Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Daniel Laaksonen

Priorisoinnin hyödyntäminen testiautomaatiossa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikan tutkinto-ohjelma

Insinöörityö

27.11.2019

Tekijä Otsikko Sivumäärä Aika	Daniel Laaksonen Priorisoinnin hyödyntäminen testiautomaatiossa 34 sivua + 1 liite 27.11.2019
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Pelisovellukset
Ohjaajat	Lehtori Antti Laiho Concept Business Lead Teemu Frisk

Insinöörityö käsittelee testiautomaation tulosten keräämistä ja tulosten avulla tehtävää priorisointia, joiden avulla on mahdollista tehostaa testiautomaatiota. Päämääränä oli kehittää testiautomaation priorisoimisen konseptia jatkuvan integraation projekteissa.

Insinöörityö tehtiin kansainvälisen ohjelmistoyrityksen alla toimivalle yksikölle. Yksikkö tarjoaa ohjelmistoautomaatiokonsultteja ja tekee konseptikehitystä ohjelmistoautomaatioon liittyen. Yksikön laaja osaaminen ohjelmistoautomaation puolella ja Robot Frameworkin kehitykseen osallistuminen tukivat vahvasti projektin etenemistä.

Työssä perehdyttiin testitulosten keräämiseen ja niiden datan tallentamiseen erilliseen tietokantaan eri ympäristöistä. Projektissa tutkittiin tapoja analysoida tallennettua dataa ja etsiä siitä yhtäläisyyksiä tiedostomuutoksiin. Raportissa esitellään myös kattavasti modernia sovelluskehitystä erityisesti testiautomaation ja ketterän kehityksen näkökulmasta. Työssä löydettiin erilaisia tapoja rakentaa priorisointimalleja ja mahdollisuuksia kerätä tietoa projektista ja hyödyntää sitä priorisoinnissa.

Työ oli osa TestManager-nimisen sovelluksen kehitystä. Kehityksessä hyödynnettiin modernin sovelluskehityksen menetelmiä ja pyrittiin mahdollisimman ketterään toimintamalliin. Projektissa käytettiin modernille sovelluskehitykselle ominaisia työkaluja ja teknologioita. Sovellus toteutettiin mikropalveluina, joita pystytään suorittamaan eri pilviympäristöissä.

Insinöörityön lopputuloksena saatiin toimiva prototyyppi sovelluksesta, jonka avulla pystytään tuottamaan lisäarvoa testaajille, kehittäjille ja liiketoimintapuolen henkilöstölle. Sovellus kerää testidataa ja analysoi tiedostomuutosten vaikutusta testeihin. Sovellus tarjoaa myös visuaalisen käyttöliittymän tiedon hyödyntämiseen. Insinöörityössä tehtyä tutkimusta ja konseptia käytetään TestManager-sovelluksen jatkokehityksessä.

Avainsanat	Robot Framework, testiautomaatio

Author Title Number of Pages Date	Daniel Laaksonen Advantages of prioritizing in test automation 34 pages + 1 appendice 27 November 2019
Degree	Bachelor of Engineering
Degree Programme	Information and communications technology
Professional Major	Game Applications
Instructors	Antti Laiho, Senior Lecturer Teemu Frisk, Concept Business Lead

This thesis deals with the collection of test automation results and prioritization of results that can be used to enhance test automation. The goal was to develop a concept around prioritizing in test automation, which is taking advantage of continuous integration.

The thesis has been done for the unit under an international software company. The unit provides software automation consultants and develops concept for software automation. The unit's extensive expertise in software automation and its involvement in the development of the Robot Framework strongly supported the project's progress.

In this thesis, we got acquainted with collecting test results and storing data in a separate database. The project explored ways to analyze stored data and looked for similarities in file changes. The report also comprehensively introduces modern application development, especially from the perspective of test automation and agile development. In the thesis, different ways to build prioritization models and opportunities to gather information about the project and use it in prioritization were found.

This work is part of the development of TestManager. The development utilized modern application development methods and strived for the most agile operating model. The project used tools and technologies that are typical for modern application development. The application was designed as micro services that can be run in different cloud environments.

The result of this thesis was a functional prototype of the application that can deliver value for testers, developers and the business side of the project. The application collects test data and analyzes the effect of file changes on tests. The application also provides a visual interface for utilizing information. The research and concept of the thesis will be used in further development of the TestManager application.

Keywords	Robot Framework, Test automation

Sisällys

Lyhenteet

1 Johdanto 1

2 Ohjelmistotestaamisen historia ja projektin tavoitteet 2

2.1 Visualisoinnin tavoitteet 2
2.2 Priorisoinnin tavoitteet 3

3 Ohjelmistotestaus ja työkalut 4

3.1 Moderni sovelluskehitys 4
3.2 DevOps-menetelmät 5
3.3 Testiautomaatio 6
3.4 Hyväksymistestivetoinen ohjelmistokehitys 6
3.5 Robot Framework -automaatiokehys 7
3.6 TestArchiver-työkalu 8
3.7 Docker-teknologia 9
3.8 Testien ajaminen kehitysputkissa 10

4 TestManager projektin suunnittelu ja toteutus 11

4.1 TestManagerin arkkitehtuuri 12
4.2 Testien priorisointi 18
4.3 TestManagerin käyttö 22
4.4 TestManagerin oma kehitysputki 24

5 TestManagerin jatkokehitys ja projektista saatu palaute 25

5.1 Uusi käyttöliittymä 26
5.2 Arkkitehtuurin uudistaminen 28
5.3 Priorisoinnin laaja implementointi 28
5.4 Palaute 30

6 Yhteenveto 32

Lähteet 33

Liitteet

Liite 1. SALabs-yksikön GitHub-säilö

Lyhenteet

API	Application programming interface. Ohjelmointirajapinta, jonka avulla sovellukset voivat keskustella keskenään.
ATDD	Acceptance test driven development. Ohjelmistokehitystapa, jossa hyväksyntätestit kirjoitetaan ennen varsinaisen ohjelmistokehityksen aloittamista.
Azure	Microsoftin pilvipalvelu, joka tarjoaa erilaisia ratkaisuja pilviympäristöihin liittyen.
Backlog	Lista projektissa kehitettävistä ominaisuuksista.
CI	Continuous Integration. Ohjelmistokehitys tehdään pieninä palasina, joita integroidaan jatkuvasti tuotannon kaltaiseen päähaaraan. Kaikki integrointi olisi tavoitteellista testata muiden ohjelmistomuutosten kanssa.
CD	Continuous Deployment. Ohjelmistomuutokset testataan automaattisesti ja yhdistetään tuotantoversioon automaattisesti.
DevOps	Toimintamalli, jolla pyritään automatisoimaan ohjelmistokehitykseen, testaukseen ja ylläpitoon liittyviä toimintoja.
Docker	Virtualisointityökalu, jolla on mahdollista korvata perinteisiä virtuaalipalvelimia.
UI	User Interface. Käyttöliittymä, jonka kautta tuotetta käytetään.
Git	Ohjelmistokehityksen versiohallintajärjestelmä.
RPA	Robotic Process Automation. Prosessien automatisointiin käytetty teknologia.
SUT	System Under Test. Ohjelmisto, jota vasten testejä tehdään.

XML	Extensive Markup Language. Standardi rakenteelliselle merkintäkielelle.
Jenkins	Avoimen lähdekoodin automaatiopalvelinarkkitehtuuri.
Python	Tulkattava ohjelmointikieli, joka on tehokas vaativaan laskentaan.
Node.js	Avoimen lähdekoodin ympäristö JavaScript-koodin suorittamiseen.

1 Johdanto

Laadunvarmistus on kiinteä osa modernia sovelluskehitystä, jossa sovelluksen kehitys etenee DevOps-malleja noudattaen. Sovelluksen kehittäminen tehdään pieninä ominaisuuksina, joita jatkuvasti integroidaan. Tämänkaltainen kehittäminen vaatii myös jatkuvaa laadunvarmistusta. Testaamisen automatisointi nopeuttaa laadunvarmistusta huomattavasti ja antaa kehittäjille mahdollisuuden keskittyä sovelluksen kehittämiseen testaamisen sijasta. Testitulosten ymmärtäminen saattaa kuitenkin vaatia avustusta testiautomaation suunnittelijalta tai tietoa siitä, mitkä testit olisi ajettava mitäkin ohjelmistomuutosta vasten.

Insinöörityö on tehty Siili Solutions Oyj:n SALabs-yksikölle, jonka nimi on lyhenne sanoista Software Automation Laboratories. Yksikkö kuuluu Siilin New Solutions -osastoon, jonka tavoitteena on kehittää uusia liiketoimintaratkaisuja tavallisen konsulttityön lisäksi. Yksikkö on keskittynyt tuottamaan konsultointia testiautomaation, ohjelmistoautomaation ja DevOps-toimintamallin ympärille. Yksikön tehtäviin kuuluun myös konseptikehitys oman erikoisosaamisen alueella, ja tämä insinöörityö käsittelee konseptikehityspuolen kehittämän TestManager-konseptin priorisointipuolta.

Insinöörityön tarkoituksena oli kehittää työkalu testihistorian analysoimiseen, virheiden tunnistamiseen ja testiajojen priorisointiin. Insinöörityöraportissa käsitellään testiautomaatiodatan analysointia, sen rikastamista ja testien priorisointia tämän perusteella. Konseptikokonaisuuden työnimeksi valittiin TestManager, joka viittasi käyttöliittymään, josta myös managerit voivat seurata projektin etenemistä testien osalta sekä ymmärtää projektin tilannetta.

Insinöörityön osa konseptissa oli erityisesti priorisointipalvelun kehittäminen ja kehitysympäristön rakentaminen. Projektitiimi koostui projektipäälliköstä, full stack -kehittäjistä, backend-kehittäjistä, automaatiotestaajista sekä DevOps-osaajista. Projekti käynnistyi huhtikuussa 2019 ja jatkuu uuden kehitysvaiheen mukana. Insinöörityön osalta työ päättyi lokakuuhun 2019, jolloin projektin jatkosuunnitelmat valmistuivat.

Insinöörityössä ei mainita Siilin asiakkuuksia nimeltä salassapitovelvollisuuksien takia, vaan eri asiakasprojekteihin viitataan tiettyinä asiakasprojekteina.

2 Ohjelmistotestaamisen historia ja projektin tavoitteet

Testaaminen on ollut osa ohjelmistokehitystä niin kauan, kuin ohjelmistoja on kehitetty. Testaamalla voidaan varmistaa ohjelmiston toimivuus halutulla tavalla, ja sitä tulisi tehdä aktiivisesti, jotta mahdolliset virheet havaittaisiin nopeasti.

Manuaalitestaajilla tarkoitetaan kehitystiimin jäseniä, jotka alkavat testata uutta ominaisuutta sen valmistuttua kehittäjiltä. Testaajan antaman palautteen perusteella ominaisuus palautuu takaisin kehittäjälle, jos siitä löydetään virheitä. Mikäli ominaisuus toimii halutulla tavalla, se siirretään seuraavaksi tarkasteltavaksi ennen julkaisua.

Testiautomaatiossa testaaminen automatisoidaan sovellusten avulla. Sovellus testaa kehitettävän sovelluksen toiminnallisuuksia ja koostaa testiajosta raportin. Osassa testiautomaatiokehyksiä raportti on pelkästään konsolituloste ja osassa laaja lokitiedosto.

TestManagerin ensisijaiset tavoitteet olivat kehittää yksittäisen testiajon visualisointia ja projektin testihistorian visualisointia sekä tutkia mahdollisuuksia hyödyntää koneoppimista ja tietokoneen suorittamaa priorisointia testauksen kehittämiseksi.

Insinöörityön osalta tavoitteena oli tutkia erityisesti priorisoinnin hyödyntämistä ja löytää erilaisia keinoja löytää yhtäläisyyksiä testitulosten ja tiedostomuutosten välillä.

2.1 Visualisoinnin tavoitteet

Testiautomaatiosovelluksilla on erilaisia lähestymistapoja testiajon raportointiin. Osa työkaluista kertoo ajon tulokset komentorivitulosteena ja osa tallentaa testiajon vaiheineen esimerkiksi XML-tiedostona. Esimerkiksi Robot Framework kirjoittaa XML-tiedostoa rivi kerrallaan ajon edetessä ja ajon valmistuttua rakentaa XML-tiedoston perusteella log.html-nimisen tiedoston, josta voi tarkistella testiajoa selaimen kautta. Kuvassa 1 on esiteltynä kyseinen tiedosto esimerkkitestin tuottamana.

REPORT

For Tests Log

Generated
20191102 10:00:59 UTC+02:00
19 seconds ago

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	4	4	0	00:00:00	
All Tests	4	4	0	00:00:00	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
loops	4	4	0	00:00:00	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
For Tests	4	4	0	00:00:00	

Test Execution Log

SUITE For Tests 00:00:00.079
Full Name: For Tests
Source: c:\Users\daniel.laaksonen\Documents\TestArchiver\robot_tests\tests\loop_suite\for_tests.robot
Start / End / Elapsed: 20191102 10:00:59.182 / 20191102 10:00:59.261 / 00:00:00.079
Status: 4 critical test, 4 passed, 0 failed
4 test total, 4 passed, 0 failed

TEST For-Loop-In-Range 00:00:00.003
Full Name: For Tests.For-Loop-In-Range
Tags: loops
Start / End / Elapsed: 20191102 10:00:59.206 / 20191102 10:00:59.209 / 00:00:00.003
Status: PASS (critical)
FOR ${INDEX} IN RANGE [1 | 3] 00:00:00.002

TEST For-Loop-Elements 00:00:00.007

TEST For-Loop-Exiting 00:00:00.005

TEST Repeat-Action 00:00:00.001

Kuva 1. Robot Frameworkin luoma visualisointi lokitiedostosta.

Konseptin tavoitteena oli rakentaa visuaalisesti miellyttävä ja helposti lähestyttävä esitys testiajosta. Sovelluksen tulisi palvella testaajien lisäksi managereita, jotka ovat kiinnostuneita projektin etenemisestä ja tilasta.

Projektille haluttiin toteutus, jossa esitellään yhden testiajon näkymä, historianäkymä ja mahdollisuus vertailla testiajoja keskenään. Jokaisen näkymän oli tarkoitus antaa käyttäjälle mahdollisimman paljon tietoa testeistä.

2.2 Priorisoinnin tavoitteet

Tietyssä Siilin asiakkuudessa koko testikokonaisuuden ajaminen kestää jopa kaksi viikkoa, minkä aikana kehittäjillä ei ole mahdollista edetä projektin kehityksessä toivotulla vauhdilla. Tämänkaltaiset ongelmat yleisesti ratkaistaan ajamalla vain tietty osa testeistä kerrallaan ja koko testipaletin ajaminen jää esimerkiksi lomaviikoille. Projektien edetessä testien määrä kasvaa useimmiten rinnalla, ja näin testien suoritusajat nousevat huomattavasti. Pidentyvien testiajojen suoritusaikojen vuoksi projekteissa nousee esiin tarve rajata vain osa testeistä ajettavaksi, mikä yleensä tehdään manuaalisesti, ja osa virheitä tunnistavista testeistä saattaa jäädä ajojen ulkopuolelle. Tähän haluttiin kehittää ratkaisu, joka tunnistaa, mitkä testit ovat relevantteja eri lähdekoodin osien muokkauksille.

Konseptikehityksen näkökulmasta tavoitteena oli myös selvittää mahdollisuuksia analysoida testien laatua kerätyn datan avulla ja antaa palautetta testien laadusta. Testien laadun ohella pyrittiin etsimään keinoja tunnistaa testien suoritukseen liittyviä muutoksia ja niiden vaikutuksia suoritusaikoihin.

3 Ohjelmistotestaus ja työkalut

Tämä luku käsittelee tietoa modernista ohjelmistokehityksestä ja laadunvarmistuksesta projekteissa yleisellä tasolla. Sen lisäksi esitellään TestManagerin työkaluja, joiden toiminnallisuuksilla on iso osa itse konseptin kehityksessä.

3.1 Moderni sovelluskehitys

Modernin sovelluskehityksen menetelmien tarkoituksena on parantaa asiakastyytyväisyyttä ja ohjelmistokoodin laatua, nostaa kehitystiimin tehokkuutta ja välttää isoja ongelmia. Suosittuja modernin sovelluskehityksen menetelmiä ovat mikropalveluratkaisut, testiautomaatio, liputetut ominaisuuspäivitykset ja tulosperusteinen ohjelmistokehitys. [1.]

Mikropalveluilla tarkoitetaan sovelluksen toiminnallisuuksien purkamista pieniksi osiksi, joissa jokainen palvelu hoitaa yksittäistä tehtävää. Atlassianin tekemissä tutkimuksissa huomattiin, että 71 % mikropalvelurakenteita käyttävistä tiimeistä pystyy tehokkaammin testaamaan ja julkaisemaan uusia ominaisuuksia. [1.] Mikropalveluiden käyttäminen projekteissa mahdollistaa myös kuormantasaajien tehokaan käyttämisen, mikä helpottaa palveluiden skaalaamista käytön ja tarpeen mukaisesti.

Tulosperusteisessa ohjelmistokehityksessä pyritään seuraamaan tuotantoon siirretyiden ominaisuuspäivityksien asiakastyytyväisyyttä. Ohjelmistokehitystä suunnitellaan tavoitteisiin perustuen eikä suoraan varsinaisiin ominaisuuksiin. Menetelmää hyödynnetään usein liputettujen ominaisuuspäivityksien yhteydessä, jolloin ominaisuuksia laitetaan käyttäjillä päälle ja pois päältä ja seurataan käyttäjien palautetta ominaisuuksista. Saatu palaute ohjaa tiimin seuraavia askeleita ohjelmiston kehityksessä. [2.] Palveluina toteutettu arkkitehtuuri mahdollistaa nykyaikaisissa pilviympäristöissä tehokkaan liputettujen ominaisuuspäivityksillä testaamisen. Tehokkuus perustuu siihen, että on mahdollista

käynnistää esimerkiksi kymmenen palvelua, joista yksi käyttää uutta ominaisuutta. Näin saadaan 10 % käyttäjistä kokeilemaan uudella ominaisuuspäivityksellä varustettua palvelua.

Modernille ohjelmistokehitykselle on myös ominaista pyrkiä tunnistamaan ongelmia ja virheitä mahdollisimman aikaisessa vaiheessa, jolloin on mahdollista välttää projektin laajaa refaktorointia. Testiautomaatiolla pyritään tunnistamaan laatuun liittyviä ongelmia ja käyttäjien antamaa palautetta liputetuista ominaisuuspäivityksistä käytetään jatkokehityksen ohjaamiseen.

3.2 DevOps-menetelmät

DevOps-termi on laajasti käytössä ohjelmistokehityksessä, ja usein se sekoitetaan työtehtäväksi. DevOpsille ei ole akateemista määrittelyä, vaan se koostuu menetelmistä, jotka pyrkivät lyhentämään järjestelmien kehityssykliä ja tarjoamaan jatkuvaa toimitusta korkealla laadulla. [3.]

Jatkuva integrointi tarkoittaa sovelluksen kehittämistä ominaisuus kerrallaan ja ominaisuuksien jatkuvaa integrointia keskenään. Jatkuvuuden varmistamiseksi manuaalisia vaiheita automatisoidaan, kuten sovelluksen kokoaminen ja testaaminen. Jokaisen tiedostomuutoksen pohjalta olisi suositeltavaa suorittaa ominaisuuteen liittyvät testit ja ominaisuutta integroitaessa pääkehityshaaraan suorittaa projektin laajuiset automaatiotestit. Ideaalitilanteessa testausympäristö vastaa täysin tuotantoympäristöä, jolloin sovelluksen toimintaa voidaan mallintaa mahdollisimman tarkasti. [4.]

Jatkuvalla toimituksella viitataan sovelluskehityksen suunniteluun, jossa uusia pieniä ominaisuuksia tuodaan jatkuvasti tuotantoon isojen päivitysten sijaan. Malli tehostaa päivitysten välistä aikaa ja tähtää siihen, että seuraavan ominaisuuden kehittäminen aloitetaan välittömästi edellisen valmistuttua. Jatkuvan toimittamisen yleistyminen lähihistoriassa perustuu myös vahvasti pilvipalveluiden ja palvelinsalien lisääntymiseen, mikä nopeuttaa testaamista ja sovelluksen rakentamista sekä laskee tuotantoon siirtämiseen liittyviä kuluja. [5.]

3.3 Testiautomaatio

Testiautomaatiolla tarkoitetaan erillisen sovelluksen käyttämistä testauksen suorittamiseen. Testit suunnitellaan varmistamaan sovelluksen toimivuutta ja löytämään mahdollisia virheitä toiminnallisuuksien käyttäytymisessä. [6.]

Testiautomaatiota ajetaan usein automaatiopalvelimelta, kuten Jenkins-palvelimelta. Useimmat Git-versionhallintasovellukset eivät sisällä automatisointipalveluita, minkä takia käytetään ulkopuolisia sovelluksia automaation suorittamiseen.

Testiautomaation tarkoituksena ei ole korvata manuaalitestaajia, vaan automatisoida testit, jotka ovat toistuvia tai ovat monimutkaisia, kuten laskentaa vaativat testit. Nämä testit toimivat usein laadunvarmistuksessa, minkä takia niitä ajetaan jatkuvaa integrointia hyödyntävissä projekteissa. [7.] Kehittäjät voivat itsenäisesti ajaa testejä kehitysputkissa, mikä mahdollistaa nopean palautteen lähdekoodimuutoksen vaikutuksesta ohjelmiston toimintaan. Projekteissa ilman testiautomaation hyödyntämistä kehittäjät yleisesti tekevät isompia muutoksia lähdekoodiin ennen testaajalle siirtämistä. Tämän vuoksi virheiden löytäminen on hitaampaa ja se vaikuttaa koko ohjelmistokehityksen nopeuteen.

3.4 Hyväksymistestivetoinen ohjelmistokehitys

Acceptance test-driven development (ATDD) eli hyväksymistestivetoinen ohjelmistokehitys on yleistyvä ohjelmistokehitysmenetelmä, jossa hyväksymistestit kirjoitetaan, ennen kuin ohjelmistokehittäjät alkavat kirjoittaa varsinaista sovellusta [8]. Menetelmän hyväksymistestien määrittelyt kirjoitetaan yhdessä testaajien, kehittäjien ja tilaajan kanssa. Projektiin tehdään kehityksen alettua myös muita testejä, mutta hyväksymistestit toimivat ominaisuuksien validoijina. Kun hyväksymistesti menee onnistuneesti läpi, voidaan ominaisuutta pitää valmiina projektin kehityksen näkökulmasta. [9.]

Hyväksymistestit toimivat myös selkeinä mittareina tilaajille ja liiketalouspuolen henkilöille. Testien avulla on helppoa arvioida projektin etenemistä ja ymmärtää kehittämistä ilman teknistä osaamista.

3.5 Robot Framework -automaatiokehys

Robot Framework on avoimen lähdekoodin sovellus hyväksymistestaukseen, hyväksymistestivetoiseen kehitykseen ja ohjelmistorobotiikkaan. Robot Framework sai alkunsa vuonna 2005 Pekka Klärkin diplomityöstä Nokia Networksille, joka jatkokehitti sovellusta. Kesäkuussa 2008 Robot Framework julkaistiin avoimena lähdekoodina, ja Nokia Networks siirtyi sponsoroimaan sovelluksen jatkokehitystä. [10.]

Valtaosassa Siilin asiakkuuksia Robot Framework on valikoitunut standardiksi testiautomaation toteutuksille, ja Siili Solutions on yksi sovellusta kehittävän Robot Framework Foundationin tukijoista. Robot Framework valikoitui TestManagerin ensisijaiseksi testiautomaatiokehykseksi.

Robot Frameworkilla tehdyt testit kirjoitetaan tiedostoon, joka on .robot-päätteinen. Tiedosto vastaa Suitea eli kansiota testeistä. Yhdessä Suitessa on mahdollista olla useita testejä. Tavallinen käyttö esimerkiksi web-testauksessa on kirjoittaa .robot-tiedosto jokaista sivua kohden ja sen sisällä on erillisinä testeinä kaikki testattavat ominaisuudet. Esimerkkikoodissa 1 on esiteltynä robottitestitiedosto, jossa ensimmäinen testi avaa selaimen Googleen ja ottaa kuvankaappauksen. Toinen testi kirjoittaa lokiin muuttujaan sidotun arvon, eli testissä tekstin ”Robot Framework”.

```
*** Settings ***
Library             SeleniumLibrary
*** Variables ***
${framework}         Robot Framework

***Test Cases***
First Test
     Open Browser              https://www.google.fi        Chrome
     Capture Page Screenshot
Second Test
     Log                       ${framework}
```

Esimerkkikoodi 1. Yksinkertainen robottitestitiedosto.

Robot Framework on Pythonilla kirjoitettu ympäristö, joka perustuu avainsanalla suoritettavaan testaukseen. Avainsanoja pystyy itse kirjoittamaan, ja pääajatus syntaksissa on, että testit pysyvät helposti ymmärrettävinä myös henkilöille, jotka eivät ole

kirjoittaneet testejä. [10.] Robot Framework tukee myös ulkopuolisia kirjastoja, joita käyttäjät voivat luoda robot-syntaksilla tai suoraan kirjoittaa Pythonilla. Mainittujen ominaisuuksien vuoksi Robot Framework soveltuu erinomaisesti eri SUT:ien eli testattavien järjestelmien laajaan testaamiseen, eikä se ole rajoittunut esimerkiksi pelkästään käyttöliittymätestaukseen. Rakenteensa ansiosta Robot Framework on suosittu työkalu myös ohjelmistorobotiikan puolella. [11.]

Robot Framework generoi testiajosta XML-tiedoston ja selaimeen avautuvan lokitiedoston. Lokitiedostoon on mahdollista sisällyttää myös kuvankaappauksia testiajon varrelta. XML-tiedostoa kirjoitetaan testien edetessä, minkä ansiosta ajon tiedot tallentuvat, vaikka testejä suorittava robotti kaatuisi. Log- ja Report-tiedostot edellyttävät testiajon suorittamista loppuun asti. [11.]

3.6 TestArchiver-työkalu

TestArchiver on SALabs-yksikössä työskentelevän Tommi Oinosen kehittämä työkalu testidatan tallentamiseen SQL-tietokantaan. Työkalu tukee SQLite- ja PostgreSQL-kantoja tallentamiseen. [12.] Työkalua on mahdollista käyttää kuuntelijalla, joka seuraa testiajoa ja raportoi jatkuvasti tietokantaan. Tämä ominaisuus mahdollistaa tiedon tallentumisen, vaikka testejä suorittava sovellus kaatuisi. Vaihtoehtoinen tapa käyttää sovellusta on lukea valmis testiraportti ja siirtää sen tiedot kantaan.

TestArchiver lukee Robot Frameworkin XML-tiedostoja ja Mocha-yksikkötestien tuloksia. Työkalu yhtenäistää eri testityökalujen dataa, jolloin samaan tietokantaan on mahdollista tallentaa eri testikehysten tuloksia. Testiajoja voidaan määritellä kuuluvan samaan erään, jolloin koko testi ajo voi kuulua esimerkiksi Mocha-yksikkötesteistä ja Robot Frameworkin suorittamista end-to-end-testeistä. [12.]

TestArchiver on TestManagerin kanssa osa konseptikokonaisuutta, jossa TestArchiver tallentaa dataa tietokantaan ja TestManager visualisoi dataa sekä suorittaa priorisointia.

3.7 Docker-teknologia

Konttiympäristöt ovat yleinen osa modernia ohjelmistokehitystä, jossa arkkitehtuuri perustuu palveluihin. Tämänkaltaisessa sovelluksessa eri palvelut ovat eri konttien sisällä, mikä tekee sovelluksen ylläpitämisestä ja kehittämisestä yksinkertaisempaa. Docker on muodostunut ohjelmistoalan standardiksi konttiratkaisuihin Googlen Kuberneteksen kanssa. Kubernetes hallinnoi podeja, jotka voivat olla esimerkiksi Docker-kontteja.

Konttiratkaisut ovat erityisesti testausvaiheessa todella käytännöllisiä, sillä niiden avulla on mahdollista tunnistaa eri konteissa tapahtuvia virheitä. Koska kontit ovat suljettuja ympäristöjä, voidaan varmistaa, ettei toinen ohjelma vaikuta itse suoritukseen.

Kontteja on mahdollista pitää käynnissä vain tarpeen mukaan ja skaalata helposti. Mikäli tiettyjä palveluita ei käytetä jatkuvasti, voidaan kontti poistaa käytöstä ja heti tarpeen vaatiessa ottaa takaisin käyttöön ja jatkaa siitä, mihin se viimeksi jäi.

Kontit käyttävät Docker Daemonia, joka on isäntäkoneella toimiva sovellus. Daemon jakaa koneen kernelin virtualisoimalla konteille, jotka eivät muutoin pääse käsiksi isäntäkoneeseen. Samoja kontteja voidaan siis suorittaa Linux-, macOS- ja Windows-käyttöjärjestelmillä ilman yhteensopivuusongelmia. [13.]

Konttien rakentamiseen käytetään Dockerfile-tiedostoa, jossa määritellään kontin toiminnot. Esimerkkikoodi 2 luo kontin Ubuntu-käyttöjärjestelmän viimeisimmän version pohjalle ja asentaa valmiudet web-palvelimelle. Kontin rakennusvaiheessa Docker Daemon luo kontista kuvan, Docker Imagen. Kuvan avulla on mahdollista rakentaa kontteja ilman muuta määrittelyä. [13.]

```
FROM ubuntu: latest

RUN apt-get update
RUN apt-get install -y python python-pip wget
RUN pip install Flask

ADD hello.py /home/hello.py

WORKDIR /home
```

Esimerkkikoodi 2. Dockerfile, joka luo Ubuntuun perustuvan web-palvelinkontin.

Docker-Compose on orkestroija konteille. Sen avulla voidaan määritellä monen kontin toimintaa ja rakentaa konttien väleille omia verkkoja sekä määritellä riippuvaisuuksia. [13.]

3.8 Testien ajaminen kehitysputkissa

Kun testaajat kehittävät testejä projektiin, ajetaan testit yleensä paikallisesti ja niiden valmistuttua testit siirretään kehitysputkissa automaattisesti ajettaviksi. Kehitysputkissa ajettavien tulisi olla vakaita ja varmistaa sovelluksen toimivuus integroinnin jälkeen. Näin testitulokset kertoisivat mahdollisimman paljon testattavan järjestelmän laadusta.

Testien ajaminen on hyvin automatisoitu, eivätkä automaatiopalvelimet varaa testejä suorittavia koneita, ennen kuin kehitysputki käynnistyy. Tässä kohtaa suorittavalle koneelle siirretään kaikki tarvittava data ja ohjelmisto ajoa varten. Suosittu ratkaisu on ajaa testit konteissa. Esimerkkikoodissa 3 on esiteltynä TestManagerin testaamiseen käytetty Docker-Compose.

```
version: '3.3'
services:
  front:
    build:
      context: ./front/
    ports:
      - "4000:4000"
  backend:
    build:
      context: ./backend/
    ports:
      - "5432:5432"
  prioritize:
    build:
      context: ./prioritize/
    ports:
      - "4001:4001"
  selenium-hub:
    image: selenium/hub:3.141.59-oxygen
    container_name: selenium-hub
    ports:
      - "4444:4444"
  firefox:
    image: selenium/node-firefox:3.141.59-oxygen
    volumes:
      - /dev/shm:/dev/shm
    depends_on:
      - selenium-hub
    environment:
      - HUB_HOST=selenium-hub
      - HUB_PORT=4444
```

```
chrome:
  image: selenium/node-chrome:3.141.59-oxygen
  volumes:
    - /dev/shm:/dev/shm
  depends_on:
    - selenium-hub
  environment:
    - HUB_HOST=selenium-hub
    - HUB_PORT=4444
robot:
  user: root
  container_name: robot
  build:
    context: ./robot_tests/
    dockerfile: ./Dockerfile
  depends_on:
    - selenium-hub
    - front
  environment:
    - SELENIUM_URL=http://selenium-hub:4444
    - ROBOT_ARGS
  volumes:
    - ./results/logs:/home/robot/test/logs/
    - ./results/screenshots:/home/robot/test/screenshots/
```

Esimerkkikoodi 3. Docker-Compose, joka määrittelee ympäristön, jossa TestManageria testataan Firefox- ja Chrome-selaimilla.

Konttiteknologiaan perustuvissa ratkaisuissa Selenium Grid on yleinen työkalu testien ajamiseen. Selaimia mallintavat kontit rekisteröityvät Selenium Hubille ja valmistautuvat ottamaan tehtäviä vastaan. Tämänkaltainen ratkaisu mahdollistaa testien suorittamisen kahdella eri selaimella helposti. Vaihtoehtoisesti on mahdollista luoda enemmän selainkontteja, joille yksittäinen robottikontti jakaisi testejä. [14.]

4 TestManager-projektin suunnittelu ja toteutus

TestManagerin projektitiimi sai itse valita työkalut ja teknologiat sekä suunnitella arkkitehtuurin itsenäisesti. Tiimi jakoi tehtäviä osaamisen perusteella ja teki backlog-kehityslistan ominaisuuksista, joita projektin eteenpäin vieminen vaati.

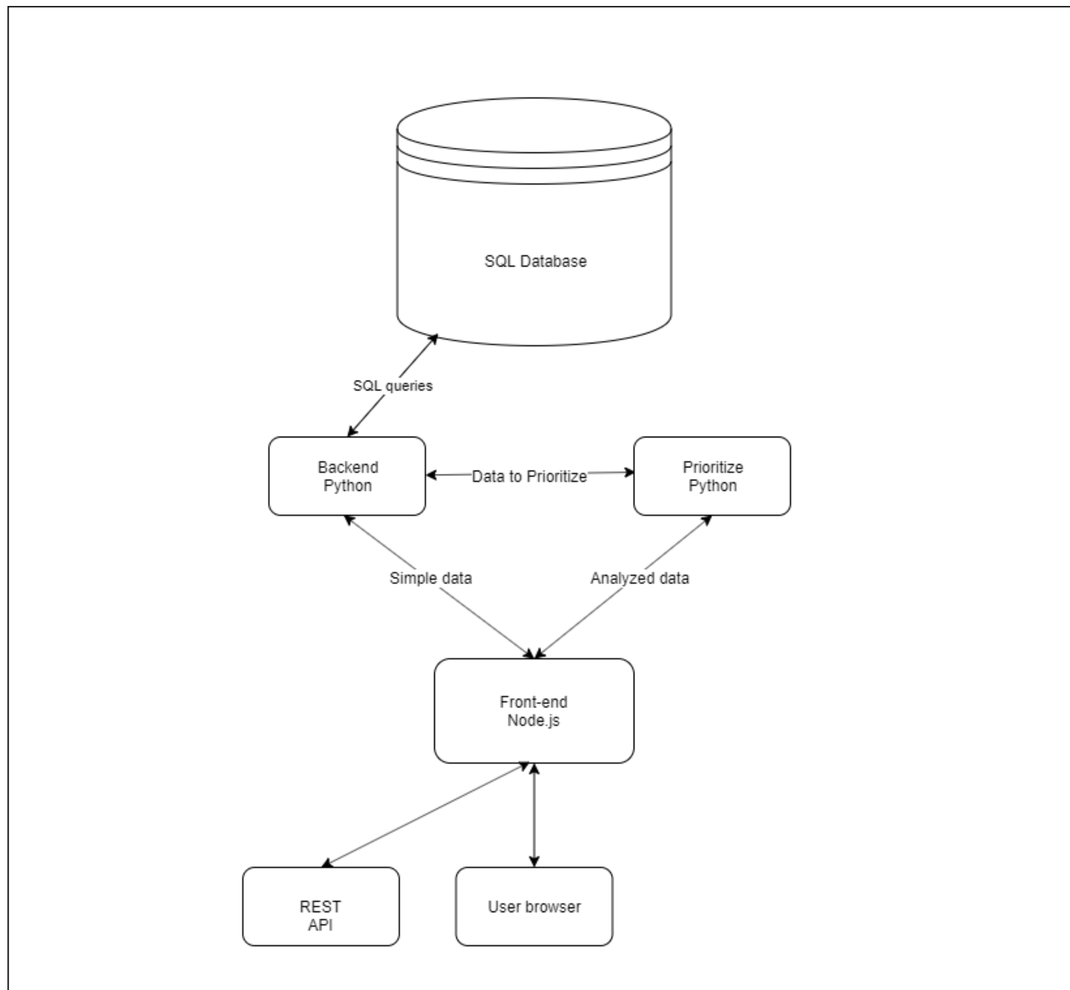
Konseptin liiketoimintamalli ei ollut vahvistunut projektin alkaessa, vaan sen odotettiin tarkentuvan konseptista saadun palautteen perusteella. Tämän vuoksi projektia suunniteltiin niin, ettei se sisältäisi ulkopuolisia moduuleita, jotka vaatisivat lisensoinnin. Lisensoimattomuuden taustalla oli mahdollisuus viedä projektin avoimen lähdekoodin ohjelmaksi tulevaisuudessa.

Projektin alkaessa tehtävien backlogia pidettiin Trellossa, joka on sovellus kanban-taulujen rakentamiseen ja ylläpitämiseen. Kanban-taululle listattiin projektiin liittyviä ominaisuuksia ja tehtäviä eli kortteja. Taululla olevia kortteja oli mahdollista kommentoida ja liittää niihin erilaisia tietoja ja alatehtäviä. Taulun avulla oli helppoa seurata ominaisuuksien kehityksen etenemistä ja pitää työjärjestys selkeänä.

Projektin edetessä projektinhallinta keskitettiin Azuren DevOps-palveluun, johon jo aiemmin oli siirretty versionhallinta ja kehitysputket. Palvelun omat taulut olivat kehittämisen näkökulmasta paremmin projektiin soveltuvia, sillä tauluissa pystyi tekemään viittauksia suoraan versionhallintaan ja julkaisuihin.

4.1 TestManagerin arkkitehtuuri

TestManagerille suunniteltiin modulaarinen microservice-tyyppinen arkkitehtuuri, jossa sovellus koostuu erillisistä palveluista, jotka ovat yhteydessä toisiinsa. Mikäli sovelluksessa syntyy virhetila, vain palvelu, jossa virhe tapahtui, sammuu ja orkestroija osaa uudelleen käynnistää palvelun. Tämä ansiosta käyttäjä ei välttämättä edes huomaa, että esimerkiksi priorisointipalvelu kaatuisi taustalla ja käynnistyisi uudelleen. Kuvassa 2 on esitelty TestManagerin arkkitehtuuria.



Kuva 2. TestManagerin arkkitehtuuri.

TestManagerin kehittäjätiimistä osa työskenteli projektissa vain asiakasprojektien välissä. Myös tämän vuoksi sovellus oli suunniteltava siten, että muiden kehittäjien olisi mahdollisimman helppo jatkaa kehittämistä.

Sovellus on suunniteltu toimimaan eri käyttöjärjestelmillä mahdollisimman pienellä määrällä muita riippuvaisuuksia. Sen lisäksi, että sovellusta voi käyttää suoraan komentoriviltä, sitä voi käyttää konttiympäristöissä.

Jokainen TestManagerin palvelu on myös oma Docker-konttinsa, mikä helpottaa eri toimintojen kehittämistä ja asiakkailla käytössä olevien versioiden päivittämistä.

Front-end-palvelu

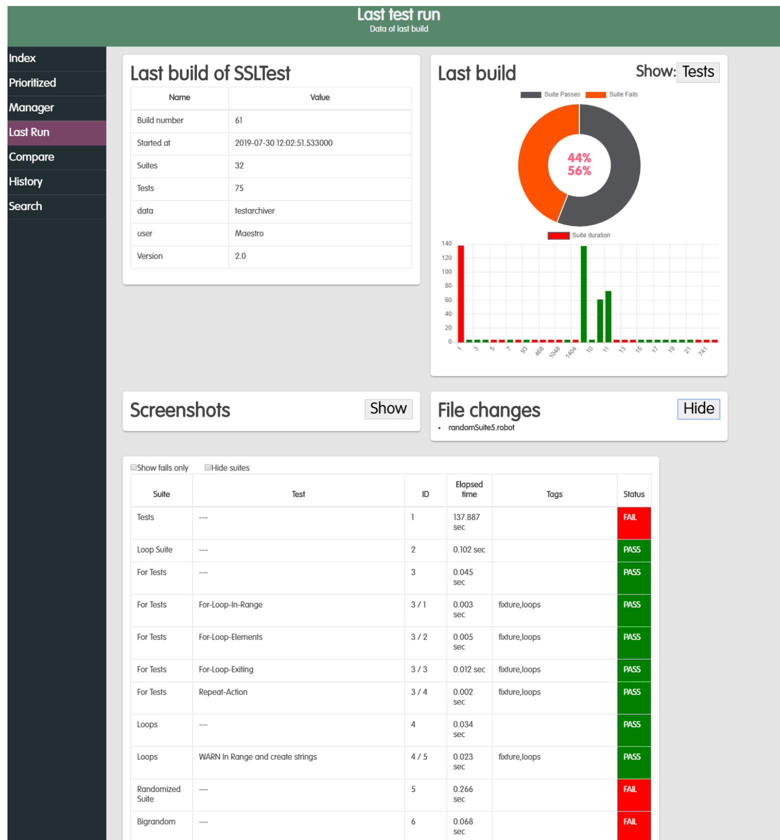
TestManagerin käyttöliittymä on toteutettu Node.js-ympäristönä. Node.js on alustariippumaton avoimen lähdekoodin ympäristö JavaScript-koodin suorittamiseen. Rakenteen ansiosta sovellusta voidaan kehittää samanaikaisesti eri käyttöjärjestelmillä sekä käyttää tuotannossa ilman ohjelmistomuutoksia. Ympäristöön on mahdollista lisätä erilaisia Node-moduuleita, jotka toimivat kirjastoina ja tuovat lisää ominaisuuksia ympäristöön. Käyttöjärjestelmäkohtaiset Node-moduulit asennetaan laitekohtaisesti, sillä eri käyttöjärjestelmille on eroavaisuuksia, kuten järjestelmäpolun ilmaisutapa. Asennus tehdään käskyllä ”npm install”. [15.]

Palvelua käynnistettäessä määritellään, millaisessa ympäristössä sitä tullaan käyttämään ja onko priorisointipalvelu käytössä vai suoritetaanko parsinta koostettuja näkymiä varten Noden palvelinpuolella.

TestManagerin sivut on tehty hyödyntäen Pug-pohjamoottoria, jolle määritellään käytettävä pohja, jonka perusteella se rakentaa HTML-sivuja. Pugin avulla koodirivien määrä pienenee huomattavasti ja tiedostojen luettavuus paranee tavalliseen HTML-syntaksiin nähden. Syntaksin ansiosta usean kehittäjän on helpompi osallistua sivujen kehittämiseen. Huonoja puolia Pugissa on syntaksin tarkkuus välilyöntien ja tyhjän alueen suhteen, jotka määrittelevät syntaksissa elementtien sijainnin hierarkiassa. Syntaksi ei myöskään tue HTML-syntaksia, vaan kaikki elementit on ensin muokattava Pugin mukaiseksi. [16.]

Pug-pohjien avulla sovellus luo käyttäjälle eri näkymiä, kuitenkin hyödyntäen samoja pohjia. Esimerkiksi viimeisen testiajon näkymä perustuu samaan pohjaan kuin yksittäisen ajon näkymä. Taustalla oleva logiikka rakentaa sivuja sen perusteella, mitä dataa se saa itselleen. Mikäli tiettyjä objekteja ei siirry datan mukana, se jättää visualisoimatta niille varatun palan. Esimerkiksi tieto tiedostonmuutoksista tulee näkyville vain, jos kantaan on tallentunut tieto tiedostomuutoksista. Kuvassa 3 esitellään aiemmin mainittuja ominaisuuksia.

Metropolia



Kuva 3. Viimeisimmän testiajon visualisointi TestManagerissa.

Käyttöliittymän kautta on helppoa etsiä testejä ajan perusteella tai järjestyksen perusteella. Ajan perusteella suoritettava haku etsii kannasta hakuehtoja vastaavat testiajot ja valitsee pohjan, jota käytetään tiedon visualisoimiseen, sen perusteella, onko testiajoja yksi vai enemmän. Usean testiajon sisältävä näkymä on esitelty kuvassa 4. Taulukon yläreunassa olevaa ajon järjestysnumeroa painamalla avautuvat tarkasteltavan ajon tiedot käyttäjälle.

Build history
History of recent builds

Index
Prioritized
Manager
Last Run
Compare
History
Search

History of All builds

Change branch
Change length

Hide fully passing
Hide fully failing

Suite	Test	Build 61	Build 60	Build 59	Build 58	Build 57	Build 56	Build 55	Build 54	Build 53	Build 52
		2019-07-30 12:02:51	2019-07-16 14:25:28	2019-07-16 14:21:05	2019-07-03 15:37:14	2019-06-13 13:09:35	2019-06-13 12:57:48	2019-06-13 12:02:35	2019-06-13 11:42:56	2019-06-13 11:27:04	2019-06-13 11:20:39
Tests		FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
Loop Suite		PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS
For Tests		PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS
For Tests	For-Loop-In-Range	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS
For Tests	For-Loop-Elements	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS
For Tests	For-Loop-Exiting	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS
For Tests	Repeat-Action	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS

Kuva 4. Historian visualisointi TestManagerissa.

Käyttöliittymästä on kaksi erilaista versiota, joiden erona on toiseen rakennettu todennuspalvelun implementointi. Testidata on useasti arkaluontoista tietoa sisältävää tietoa, ja siksi kehitettiin versio, jossa käytetään OAuth-tunnistautumista. Mikäli käyttäjä haluaa käyttää versiota ilman tunnistautumista, olisi hyvä rajata sovellus toimimaan sisäverkossa tai muuten estää ulkopuolisilta pääsy dataan.

Backend-palvelu

Backend koostuu kolmesta tasosta, jotka on kirjoitettu Pythonilla. Palvelu käyttää Tornado-nimistä kirjastoa API-rajapinnan toteutukseen. Tornado valittiin palvelinpuolen ohjelmistokehykseksi sen laajojen ominaisuuksien ja erinomaisen projektiin sopivuuden vuoksi. Kehys on ideaalinen erityisesti isoihin ja pitkiin kyselyihin, joita tietokannalle tehdään. [17.]

Ensimmäisellä tasolla palvelu kuuntelee API-kutsuja ja niiden perusteella suorittaa eri funktioita, jotka käyttävät seuraavien tasojen ominaisuuksia. Kolmas taso koostuu puhtaista SQL-kyselyistä, joita toinen taso kerää yhteen niin, että se saa muodostettua ensimmäisen tason kyselyt.

Metropolia

Prioritize-palvelu

Prioritize-palvelu vastaa projektissa datan jalostamisesta ja tallentaa priorisointimallia välimuistiin. Uuden testiajon tullessa tietokantaan palvelu kerää datan ja päivittää sen perusteella priorisointimallia.

Palvelu on kirjoitettu Pythonilla, koska se soveltuu erinomaisesti isojen dataobjektien käsittelyyn ja laskentaan. Toinen syy ohjelmointikielen valintaan oli backend-palvelun kirjoittaminen samalla kielellä, minkä ansiosta tulevaisuudessa yksi kehittäjä pystyisi helpommin jatkamaan molempien palveluiden kehittämistä.

Prioritize toimii API-rajapinnan kautta, joka on toteutettu Tornado-kirjaston avulla. Kutsun tullessa palvelu tarkistaa, onko sillä palautettava data jo valmiina. Mikäli pyyntöön vastaaminen vaatii lisää dataa kannasta, ottaa palvelu yhteyden backend-palveluun ja hakee siltä tarvitsemansa tiedon. Priorisoinnin ohella palvelu vastaa tiedon parsimisesta visualisointia varten. TestManagerin historia- ja vertailunäkymät käyttävät palvelun parsimaa dataa, minkä ansiosta käyttöliittymän takana oleva logiikka on mahdollista pitää yksinkertaisena ja selaimeen lähettävä on noin 15 % datasta, joka ilman priorisointipalvelua siirrettäisiin.

Priorisointimalleja on yksittäisille kehityshaaroille ja koko projektin historialle. Priorisointimallin kattavuutta ja kohdehaaraan liittyviä kyselyitä on mahdollista säätää palvelulle tulevien kyselyjen parametreissä.

Erona muihin palveluihin on se, että muut palvelut eivät ole riippuvaisia priorisoinnista, vaan sovellustasolla ohjelmisto on suunniteltu toimimaan myös ilman priorisointia. Tämä perustuu suunnitelmaan, jossa muut palvelut saattaisivat tulla avoimen lähdekoodin projekteiksi ja priorisointipalvelun saisi käyttöön tilatessaan konsultin Siililtä.

4.2 Testien priorisointi

Ensimmäinen versio priorisoinnista keräsi historiasta testejä, jotka vaihtoivat tilaa useasti tai olivat epäonnistuneet useasti. Lähestymistapa oli kuitenkin liian pelkistetty, ja käyttäjät pystyvät tunnistamaan vastaavia testejä jo pelkällä historianäkymällä.

Seuraavassa versiossa mukaan lisättiin analyysi testien ajankulutuksesta ja ilmoitukset, mikäli suoritusaika on muuttunut mediaaniin nähden. Analyysit ajankäytöstä lasketaan lyhyellä historialla, keskipitkällä historialla ja koko projektin historian pituudella. Pituudet ovat parametrisoituja, ja niitä on mahdollista säätää API-rajapinnan kautta. Prioriteettiin vaikuttavat myös tilanteet, joissa suoritusajat rupeavat nousemaan ilman, että testin tila vaihtuu. Kun ero mediaaniin pienentyy, myös priorisointiarvo laskee ja testin nähdään vakiintuvan. Ajankäytön seuraaminen on erittäin tärkeää, sillä se voi kertoa virheestä testattavassa sovelluksessa, vaikka testin tila ei vaihdu.

Mikäli tietokantaan tallennettu data on Robot Frameworkin tuottamaa, pystyy priorisointipalvelu tunnistamaan mahdolliset muutokset yksittäisen testin valmisteluvaiheesta, suorituksesta ja alasajosta. Seuraamalla näitä tietoja on mahdollista tunnistaa, mikäli testin suoritukseen on tullut muutos ilman, että itse testiin tehdään muutosta. Tämänkaltainen tilanne voi syntyä esimerkiksi testin ulkopuolelta haetun muuttujan muututtua.

Versionhallinasta on mahdollista saada tietoa eri versioiden välillä Git Diff -funktion avulla. Tieto on mahdollista kirjata TestArchiverin metadatakenttään, josta priorisointipalvelu etsii tietoa tiedostomuutoksista. Käyttäjälle jää tehtäväksi määritellä, millä tasolla muutoksia kirjataan. Diff mahdollistaa tiedon tallentamisen tiedostotasolla, funktiotasolla tai rivitasolla. Tason perusteella priorisointipalvelu tekee vertailua testien ja tiedostomuutosten kesken, mikä historian avulla mahdollistaa yhtäläisyyksien tunnistamista. Priorisointipalvelun priorisointimallissa on kerätty tiedot tiedostomuutoksista, jotka vaikuttavat yksittäiseen testiin, sekä tiedoston vaikuttamista testeistä. Tunnistettuja yhtenäisyyksiä on visualisoituna taulukossa 1.

Taulukko 1. Havainnollistus testien ja tiedostomuutosten yhtäläisyyksien yhdistämisestä.

Tiedostomuutos	Tilaa muuttavat testit
Layout.pug	Test Manager View, Test Last Run, Test History, Test Compare, Test Search, Test Prioritized
Manager.pug	Test Manager View
index.js	Test Last Run, Test Search
Server.py	Test History, Test Prioritized, Test Compare
main.css	---

Automaattitestit on voitu suunnitella vaiheistetuksi. Tällöin testit ovat riippuvaisia edellisen vaiheen testistä ja sen tuloksesta. Tämä ei kuitenkaan ole yleisesti hyvä tapa tehdä testejä, koska itse testi ei tässä tapauksessa ole yksiselitteinen, vaan sillä on ulkopuolisia riippuvaisuuksia. Tämän takia priorisointipalvelu on suunniteltu niin, ettei se ota huomioon mahdollisia riippuvaisuuksia muihin testeihin.

Priorisointiarvo

Yksittäisten listojen sijasta pyrittiin suunnittelemaan yksi lista, johon korkean prioriteetin testit tulevat. Prioriteettiarvon lisäksi listaan tallentuvat syyt, miksi testit ovat listalla. Tähän ratkaisuun päädyttiin, kun huomattiin, että käyttäjän on huomattavasti helpompi tutkia yhtä listaa kuin koostaa oma lista neljästä eri listasta. Tämä loi tarpeen kehittää kaava, joka pisteyttää testin huomiontarpeen. Laskenta tapahtuu säikeistettynä erikseen eri pituuksilla ja syillä. Laskentojen valmistuttua jokaisen testin kohdalla lasketaan yhdistetty priorisointiarvo tulosten perusteella. Ominaisuutta esitellään kuvassa 5, jossa on visualisoitu huomiota vaativia testejä.

Prioritize
Prioritized tests

Index
Prioritized
Manager
Last Run
Compare
History
Search

Prioritized list of tests

These tests are marked as critical to be run next.

Name	Fails	Reason	Priorization value
Random Test 2	17/18	Fail amount	7,13
Random Test 8	15/18	Unstable	5,87
Random Test 9	5/12	Not executed in last 6 runs + Unstable	5,1
Increase Sleep 1	6/18	Runtime growing + Unstable	4,11
Master Test	5/18	Filechanges	3,81
Increase Sleep 2	0/18	Runtime growing	3,48
Random Test 4	5/18	Unstable	3,12

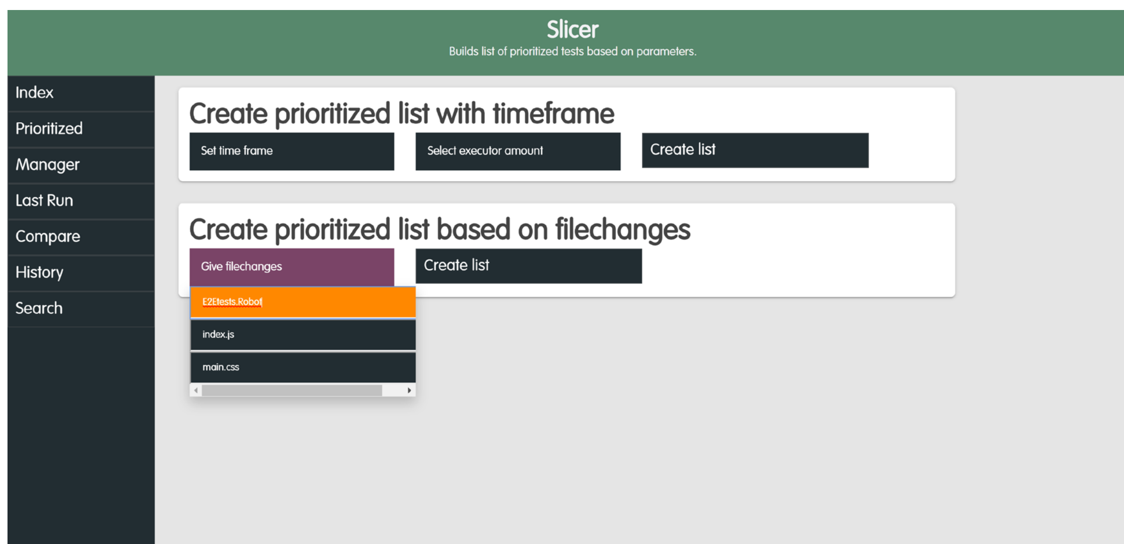
Kuva 5. Esimerkki priorisointipalvelun kehittämästä listasta.

Priorisointiarvoa nostaa myös tieto siitä, ettei testiä ole ajettu hetkeen. Projekteille ominaista on, että testien nimet muuttuvat tai testi poistetaan projektista. Testin nimen muuttuessa priorisoinnin näkökulmasta testistä tulee uusi testi ja vanha jää historiaan.

Slicer-työkalu

Slicer on TestManagerin priorisointipalveluun suunniteltu ominaisuus, jolle on mahdollista antaa tietoja kehityksestä, minkä perusteella työkalu rakentaa listan ajettavista testeistä. Ominaisuus on kehittämisvaiheessa, ja siksi sitä ole toistaiseksi lisätty sovelluksen päähaaraan, jolloin se siirtyisi käytössä oleviin kontteihin.

Suurin työkalun tarjoama hyöty on testiajon keventäminen. Käyttäjä pystyy syöttämään TestManagerille, mihin tiedostoihin on tullut muutoksia, minkä perusteella työkalu käyttää priorisoitua mallia ja laskee, mitkä testit saattavat vaihtaa tilaa muutoksen perusteella. Tämän perusteella käyttäjälle laaditaan lista, jossa on nimetty testit, joita suositellaan ajettavaksi tiedostomuutoksia vasten. Näin on mahdollista saada nopeasti palaute tiedostomuutosten vaikutuksesta kehitettävään sovellukseen. Kuvassa 6 esitellään tiedostonimen syöttämistä tiedostomuutosten listaan.



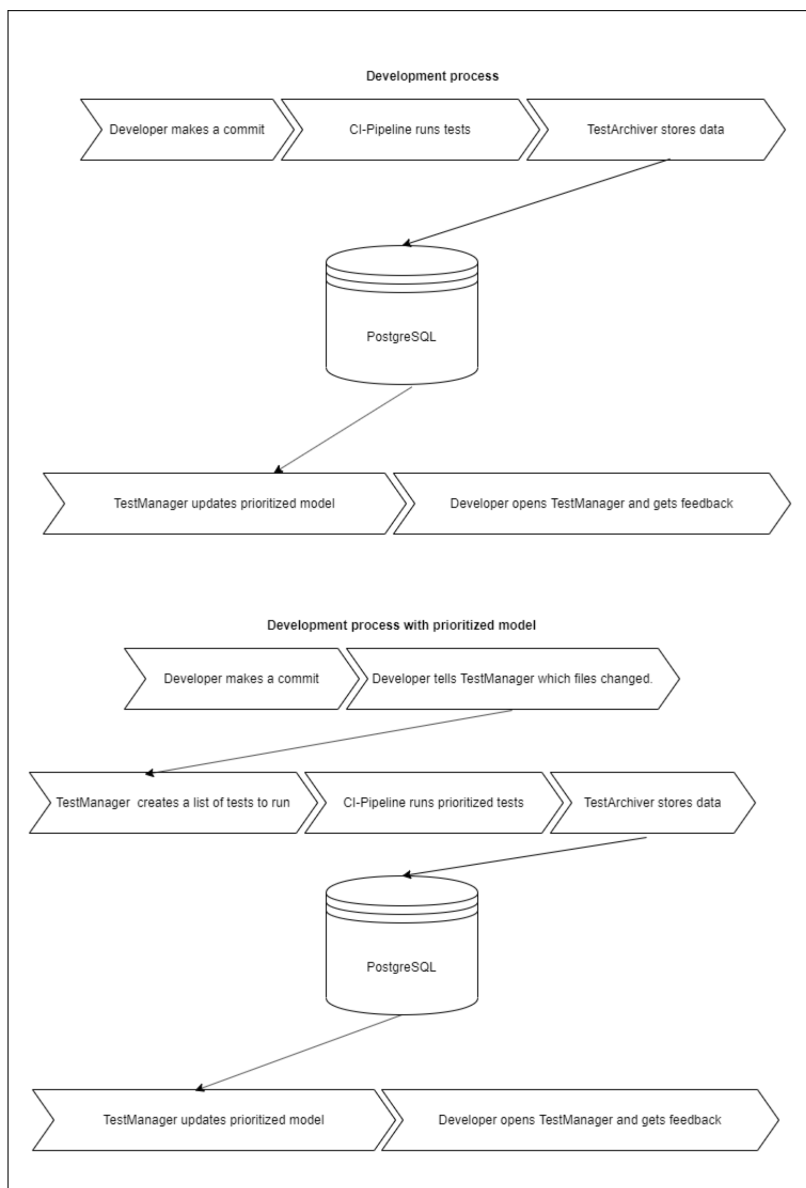
Kuva 6. Slicer-työkalun käyttöliittymä.

Toinen käyttötapaus on listauksen tekeminen aikamääreisiin ja testejä suorittavien koneiden määrään perustuen. Käyttäjä pystyy antamaan parametreinä ajan, jonka sisälle testiajojen on mahduttava, sekä mahdollisten testejä ajavien laitteiden määrän. Esimerkkinä käyttäjä ilmoittaa ajaksi 16.00–08.00 ja koneiden määräksi kaksi. Näin työkalu laatii molemmille koneille listat priorisoiduista testeistä, joiden arvioitu suoritusaika on alle 16 tuntia.

Työkalu on tarkoitettu tehostamaan ja nopeuttamaan kehittämistä, mutta sen ei ole tarkoitus muuttaa, mitä testejä ajetaan. Priorisointimalli päivittyy jokaisella ajolla, minkä takia olisi tärkeää suorittaa testiajoja niin, että kaikki osuudet tulevat testattua. Mikäli ajetut testit vastaavat edellistä priorisointilistaa, merkitsee palvelu ajon priorisointiajoksi ja käyttää tietoa priorisointimallin tarkkuuden analysointiin.

4.3 TestManagerin käyttö

TestManager konseptille suunniteltiin kaksi erilaista käyttötapaa. Toisessa käyttäjä ajaa testit, minkä jälkeen tuloksia arvioidaan ja käyttäjälle annetaan palaute viimeiseen testiajoon perustuen. Toinen vaihtoehto käyttää sovellusta on antaa sille tiedot viimeisimmistä tiedostomuutoksista, minkä perusteella priorisointipalvelu kokoaa listan huomiota vaativista testeistä ja vertailee odotuksia tuloksiin. Käyttötapojen vaiheet esitellään kuvassa 7.



Kuva 7. Esimerkki TestManagerin erilaisista käyttötapauksista.

Kehitysympäristöissä tehtävät testit ajetaan yleensä ominaisuuksia vastaan, jotka ovat jo etukäteen testattuja kehittäjän tietokoneella. Priorisoinnin tarkkuutta nostaa, mikäli kehittäjä ajaa testejä myös tietokoneellaan ja tallentaa tulokset kantaan TestArchiverin avulla. Näin priorisointipalvelun on helpompi tunnistaa yhteydet tiedostojen ja testien välillä.

Metropolia

4.4 TestManagerin oma kehitysputki

TestManagerille rakennettiin aluksi oma kehitysputki GitLabin verkkopalveluun, mutta projektin edetessä päädyttiin siirtämään projekti Azure DevOps -palveluun. Palvelu on huomattavasti monipuolisempi ominaisuuksiltaan ja toimii paremmin yhdessä muiden Azuren pilvipalvelutuotteiden kanssa. Azure DevOpsin käyttäminen myös paransi tiimin osaamista Azuren tuotteista, joiden kysyntä lisääntyy jatkuvasti asiakkaiden puolelta.

Jokainen koodimuutos projektiin käynnistää kehitysputken, joka ajaa sovellukselle testit ja lähettää tulokset sähköpostitse tiimin jäsenille. Master-kehityshaaraan tuleva muutos ajaa myös testit, ja mikäli kaikki testit menevät läpi, rakennetaan jokaisesta palvelusta uudet Docker-kontit ja pusketaan Azure Container Registry (ACR) -palveluun, joka toimii konttien kuvien (Docker Image) tallennusympäristönä. Palvelussa on Webhook-ominaisuus, jonka kautta kontteja käyttävät palvelut saavat tiedon päivitetystä kontin kuvasta. Näin TestManageria käyttävät ympäristöt pystyvät lataamaan päivitetyt kontit ja uudelleen käynnistämään palvelut. Tämä mahdollistaa sen, että käyttäjillä on jatkuvasti uusin versio käytössään.

TestManagerin kehityksestä kerätään myös dataa talteen esimerkkikoodin 4 mukaisesti. Tämä konsoliskripti ajetaan aina testivaiheen valmistuttua. Metadata-vivulla on mahdollista tallentaa tietokantaan lähes mitä tahansa tietoa, jota puolestaan priorisointipalvelu vertailee muihin tuloksiin. Tämän kautta TestManager saa myös tiedostomuutokset analyysia varten. Myös mahdolliset linkit testeistä tallennettuihin kuviin on mahdollista sisällyttää metadatana, jolloin käyttäjäliittymässä TestManager yhdistää kuvankaappaukset suoraan testeihin.

```
echo "Downloading TestArchiver"
git clone https://github.com/salabs/TestArchiver.git
echo "Running TestArchiver"
python3 TestArchiver/test_archiver/output_parser.py \
        --database $DATABASE_NAME \
        --host $DATABASE_HOST \
        --user $DATABASE_USER \
        --pw $DATABASE_PW \
        --dbengine postgresql \
        --metadata "AUTHOR:$LATEST_COMMIT_AUTHOR" \
        --metadata "MESSAGE:$LATEST_COMMIT_MESSAGE" \
        --metadata "changedfiles:${CHANGES}" \
        --metadata "screenshots:${ARTIFACTS}" \
        --metadata "branch:$BUILD_SOURCEBRANCHNAME" \
        --metadata "BuildNumber:$BUILD_BUILDNUMBER" \
        --metadata "BuildReason:$BUILD_REASON" \
        --series "$BUILD_SOURCEBRANCHNAME" \
        ./results/logs/output.xml
```

Esimerkkikoodi 4. Kehitysputkessa tapahtuva tiedon tallennus.

Kehitysputkessa TestManageria testaavat testit ovat kirjoitettu Robot Frameworkilla. Testit ottavat huomioon suoritusympäristön ja sen testataanko autentikoinnilla olevaa versiota vastaan, mikä edellyttää robotilta sisäänkirjautumista. Testeissä on käytetty kehityksessä havaittuja hyviä tapoja tuottaa laadukasta tietoa testeistä, kuten kuvankaappauksien nimeämistä testin mukaisesti. Testit on suunniteltu testaamaan sovelluksen käyttöliittymää ja API-rajapintaa.

Eri kehitysputket toimivat keskenään erilaisilla logiikoilla ja nimeävät muuttujia erilaisilla logiikoilla, minkä vuoksi TestManageria varten on kehitetty tallennusta tekeviä skriptejä kehitysputkia vasten. Tämä nopeuttaa huomattavasti TestManagerin liittämistä uuteen projektiin, vaikka skriptejä on käytännössä pakko räätälöidä jokaiselle projektille erikseen.

5 TestManagerin jatkokehitys ja projektista saatu palaute

TestManager-projektin alkaessa tiimille oli määritelty MVP-toteutus eli vaatimukset minimaalisesta tuotteesta, jolla on mahdollista kerätä käyttäjiltä palautetta ja tutkia markkinoita. Toteutuksen oleellisimpia kohtia olivat historian visualisointi, viimeisen testiajon selkeä esittäminen ja kevyt versio priorisoinnista. MVP-toteutus valmistui elokuussa 2019, minkä jälkeen TestManageriin alettiin kehittää uusia ominaisuuksia, joiden avulla on mahdollista kerätä lisää valinnaista dataa tietokantaan. Ominaisuuksien avulla on

mahdollista luoda tarkemmat priorisointimallit tai esittää käyttäjälle lisätietoa testiajoon liittyen.

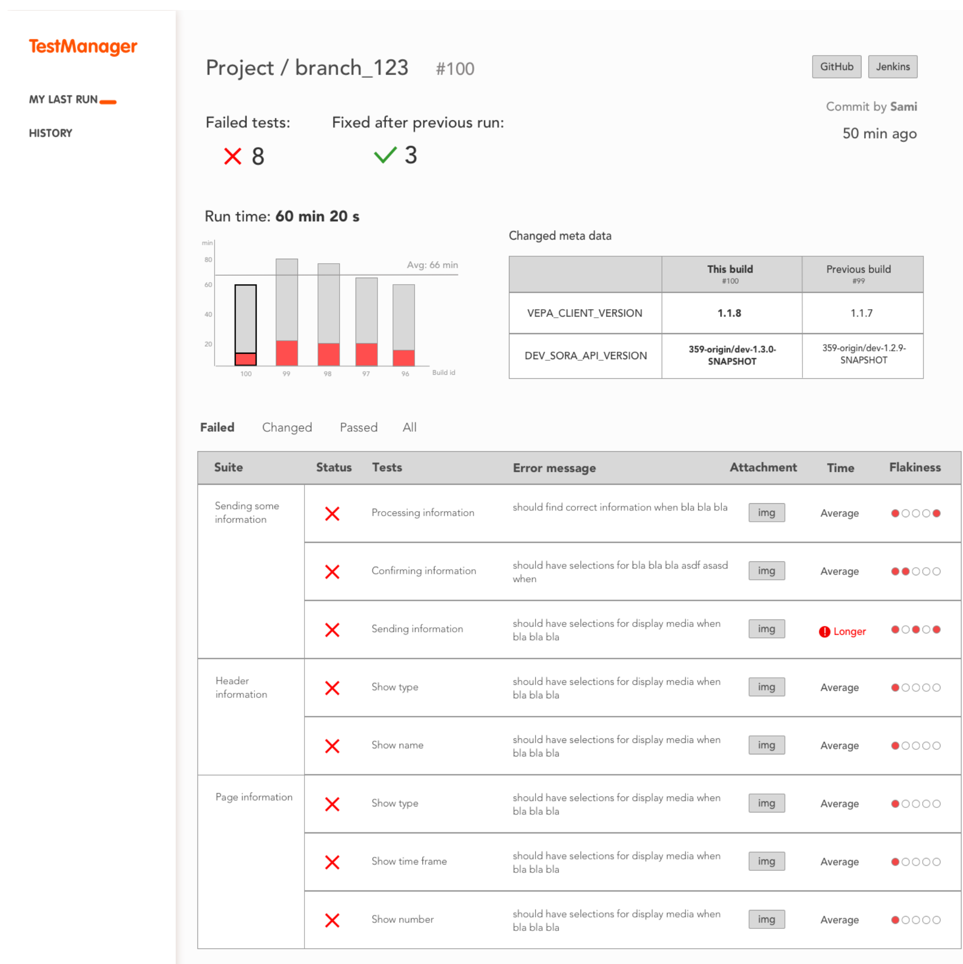
Lokakuussa 2019 pidetyssä kokouksessa TestManagerin kehityksen jatkaminen varmistui ja tiimi sai lisäbudjetin konseptin eteenpäin viemiseen. Kokouksen isoin päätös oli viedä projekti avoimen lähdekoodin sovellukseksi, mikä tarkoittaa, että sovellus tulee ilmaiseksi käytettäväksi ja kuka tahansa pystyy tekemään muutoksia omaan versioonsa. Tässä tapauksessa liiketoimintamalli on TestManageriin liittyvän konsultoinnin ympärillä, kun sovellusta otetaan käyttöön uusissa projekteissa tai sitä on tarvetta muokata asiakkaalle paremmin sopivaksi.

5.1 Uusi käyttöliittymä

SALabs sai toiselta yksiköltä projektiin hetkeksi käyttöliittymäsuunnittelijan, joka keräsi erilaisia käyttäjätarinoita, listasi eri käyttäjien tarpeita ja suunnitteli helposti lähestyttävää ulkoasua.

Suunnittelijan palautteen perusteella käyttäjät jaettiin kolmeen eri ryhmään: kehittäjät, testaajat ja managerit. Jokaisella eri ryhmällä on erilaisia tarpeita sovelluksen käytön suhteen, ja uusittu versio tulee valitsemaan käyttäjälle ensin näytettävät tiedot ryhmän perusteella.

Uusi käyttöliittymä on suunniteltu antamaan palautetta käyttäjälle mahdollisimman pienellä määrällä selailua tai tiedon etsimistä. Näin käyttäjä saisi mahdollisimman paljon hyötyä sovelluksen käyttämisestä. Kuvassa 8 on nähtävissä vedos, jossa näkymä on personoitu kehittäjälle.



Kuva 8. Käyttöliittymäsuunnittelijan tekemä vedos uusitusta ulkoasusta.

Uuden käyttöliittymän toteuttaminen tarvitsee priorisointipalvelusta saatua dataa eikä enää toimi pelkän backend-palvelun kautta. Tämän vuoksi palvelu suunnitellaan kiinteäksi osaksi sovellusta, mutta käyttöliittymä toimii myös ilman sen tarjoamaan dataa.

Uuden käyttöliittymän rakentaminen alkoi lokakuussa 2019, ja tavoitteena on, että se on käytössä tammikuussa 2020 järjestettävässä Robocon-tapahtumassa. Käyttöliittymä tulee jatkossa käyttämään React-ohjelmistokehystä, joka parantaa sivujen responsiivisuutta ja käyttäjäkokemusta.

5.2 Arkkitehtuurin uudistaminen

Saadun palautteen perusteella yksi oleellisimpia muutoksia TestManageriin oli mahdollisuus tukea useaa eri projektia yhden palvelun kautta. Tietokantarakenteet mahdollistavat tämän ominaisuuden, mutta backend-puolen toteutus ei ole kykenevä tekemään tietokantaan kyselyjä, joissa se hakisi tiettyyn projektiin liittyvää dataa.

Backend-palvelussa huomattiin virhe, johon sovelluksessa ei ollut varauduttu. Mikäli tietokantaan tulleet ajot on syötetty valinnaisella tiedolla ajon järjestysnumerosta ja jokin numero puuttuu välistä, kaatuu käyttöliittymä tarkistaessaan mahdollisia ajoja. Virhe korjataan niin, että backend palauttaa tyhjän objektin, mikäli numero puuttuu välistä.

TestManager oli suunniteltu pelkästään esittämään ja jalostamaan dataa, minkä vuoksi suunnittelussa vältettiin ominaisuuksia, jotka vaatisivat tietokantaan kirjoittamista. Taustalla oli visio pitää TestManagerin käyttäminen mahdollisimman yksinkertaisena. Käyttäjälle personoitu näkymä kuitenkin vaatii käyttäjän tunnistamisen ja siten TestManagerin ylläpitoa. TestManagerin on siis jatkossa tallennettava omaa dataansa tietokantaan, ja sinne priorisoitujen mallien tallentaminen olisi luonteva vaihtoehto.

Arkkitehtuurisesti suurimpia uudistuksia tulee olemaan mahdollinen backend-palvelun integroiminen suoraan tietokantaan, jolloin kyselyt lähtisivät suoraan kannalle, missä ne ovat erilaisina näkyminä ja funktioina. Tämän toteutusmallin hyviä puolia on se, että kutsuista tulee nopeampia suorittaa ja tietokantaan rakennetuilla funktioilla ei ole riippuvaisuuksia.

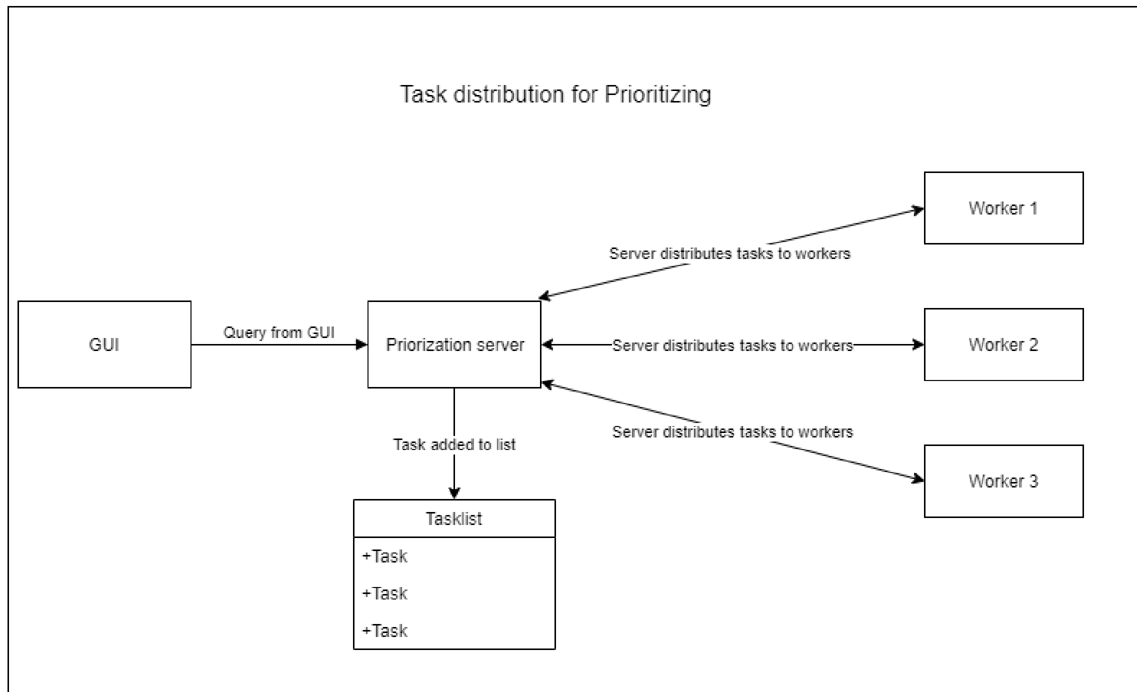
5.3 Priorisoinnin laaja implementointi

Priorisoidun mallin laatua olisi mahdollista parantaa tunnistamalla itse testattavasta järjestelmästä johtumattomista syistä aiheutuvia testien epäonnistumisia. Testi voivat epäonnistua esimerkiksi kolmannen osapuolen tuottaman palvelun käyttökatkoksesta. Tämänkaltainen testitulos voi aiheuttaa puolestaan priorisoinnin puolella vääränkaltaisia oletuksia yhtäläisyyksistä. TestManageriin on rakennettu mahdollisuus piilottaa tuloksista tietty testiajo API-rajapinnan kautta, mutta mahdollisten käyttäjävirheiden takia sitä

ei ole implementoitu käyttöliittymään. Mahdollinen ratkaisu olisi teettää testattaviin sovelluksiin ensin savutestit, joista ilmenee toiminnallisuuksien ja palveluiden tila.

Priorisoinnin käytön lisääminen kehitysympäristöön on myös mahdollista automatisoida, mikä poistaisi käyttäjältä tarpeen syöttää TestManageriin tietoa tiedostomuutoksista ja kehitysputki keskustelisi suoraan TestManagerin kanssa siitä, mitkä testit olisi ajettava tiedostomuutoksia vasten. Tämänkaltainen ratkaisu kuitenkin tarkoittaa kehitysympäristö- ja testisovelluskohtaista ominaisuutta, joten vastaava ominaisuus olisi rakennettava asiakaskohtaisesti.

Projektien edetessä tietokantaan tallentuvan datan määrä lisääntyy jatkuvasti, ja siksi myös priorisointilaskennan suoritusajat kasvavat. Eräällä Siilin asiakkaalla testiautomaation tuottaman datan koko on noin kaksi gigatavua päivässä. Tämän ongelman ratkaisuksi suunniteltiin malli, jossa priorisointipalvelu ei itse suorittaisi priorisointiin liittyviä tehtäviä. Tehtäviä suorittaisivat palvelulle rekisteröityvät palvelimet tai kontit. Näin sovellusta pystyttäisiin skaalaamaan jatkuvasti käyttöasteen mukaisesti. Ominaisuus myös helpottaisi monen projektin tukemista yhden sovellusinstanssin kautta. Tämän ominaisuuden toteuttamisesta ei ole tehty päätöstä, sillä sen toteutus vie runsaasti aikaa ja toiminnallisuudet on suunniteltava huolellisesti. Kuvassa 9 on nähtävissä ominaisuuden suunnittelukaavio.



Kuva 9. Suunnittelukaavio priorisoinnin muuttamiseksi tehtäväperusteiseksi.

5.4 Palaute

TestManagerin kehitykseen pyrittiin saamaan mukaan oikeita projekteja, joiden dataa pystyttiin hyödyntämään kehittämisessä ja keräämään käyttäjäpalautetta TestManagerin käytöstä.

Projektin alkuvaiheilla aloitettiin datan kerääminen toisessa Siilin sisäisessä projektissa, KnoMessa. KnoMe on tietopankki, jonne on listattu jokaisen konsultin työhistoria ja osaaminen, ja tietojen perusteella myyjät etsivät KnoMesta projekteihin sopivia konsultteja. Ensimmäisen käyttöliittymän valmistuttua KnoMen kehittäjät pystyivät käyttämään TestManageria omilta tietokoneiltaan, mutta tiedon keräys tapahtui vasta kehitysputkesta. Kehittäjät testasivat paikallisesti tietokoneillaan ominaisuudet ennen versionhallintaan siirtämistä, mikä johti kaikkien testien läpimenon kehitysputkessa. Tämän takia TestManager näytti kaikkien testien menevän läpi ilman minkäänlaista ongelmaa, minkä takia kehitystiimi ei nähnyt TestManagerille suurta arvoa omassa projektissaan nykyisillä testausmenetelmillään. Projektista saatua dataa ei myöskään voitu hyödyntää kehityksessä, koska lähes kaikki tieto testiajoissa tapahtuneista virheistä puuttui.

Projektin edetessä vaiheeseen, jossa konseptia esiteltiin asiakkaalle, saatiin asiakas näkemään sovelluksen tarjoamat hyödyt ja suostumaan lähtemään mukaan kokeiluprojektiin. Yhteistyökokeilusta saatiin todella arvokasta palautetta, kuten ymmärrys, etteivät kehittäjää ja testaajaa kiinnosta samat asiat testiraporttien suhteen. Kehittäjälle oli tärkeämpää saada vahvistus, että kehityksessä ollut sovellus toimi samoin tai paremmin kuin ennen tiedostomuutoksia. Testaajia puolestaan kiinnosti enemmän itse testiajon analytiikka ja esimerkiksi se, oliko jokin testiajo kestänyt kauemmin kuin aiemmilla kerroilla.

TestManagerin ensiesittely

TestManagerin tiimi esitteli TestManagerin konseptia ja kehitysversiota 19.9.2019 järjestetyssä Robot Framework Meetup -tapahtumassa. Tapahtumaan osallistui 40 Robot Frameworkista kiinnostunutta henkilöä sekä Siilin työntekijöitä.

TestManagerin esityksessä esiteltiin laajasti SALabsin konseptikehystä TestManagerin ympärillä ja TestManager-konseptia. Lisäksi demonstroitiin Azuren pilvipalvelussa toimivalla esimerkkisovelluksella erilaisia ominaisuuksia, joita TestManager tarjoaa käyttäjälle.

Saatu palaute TestManagerista oli positiivista ja herätti kiinnostuksen testiautomaatio-osaajien keskuudessa. Esityksen lopussa järjestetyssä kyselytilaisuudessa yleisölle tuli todella isona yllätyksenä, että modulaarisen rakenteen ansiosta uusi instanssi pystytään perustamaan Azuren pilvipalveluun palvelemaan toista projektia noin viidessätoista minuutissa.

Vala Groupille esittely

Siili Solutions omistaa enemmistön Vala Groupista, joka on DevOps-osaajiin ja testiautomaatioon erikoistunut konsulttiyritys. SALabs tekee runsaasti yhteistyötä Valan kanssa, koska molemmat tarjoavat konsultteja samalle sektorille. Tulevaisuudessa yksiköiden välistä yhteistyötä pyritään kehittämään ja siten lisäämään yksiköiden tunnettuutta automaation erikoisosaajina.

Valalle järjestettiin oma esittelytilaisuus, jossa käytiin läpi SALabsin konseptikehitystä ja erityisesti TestManageria. Tämän perusteella saatiin arvokasta palautetta siitä, minkälaisia ominaisuuksia konseptiin olisi lisättävä ja miten Valan konsultteja saataisiin ottamaan sovellus käyttöön asiakasprojekteissa. Palautteen perusteella konsepti koettiin hyödylliseksi ja kehitystä tehostavaksi elementiksi ohjelmistokehityksessä. Erityisesti kehuja sai visualisointi ja mahdollisuus koostaa usean eri testikehyksen tulokset samaan paikkaan.

6 Yhteenveto

Insinöörityössä perehdyttiin moderniin ohjelmistokehitykseen sekä testiautomaatioon ja sen tehostamiseen erilaisten työkalujen avustuksella. Työssä ilmeni testiautomaation tärkeys ohjelmistokehityksessä ja se, miten ohjelmistokehitystä pystytään tehostamaan automatisoinnin avulla.

Insinöörityössä suunniteltiin testiautomaation tehostamiseen TestManager-sovellus, joka osoittautui onnistuneeksi ja jonka avulla on saatu kerättyä laajasti tietoa testidatan visualisoinnista ja eri menetelmien käyttämisestä priorisoinnin toteuttamiseksi. Myös konseptikehityksen osalta TestManageria voidaan pitää onnistuneena projektina, sillä projekti on tuonut lisää näkyvyyttä yksikölle ja lisännyt kehittäjien osaamista testiautomaation ja ohjelmistokehityksen tehokkaassa yhdistämisessä.

Insinöörityön suorittaminen edellytti laajaa modernin sovelluskehityksen ja jatkuvan integroinnin tehokkaan implementoinnin ymmärtämistä. Työ opetti laajasti erilaisia tapoja käyttää testiautomaatiota ja toi esiin eri lähestymistapoja kehitysympäristöihin. TestManager-projektiin osallistuneet henkilöt ovat projektin myötä oppineet laajasti erilaisista testaustavoista ja mahdollisista käyttökohteista.

TestManager on osoittautunut onnistuneeksi konseptikokeiluksi, ja SALabs on päättänyt viedä konseptia eteenpäin. Sovelluksen ensimmäinen avoimen lähdekoodin versio on tarkoitus julkaista tammikuussa 2020 järjestettävässä Robocon-tapahtumassa. Liitteessä 1 on SALabs-yksikön GitHub-tili, jossa TestManager julkaistaan.

Lähteet

1 Regan, Sean. 2019. Happy customers, quality code: the new trends in software development. Verkkoaineisto. <https://www.atlassian.com/blog/software-teams/modern-software-development-trends>. Luettu 1.11.2019.

2 Melnicki, Mike. 2019. Done is dead – welcome to outcome-driven development. Verkkoaineisto. <https://sdtimes.com/agile/done-dead-welcome-outcome-driven-development/>. Luettu 4.11.2019.

3 Loukides, Mike. 2012. What is DevOps? Verkkoaineisto. <http://radar.oreilly.com/2012/06/what-is-devops.html>. Luettu 4.11.2019.

4 Fowler, Martin. 2006. Continuous Integration. Verkkoaineisto. <https://www.martinfowler.com/articles/continuousIntegration.html>. Luettu 4.11.2019.

5 Hammond, Jeffrey. 2011. The Relationship Between Dev-Ops And Continuous Delivery: A Conversation With Jez Humble Of ThoughtWorks. Verkkoaineisto. <https://go.forrester.com/blogs/11-09-09-the_relationship_between_dev_ops_and_continuous_delivery_a_conversation_with_jez_humble_of_thought/>. Luettu 4.11.2019

6 Zallar, Kerry. 2000. Practical Experience in Automated Software Testing. Verkkoaineisto. <http://www.methodsandtools.com/archive/archive.php?id=33>. Luettu 1.11.2019.

7 McPeak, Alex. 2015. Deciding What to Automate When You Can't Test Everything. Verkkoaineisto. <https://crossbrowsertesting.com/blog/test-automation/prioritizing-test-automation/>. Luettu 30.10.2019.

8 Gärtner, Markus. 2012. ATDD by Example: A Practical Guide to Acceptance Test-Driven Development. Addison-Wesley Professional.

9 Pugh, K. 2011. Lean-Agile Acceptance Test-Driven Development. Boston: Pearson Education, Inc.

10 Laukkanen, Pekka. 2006. Data-Driven and Keyword-Driven Test Automation Frameworks. Diplomityö. Teknillinen Korkeakoulu – Aalto-yliopisto.

11 Robot Framework. 2019. Verkkoaineisto. Robotframework.org. <https://robotframework.org/> Luettu 1.11.2019.

12 Oinonen, Tommi. 2019. SALabs/TestArchiver. Verkkoaineisto. GitHub. <https://github.com/Salabs/TestArchiver> Luettu 1.11.2019.

13 Docker Documentation. 2019. Verkkoaineisto. Docker Documentation. <https://docs.docker.com> Luettu 2.11.2019.

14 Selenium - Web Browser Automation. 2019. Verkkoaineisto. Seleniumhq.org. <https://www.seleniumhq.org/> Luettu 3.11.2019.

15 About Node.js. 2019.Verkkoaineisto. Nodejs.org. <https://nodejs.org/en/about/> Luettu 3.11.2019.

16 Morelli, Brandon. 2017. What is Pug.js (Jade) and How can we use it within a Node.js Web Application? Verkkoaineisto. <https://codeburst.io/what-is-pug-js-jade-and-how-can-we-use-it-within-a-node-js-web-application-69a092d388eb>. Luettu 4.11.2019.

17 Tornado. 2019. Verkkoaineisto. Tornado <https://www.tornadoweb.org/en/stable/>. Luettu 2.11.2019.

SALabs-yksikön GitHub-säilö.

https://github.com/salabs/

Sovelluksen lähdekoodi julkaistaan GitHubissa projektin valmistuttua.