Haaga-Helia
ammattikorkeakoulu Oy

Automaatiotestausprosessin kehittäminen

Pekka Pönkänen

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2017

Tekijä(t)
Pekka Pönkänen

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma (HETI)

Raportin/Opinnäytetyön nimi
Automaatiotestausprosessin kehittäminen

Sivu- ja liitesivumäärä
37 + 3

FatAmigos on kolmen hengen startup-yritys, jonka tavoitteena on löytää kohdennettuja tapahtumia kuluttajille. FatAmigos tilasi opinnäytetyön, sillä yritys tarvitsi testiautomaatiota kehityksen tueksi. Opinnäytetyön suoritettava toteutus kohdistuu FatAmigosin automaatiotestausprosessin rakentamiseen. Prosessissa suoritetaan automatisoitu testi, joka liitetään osaksi jatkuvaa integraatiota (CI). Testitapauksessa syötetään FatAmigosin aloitussivulla nimi, sähköposti ja painetaan lähetä painiketta. Ohjelmistokehityksessä jatkuvan integraation ja testauksen rooli on keskeisessä asemassa, koska virheiden löytäminen aikaisessa vaiheessa on tärkeää.

Tietoperustan ensimmäisessä pääluvussa käsitellään ohjelmistotestausta, testiautomaatiota ja ohjelmistokehityksen eri malleja testaamisen näkökulmasta. Luvussa tarkastellaan, kuinka testaus määritellään, mikä on testiautomaation tarkoitus sekä tutkitaan erilaisia ohjelmistokehitysmalleja. Ohjelmistotestauksessa käydään läpi testaamisen roolia ja tavoitteita. Ohjelmistomalleihin on valittu kaksi yleistä toimintamallia: Scrum ja Vesiputous.

Toisessa pääluvussa esitellään testiautomaatiossa hyödynnettäviä työkaluja. Näitä työkaluja voidaan käyttää esimerkiksi: hyväksymisvetoiseen-, käyttöliittymä-, suorituskyky- ja puhelimien testaukseen.

Kolmannessa luvussa selvitetään jatkuvaa integraatiota, versionhallintaa sekä lähdekoodia. Jatkuva integraatio on osa ohjelmistokehitysmallia, jolla parannetaan työskentelyä ohjelmistokehityksessä. Versionhallinnan ja lähdekoodin perusteet ja käsitteet käydään läpi kappaleessa, sekä esitellään jatkuvan integraation työkalu TeamCity ja versionhallintatyökalu GitLab.

Neljännessä luvussa alustetaan projektisuunnitelma, tavoitteet ja toteutus. Luvussa käsitellään työkalut ja perustellaan valinnat. Toteutuksessa kerrotaan, kuinka FatAmigosin infrastruktuuri on rakennettu. Versionhallinnan käyttämisen alustavat toiminnot avataan ja aloitetaan testitapauksien laatiminen. Testitapaus rakennetaan Robot Frameworkin avulla. Testitapaus liitetään CI-putkeen sekä käsitellään GitLab-työkalun konfiguraatiotiedostossa. Lopuksi yhdistetään testitapaus kehitysputkeen ja ajetaan testi.

Opinnäytetyön testitulokset ovat positiiviset. Robot Framework -raporteista voidaan päätellä onnistunut testitapaus. Suunnitelulla asetelmalla varmistutaan siitä, että aloitussivun komponentit toimivat. GitLab ja Robot Framework olivat onnistuneita valintoja FatAmigosin kehityksen kannalta, sillä toimivuus oli varmaa ja konfigurointi suoraviivaista.

Asiasanat
Testaus, testiautomaatio, jatkuva integraatio, versionhallinta

Sisällys

1 Johdanto

Jatkuva integraatio (Continuous Integration) on ohjelmistokehityskäytäntö joka mahdollistaa ohjelmiston kehittämisen yhdessä ohjelmistoa kehittävien kanssa. Jatkuvalla integraatiolla pystytään hallitsemaan kehitettävän ohjelmiston testausta, päivittäisten muutosten päivittämistä, ohjelmiston käyttöönottoa, versionhallintaa, havaitsemaan virhetilanteet sekä jakamaan tietoa ihmisten välillä. Opinnäytetyön tavoitteena on selvittää:

- Mitä on ohjelmistotestaus ja testiautomaatio?
- Mitkä ovat testauksen eri tasot?
- Miten testityökalu saadaan yhdistettyä osaksi jatkuvaa integraatiota?

Ohjelmistotestauksesta on tullut yhä tärkeämpää tekniikan kehittyessä sekä samalla automaatio on helpottanut ohjelmistokehitystä. Tämän työn teoriaosuudessa käydään läpi ohjelmistotestauksen perusteita ja testauksen eri tasoja. Testausta pystyy suorittamaan kaikissa osuuksissa ohjelmistokehityksen aikana. Ohjelmistoja on paljon erityyppisiä ja niiden testaukseen erilaisia malleja. Esimerkiksi ohjelmistot, joissa käsitellään henkilötietoja, vaativat tietoturvatestausta tai järjestelmät joissa on paljon käyttäjiä, tarvitsevat skaalautuvuustestausta. Testiautomaatiolla saadaan jatkuvassa integraatiossa nopeasti tuloksia päivittäisistä ohjelmistojen koostamisista ja havaitaan virhetilanteet nopeasti.

Työni toteutus suoritetaan pienelle startup-yritys FatAmigosille, jonka ensisijainen tarkoitus on löytää käyttäjälle kohdennettuja tapahtumia. FatAmigosin sovelluskehitys on vasta alkuvaiheessa, joten opinnäytetyössä luodaan tarvittava ympäristö jatkuvalle integraatiolle sekä testiautomaatiota web-sovellusta vasten. Tällä asetelmalla FatAmigos pääsee kehittämään sivustoa sekä tulevaa asiakassovellusta. Työssä käsitellään testiautomaatiotyökalun liittämistä jatkuvaan integraatioon, testitapauksien luomista sekä testituloksien läpikäyntiä.

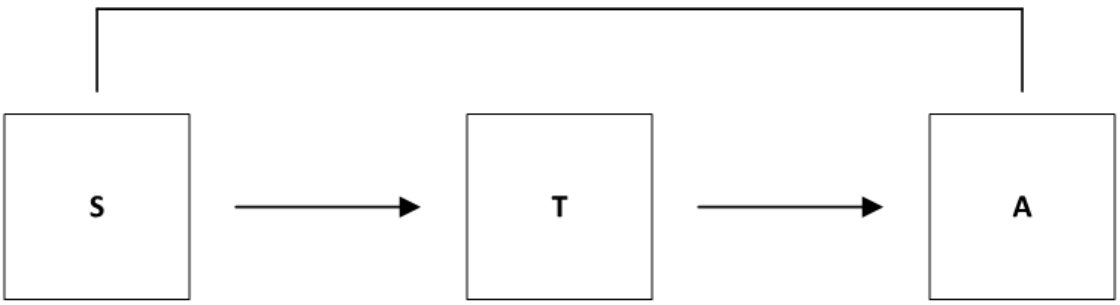
Työskentelen suuryrityksessä ohjelmistokehityksen parissa ohjelmistotestaajana. Työni sisältää manuaalitestausta sekä testiautomatisointia erilaisilla työkaluilla CI-ympäristöissä. Kiinnostukseni testiautomaatiota sekä CI-ympäristöjä kohtaan on tullut työni kautta ja siksi valitsen ne tämän opinnäytetyön aiheeksi. Opinnäytetyö on toiminnallinen. Se sisältää testaukseen, automaatioon, testauksen eri tasoihin ja työkaluihin liittyvän teoriaosuuden sekä

suoritettavan käytännön testaustyön osuuden. Käytännön osuudessa hyödynnetään jatkuvan integraation ja testiautomaation työkaluja. Opinnäytetyö on rajattu ohjelmistotestaukseen ja jatkuvaan integraatioon, joten lukijalta vaaditaan yleistä tietämystä ohjelmistokehityksestä ja siihen liittyvästä termistöstä.

2 Testaus ohjelmistotuotannossa

Ohjelmistotuotanto (software engineering) tarkoittaa tietokoneohjelmistojen rakentamista, yleisten tekniikoiden, työkalujen ja menetelmien periaatteita. Ohjelmisto–nimityksellä tarkoitetaan tietokoneohjelmaa ja siihen liittyviä dokumentaatioita. Järjestelmällä taas yhdistetään ohjelmiston ja laitteiston kokonaisuus. (Haikala & Mikkonen 2011, 11.)

Ohjelmistokehityksessä kuvataan asioita prosesseina, eli toisiaan seuraavina vaiheina. Kehitystyö koostuu monesti selkeistä vaiheista, jotka vievät usein aikaa. Prosesseja seuraamalla ja tarkastelemalla pystytään toimimaan järjestelmällisesti ja ottamaan huomioon ne asiat, jotka kussakin vaiheessa olisi syytä tehdä ennen kuin siirrytään seuraavaan vaiheeseen. Kaikenlainen kehittämistyö voidaan jäsentää yksinkertaiseksi muutostyön prosessiksi (kuvio 1). Prosessi alkaa suunnitteluvaiheella (S). Tämä vaihe sisältää suunnitelman siitä, miten päästään toivottuun tulokseen, työkalujen valinnan, taustatyöt ympäristöineen ja myös sidosryhmiin tutustumisen. Tämä vaihe suorittaa suunnitteluvaiheen (S). Seuraava vaihe on toteuttaminen, joka näkyy kuviossa yksi kirjaimella (T). Prosessin viimeinen vaihe on arviointi (A). (Ojasalo, Moilanen & Ritalahti 2015, 22-23.)



Kuvio 1. Muutostyön prosessi (Ojasalo, Moilanen & Ritalahti 2015, 23)

2.1 Ohjelmistotestaus

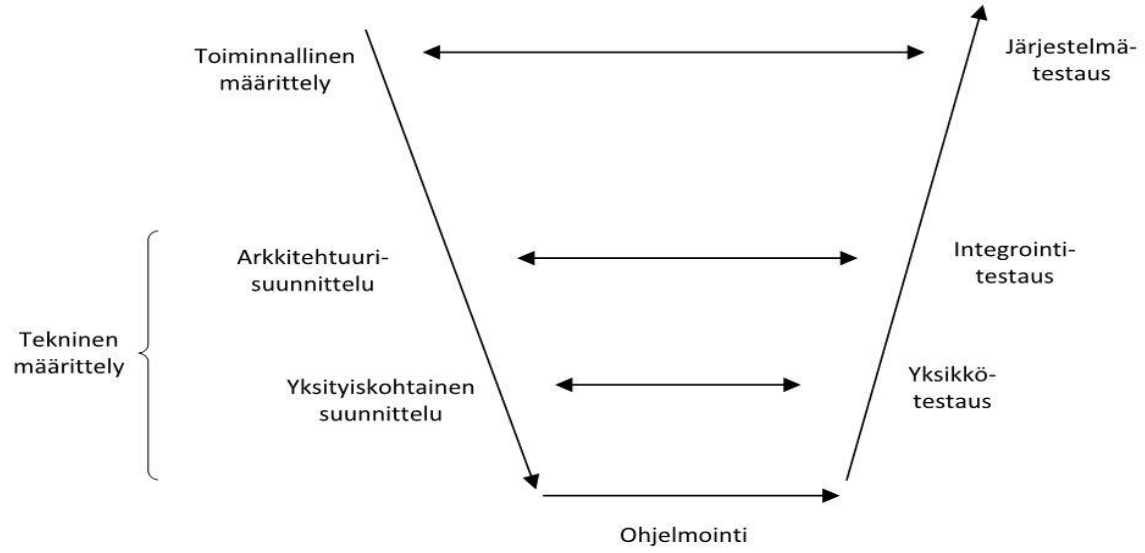
Ohjelmistotestauksella tarkoitetaan työtä, jolla halutaan varmistaa, että toteutettavasta ohjelmistotuotteesta tulee toivotun kaltainen ja että sen kaikki ominaisuudet toimivat, kuten on tarkoitus. Testauksella varmistetaan, että kehitys on menossa oikeaan suuntaan ja että tuotetta tehdään oikein. Testauksen tavoitteena on tarkistaa, mitä on saatu aikaiseksi sekä tunnistaa ne kohdat, joissa tuotos poikkeaa suunnitelmista. (Kasurinen 2013, 10.)

Testaus on yksi kokonaisuus (kuvio 2) ohjelmistokehityksen elinkaaressa. Testauksella kontrolloidaan tuotteen toimivuutta ja laatua. Ohjelmistokehityksen tarkoituksena on löytää ja tunnistaa ohjelmistojen valmistamiseen soveltuvia perusperiaatteita, jotka tunnistetaan toimiviksi ja joiden avulla voidaan esimerkiksi saada aikaan hyvää laatua tai karsia ylimääräisiä kustannuksia. Olennaista on esimerkiksi miettiä, millä tavalla projektityötä tulisi johtaa, millaisia mittareita ohjelmistoprojektin etenemisessä mahdollisesti kannattaa seurata ja mitä kaikkea testaussuunnitelmaa koostaessa tulee huomioida. Nämä ovat kaikki aiheita, jotka kuuluvat ohjelmistotuotannon piiriin. (Kasurinen 2013, 22.)

Testaajan rooli ei ole yksiselitteinen, koska rooleja saattaa olla useita. Testaaja voi keskittyä esimerkiksi erityisesti manuaalitestaukseen tai vaikkapa automaatiotestien ohjelmointiin ja sitä kautta kehitystyöhön. Usein testaaja tekeekin erilaisia asioita, kuten kirjoittaa käyttötapauksia, suunnittelee demoja, ohjelmoi testejä ja dokumentoi löydettyjä ohjelmointivirheitä. Tämän myötä testaajan roolin voi olla hyvinkin erilainen eri ohjelmistotalojen välillä. (Kasurinen 2013, 10-11.)

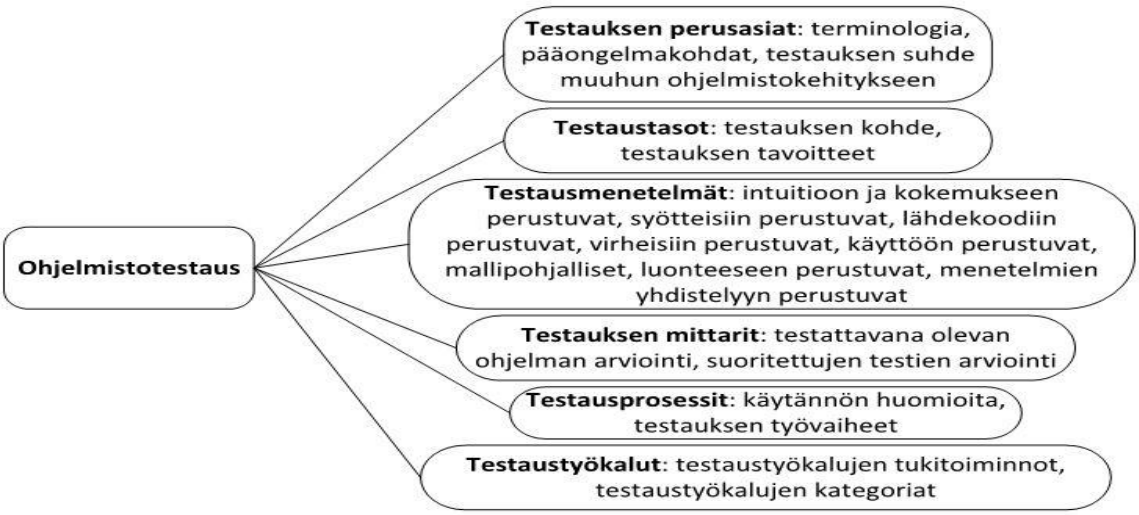
Puhekielessä testauksella tarkoitetaan kaikenlaista kokeilua. Ohjelmistojen testauksen yhteydessä testauksella tarkastetaan jonkin ohjelman osa ja etsitään suunnitelmallisesti virheitä. Tämä on suunnitelmallista testausta, jossa edetään tietyssä järjestyksessä ja keskitytään mahdollisten virheiden löytämiseen. Usein testausta suoritetaan myös umpimähkään ja ilman suunnittelua syöttäen satunnaisia syöttöaineistoja, jolloin virheiden etsintä jää pois. Jos testaajana toimii ohjelman tekijä, tavoitteena saattaa olla ohjelman toimivuus, eikä virheiden löytäminen. (Haikala & Mikkonen 2011, 205.)

Ohjelmistokehityksen ja testauksen suhdetta kuvataan V-mallilla (kuvio 2), jonka mukaan testausta suunnitellaan. Mallissa ilmenee kolme erilaista testaustasoa: yksikkötestaus (unit testing), integrointitestaus (integration testing) ja järjestelmätestaus (system testing). (Haikala & Mikkonen 2011, 206.)



Kuvio 2. Testauksen V-malli (Haikala & Mikkonen 2011, 207)

Ohjelmistotestaukseen (kuvio 3) liittyy useita työvaiheita, joita ovat esimerkiksi suunnittelu, testaussuunnitelma, testitapauksien kirjoittaminen, testiympäristöjen luominen, testien suorittaminen ja tuloksien tarkastelu. Tyypillisesti suurin osa ajasta ohjelmistokehitysprojektissa menee virheiden korjaamiseen (debugging) ja jäljitykseen, joten testaukseen suositellaan panostamaan parhaalla mahdollisella tavalla. (Haikala & Mikkonen 2011, 205.)



Kuvio 3. SWEBOK-mallin testaamisen osa-alueet (Kasurinen 2011, 46)

Testaukselle on syntynyt 7 periaatetta ”Seven Testing Principles”, jotka ovat Grahamin, van Veerendaalin, Evansin ja Blackin (2007, 22) ne tarjoavat testaukselle yleisiä ohjeita.

Periaate 1: Testaus osoittaa, että viat ovat läsnä: testauksella voidaan osoittaa havaitut ongelmat ja vähentää virheiden todennäköisyyttä.

Periaate 2: Täydellinen testaus on mahdotonta: kaiken testaaminen, sisältäen kaikki kombinaatiot syötteistä ja edellytyksistä, ei ole toteutettavissa. Käytetään perusteellista testausta, painopisteenä ohjelmiston testauspyrkimys ja tavoitteet.

Periaate 3: Aikainen testaaminen: testaus pitäisi aloittaa mahdollisimman nopeasti. Testauksen pitää keskittyä määritettyihin tavoitteisiin.

Periaate 4: Vikojen kasaantuminen: pieni osa moduuleista sisältää suurimman osan virheistä, kun testataan ohjelmistoa. Niistä ilmenee useimmat toimintahäiriöt.

Periaate 5: Hyönteismyrkkyparadoksi: jos samat testit toistuvat uudestaan ja uudestaan, lopulta testitapaukset eivät löydä uusia vikoja. Tämän paradoksin voittamiseksi testitapauksia pitäisi säännöllisesti tarkastella sekä luoda ja päivittää testitapauksia. Tämän myötä järjestelmästä mahdollisesti löytää enemmän vikoja.

Periaate 6: Testaus on tilanneriippuvaista: testaaminen suoritetaan eri tavalla, koska konteksti muuttuu järjestelmien myötä. Esimerkiksi turvallisuuden kannalta kriittinen ohjelmisto testataan eri tavalla kuin verkkokauppasivusto.

Periaate 7: Virheettömyyden harhaluulo: vikojen etsiminen ja korjaaminen ei auta, jos järjestelmä on käyttökelvoton eikä vastaa käyttäjien tarpeisiin ja odotuksiin.

2.2 Testiautomaatio

Testiautomaatio on testaustoiminnan muoto, jossa luodaan työkalut ja ympäristö ohjelman automaattista testaamista varten. Automaation tavoitteena on ottaa joukko toistuvasti tapahtuvia testitapauksia ja rakentaa ympäristö niiden nopeaa testaamista varten. Pohjim-

millaan ajatuksena on ollut, että testaajat voitaisiin vapauttaa muihin tehtäviin. Jos ohjelmistosta tulee joka päivä uusi versio (daily build), testaajien ei ole järkevää tehdä perustestejä joka päivä, kun ne pystytään tekemään testiautomaatiolla. (Kasurinen 2013, 76.)

Testiautomaatiolla pystyy suorittamaan useita erilaisia testauksen tasoja, joita esittelen opinnäytetyössä myöhemmin. Otollisimpia kohteita testiautomaatiolle ovat yksikkö-, rajapinta-, regressio- ja funktionaaliset testit. Kokonaista ohjelmistoa on hyvin vaikeaa, tai jopa mahdotonta kokonaan testata automaattisesti, koska tarvittavat työkalut eivät vielä tähän pysty ja kulut ovat korkeita. Ohjelmistoprojektin alussa olisi hyvä päättää, mitkä asiat halutaan automatisoida. Mistä saisi eniten hyötyä projektin alkuvaiheessa? (Kasurinen 2013, 76.)

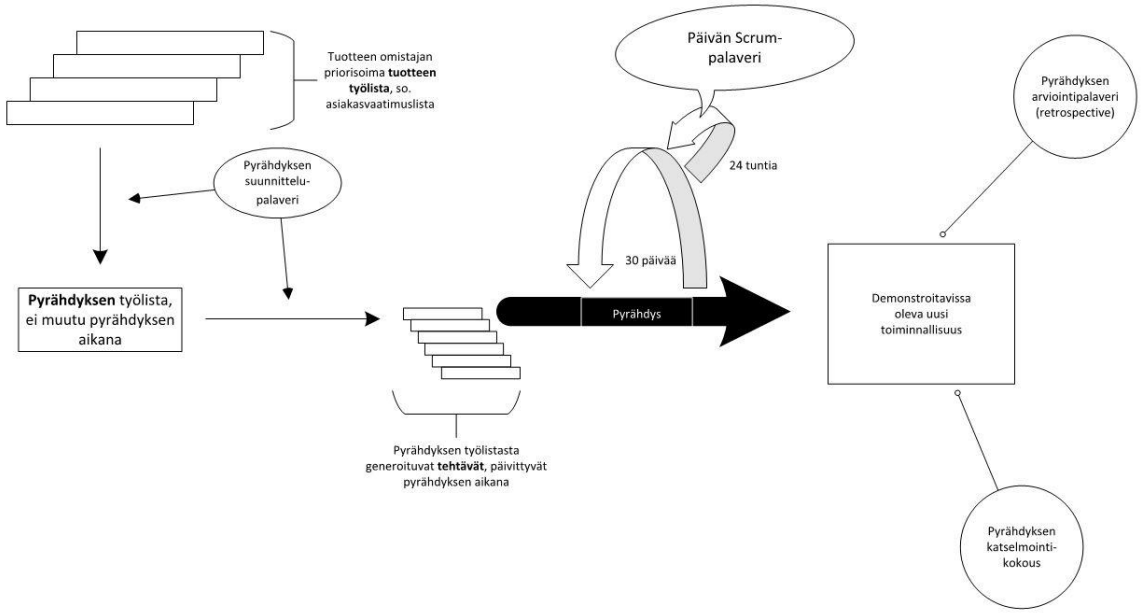
Duvall, Matyas ja Glover (2007, 15) kertovat, että ilman automaatiota projektin sidosryhmien ja kehittäjien on vaikea olla varmoja siitä, onko järjestelmä vakaa. Useimmat kehittäjät käyttävät projekteissa testityökaluja, kuten JUnit tai muita xUnit työkaluja, ajaessaan testejä. Automaatiota pystytään ajamaan monilla eri tavoilla hyödyntämällä jatkuvaa integraatiota (continuous integration) ja nopeuttamaan testausta. Testiautomaatioon voidaan tuoda esimerkiksi kuormitus-, rasitus- tai penetraatiotestausta. Myöhemmissä kappaleissa käydään läpi muita mahdollisia testiautomaatiotasoja.

2.3 Ohjelmistokehityksen eri mallit testaamisen näkökulmasta

Ohjelmistokehitysprojektit ovat hyvin erilaisia ja siksi on myös kehitetty erilaisia menetelmiä hallita niitä. Kuten aikaisemmin mainitsin, testaus on osa ohjelmistokehitystä, ja täten läsnä kaikissa menetelmissä. Eroavaisuuksia ilmenee testauksessa ja sen ajoittamisessa ohjelmistoprojekteissa.

Scrum on ohjelmistotuotannon menetelmä, joka on ketterä ja yksinkertainen. Luonteeltaan se on helposti hallittava ja siksi tullut suosituksi yritysmaailmassa. Scrumin vahvuudet tulevat esille projekteissa, joissa ryhmä ihmisiä tekee tiiviisti töitä keskenään ja pystyy kommunikoimaan ja vaihtamaan tietoa vapaasti. Scrum-mallissa ryhmän sisäinen kommunikaatio on suuressa roolissa. Scrum menetelmässä (kuvio 4) ohjelmistoa kehitetään muutaman viikon sykleissä, joissa on tarkoituksena saada rakennettua jokin tietty toiminnallisuus tai ominaisuus sekä testata sen toimivuus. Näitä syklejä kutsutaan nimellä sprintti (sprint). Työpäivät aloitetaan Scrum-mallissa palaverilla, jossa käydään läpi päivän sisällä

tehtävät asiat sekä kerrotaan ryhmälle yleisesti meneillään olevista tehtävistä ja mahdollisista ongelmista. (Kasurinen 2011, 28.)

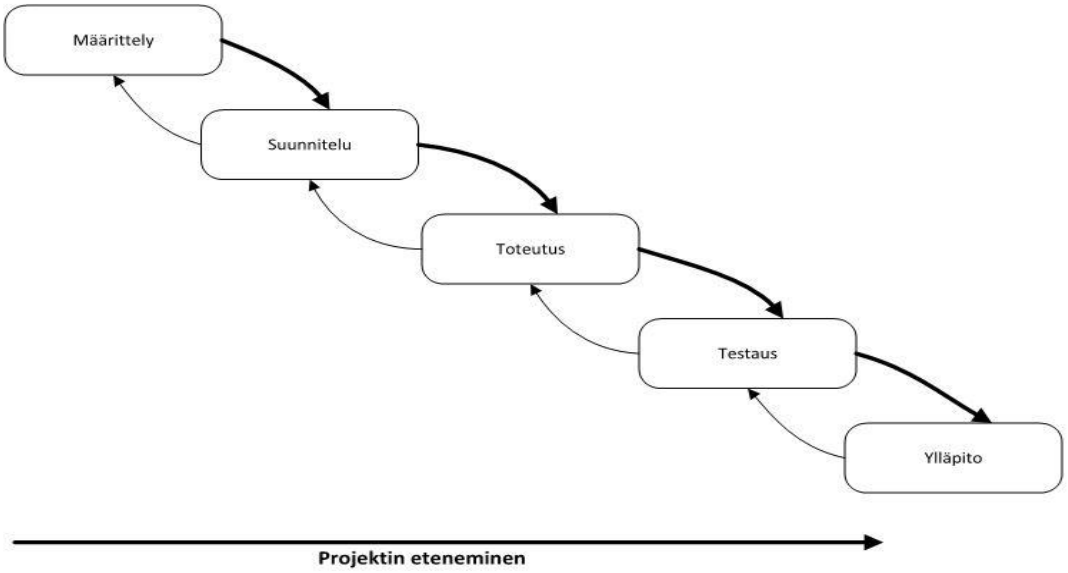


Kuvio 4. Scrum-prosessi (Haikala & Mikkonen 2011, 48)

Scrum on alun perin Japanista ja siitä tehtiin ensimmäisiä julkaisuja Harward Business Review lehdessä vuonna 1986. Tuolloin esitettiin Scrumin periaatteita. Scrumia pidetään ketteränä menetelmänä, mutta myös ketteryyden kankeuttajana. Scrumissa ei oteta kantaa projektin työkaluihin tai kehitysmenetelmiin ja täten Scrum on enemmän projektin iteraatio-vaiheeseen organisoiva tapa, kuin projektimenetelmä. Iteraatio Scrumissa tarkoittaa aikaväliä, jonka aikana kehitystä tapahtuu. (Agile Alliance). Scrumilla ei yksinään pystytä tekemään onnistumisia, vaan sen lisäksi tarvitaan muita projektihallinnan tapoja tueksi. (Haikala & Mikkonen 2011, 47.)

Vesiputousmallissa (kuvio 5) tyyli on erilainen kuin Scrum-mallissa. Perinteisessä vesiputousmallissa edetään seuraavasti: määrittely, suunnittelu, toteutus, testaus ja ylläpito. Testausta suoritetaan vesiputousmallissa vasta ennen käyttöönottoa, hieman ennen projektin päättymistä. Vesiputousmalli on ongelmallinen siinä mielessä, että vaatimuksia ei voi tietää alussa aivan tarkkaan ja projektin aikataulua on siksi vaikea määrittää. Lyhyet projektit pystyvät toimimaan vesiputousmallilla, mutta useamman kuukauden tai vuoden

mittaisissa projekteissa voi olla vaarana, että ei tiedetä milloin projekti on valmis. Kasurinen kertoo, että vesiputousmalli on silti suosittu ja tunnettu ohjelmistokehityksessä. Vesiputousmalli on yksinkertainen ja selkeä malli. (Kasurinen 2011, 24.)



Kuvio 5. Perinteinen vesiputousmalli (Kasurinen 2011, 23)

3 Testauksen eri tasot

Tässä kappaleessa käydään läpi muutamia eri testitasoja ja niiden taustoja. Testitasoja on monia erilaisia, ja niitä hyödynnetään eri vaiheissa ohjelmistokehitystä. Esitettävät testaustasot ovat suorituskykytestaus, rasitustestaus, kuormitustestaus, skaalautuvuustestaus, savutestaus, yksikkötestaus, integrointitestaus, järjestelmätestaus ja ohjelmistorajapintatestaus.

Suorituskykytestaus on rasitustestauksen kevyempi malli, jossa tavoitteena on osoittaa järjestelmän suoriutuvan sille kohdistetusta kuormituksesta. Tämä voi tarkoittaa esimerkiksi sitä, että järjestelmän maksimikäyttäjämäärät, vasteajat tai käsittelykapasiteetti ovat sitä luokkaa, mitä vaatimusmäärittelyssä on kuvattu. (Kasurinen 2013, 72.) Suorituskykytestauksen ala-lajeiksi luokitellaan rasitustestaus, kuormitustestaus ja skaalautuvuustestaus.

Rasitustestauksella pyritään varmistamaan järjestelmän tai sen yksittäisen komponentin kykyä käsitellä suuria kuormia tai toimivuutta tilanteissa, joissa kapasiteetti on jo valmiiksi ohut tai vähentynyt. Rasitustestausta tehtäessä järjestelmän suorituskyvyn pitäisi laskea tasaisesti ja ennakoivasti ilman häiriöitä, kun rasitusta lisätään. Testauksen aikana pitäisi järjestelmästä testata toiminnalle tärkeitä prosesseja mahdollisten vikojen tai muiden epäjohdonmukaisuuksien löytämiseksi. Rasitustestauksen tavoite on löytää järjestelmän ääirajat, jolloin voidaan korjata heikkoudet. Testauksen tuloksena voidaan parantaa järjestelmän toimivuutta, esimerkiksi lisäämällä muistia, tietokannan kokoa tai prosessoritehoja. (ISTQB 2013, 33.)

Kuormitustestaus (*load testing*) on yksi testaustoiminnan muoto, jolla laitetaan ohjelmisto kuormitukselle, esimerkiksi luomalla virtuaalikäyttäjiä (*virtual users*) ja tekemällä niillä testejä. Esimerkki kuormitustestauksesta voisi olla verkkokauppa, joka on suunniteltu palvelemaan 250 käyttäjää yhtäaikaisesti. Testauksessa luotaisiin 250 virtuaalikäyttäjää tekemään eri tyyppisiä toimintoja, kuten selaamaan tuotekatalogia, tekemään tilauksia, tekemään muutoksia tilauksiin, ostamaan tuotteita, muuttamaan asetuksia ja niin edelleen. Kuormitustestausta voi tehdä myös esimerkiksi kuvankäsittelyohjelmalle, laittamalla sen avaamaan jatkuvasti raskaita tiedostoja tai tekstinkäsittelyohjelmalle pistämällä sen avaamaan isoja massoja dataa. (Kasurinen 2013, 71.)

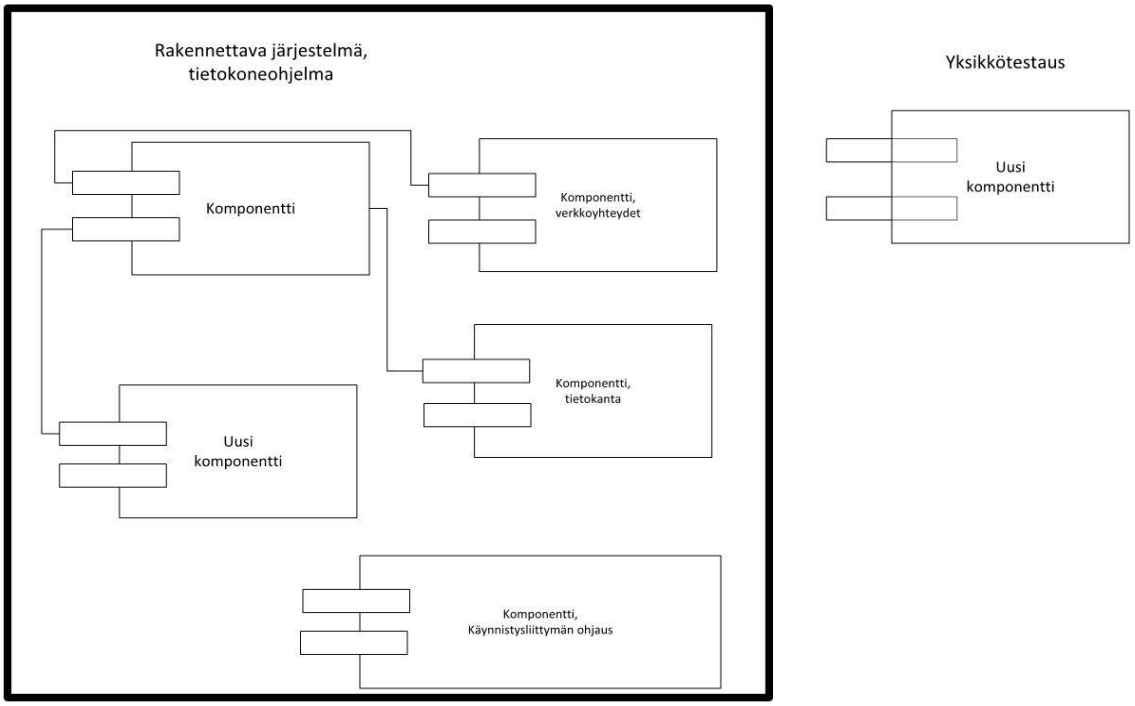
Skaalautuvuustestaus painottuu järjestelmään kohdistuviin tehokkuusvaatimuksiin, jotka saattavat olla isommat kuin senhetkisessä järjestelmässä. Skaalautuvuustesteissä tavoitteena on kasvattaa järjestelmää hetkellisesti, esimerkiksi tehdä enemmän käyttäjiä tai tuoda enemmän aineistoa, että suoritustaso laskisi järjestelmässä tai että järjestelmä ei kestäisi tätä ja kaatuisi. Testien avulla saadaan raja-arvoja järjestelmälle ja sen myötä voidaan jatkossa reagoida tilanteisiin, jos ilmenee tuotannossa ongelmia. Tuotantoympäristö voidaan muokata sopivaksi esimerkiksi hankkimalla lisää laitteistoa. (ISTQB 2013, 33.)

Savutestaus (smoke test) on termi ohjelmistokehityksessä testaukselle, jolla käydään ohjelmiston perusasiat läpi ja testataan toimivuus. Käytännössä savutesti on tarkistuslista, jolla tarkistetaan ohjelmiston perustoimivuus. Tarkistettavia asioita voisivat olla esimerkiksi:

- Käynnistyykö sovellus?
- Pystyykö käyttäjä kirjautumaan sisälle palveluun?
- Pystyykö käyttäjä tallentamaan syötteitä järjestelmään?
- Toimivatko kaikki etusivun ylälaidan painikkeet? (Kasurinen 2013, 72.)

Kasurinen (2013, 72) mainitsee, että savutestit ovat hyvä indikaattori, jolla saadaan helposti pois perusvirheet, jotka voivat vaikuttaa muihin toiminnallisuuksiin. Testaajat saattavat rutinoitua helposti sovellukseen, ja savutestit ovat yksi tapa pitää perusasiat hallussa. Myös vaatimusmäärittelyssä tai suunnittelussa tapahtuneet unohdukset, voivat tulla ilmi perinteisessä savutestissä. (Kasurinen 2013, 72.)

Yksikkötestaus on yleisimpiä testaustoimenpiteitä ohjelmistokehityksessä ja se samalla on testausmenetelmä. Yksikkötestauksella halutaan selvittää yksittäisen funktion, olion tai moduulin toimintaa (kuvio 6). Lähes poikkeuksetta tämän testauksen tekee ohjelmistokehittäjä itse. Yksikkötestauksen tarkoituksena on varmistaa järjestelmään luodun toiminnon tai muutoksen toimivuus. Yksikkötestissä tarkistetaan, että muutos kääntyy varmasti ja että se toimii funktionaalisesti oikein. Tähän voidaan luoda joukko oliokutsuja ja testata komponentin toimivuus sitä kautta. Komponentin testaaminen heti säästää kuluja myöhemmin, jos komponentissa olevat ongelmat huomataan ennen kuin se yhdistetään laajempaan järjestelmään. (Kasurinen 2013, 51-52.)



Kuvio 6. Uutta komponenttia testataan muista osista erillään. (Kasurinen 2011, 52)

Kasurinen (2011, 52-53) kertoo, että ongelma syntyy yksittäisistä komponenteista, koska komponentti ei pysty tekemään yksin juuri mitään. Komponentit vaikuttavat useampaan komponenttiin kerrallaan, joten testauksessa pitää tehdä testitynkiä (test studs), jotka simuloivat komponenttien keskinäistä liikennettä. Testityngillä halutaan tehdä testaustapaus mahdollisimman aidoksi. Testitapauksia on erilaisia, ja niitä ovat muun muassa: syötettyjen arvojen tyypit, syntaksivirheet, erikoismerkkien käsittely, muistinkättelyä tai näkyvyysrajat. (Kasurinen 2011, 52-53.)

Integrointitestaus (integration testing) on eri osien yhdistämistä, jotta saadaan järjestelmä toimimaan yhtenäisenä kokonaisuutena. Integrointitestauksessa tarkastellaan uuden komponentin lisäämistä aiemmin testattua järjestelmää vasten. Mahdollisesti uusi komponentti sisältää useita kytkentöjä ja tätä varten joudutaan luomaan tynkiä, jotka ovat sijaiskomponentteja, joiden avulla voidaan testata järjestelmää. Integraatiotestauksen tärkein tavoite on käydä läpi järjestelmän kaikki osat ja varmistaa niiden toimivuus yhdessä. Esimerkkinä voisi käyttää moduulien välistä viestintää tai tietokantayhteyksien toimivuutta järjestelmien välillä. (Kasurinen 2013, 54.)

Järjestelmätestauksen tarkoituksena on käydä läpi koko järjestelmän käyttäytyminen. Tehokkaat yksikkö- ja integraatiotestaukset ovat tuottaneet tulosta, joten järjestelmätestaus on hyvä menetelmä tilanteen arviointiin. Järjestelmätestausta pidetään hyvänä indikaattorina arvioitaessa ei-toiminnallisia järjestelmävaatimuksia, kuten nopeutta, tarkkuutta, turvallisuutta tai luotettavuutta. Ulkoiset toimijat, kuten rajapinnat muihin sovelluksiin, laitteisiin tai toimintaympäristöihin tarkistetaan yleensä tällä tasolla. (Bourque & Fairley 2014, 4-5.)

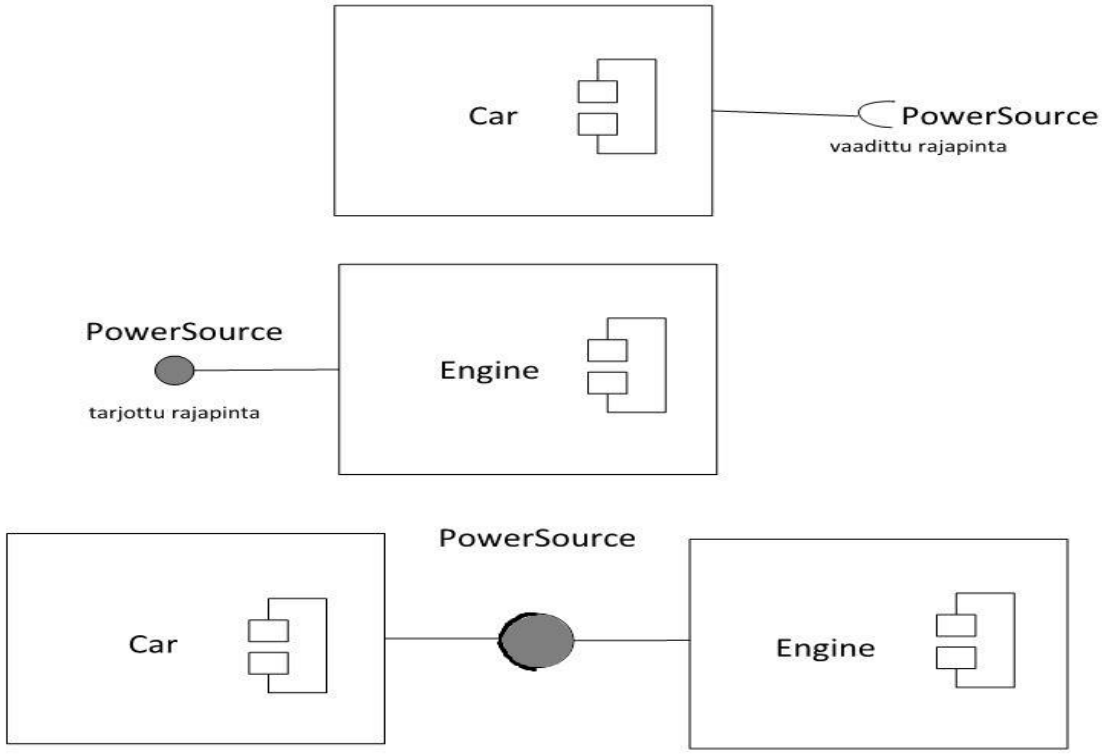
Järjestelmätestauksen suorittajina suositaan kehitystyöstä riippumattomia testaajia. Järjestelmätestaukseen voidaan ottaa mukaan myös kenttätestausta (field testing) tai hyväksymistestausta (acceptance testing, ATDD). Järjestelmätestauksessa käydään läpi kuormitustestaukset, asennustestit (asennuksen läpikäynti), luotettavuustestit (virhetilanteesta selviytyminen) ja käytettävyystestit. (Haikala & Mikkonen 2011, 208.)

Ohjelmistorajapintatestaus (API, Application Programming Interface) mahdollistaa prosessien, ohjelmien ja/tai järjestelmien välisen tiedonkulun. API tarkoittaa ohjelmakoodia, joka mahdollistaa järjestelmien tiedonkulun. APIa käytetään usein asiakas/palvelin-järjestelmissä, joissa yksi tai useampi prosessi tuottaa toiminnallisuuden muita prosesseja varten. (ISTQB 2013, 16.)

API-testaus on tärkeätä, koska yhä useampia järjestelmiä hajautetaan, ja työt puretaan toisille käsittelijöille. API-testaus soveltuu järjestelmiin, jotka koostuvat muista järjestelmistä tai sovelluksista. APIn testaamiseen ei suoranaisesti vaadita erityistyökaluja, koska graafista käyttöliittymää ei usein ole. Niinpä muilla työkaluilla luodaan testiympäristö ja testiaineisto sekä suoritetaan API-kutsu, jolla määritellään lopputulos. (ISTQB 2013, 16.)

Rajapinnasta näkee vähintään, miten palvelua käytetään, toisin sanoen parametrit, palvelun nimen ja niiden tyypit. Tästä kokonaisuudesta käytetään nimitystä kutsumuoto (signature). Käytännössä rajapinnan pitäisi kertoa olennainen tieto, jota käyttäjä tarvitsee palvelun käyttämiseen. Rajapinnat määräävät arkkitehtuurin ominaisuudet ja tavat, joilla komponentit keskustelevat keskenään. Osa suunnittelumalleista ja arkkitehtuurista perustuu rajapintoihin. Ohjelmistokehitykseen sisältyy huolellinen rajapintasuunnittelu, kuten myös ylläpidettävyydelle, joustavuudelle ja testaukselle. Rajapintasuunnittelu alkaa melkein heti, kun arkkitehtuurimalli on määritelty. (Koskimies & Mikkonen 2005, 58.)

Koskiniemi ja Mikkonen kuvaavat UML-mallissa (Unfied Modeling Language) auton ja voimalähteen toimintaa. Auto (car) tarvitsee jonkin voimalähteen, ja tähän hyödynnetään rajapintaa ”PowerSource”. Moottori (engine) pystyy toteuttamaan tämän rajapinnan (tarjottu rajapinta). Kuvion alaosasta (kuvio 7) näkyy, että auton ja moottorin komponentit liitetään rajapinnan avulla toisiinsa. (Koskimies & Mikkonen 2005, 60.)



Kuvio 7. Tarjotut ja vaaditut rajapinnat UML-mallissa (Koskiniemi & Mikkonen 60)

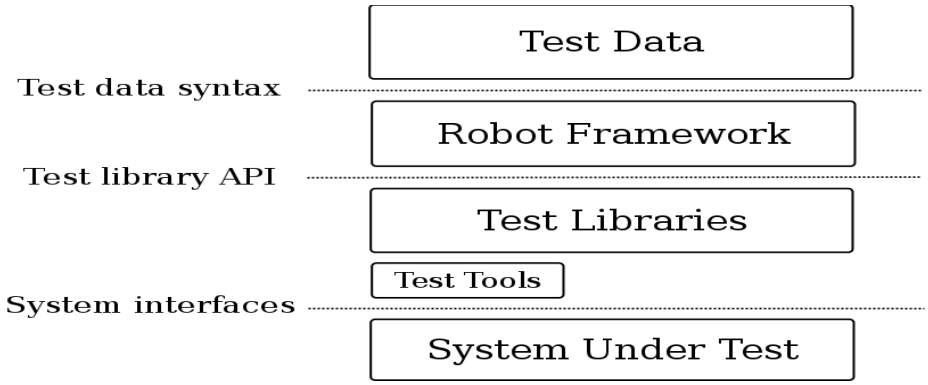
4 Testiautomaatiotyökalut

Tässä kappaleessa käsitellään erilaisia testiautomaatioon liittyviä työkaluja, joista osaa hyödynnetään toteutuksessa itsessään. Erilaisia työkaluja on markkinoilla paljon ja osa niistä on kaupallistettu ja osa on avoimen lähdekoodin sovelluksia.

Regressio- ja julkaisutestauksen suorittamiseksi testauksen pitäisi olla hyvin pitkälle automatisoitua. Automatisointiin voidaan käyttää testaustyökaluja, joita ovat esimerkiksi testipetigeneraattorit, vertailijat, testitapausgeneraattorit ja testikattavuustyökalut. Järjestelmätestauksessa voidaan käyttää toistotyökaluja, nauhoittaa testitapauksia ja syötteitä sekä käyttää niitä myöhemmin uudelleen. (Haikala & Mikkonen 2011, 213.)

4.1 Robot Framework

Robot Framework on alustasta riippumaton avoimen lähdekoodin testiautomaatiotyökalu. Sitä käytetään erityisesti hyväksymistestaukseen ja hyväksymisvetoiseen (ATDD) kehitykseen, mutta myös muille tasoille yksikkötestauksesta ylöspäin (kuvio 8). Robot Frameworkin testitapaukset kirjoitetaan luonnollisella kielellä, hyödyntäen erityyppisten kirjastojen avainsanoja. Robot Frameworkissa on valmiita standardi-kirjastoja sekä ulkopuolisia kirjastoja eri käyttötarkoituksiin. Kirjastoista löytyy tukea esimerkiksi käyttöjärjestelmälle, protokollille HTTP/HTTPS, Androidille, web-testaukseen, iOS:lle ja tietokannoille. Kantava ajatus on, että kaikki sidosryhmät ymmärtävät, mitä ominaisuuksia tai vaatimuksia testataan. Uusia ylemmän tason avainsanoja luodaan samalla syntaksilla tarpeen mukaan. Robot Framework on kehitetty Python ohjelmointikielellä, ja se tukee myös Jython (JVM) ja IronPython (.NET) -viitekehityksiä. Robot Framework on avoimen lähdekoodin viitekehys, jonka takana on Robot Framework Foundation. (Robot Framework 2017.)

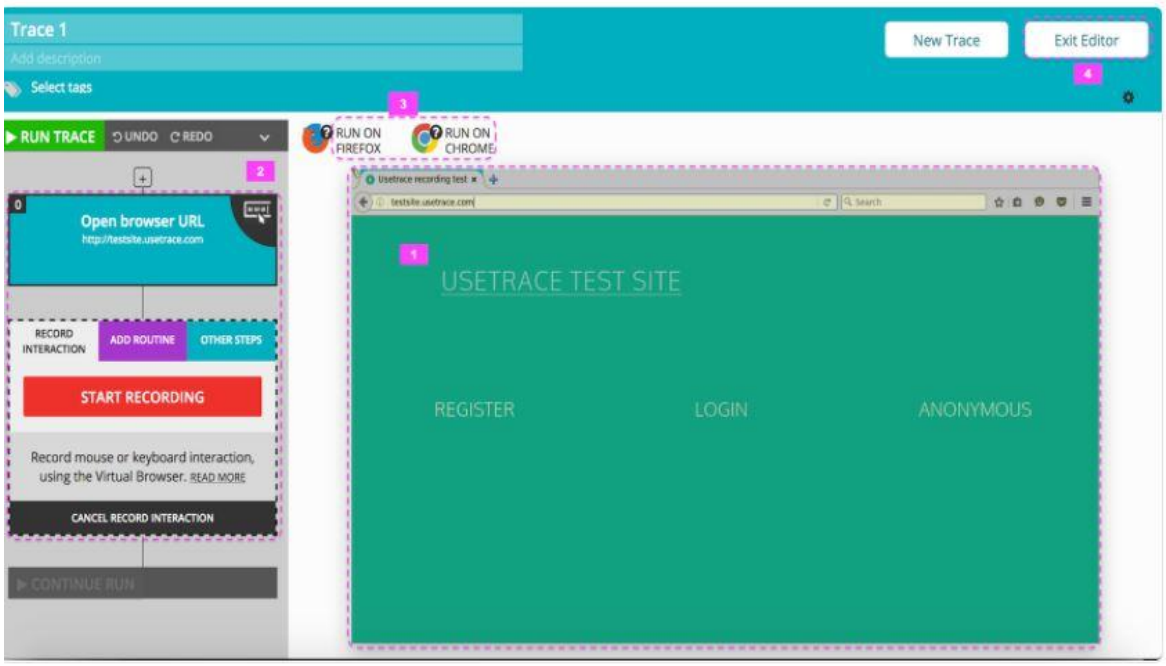


Kuvio 8. Robot Frameworkin arkkitehtuuri (Robot Framework 2017)

4.2 Usetrace

Usetrace-sovelluksen avulla pystytään automatisoimaan käyttöliittymätestausta ja tarkistamaan, että web-sovelluksessa näkyy loppukäyttäjille tarvittavat ominaisuudet ja että ne toimivat odotetusti. Usetrace sisältää regressio-, suorituskyky- sekä kuormitustestauksen. (Usetrace 2017a.)

Usetrace-sovelluksen käyttö ei vaadi ohjelmointia. Automaattitestejä kutsutaan jäljiksi (kuva 1), jotka ovat visuaalisia ja syntyvät käyttäjän poluista, jotka muodostuvat testattavan sovelluksen kautta. Jäljitystä tehdään vuorovaikutteisesti web-sivuston kautta. Jäljet kertovat kaikki testin aikana tehdyt painallukset ja näyttävät ovatko elementit tai tekstit olemassa. Usetrace tallentaa nämä tapahtumat rutiiniksi, joita pystyy käyttämään uudelleen ja näin ollen antamaan ylläpidettävän automaatiotestisarjan sovelluksen käyttöliittymälle. (Usetrace 2017a.)

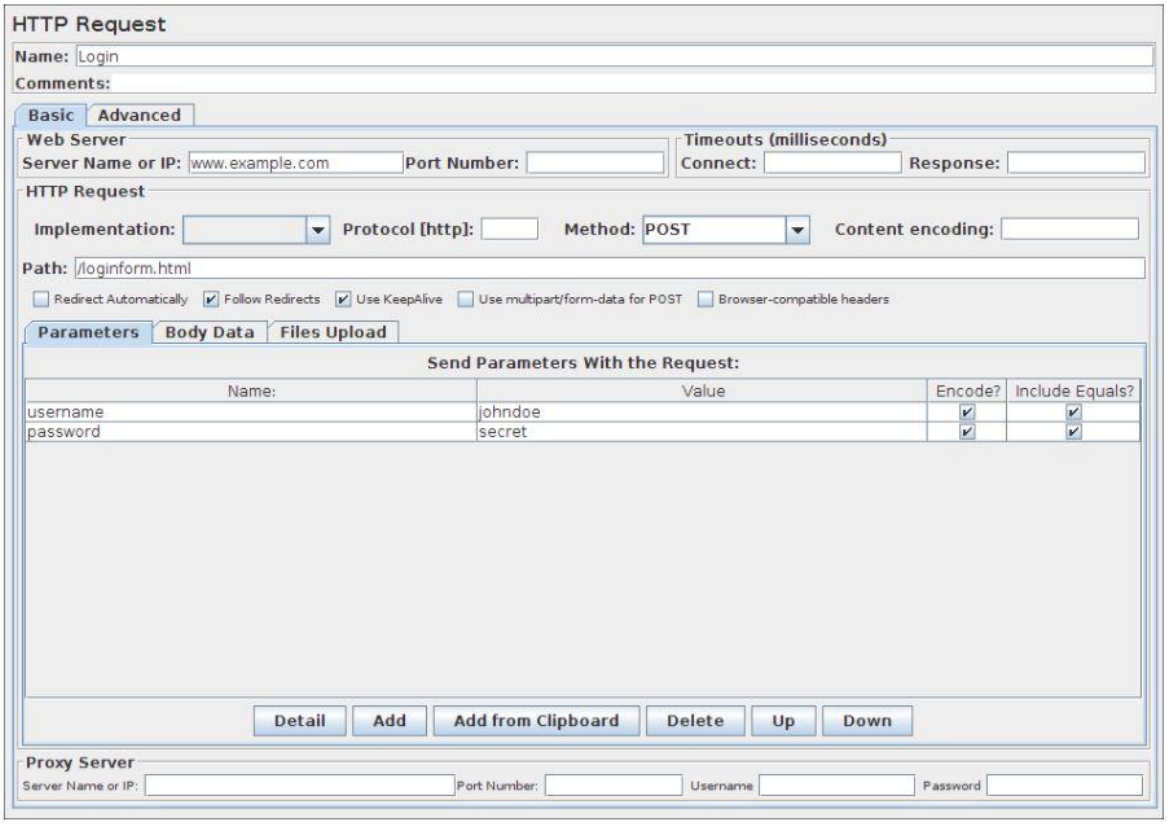


Kuva 1. Usetrace editori (Usetrace 2017b)

4.3 JMeter

Apachen JMeter –työkalua voidaan käyttää suorituskyvyn testaamiseen, staattisiin ja dynaamisiin resursseihin sekä web-palveluihin. JMeterillä pystytään simuloimaan suuria

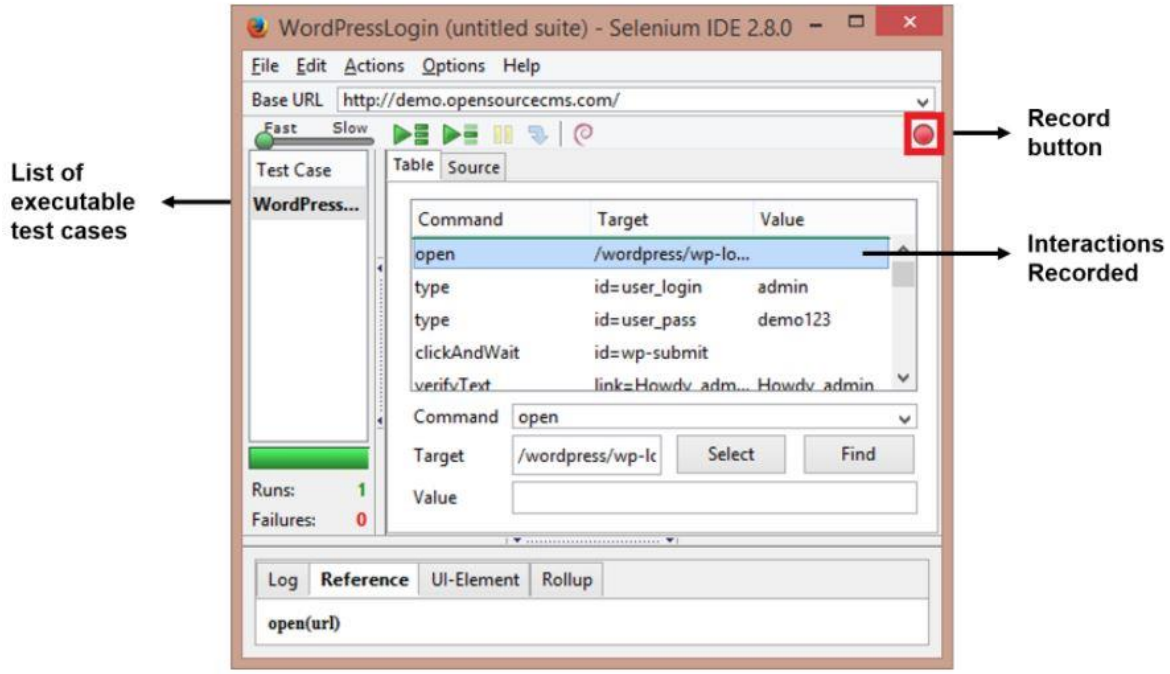
kuormituksia palvelimille ja verkkoihin. JMeter on avoimen lähdekoodin työkalu, joka on toteutettu täysin Javalla. Aluksi JMeteria kaavailtiin web-sovellusten testaamiseen, mutta sitä alettiin käyttää myöhemmin myös muihin testitoimintoihin. JMeterillä pystyy tekemään suoritus- ja kuormitustestausta esimerkiksi HTTP:n (hypertext transfer protocol) protokollaa hyödyntäen (kuva 2) tai useiden ohjelmointikielen kanssa. JMeter ei ole selain vaan toimii protokolla-tasolla. (The Apache Software Foundation 2017a.)



Kuva 2. Esimerkki HTTP-kirjautumispyynnöstä (The Apace Software Foundation 2017b)

4.4 Selenium

Selenium on joukko web-sovelluksille luotuja testityökaluja, joita hyödynnetään selaimen testauksen automatisointiin. Selenium on avoimen lähdekoodin sovellus, joka tarjoaa ison edun kilpailijoihin verrattuna. Tukena Seleniumilla ovat suosituimmat selaimet, kuten Internet Explorer, Chrome, Safari, Opera ja Firefox. Käyttöjärjestelmäalustoista toimivat kaikki yleisimmät: Windows, Mac, Linux, iOS ja Android. Selenium-tuoteperheeseen kuuluu Selenium WebDriver ja Selenium IDE. Selenium IDE on Firefox liitännäinen (kuva 3), joka on tarkoitettu nopeaan ja helppoon testien suorittamiseen. IDE sisältää nauhoitusominaisuuden, joka mahdollistaa testin uudelleenkäyttämisen helposti. (Vardan 2017.)



Kuva 3. Selenium Firefox liitännäinen (Vardan 2017)

Toinen tuote on Selenium WebDriver, joka tarjoaa ohjelmointirajapinnan testitapauksien tekemiseen ja toteutumiseen. Testitapaukset toimivat siten, että verkkoelementit tunnistetaan web-sivustolla ja näillä elementeillä suoritetaan toimintoja. Jokaisella selaimella on oma WebDriver, johon sovellusta ajetaan. (Vardan 2017.)

4.5 Appium

Appium on avoimen lähdekoodin natiivi automaatiotyökalu puhelimille. Appiumilla pystyy testaamaan hybridejä, natiiveja sovelluksia ja Android- ja iOS-alustoja. Puhelimen web-sovelluksista Appium tukee iOS:ssä Safaria ja Chromea ja Androidissa sisäänrakennettua selainta. Appium tukee myös WebDriver rajapintaa, tässä tapauksessa Selenium WebDriveria. Appium palvelin on kirjoitettu Node.js ja sen pystyy asentamaan kätevästi Node Package Managerilla (npm). (Appium 2017.)

5 Jatkuva integraatio (CI) ja versionhallinta

Tässä kappaleessa käsitellään termejä ja tutustutaan erilaisiin jatkuvan integraation tuotteisiin sekä versionhallintaan.

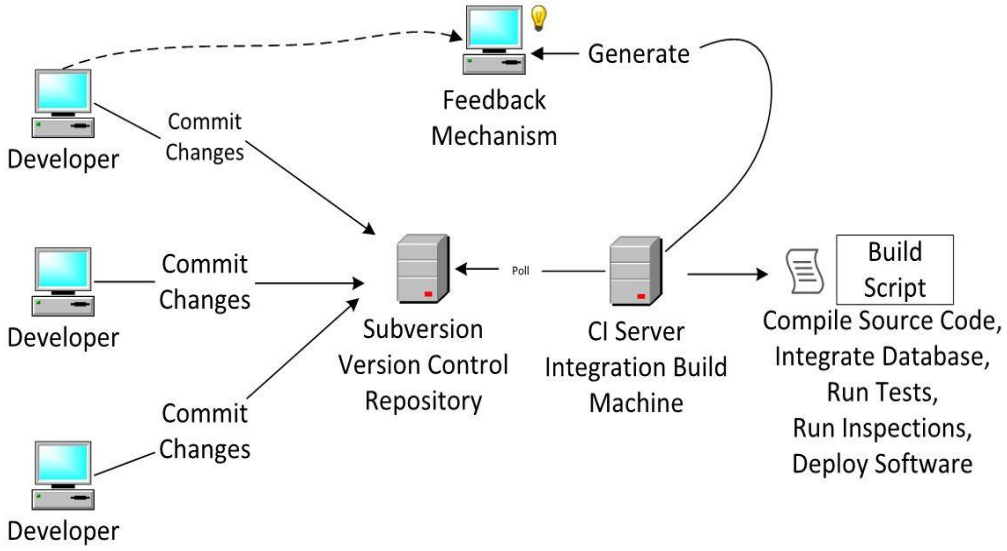
5.1 Jatkuva integraatio (CI)

Jatkuvalla integraatiolla tarkoitetaan ohjelmistokehityskäytäntöä, jossa ryhmässä työskentelevät ihmiset integroivat työtään jatkuvasti ja useimmiten tekevät muutoksia päivittäin. Kaikki integraatiot ja muutokset varmistetaan automaattisilla testeillä (build), integraatiovirheiden havaitsemiseksi mahdollisimman nopeasti. Useimmat ryhmät ovat sitä mieltä, että tämä lähestymistapa vähentää integraatio-ongelmia ja mahdollistaa ohjelmiston nopean kehityksen. (Fowler 2006.)

Jatkuva integraatio alkaa siitä, kun ohjelmistokehittäjä tallentaa lähdekoodia säilytyspaikkaan. Tyypillisesti ohjelmistokehittäjät tallentavat muutokset, jotka laukaisevat jatkuvan integraation kierron: ohjelmistokehittäjät tekevät muutoksia lähdekoodiin, tietokanta-asiantuntijat vaihtavat rakenteita ja kehitysryhmät tekevät muutoksia tiedostoihin tai muutoksia, jotka kohdistuvat lähdekoodiin. (Duvall, Matyas & Glover 2007, 4.)

Duvallin, Matyasin ja Gloverin (2007, 5) mukaan jatkuvan integraation vaiheet menevät tyypillisesti näin (kuvio 9).

1. Kehittäjä tallentaa ohjelmakoodin versionhallintaan, minkä jälkeen jatkuvan integraation palvelin hakee uudet muutokset versionhallinnasta. Versionhallinnasta tehtävät pyynnöt voivat olla tyypiltään joko ajastettuja tai pyyntöjä jatkuvasti hakevia.
2. Ohjelmakoodin tallennuksen yhteydessä, CI-palvelin huomaa muutoksen versionhallinnassa. Muutoksen seurauksena CI-palvelin hakee uuden ohjelmakoodin ja kääntää sen ohjelmaksi.
3. CI-palvelin antaa käännöksen jälkeen käännösprosessista palautteen. Palaute sisältää käännösprosessin aikana ilmi tulleet virheet ja muut tulokset.
4. CI-palvelin jatkaa automatisoitua prosessia aina uuden ohjelmakoodin tallentuessa versionhallintaan.



Kuvio 9. Jatkuvan integraation komponentit (Duvall, Matyas & Glover 2007, 5)

Järjestelmän vakaus ja toiminallisuus voidaan varmistaa vain siten, että jatkuvaan integraatioon liittyy aina automatisoitu testaus. Myöhemmin järjestelmän kehityksen edetessä testitapauksia ja ohjelmaa muutetaan, jolloin jatkuvan integraation järjestelmä rakentaa kehitettävästä järjestelmästä suoritettavan version ja ajaa testit tätä versiota vasten. Haikala ja Mikkonen (2011, 176-176) kertovat, että integroinnin suunnittelussa pitää huomioida, että jos testit tai käännös eivät mene läpi, niin järjestelmä on rikkinäinen. Muut kehittäjät eivät pysty lisäämään uusia toimintoja, ennen kuin ongelmat on ratkaistu. Tämä vaatii välitöntä huomiota kehitysryhmältä. Joskus käy niin, että testit ovat liian pitkiä ja epäonnistuvat, jolloin tarvitaan paljon muutoksia testeihin ja läpimenoaikoihin. (Haikala & Mikkonen 2011, 175-176.)

5.2 Versionhallinta ja lähdekoodi

Versionhallinta on järjestelmä, joka auttaa seuraamaan lähdekoodiin tehtäviä. Muutoksista kerrotaan versionhallinnalle, ja järjestelmä ottaa kuvan sen hetkisen tilanteen lähdekoodista. Ilman versionhallintaa kehittäjät joutuisivat tekemään monia kopioita tiedostoistaan, koska muutoksia tapahtuu paljon. Versionhallinta selvittää ongelmat pitämällä kaikki versio koodista sekä kertomalla nykyisen tilanteen. (Outlaw 2017.)

Jos kehittäjä kirjoittaa C-kielellä lausekkeita Windows–käyttöjärjestelmän muistiosovellukseen ja tallentaa sen, niin käyttöjärjestelmä kertoo, että tekstitiedosto sisältää lähdekoo-

dia. Lähdekoodilla tarkoitetaan tietokoneohjelman luomaa peruskomponenttia, jonka kehittäjä luo. Kehittäjät pystyvät hyödyntämään tekstieditoreita, integroituja kehitysympäristöjä tai visuaalisia työkaluja lähdekoodin luomiseen. Usein ohjelmistojen kehitysympäristöissä käytetään järjestelmiä, jotka tukevat kehittäjiä. Lähdekoodilla voidaan mahdollistaa osallistuminen erilaisiin yhteisöihin, esimerkiksi jakamalla koodia oppimistarkoitukseen tai kierrättämällä lähdekoodia muille muuhun käyttöön. (Rouse 2016.)

5.3 TeamCity

TeamCity on jatkuvan integraation työkalu, jonka on perustanut Jetbrains niminen yritys. Ohjelmisto on lisensoitu, mutta siitä on saatavilla myös pienemmälle kehitykselle ilmainen versio. TeamCityssä on integraatiot moniin IDE:ihin (integrated development environment), esimerkiksi omaan IntelliJ-työkaluun sekä Eclipse -ja Visual Studioon. Testauksen viitekehyksiin TeamCity tukee JUnit ja TestNG. (TeamCity 2017.)

5.4 GitLab

GitLab on tietolähteen (repository) hallintatyökalu, joka sisältää sisäisen dokumenttijärjestelmän, jatkuvan integraation (Continuous integration) ja jatkuvan toimituksen (Continuous delivery). GitLabin pystyy asentamaan yritykselle omaan käyttöön tai käyttämään suoraan GitLabin tarjontaa. Työkaluja on myös mahdollistaa ottaa vain osittain käyttöön. (Carias 2015.)

6 Automatisoidut testit osana jatkuvaa integraatiota

6.1 Tausta, tavoite ja projektisuunnitelma

Tämän työn toiminnallinen osuus perustuu startup–yritys FatAmigosin toimintaprosessin rakentamiseen. Työn taustaksi ja tavoitteeksi on määritelty testaustyökalun yhdistäminen jatkuvaan integraatioon sekä automatisoituja testejä. Ratkaisulla saadaan tehostettua FatAmigosin julkaisuprosessia ja kehitystä.

Kehittäjä tallentaa muutoksensa lokaaliin tietovarastoon ja tämän jälkeen suorittaa push-metodilla muutokset versionhallintaan. Push-metodin jälkeen jatkuva integraatio huomaa muutoksen ja aloittaa ympäristön rakentamisen testejä varten. Testien suorittamisen jälkeen pystymme tarkastelemaan testien tulokset. Tätä kokonaisuutta myös kutsutaan myös putkeksi (pipeline).

Projektisuunnitelma aloitettiin valitsemalla työkalut jatkuvaan integraatioon, testaukseen sekä sovelluksen kehittämiseen sopiva alusta. Versiohallintaan, alustaan sekä testityökaluihin oli paljon valinnanvaraa ja täten jouduimme selvittämään, mitkä työkalut toimivat parhaiten FatAmigosin kanssa.

Testityökaluja on paljon valittavissa verkossa, mutta tässä tapauksessa päädyimme Robot Frameworkiin (RF) koska se on ilmainen, hyvin suoraviivainen ja olemassa olevat kirjastot tukevat helposti tätä toteutusta. Testitapaukset luotiin avainsanoja (keyword-driven) ajatuksella, jotka ovat kirjoitettu luonnollisella kielellä. Robot Frameworkissa hyödynsimme Selenium2Library kirjastoa.

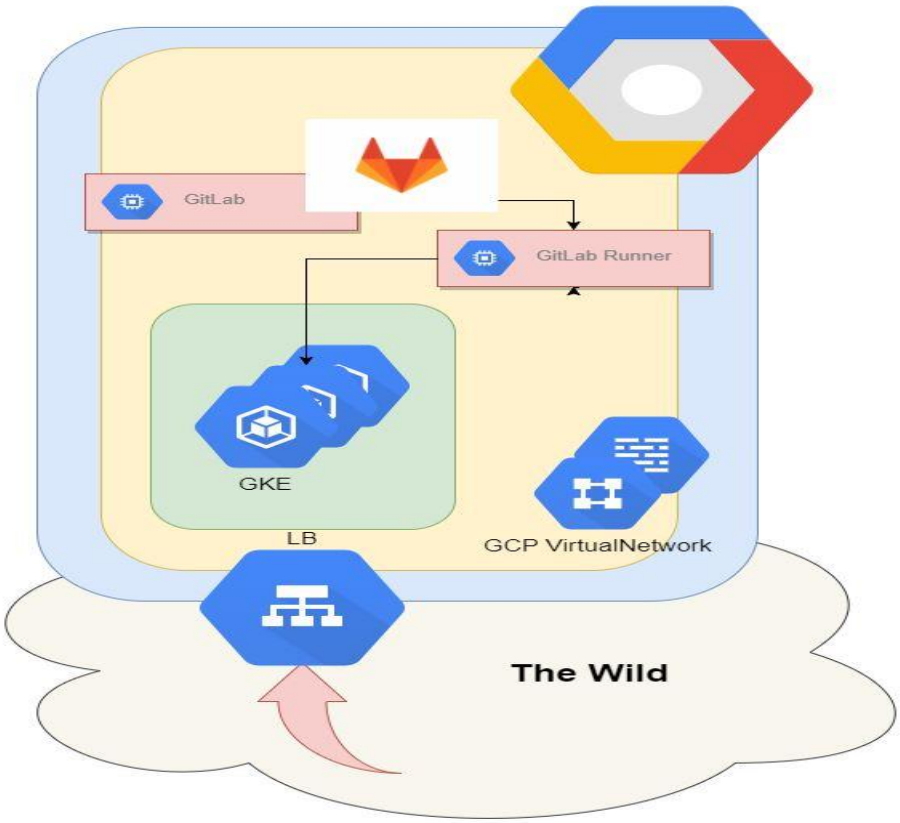
Versionhallintatyökaluksi valitsimme GitLabin, koska sen ilmaisessa versiossa saimme CI-työkalun, Wikin sekä tehtävienhallintatyökalun. Ratkaisumme ottaa Terraform-resurssienhallintatyökalu asentaessa GitLabia, vaikutti myös päätökseen.

Palvelimet päätimme ostaa Googlelta (Google Cloud Platform), koska palvelimien saatavuus oli nopeaa ja helppoa. Google Cloud Platformin kautta saa ostettua paljon palveluita, kuten virtuaalityöasemia, tilaa (storage), tietokantoja (database), IoT (Internet of Things), verkkopalveluita ja monia muita palveluita. (Google Cloud Platform 2017).

Resurssienhallintaan valitsimme Terraform-työkalun, jonka avulla asensimme GitLabin FatAmigosin käyttöön. Terraform on infrastruktuurin rakentamiseen ja muokkaamiseen tarkoitettu työkalu. Terraformin vahvuuksia on hyvä toimivuus suosittujen palvelutarjoajien kanssa ja että se on ilmainen. Terraformilla pystyy esimerkiksi: hallitsemaan talletuksia, DNS-merkintöjä ja SaaS-ominaisuuksia (software as a service). (Terraform 2017).

6.2 Toteutus

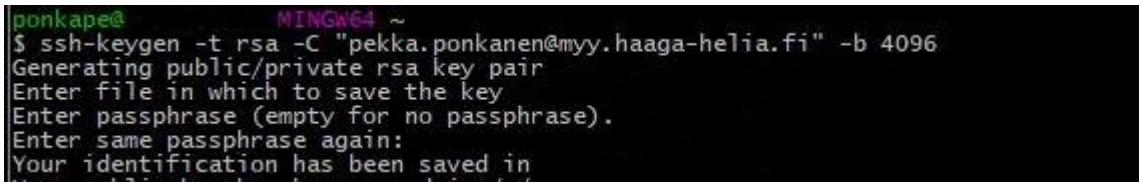
FatAmigosin perustaja Juhani Atula (Atula 2017) kertoo, että Terraform on suoraviivainen työkalu resurssienhallintaan sekä Terraformia käyttämällä ei jouduta toimittaja riippuvaisiksi. Terraformia hyödyntäen saimme GitLab-ympäristön ja GitLab runnerit Google Cloud Platformiin. Infrastruktuurissa (kuva 4) näkyy GitLabin lisäksi myös Google Kubernetes Engine (GKE) ja kuormantasaaja (LB, load balancer). Kubernetes on Googlen perustama avoimen lähdekoodin alusta ja sillä hallitaan sovelluskonttien asentamista, skaalautumista sekä monitorointia klusteriympäristöissä. (Kubernetes 2017).



Kuva 4. FatAmigosin infrastruktuuri

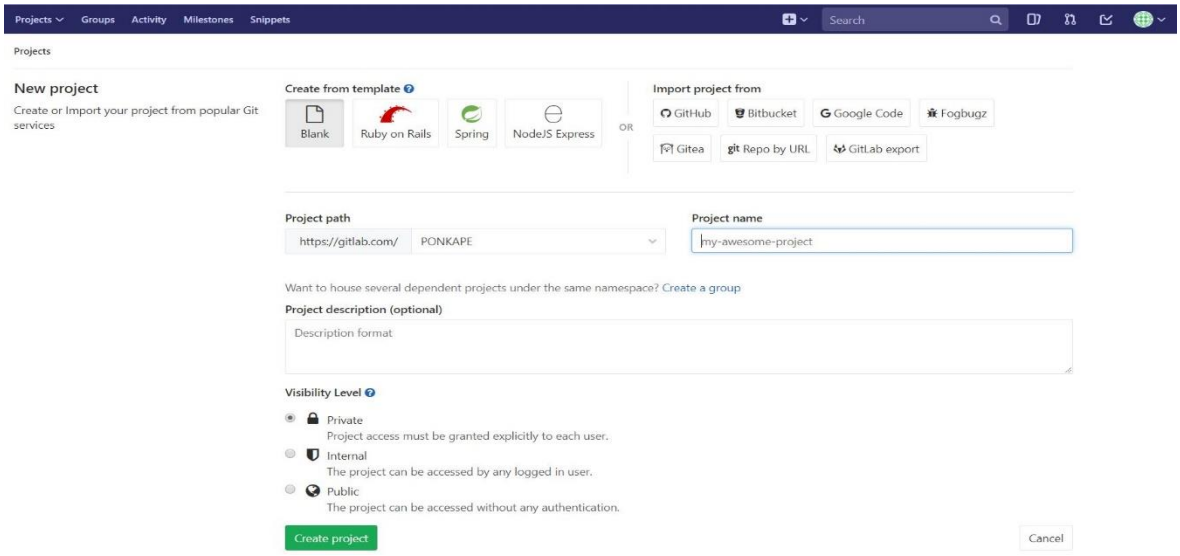
Alustan rakentamisen jälkeen siirsimme lähdekoodit alkuperäisestä tietovarasto (repository) GitHubista versionhallinta GitLabiin. GitLabiin löytyi useita tapoja tuoda lähdekoodi toisesta versionhallinnasta, koska valmiita integraatioita löytyi useita.

Ennen uuden projektin luomista, on tehty käyttäjätunnus GitLabiin ja lisätty tunnuksen alle julkinen SSH-avain. Avainparin saa luotua komentoriviltä (kuva 5).



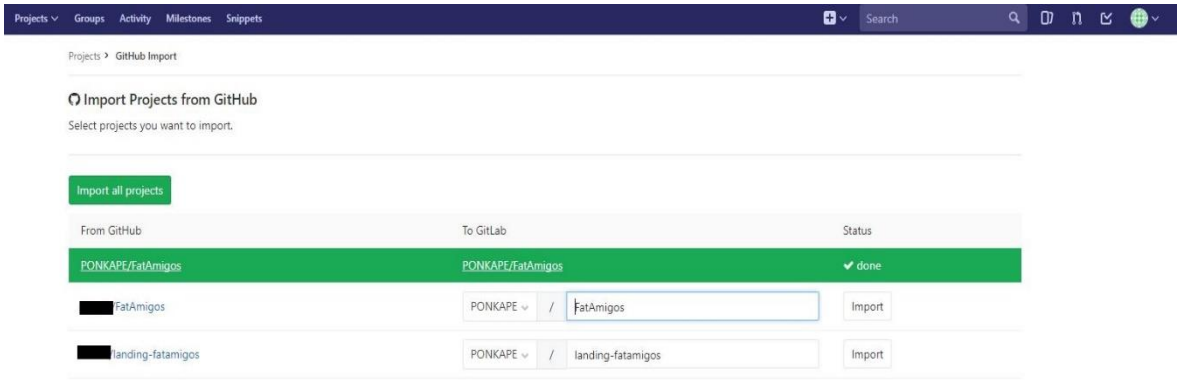
Kuva 5. SSH-avaimen luominen

Avaimen lisäyksen jälkeen luodaan uusi projekti (kuva 6) ja valitaan sopiva tapa tuoda lähdekoodi GitLab projektille. Tässä tapauksessa käytettiin GitHub–painiketta ja haettiin FatAmigosin tietovarasto (repository) sekä tuotiin se GitLabiin (import).



Kuva 6. Ote GitLabin new project sivustolta

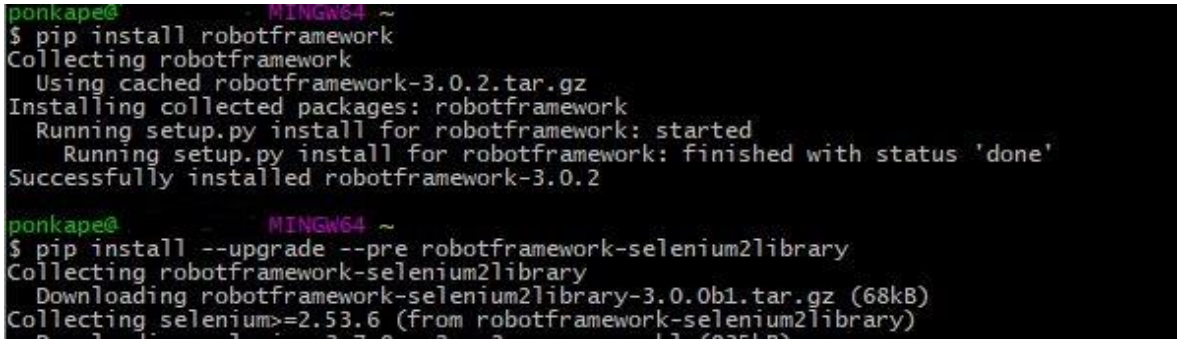
Tämän jälkeen pystyy valitsemaan olemassa olevista GitHub –tietovarastoista haluamansa ja tuoda sen GitLabiin (kuva 7).



Kuva 7. Kuva tietovaraston tuonnista GitLabissa

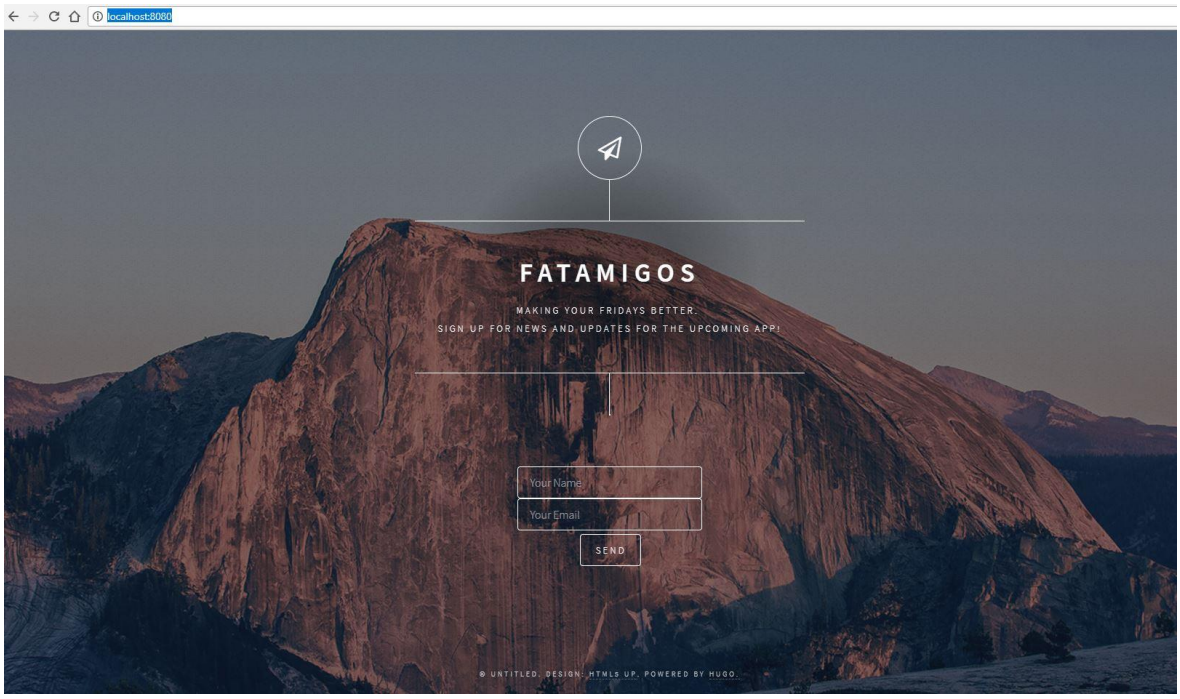
Testitapauksien kirjoittamisessa tarkasteltiin ensiksi olemassa olevaa toteutusta ja tarkisteltiin Robot Framework (RF) kirjastoja sekä mahdollisia testitapauksia. Päämääränä on saada testit kulkeutumaan jatkuvan integraation (CI) läpi ja tuottamaan tulokset kehittäjille. FatAmigosin aloitussivu (landing page) on hyvin suoraviivainen, joten toteutuksessa otetaan yksi käyttöliittymätesti (User interface test). Tämä testi kuuluu savutestauksen piiriin, koska kyseessä on järjestelmän perustoiminallisuuksiin kuuluva järjestelmän etusivu.

Robot Frameworkin ja kirjastojen asennukset suoritettiin ensin, että pääsin kirjoittamaan itse testitapauksia. Ensimmäisenä asennettiin tarvittavat komponentit: Python 2.7, Robot Framework ja Selenium2Library. Chromea varten asennettiin ChromeDriver, jonka tehtävä on hallita selainta. Tämä ajuri sijoitettiin samaan kansioon kuin itse Python. Python asennuksen jälkeen RF sekä Selenium2Libraryn asennus (kuva 8) onnistuu nopeasti Pythonin pakettihallintatyökalulla (pip):



Kuva 8. Robot Framework ja Selenium2Libray asennukset

FatAmigosin aloitussivuun (kuva 9) testiksi suunniteltiin testiautomaatio, joka syöttää nimen, sähköpostiosoitteen sekä lähettää lomakkeella olevat tiedot.



Kuva 9. FatAmigosin aloitussivu

Robot Framework testejä pystyy kirjoittamaan melkein millä tahansa tekstieditorilla, joten testitapauksien kirjoittamisen aloittaminen on nopeaa ja helppoa. Testitapaukset on hyvä kirjoittaa aina uudelleenkäytettäväksi, että aikaa ei uppoutuisi liikaa niiden korjaamiseen, jos järjestelmä muuttuu (Emery 2009, 10). Testitapaus luodaan päättömänä (headless), koska FatAmigos halusi nopean testin aloitussivulle. Testitapauksen päättömyys tarkoittaa sitä, että graafista näkymää ei tule ja selain instanssi avataan ajuritasolla. Riippuen testitapauksesta, käytetään päätöntä testausta (headless) tai normaalisti avaamalla selaimen käyttöliittymä ja ajamalla testi.

Testitapauksia pystyy hahmottelemaan aikaisessa vaiheessa, kun tiedetään sivuston tai asiakassovelluksen päämäärä. FatAmigosin tapauksessa aloitussivu (landing page) oli nopeasti rakennettu ja päästiin suoraan kehittämään testejä lähdekoodia vasten. FatAmigosin kehitystiimi oli alustanut Docker-menetelmän projektille, joka mahdollisti kehittämisen ja testaamisen vaivattomasti. Docker mahdollistaa paketoimaan sovelluksen kaikkineen riippuvuuksineen omaksi paketiksi, eli kontiksi. Omalta työasemalta ensiksi clone-komennolla tuotiin tietovarasto (kuva 10) sekä tämän jälkeen käynnistettiin Docker.

```
PS C:\thesis> git clone https://gitlab.hypeisreal.com/spatula/landingpage.git
Cloning into 'landingpage'...
remote: Counting objects: 293, done.
remote: Compressing objects: 100% (175/175), done.
remote: Total 293 (delta 125), reused 270 (delta 105)R
Receiving objects: 100% (293/293), 1.27 MiB | 0 bytes/s, done.
Resolving deltas: 100% (125/125), done.
```

Kuva 10. GitLabista tuodaan lähdekoodi clone-komennolla

Onnistuneen clone-komennon jälkeen pystyttiin käynnistämään aloitussivu Dockerin avulla (kuva 11).

```
PS C:\thesis> cd .\landingpage
PS C:\thesis\landingpage> docker run -d -v "$(pwd)/public:/usr/share/nginx/html" -p 8080:80 nginx
9fa5a1d38bcb883a1dc94c42ca610c5b7c03c1b0b01e5db1073957ff3d0bafe1
PS C:\thesis\landingpage>
```

Kuva 11. Docker käynnistetään Powershellissä

Tämän jälkeen avataan selaimella osoite http://localhost:8080 ja voidaan aloittaa Robot Framework testien laatiminen sivustoa vasten. Testitapauksessa headless.robot (kuva 12) nähdään neljä osiota: settings, variables, keywords sekä test cases.

```
*** Settings ***
Library           Selenium2Library
Suite Setup       Set Chrome Options

*** Variables ***
${SERVER}           http://localhost:8080
@{CHROME_ARGUMENTS}    --disable-infobars    --headless    --disable-gpu
${PAGE_TEXT}      FatAmigos
${TIMEOUT}        5s
${NAME}           Amigo
${EMAIL}          tresfatamigos@gmail.com
${SUCCESS}        Form submitted successfully

*** Keywords ***
Set Chrome Options
    [Documentation]    Set Chrome options for headless mode
    ${OPTIONS}=    Evaluate    sys.modules['selenium.webdriver'].ChromeOptions()    sys, selenium.webdriver
    : FOR    ${OPTION}    IN    @{CHROME_ARGUMENTS}
    \    Call Method    ${OPTIONS}    add_argument    ${OPTION}
    [Return]    ${OPTIONS}

*** Test Cases ***
Enter name, e-mail & submit
    [Documentation]    Headless test for Fatamigos landing page
    ${CHROME_OPTIONS}=    Set Chrome Options
    Create Webdriver    Chrome    chrome_options=${CHROME_OPTIONS}

    Go To    ${SERVER}
    Wait Until Page Contains              ${PAGE_TEXT}    ${TIMEOUT}
    Input text                            //input[@name='name']        ${NAME}
    Input text                            //input[@name='_replyto']       ${EMAIL}
    Wait Until Page Contains Element      css=input[type="submit"]
    Click Element                         css=input[type="submit"]
    Wait Until Page Contains              ${SUCCESS}      timeout=45s
```

Kuva 12. headless.robot

Settings-osiossa (kuva 13) määritellään tähän tiedostoon ja testeihin liittyvät riippuvuudet, kuten tarvittavat kirjastot tai omat kirjastot.

```
*** Settings ***
Library             Selenium2Library
Suite Setup         Set Chrome Options
```

Kuva 13. Settings

Variables-osiossa (kuva 14) voidaan määritellä Robot Framework-syntaksilla omia muuttujia, joita pystyy käyttämään läpi tiedoston. Esimerkiksi tässä tapauksessa määriteltiin muuttujaksi serveri, nimi ja sähköposti.

```
*** Variables ***
${SERVER}            http://localhost:8080
@{CHROME_ARGUMENTS}    --disable-infobars    --headless    --disable-gpu
${PAGE_TEXT}       FatAmigos
${TIMEOUT}         5s
${NAME}            Amigo
${EMAIL}           tresfatamigos@gmail.com
${SUCCESS}         Form submitted successfully
```

Kuva 14. Variables

Keywords-osio (kuva 15) on tarkoitettu mukautetuille avainsanoille ja kirjastoista tuoduille avainsanoille. Testitapaus-osiossa käsiteltiin Chromen headless-ominaisuutta ja ChromeDriverin-instanssin rakentamista.

```
*** Keywords ***
Set Chrome Options
    [Documentation]    Set Chrome options for headless mode
    ${OPTIONS}=    Evaluate    sys.modules['selenium.webdriver'].ChromeOptions()    sys, selenium.web
    : FOR    ${OPTION}    IN    @{CHROME_ARGUMENTS}
    \    Call Method    ${OPTIONS}    add_argument    ${OPTION}
    [Return]    ${OPTIONS}
```

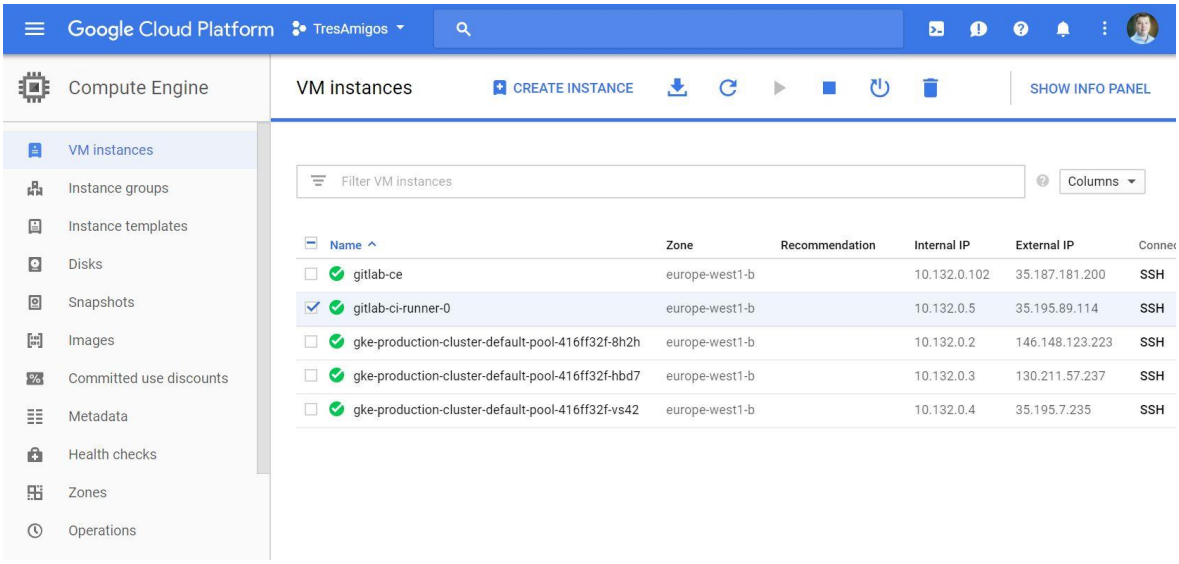
Kuva 15. Keywords

Test cases-osiossa (kuva 16) määritetään testitapaus ja sen tavoitteet. Testin alussa luodaan Chrome-instanssi ja avataan sivusto ${SERVER} muuttujaa hyödyntäen auki. Lähettämisen jälkeen tarkastetaan ”Wait Until Page Contains” avainsanalla ilmestyykö muuttujassa ${SUCCESS} teksti ”Form submitted succesfully”.

```
*** Test Cases ***
Enter name, e-mail & submit
    [Documentation]    Headless test for Fatamigos landing page
    ${CHROME_OPTIONS}=    Set Chrome Options
    Create Webdriver    Chrome    chrome_options=${CHROME_OPTIONS}
    Go To                                   ${SERVER}
    Wait Until Page Contains                ${PAGE_TEXT}    ${TIMEOUT}
    Input text                              //input[@name='name']          ${NAME}
    Input text                              //input[@name='_replyto']        ${EMAIL}
    Wait Until Page Contains Element        css=input[type="submit"]
    Click Element                           css=input[type="submit"]
    Wait Until Page Contains                ${SUCCESS}      timeout=45s
```

Kuva 16. Test Cases

Testitapausten jälkeen asennetaan kaikki tarvittavat komponentit GitLab runner -palvelimelle. GitLab runner -palvelin tekee kaikki testeihin liittyvät toimenpiteet, eli toimii testipalvelimena (kuva 17).

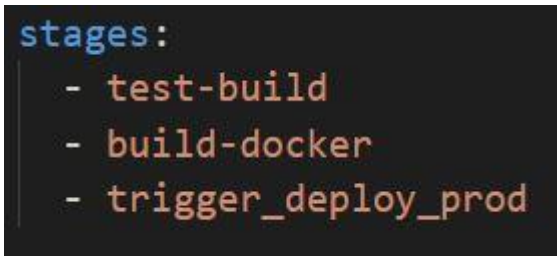


Kuva 17. Kuvakaappaus Google Cloud Platformista GitLab-runner valittuna

GitLab CIn käyttäminen on tehty suoraviivaiseksi ja helpoksi. GitLab CI tarvitsee toimiakseen tiedoston gitlab-ci.yml, joka tallennetaan tietovaraston (repository) juureen (root). Gitlab-ci.yml on konfigurointitiedosto, joka kertoo tarvittavat asetukset jatkuvalle integraatiolle. Kehityksessä FatAmigosin kehittäjät tallentavat lähdekoodia tietovarastoon ja GitLab tarkistaa gitlab.ci.yml tiedoston sekä aloittaa tehtävät hyödyntäen runneria. Tiedostossa on paljon osia, joista kerron test-build-osuuden. FatAmigosin yaml-tiedosto on liitteessä 1.

Työvaiheet (stages) määrittävät putken (pipeline) toiminallisuuden. Tässä tiedostossa on käytössä GitLabin oletuksilla olevat työvaiheet (kuva 18). Työvaiheet ovat: test-build,

build-docker & trigger_deploy_prod. Testeihin liittyen kiinnostavin on test-build. Onnistuneesta ajosta lähtee käyntiin build-doccker sekä trigger_deploy_prod jotka suorittavat julkaisun tuotantoon. Kehityksen aikana satunnaisesti GitLab-palvelu itsessään jumaht ja se jouduttiin käynnistämään uusiksi. Tätä yritettiin selvittää lokien avulla, mutta lokeja on paljon ja juurisyytä ei saatu selville heti.



Kuva 18. Stages

Työvaihe test-build (kuva 19) on testien kannalta kaikista tärkein. Tiedoston script kohdassa määritellään Dockerin käynnistäminen ja tämän jälkeen ajetaan komento ajamaan testitapaus headless.robot. Ajon jälkeen tulokset (artifacts) tulevat pyydettyyn hakemistoon.

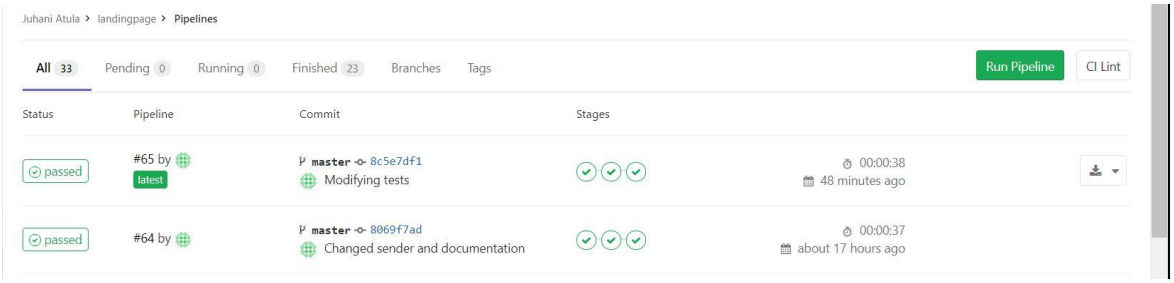
```
test-build:
  stage: test-build
  script:
    - docker run --name server -d -v "$(pwd)/public:/usr/share/nginx/html" -p 8080:80 nginx
    - robot --outputdir ui-tests  ./ui-tests/headless.robot
  artifacts:
    paths:
    - ui-tests
    expire_in: 1 hour
    when: always
  after_script:
    - docker stop server && docker rm server
  tags:
    - k8s
```

Kuva 19. Test-build

Gitlab runnerin toteuttaa pyydetyt tehtävät ja antaa tulokset GitLabiin. Runneria pystytään ajamaan erilaisissa alustoissa, kuten omassa työasemassa, Docker kontissa (container) tai pilvessä. Runner tukee Bashia (Unix) ja Powershelliä (Windows) ja sitä pystyy ajamaan projekteissa jaettuna (shared) tai yksittäisenä (specific).

6.3 Tulokset

Testitulokset olivat positiivisia ja sähköpostin lähetys onnistui. GitLabin putkesta pystyimme näkemään kaikki vaiheet, jotka jatkuva integraatio on käynyt läpi (kuva 20). Kuvasta näkyy, että kolme vaihetta (stage) on mennyt läpi putkesta ja suoriutuneet tehtävistä.



Kuva 20. Kaikki vaiheet menivät läpi GitLabissa.

Test-build (kuva 21) näyttää kaikki vaiheet, jotka on määritetty gitlab-ci.yml –tiedostossa kohdassa test-build. Kuvassa ilmenee testi nimeltä ”Headless” ja sen kaikki testit ovat PASS-tilassa, eli ovat menneet läpi.

```
Running on gitlab-ci-runner-0...
Fetching changes...
Removing ui-tests/log.html
Removing ui-tests/output.xml
Removing ui-tests/report.html
Removing ui-tests/selenium-screenshot-1.png
HEAD is now at 2b76d0d Some cleaning
From https://gitlab.hypeisreal.com/spatula/landingpage
   2b76d0d..f87f717  master     -> origin/master
Checking out f87f717b as master...
Skipping Git submodules setup
$ docker run --name server -d -v "$(pwd)/public:/usr/share/nginx/html" -p 8080:80 nginx
b323c16792313804e4f70bddce3be9e8aae4ee41bf9a2b19f09117ccc3133fe3
$ robot --outputdir ui-tests  ./ui-tests/headless.robot
==============================================================================
Headless
==============================================================================
Enter name, e-mail & submit :: Headless test for Fatamigos landing... | PASS |
------------------------------------------------------------------------------
Headless                                                              | PASS |
1 critical test, 1 passed, 0 failed
1 test total, 1 passed, 0 failed
==============================================================================
Output:  /home/gitlab-runner/builds/e008c296/0/spatula/landingpage/ui-tests/output.xml
Log:     /home/gitlab-runner/builds/e008c296/0/spatula/landingpage/ui-tests/log.html
Report:  /home/gitlab-runner/builds/e008c296/0/spatula/landingpage/ui-tests/report.html
Running after script...
$ docker stop server && docker rm server
server
server
Uploading artifacts...
ui-tests: found 5 matching files
Uploading artifacts to coordinator... ok            id=77 responseStatus=201 Created token=YgYA7B3D
Job succeeded
```

Kuva 21. Test-build näkymä GitLabissa.

Robot Frameworkista tulee oma raportti (kuva 22) ja loki (liite 3), jotka näkyvät testien päätteeksi hakemistopoluissa. Onnistunut raportti näyttää vihreältä ja klikkaamalla testitapausta, aukeaa loki kaikkineen tietoineen, joita on käytetty headless.robot tiedostossa.

Kuva 22. Robot Framework raportti

7 Pohdinta

Opinnäytetyön tavoitteena oli käydä läpi testausta, testaustyökaluja, jatkuvaa integraatiota sekä niiden yhdistämistä prosessiksi. Halusin selventää testauksen eri tasoja sekä miten jatkuva integraatio toimii käytännössä. Nykypäivänä ohjelmistokehityksessä testausta tehdään automaatiota hyödyntäen, koska tekniikka ja työkalut ovat kehittyneet sekä niiden haltuun ottaminen on tehty helpoksi. Automaatio on tullut nykypäivänä vahvasti esille ja yleisesti pyritään siihen, että mahdollisimman paljon automatisoitaisiin, jotta aikaa jäisi myös muuhun ajattelutyöhön. Testiautomaatiota rakentaessa ja testejä luodessa tulisi analysoida miksi testi halutaan automatisoida ja onko testi uudelleenkäytettävä sekä hyödyllinen kehitykselle. Manuaalisesti suoritettava testaus jää helposti vähemmälle huomiolle, vaikka sen rooli on yhä tärkeä testausprosesseissa. Kaikkea ei pystytä automatisoimaan, jonka lisäksi automaatio ei takaa sitä, että kaikki virheet tulisivat ilmi.

Lopputyön toteutuksessa tehty automaatioprosessi pystyy viemään kehittäjien muutokset suoraan tuotantoon sekä suorittaa testiautomaatioita. Tämä tukee huomattavasti FatAmigosin sovelluskehitystä ja samaa ratkaisua pystytään hyödyntämään seuraavissa FatAmigosin hankkeissa. Tulokset osoittavat testityökalun toimivuuden sekä sen sujuvuuden jatkuvassa integraatiossa. Esimerkiksi palvelun käyttäjät jotka haluavat saada tietoa tulevasta sovelluksesta, saavat syötettyä heidän nimen ja sähköpostiosoitteen sivuston

kautta. FatAmigosin kehitystiimi tietää ohjelmistokehityksen aikana, että sivusto toimii halutulla tavalla.

Itse opinnäytetyön kirjoittamiseen ja tekniseen toteutukseen perehtyminen oli kiinnostavaa. Opinnäytetyössä perehdyin uusiin palveluihin, kuten pilvipalveluun (Google Cloud Platform), uuteen tietovarastoon (GitLab) sekä jatkuvan integraation prosessiin. Lähdekirjallisuus auttaa ymmärtämään jatkuvan integraation prosessia, kokonaisuutta sekä sen arvoa. Tutustuin myös Docker-kontteihin, jotka ovat tärkeitä tämän hetkisessä sovelluskehityksessä ja ovat hyödyllisiä testiautomaatiota tehdessä. Opinnäytetyöprosessi on ollut kiinnostava, mieluisa sekä opettava, ja yhteistyökumppani FatAmigosille tehty työ tulee olemaan käytössä yrityksessä jatkossakin.

Jatkoselvityksenä opinnäytetyölle: Startup-yritys FatAmigosin olisi tärkeää rakentaa ympäristöä tulevalle asiakassovellukselle ja turvata kehitystä testiautomaatiolla.

Lähteet

Agile Alliance 2017. Iteration. Luettavissa: https://www.agilealliance.org/glossary/iteration/. Luettu: 22.11.2017.

Appium 2017. Introduction. Luettavissa: http://appium.io/introduction.html. Luettu: 12.09.2017.

Atula, J. 2017. Terraforming. Luettavissa: https://blog.mecloud.online/terraforming/. Luettu: 22.11.2017.

Bourque, P. & Fairley, R. 2014. Guide to the Software Engineering Body of Knowledge (SWEBOK v3.0). Luettavissa: http://www4.ncsu.edu/~tjmenzie/cs510/pdf/SWEBOKv3.pdf. Luettu: 20.08.2017.

Carias, K. 2015. Simple words for a GitLab Newbie. Luettavissa: https://about.gitlab.com/2015/05/18/simple-words-for-a-gitlab-newbie/. Luettu: 22.10.2017.

Duvall, P., Matyas, S. & Glover, A. 2007. Continuous Integration. Addison-Wesley. Boston.

Emery, D. 2009. Writing maintainable automated acceptance tests. Luettavissa: http://dhemery.com/pdf/writing_maintainable_automated_acceptance_tests.pdf. Luettu: 01.11.2017.

Fowler, M. Continuous Integration. Luettavissa: https://www.martinfowler.com/articles/continuousIntegration.html. Luettu: 20.06.2017.

Google Cloud Platform 2017. Products & Services. Luettavissa: https://cloud.google.com/products/. Luettu: 19.11.2017.

Graham, D., Veenendaal, E. Evans, I. & Black, R. 2007. Foundations of Software Testing. Luettavissa: https://s3-eu-west-1.amazonaws.com/kurapov/file/166.pdf. Luettu: 16.08.2017.

Haikala, I. & Mikkonen, T. 2011. Ohjelmistotuotannon käytännöt. Talentum. Helsinki.

ISTQB 2013. Advanced Level Syllabus. Luettavissa: http://www.fistb.fi/sites/fistb/files/liitteet/Advanced%20Syllabus%20-%20TTA%20FIN%2020131119%20final.pdf. Luettu: 28.06.2017.

Kasurinen, J. 2013. Ohjelmistokehityksen käsikirja. Docendo. Jyväskylä.

Koskimies, K. & Mikkonen, T. 2005. Ohjelmistoarkkitehtuurit. Talentum. Helsinki.

Kubernetes 2017. What is Kubernetes?. Luettavissa: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. Luettu: 22.11.2017.

Ojasalo, K., Moilanen, T. & Ritalahti, R. 2015. Kehittämistyön menetelmät. Sanoma Pro. Helsinki.

Outlaw, R. 2017. What is version control?. Luettavissa: https://www.visualstudio.com/learn/what-is-version-control/. Luettu: 05.09.2017.

Robot Framework 2017. Generic test automation framework for acceptance testing and ATDD. Luettavissa: https://robotframework.org. Luettu: 20.06.2017.

Rouse, M. 2016. Source code. Luettavissa: http://searchmicroservices.techtarget.com/definition/source-code. Luettu: 01.07.2017.

TeamCity 2017. Testing Frameworks. Luettavissa: https://confluence.jetbrains.com/display/TCD10/Testing+Frameworks. Luettu: 07.06.2017.

Terraform 2017. What is Terraform?. Luettavissa: https://www.terraform.io/intro/index.html. Luettu 10.11.2017.

The Apache Software Foundation 2017a. What can I do with it?. Luettavissa: http://jmeter.apache.org/. Luettu: 18.07.2017.

The Apache Software Foundation 2017b. Adding Users. Luettavissa:

http://jmeter.apache.org/usermanual/build-web-test-plan.html#adding_users. Luettu: 18.07.2017.

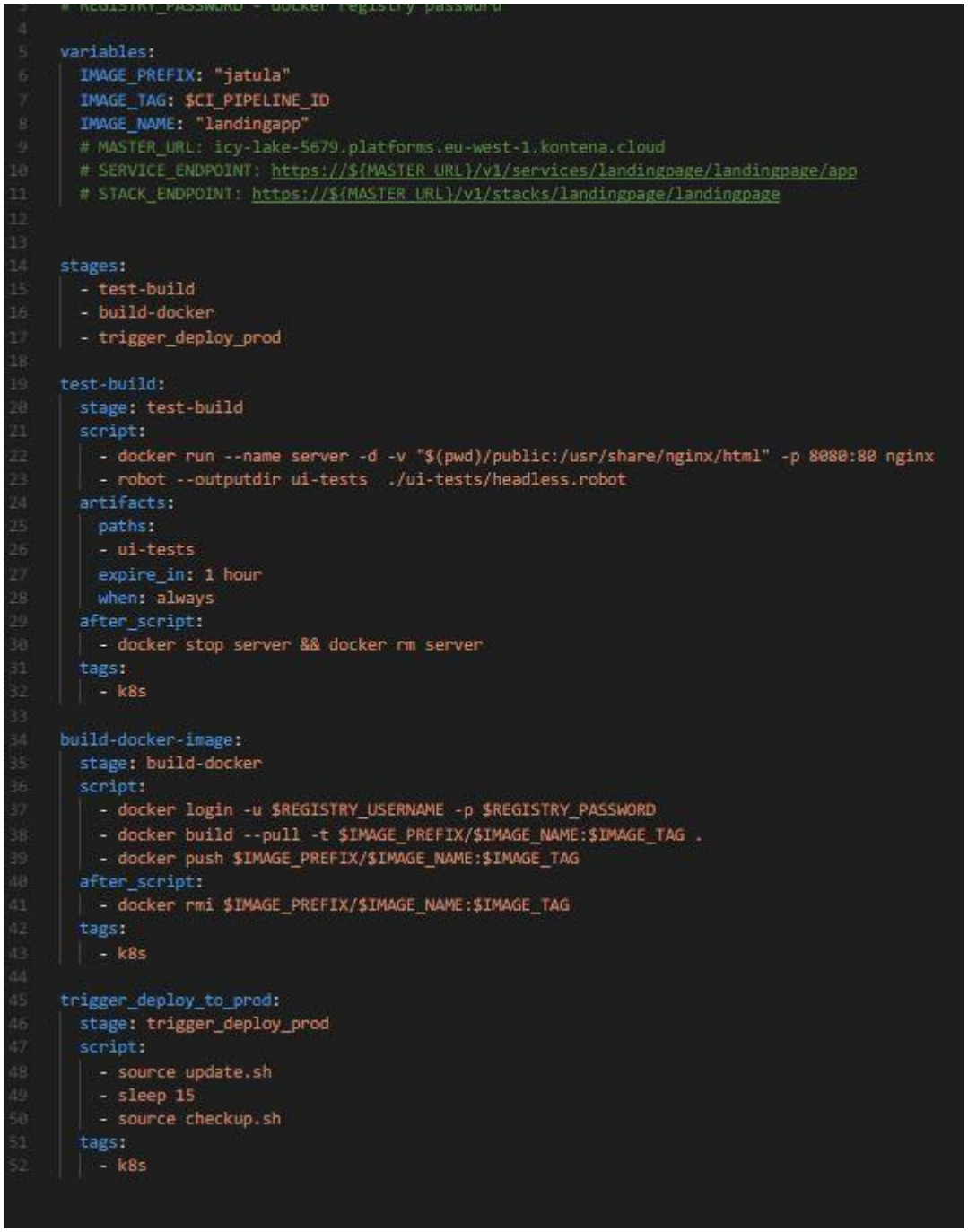
Usetrace 2017a. FAQ. Luettavissa: http://docs.usetrace.com/faq/. Luettu: 01.07.2017.

Usetrace 2017b. Quick start. Luettavissa: http://docs.usetrace.com/tutorials/1_quickstart/. Luettu: 01.07.2017.

Vardan, 2017. What is Selenium?. Luettavissa: https://www.edureka.co/blog/what-is-selenium/. Luettu: 28.08.2017.

Liitteet

Liite 1. Tiedosto .gitlab-ci.yml



```
# REGISTRY_PASSWORD - docker registry password

variables:
  IMAGE_PREFIX: "jatula"
  IMAGE_TAG: $CI_PIPELINE_ID
  IMAGE_NAME: "landingapp"
  # MASTER_URL: icy-lake-5679.platforms.eu-west-1.kontena.cloud
  # SERVICE_ENDPOINT: https://${MASTER_URL}/v1/services/landingpage/landingpage/app
  # STACK_ENDPOINT: https://${MASTER_URL}/v1/stacks/landingpage/landingpage


stages:
  - test-build
  - build-docker
  - trigger_deploy_prod

test-build:
  stage: test-build
  script:
    - docker run --name server -d -v "$(pwd)/public:/usr/share/nginx/html" -p 8080:80 nginx
    - robot --outputdir ui-tests  ./ui-tests/headless.robot
  artifacts:
    paths:
    - ui-tests
    expire_in: 1 hour
    when: always
  after_script:
    - docker stop server && docker rm server
  tags:
    - k8s

build-docker-image:
  stage: build-docker
  script:
    - docker login -u $REGISTRY_USERNAME -p $REGISTRY_PASSWORD
    - docker build --pull -t $IMAGE_PREFIX/$IMAGE_NAME:$IMAGE_TAG .
    - docker push $IMAGE_PREFIX/$IMAGE_NAME:$IMAGE_TAG
  after_script:
    - docker rmi $IMAGE_PREFIX/$IMAGE_NAME:$IMAGE_TAG
  tags:
    - k8s

trigger_deploy_to_prod:
  stage: trigger_deploy_prod
  script:
    - source update.sh
    - sleep 15
    - source checkup.sh
  tags:
    - k8s
```

Liite 2. Tiedosto headless.robot

```
*** Settings ***
Library          Selenium2Library
Suite Setup      Set Chrome Options

*** Variables ***
${SERVER}           http://localhost:8080
@{CHROME_ARGUMENTS}    --disable-infobars    --headless    --disable-gpu
${PAGE_TEXT}   FatAmigos
${TIMEOUT}        5s
${NAME}           Amigo
${EMAIL}          tresfatamigos@gmail.com
${SUCCESS}        Form submitted successfully

*** Keywords ***
Set Chrome Options
   [Documentation]    Set Chrome options for headless mode
   ${OPTIONS}=    Evaluate    sys.modules['selenium.webdriver'].ChromeOptions()    sys,
selenium.webdriver
   : FOR    ${OPTION}    IN    @{CHROME_ARGUMENTS}
   \    Call Method    ${OPTIONS}    add_argument    ${OPTION}
   [Return]    ${OPTIONS}

*** Test Cases ***
Enter name, e-mail & submit
   [Documentation]    Headless test for Fatamigos landing page
   ${CHROME_OPTIONS}=    Set Chrome Options
   Create Webdriver    Chrome    chrome_options=${CHROME_OPTIONS}

   Go To    ${SERVER}
   Wait Until Page Contains              ${PAGE_TEXT}    ${TIMEOUT}
   Input text                        //input[@name='name']       ${NAME}
   Input text                        //input[@name='_replyto']      ${EMAIL}
   Wait Until Page Contains Element       css=input[type="submit"]
   Click Element                        css=input[type="submit"]
   Wait Until Page Contains                 ${SUCCESS}                 timeout=45s
```

Liite 3. Tiedosto log.html

Headless Test Log

Generated
20171119 15:31:45 GMT+02:00
16 minutes 1 second ago

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	1	0	00:00:04	
All Tests	1	1	0	00:00:04	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Headless	1	1	0	00:00:04	

Test Execution Log

SUITE **Headless**

Full Name: Headless
Source: /home/gitlab-runner/builds/e008c296/0/spatula/landingpage/ui-tests/headless.robot
Start / End / Elapsed: 20171119 15:31:41.219 / 20171119 15:31:45.385 / 00:00:04.166
Status: 1 critical test, 1 passed, 0 failed
1 test total, 1 passed, 0 failed

SETUP **Set Chrome Options**

TEST **Enter name, e-mail & submit**

Full Name: Headless.Enter name, e-mail & submit
Documentation: Headless test for Fatamigos landing page
Start / End / Elapsed: 20171119 15:31:41.330 / 20171119 15:31:45.383 / 00:00:04.053
Status: PASS (critical)

KEYWORD ${CHROME_OPTIONS} = **Set Chrome Options**
KEYWORD Selenium2Library . **Create Webdriver** Chrome, chrome_options=${CHROME_OPTIONS}
KEYWORD Selenium2Library . **Go To** ${SERVER}
KEYWORD Selenium2Library . **Wait Until Page Contains** ${PAGE_TEXT}, ${TIMEOUT}
KEYWORD Selenium2Library . **Input Text** //input[@name='name'], ${NAME}
KEYWORD Selenium2Library . **Input Text** //input[@name='_replyto'], ${EMAIL}
KEYWORD Selenium2Library . **Wait Until Page Contains Element** css=input[type="submit"]
KEYWORD Selenium2Library . **Click Element** css=input[type="submit"]
KEYWORD Selenium2Library . **Wait Until Page Contains** ${SUCCESS}, timeout=45s