Joel Känsälä

Testausautomaation kehittäminen Robot Framework-kehyksellä

Insinööri (AMK)

Tieto- ja viestintätekniikka

Syksy 2021

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä: Känsälä Joel

Työn nimi: Testausautomaation kehittäminen Robot Frameworkilla

Tutkintonimike: Insinööri (AMK), Tieto- ja viestintätekniikka

Asiasanat: DevOps, Robot Framework, Docker, automaatiotestaus

Opinnäytetyössä esitellään, miten Robot Framework -ohjelmistokehystä voidaan hyödyntää osana DevOps-ohjelmistokehitysmallia. Tavoitteena oli luoda kehitysympäristö, jolla kehittäjä voi kirjoittaa testitapauksia, jotka suoritetaan automaattisesti osana ohjelmiston jatkuvaa integraatiota ja toimitusta. Käytännössä testausympäristö kehitettiin siten, että jokaiselle ohjelmistoon tehdylle muutokselle pystyttiin automaattisesti suorittamaan vaaditut testaukset, minkä jälkeen ohjelmisto vieminen tuotantoympäristöön voitiin suorittaa automaattisesti.

Opinnäytetyön aikana tutustuttiin ohjelmistokehityksen teoriaan paneutumalla perinteisiin ja ketteriin ohjelmistokehitysmalleihin. Lisäksi selvitettiin, mitä testaustyyppejä ohjelmistokehityksen aikana voidaan hyödyntää ja miten testausautomaatiota hyödynnetään osana DevOps-ohjelmistokehitysmallia.

Työtä varten myös tutustuttiin yleisiin testausautomaatiossa käytettyihin työkaluihin. Robot Frameworkin lisäksi käytiin läpi Selenium WebDriver -kirjaston hyödyntämistä osana käyttöliittymä testausta. Docker-ohjelmistoon tutustuminen myös koitui tarpeelliseksi, sillä ohjelmiston kontitusta käytettiin hyväksi monessa vaiheessa työtä.

Työssä saatiin aikaiseksi Vue.js-pohjainen testisovellus, jonka käyttöliittymätestausta varten kirjoitettiin Robot Framework -testitapaukset. Ohjelmiston kehitysympäristö konfiguroitiin siten, että ohjelmistoon tehdyt muutokset testattiin CI-/CD-putken avulla, minkä jälkeen testattu ohjelmisto vietiin julkisesti käytettävään testausympäristöön.

Abstract

Author: Känsälä Joel

Title of the Publication: Test Automation Development Using Robot Framework

Degree Title: Bachelor of Engineering

Keywords: DevOps, Robot Framework, Docker, Test Automation

This thesis covers methods of how Robot Framework can be utilized as part of the DevOps software development cycle. The goal was to create a software development environment, where developers can write automated test cases which are run as part of the software's continuous integration and delivery pipeline. In practice, whenever changes were made to the software during testing, they were automatically integrated and tested before being delivered into a production environment.

In this thesis, software development theory is discussed by focusing on traditional and agile software development models. In addition, the common software testing methods used during software development were explored, along with how test automation can be leveraged as part of DevOps software development.

As part of the thesis, common software tools used in test automation were also examined. In addition to Robot Framework, the use of Selenium WebDriver library as part of user interface testing was studied. Docker software was also necessary, as container technology proved invaluable as it was used multiple times in the development process.

The result of the project was a Vue.js based test application with included Robot Framework test cases for user interface testing. Development environment was also configured to test changes made as part of the CI/CD pipeline. After passing the required tests, the application was automatically deployed into a production environment using continuous deployment.

Sisällys

Symboliluettelo

API: Application Programming interface. Ohjelmointirajapinta. Määrittelee, miten eri ohjelmat keskustelevat keskenään.

Artefakti: Ohjelmistotuotannon aikana kerätty tietoa sisältävä sivutuote.

Binääritiedostot: Tietokoneen luettavaksi tarkoitettu ei-teksti tiedosto. Usein lopputuloksena käännetyille ohjelmille.

Continuous Delivery (CD): Jatkuva toimitus. DevOps-ohjelmistotuotannon menetelmä, jonka tarkoituksena on lyhentää julkaisuun menevää aikaa automatisoimalla testausta ja hyväksyntää.

Continuous Deployment: DevOps-ohjelmistotuotannon menetelmä, jossa automatisoidaan valmiin ohjelmiston vienti tuotantoympäristöön.

Continuous Integration (CI): Jatkuva integraatio. Ohjelmistotuotannon menettely, jossa usean tekijän muutokset yhdistetään yhteen ohjelmistoprojektiin.

DevOps: Ohjelmistokehitysmalli, joka yhdistää ketterät menetelmät ja nopean ohjelmistoiteraation.

Paketinhallintajärjestelmä: Tietokoneohjelma, joka hallitsee tietokoneelle asennettuja paketteja ja ylläpitää tietoa saatavista versoista.

Repositorio: Pakettivarasto. Tietovarasto, jossa säilytetään verkon yli jaettavia ohjelmistopaketteja.

SPA: Single Page Application. Websovellus tai nettisivu, joka päivittää sivua dynaamisesti lataamatta täysin uutta sivustoa.

Staging: Ohjelmistoprosessi, jossa ohjelmisto rakennetaan ja testataan ennen käyttöönottoa tuotantoympäristössä.

Webhook: Webpohjainen menetelmä muokata webpohjaisen sivun tai sovelluksen toiminnallisuutta, kun käyttäjä suorittaa tietyn toiminnon.

Zip-tiedosto: Tiedoston pakkaukseen käytetty tiedostomuoto. Yleisesti käyttää tiedostopäätettä .zip.

1 Johdanto

Työn toimeksiantajana toimi eräs suomalainen ohjelmistokehitysyritys, joka kehittää asianhallintajärjestelmiä ja ratkaisuja monille eri toimialueille. Toimeksiantajan kehittämää tuotetta on tähän asti kehitetty vähäisellä testauksella. Tuotekehityksessä on kuitenkin ilmennyt selvä tarve automatisoidulle käyttöliittymätestaukselle. Tämä siirtäisi kehittäjien ja testaajien työmäärää vähentämällä palautuvia virheitä sekä manuaalista testaamista.

Opinnäytetyön aikana selvitetään ohjelmistotestauksen teoriaa ja luodaan esiteltävä malli, miten Robot Framework -kirjastolla voidaan suorittaa testausautomaatiota osana ohjelmiston kehitysprosessia. Käytännön työ suunnitellaan siten, että kehittäjät ja testaajat voisivat testausympäristössä luoda valmiita testejä usein toistuville tai aikaisemmin todetuille ongelmille. Tämän jälkeen he voivat ajaa testit paikallisesti sekä automaattisesti jokaisen muutoksen yhteydessä osana ohjelmistokehityksen tuotantolinjaa.

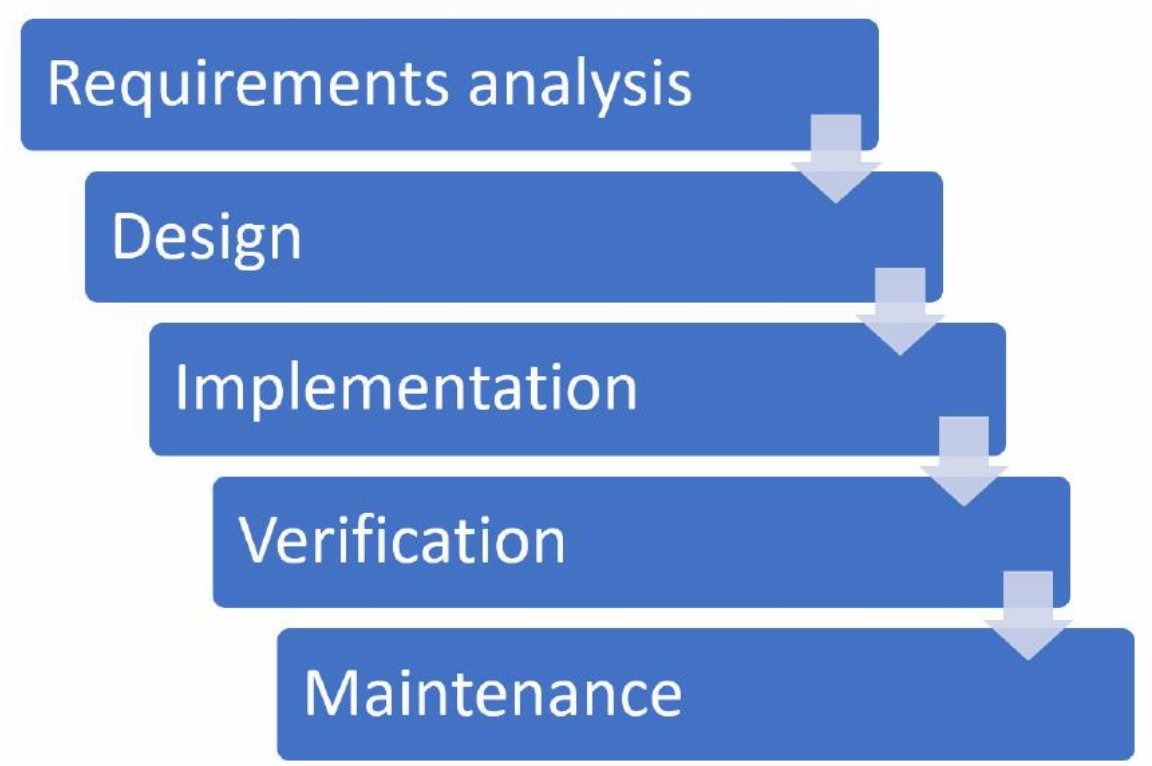
Testausautomaatiosta on tullut välttämätön osa tuotekehitystä. Vajaa ohjelmistotestauksen määrä lisää virheitä koodissa, mikä aiheuttaa ylimääräistä työtä, vähentäen tehokkuutta. Läpikotaisesti testattu koodi taas antaa kehittäjilleen varmuutta, että tuotettu ohjelmisto toimii sovitulla tavalla. Tämä auttaa havainnoimaan virheitä varhaisessa vaiheessa, vähentää ylimääräistä työtä ja nopeuttaa kehitystä.

2 Ohjelmistotestauksen teoria

Ohjelmistotuotannolla tarkoitetaan tieteen alaa, jossa käsitellään tietokoneohjelmien valmistamista. Ohjelmistotuotantoa tutkimalla pyritään löytämään aina parempia ohjelmiston valmistamiseen soveltuvia periaatteita, joiden avulla tehostetaan toimintaa, luodaan korkeampaa laatua ja karsitaan ylimääräisiä kustannuksia. Ohjelmistotuotannolla on pitkä historia ja ohjelmistojen vaatimusten monimutkaistuessa ohjelmistokehitystä on kehitetty tehokkaammaksi ja ketterämmäksi. [1.]

2.1 Ohjelmistokehitysmallit

Vesiputousmalli-ohjelmistokehitysmalli perustuu ohjelmiston kehityskaareen, jossa on selkeä prosessi ohjelman rakentamiselle. Vesiputousmalli jakaa ohjelmistokehitysprojektissa tuotettavat palaset selkeiksi kehitysvaiheiksi, missä jokainen vaihe riippuu edellisen vaiheen tuotoksista. Nämä vaiheet voidaan suorittaa kuvassa 1 kuvatussa järjestyksessä. [2.]



Kuva 1: Ohjelmistokehityksessä käytetty vesiputousmalli. Ylhäältä luettuna: vaatimusmäärittely, suunnittelu, toteutus, tarkistus ja ylläpito. [2.]

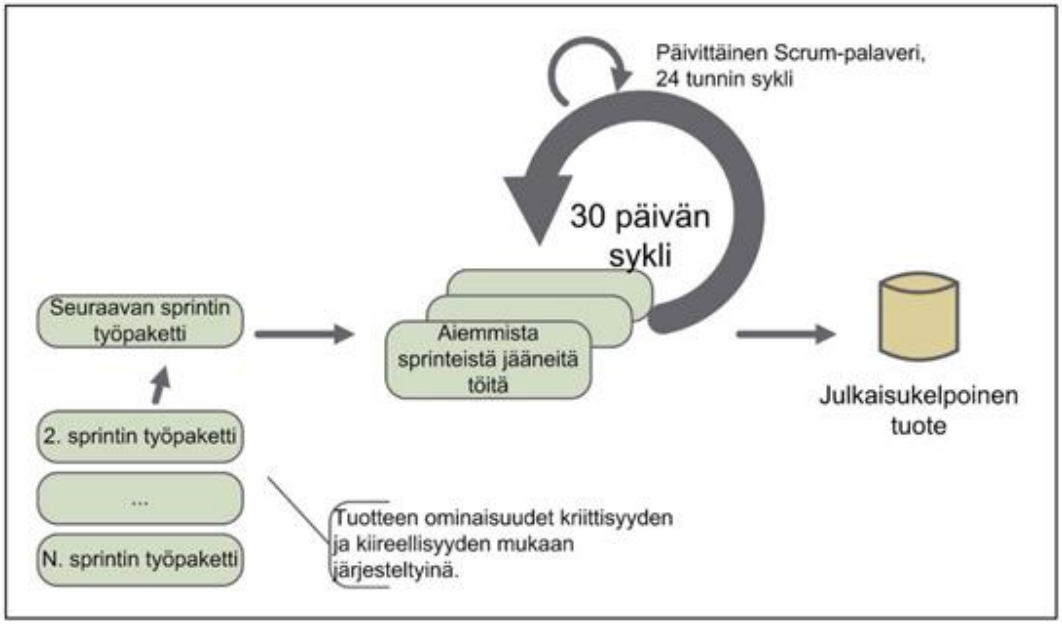
Vesiputousmalli toimii, kun ohjelmiston vaatimukset ovat määritelty tarkasti ja on oletus, että vaatimukset eivät muutu ohjelmistokehityksen aikana. Asiakas ottaa kantaa ohjelmiston kehitykseen vain projektin alussa, koska tuote toimitetaan projektin lopussa yhdessä erässä. [2.]

Todellisessa elämässä asiakas ei tiedä kaikkia ohjelmiston vaatimuksia tai uusien ominaisuuksien tarpeellisuus voi ilmetä vasta projektin aikana. Vaatimusten muuttaminen voi tulla kalliiksi, etenkin jos muutokset tehdään projektin lopussa. Vesiputousmallin käyttö ohjelmistokehityksessä onkin nykyaikana tästä syystä väistynyt uudempien ohjelmistokehitysmenetelmien tieltä. [2.]

Suurin osa ohjelmistokehitysprojekteista ei voi noudattaa tarkkaa ohjelmistosuunnitelmaa: Asiakkaan vaatimukset voivat muuttua, testauksessa ilmenee käytettävyysongelmia tai ominaisuuden toteutus osoittautuu teknologiseksi mahdottomuudeksi. Kehittäjien pitää kyetä muuttamaan suunnitelmia. Raskas hallinto ja byrokratia voivat hidastaa ohjelmistokehitystä, jos jokaisen muutoksen vaiheessa pitää tehdä edeltävät vaiheet alusta alkaen. Ketterät menetelmät perustuvatkin ohjelmistokehitysprosessin hallinnoinnin keventämiseen, jotta projektin aikana tapahtuviin muutoksiin pystytään, ainakin teoriassa reagoimaan ”ketterämmin”. [3.]

Vaikka ketterien menetelmien periaatteet ovat paljon vanhempia, vuonna 2001 julkaistu ”Ketterän ohjelmistokehityksen julistus” (Manifesto for Agile Software Development) [4] toi nämä valtavirran huomioon. Yksi suosituimmista ketterän ohjelmistotuotannon menetelmistä on Scrum-menetelmä. Scrum on suhteellisen yksinkertainen ja helppo hallinnoida ja on saavuttanut menestystä yritysmaailmassa. Scrum toimii parhaiten projekteissa, jossa työskennellään yhdessä suhteellisen tiiviissä tiimissä, jossa kaikki tiimin jäsenet pystyvät helposti jakamaan tietoa keskenään. [3.]

Scrum-mallissa projektissa tehtävä työ jaotellaan selkeiksi osiksi asiakkaan toiveiden ja vaatimusten perusteella. Näistä vaatimuksista luodaan tehtävälista, joka järjestetään tehtävien tärkeyden ja toteutusjärjestyksen perusteella. Projektin aikataulu jaotellaan erillisiksi ”sprinteiksi”, jossa kehittäjät päättävät, mitkä tehtävät suoritetaan ja miten ne toteutetaan. Vanhemmissa malleissa sprintti määriteltiin alle kolmenkymmenen päivän jaksoihin, mutta nykyaikana monet tiimit suorittavat kahden viikon pituisia sprinttejä [5]. Koska Scrum-mallissa työ jaotellaan iteraatioihin, vaatimuksia ja suunnitelmia voidaan muuttaa sprinttien välissä, jolloin testien tulokset ja asiakkaan palaute voidaan ottaa huomioon ohjelmistosuunnittelussa. Tavallisen sprintin rakenne on esitetty kuvassa 2. [3.]



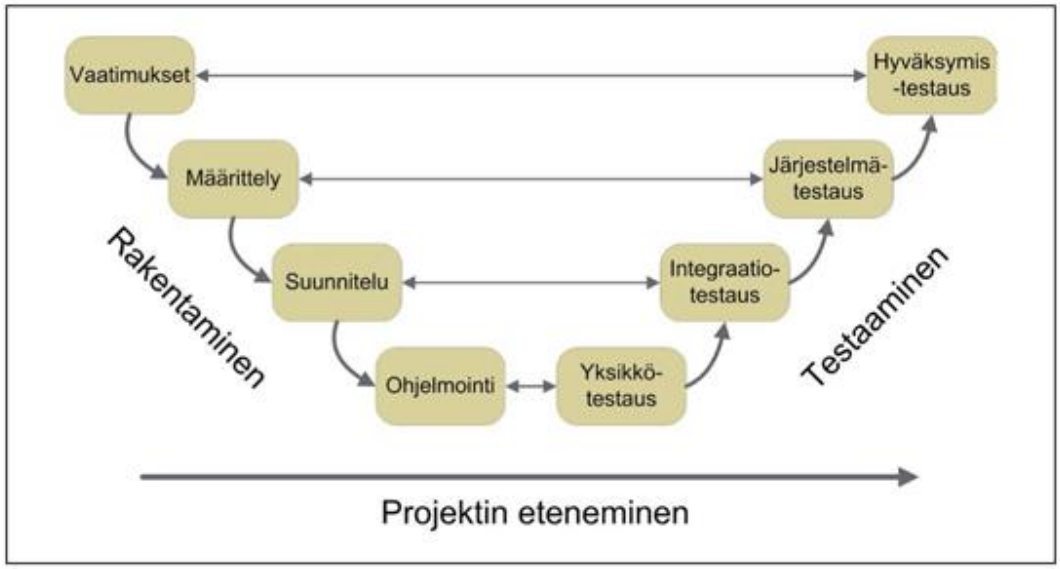
Kuva 2: Scrum-tiimissä suoritettava yksi sprintti-iteraatio [3].

2.2 Ohjelmistotestaus

Ohjelmistotestaus tarkoittaa työtä, jolla varmistetaan ohjelmiston toimivuus ja että sen toteutus vastaan sille määritettyjä vaatimuksia. Ohjelmistotestaus on osa ohjelmistotuotantoa, mutta se on kokonaisuutena paljon laajempi ja monimuotoisempi kuin esimerkiksi pelkkä ohjelmointi. Testaajan tehtävä ei välttämättä rajoitu pelkän testaajan piiriin, vaan kehittäjä voi joutua kirjoittamaan dokumentaatiota ja testaaja kirjoittamaan yksikkötestejä. Testauksen tavoitteet myös vaihtelevat ohjelmistokehitysprojektien välillä. Peliyritys voi painottaa käyttäjäkokemuksen testausta ja graafisten elementtien toimivuutta. Nettisivun testauksessa voidaan haluta varmistaa, että kaikki linkit toimivat ja sivusto on mahdollisimman helppokäyttöinen käyttää. [3.]

2.2.1 Testaustyypit

Ohjelmistotestauksen tasoja ja vaiheita voidaan kuvata ohjelmistokehityksessä V-mallilla, joka kuvaa ohjelmiston rakentamisen ja testaamisen välistä yhteyttä kuvan 3 mukaisesti. V-malli jakaa ohjelmiston teknisen testauksen kolmeen eri vaiheeseen: Yksikkö-, integraatio- ja järjestelmätestaukseen. Hyväksymistestaus on V-mallin viimeinen testausvaihe ja se suoritetaan yleensä lopullisessa tuotantoympäristössä ennen valmiin ohjelmiston luovuttamista asiakkaalle. [3.]



Kuva 3: Ohjelmistotestauksen V-malli [3].

Yksikkötestauksessa tarkastellaan yksittäistä ohjelmiston komponenttia. Yksikkötestauksen tarkoituksena on, että yhden moduulin, funktion tai olion toiminta varmistetaan kehityksen yhteydessä. Yksikkötestien rakentaminen kuuluu yleensä kehittäjille, jotka määrittelevät komponentin syötteet ja sen tulokset. [3.]

Yksikkötestausta seuraa integraatiotestaus, jossa ohjelmiston komponentteja yhdistetään toiminnallisiksi osiksi. Integraatiotestauksen tavoitteena on käytännön tasolla kokeilla, että järjestelmän osat toimivat myös yhdessä. Integraatiotestauksen testitapaukset ovat laajempia kuin yksikkötestit, mutta eivät välttämättä sisällytä koko järjestelmän toimintaa. Kehityksen aikana voi olla tarpeellista kehittää ohjelmistotynkiä, jotka tilapäisesti mallintavat puuttuvien komponenttien toimintaa. [3.]

Järjestelmätestauksessa tarkastellaan ohjelmiston toimintaa kokonaisuutena. Toisin kuin integraatiotestauksessa, ohjelmisto on täysin tai lähes täysin, koottu eikä sisällä puuttuvia toiminnallisuuksia. Yleensä järjestelmätestauksella tarkoitetaankin sitä testaustyötä, mitä tehdään kokonaiselle järjestelmälle. Varsinainen toteutus riippuu projektista ja sen aikana yleensä suoritetaan yleisesti musta-, harmaa- ja lasilaatikko-testauksia sekä regressiotestausta. Järjestelmätestauksen aikana toimitaan vielä testiympäristössä. Vaikka tämä ympäristö mallintaisikin kohdeympäristöä erittäin tarkasti, kohdeympäristössä suoritettua testausta kutsutaan hyväksymistestaukseksi. [3.]

Hyväksymistestaus suoritetaan ennen projektin luovuttamista asiakkaalle ja sen tarkoituksena on osoittaa tuotteen olevan tarpeeksi korkealaatuinen ja kyvykäs täyttämään asiakkaan kanssa tehdyt vaatimusmäärittelyt. Hyväksymistestauksessa siis tarkastetaan virallisesti tuotteen olevan valmis. Hyväksymistestauksessa on myös tavallista, että järjestelmää käytetään asiakkaan omassa ympäristössä eikä esimerkiksi tuotekehityksen omalla testipalvelimella. [3.]

2.2.2 Korkeamman tason testimenetelmät

Testaajalle tärkeä testausmenetelmä on mustalaatikkotestaus, jota voidaan myös kutsua testipohjaiseksi tai syöte/tulos- pohjaiseksi testaukseksi. Tässä menetelmässä ohjelmistoa tarkastellaan ”mustana laatikkona”, jonka sisäisestä toiminnasta ei tunneta mitään. Mustalaatikkotestauksessa keskitytään löytämään tilanteita, joissa ohjelmisto ei toimi toivotulla tavalla. Yleensä pelkästään mustalaatikkomenetelmällä tehty testaus ei ole kattavaa, sillä kaikkien syötteiden läpikäynti olisi teoreettisesti ja taloudellisesti mahdotonta. Mustalaatikkotestit pitäisi siis valita siten, että mahdollisimman monta virhettä pystytään löytämään mahdollisimman pienellä määrällä testitapauksia. [6.]

Lasilaatikkotestaus eroaa mustalaatikkotestauksesta siten, että lasilaatikkotestauksessa ohjelmiston sisäinen toiminta on täysin tunnettu. Lasilaatikkotestauksessa testattavat syötteet määritellään lähdekoodista, jolloin toimitaan käänteisesti mustalaatikkotestaukseen verrattuna. Tarkoituksena lasilaatikkotestauksessa on valita lähdekoodin perusteella syötteitä, joilla tarkistetaan kaikki mahdolliset tilanteet koodissa [6]. Lasilaatikkotestaus vaatii hyvän tiedon lähdekoodista, mistä syystä sitä tekevät pääasiassa kehittäjät. Lasilaatikkotestauksella varmistetaan, että järjestelmän tuottamat tulokset eivät ole sattumanvaraisia ja että järjestelmä toiminta kattaa kaikki mahdolliset tilanteet. [3.]

Harmaalaatikko testauksessa pyritään yhdistämään mustalaatikkotestauksessa mallinnetut testitapaukset sekä lasilaatikkotestauksen lähdekooditarkastukset. Valitsemalla lasilaatikkotestauksella valittuja syötteitä ja suorittamalla niitä mustalaatikkotestausmenetelmillä, testaaja varmistaa ohjelmiston toiminnallisuuden yleisimmissä tilanteissa. [7.] Se soveltuu tilanteisiin, jossa vain osa järjestelmän toiminnasta on tunnettu, esimerkiksi kun järjestelmä käyttää hyväksi kolmannen osapuolen ohjelmia tai kirjastoja [3].

Edellä esiteltyjä testausmuotoja käytetään pääasiassa yksikkö-, integraatio- ja järjestelmätestauksen aikana. Regressiotestaaminen ei suoraan sijoitu tiettyyn ohjelmistokehitysvaiheeseen, vaan toimii yleisterminä kaikelle testaukselle, jolla varmistetaan järjestelmän toimivuus muutosten jälkeen. Tärkein asia, mitä regressiotesteillä varmistetaan, on että aikaisemmin korjatut virheet eivät esiinny muutosten jälkeen. Regressiotestaus voidaan suorittaa aina, kun projektissa on saavutettu osatavoite tai muuten tehty merkittäviä muutoksia, esimerkiksi kun ohjelmistosta asennetaan uusi versio tuotekehityspalvelimelle. [3.]

2.2.3 Käytettävyystestaus

Käytettävyystestauksessa varmistutaan järjestelmän käyttöliittymän toimivuudesta ja helppokäyttöisyydestä. Käytettävyystestaus ei rajoitu pelkästään projektin loppuun, vaan iso osa siitä voidaan suorittaa jo suunnitteluvaiheessa. [3.]

Käytettävyystestausta voidaan tehdä useilla eri menetelmillä. Yleisiä tapoja ovat käyttäjäkokeilut ja niitä seuraavat haastattelut, asiantuntijatestit ja kohdekäyttäjäryhmä testaus. Käytettävyystestauksessa voidaan myös hyödyntää työkaluja, joilla kerätään käyttödataa. Testaustilanteista voidaan tallentaa napinpainallukset ja klikkaukset, katseenseurantamonitorilla silmän liikkeet tai videoida koekäyttäjän kasvonilmeitä järjestelmän käytön aikana. [3.]

2.2.4 Kuormitustestaus

Kuormitustestauksessa ohjelmistoa kuormitetaan mallintamalla lopullista käyttötapaa. Tätä voidaan tehdä esimerkiksi luomalla virtuaalikäyttäjiä, jotka suorittavat normaalia käyttöä vastaavia toimintoja, kuten sisäänkirjautumista ja tiedon lisäämistä, lukemista ja poistamista. Käytännössä siis kuormitustestauksen suorittaminen tehdään automaatiotestausmenetelmillä, mutta riippuen kehitettävästä ohjelmistosta voidaan kuormitustestausta tehdä alfa- ja beta-testauksella, jossa ohjelmistoa käyttää suuri käyttäjämäärä. [3.]

Kuormitustestauksella on kolme tehtävää. Tunnistaa järjestelmän pullonkaulat, selvittää, millaiseen maksimikapasiteettiin järjestelmä pystyy ja varmistaa laitteiston suorituskyky normaaleissa olosuhteissa ilman pitkiä vasteaikoja, virheitä tai kaatumisia. [3.]

2.2.5 Testausautomatisaatio

Hyvin suunniteltu testausprosessi on välttämätön laajassa ohjelmistoprojektissa. Hyvä suunnittelu myös mahdollistaa automaatiotestauksen hyväksikäytön kehityksen aikana. Haastavin asia automaatiotestauksen lisäämisessä on varmistaa, että lisätään oikeantyyppistä testausta. [3.]

Toisin kuin usein ajatellaan, automaatiotestaus ei korvaa käytännön testausta. Automaatiotestaus varmistaa, että ohjelma toimii, kuin taas käytännön testauksen tarkoituksena on rikkoa ohjelmistoa ja siten löytää virheitä. [3.]

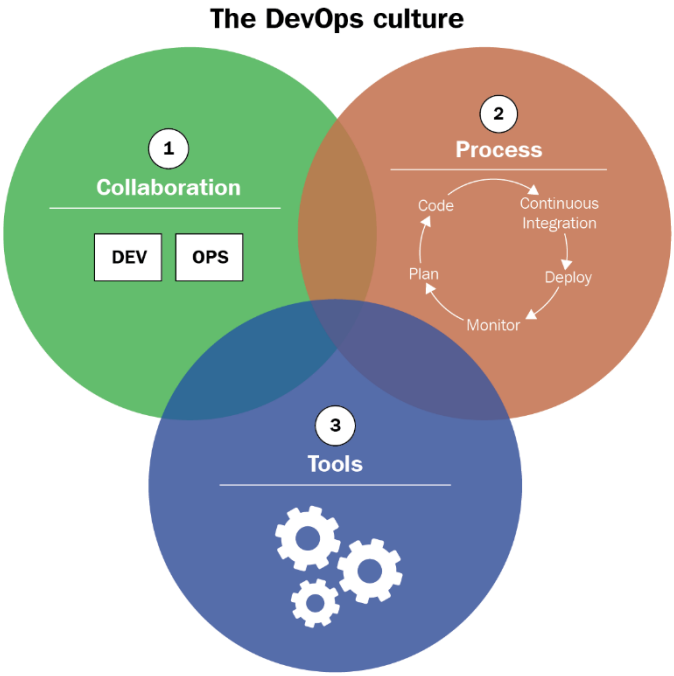
2.3 DevOps-ohjelmistokehitys

DevOps on ohjelmistonkehitys malli, jossa yrityksen ohjelmiston kehitysyksikkö ja operatiivinen osaston toiminta pyritään yhdistämään mahdollisimman sulavaksi. DevOps-kulttuurin tavoitteena on nopeuttaa ohjelmistokehitysprosessia, missä kehittäjät voivat luoda ominaisuuksia, jotka hyödyttävät asiakkaita ja IT-ylläpitäjät voivat varmistaa järjestelmien vakauden ja toiminnallisuuden. [8.]

DevOps kehittyi vastavetona yleisiin ongelmiin ohjelmistokehityksessä. Aikaisemmin melkein jokaisessa IT-organisaatiossa kehitystiimi ja operaatio tiimi eroteltiin erillisiksi osastoiksi, mikä monesti johtaa ongelmiin, kuten teknisen velan kertymiseen, johtuen vaillinaisesta kommunikaatiosta osastojen välillä. Tekniseksi velaksi kutsutaan kaikkia päätöksiä, jotka myöhemmin tuottavat ongelmia, joita on vaikeampi ratkaista myöhemmin ja jotka vähentävät vaihtoehtoja tulevaisuudessa. [9.]

DevOps-kehitysmalli voidaan ajatella olevan edistynyt muoto ketteristä ohjelmistokehitysmalleista, jotka parantavat kehittäjien sekä liiketoimintatiimin välistä toimintaa, mutta jättävät operatiivisen puolen pois. DevOps-kehitysmallin päätavoitteena on parantaa kehittäjien ja ylläpitäjien välistä kommunikaatiota. Tällöin pystytään parantamaan ohjelmiston tuottamiseen käytettyä prosessia alusta loppuun, jolloin ohjelmiston käyttöönottoja voidaan suorittaa useammin ja nopeammin. [8.]

DevOps-mallin toiminta perustuu kolmeen pääperiaatteeseen: yhteistyön kulttuuriin, prosessiin ja työkaluihin. Nämä osa-alueet eivät ole täysin itsenäisiä vaan liittyvät toisiinsa, kuten kuvassa 4 on esitetty. [8].



Kuva 4: DevOps-kehitysmallin kolme periaatetta: yhteistyö kehittäjien ja operaatioiden välillä, ohjelmistokehitysprosessi ja työkalut. [8.]

DevOps-mallissa on tärkeää, että kehittäjien ja operatiivisen yksikön välinen yhteistyö toimii sulavasti. Työntekijöitä ei erotella erillisiin yksiköihin työnkuvan perusteella, vaan jokainen tiimi sisältää kehittäjiä, järjestelmäasentajia sekä testaajia. Näiden tiimien tehtävänä on tuottaa lisää arvoa tuotteeseen mahdollisimman nopeasti. [8.]

Jotta järjestelmän käyttöönotto olisi mahdollisimman nopeaa, tiimien pitää noudattaa ketterän ohjelmistokehityksen periaatteita. Ohjelmistokehitys prosessin tulisi olla iteroivaa, mitattavaa ja palautetta antavaa. Prosessi tulisi myös integroida suoritettavan työn virtaukseen ja sen tulisi seurata DevOps-periaatteita. [8.]

Työkalujen valinta on tärkeää DevOps-mallissa. Työkalujen tulisi olla yhtenäiset tiimien välillä, jotta tiedonkulku tiimien välillä olisi mahdollisimman vaivattomasti. Kehittäjien tulee pyrkiä sisällyttämään monitorointi- ja tietoturvatyökaluja ylläpidon käyttöön, jotta he voivat varmistua ohjelmiston laadusta. Ylläpitäjien tulee kehittää koodin rakentamisen ja käyttöönottoon automatisointia sekä infrastruktuurin ohjelmallista kehitystä. [8.]

2.3.1 Jatkuva integraatio (Continuous Integration)

Jatkuva integraatio (Continuous Integration tai CI) on ohjelmistokehityskäytänne, jossa tiimin jäsenet integroivat oman työnsä usein – yleensä päivittäin – yhteiseen koodiperustaan. Jokainen integraatio tarkistetaan automaattisesti kehityspalvelimella, jolla voidaan myös ajaa määriteltyjä testejä automaattisesti. [10.]

DevOps-kulttuurissa jatkuva integraation hyödyntäminen vaatii, että tiimi sopii keskenään prosessista, miten uusia ominaisuuksia suunnitellaan, kehitetään ja yhdistetään kokonaisuuteen. Yleensä tällaisessa prosessissa käytetään hyväksi versiohallintapohjaisia työmalleja, jossa kehittäjä luo uuden versiohaaran ominaisuudesta, mikä voidaan arvioida ja katsoa läpi ennen yhdistämistä päähaaraan. Ohjelmistoautomaatiotyökaluilla, kuten Jenkins ja Azure DevOps, voidaan suorittaa muutosten integraatio mahdollisimman nopeasti ja havaita virheet ajoissa. [8.]

2.3.2 Jatkuva toimitus (Continous Delivery)

Kun jatkuva integraatio on suorittanut ohjelman rakentamisen ja yksikkötestauksen, voidaan se viedä automaattisesti yhteen tai useampaan ei-tuotantoympäristöön. Tätä työtä kutsutaan usein nimellä ”Staging” ja tämän vaiheen toteuttamiseksi käytetään jatkuvaa toimitusta eli Continuous Delivery (CD). [8.]

Jatkuvan integraation luomat ohjelmistopaketit (binääri-, kirjasto- ja Zip-tiedostot) asennetaan ympäristöihin käyttäen hyväksi automatisoituja tehtäviä. Näihin tehtäviin kuuluu toimintoja, kuten pura paketit, pysäytä ajettavat prosessit, korvaa konfiguraatiot, kopioi tiedostot jne. Jatkuva toimitus voi myös suorittaa asennuksen jälkeen ympäristölle järjestelmä- ja hyväksymistestejä. [8.]

Jatkuvan toimituksen suorittamat testit, pitäisi suorittaa koko järjestelmää ja sen riippuvuuksia vasten. CI:n ja CD:n suorittamien testien ero näkyy parhaiten, kun järjestelmä koostuu useasta eri osasta ja rajapinnasta. [8.]

2.3.3 Jatkuva käyttöönotto (Continuous Deployment)

Jatkuvan käyttöönoton voidaan ajatella olevan jatkoa jatkuvalle toimitukselle. Continuous Deployment tarkoittaa prosessia, joka automatisoi koko CI-/CD-putken, siitä hetkestä, kun kehittäjä lisää muutokset, siihen että muutokset viedään tuotantoon. [8.]

Jatkuvan käyttöönoton toteuttaminen vaatii tarpeeksi laajan testausautomaation, jotta ohjelmiston toimivuus voidaan varmistaa mahdollisimman hyvin. Testauksen pitää olla myös tarpeeksi laaja, että ohjelma voidaan viedä tuotantoympäristöön ilman hyväksyntää vaativia toimenpiteitä. Jatkuvan käyttöönoton prosessin pitää myös pystyä palauttamaan ohjelmiston aikaisempaan versioon, jos tuotannossa havaitaan ongelma. [8.]

2.4 Robot Framework

Robot Framework on geneerinen avoimen lähdekoodin ohjelmistoautomaatio. Sitä voidaan käyttää hyväksi monenlaisissa automatisoiduissa prosessihallinnoissa, mutta useimmin Robot Framework valitaan testausautomaation kehittämiseen. [11.]

Robot Frameworkin alkuperäinen kehittäjä on Pekka Klärck. Pekka Klärck suoritti diplomi-insinöörin tutkinnon Helsingin yliopistossa vuonna 2006 [12]. Klärckin Diplomityö: ”Data-Driven and Keyword-Driven Test Automation Frameworks” sisältää konseptin testiautomaation tukikehyksestä (Testautomation Framework). Tämä tuli toimimaan pohjana ensimmäiselle versiolle Robot Frameworkista, joka julkaistiin avoimena lähdekoodina vuonna 2008. [13.]

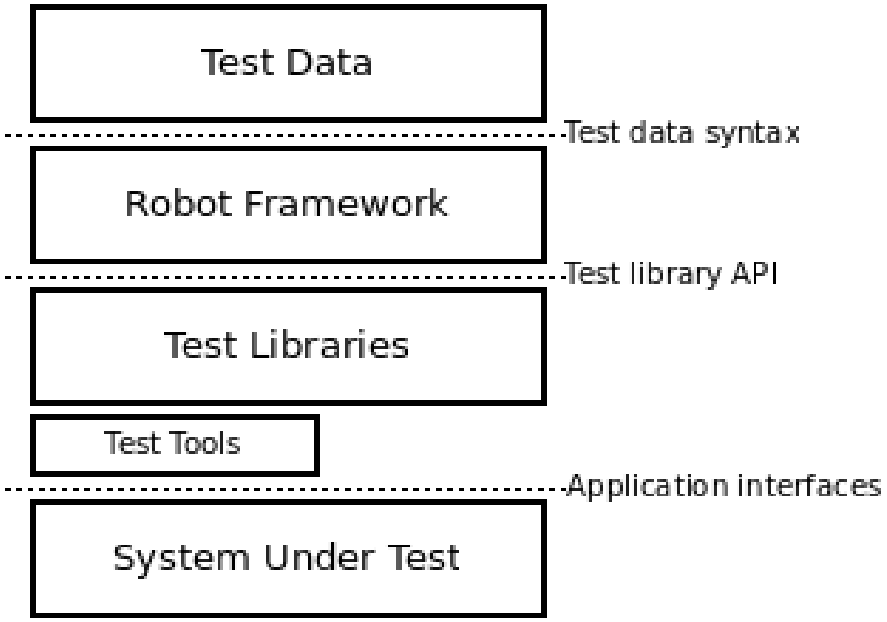
Yksi Robot Frameworkin vahvuuksista on sen laajennettavuus sisäänrakennettujen sekä ulkoisten kirjastojen avulla. Näiden avulla Robot Framework voi tehdä monenlaisia asioita, kuten käynnistää ja ohjata selaimia, luoda yhteyksiä ja muokata tietokantojen tauluja tai tehdä HTTP-kyselyjä. Robot Framework on kirjoitettu Python-kielellä, joten kehittäjä voi lisätä omia toiminnallisuuksia kirjoittamalla omia Python-kirjastoja, olettaen että olemassa olevat kirjastot eivät riitä. [11.]

2.4.1 Arkkitehtuuri

Yleisellä tasolla Robot Framework koostuu erillisistä kerroksista, jotka ovat esitetty kuvassa 1. Testausaineisto esitetään yleensä helposti luettavassa taulukkomuodossa, kuten Excel- tai CSV-

tiedostona. Robot Framework prosessoi tämän datan, ja suorittaa Robot Frameworkissa kirjoitetut testitapaukset ja luo näistä raportit HTML-muodossa. [14.]

Robot Framework ei itsessään tiedä mitään testattavasta ohjelmistosta, vaan kaikki interaktiot suoritetaan erillisten kirjastojen avulla. Kirjastoja voidaan lisätä riippuen testauksen vaatimuksista. Tässä työssä käytetään hyväksi Selenium Webdriver -kirjastoa selainten ohjaamisessa. [14.]



Kuva 5: Robot Framework Arkkitehtuuri [14].

2.4.2 Kehitettävyys ja laajennettavuus

Robot Frameworkin on implementoitu käyttäen Python-ohjelmointikieltä. Se voidaan kuitenkin myös kääntää muilla kielillä, kuten Jython tai IronPython. Tämä tekee Robot Frameworkista käyttöliittymä riippumattoman. Robot Frameworkin kirjoittaminen on suunniteltu mahdollisimman ihmisluettavaksi, jolloin testitapausten kuvaus ja toteutus vastaavat toisiaan lähemmin kuin muissa ohjelmointikielissä. [14.]

Koska Robot Framework toimitetaan Python-kirjastona, sitä voidaan laajentaa käyttämällä muita kirjastoja. Yksi yleisesti Robot Frameworkin kanssa käytetty kirjasto on Selenium WebDriver. Selenium WebDriverilla Robot Framework pystyy ajamaan nettiselaimia, jolloin pystytään mallintamaan ihmiskäyttäjää. [15.]

Kehittäjä voi myös tarvittaessa laajentaa Robot Framework -ympäristöä lisäämällä avainsanoja ja toimintoja. Tämä tapahtuu kirjoittamalla omia kirjastoja Python-kielellä, jotka tuottavat kehittäjän määrittämiä avainsanoja. Vaihtoehtoisesti kirjastoja voidaan kirjoittaa myös Java-kielellä, jos Robot Framework on käännetty Jython-kielellä tai .Net kielillä kääntämällä Robot Framework ensin IronPython-kielellä. [14.]

2.5 Selenium WebDriver -kirjasto

Selenium Webdriver on avoimeen lähdekoodiin perustuva selaimen automatisaatio-ohjelmisto. Selenium Webdriver käyttää selainta samalla tavalla kuin oikea käyttäjä. Selenium Webdriver voidaan lisätä kirjastona Robot Framework -kehykseen, jolloin testitapaukset pystyvät suorittamaan selaimella tehtyjä toimintoja. [15.]

Selenium Webdriver voidaan käyttää itsenäisesti tai se voidaan asentaa kirjastona käyttäen PyPi-pakettipalvelua. Selenium Webdriver -kirjastolla voidaan suorittaa testejä useimmilla eri selaimilla, kuten Chrome ja Firefox. [15.]

2.6 Docker

Docker on Linux- ja Windows-käyttöjärjestelmille asennettava ohjelmisto, jolla voidaan luoda ja hallita kontteja. Dockerin on alun perin kehittänyt Docker Inc. Alun perin Docker oli työkalu Linux-konttien luomista ja hallinnointia varten, josta jatkokehitettiin itsenäinen ohjelmisto, jonka käyttö on levinnyt maailmanlaajuiseen käyttöön. [16.]

“Kontti” on kevyt, eristetty sovellusympäristö, joka näkyy itsenäisenä laitteena ja sisältää kaiken sovelluksen ajamiseen. Jokainen näistä konteista eristetään toisistaan, mikä mahdollistaa useamman kontin ajamisen yhdellä koneella. Docker perustuu avoimeen lähdekoodiin, ja sen avulla voidaan ajaa kehitettäviä ohjelmistoja erilaisissa ympäristöissä ohjelmistokehitystä ja automaatiotestausta varten. [16.]

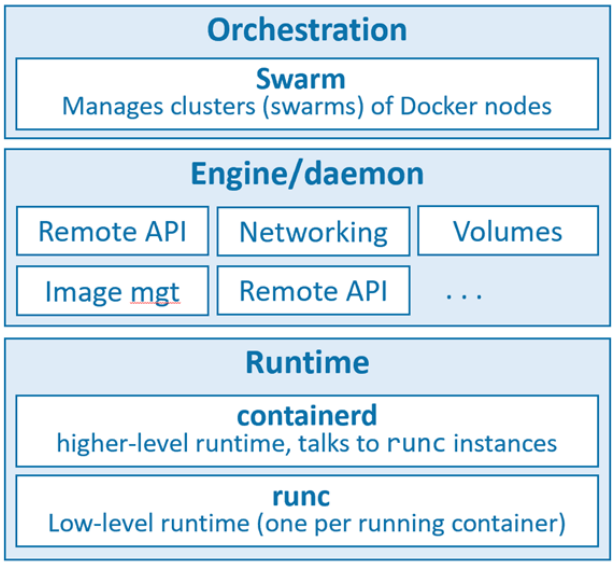
2.6.1 Yleistä

Dockerin toiminta perustuu käyttöjärjestelmän virtualisointiin. Yksi kone voi käynnistää ja ladata useita kontteja, jotka käyttävät hyväkseen samaa Linux-ydintä. Docker-kontit jakavat käyttöjärjestelmän resurssit keskenään, joten järjestelmän ei tarvitse jaotella resurssejaan ennen kontin luontia, olettaen että konttien käyttöjärjestelmän ovat identtisiä. Tämä on suurin eroavaisuus virtuaalikoneisiin. Virtuaalikonetta luotaessa, käyttöjärjestelmän pitää jakaa tietty määrä resursseja, kuten muistia ja prosessorikapasiteettia, virtuaalikoneen käyttöä varten. [16.]

Luodakseen kontin, Docker vaatii pohjan, jonka perusteella mallinnettava järjestelmä luodaan. Näitä pohjia kutsutaan ”Imageiksi” ja ne sisältävät kaiken tarvittavan tiedon, jolla Docker pystyy mallintamaan järjestelmää, kuten sovelluksen lähdekoodin, kirjastoriippuvuudet ja käyttöjärjestelmän rakenteet. Virtuaalikone ympäristössä imagea vastaava on Virtuaalikone-template. Samalla tavalla kuin VM template vastaa pysäytettyä virtuaalikonetta, Docker imagea voidaan ajatella pysäytetyksi Docker-kontiksi. [16.]

2.6.2 Docker-arkkitehtuuri

Puhuttaessa Dockerista, usein tarkoitetaan teknologiaa, joka sisältää kaiken tarvittavan konttien luomiseen ja ajamiseen. Tämä teknologian arkkitehtuuri rakentuu kuvan 6 mukaisesti ja koostuu kolmesta osasta: Runtime, Daemon tai Engine ja Orchestrator. [16.]



Kuva 6: Dockerin arkkitehtuurikuvaus [16].

Runtime-osio toimii lähimpänä käyttöjärjestelmää ja on vastuussa konttien käynnistämisestä ja pysäyttämisestä. Alemman tason toiminnoista huolehtii runc ja ylemmän tason ajosta huolehtii containerd. [16.]

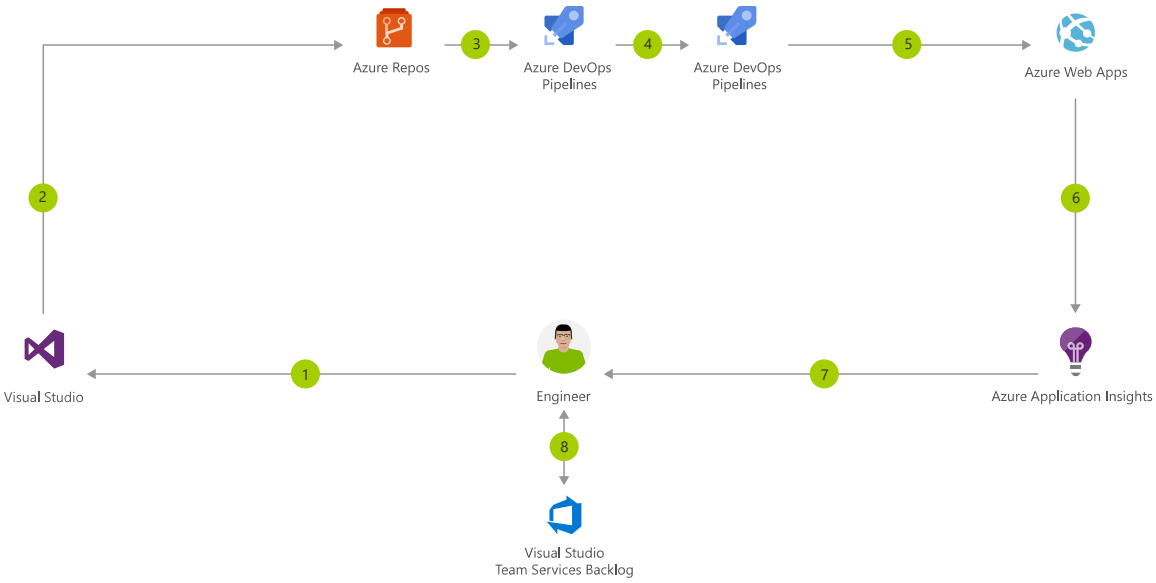
Jokaisen kontin luomista varten tarvitaan runc-instanssi. Yksinkertaistettuna runc on CLI wrapperi, jonka ainoa tarkoitus on kommunikoida käyttöjärjestelmän kanssa ja luoda kontteja. Yhdellä Docker-ympäristöasennuksella on yleensä yksi containerd-prosessi, joka hallitsee yhtä tai useampaa runc-instanssia, jokaista käynnistettyä konttia kohti. Containerd (lausutaan container-dee) hallitsee luotujen konttien elinkaarta. Containerd sisältää operaatiot konttien käynnistämiseen, pysäyttämiseen ja poistamiseen. Kuitenkaan containerd ei luo kontteja, vaan käyttää hyväkseen runc-instanssia kontin luomiseen. Konttien hallitsemisen lisäksi containerd:tä on laajennettu käsittelemään imageiden tuomista, volyymien luonnin ja verkostojen hallitsemisen. [16.]

Runtime-rajapinnan päällä toimii Docker Daemon. Sen tarkoituksena on tarjota standardi rajapinta, joka mahdollistaa Docker-toimintojen suorittamisen. Docker Daemon suorittaa ylemmän tason tehtäviä, kuten ylläpitää Dockerin etähallintarajapintaa, hallitsee ladattuja imageja, volyymeja sekä verkkoja. [16.]

Runtime ja Docker Daemon tason päällä toimiva Orchestrator tukee konttien ajamista klustereina, joita kutsutaan “Swarmeiksi”. Docker asennuksen yhteydessä toimitetaan tähän tarkoitukseen Docker Swarm -työkalu, mutta moni käyttää tämän tilalla Kubernetes-ohjelmistoa. [16.]

3 Testausympäristön kehittäminen

Robot Frameworkin hyödyntämisen havainnollistamiseksi kehitetään Vue.js-pohjainen web-sivu, johon kohdistuvat muutokset testataan ja päivitetään sovellukselle automaattisesti. Kuvasta 7 löytyy esimerkki Azure DevOps -palvelussa suoritettavasta CI-/CD-prosessista. Kehittäjä tekee muutoksia ohjelmiston lähdekoodiin ja vie muutokset Azure DevOps -versionhallintaan [17]. Viennin yhteydessä ajetaan CI-pipeline, joka suorittaa staattisen analyysin koodille rakentamalla koodin binääritiedostot. Jos kehittäjän muutoksista ei löydy ongelmia, viedään muutokset kehitysh aaraan.



Kuva 7: Esimerkki kuvitus Azure DevOps -palvelun CI-/CD-prosessista [18].

Tämän jälkeen voidaan käynnistää CD-pipeline, joka suorittaa ohjelmistolle Robot Framework-testit ja julkaisee ohjelmiston palvelimelle. Continuous deploymentin osana Azure DevOps luo Dockerilla kaksi konttia: yhden Vue.js-testiohjelmistolle, ja toisen Robot Framework -testeille. Azure-palvelu luo ensin testattavan ohjelmiston sisältävän kontin ja suorittaa sitä vasten Robot Frameworkillä kirjoitetut testit ja palauttaa tulokset. Kun testit ovat menneet onnistuneesti läpi, ohjelmistosta siirretään Docker Imagena Azure Container Registry -palveluun [19]. Kun tähän palveluun tulee uusi image, päivitetään Azure Web Application-palveluun uusin versio sovelluksesta imagen pohjalta.

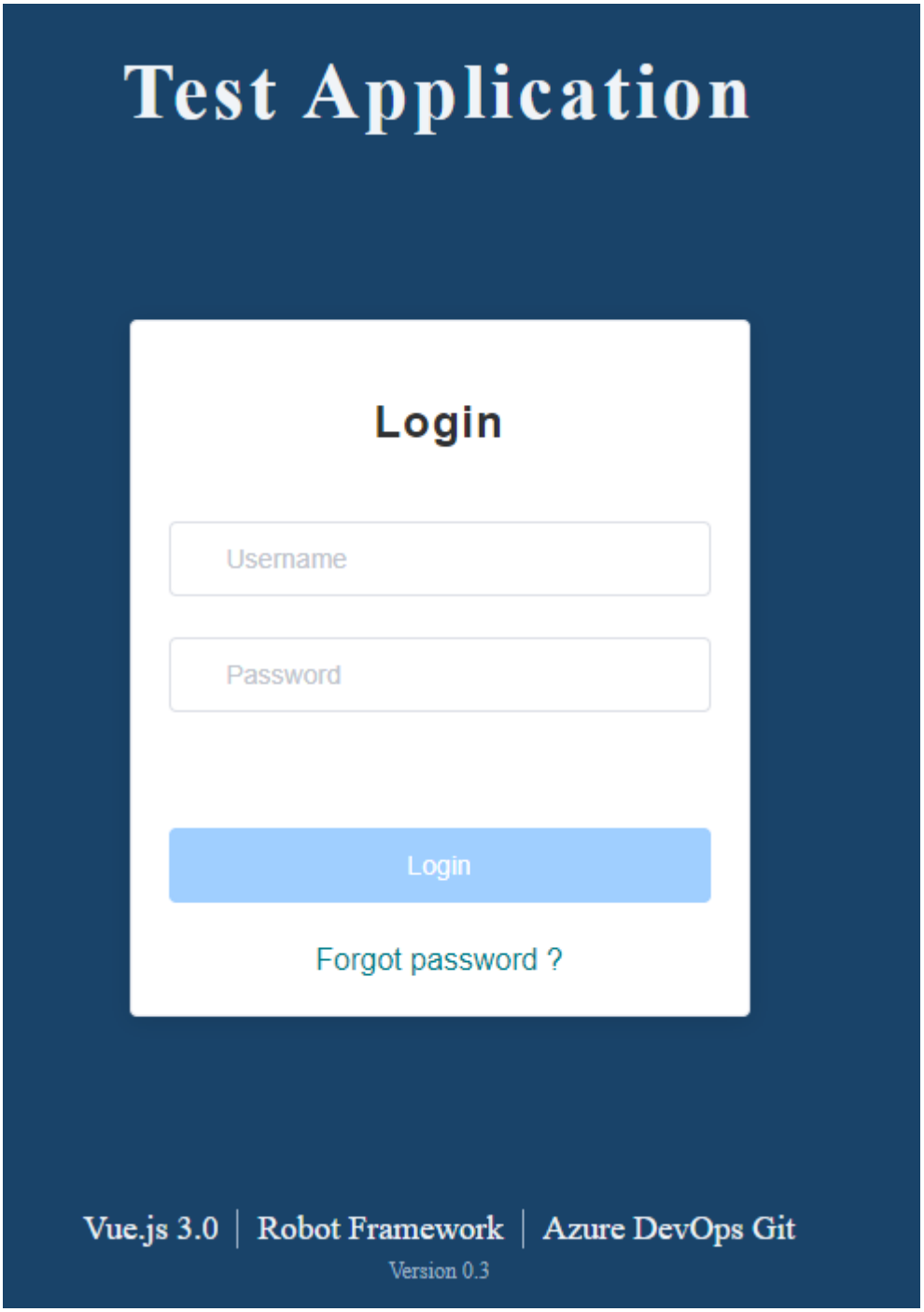
3.1 Kehitysympäristö

Ohjelmiston versionhallintaa varten luotiin repositorio Microsoftin Azure DevOps -palveluun. Projektin seurannan lisäksi Azure DevOps -palvelussa pystyy konfiguroimaan CI-/CD-putket, joita tässä työssä on tarkoitus demonstroida. Ohjelmiston lähdekoodin kirjoittamiseen käytettiin pääosin Visual Studio Code -tekstieditoria [20].

Web-ympäristön kehittämistä varten vaadittiin Node.js ajoympäristö. Node.js-ajoympäristö [21] mahdollistaa Javascript-koodin ajamisen paikallisessa ympäristössä. Lisäksi sen mukana tulee Node Package Manager -paketinhallinta työkalu (npm) [22]. Npm:n avulla asennettiin Vue.js-kehitysympäristö, jonka avulla voitiin kehittää yksinkertainen nettisovellus. Automaatiotestien kehittämistä varten asennettiin Python-ohjelmointikielen asennuspaketit, jonka mukana tuli Python Package Index -paketinhallinta työkalu (PyPI). PyPI-työkalulla asennettiin automaatiotestaus kehitystä varten Robot Framework, Selenium Webdriver ja Robot Framework DataDriver -kirjastot [23]. Docker-kehitystä varten asennettiin Docker Desktop Windowsille.

3.2 Testisovelluksen kehittäminen

Testausta varten luotiin yksinkertainen Vue.js-pohjainen websovellus. Sovellus mahdollistaa sisäänkirjautumisen, navigoinnin sekä lomakkeen täyttämisen. Sovellus sisälsi vain käyttöliittymän ja kaikki API-kutsut mallinnettiin käyttämään sisäistä API-tynkään, joka kirjoitettiin Javascript-kielellä. Kuvassa 8 näkyy kehitetyn websovelluksen kirjautumissivu.



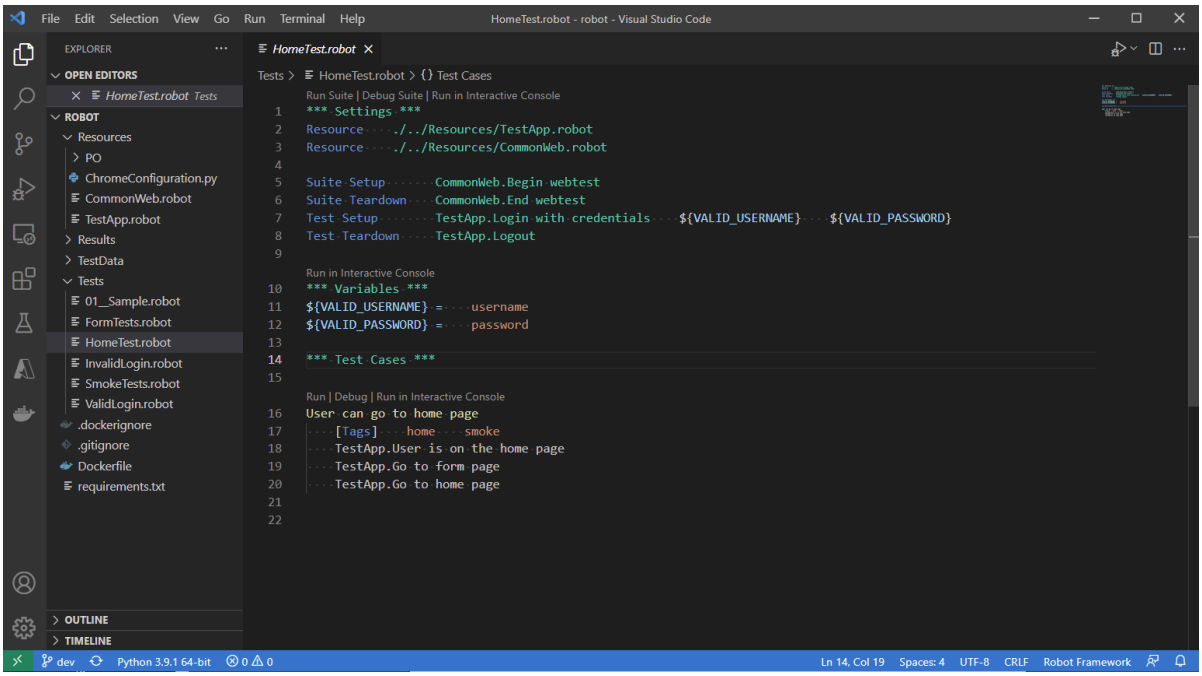
Kuva 8: Testisovelluksen kirjautumissivu

Testisovellus vastaa SPA-sovellusta, jossa on kolme näkymää: Kirjautumissivu, joka vaatii käyttäjältä käyttäjätunnuksen ja salasanan, käyttäjän oma sivusto ja lomakesivusto, jossa käyttäjä voi täyttää yksinkertaisen lomakkeen ja lähettää sen eteenpäin. Lomakkeen lähetyksen jälkeen sovellus suorittaa API-kutsua mallintavan toiminnon ja ilmoittaa käyttäjälle onnistumisen tai virheen.

Testisovellus voitiin käynnistää komentokehotteella Vue.js asennuksessa mukana tulleella Vue-cli-työkalulla. Tämä käynnistää paikallisen testiserverin, johon kehittäjä voi ottaa yhteyden nettiselaimella.

3.3 Testitapausten kehittäminen

Testitapausten kehittämiseen käytettiin Robot Framework -automaatioympäristöä. Projektin kehityksessä pyrittiin ottamaan huomioon, miten kehittäjät ja testaajat luovat uusia testitapauksia. Ohjelmistot ovat usein erittäin monimutkaisia, jolloin testausautomaation käytettyjen testitapausten määrä voi kasvaa räjähdysmäisesti, ellei niitä kehitetä uudelleen käytettäviksi ja helposti päivitettäviksi. Kuvassa 9 avatussa kehitysympäristössä näytetään, miltä yksinkertainen testitapaus näyttää.



Kuva 9: Robot Framework testien paikallinen kehitysympäristö Visual Studio Codessa.

Tämän vuoksi testitapausten kehittämisessä käytettiin Page Object Model- ohjelmistomallia [24]. Page Object Model on ohjelmistosuunnittelu malli, jossa web-sovelluksen käyttöliittymä jaetaan page objecteiksi. Yksi page objecti sisältää kaiken toiminnallisuuden koskien tiettyä osaa web-sovelluksesta. Esimerkiksi sisäänkirjautumisen toiminnallisuus voidaan sisällyttää ”login”- page objektiin, jonka funktioilla voimme uudelleen käyttää toimintoja. Jos tietty sivu sisällyttää erittäin

paljon toiminnallisuuksia, tarvittaessa page object voidaan jakaa komponentteihin, jotka voivat sisältää tiettyjä osia sivusta, joista muodostetaan monimutkaisempia sivustoja.

Testitapausten tarkoituksen oli testata testisovelluksen käyttöliittymän toiminnallisuutta. Tähän hyödynnettiin aikaisemmin asennettua Selenium Webdriver -kirjastoa. Tämän avulla kirjoitettiin testitapauksia, jotka mallintavat käyttäjän toimintoja nettisivulla. Tässä työssä testitapauksilla testattiin sisään- ja uloskirjautumista, navigointia sivujen välillä sekä lomakkeen täyttämistä ja lähettämistä.

Testitapauksia kehitettäessä otettiin myös huomioon testidatan lukeminen tiedostoista. Testitapaukset usein riippuvat erilaisista syötteistä. Jos jokaista testitapausta varten kirjoitettaisiin oma testitapaus, niin testien ylläpitäminen voi koitua mahdottoman raskaaksi. Data pohjaisessa testikehityksessä käytettiin hyväksi DataDriver-kirjastoa, jolloin kehittäjä voi kirjoittaa yhden tiedoston kuvan 10 mukaisesti. Tämän avulla testitapauksia voidaan kirjoittaa testitiedostoiksi eri tiedostomuodoissa, mistä luodaan erillisiä testitapauksia datan perusteella.

```
1  *** Settings ***
2  Resource    ./../Resources/CommonWeb.robot
3  Resource    ./../Resources/TestApp.robot
4
5  Library    DataDriver    file=../TestData/invalidLogin.csv
6
7  Suite Setup    CommonWeb.Begin webtest
8  Test Setup    CommonWeb.Refresh page
9  Suite Teardown    CommonWeb.End webtest
10
11 Test Template    User cannot login with invalid credentials
12
13 *** Test Cases ***
   Run | Debug | Run in Interactive Console
14 User cannot login with invalid credentials ${username} ${password}
15
16 *** Keywords ***
   Run in Interactive Console
17 User cannot login with invalid credentials
18     [Arguments]    ${username}    ${password}
19     TestApp.Try login with invalid credentials    ${username}    ${password}
20
```

Kuva 10: Robot Framework datapohjainen testi tiedosto.

Robot Framework -testien ajaminen paikallisessa ympäristössä tapahtui komentokehotteen avulla. Testien ajamisen jälkeen Robot Framework julkaisi testitulokset kansioon, josta kehittäjä pystyi tarkastelemaan tuloksia ajon jälkeen.

3.4 Dockerin käyttö testisovelluksen testauksessa

Kun testisovellus oli tarpeeksi kehitetty ja Robot Framework -testitapaukset kirjoitettu, aloitettiin sovellusten kontitus Docker-ajoa varten. Testisovelluksen ja automaatiotestien kansioon lisättiin Dockerfile-tiedosto. Kuvassa 15 on esitelty testisovelluksen Dockerfile- tiedoston sisältö.

```
# Build Stage
FROM node:lts-alpine AS build-stage

WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Production stage
FROM nginx:stable-alpine AS production-stage
COPY --from=build-stage /app/dist /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Kuva 11: Testisovelluksen Dockerfile-tiedosto, joka mahdollistaa sovelluksen ajon kontissa.

Testisovelluksen imagella luotiin kontti, joka ensin rakentaa testisovelluksen binääritiedostot ja aloittaa niiden jakamisen käynnistämällä paikallisen webpalvelimen käyttäen Nginx-ohjelmistoa [25]. Kontin käynnistäminen onnistuu komentokehotteen kautta ja käytännössä toimi vastaavasti kuin paikallinen ajo ilman Dockeria.

Robot Framework-testien kontitusta varten luotiin oma Dockerfile-tiedosto. Jotta Robot Framework testejä voidaan ajaa kontissa, pitää kontin ensin asentaa vaaditut ohjelmistot, kuten Python-kirjastot sekä Chromen webselain sekä sen webajurit.

Sovellusten kontittamisen jälkeen tarkasteltiin automaatiotestien ajamista kontitettua testisovellusta vasten. Tätä varten hyödynnettiin Docker Desktop -asennuksen mukana tulevaa Docker Compose -työkalua [26]. Docker Compose -ohjelmistotyökalu on tarkoitettu useammasta kontista koostuvien ohjelmistokokonaisuuksien määrittelemiseen. Ohjelmisto määritellään YAML-tiedostossa, jossa määritellään käytettävät kontit, niihin yhdistettävät volyymit ja niiden väliset verkkoyhteydet. Tämän jälkeen ohjelmistokokonaisuus voidaan käynnistää yhdellä komennolla.

Testisovelluksen ja automaatiotestien kontit määriteltiin kokonaisuudeksi kuvan 12 YAML-tiedoston mukaan.

```yaml
version: '3'

services:
  test-app-vue:
    build: ./test-app
    container_name: test-app
    ports:
      - "8080:80"
    volumes:
      - ./test-app:/usr/src/app
    networks:
      - test-network

  robot-tests:
    build: ./robot
    container_name: robot-test
    volumes:
      - ./robot/Tests:/usr/src/projects/Tests
      - ./robot/TestData:/usr/src/projects/TestData
      - ./robot/Resources:/usr/src/projects/Resources
      - ./robot/Results:/usr/src/projects/Results
    command: ["robot", "-d", "/usr/src/projects/Results",
              "-v", "BROWSER:headlesschrome",
              "-v", "URL:http://test-app:80", "/usr/src/projects/Tests/"]
    depends_on:
      - test-app-vue
    networks:
      - test-network

networks:
  test-network:
```

Kuva 12: Docker Compose YAML-tiedosto, joka määrittelee, miten testisovellus ja robot-testi toimivat yhdessä.

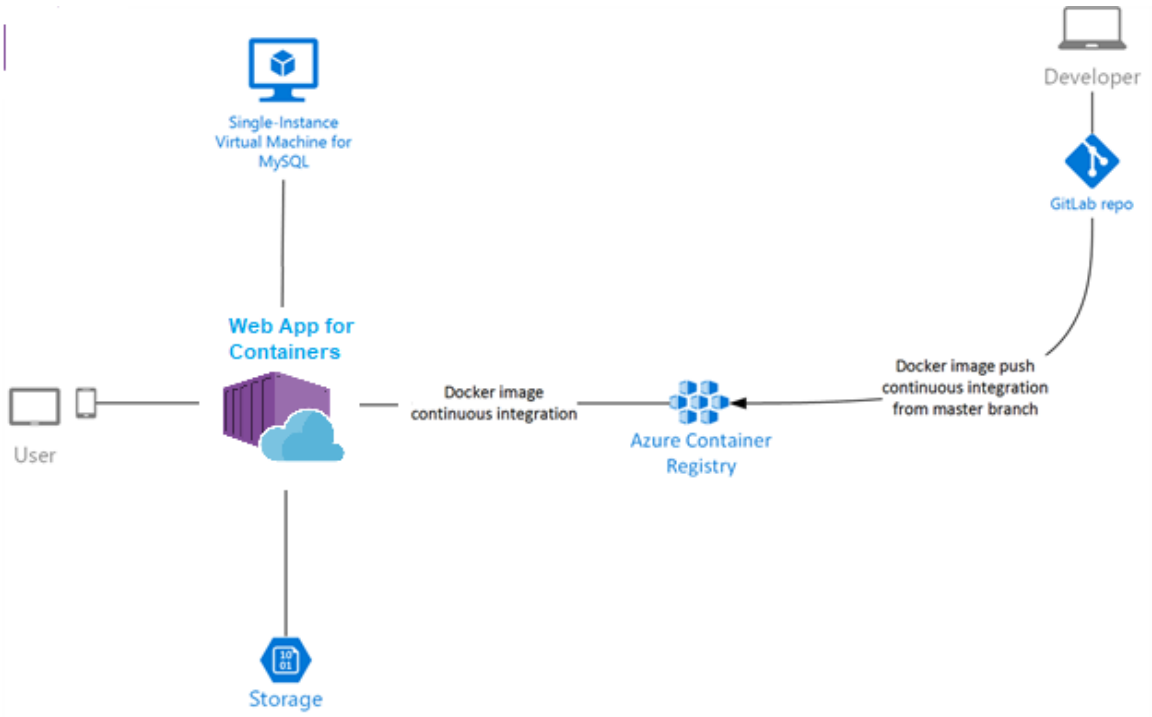
Kun ohjelmistokokonaisuus oli määritelty, kehittäjä pystyi suorittamaan testisovelluksen testauksen yhdellä komennolla. Tämä tarkoitti, että yhdellä komennolla Docker rakensi molemmat imaget ja käynnisti ne kontteina. Tämän jälkeen se loi aliverkon, käynnisti testisovelluksen ja ajoi automaatiotestit testisovelluksen konttia vasten.

3.5 Testien ajaminen osana CI-/CD-putkea

Kun testien ajo kontteina oli mahdollista, alettiin kehittää ohjelmiston viemistä ja testien suorittamista DevOps-ympäristössä. Testiohjelmiston lopullinen jakaminen suoritettiin Microsoftin Azuren-palveluilla. Sen jakamista varten luotiin Azuressa App Service-palvelu, jonka tehtävänä oli jakaa testiohjelmisto sivu käyttäjille.

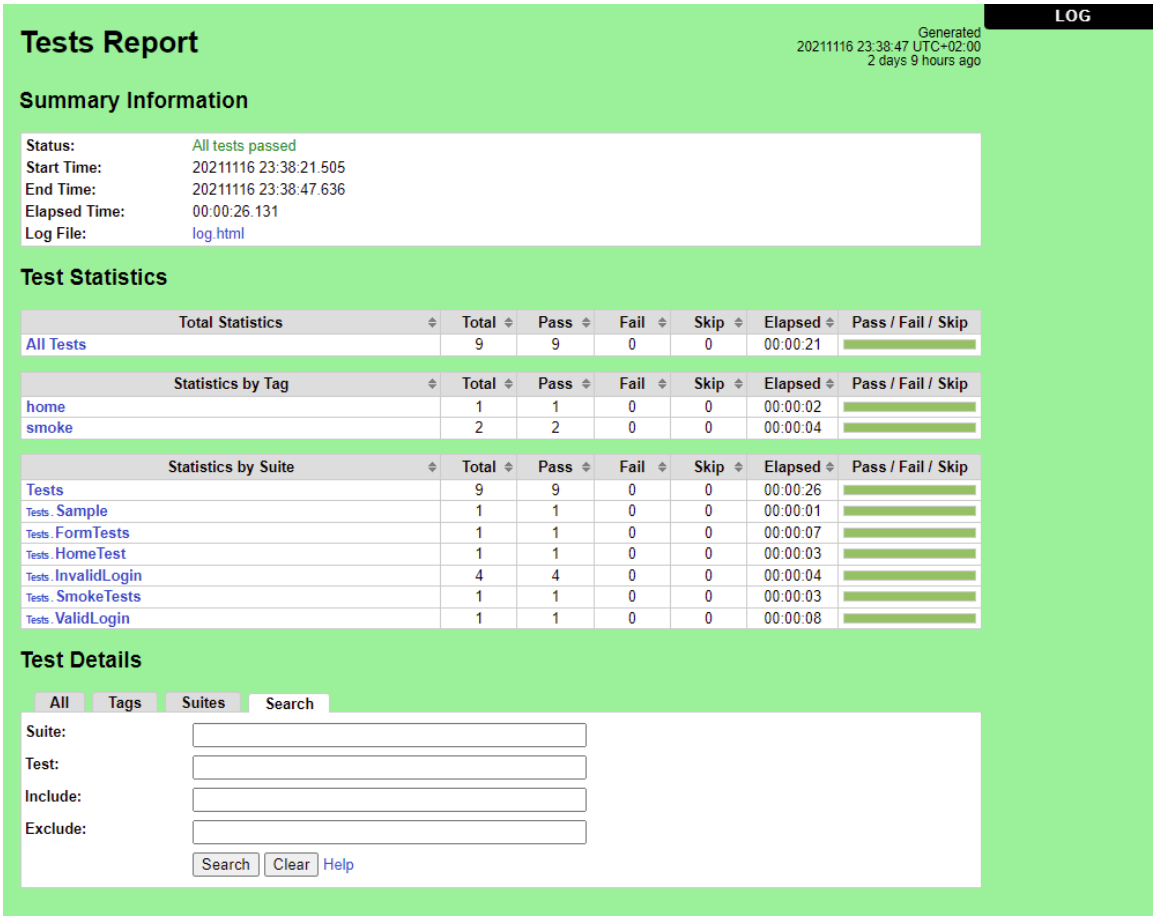
DevOpsissa toimivien Continuous Integration ja Continuous Delivery putket määriteltiin Azure Pipelinen toiminnoilla [27]. CI-/CD- putken tehtävät määriteltiin YAML-tiedostossa. Joka kerta kun ohjelmiston kehityshaaraan tehtiin muutoksia, käynnistettiin CI-putki, joka rakensin testiohjelmiston binääritiedostot.

CI-putken valmistuminen taas käynnisti automaattisesti CD-putken, jonka tehtävä oli suorittaa sovellukselle testit käyttäen Docker Composea ja julkaista tulokset artefaktina. Kun testit oli suoritettu onnistuneesti ja tulokset julkaistu, rakennettiin sovelluksen Docker image ja puskettiin se eteenpäin Azuressa palvelussa toimivaan Azure Container Registry resurssiin. Kuvassa 13 näkyy prosessi, miten uusimmat muutokset sisältävät Docker image päivittyy Azuressa App Service-palveluun.



Kuva 13: Esimerkkikuvaus Docker imagen viemisestä konttipohjaiseen Web App -palveluun [28].

Kun Azure Container Registry -palveluun päivittyi uusin versio testisovelluksen Docker-imagesta, käynnistyi webhook-palvelu, joka vei imagen Azuren App Service -palveluun, josta kontitettu ohjelmisto jaettiin käyttäjille. Robot Frameworkin tuottamat testitulokset tallennettiin Azure artefaktina pipelineajon yhteydessä, joka sisältää kuvan 14 mukaisen testiraportin.



LOG

Tests Report

Generated
20211116 23:38:47 UTC+02:00
2 days 9 hours ago

Summary Information

Status:	All tests passed
Start Time:	20211116 23:38:21.505
End Time:	20211116 23:38:47.636
Elapsed Time:	00:00:26.131
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	9	9	0	0	00:00:21	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
home	1	1	0	0	00:00:02	
smoke	2	2	0	0	00:00:04	

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
Tests	9	9	0	0	00:00:26	
Tests.Sample	1	1	0	0	00:00:01	
Tests.FormTests	1	1	0	0	00:00:07	
Tests.HomeTest	1	1	0	0	00:00:03	
Tests.InvalidLogin	4	4	0	0	00:00:04	
Tests.SmokeTests	1	1	0	0	00:00:03	
Tests.ValidLogin	1	1	0	0	00:00:08	

Test Details

All | Tags | Suites | Search

Suite:
Test:
Include:
Exclude:
Search Clear Help

Kuva 14: Robot Frameworkin tuottama testiraportti.

4 Yhteenveto

Tämän opinnäytetyön tavoitteena oli toteuttaa toimiva kehitysympäristö, jossa kehitettävälle testisovellukselle voitiin suorittaa käyttöjärjestelmän automaatiotestausta Robot Framework -ohjelmistokehyksellä. Lisäksi otettiin tavoitteeksi suorittaa automaatiotestaus osana jatkuvaa integraatiota ja - toimitusta Azure DevOps-ympäristössä.

Teoriaosiossa käytiin läpi ohjelmistotestauksen periaatteita ja osa-alueita. Osiossa tehtiin vertailuja, miten aikaisemmat ohjelmistokehitysmallit, kuten vesiputousmalli, eroavat nykyaikaisista ketteristä menetelmistä, kuten Scrum-mallista. Lukijalle myös esiteltiin ohjelmistotestauksen merkitystä ohjelmistokehityksen aikana ja minkä tyyppisiä testejä ohjelmistokehityksen aikana voidaan suorittaa.

Teoriassa käytiin myös läpi DevOps-kehitysmallin periaatteita ja miten jatkuvaa integraatiota ja jatkuvaa toimitusta suoritetaan ohjelmistokehityksen aikana. Seuraavaksi esiteltiin Robot Framework -ohjelmistokehystä, sen yhteydessä käytettyjä kirjastoja sekä Docker-ohjelmiston arkkitehtuuria ja käsitteitä.

Käytännön osiossa kehitettiin Vue.js-ohjelmistokehyksellä web-pohjainen testisovellus, jolla mallinnettiin tyypillistä SPA-nettisivua. Sovelluksen käyttöliittymän automaatiotestausta varten kehitettiin Robot Frameworkilla testitapauksia hyödyntäen Page Object Model -ohjelmistomallia, jotta testitapausten päivittäminen ja uusien kirjoittamien olisi kehittäjille ja testaajille mahdollisimman helppoa.

Kun testiohjelmisto ja testitapauksen oli kirjoitettu, lisättiin ympäristöön mahdollisuus ajaa testiympäristöä ja testejä konteissa hyödyntäen Docker-ohjelmistoa. Versionhallintaa varten ohjelmiston lähdekoodin vietiin versionhallintaan Azure DevOps -palveluun, jossa siihen lisättiin CI-/CD-putki. Azure DevOps -palvelu määriteltiin siten, että testisovellukselle suoritettiin automaatiotestit osana CI-/CD-putkea.

Tämän jälkeen testisovellukselle luotiin Continuous Deployment -putki, jonka avulla testisovelluksesta tehty Docker-image vietiin automaattisesti Azure Container Registry -palveluun, josta se jaettiin tuotantokäyttöön webpalvelussa.

Lähteet

1 Bisht S. Robot Framework Test Automation [Internet]. Olton: Packt Publishing, Limited; 2013. Saatavilla: http://ebookcentral.proquest.com/lib/kajaani-ebooks/detail.action?docID=1532018

2 Zaal, Stefano, Demiliani, Amit, Malik. Azure DevOps Explained [Internet]. www.packtpub.com: Packt publishing; 2020 [viitattu 23. 3. 2021]. Saatavilla: https://subscription.packtpub.com/book/cloud_and_networking/9781800563513

3 Kasurinen JP. Ohjelmistotestauksen käsikirja [Internet]. Jyväskylä: Docendo; 2014. Saatavilla: https://www.ellibslibrary.com/

4 Beck K, Beedle M, Bennekum A van, Cockburn A, Cunningham W, Fowler M. Manifesto for Agile Software Development - Suomennos [Internet]. 2001 [viitattu 31. 7. 2021]. Saatavilla: https://agilemanifesto.org/iso/fi/manifesto.html

5 Viscardi, Stacia. The Professional ScrumMaster's Handbook. Packt publishing website: Packt; 2013.

6 Myers GJ, Sandler C, Badgett T. The Art of Software Testing [Internet]. Hoboken: John Wiley & Sons, Incorporated; 2011. Saatavilla: http://ebookcentral.proquest.com/lib/kajaani-ebooks/detail.action?docID=697721

7 Gray Box Testing [Internet]. 2012 [viitattu 11. 8. 2021]. Saatavilla: https://softwaretestingfundamentals.com/gray-box-testing/

8 Krief, Mikael. Learning DevOps [Internet]. www.packtpub.com: Packt publishing; 2019 [viitattu 27.4.2021]. Saatavilla: https://subscription.packtpub.com/book/cloud_and_networking/9781838642730

9 Kim G, Debois P, Willis J, Humble J, Allspaw J. The DevOps handbook : how to create world-class agility, reliability, & security in technology organizations. 2016.

10 Fowler, Martin. Continuous Integration [Internet]. 2006 [viitattu 1. 5. 2021]. Saatavilla: https://martinfowler.com/articles/continuousIntegration.html

11 Robot Framework - Home Page [Internet]. [viitattu 6. 4. 2020]. Saatavilla: https://robotframework.org/

12 Eliga Oy - Experience [Internet]. [viitattu 13. 4. 2020]. Saatavilla: http://eliga.fi/experience.html

13 Laukkanen, Pekka. Data-Driven and Keyword-Driven Test Automation Frameworks . Helsinki University of Technology; 2006.

14 Robot Framework User Guide - Version 3.2.2 [Internet]. [viitattu 10. 3. 2021]. Saatavilla: https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html

15 Selenium documentation [Internet]. [viitattu 6. 4. 2020]. Saatavilla: https://www.selenium.dev/documentation/en/

16 Poulton, Nigel. Docker Deep Dive [Internet]. Packt publishing; 2020 [viitattu 21. 3. 2021]. Saatavilla: https://subscription.packtpub.com/book/cloud_and_networking/9781800565135

17 Azure DevOps Services | Microsoft Azure [Internet]. [viitattu 19. 11. 2021]. Saatavilla: https://azure.microsoft.com/en-us/services/devops/

18 CI/CD for Azure Web Apps - Azure Solution Ideas [Internet]. [viitattu 19. 11. 2021]. Saatavilla: https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/azure-devops-continuous-integration-and-continuous-deployment-for-azure-web-apps

19 Managed container registries - Azure Container Registry [Internet]. [viitattu 19. 11. 2021]. Saatavilla: https://docs.microsoft.com/en-us/azure/container-registry/container-registry-intro

20 Visual Studio Code - Code Editing. Redefined [Internet]. [viitattu 18. 11. 2021]. Saatavilla: https://code.visualstudio.com/

21 Node.js [Internet]. [viitattu 18. 11. 2021]. Saatavilla: https://nodejs.org/en/

22 About npm | npm Docs [Internet]. [viitattu 18. 11. 2021]. Saatavilla: https://docs.npmjs.com/about-npm

23 DataDriver Library [Internet]. [viitattu 18. 11. 2021]. Saatavilla: https://robocorp.com/docs/libraries/3rd-party-libraries/datadriver-library

24 Page object models [Internet]. [viitattu 11. 11. 2021]. Saatavilla: https://www.selenium.dev/documentation/guidelines/page_object_models/

25 What is NGINX? [Internet]. [viitattu 18. 11. 2021]. Saatavilla: https://www.nginx.com/resources/glossary/nginx/

26 Overview of Docker Compose [Internet]. 2021 [viitattu 18. 11. 2021]. Saatavilla: https://docs.docker.com/compose/

27 What is Azure Pipelines? - Azure Pipelines [Internet]. [viitattu 18. 11. 2021]. Saatavilla: https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines

28 web app for containers [Internet]. [viitattu 19. 11. 2021]. Saatavilla: https://wedoazure.ie/tag/web-app-for-containers/