matkapuhelimista supertietokoneisiin (Oracle, 2013). Se on hyvin monipuolinen ja alustariippumaton olio-ohjelmointia tukeva ohjelmointikieli, jonka mukana toimitetaan erittäin laaja standardikirjasto kaikkien yleisten ohjelmistotarpeiden täyttämiseen. Toisin kuin esimerkiksi suosittu C++-ohjelmointikieli, Javan koodi pitää ensin kääntää tavukoodiksi, joka sitten tulkataan ajamalla se Javan virtuaalikoneessa. Tulkkauksessa saavutetaan hyötynä edellä mainittu laitteistoriippumattomuus sekä muistinhallintaa auttavat valmiit työkalut, mutta haittapuolena tulevat ohjelmiston suuremmat tietoturvariskit (S. Poole, 2018) ja koodin tulkkaukseen kuluva ylimääräinen aika.

Liaison Technologies Oy on toteuttanut osan ohjelmistotestauksestaan käyttämällä TestNG-frameworkia, jossa itse testit on kirjoitettu puhtaalla Java-koodilla. Täten heillä löytyy jo suuri valikoima testejä varten toteutettuja Java-apukirjastoja, joita halutaan hyödyntää myös Robot Framework -testauksessa. Tästä johtuen Javan valinta työkaluksi tähän projektiin oli itsestään selvää, sillä vanhoja, jo luotuja apukirjastoja halutaan käyttää hyödyksi mahdollisimman tehokkaasti.

3.3 Gradle

Gradle on vuonna 2007 julkaistu avoimeen lähdekoodiin perustuva työkalu ohjelmistoprojektien buildaamiseen ja riippuvaisuuksien hallintaan. Se pohjautuu suosittuihin Apache Ant ja Apache Maven -työkaluihin, mutta tarjoaa ohjelmoitavan Groovy-rajapinnan entisen XML-tyyppisen build-määrittelyn sijaan. Se kehitettiin erityisesti usean projektin sisältävien kokonaisuuksien build-vaihetta varten, sillä se sisältää tuen niin kutsutulle inkrementaaliselle buildaamiselle, jossa kokonaisprojektista buildataan vain ne osat, joita on päivitetty (Gradle, 2015).

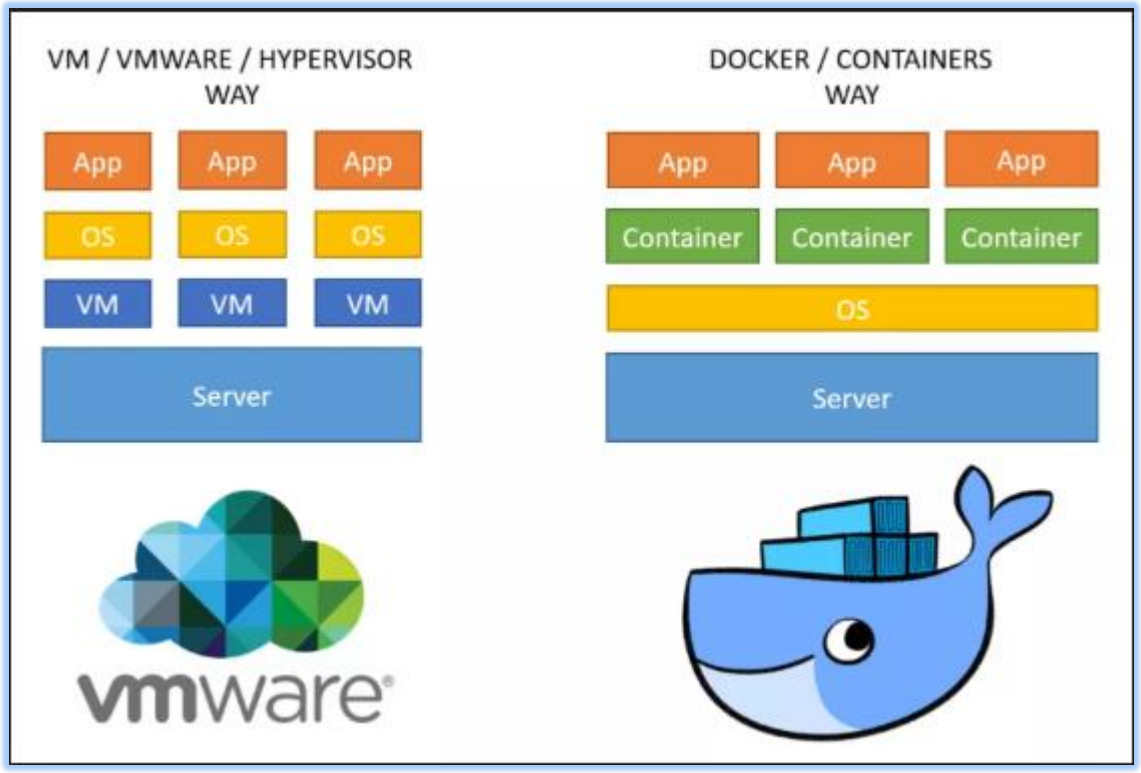
Gradle on noussut monipuolisuutensa, vanhojen hyvien mallien päälle rakentamisensa sekä avoimen lähdekoodinsa vuoksi erittäin suureen suosioon. Se on muun muassa vakiona mukana Android Studio -työkalussa, joka on Android-käyttöjärjestelmän virallinen

kehitysympäristö. Androidin osuus nykypäivän mobiilimarkkinoista on 87% (IDC Corporate USA, 2018) ja sovelluksia on saatavana yli 2 miljoonaa kappaletta. Gradlen käyttö ja eteenpäin kehitys on siis jatkuva prosessi, jonka takana on miljoonia ohjelmistokehittäjiä.

Gradlea on käytetty Liaison Technologies Oy:ssä useiden projektien buildaamiseen, joten on luonnollista, että se otetaan käyttöön myös tässä soveltuvuusselvityksessä. Gradlea käytetään vain Robot Framework -avainsanakirjaston buildaamiseen ja ajamiseen, sillä itse testit sisältävä kansio voidaan ajaa sellaisenaan käyttämällä Pythonin Robot Framework -liitännäistä.

3.4 Docker

Docker on vuonna 2013 julkaistu ohjelmisto, joka mahdollistaa käyttöjärjestelmätason virtualisoinnin perinteisiä virtuaalikoneita tehokkaammin. Se ei ole virtuaalikonetta nopeampi toiminnoissaan, mutta käyttää isäntäkoneensa resursseja hyväksi huomattavasti paremmalla hyötysuhteella muun muassa muistin varauksen suhteen (Container Journal, 2016). Dockerin avulla voidaan luoda imageihin perustuvia säiliöitä, joiden sisällä voidaan ajaa käytännössä mitä tahansa ohjelmistoja ja konfiguraatioita. Säiliöihin voidaan konfiguroida yhteysasetuksia, joilla määritellään miten säiliöt kommunikoivat muiden säiliöiden ja tietokoneiden kanssa. Kaikki säiliöt ajetaan saman käyttöjärjestelmätason päällä, toisin kuin perinteisissä virtuaalikoneissa (kuvio 4), mikä säästää paljon resursseja usean säiliön toteutuksissa.



KUVIO 4. Tavallisen virtuaalikoneen ja Dockerin ero (https://www.e4developer.com/2018/01/18/microservices-toolbox-docker/)

Dockerin säiliöt perustuvat imageihin, joita luodaan *Dockerfile*-tiedostojen avulla. Niissä määritellään koodiperäisesti se, mitä pohjaimagea luotava image käyttää (esim. Alpine-Linux-jakelu) sekä sille suoritettavat komennot (Docker, 2018). Komennoilla voidaan asentaa säiliössä tarvittavat ohjelmistot samaan tapaan, kuin ne asennettaisiin komentoriviltä. Tämän lisäksi on mahdollista siirtää tiedostoja isäntäkoneen levyltä suoraan imagen sisään säiliön käytettäväksi. Kun *Dockerfile*-tiedosto on valmis, Docker-ohjelmistolla voidaan rakentaa siitä pysyvä imagetiedosto, joka sitten merkitään tagilla vastaamaan esimerkiksi säiliössä ajettavan kokoonpanon versionumeroa. Tämän jälkeen image on ajettavissa missä tahansa Docker-imageja tukevassa palvelussa niin, että heti käynnistyessään kaikki *Dockerfilessä* määritellyt ohjelmistot, konfiguraatiot ja käskyt on jo toteutettuna säiliön sisällä. Docker-imageja voidaan helposti verrata virtuaalikoneiden imageihin, mutta pääerona on se, että käyttöjärjestelmää ei tarvitse virtualisoida erikseen jokaista säiliötä varten.

Palveluiden säiliöiminen mahdollistaa niiden helpon hallinnan ja skaalauksen esimerkiksi Kubernetes-orkestrointijärjestelmässä. Etenkin mikropalveluihin perustuvat ohjelmistot hyötyvät Docker-tyyppisestä säiliöinnistä virtuaalikoneisiin verrattuna: suuri ohjelmisto

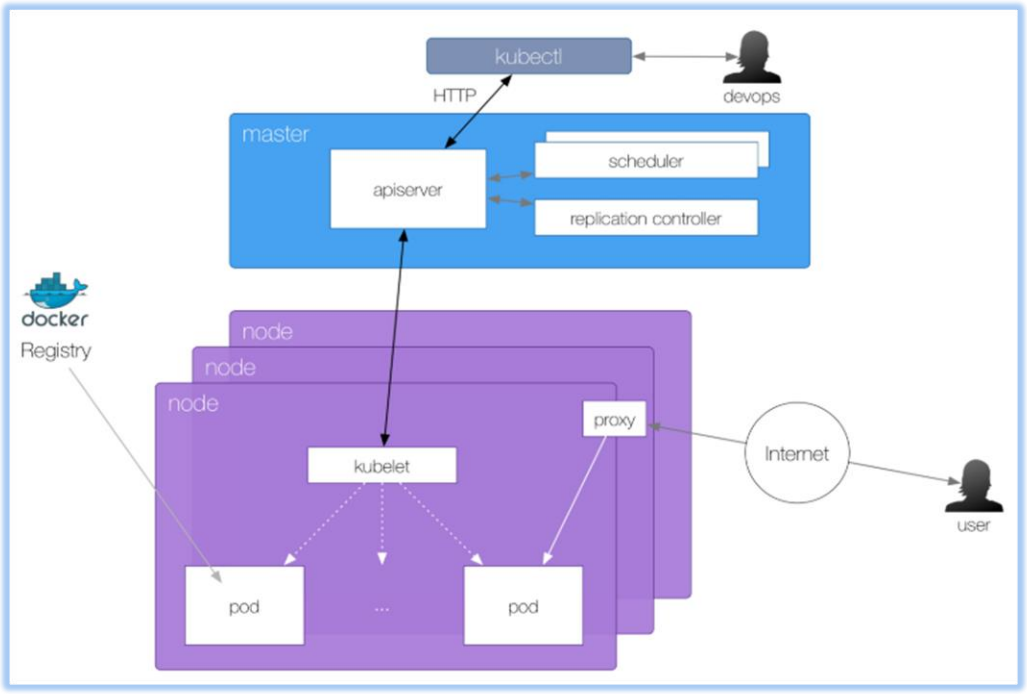
voidaan jakaa lukuisiin pienempiin komponentteihin, jotka sitten ajetaan omissa säiliöissään, ja joille mahdollistetaan kommunikointi toistensa kanssa. Kun yhtä ohjelmistokomponenttia päivitetään, ei tarvitse buildata koko ohjelmiston imagea uudelleen, vaan riittää että kyseisen komponentin säiliö päivitetään. Kaikki säiliöt voivat jakaa saman virtuaalisen käyttöjärjestelmän, jolloin komponenttien päivitys tapahtuu erittäin nopeasti (Ruby Garage, 2018).

Ohjelmistotestaamisessa käytettävät säiliöt halutaan yleensä tuhota testiajojen jälkeen, eikä niille siten ole järkeä toteuttaa kokonaan omaa käyttöjärjestelmätasoa. Niitä varten kevyet Docker-säiliöt ovat optimaalinen ratkaisu. Docker-imageen voidaan jo rakennusvaiheessa siirtää ja asentaa kaikki tarvittavat testitiedostot ja ohjelmistot testien ajamista varten. Sen jälkeen aina kun testattaviin ohjelmistoihin tulee päivityksiä, testiajot voidaan käynnistää yksinkertaisesti ajamalla Docker-image tietyillä parametreilla. Se toimii aina täsmälleen samalla tavalla ja se voidaan ajaa missä tahansa Docker-imageja tukevassa ympäristössä. Kun testiajot on suoritettu loppuun, säiliö voidaan tuhota yksinkertaisesti ja nopeasti ja rakentaa uudelleen yhtä helposti seuraavaa testiajoa varten.

3.5 Kubernetes

Kubernetes on alun perin Googlen suunnittelema vuonna 2014 julkaistu avoimeen lähdekoodiin perustuva ohjelmisto säiliöiden automaattista orkestrointia varten. Se mahdollistaa säiliöityjen ohjelmistojen ajamisen, hallinnan ja skaalauksen perustuen eri tyyppisiin konfiguraatioihin (Red Hat, 2018). Kubernetes tukee lukuisia eri säiliöintityökaluja, kuten esimerkiksi Dockeria, joten se sopii mainiosti työkaluksi tässä opinnäytetyössä toteutettavia testiajoja varten.

Kuviossa 5 on esitelty Kuberneteksen arkkitehtuurin pääpiirteet. Kubernetes sisältää yhden master-noodin sekä yhden tai useamman slave-noodin podeja varten. Podit ovat ikään kuin säiliöitä säiliöille, eli yhdessä podissa voidaan ajaa yhtä tai useampaa säililötä.
Kun kehittäjä (kuviossa 5 *devops*) lähettää kubectl-työkalulla master-noodin apiserverille uuden käskyn, master-noodi ottaa yhteyttä slave-noodiin ja antaa käskyn siellä olevalle kubelet-kontrollerille. Kubelet voi tämän jälkeen käskystä riippuen esimerkiksi luoda uuden podin testiajoja varten tarkoitetulle säiliölle. Kubelet osaa automaattisesti hakea konfiguraatiotiedoston perusteella oikean imagen ajettavaksi podin sisällä.



KUVIO 5. Kubernetes-arkkitehtuuri (https://mesosphere.com/blog/kubernetes-and-the-dcos/)

Jatkuvasti päällä pysyvissä toteutuksissa, kuten esimerkiksi web-palvelimissa, Kubelet voidaan konfiguroida varmistamaan, että podin sisältö on aina saatavilla. Tämä toteutetaan replikoimalla sama sisältö useaan eri podiin, jolloin yhden podin kaatuminen ei vaikuta palvelimen toimintaan. Myös palvelimen päivitys voidaan tällöin toteuttaa *rolling update* -menetelmällä, eli päivittämällä palvelimen ohjelmisto yhteen podiin kerrallaan. Silloin palvelun käyttäjän näkökulmasta palvelun saatavuus on täysin jatkuva, ja kun kaikki podit on saatu päivitettyä, kaikille käyttäjille tarjoillaan uusimman version mukaista sisältöä. Podeihin saadaan yhteys ulkomaailmasta käyttämällä jokaisesta noodista löytyvää välityspalvelinta (kuvion 5 *proxy*).

Koska tässä opinnäytetyössä toteutetaan ohjelmiston testaaminen, ei podeja tarvitse luoda kuin yksi. Se konfiguroidaan *job*-tyyppiseksi, mikä kertoo Kubeletille, että kyseisen podin sisältö halutaan saada ajettua kokonaan läpi, eikä sen ole tarkoitus jäädä pysyvästi päälle. Myöskään yhteysasetuksia podiin ei tarvitse määrittää, sillä testiajot on kokonaisuudessaan määritelty testitiedostojen sisällä, eikä podille tai sen säiliöille tarvitse antaa ulkopuolisia käskyjä ennen ajoa, tai sen aikana.

4 ROBOT FRAMEWORK -TESTI

Tässä osiossa toteutetaan Robot Framework -testitiedostot kappaleessa 2 määritellyn soveltuvuusselvityksen esimerkkitestin suorittamiseksi. Testin kehitys aloitetaan paloittelemalla testi eri osavaiheisiin, jonka jälkeen luodaan kaavio kuvaamaan testirakennetta. Lopuksi kaavion ja testin vaiheiden perusteella kirjoitetaan puhtaaksi Robot Frameworkilla ajettava testitiedosto sekä muuttujia varten luotava resurssitiedosto. Testitiedostojen valmistuttua siirrytään toteuttamaan testin avainsanat mahdollistava ulkoinen etäkirjasto, sillä Robot Frameworkin sisäänrakennetut avainsanat eivät kata tämän testin tarpeita. Ennen varsinaisten testitiedostojen luontia on hyvä ymmärtää Robot Framework -koodin syntaksin perusteet, joiden teoriapohja on esitetty kokonaisuudessaan Robot Frameworkin käyttöoppaassa (Robot Framework, 2018, kappale 2.1).

4.1 Robot Framework -syntaksi

Robot Framework -tiedostojen syntaksi on tabulaarinen, eli kirjastot, resurssit, avainsanat ja niiden parametrit pitää erottaa toisistaan vähintään kahdella välilyönnillä. Sen lisäksi testien avainsanakutsut pitää sisentää erilleen alkusarakkeesta ja kirjoittaa aina kokonaisuudessaan yhdelle riville. Koodissa 1 on esitelty Robot Framework -testitiedoston eri osiot oikean syntaksin mukaisesti. Koodi sisältää kaksi eri osiota: *Settings* ja *Test Cases*. *Settings*-osiossa on määritelty mitä kirjastoja ja resurssitiedostoja testissä halutaan käyttää, ja *Test Cases* -osiossa testitiedoston eri testivaiheet.

```
*** Settings ***
Library    Remote           http://localhost:8270/
Resource   resource.robot

*** Test Cases ***
First test case
    Keyword Call           Parameter 1    Parameter 2
    Another Keyword        ${PARAMETER_FROM_RESOURCE}    Parameter 2

Second test case
    Yet Another Keyword    Parameter 1    &{ANOTHER_RESOURCE_PARAMETER}
```

KOODI 1. Robot Framework -testitiedoston syntaksi

Esimerkkikoodin *Settings*-osiossa määritellään testissä käytettäväksi osoitteessa *http://localhost:8270/* sijaitseva ulkoinen etäkirjasto sekä *resource.robot*-niminen resurssitiedosto. *Test Cases* -osiossa koodin ensimmäisestä sarakkeesta alkavat sinisellä värjätyt lausekkeet merkitsevät aina uutta testivaihetta ja itse teksti sen nimeä. Testien alle on listattu kyseisessä testissä suoritettavat avainsanakutsut parametreineen. Parametrit voidaan antaa suoraan tekstinä, kuten esimerkin *Keyword Call* -avainsanalle on annettu, tai perustuen muuttujien arvoihin, kuten *Another Keyword* ja *Yet Another Keyword* -avainsanojen tapauksessa.

Koodissa 2 on esitelty esimerkin *resource.robot*-resurssitiedoston sisältö. Testissä käytettävät muuttujat toteutetaan *Variables*-osion sisälle ja koodin syntaksi on varsinaisen testitiedoston tapaan tabulaarinen. Esimerkissä on käytetty kahta eri tyyppistä muuttujaa: *$*-merkillä luotuja skalaarimuuttujia ja *&*-merkillä luotuja sanakirjamuuttujia. Skalaarimuuttujilla on aina vain yksi tietty arvo kerrallaan, kun taas sanakirjamuuttujat sisältävät listan avain-arvo-pareja. Robot Framework tukee myös @-merkillä luotavia listamuuttujia, mutta niille ei tässä työssä ole tarvetta.

```
*** Variables ***
${PARAMETER_FROM_RESOURCE}      Scalar parameter value
&{ANOTHER_RESOURCE_PARAMETER}   key=value    anotherKey=anotherValue
```

KOODI 2. Robot Framework -resurssitiedoston esimerkkitapaus.

Kaikki koodissa 2 määritellyt muuttujat ovat käytettävissä missä tahansa Robot Framework -testitiedostossa, joka sisällyttää kyseisen resurssitiedoston mukaan testeihin tiedoston alussa. Muuttujat voidaan toteuttaa myös itse testitiedostoon luotavaan *Variables*-osioon, mutta jotta itse testitiedosto pysyisi siistinä, tässä työssä muuttujat toteutetaan aina erilliseen tiedostoon.

4.2 Varsinaisen testin jakaminen vaiheisiin

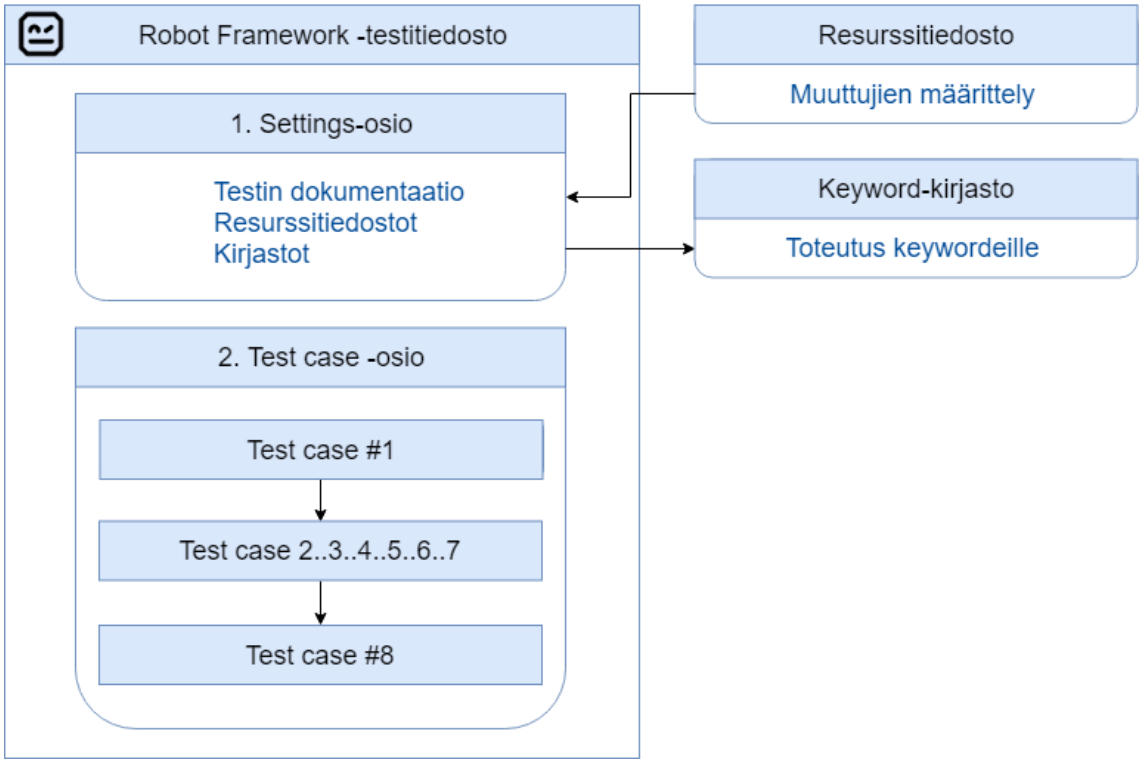
Perustuen kappaleessa 2 määriteltyyn soveltuvuusselvitykseen, voidaan SFTP-palvelinta hallitsevan ALLOY-ohjelmiston testaus jakaa seuraaviin vaiheisiin:

1. Luo tiedosto SFTP-palvelimen *input*-kansioon.
2. Varmista tiedoston olemassaolo *input*-kansiosta.
3. Aktivoi tiedoston siirto prosessoitavat-kansioon kutsumalla ohjelmistoa.
4. Varmista tiedoston olemassaolo prosessoitavat-kansiosta
5. Aktivoi tiedoston prosessointi kutsumalla ohjelmistoa.
6. Varmista tiedoston olemassaolo prosessoidut-kansiosta
7. Aktivoi tiedoston siirto *output*-kansioon kutsumalla ohjelmistoa.
8. Varmista tiedoston olemassaolo *output*-kansiosta.

Testin eri vaiheet toteutetaan Robot Framework -testitiedoston sisällä *Test Cases* -osioon niitä varten luotuihin erillisisiin testivaiheisiin.

4.3 Testitiedostojen rakenne

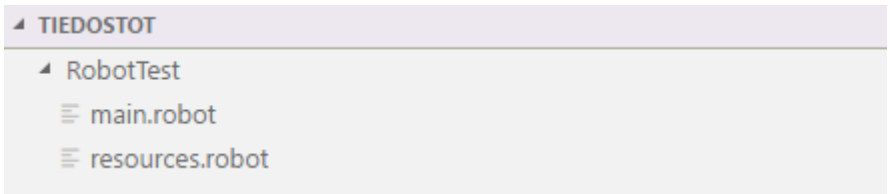
Kuvio 6 ilmaisee luotavien Robot Framework testitiedostojen rakenteen sekä tiedostojen ja kirjastojen väliset suhteet. Testitiedoston sisälle merkityn *Test case* -osion sisällä olevat kahdeksan *test casea* kuvaavat kappaleessa 4.2 lueteltuja vaiheita. Jokaista vaihetta varten on varattu yksi *test case* ja ne suoritetaan synkronisesti järjestyksessä yksi kerrallaan.



KUVIO 6. Robot Framework -testitiedostojen rakenne.

4.4 Testitiedostojen luonti

Testin kehitys aloitetaan luomalla soveltuvuusselvityksessä tarvittavat testitiedostot. Luodaan ensin uusi kansio nimeltä *RobotTest* polkuun *C:\Tiedostot*. Lisätään sitten *RobotTest*-kansioon kaksi eri testitiedostoa: *main.robot* ja *resources.robot*. *main.robot* tulee sisältämään kaikki testin ajamiseen liittyvät kirjastosisällytykset, testivaiheet ja avainsanakutsut, kun taas *resources*-tiedostossa määritellään tarvittavat muuttujat. Kuviossa 7 on esitelty lopputuloksena syntynyt kansio- ja tiedostorakenne.



KUVIO 7. Robot Framework -testin tiedostopuu.

4.4.1 main.robot

Varsinaiseen testitiedostoon, eli *main.robot*-tiedostoon luodaan kaksi eri osiota: *Settings*-osio kirjastojen ja resurssitiedostojen sisällytystä varten sekä *Test Cases* -osio varsinaisia testitapauksia ja avainsanakutsuja varten. Robot Framework testin kehitys aloitetaan lisäämällä tiedoston alkuun *Settings*-osioon viittaus käytettävään avainsanakirjastoon sekä resurssitiedostoon alla olevan koodin 3 mukaisesti. Osion ensimmäinen rivi kertoo Robot Frameworkille, että testissä halutaan ottaa käyttöön ulkoisen, osoitteessa *http://localhost:8270/* sijaitsevan etäkirjaston avainsanat. Seuraavalla rivillä testissä otetaan käyttöön *resources.robot*-tiedostossa olevat muuttujat.

```
*** Settings ***
Library    Remote              http://localhost:8270/
Resource   resources.robot
```

KOODI 3. Avainsanakirjaston ja resurssitiedoston sisällytys testitiedostossa.

Seuraavaksi lisätään koodin 4 mukaisesti tiedostoon *Test Cases* -osion aloitusmerkintä sekä testin ensimmäinen vaihe, jossa luodaan tiedosto palvelimen *input*-kansioon. Ensin suoritetaan tiedoston luonti *Sftp Create File* -avainsanalla, jolle annetaan parametreina kirjautumistunnukset, luotava tiedosto, tiedostopolku palvelimella ja palvelimen osoite. Sitten odotetaan korkeintaan kolme minuuttia *Verify File Exists* -avainsanaa palauttamaan ilmoitus siitä, että tiedosto tosiaan löytyy palvelimelta. Nämä testivaiheet kattavat kaksi ensimmäistä kahdeksasta testin osasta.

```
*** Test Cases ***
Create file into SFTP server's input folder
  Sftp Create File  ${CREDENTIALS}  ${RESOURCE}  ${INPUT_FOLDER}
${SERVER}

Verify file exists in SFTP server’s input folder
  Wait Until Keyword Succeeds  3min  5s  Verify File Exists
${CREDENTIALS}  ${FILENAME}  ${INPUT_FOLDER}  ${SERVER}
```

KOODI 4. Tiedoston luonti *input*-kansioon ja olemassaolon varmistus.

Seuraavissa testivaiheissa (koodi 5) kutsutaan *Download File* -avainsanalla ohjelmistoa siirtämään tiedosto *input*-kansiosta prosessoitavien tiedostojen kansioon ja varmistetaan jälleen tiedoston löytyminen kansiosta siirron jälkeen.

```
Move file from the input folder to the process folder
  Download File

Verify file exists in SFTP server’s process folder
  Wait Until Keyword Succeeds  3min  5s  Verify File Exists
${CREDENTIALS}  ${FILENAME}  ${PROCESS_FOLDER}  ${SERVER}
```

KOODI 5. Tiedoston siirto ohjelmistokomponentin avulla.

Kun tiedosto on siirretty prosessoitavien tiedostojen kansioon, kutsutaan seuraavaksi ohjelmiston prosessointifunktiota. Prosessointifunktio siirtää tiedoston prosessoitavien tiedostojen kansiosta prosessoitujen tiedostojen kansioon. Tässä vaiheessa tiedosto ajetaan ohjelmiston prosessointitoiminnallisuuden läpi, ja oikeissa asiakastapauksissa siihen voitaisiin tehdä mitä tahansa muutoksia, mutta soveltuvuusselvityksessä tälle ei ole tarvetta. Koodissa 6 esitellään kappaleen 4.2 osavaiheiden 5 ja 6 testitoteutus.

```
Process the file and move it to the processed folder
  Process File

Verify file exists in the processed folder
  Wait Until Keyword Succeeds  3min  5s  Verify File Exists
${CREDENTIALS}  ${FILENAME}  ${PROCESSED_FOLDER}  ${SERVER}
```

KOODI 6. Tiedoston prosessointi ja siirto prosessoitujen tiedostojen kansioon

Testin viimeinen vaihe kattaa tiedoston siirron prosessoitujen tiedostojen kansiosta *output*-kansioon, josta kuka tahansa voisi sen noutaa valmiiksi prosessoituna. Koodissa 7 esitelty vaihe kattaa testin osavaiheet 7 ja 8.

```
Move the file to the output folder
  Upload File

Verify file exists in the output folder
  Wait Until Keyword Succeeds  3min  5s  Verify File Exists
${CREDENTIALS}  ${FILENAME}  ${OUTPUT_FOLDER}  ${SERVER}
```

KOODI 7. Tiedoston siirto *output*-kansioon ja olemassaolon varmistus

Edellä esitellyt koodit 3-7 kattavat koko *main.robot*-testitiedoston sisällön, joka on kokonaisuudessaan esitetty tämän työn liitteessä 1.

4.4.2 resources.robot

Kaikki *main.robot*-tiedoston muuttujat pitää määritellä *resources.robot*-tiedostossa, jotta Robot Framework osaa käyttää niitä testiajoissa. Resurssitiedostoon lisätään ensin *Variables*-osio, jonka alle voidaan määritellä tarvittavat muuttujat koodin 8 mukaisesti.

```
*** Variables ***
&{CREDENTIALS}
...                 username=test_user
...                 password=p455word!9

${RESOURCE}         common/test_file.txt
${FILENAME}         test_file.txt

${SERVER}           sftp://liaison.dev:22
${INPUT_FOLDER}     INPUT
${PROCESS_FOLDER}   PROCESS
${PROCESSED_FOLDER}  PROCESSED
${OUTPUT_FOLDER}    OUTPUT
```

KOODI 8. *resources.robot* -tiedoston sisältö.

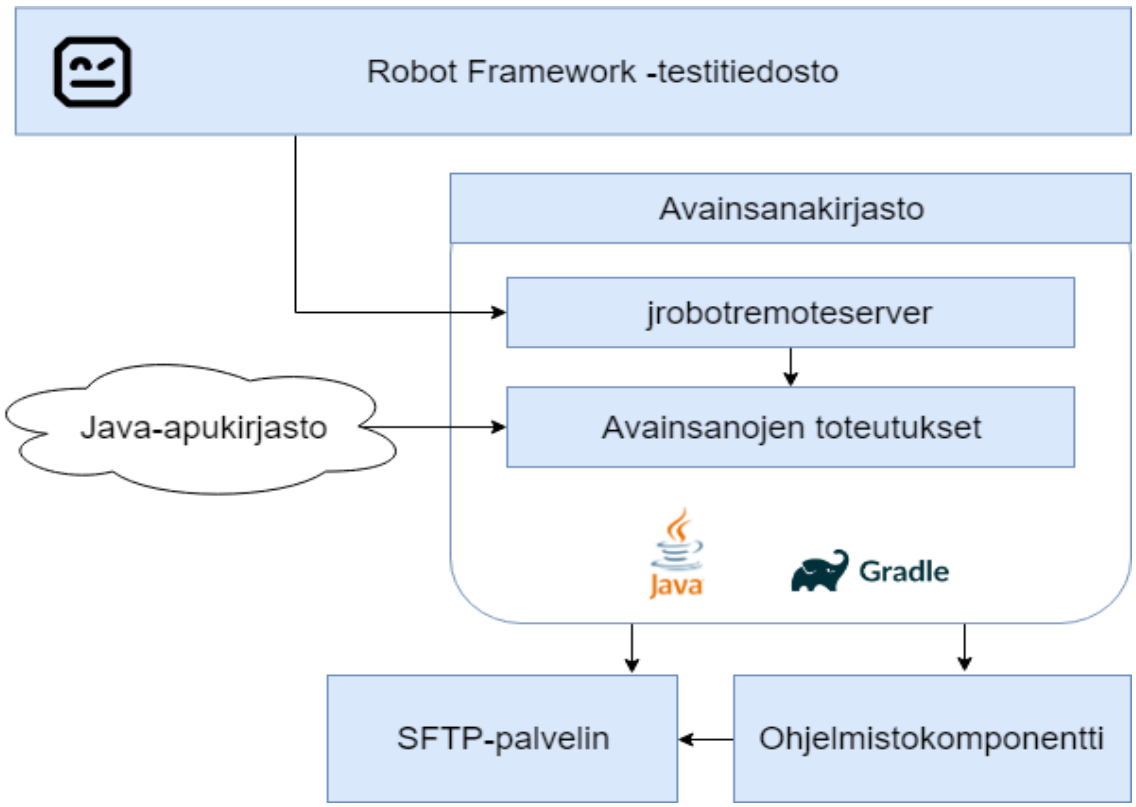
Koodi sisältää SFTP-palvelimen osoitteen ja käyttäjätunnuksen, testissä siirreltävän tiedoston sijainnin Java-apukirjaston sisällä, käytettävän tiedostonimen ja kaikki testissä tarvittavat kansionimet. Kooditiedosto on sisällytetty työn liitteisiin liitteenä 2.

5 ROBOT FRAMEWORK -AVAINSANAKIRJASTO

Robot Framework -testeissä käytettävät funktiokutsut, eli avainsanat, on aina toteutettu erillisiin avainsanakirjastoihin. Robot Framework itsessään sisältää valikoiman sisäänrakennettuja *built-in*-kirjastoja yleisimpiä tarpeita varten, mutta se tukee myös täydellisesti käyttäjien omien kirjastojen luontia. Kirjastot luodaan tavallisesti Python-ohjelmointikieltä käyttäen, sillä Robot Framework itsekin on toteutettu sillä. Uusia kirjastoja voidaan kuitenkin luoda millä tahansa ohjelmointikielellä, mikäli niihin viitataan testitiedostoissa aktiivisina etäkirjastoina. Etäkirjastojen toimintaperiaate on verrattavissa palvelimeen: kirjastosovellus käynnistetään ja se odottaa etäyhteyksiä Robot Framework -testeistä suorittaakseen niissä kutsuttavien avainsanojen toiminnallisuuden. Tässä kappaleessa toteutetaan avainsanakirjasto edellisessä kappaleessa luotua Robot Framework -testiä varten. Kaikki tarpeellinen informaatio myös avainsanakirjastojen toteuttamiseen on esitetty Robot Framework käyttöoppaassa (Robot Framework, 2018, kappale 4.1).

5.1 Kirjaston komponentit

Soveltuvuusselvityksen Robot Framework -avainsanakirjasto toteutetaan Java-ohjelmointikielellä. Kirjastoa varten luodaan uusi Java-projekti, jossa otetaan käyttöön yrityksen sisäisessä TestNG-tyyppisessä testauksessa käyttämä Java-apukirjasto sekä kolmannen osapuolen luoma Robot Framework etäyhteyden mahdollistava *jrobotremoteserver*-kirjasto. Projektinhallinnassa ja kirjastopalvelimen ajossa käytetään Gradle-työkalua. Kuviossa 8 on esitelty testitiedostojen, avainsanakirjaston, Java-apukirjaston, *jrobotremoteserverin* ja testattavan ohjelmiston väliset suhteet.



KUVIO 8. Robot Framework -avainsanakirjaston komponentit sekä testattava ohjelmisto.

5.1.1 Java-apukirjasto

Kirjastossa sisällytetään käytettäväksi yrityksen aiemmissa TestNG-tyyppisissä testeissä käyttämä Java-apukirjasto nimeltään *test-utils*. Kirjasto määritellään projektin *build.gradle*-tiedostossa ulkoisessa Maven-tyyppisessä repositoriossa sijaitsevaksi riippuvaisuudeksi, joka ladataan yrityksen palvelimelta ja rakennetaan build-vaiheessa avainsanakirjaston käytettäväksi. Java-apukirjastosta löytyy toiminnallisuudet tiedoston luomiseksi SFTP-palvelimelle sekä testattavan ohjelmiston hallintaan liittyvät funktiot. Kaikkia apukirjaston ominaisuuksia voidaan lopulta hyödyntää avainsanafunktioiden sisällä.

5.1.2 *jrobotremoteserver*-kirjasto

Jotta Robot Framework -testeistä saataisiin yhteys avainsanojen toiminnallisuudet tarjoavaan kirjastoon, siihen täytyy toteuttaa toiminnallisuus etäyhteyttä varten. *jrobotremoteserver* on kolmannen osapuolen toteuttama etäyhteyskirjasto Robot Framework etätestaamista varten Javaan perustuvissa avainsanakirjastoissa (GitHub, 2018). Kirjaston avulla voidaan aloittaa tietyn portin kuunteleminen mahdollisia testiyhteyksiä varten, ja kun yhteys on luotu, *jrobotremoteserver* tarjoaa listan saatavilla olevista avainsanoista, sekä välittää testiltä saamansa avainsanakutsut niitä vastaaville funktioille. *jrobotremoteserverin* ja testiajon välinen yhteydenpito toteutetaan XML-RPC-protokollaan perustuvalla yhteydellä. Tässä projektissa käytetään *jrobotremoteserverin* Github-repositorion jar-julkaisupaketin versiota 3.0.

5.2 Avainsanakirjaston toteutus

Kirjaston toteutus aloitetaan luomalla uusi kansio *RobotLibrary* polkuun *C:\Tiedostot*. Kyseisen kansion sisälle alustetaan uusi Java-projekti Gradle-työkalun avulla sekä lisätään riippuvaisuudet *jrobotremoteserveriin* ja Java-apukirjastoon. Lopuksi luodaan toteutukset Robot Framework -testitiedostossa kutsuttaville avainsanoille erilliseen kooditiedostoon, jonka rajapinnan *jrobotremoteserver* sitten tarjoaa yhteyden ottaville testiajoille.

5.2.1 Projektitiedostojen luonti

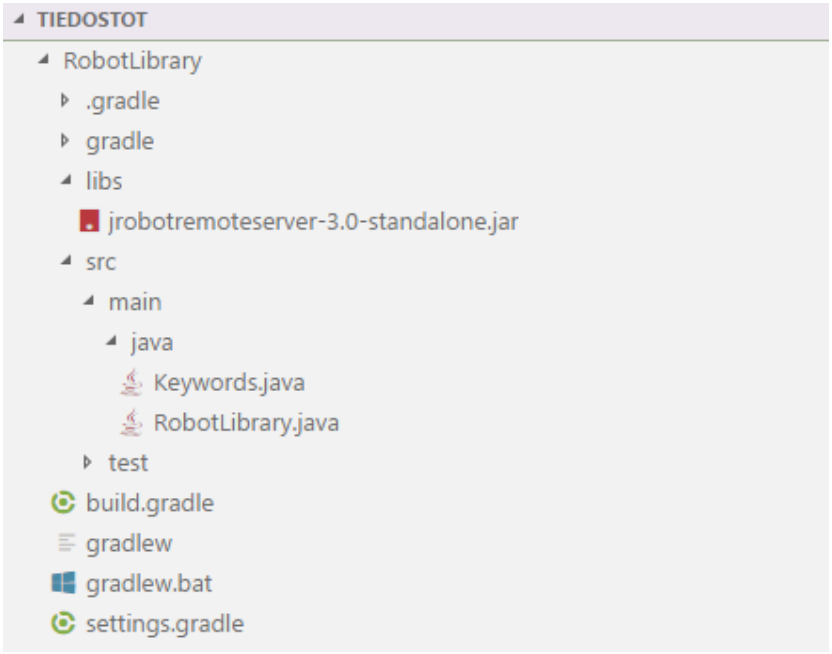
Luodaan Gradle-työkalulla uusi Java-projekti *RobotLibrary*-kansioon ajamalla koodin 9 mukainen komento komentorivillä.

```
gradle init --type java-application
```

KOODI 9. Uuden Java-projektin alustus Gradle-työkalulla.

Gradle luo kansioon automaattisesti muun muassa *build.gradle*-tiedoston sekä kansiopolut ja kooditiedostot Java-projektia varten. Nimetään *src/main/java*-kansiossa oleva *App.java* uudelleen *RobotLibrary.java*:ksi ja luodaan sen rinnalle uusi tyhjä *Keywords.java*-tiedosto avainsanatoteutuksia varten. Poistetaan toistaiseksi kokonaan

src/test/java-kansiossa oleva *AppTest.java*-yksikkötesti projektista. *build.gradle*-tiedosto tulee sisältämään kaiken projektiin liittyvän konfiguroinnin, kuten kirjastoriippuvaisuudet. *RobotLibrary.java* tulee toimimaan ohjelmiston pääluokkana, joka alustaa käyttöön sovelluksen *jrobotremoteserver*-palvelimen sekä *Keywords.java*-avainsanaluokan. Yrityksen oma Java-apukirjasto ladataan yrityksen palvelimella sijaitsevasta Maven-repositoriosta build-vaiheen aikana, mutta *jrobotremoteserveriä* varten pitää luoda RobotLibrary-kansioon uusi *libs*-kansio, johon Siirretään j*robotremoteserverin* GitHub-repositoriosta ladattava *jrobotremoteserver-3.0-standalone.jar*-paketti. *Tiedostot*-kansion hakemistopuun pitäisi suoritettujen toimenpiteiden jälkeen vastata kuviota 9.



KUVIO 9. Robot Framework -avainsanakirjaston hakemistopuu.

5.2.2 build.gradle

Gradle-projekteissa luodaan aina *build.gradle*-tiedosto, jossa määritellään miten kyseinen projekti tulee buildata ja suorittaa. Tässä tapauksessa sinne tulee lisätä viittaukset käytettäviin kirjastoihin, tieto ajon aikana suoritettavasta luokasta sekä viittaus yrityksen omaan ulkoiseen Maven-repositorioon, jossa Java-apukirjasto sijaitsee. Lisätään *build.gradle*-tiedostoon koodin 10 mukaisesti ajettavan luokan nimi sekä kirjastoja varten kaksi osiota: *repositories* ja *dependencies.*

```
mainClassName = 'RobotLibrary'

repositories {
    maven { url 'http://nexus.liaison.dev/repositories/' }
}

dependencies {
    compile files ('libs/jrobotremoteserver-3.0-standalone.jar')
    implementation "com.liaison:test-utils:4.0"
}
```

KOODI 10. *build.gradle*-tiedoston konfigurointi.

mainClassName-muuttujalla määritellään Gradlelle minkä luokkatiedoston *main*-funktio sen pitää ajaa, kun ohjelma halutaan suorittaa. Koska projektin päätiedoston nimi muutettiin, tulee myös *mainClassName*-muuttujan arvo päivittää. *Repositories*-osiossa Gradlelle ilmoitetaan mistä osoitteesta se voi etsiä tarvittavia kirjastoja ja d*ependencies*-osiossa vuorostaan määritellään projektin kirjastoriippuvaisuudet. *compile files* -komento kääntää *libs* kansiosta *jrobotremoteserver*-kirjaston projektin käyttöön ja *implementation*-kutsu kääntää yrityksen palvelimelta haettavan *test-utils*-Java-apukirjaston version 4.0 projektin käytettäväksi. Muita muutoksia *build.gradle*-tiedostoon ei tässä vaiheessa tarvita ja tiedoston koko sisältö on esitelty liitteessä 3.

5.2.3 RobotLibrary.java

RobotLibrary.java-tiedosto kehitetään sisältämään kaikki tarvittava avainsanakirjaston ajamiseen palvelimen tyyppisenä palveluna. Aloitetaan päivittämällä tiedoston sisällä olevan luokan nimeksi *RobotLibrary* ja periytetään se *jrobotremoteserver*-kirjaston sisältämästä *AnnotationLibrary*-luokasta käyttämällä Javan *extends*-kutsua. Tällöin *RobotLibrary*-luokkaa voidaan käyttää sekä itse palvelimen ajoon, että avainsanaluokkien hallintaan ja tarjoamiseen testitilanteissa. Lisätään myös *import*-kutsut koodissa käytettäviä *AnnotationLibrary-, RemoteServer-, List- ja Arraylist*-luokkia varten. *RemoteServer* ja *AnnotationLibrary* ovat peräisin *jrobotremoteserver*-kirjastosta, *List* ja *Arraylist* puolestaan kuuluvat Javan standardikirjastoon. Edellä mainitut muutokset ja luokan *main*-funktion esittely on toteutettu koodissa 11. Koodista on myös poistettu Gradlen tyhjät esimerkkifuntkiot.

```
import org.robotframework.javalib.library.AnnotationLibrary;
import org.robotframework.remoteserver.RemoteServer;
import java.util.List;
import java.util.ArrayList;

public class RobotLibrary extends AnnotationLibrary {

    public static void main(String[] args)
    {
        //Do nothing
    }

}
```

KOODI 11. *RobotLibrary*-luokan alustus ja *import*-kutsut.

Seuraavaksi toteutetaan *main*-funktion sisään palvelimen luonti ja käynnistys. *RemoteServer*-luokasta luodaan uusi instanssi, joka asetetaan tarjoamaan *Keywords.java*-tiedoston sisälle toteutettavat avainsanat porttiin 8270. *configureLogging*-kutsu varmistaa, että palvelin kirjoittaa terminaaliin lokia ajon aikaisista tapahtumista. *main*-luokan esittelyriville pitää lisäksi lisätä *throws Exception* -annotaatio, sillä *RemoteServer*-luokka käyttää *Exception*-tyyppisiä ilmoituksia ongelmatilanteissa, kuten jos avainsanaluokkaa ei löydy, tai jos samaa porttia yritetään kuunnella monella eri instanssilla. *main*-funktion sisältö on esitelty koodissa 12.

```
public static void main(String[] args) throws Exception
{
    RemoteServer server = new RemoteServer();
    server.putLibrary("/", new RobotLibrary());
    server.configureLogging();
    server.setPort(8270);
    server.start();
}
```

KOODI 12. *RobotLibrary*-luokan *main*-funktio

Koska *RobotLibrary*-luokka periytettiin *AnnotationLibrary*-luokasta, voidaan *RemoteServer*-instanssille antaa *putLibrary*-kutsulla tarjottavaksi kirjastoksi *RobotLibrary*-luokka itse. Luokkaan pitää kuitenkin määritellä erikseen tiedostot, joiden sisälle avainsanojen toteutukset on luotu. Koodissa 13 on toteutettu kaikkien samassa kansiossa olevien luokkatiedostojen lisääminen *RobotLibraryn parent*-luokan, eli *AnnotationLibrary*-luokan tietoihin.

```
protected static final List<String> keywords = getKeywords();

public static List<String> getKeywords()
{
    List<String> classes = new ArrayList<>();
    classes.add("*.class");
    return classes;
}

public RobotLibrary() { super(keywords); }
```

KOODI 13. luokkatiedostojen lisääminen *RobotLibraryn* avainsanatarjontaan.

Koodissa 13 aloitetaan luomalla *keywords*-niminen listamuuttuja, johon *getKeywords*-funktiolla lisätään arvo **.class*. Sitten *RobotLibrary*-luokan konstruktorissa kyseinen lista lähetetään *parent*-luokalle, eli *AnnotationLibrary*-luokalle käytettäväksi *super*-kutsun avulla. Tällöin *main*-funktion *putLibrary* kutsu johtaa siihen, että *RemoteServer*-instanssi pystyy uuden *RobotLibrary*-instanssin kautta tarjoamaan kaikki samassa kansiossa toteutetut Robot Framework -avainsanat testiajoille.

Nyt projektiin on toteutettu kaikki tarvittava avainsanoja tarjoavan palvelimen ajamista varten ja *RobotLibrary.java*-tiedosto on esitelty kokonaisuudessaan liitteessä 4. Seuraavaksi toteutetaan Robot Framework -testeissä tarvittavat avainsanat *Keywords.java*-tiedostoon.

5.2.4 Keywords.java

Keywords.java-tiedostoon toteutetaan kaikki kappaleessa 4.4.1 esitetyn *main.robot* testitiedoston sisältämien avainsanojen toiminnallisuudet. Kun Robot Framework -testi kutsuu testitiedostoon määritellystä etäkirjastosta avainsanaa, kutsu ohjataan *RobotLibrary*-luokan kautta tämän Java-luokan toteutettavaksi. Kooditiedoston kehitys aloitetaan luomalla sen sisään tyhjä *Keywords*-niminen luokka ja lisäämällä tarvittavat *import*-kutsut luokassa käytettäville Robot Framework -annotaatioille. *Keywords.java*-tiedoston pohja on esitelty koodissa 14.

```
import org.robotframework.javalib.annotation.RobotKeyword;
import org.robotframework.javalib.annotation.RobotKeywords;
import org.robotframework.javalib.annotation.ArgumentNames;

@RobotKeywords
public class Keywords {

}
```

KOODI 14. *Keywords.java*-tiedoston pohja

Robot Framework -avainsanoja sisältävät luokat pitää aina ilmoittaa *@RobotKeywords*-annotaatiolla, jotta *RemoteServer*-luokka tietää mitkä luokat ovat tarkoitettuja tarjottavaksi testiajoja varten. Esimerkiksi avainsanafunktioiden mahdollisesti tarvitsemia apuluokkia ei merkata tällä annotaatiolla, sillä niitä ei ole tarkoitettu kutsuttavaksi suoraan Robot Framework -testeistä. *@RobotKeyword*-annotaatio puolestaan on varattu ilmoitettavaksi tämän luokan sisällä olevien avainsanoiksi tarkoitettujen funktioiden toteutuksen edessä. Vain sillä merkatut avainsanat ovat käytettävissä Robot Framework -testiajoissa. Mikäli avainsana ottaa vastaan parametrejä, pitää se merkata myös parametrien nimet sisältävällä *@ArgumentNames*-annotaatiolla. Soveltuvuusselvityksen Robot Framework -testin perusteella tiedostoon tulee toteuttaa seuraavat avainsanat: S*ftp Create File, Verify File Exists, Download File, Process File* ja *Upload File.*

Sftp Create File -avainsana luo SFTP-palvelimelle Java-apukirjaston sisältämän *SFTPClientUtil*-luokan avulla tiedoston avainsanakutsun parametreissä määritellyllä tavalla. *Verify File Exists* -avainsana käyttää samaa apuluokkaa palvelimen tiedostojen listaamiseen ja täten tietyn tiedoston olemassaolon varmistamiseen. *Download File, Process File* ja *Upload File* -avainsanat puolestaan käyttävät Java-apukirjaston *TriggerUtil*-luokkaa kutsuakseen testattavan ohjelmiston toimintoja.

Aloitetaan avainsanojen toteutus lisäämällä *Keywords*-luokkaan koodin 15 mukaisesti ensimmäinen avainsanatoteutus: *Sftp Create File*. Avainsana ohjaa sille parametrina annetut tiedot *SFTPClientUtil*-apuluokan *putFile*-funktiolle, joka ottaa yhteyden SFTP-palvelimelle ja siirtää *resource*-parametrin mukaisen tiedoston palvelimella sijaitsevaan kansioon. Funktioon on lisätty myös pakolliset *@RobotKeyword*- ja *@ArgumentNames*-annotaatiot. Kyseessä on aktiivinen avainsana, joka suorittaa toiminnon, mutta ei palauta

mitään tietoja testiajolle. Jos *putFile*-funktion suorituksessa tapahtuu virheitä, testiajo merkitään automaattisesti epäonnistuneeksi.

```
@RobotKeyword
@ArgumentNames({"credentials", "resource", "folder", "server"})
public void sftpCreateFile (Map<String, String> credentials, String
resource, String folder, String server) {

    SFTPClientUtil.putFile(credentials, resource, folder, server);

}
```

KOODI 15. *Keywords*-luokan *Sftp Create File* -avainsanan toteutus.

Seuraavaksi toteutetaan luokkaan *Verify File Exists* -avainsana. Se on tyypiltään varmistava avainsana, joka tarkastaa palvelimen tietystä kansiosta parametrissa annetun tiedostonimen olemassaolon. Mikäli, tiedosto löytyy, avainsanan suoritus loppuu ja testi jatkaa kulkuaan, mutta jos tiedostoa ei löydy, avainsanasta tehdään uusi *Exception*-tyyppinen kutsu, jonka seurauksena testiajo epäonnistuu. Koodissa 16 on esitelty *Verify File Exists* -avainsana kokonaisuudessaan.

```
@RobotKeyword
@ArgumentNames({"credentials", "fileName", "folder", "server"})
public void verifyFileExists (Map<String, String> credentials, String
fileName, String folder, String server) {

List<String> files;
    files = SFTPClientUtil.getFileList(credentials, folder, server);

    if (!files.contains(fileName))
        throw new Exception ("File not found!");

}
```

KOODI 16. *Verify File Exists* -avainsanan toteutus.

SFTP-palvelimen hallintaan liittyvien avainsanojen lisäksi toteutetaan vielä ohjelmiston manipulointia varten *Download File, Process File* ja *Upload File* -avainsanat. Kaikkien kolmen avainsanan toiminnallisuus on periaatteessa sama: kutsutaan Java-apukirjaston *TriggerUtil*-luokkaa aktivoimaan testattavasta ohjelmistosta tietyn niminen toiminto. Kyseisten avainsanojen toteutukset on esitelty koodissa 17.

```
@RobotKeyword
public void downloadFile() {
    TriggerUtil.triggerProfile("DOWNLOAD");
}

@RobotKeyword
public void processFile() {
    TriggerUtil.triggerProfile("PROCESS");
}

@RobotKeyword
public void uploadFile() {
    TriggerUtil.triggerProfile("UPLOAD");
}
```

KOODI 17. *Download File, Process File* ja *Upload File* -avainsanojen toteutukset.

Lopuksi lisätään kooditiedostoon vielä koodin 18 mukaisesti *import*-kutsut käytettyjä kirjastoja varten. Näitä ovat Javan standardikirjastosta löytyvä *List*- ja *Map*-luokat sekä yrityksen Java-apukirjastosta löytyvät *SFTPClientUtil*- ja *TriggerUtil*-luokat. *Keywords*.java-tiedosto on luettavissa kokonaisuudessaan tämän dokumentin liitteessä 5.

```
import java.util.List;
import java.util.Map;
import com.liaison.util.SFTPClientUtil;
import com.liaison.util.TriggerUtil;
```

KOODI 18. *Keywords.java*-tiedoston loput *import*-kutsut.

5.3 Avainsanakirjaston buildaaminen

Edellisessä kappaleessa luotiin kaikki avainsanakirjastossa tarvittavat kooditiedostot ja ohjelmisto on nyt build-vaihetta ja varsinaista ajoa vaille valmis. Navigoidaan *RobotLibrary*-projektin juurikansioon ja suoritetaan komentoriviltä koodin 19 mukainen Gradle-komento.

```
gradle build run
```

KOODI 19. Projektin buildaus ja ajaminen Gradle-työkalulla.

Kyseinen komento käskee Gradlea suorittamaan *build*- ja *run*-nimiset taskit. *build*-task aloitetaan luomalla juurikansioon uuden *build*-nimisen kansion. Sitten se kääntää ensin *libs*-kansiossa sijaitsevan *jrobotremoteserver-3.0-standalone.jar*-tiedoston sisältämän koodin projektin käytettäväksi, jonka jälkeen se hakee myös yrityksen omasta Maven-repositoriosta tarvittavan Java-apukirjaston sisällytettäväksi projektiin. Lopuksi se kääntää itse projektin omat Java-tiedostot ajettavaan muotoon samaan *build*-kansioon.

build-taskin jälkeen Gradle suorittaa seuraavaksi *run*-vaiheen, jossa se ajaa *build*-kansiosta *mainClassName*-muuttujaan asetetun Java-luokan, eli *RobotLibrary*-luokan *main*-funktion. Kyseinen funktio käynnistää ohjelman sisälle palvelimen, joka tarjoaa *Keywords.java*-tiedostossa määritellyt avainsanat käytettäväksi siihen yhteyttä ottaville testiajoille. Seuraavassa kappaleessa avainsanakirjasto käynnistetään varsinaista Robot Framework -testiajoa varten.

6 ROBOT FRAMEWORK -TESTIN AJAMINEN

Soveltuvuusselvitystä varten on edellisten kappaleiden aikana toteutettu Robot Framework -testikokonaisuus sisältäen sekä testitiedostot että niiden avainsanakutsuja varten toteutetun avainsanakirjaston. Seuraavana vaiheena on ajaa testit läpi paikallisesti ja varmistaa, että kaikki toimii kuten pitää. Aloitetaan käynnistämällä avainsanakirjasto odottamaan testiyhteyttä, jonka jälkeen suoritetaan itse Robot Framework -testiajo. Tarkemmat tiedot Robot Framework -testien ajamisen vaatimuksista on luettavissa Robot Frameworkin käyttöoppaasta (Robot Framework, 2018, kappale 1.3.2).

6.1 Avainsanakirjaston käynnistys

Avainsanakirjasto käynnistetään ajamalla *C:\Tiedostot\RobotLibrary*-kansion juuressa komentorivillä kirjaston buildaava ja suorittava *gradle build run* -komento. Gradle buildaa projektin kappaleessa 5.3 esitellyllä tavalla, jonka jälkeen se käynnistää *run*-komennolla kirjaston odottamaan Robot Framework -testiyhteyttä. Komentorivin lokista voidaan koodin 20 mukaisesti todeta kirjaston käynnistyminen ja portin 8270 kuuntelu.

```
gradle build run

INFO: org.robotframework.remoteserver.RemoteServer - Robot Framework
remote server starting
INFO: oejs.Server:jetty-7.x.y-SNAPSHOT
INFO: oejsh.ContextHandler:started o.e.j.s.ServletContextHandler{/,null}
INFO: oejs.AbstractConnector:Started SelectChannelConnector@0.0.0.0:8270

INFO: org.robotframework.remoteserver.RemoteServer  - Robot Framework
remote server started on port 8270.
```

KOODI 20: Avainsanakirjaston suoritusloki terminaalissa.

6.2 Testien ajaminen

Kun avainsanakirjasto on käynnistetty, voidaan siirtyä suorittamaan itse testiajoa. Testin ajamiseen tarvitaan Python-työkalu ja sen päälle asennettava Robot Framework -liitännäinen. Python asennetaan osoitteesta http://python.org/, jonka jälkeen Pythonin pip-työkalua käyttäen voidaan koodin 21 mukaisesti asentaa sen Robot Framework -liitännäinen.

```
pip install robotframework
```

KOODI 21. Robot Framework -liitännäisen asentaminen Python-työkalun päälle.

Navigoidaan testitiedostot sisältävän *C:\Tiedostot\RobotTest*-kansion juureen ja suoritetaan Robot Framework -testi ajamalla komentorivillä koodin 22 mukainen *robot*-käsky.

```
robot main.robot
```

KOODI 22. Robot Framework -testin suoritus komentoriviltä

robot on Robot Framework -testien ajamiseen tarkoitettu käsky, joka saa parametrina ajettavan testitiedoston tai -kansion nimen sekä liudan vaihtoehtoisia konfigurointiparametreja, joita tässä tapauksessa ei toisaalta tarvita. Robot Framework suorittaa kaikki testitiedostossa olevat vaiheet näyttäen samalla terminaalissa tietoa testin kulusta sekä onnistuneista ja epäonnistuneista testivaiheista kuten koodin 23 terminaalilokista käy ilmi.

```
==============================================================================
Main
==============================================================================
Create file into SFTP server's input folder                   | PASS |
------------------------------------------------------------------------------
Verify file exists in SFTP server's input folder              | PASS |
------------------------------------------------------------------------------
Move file from the input folder to the process folder         | PASS |
------------------------------------------------------------------------------
Verify file exists in SFTP server's process folder            | PASS |
------------------------------------------------------------------------------
Process the file and move it to the processed folder          | PASS |
------------------------------------------------------------------------------
Verify file exists in the processed folder                    | PASS |
------------------------------------------------------------------------------
Move the file to the output folder                            | PASS |
------------------------------------------------------------------------------
Verify file exists in the output folder                       | PASS |
------------------------------------------------------------------------------
Main                                                          | PASS |
8 critical tests, 8 passed, 0 failed
8 tests total, 8 passed, 0 failed
==============================================================================
Output:  C:\Tiedostot\RobotTest\output.xml
Log:     C:\Tiedostot\RobotTest\log.html
Report:  C:\Tiedostot\RobotTest\report.html
```

KOODI 23. Robot Framework -testiajon terminaaliloki.

6.3 Testiajon lopputulos

Robot Framework -soveltuvuusselvityksen testissä suoritettiin kahdeksan eri vaihetta ja tässä ajossa ne kaikki läpäistiin onnistuneesti. Koodin 23 lokin lopussa Robot Framework ilmoittaa testin aikana luoduista raporttitiedostoista, joista voidaan lukea muun muassa epäonnistuneiden testien tapauksessa virheeseen johtanut avainsanakutsu lisätietoineen. Testitiedostoissa voidaan myös kutsua testivaiheiden sisällä lokiin informaatiota kirjoittavia avainsanoja esimerkiksi debugatessa toimimattomia testejä.

Testin kulku oli identtinen yrityksessä aiemmin ajettuihin TestNG-tyyppisiin testeihin verrattuna. Robot Framework -testin avainsanakutsut käyttivät samaa Java-apukirjastoa, jota Javaan perustuvissa TestNG-testeissä kutsutaan. Soveltuvuusselvitys siitä, voidaanko vanhat testit kääntää ajettavaksi Robot Framework -työkalulla, on siis onnistunut. Seuraavassa vaiheessa työssä luotu Robot Framework -testi pitää vielä toteuttaa ajettavaksi Kubernetes-ympäristössä, jotta koko soveltuvuusselvitys voitaisiin julistaa menestykseksi.

7 KUBERNETES-TOTEUTUS

Aiemmissa kappaleissa suoritettiin yrityksen asettaman Robot Framework -soveltuvuustestin toteutus testitiedostoineen ja avainsanakirjastoineen. Lopuksi testiajo ajettiin onnistuneesti paikallisesti, joten seuraava vaihe on saattaa koko testiajo suoritettavaksi yrityksen Kubernetes-ympäristössä. Yritys on ennenkin toteuttanut palveluita Kubernetes-palvelimelleen, joten tämänkin projektin sinne saattamiselle on asetettu selvät vaiheet:

1. Toteutetaan Robot Framework -testin ja avainsanakirjaston säiliöinti Docker-ohjelmistolla. Kun säiliö suoritetaan, testi ajetaan automaattisesti avainsanakirjastoa käyttäen.
2. Julkaistaan luotu Docker-image yrityksen Docker-rekisteriin.
3. Toteutetaan Docker-imagen Kubernetes-ajoa varten Kubernetes-määrittelytiedosto.
4. Ajetaan Kubernetes-tiedosto sisään yrityksen Kubernetes-palvelimelle ja todetaan testiajon suoritus ja onnistuminen.

7.1 Docker-imagen luominen

Docker-työkalu käyttää säiliöiden määrittelyyn *Dockerfile*-nimistä tiedostoa. Siinä määritellään mitä ympäristöä säiliön sisällä halutaan käytettävän (esim. valittu Linux-distribuutio) sekä sen päälle suoritettavat komennot ja tiedostosiirrot. Lisäksi tiedoston loppuun lisätään *CMD*-komento, joka määrittelee säiliöltä suoritettaessa ajettavan komennon. Kun tiedosto on valmis, Dockerilla buildataan sen pohjalta imagetiedosto, jossa kaikki määritellyt toiminnot on jo suoritettu. Build-vaiheessa voidaan siis toteuttaa ympäristön sisälle kaikkien tarvittavien työkalujen asennus sekä avainsanakirjaston oma build-vaihe. Tällöin imagea ajettaessa voidaan suoraan käynnistää avainsanakirjasto ja suorittaa Robot Framework -testi edellä mainitun *CMD*-komennon avulla.

7.1.1 Dockerfile-tiedoston luonti

Aloitetaan luomalla *C:\Tiedostot*-kansioon uusi tiedosto nimeltä *Dockerfile.* Siirrytään muokkaamaan tiedostoa ja aloitetaan määrittelemällä koodin 24 mukaisesti, että säiliö halutaan rakentaa *Alpine*-Linux-distribuution päälle, jossa on myös esiasennettuna Gradlen versio 4.10.2. Tämän lisäksi otetaan säiliössä käyttöön *root*-käyttäjä, jolla on oikeudet asentaa paketteja ja suorittaa sovelluksia

```
FROM gradle:4.10.2-alpine
USER root
```

KOODI 24. Docker-säiliön *base-imagen* konfigurointi ja käyttäjän valinta.

Seuraavaksi lisätään *Dockerfileen* käskyt kopioida *RobotLibrary*- ja *RobotTest*-kansiot säiliön sisälle koodin 25 mukaisesti. Molemmat kansiot siirretään säiliön sisällä samalla käskyllä luotavaan */usr/Robot/*-kansioon.

```
COPY RobotTest /usr/Robot/RobotTest/
COPY RobotLibrary /usr/Robot/RobotLibrary/
```

KOODI 25. Robot Framework -testin ja avainsanakirjaston siirto säiliön sisälle.

Kun tiedostot on siirretty, voidaan koodin 26 käskyillä navigoida avainsanakirjaston kansioon ja buildata se Gradle-työkalulla.

```
WORKDIR /usr/Robot/RobotLibrary/
RUN gradle clean build
```

KOODI 26. Avainsanakirjaston buildaus säiliön sisällä.

Sitten asennetaan koodin 27 mukaisesti säiliön sisälle Python-työkalu ja sen Robot Framework -liitännäinen. Käskyn *no-cache*-parametrit on lisätty pitämään Docker-imagen koko mahdollisimman pienenä. Kaikki *Dockerfilen* aikana säiliön sisälle luodut tiedostot kasvattavat sen kokoa, eikä tässä tapauksessa välimuistin säilyttämiselle ole tarvetta.

```
RUN set -o errexit -o nounset \
        && apk add --no-cache --update python3 \
        && pip3 install --upgrade pip \
        && pip3 install --upgrade --no-cache-dir robotframework
```

KOODI 27. Python-työkalun ja Robot Framework -liitännäisen asennus.

Lopuksi *Dockerfileen* lisätään vielä *CMD*-komento, joka ajetaan, kun tiedoston perusteella luotu imagetiedosto suoritetaan. Koodissa 28 on esitelty *CMD*-käsky kokonaisuudessaan. Käsky on kolmivaiheinen: ensin käynnistetään säiliön sisälle buildattu avainsanakirjasto *gradle run* -komennolla, jonka jälkeen odotetaan 10 sekuntia palvelimen valmistumista. Sitten suoritetaan *robot*-käskyllä Robot Framework -testin ajo. *CMD*-käskyssä annetut */dev/null*-tiedot varmistavat sen, että käynnistetty avainsanakirjasto ei tulosta lokiaan testiajon päälle. *Dockerfile*-tiedosto on tätä komentoa myöten valmis buildattavaksi ja se on esitelty kokonaisuudessaan liitteessä 6.

```
CMD ["sh", "-c", "(gradle run </dev/null &>/dev/null &) && sleep 10s &&
robot ../RobotTest"]
```

KOODI 28. *Dockerfilen CMD*-käsky.

7.1.2 Docker-imagen buildaus

Kun *Dockerfile* on saatu valmiiksi, voidaan Docker-työkalulla buildata sen perusteella Docker-imagetiedosto. Navigoidaan komentorivillä *C:\Tiedostot*\-kansioon ja ajetaan koodin 29 mukainen käsky.

```
docker build -t robot-framework-test .
```

KOODI 29. Docker-imagen buildaaminen Docker-työkalulla.

Dockerin *build*-komento buildaa kansiossa olevan *Dockerfilen* sisällön perusteella uuden Docker-imagen tietokoneelle paikallisesti. Komennon *-t*-parametri määrittää imagen nimeksi *robot-framework-test* ja lopussa oleva piste kertoo Dockerfilen löytyvän komentorivin sen hetkisestä kansiosta. Image on nyt buildattu ja se voidaan lähettää yrityksen Docker-rekisteriin.

7.1.3 Docker-imagen lähetys Docker-rekisteriin

Yrityksen Kubernetes-palvelin luo podeja hakemalla niiden sisällä käytettävien säiliöiden imagetiedoston yrityksen omasta Docker-rekisteristä palvelimelta *docker.liaison.dev*. Myös edellisessä kappaleessa buildattu Docker-image tulee täten lähettää sinne ennen kuin se voidaan ajaa Kuberneteksessä. Koodissa 30 olevilla komennoilla suoritetaan imagetiedoston lähetys palvelimelle.

```
docker login docker.liaison.dev
docker tag robot-framework-test docker.liaison.dev/robot-framework-test
docker push docker.liaison.dev/robot-framework-tests
```

KOODI 30. Docker-imagen lähetys yrityksen Docker-rekisteriin.

7.2 Kubernetes-määrittelytiedoston luonti

Kubernetes-määrittelytiedostossa määritellään Kuberneteksessä ajettavan toteutuksen tiedoista muun muassa toteutuksen tyyppi, käytettävissä olevat resurssit, säiliöiden imaget ja käytettävät ympäristömuuttujat. Tiedoston sisällössä käytetään YAML-merkintäkieltä, joka on vahvasti JSON-merkintäkielen tyylinen, joskin hieman selkolukuisempi.

Luodaan uusi Kubernetes-määrittelytiedosto nimellä *K8sfile.yaml* kansioon *C:\Tiedostot*. Alla olevassa koodissa 31 on esitelty tiedostoon lisättävän määrittelyn sisältö kokonaisuudessaan. Se on myös liitetty tähän asiakirjaan liitteenä 7.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: robot-framework-test
spec:
  template:
    spec:
      restartPolicy: Never
      imagePullSecrets:
      - name: liaison-imagepullsecret
      containers:
      - name: robot-framework-test
        image: docker.liaison.dev/robot-framework-test:latest
        imagePullPolicy: Always
```

KOODI 31. Kubernetes-toteutuksen *K8sfile.yaml*-tiedoston sisältö.

Koodissa on määritelty Kubernetes-toteutuksen tyypiksi testiajoihin soveltuva *Job*. Se kertoo Kubernetekselle, että tämä toteutus merkitään onnistuneeksi vasta, kun kaikki sen sisältämät säiliöt, eli testiajot, on suoritettu loppuun. Kontrastina voidaan pitää *Deployment*-tyyppistä toteutusta, jota käytetään muun muassa Internet-sivustojen palvelimissa. Niissä onnistuneeksi tilaksi lasketaan se, kun kaikki valitut säiliöt ovat päällä ja toiminnallisia.

Metadataan on merkitty jobin nimi ja *spec*-osioon tiedot Jobin sisällöstä. *restartPolicy* määrittää missä tapauksissa jobin sisältämät säiliöt halutaan käynnistää uudelleen ja *imagePullSecrets* sisältää kirjautumistiedot yrityksen Docker-rekisteriin imagen hakua varten. *liaison-imagepullsecret* on konfiguroitu yrityksen Kubernetes-palvelimelle vastaamaan tarvittavaa käyttäjätunnusta ja salasanaa. *containers*-osiossa on määritelty jobissa luotavien säiliöiden tiedot, kuten nimi ja käytettävä imagetiedosto. Tässä tapauksessa säiliöitä on vain yksi, nimeltään *robot-framework-test,* jonka alle on määritelty, että kyseisen säiliön image halutaan hakea yrityksen Docker-rekisteristä.

7.2.1 Kubernetes-toteutuksen suorittaminen

Nyt kun testiajon Docker-image löytyy yrityksen Docker-rekisteristä, ja Kubernetes-määrittelytiedosto on toteutettu, voidaan siirtyä suorittamaan testiajo Kubernetes-palvelimen sisällä. Tähän käytetään Kuberneteksen hallintaa varten luotua kubectl-työkalua, joka luo ensin yhteyden palvelimeen, jonka jälkeen sen avulla voidaan ajaa palvelimelle sisään Kubernetes-konfiguraatioita. Tämän vaiheen suorittamisen helpottamiseksi edellisen kappaleen *K8sfile.yaml*-tiedosto on siirretty uuteen *Alpine-Linux*-tyyppiseen Docker-säiliöön, johon kubectl on jo valmiiksi asennettuna.

Aloitetaan asettamalla kubectl-työkalulle tarvittavat yhteystiedot yrityksen Kubernetes-palvelimelle koodin 32 mukaisella komennolla. Se lataa tiedot *config*-tiedostosta ja kaikki seuraavat kubectl-käskyt ohjataan niiden perusteella oikeaan osoitteeseen. Tässä tapauksessa käytetty *config*-tiedosto löytyy kokonaisuudessaan työn liitteestä numero 8.

```
$ kubectl config use-context liaison-dev
```

KOODI 32. kubectl-työkalu ja *config*-tiedoston kontekstin valinta

Kun *config*-tiedostosta on määritelty *liaison-dev* aktiiviseksi kontekstiksi, voidaan seuraavaksi koodin 33 komennolla suorittaa *K8sfile.yaml*-tiedoston sisällä oleva konfiguraatio. Komennon ajamalla kubectl muuttaa *-f*-parametriin asetetun YAML-tiedoston JSON-muotoon ja lähettää sen *config*-tiedostossa määrätylle Kubernetes-palvelimelle suoritettavaksi.

```
$ kubectl create -f K8sfile.yaml
job robot-framework-test” created
```

KOODI 33. kubetcl-työkalun *create*-komento uuden toteutuksen luomiseksi

create-komennon johdosta palvelimelle luodaan uusi *Job*-tyyppinen objekti, joka *K8sfile.yaml*-tiedostoon määritellyn konfiguraation mukaan hakee yrityksen Docker-rekisteristä Robot Framework -testiajoa varten kehitetyn Docker-imagen. Docker-image ajetaan Kubernetes-palvelimella ja sen sisälle määritelty *CMD*-komento käynnistää Robot Framework -avainsanakirjaston ja ajaa testiajon sitä käyttäen.

Seuraavaksi tarkastetaan, miten testiajo onnistui, avaamalla testiajossa käytetyn podin loki. Ensin pitää selvittää podin nimi ja sen jälkeen voidaan avata sen luoma loki. Podin nimi löytyy ajamalla koodin 34 mukainen komento, joka listaa kaikki podit, joiden nimessä esiintyy *robot-framework-test.* Koska podeja on luotu tässä jobissa vain yksi, voidaan varmuudella sanoa, että kyseisen podin sisältämä loki sisältää myös testiajon tiedot.

```
$ kubectl get pods -a | grep robot-framework-test

NAME                          READY     STATUS      RESTARTS      AGE
robot-framework-test-0trsx    0/1       Completed   0             1m
```

KOODI 34. Kubernetes-podien listaaminen nimen perusteella.

Kun podin nimi on selvitetty, voidaan koodissa 35 esitellyllä käskyllä avata podin loki luettavaksi terminaaliin. Käskyn jälkeisestä lokista nähdään, että testit suoritettiin onnistuneesti täsmälleen samalla tavalla kuin kappaleessa 6.2. Ainoana erona tässä ajossa on, että raporttitiedostot löytyvät tällä kertaa Kubernetes-palvelimelta podissa olevan säiliön sisältä.

```
$ kubectl logs robot-framework-test-0trsx
…
…
==============================================================================
Main
==============================================================================
Create file into SFTP server's input folder                           | PASS |
------------------------------------------------------------------------------
Verify file exists in SFTP server's input folder                      | PASS |
------------------------------------------------------------------------------
Move file from the input folder to the process folder                 | PASS |
------------------------------------------------------------------------------
Verify file exists in SFTP server's process folder                    | PASS |
------------------------------------------------------------------------------
Process the file and move it to the processed folder                  | PASS |
------------------------------------------------------------------------------
Verify file exists in the processed folder                            | PASS |
------------------------------------------------------------------------------
Move the file to the output folder                                    | PASS |
------------------------------------------------------------------------------
Verify file exists in the output folder                               | PASS |
------------------------------------------------------------------------------
Main                                                                  | PASS |
8 critical tests, 8 passed, 0 failed
8 tests total, 8 passed, 0 failed
==============================================================================
Output:  /usr/Robot/RobotLibrary/output.xml
Log:     /usr/Robot/RobotLibrary/log.html
Report:  /usr/Robot/RobotLibrary/report.html
```

KOODI 35. Testiajon loki Kubernetes-palvelimella.

Testiajo suoritettiin onnistuneesti läpi Kubernetes-palvelimella, ja tässä vaiheessa voidaan todeta työssä määritellyn soveltuvuusselvityksen onnistuminen kokonaisuudessaan. Kappaleessa 2 esitetty ratkaistava ongelma onnistuttiin selvittämään Robot Frameworkin avulla, ja ajo saatiin siirrettyä aina Kubernetes-palvelimelle asti. Lisäksi testeissä saatiin käytettyä hyväksi yrityksen aikaisempaa Java-apukirjastoa.

8 JATKOKEHITYS

Itse soveltuvuusselvityksen päätavoitteena oli selvittää, onko yrityksen sisällä yleensä ottaen mahdollista siirtyä käyttämään Robot Frameworkia vanhojen apukirjastojen kanssa sekä toteuttaa testien ajo Kubernetes-ympäristössä. Koska soveltuvuusselvitys saatiin vietyä onnistuneesti päätökseensä, on hyvä pohtia myös Robot Framework -työkaluun liittyviä yrityksen sisäisiä jatkokehitysideoita. Soveltuvuusselvityksen kehityksen yhteydessä esille nousi kaksi huomattavaa kehitysideaa Robot Frameworkin käyttöön liittyen: Java-avainsanakirjaston ajaminen pysyvänä *Deployment*-tyyppisenä toteutuksena Kubernetes-ympäristössä sekä testiajojen automatisointi Jenkins-työkalun avulla.

Toistaiseksi avainsanakirjasto käynnistetään aina testiajon mukana saman Kubernetes-podin sisällä. Testien ajo on siis huomattavasti hitaampaa kuin jos itse kirjasto olisi koko ajan saatavilla ja Kubernetes-palvelimella käynnistettäisiin vain itse testitiedostojen ajo. Lisäksi sillä saavutettaisiin se hyöty, että uusia testitapauksia kehitettäessä kirjastoa ei tarvitsisi ajaa paikallisesti ollenkaan, vaan testeihin voitaisiin suoraan sisällyttää yhteys aina päällä olevan avainsanakirjaston Kubernetes-toteutukseen.

Testiajojen automatisointi on myös tärkeänä prioriteettina kaikissa hyvään DevOps-toimintamalliin pyrkivissä projekteissa. Yhtenä seuraavista kehitysaskeleista projektille on siis varmasti luoda Jenkins-työkalun avulla testiajojen automatisointi. Yrityksen sisällä onkin jo automatisoitu vanhojen Java-pohjaisten integraatiotestien ajo, joten siirtymä Robot Framework -testien automatisointiin on helposti toteutettavissa sen pohjalta. Molempien edellä mainittujen kehitysideoiden toteuttamisen suunnittelu aloitettiin välittömästi soveltuvuusselvityksen valmistuttua ja ne tullaan ottamaan käyttöön lähitulevaisuudessa.

9 POHDINTA

Tämän opinnäytetyön toteuttamiseen johti Liaison Technologies Oy -yrityksen tarve mahdollistaa uusien ohjelmistotestien toteutus myös vähemmän teknisen henkilökunnan toimesta. Tarvittiin siis työkalu, joka tarjoaisi perinteisten ohjelmointikielien päälle tulevan abstraktiokerroksen, tehden siten testien luomisesta perinteistä koodaamista selkokielisempää. Ratkaisuksi valittiin Robot Framework -työkalu. Opinnäytetyön tavoitteena oli toteuttaa yritykselle soveltuvuusselvitys työkalun käyttämisestä yrityksen sisäisten ohjelmistojen ja ohjelmistokomponenttien testaamisessa. Selvityksen tuli sisältää itse testin ja tarvittavien kirjastojen suunnittelu, toteutus ja lopulta ajo Kubernetes-ympäristössä. Testitoteutuksen lisäksi oli myös tärkeää pohtia missä osa-alueissa uusi testaustapa on aikaisempaa TestNG-tyyppistä testaamista parempi, tai huonompi vaihtoehto.

Jo Robot Frameworkin dokumentaatiosta käy ilmi, että työkalu tukee Java-ohjelmointikielellä toteutettuja kirjastoja. Koska kaikki yrityksen olemassa olevat työkalut on toteutettu Javalla, oli jo alusta asti selvää, että kaikki tarvittava toiminnallisuus tullaan saavuttamaan myös Robot Frameworkilla näitä kirjastoja hyväksi käyttäen. Työ aloitettiin kehittämällä jo olemassa olevasta SFTP-palvelimen manipulaatioon liittyvästä TestNG-integraatiotestistä Robot Framework -syntaksia käyttäen identtisesti toimiva testi. Siinä Java-ohjelmointikielellä toteutetut funktiot peitettiin Robot Frameworkin avulla avainsanakutsujen taakse abstraktioiduiksi testivaiheiksi. Sen jälkeen avainsanoille toteutettiin yrityksessä valmiiksi olevia Java-apukirjastoja hyväksi käyttävät avainsanafunktiot erilliseen kirjastopalvelimeen. Testiajo suoritettiin ensin paikallisesti ja siirrettiin sen jälkeen onnistuneesti Kubernetes-palvelimelle ajettavaksi.

Työn tekovaiheessa Robot Frameworkin hyödyt ja haitat tulivat selvästi ilmi vaihe kerrallaan. Heti alkuun Robot Frameworkin syntaksi osoitti potentiaalinsa testien kehityksen helpottamiseksi, sillä se on erittäin selkokielistä. Kaikille avainsanoille pitää kuitenkin edelleen toteuttaa Java-puolella toiminnallisuudet, mikä johtaa siihen, että vaikka uudet testit voitaisiinkin toteuttaa vähemmän teknisen henkilökunnan toimesta, itse avainsanatoiminnallisuuksien toteutukseen vaaditaan edelleen ohjelmistokehityksen ammattilainen. Jos uudet avainsanat kuitenkin toteutetaan aina mahdollisimman uudelleen käytettäviksi ja kustomoitaviksi, tarve niiden luonnille tulee vähenemään ajan kanssa.

Toinen selvä etu Robot Framework -testeissä Java-pohjaisiin testeihin verrattuna on se, että itse testitiedostot voidaan pitää täysin erillään niitä suorittavista avainsanoista etäkirjastopalvelimen avulla. Testitiedostot ovat puhtaita tekstitiedostoja, joita ei tarvitse erikseen kääntää, kun taas vanhat TestNG-testitiedostot olivat Java-luokkia, jotka käännettiin paikallisesti tarvittavien apukirjastojen kanssa ajettaviksi. Tästä johtuen uusia testejä voidaan kirjoittaa ja ajaa asentamatta Java-kääntäjää, tai muita raskaita ohjelmistotyökaluja testejä suorittavalle koneelle. Soveltuvuusselvityksen myötä huomattiin, että uusien testien luominen ja ajaminen on huomattavasti nopeampaa ja kevyempää Robot Frameworkilla, kuin kokonaisen Java-projektin jatkuva kääntäminen ajettavaksi pienimpiäkin testimuutoksia varten.

Testitapauksien kirjoittamisen helpottuminen ja testien ajoon suoritettavan ajan väheneminen ovat kaksi soveltuvuusselvityksen avulla saavutettua objektiivista tavoitetta, joiden perusteella voidaan tehdä se johtopäätös, että Robot Frameworkia tullaan käyttämään yrityksen sisällä tulevaisuudessa kasvavissa määrin. Yrityksessä on pitkä tausta Javaan pohjautuvien testien luomisesta, joten siirtymä Robot Frameworkin käyttöön suuressa mittakaavassa tulee varmasti kestämään jonkin aikaa. Moni eri osasto yrityksen sisällä on osoittanut selvää kiinnostusta tätä testitapaa kohtaan, joten jos Robot Frameworkin teknologiassa tai suorituskyvyssä ei tule vastaan jotain ylitsepääsemätöntä ongelmaa, uskon vakaasti, että tulevaisuudessa Robot Framework tulee nostamaan yrityksen sisäisen testien luontikapasiteetin ja suorituskyvyn uudelle, selvästi korkeammalle tasolle.

LÄHTEET

McKinsey & Company. 2015. Beyond agile: Reorganizing IT for faster software delivery. Luettu: 22.12.2018. https://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/beyond-agile-reorganizing-it-for-faster-software-delivery

Laukkanen, P. 2006. Data-Driven and Keyword-Driven Test Automation Frameworks. Luettu 23.12.2018. http://eliga.fi/Thesis-Pekka-Laukkanen.pdf

Robot Framework. 2018a. Robot Framework User Guide. Kappale 4.1.1. Supported programming languages. Luettu 24.12.2018. http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-programming-languages

Robot Framework. 2018b. Robot Framework User Guide. Luettu 28.12.2018. http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html

Oracle, 2013. Java 2013 Review: Java Takes on the Internet of Things. Luettu 25.12.2018. https://www.oracle.com/technetwork/articles/java/afterglow2013-2030343.html

Poole, S. 2018. The Anatomy of Java Vulnerabilities. Luettu 26.12.2018. https://www.slideshare.net/StevePoole/the-anatomy-of-java-vulnerabilities

Gradle.org. 2015. Introducing Incremental Build support. Luettu 27.12.2018. https://blog.gradle.org/introducing-incremental-build-support

IDC Corporate USA. 2018. Smartphone Market Share. Luettu 28.12.2018. https://www.idc.com/promo/smartphone-market-share/os

Container Journal. 2016. Docker: Not Faster than VMs, but More Efficient. Luettu 18.12.2018. https://containerjournal.com/2016/11/21/docker-not-faster-vms-just-efficient/

Docker. 2018. About images, containers and storage drivers. Luettu 19.12.2018. https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers/#images-and-layers

Ruby Garage. 2018. Advantages of Using Docker for Microservices. Luettu 26.12.2018. https://rubygarage.org/blog/advantages-of-using-docker-for-microservices

Red Hat. 2018. What is Kubernetes? Luettu 27.12.2018. https://www.redhat.com/en/topics/containers/what-is-kubernetes

GitHub, 2018. jrobotremoteserver. Luettu 25.12.2018. https://github.com/ombre42/jrobotremoteserver

LIITTEET

Liite 1. Robot Framework -testin *main.robot*-tiedosto.

```
*** Settings ***
Library    Remote              http://localhost:8270/
Resource   resources.robot

*** Test Cases ***
Create file into SFTP server's input folder
  Sftp Create File  ${CREDENTIALS}  ${RESOURCE}  ${INPUT_FOLDER}
${SERVER}

Verify file exists in SFTP server’s input folder
  Wait Until Keyword Succeeds  3min  5s  Verify File Exists
${CREDENTIALS}  ${FILENAME}  ${INPUT_FOLDER}  ${SERVER}

Move file from the input folder to the process folder
  Download File

Verify file exists in SFTP server’s process folder
  Wait Until Keyword Succeeds  3min  5s  Verify File Exists
${CREDENTIALS}  ${FILENAME}  ${PROCESS_FOLDER}  ${SERVER}

Process the file and move it to the processed folder
  Process File

Verify file exists in the processed folder
  Wait Until Keyword Succeeds  3min  5s  Verify File Exists
${CREDENTIALS}  ${FILENAME}  ${PROCESSED_FOLDER}  ${SERVER}

Move the file to the output folder
  Upload File

Verify file exists in the output folder
  Wait Until Keyword Succeeds  3min  5s  Verify File Exists
${CREDENTIALS}  ${FILENAME}  ${OUTPUT_FOLDER}  ${SERVER}
```

Liite 2. Robot Framework -testin *resources.robot*-tiedosto.

```
*** Variables ***
&{CREDENTIALS}
...                 username=test_user
...                 password=p455word!9

${RESOURCE}         common/test_file.txt
${FILENAME}         test_file.txt

${SERVER}           sftp://liaison.dev:22
${INPUT_FOLDER}     INPUT
${PROCESS_FOLDER}   PROCESS
${PROCESSED_FOLDER}  PROCESSED
${OUTPUT_FOLDER}    OUTPUT
```

Liite 3. Robot Framework -avainsanakirjaston *build.gradle*-tiedosto.

```
plugins {
    id 'java-library'
    id 'application'
}

mainClassName = 'RobotLibrary'

repositories {
    maven { url 'http://nexus.liaison.dev/repositories/' }
}

dependencies {
    compile files ('libs/jrobotremoteserver-3.0-standalone.jar')
    implementation "com.liaison:e2e-test-utils:4.+"
}
```

Liite 4. Robot Framework -avainsanakirjaston *RobotLibrary.java*-tiedosto

```
import org.robotframework.javalib.library.AnnotationLibrary;
import org.robotframework.remoteserver.RemoteServer;
import java.util.List;
import java.util.ArrayList;

public class RobotLibrary extends AnnotationLibrary {

    public static void main(String[] args) throws Exception
    {
        RemoteServer server = new RemoteServer();
        server.putLibrary("/", new RobotLibrary());
        server.configureLogging();
        server.setPort(8270);
        server.start();
    }

    protected static final List<String> keywords = getKeywords();

    public static List<String> getKeywords()
    {
        List<String> classes = new ArrayList<>();
        classes.add("*.class");
        return classes;
    }

    public RobotLibrary() { super(keywords); }

}
```

Liite 5. Robot Framework -avainsanakirjaston *Keywords.java*-tiedosto.

```
import org.robotframework.javalib.annotation.RobotKeyword;
import org.robotframework.javalib.annotation.RobotKeywords;
import org.robotframework.javalib.annotation.ArgumentNames;

import java.util.List;
import java.util.Map;
import com.liaison.util.SFTPClientUtil;
import com.liaison.util.TriggerUtil;

@RobotKeywords
public class Keywords {

@RobotKeyword
@ArgumentNames({"credentials", "resource", "folder", "server"})
public void sftpCreateFile (Map<String, String> credentials, String
resource, String folder, String server) {

    SFTPClientUtil.putFile(credentials, resource, folder, server);

}

@RobotKeyword
@ArgumentNames({"credentials", "fileName", "folder", "server"})
public void verifyFileExists (Map<String, String> credentials, String
fileName, String folder, String server) {

List<String> files;
    files = SFTPClientUtil.getFileList(credentials, folder, server);

    if (!files.contains(fileName))
        throw new Exception ("File not found!");

}

@RobotKeyword
public void downloadFile() {
    TriggerUtil.triggerProfile("DOWNLOAD");
}

@RobotKeyword
public void processFile() {
    TriggerUtil.triggerProfile("PROCESS");
}

@RobotKeyword
public void uploadFile() {
    TriggerUtil.triggerProfile("UPLOAD");
}

}
```

Liite 6. Robot Framework -testien ja avainsanakirjaston Docker-toteutus

```
FROM gradle:4.10.2-alpine

COPY RobotTest /usr/Robot/RobotTest/
COPY RobotLibrary /usr/Robot/RobotLibrary/

WORKDIR /usr/Robot/RobotLibrary/
RUN gradle clean build

RUN set -o errexit -o nounset \
        && apk add --no-cache --update python3 \
        && pip3 install --upgrade pip \
        && pip3 install --upgrade --no-cache-dir robotframework

CMD ["sh", "-c", "(gradle run </dev/null &>/dev/null &) && sleep 10s &&
robot ../RobotTest"]
```

Liite 7. Kubernetes-toteutuksen *K8sfile.yaml*-tiedosto.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: robot-framework-test
spec:
  template:
    spec:
      restartPolicy: Never
      imagePullSecrets:
      - name: liaison-imagepullsecret
      containers:
      - name: robot-framework-test
        image: docker.liaison.dev/robot-framework-test:latest
        imagePullPolicy: Always
```

Liite 8. kubectl-työkalun *config*-tiedosto.

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority: CFSSL_LIAISON.COM.pem
    server: https://kubernetes.liaison.dev:6443
  name: dev
contexts:
- context:
    cluster: dev
    user: liaison
  name: liaison-dev
current-context: liaison-dev
kind: Config
preferences: {}
users:
- name: liaison
  user:
    token: ATJwWTj2LmCzMmA5NThzNjDlAzuXZMfJ
```