Aake Hänninen

Selvitys palvelinympäristön testauksesta

Insinööri (AMK)

Tieto- ja viestintätekniikka

Kevät 2019

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä(t): Hänninen Aake

Työn nimi: Selvitys palvelinympäristön testauksesta

Tutkintonimike: Insinööri (AMK)

Asiasanat: Testausmenetelmät, testauksen työkalut, palvelinympäristö

Tässä opinnäytetyössä on tutkittu työn toimeksiantajan palvelinympäristöön sopivimpia testauksen työkaluja ja ohjelmia sekä valittu niistä soveltuvimmat vaihtoehdot. Tämän jälkeen kaikista valituista työkaluista on tehty lyhyt ohjeistus testaamisen aloittamiseksi. Lisäksi testausta varten on rakennettu toimeksiantajalle toimitettava testausohje.

Tietotekniikan ohjelmien ja palveluiden kehittyessä niiden testaamisen merkitys on kasvanut ajan myötä suuresti. Erilaisia ohjelmia, palveluita tai järjestelmiä päivittäessä pienetkin muutokset saattavat aiheuttaa lopulta merkittäviä vahinkoja, jotka voivat puolestaan aiheuttaa palveluntarjoajalle suuriakin kustannuksia.

Koska pienetkin virheet voivat aiheuttaa suuria menetyksiä, on testausta varten rakennettu laajasti erilaisia työkaluja. Lähes jokaiselle eri ohjelmointikielelle, käyttöjärjestelmälle ja -ympäristölle on nykyisin vähintään yksi ja usein useampikin testausmenetelmä tai -ympäristö. Koska vaihtoehtoja on useimmiten monta, on tehokasta testausta varten tehtävä aluksi selvitys siitä, mitkä testausmenetelmät tai työkalut ovat sopivimpia juuri haluttuun käyttötarkoitukseen.

Lopulta testaukseen valitut työkalut olivat saman yrityksen kehittämiä kuin itse palvelimen kehitysympäristökin. Syy valinnalle oli työkalujen toimintavarmuus sekä luotettavuus, kuin myös Laravelin tarjoama laaja dokumentaatio. Työkalujen yksinkertaisuuden ja tehokkuuden ansiosta saatiin luotua hyvä pohja testejä sekä ohjeistus testien kehittämiseen jatkossa. Samalla kun palvelin jatkaa kehittymistään, myös testejä tarvitaan lisää.

Abstract

Author(s): Hänninen Aake

Title of the Publication: Investigation of Server Testing

Degree Title: Bachelor of Engineering

Keywords: Testing methods and programs, server

In this Bachelor's thesis the best programs and tools for testing the client's server have been compared to each other, and the best options for the client's situation have been chosen. After this, a short guide has been written on all the chosen tools and programs to get the testing started. Additionally, a guide on testing has been built for the client.

As the programs and services of information technology advance, the importance of testing them has increased greatly over time. When different programs, services or systems are updated, even small changes can cause large problems, which in turn can cause the service provider great losses.

Since even small mistakes can cause great losses, a wide selection of different tools has been built for testing. Nowadays, almost every programming language and operating system has at least one, often multiple, different testing methods and systems. Due to there often being multiple different choices, efficient testing requires a report on which tools and methods would be the best for the wanted purpose.

In the end, the tools chosen for the testing were the ones that are developed by the same company as the server's development environment. The reason for this was their stability and reliability, as well as the extensive documentation provided by Laravel. Thanks to the simplicity and effectiveness of the chosen tools, a good base of tests as well as a guide for making more tests in the future were created. As the server continues to develop, more tests will be needed in the future.

Sisällys

1 Johdanto

Tietotekniikka on ajan myötä tullut entistä yleisemmäksi maailman digitalisoituessa. Pieniä tietokoneita löytyy nykyään lähes kaikkialta, ja teknologia on vielä nykyäänkin yleistymässä. Koska laitteet ovat ihmisten suunnittelemia ja niiden sisältämät käyttöjärjestelmät ja ohjelmat vielä ihmisten tekemiä, saattaa laitteissa ja ohjelmissa esiintyä toisinaan virheitä. Pienet virheet voivat johtaa suuriin riskeihin ja menetyksiin, joten virheiden määrää ja niiden aiheuttamaa riskiä kannattaa vähentää.

Virheitä voidaan etsiä esimerkiksi ohjelmien koodista käsin tai ohjelmointiin käytettävän ohjelman eri työkalujen avulla. Kaikki virheet eivät kuitenkaan tule esille näillä menetelmillä. Laite saattaa olla suunniteltu hyvin ja koodi kirjoitettu kielen ehtojen mukaisesti, mutta ohjelma ei välttämättä toimi kuten haluttua. Lisäksi eri ohjelmien välillä voi olla toiminnallisia virheitä tai laitteisto ei välttämättä täysin tue aiemmin tehtyjä ohjelmia. Tämän takia tarvitaankin testausta.

Testauksella pyritään etsimään toiminnallisia virheitä ajamalla itse koodia tai käyttämällä testattavaa järjestelmää. Testausta voidaan tehdä joko osa-alueittain tai kokonaisuutena, riippuen testien kriteereistä. Esimerkiksi uutta ominaisuutta lisätessä ei vanhoja, jo testattuja komponentteja tarvitse välttämättä testata eristettynä, vaan testataan ensin uusi ominaisuus erikseen, ja sen jälkeen kokonaisuutta uuden ominaisuuden kanssa. Testausta kannattaakin tehdä monessa eri tasossa, jotta virheet saadaan korjattua mahdollisimman aikaisin. Mitä pidemmälle virhe etenee, sen kalliimmaksi sen korjaaminen käy.

Opinnäytetyössäni keskityn testaamisen osalta toimeksiantajan palvelinympäristön testauksen pohjustamiseen. Pääasiallisina vastuinani ovat sopivien testauksen työkalujen ja ohjelmien etsintä sekä testausohjeistuksen tekeminen. Tavoitteena oli saada toimeksiantajalle standardoidut työkalut ja menetelmät sekä valmius testien omatoimiseen rakentamiseen jatkossa.

Testien ajamista ei työssä automatisoitu täysin, vaan testien ajaminen tehdään manuaalisesti toimeksiantajalle tehdyn skriptin avulla. Tulokset tulostetaan konsoliin, josta skripti ajetaan, ja lisäksi tallennetaan paikallisesti tiedostoihin. Vikojen ilmoitusta ei ole standardisoitu, mutta toimeksiantajan tapauksessa tiimin pienen koon vuoksi tätä ei nähty vielä tarpeellisena.

2 Testaus

Testauksella tarkoitetaan tässä tapauksessa ohjelmallista testausta, eli esimerkiksi koodin analysointia ja ohjelman ajamista virheitä etsiessä. Ohjelmallinen testaus voidaan jakaa testauksessa käytettävien menetelmien ja testauksen kohteen mukaisesti erilaisiin testauksen tyyppeihin ja tasoihin.

2.1 Testauksen tyypit

Testauksen tyyppi määräytyy testauksessa käytettävien metodien mukaan neljään pääryhmään. Työni keskittyi kahteen neljästä pääryhmästä, staattiseen ja toiminnalliseen testaukseen. Lisäksi on olemassa suorituskyky- ja turvallisuustestausta, jotka eivät kuitenkaan liity työhöni. [1.]

2.1.1 Staattinen testaus

Staattinen testaus on testausta, jossa ohjelmaa ei missään vaiheessa ajeta. Sen sijaan etsitään virheitä esimerkiksi ohjelman suunnittelusta ja itse koodista. Koodin tutkiminen tehdään yleensä jollain ohjelmalla, ja suurin osa nykyaikaisista ohjelmointiin käytettävistä työkaluista sisältääkin jo itsessään koodin staattisen analysoijan. [2.] Ohjelmointityökalun sisäisellä analysoijalla koodi tarkistetaan jo sitä kirjoittaessa, jolloin ainakin selkeimmät virheet löydetään hyvin nopeasti.

Ohjelmointityökalun analysoija ei kuitenkaan välttämättä huomaa kaikkia virheitä, ja varoitukset vaihtelevatkin runsaasti eri työkalujen välillä. Näin ollen joitain staattisia virheitä saattaa päästä huomaamatta automaattisten analysoijien läpi, ja koodia olisi hyvä silmäillä sekä itse että mahdollisuuksien mukaan pyytää myös toista henkilöä tarkistamaan koodin oikeellisuutta. Näin virheiden mahdollisuus saadaan minimoitua, mutta menetelmä vie myös paljon aikaa.

2.1.2 Toiminnallinen testaus

Toiminnallisessa testauksessa ohjelmaa tai sen osia ajetaan erilaisissa konfiguraatioissa ja ympäristöissä sekä tutkitaan ohjelman suoritusta ja ohjelman palauttamia arvoja. Toiminnallinen testaus voidaan jakaa vielä kahteen alaryhmään, musta laatikko (black box testing)- ja valkoinen laatikko (white box testing) -testauksiin. [3.]

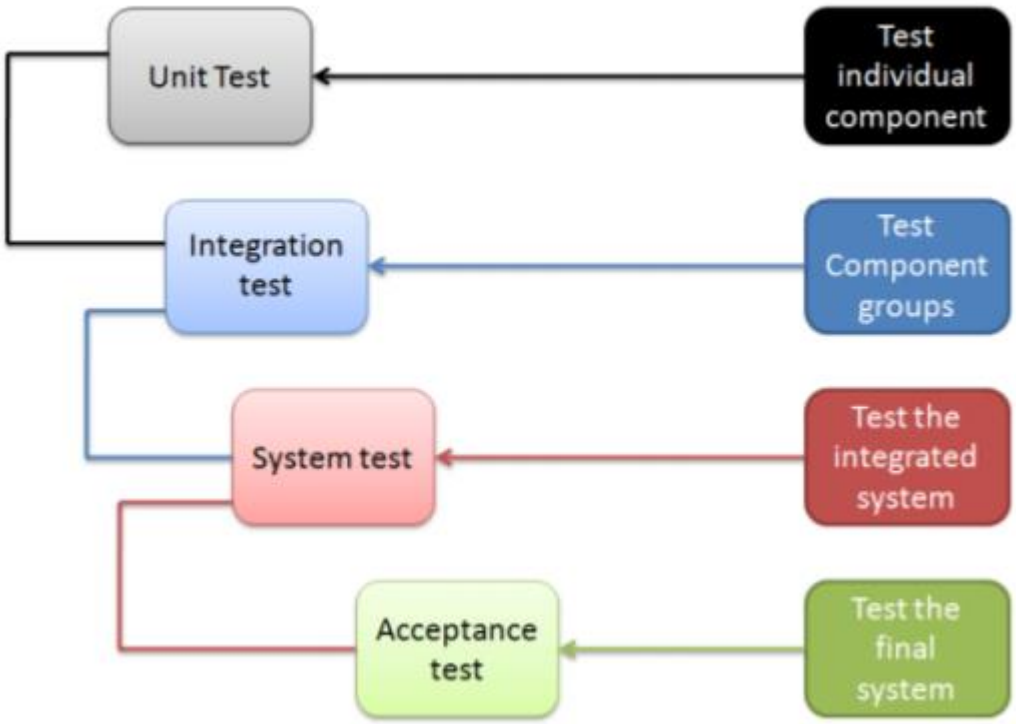
Musta laatikko -testauksella tarkoitetaan tilannetta, jossa toimintoa tai ohjelmaa testataan ilman, että sen sisäistä toimintaa tutkitaan. Testaajan ei tarvitse siis tarkkaan tietää, mitä toiminnon tai ohjelman sisällä tapahtuu. Musta laatikko -testauksissa täytyy kuitenkin tietää, mitä toiminnon tai ohjelman pitäisi antaa palautuksena, tai mihin sen pitäisi vaikuttaa. [4.] Musta laatikko -testauksessa on etuna se, että testit tehdään tällöin käyttäjän näkökulmasta, jolloin testaajana voidaankin käyttää esimerkiksi itse asiakasta.

Valkoinen laatikko -testauksessa testaaja tietää testattavan ohjelman tai toiminnon sisäisen toiminnan. Testaaja voi täten esimerkiksi syöttää arvoja, joiden hän tietää vaikuttavan sisäiseen toimintaan eri tavoin. Tällöin kaikki ohjelman tai toiminnon sisäiset reitit saadaan testattua erikseen, ja testaus on perusteellisempaa kuin musta laatikko -testeissä. [5.] Valkoinen laatikko -testaamisen etuna on juuri perusteellisuus ja se, että ilmenevät viat tai virheet on helpompi paikallistaa.

Etenkin suurempia kokonaisuuksia testatessa saattaa käydä niin, että testaaja tuntee osan ohjelman sisäisestä toiminnasta, esimerkiksi jonkin yksittäisen moduulin toiminnan. Testaus ei tällöin ole täysin valkoisen tai mustan laatikon testauksen mukaista, joten näissä tapauksissa puhutaan harmaa laatikko -testauksesta. [6.]

2.2 Testauksen tasot

Toiminnallinen testaus koostuu useasta eri tasosta, joiden testaamisella on myös omat tarkoituksensa. Yleisesti ottaen eri tasoja ovat yksikkötestaus, integraatiotestaus, järjestelmätestaus ja hyväksyntätestaus [1, 2]. Testauksen eri tasot sekä niiden lyhyet kuvaukset näkyvät kuvassa 1. Tasoista opinnäytetyön kannalta etenkin yksikkö- ja integraatiotestaus olivat tärkeitä. Myös järjestelmätestaus on otettu huomioon, mutta hyväksyntätestausta ei suoritettu erikseen.

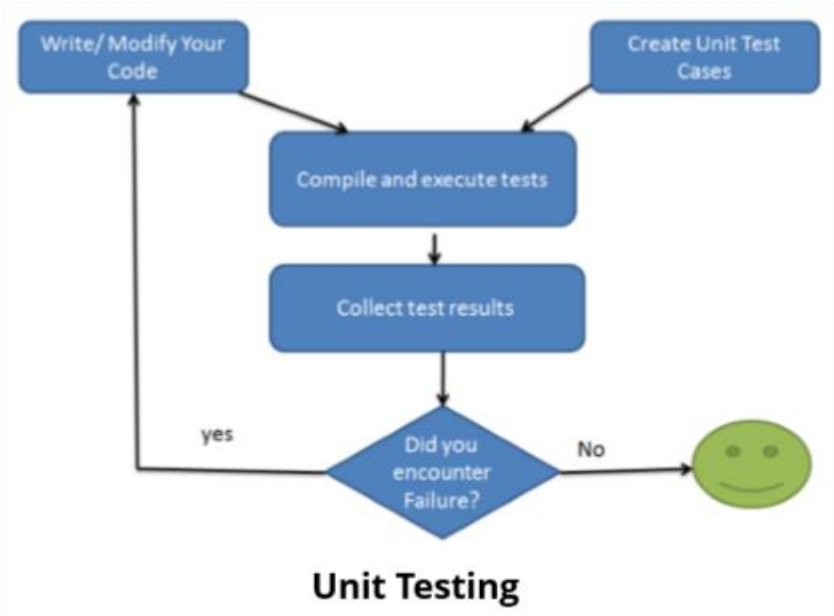


Kuva 1. Testauksen tasot [1, s. 38]

2.2.1 Yksikkötestaus

Yksikkötestaus suoritetaan yleensä kehityksen yhteydessä samaan aikaan, kun testattava kohde on vielä työn alla. Esimerkiksi suuremman ohjelman yksittäistä moduulia voidaan testata jo samalla, kun ohjelmoija vielä tekee sitä. Näin ohjelmoija näkee aina nopeasti, mikäli muutos rikkoo moduulin toiminnan ja varmistutaan siitä, että kaikki lopullisen järjestelmän osat toimivat kuten pitääkin. [2.]

Yksikkötestausta käytettäessä moduulit tulee myös rakennettua itsenäisemmiksi, jolloin eri moduuleita pystytään käyttämään myös muissa käyttötarkoituksissa. Samalla säästetään mahdollisesti aikaa muissa projekteissa, ja moduulien vaikutukset muihin vähenevät, parantaen jälleen toimintavarmuutta. [3.] Yksikkötestauksen eri vaiheet ja eteneminen on esitetty kuvassa 2.



Kuva 2. Yksikkötestauksen vaiheet [1, s. 39]

Yksikkötestaus on yleensä valkoinen laatikko -testausta, sillä testien suorittaja tietää valtaosan ajasta tarkalleen testattavan kohteen sisällön. Yksikkötestaus voi toisinaan olla myös musta laatikko -testausta, mutta ongelmien korjaaminen ja uudelleentestaaminen on tällöin huomattavan paljon hitaampaa.

2.2.2 Integraatiotestaus

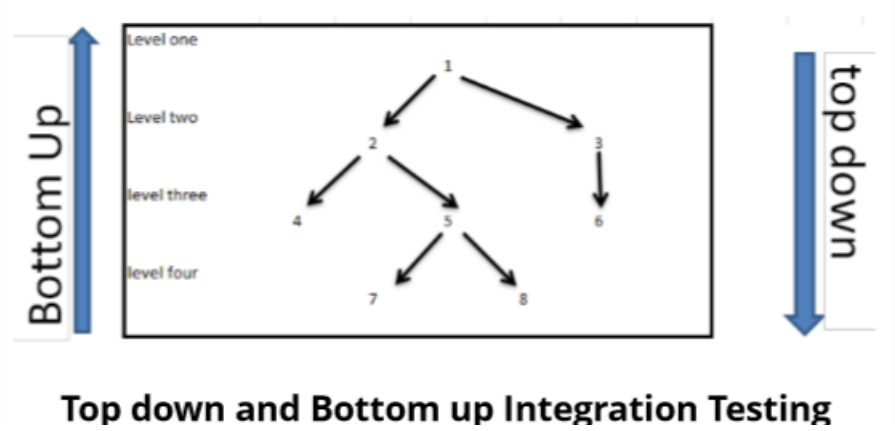
Integraatiotestauksessa yksittäisiä moduuleita yhdistetään ja testataan ryhmittäin. Näin voidaan havaita viat moduulien välisissä yhteyksissä ja kommunikaatiossa. [4.] Integraatiotestien suorittamiseen on kaksi yleisesti käytettyä tapaa, Big Bang sekä kasvava lähestymistapa.

Big Bang -lähestymistavassa kaikki yksittäiset moduulit integroidaan yhdellä kertaa, ja testataan integroitua ryhmää yhdessä. Lähestymistapa toimii etenkin pienissä järjestelmissä, mutta mikäli

vikoja ilmenee, niitä on vaikeampi löytää. [5.] Big Bang -lähestymistapa on huomattavan paljon nopeampi ja vaatii vähemmän esityötä.

Kasvavassa lähestymistavassa integroidaan aluksi enintään muutama keskenään keskusteleva moduuli yhteen ja testataan pientä ryhmää. Sen jälkeen lisätään porrastetusti lisää moduuleita ja testataan aina suurempaa kokonaisuutta. Moduulien lisäämistä jatketaan, kunnes kaikki toisilleen keskustelevat moduulit on integroitu ja testit suoriutuvat virheittä. Näin virheet on helpompi paikallistaa, sillä virheen tiedetään olevan aina juuri lisätyn moduulin ja olemassa olleen ryhmän välillä. Samalla testien suorittaminen kuitenkin hidastuu. [5.]

Integraatiotestausta voidaan suorittaa joko ylhäältä alas tai alhaalta ylös. Ylhäältä alas -menetelmässä ensin testataan korkeimman tason moduulia, ja edetään pienempiin moduuleihin. Haittapuolena tälle menetelmälle on se, että pienempiä moduuleja joudutaan aina simuloimaan niin kutsutuilla jäljitelmillä, tai korvaamaan testiä varten tehdyillä versioilla. Alhaalta ylöspäin -menetelmässä sen sijaan testaus aloitetaan pienemmistä moduuleista ja siirrytään korkeampaa tasoa kohden. Tällöin moduuleja ei jouduta simuloimaan niin paljoa. [1.] Kuvassa 3 on esitetty, millä tavalla integraatiotestaaminen etenee ylhäältä alas ja alhaalta ylös.



Kuva 3. Integraatiotestauksen eteneminen [1, s. 40]

Integraatiotestaus voi olla joko musta laatikko- tai valkoinen laatikko -testausta. Testien järjestystä luodessa täytyy tietää, mikä moduuli kommunikoi minkäkin kanssa, mutta itse testien suorittaja ei välttämättä tarkalleen tiedä ainakaan jokaisen moduulin sisäistä toimintaa. Integraatiotestausta voisikin näin ollen kutsua yleensä harmaaksi testaamiseksi.

2.2.3 Järjestelmätestaus

Järjestelmätestauksessa tarkoituksena on testata kaikkia järjestelmän osia yhdessä ja tarkistaa, että järjestelmä toimii odotusten mukaisesti. Järjestelmätestaus pyritään tekemään mahdollisimman samanlaisella laitteistolla ja ympäristöllä kuin testattavan kohteen lopullinen käyttöympäristö. [6.] Näin saadaan kitkettyä myös ympäristöstä mahdollisesti aiheutuvat virheet. Järjestelmätestauksen onnistuttua on tuote käytännössä jo valmis markkinoille, mutta useimmiten tämän jälkeen järjestetään vielä hyväksyntätestaus.

Järjestelmätestaus on yleensä mustaa testausta, sillä tässä vaiheessa testaajana harvemmin toimii ohjelman kehittäjä. Testaaja ei siis toisin sanoen tiedä yleensä ohjelman sisäistä toimintaa. Mikäli järjestelmätestaaja on ollut mukana kehittämässä ohjelmaa tai tietää sen sisäisestä toiminnasta osittain, on silloin kyseessä harmaa laatikko -testaus.

2.2.4 Hyväksyntätestaus

Hyväksyntätestaus ei ota kantaa niinkään koodissa oleviin virheisiin, vaan siihen, täyttyvätkö esimerkiksi asiakkaan tai käyttäjän vaatimukset. Hyväksyntätestauksen suorittaa yleensä joku muu kuin tuotteen tekijätiimi. Testaaja voi olla joko saman yrityksen työntekijä tai jopa täysin ulkopuolinen henkilö. [7.] Hyväksyntätestaus on yleensä musta laatikko -testausta, sillä testaaja ei tässä tapauksessa todennäköisesti tiedä ohjelman osien sisäistä toimintaa.

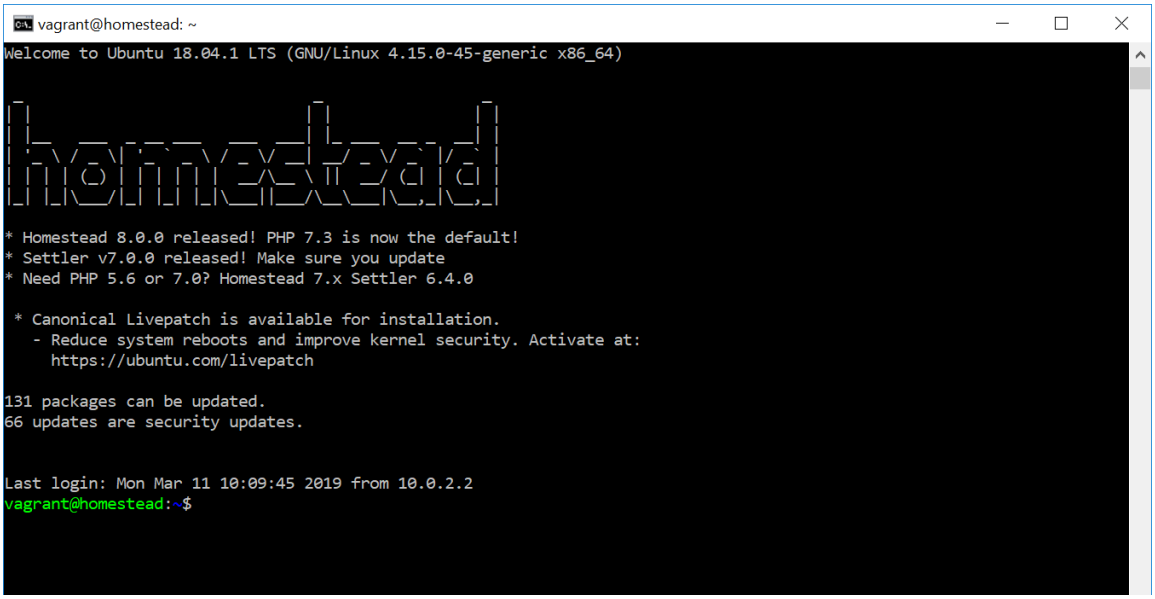
3 Palvelinympäristö

Ensimmäinen suuri osuus itse testausta on ymmärtää palvelinympäristö, työkalut ja ohjelmat, joita halutaan testata. Mikäli näitä asioita ei huomioida tarkkaan, eivät valitut työkalut välttämättä toimi tarkoitetulla tavalla, ja virheitä saattaa jäädä huomaamatta.

Palvelinympäristönä toimii Vagrant-työkalulla asennettava Laravel Homestead -kehitysympäristö. Vagrant on työkalu, joka mahdollistaa valmiiksi pakattujen virtuaalisten kehitysympäristöjen asentamisen helposti. Tällöin ympäristössä on jo valmiina usein käyttöjärjestelmä ja aloitukseen tarvittavat työkalut. [8.]

Laravel Homestead on virtuaalikoneessa toimiva kehitysympäristö, joka on tarkoitettu etenkin PHP kehitykseen. Ympäristön tarkoitus on tehdä PHP-kehityksestä yksinkertaista ja kaikille avointa, ja paketti sisältääkin valmiiksi PHP-ympäristön, Nginx-palvelimen, PostgreSQL-tietokantatyökalun sekä paljon muuta. [9.]

Vagrant-työkalun avulla virtuaalikoneen käynnistäminen sekä sen käyttäminen tapahtuu kätevästi suoraan omasta käyttöjärjestelmästä, esimerkiksi Windowsista. Virtuaalikoneen voi käynnistää käskyllä *vagrant up*, ja sen jälkeen ottaa siihen yhteyden komennolla *vagrant ssh*. Kuvassa 4 näkyy komentokehote SSH-yhteyden käynnistyttyä.



Kuva 4. Laravel / Homestead -hallinta komentokehoitteessa

3.1 Käyttöjärjestelmä

Toimeksiantajan testattavan palvelimen käyttöjärjestelmänä toimii Ubuntu. Ubuntu on Debian GNU/Linux -pohjainen jakelu, jonka kehitys painottuu etupäässä muun muassa palvelinkäyttöön [9]. Ubuntu on käyttöjärjestelmänä helppokäyttöinen, mutta tehokas, jonka seurauksena se onkin yksi suosituimpia käyttöjärjestelmiä [10]. Suosionsa ansiosta Ubuntulle löytyy runsaasti ohjelmia ja työkaluja sekä myös ohjeistuksia, mikäli eteen tulee minkäänlaisia ongelmia.

Ubuntu on täydellinen valinta etenkin pienempien palvelimien käyttöjärjestelmänä juuri suosionsa sekä suunnitellun käyttökohteensa ansiosta. Projektin toimeksiantajan palvelimen Ubuntussa ei ole lainkaan työpöytää, vaan sitä käytetään yksinomaan komentokehotteesta. Komentokehotteeseen pääsee joko avaamalla virtuaalikoneen VirtualBox ohjelmasta tai SSH-yhteydellä Vagrant-työkalulla aiemman *vagrant ssh* -käskyn avulla.

3.2 Verkkopalvelin

Testattava palvelin käyttää verkkopalvelimena Nginxiä, joka on skaalautuva avoimen lähdekoodin WWW- ja proxy-palvelin. Lisäksi Nginx sisältää monia verkkosivujen toimintaa nopeuttavia lisätoimintoja, kuten HTTP-kuormituksen tasaamisen. [11.][12.] Nginx-palvelinta käytetään toimeksiantajan palvelimella esimerkiksi palvelimen hallintaan käytettävän WWW-sivun ylläpitämiseen.

3.3 PHP

PHP eli *PHP: Hypertext Preprocessor* on avoimen lähdekoodin yleiskäyttöinen skriptauskieli, joka on ensisijaisesti tarkoitettu web-kehittämiseen, ja se voidaan upottaa suoraan HTML-koodiin. PHP on kielenä helppokäyttöinen, mutta samalla myös tehokas ja monipuolinen työkalu web-kehittämisessä. [13.] PHP:tä käytetään palvelimella esimerkiksi Nginx-palvelimen ylläpitämän websivun luonnissa ja hallinnassa.

3.4 Tietokanta

PostgreSQL on tietokantojen hallintajärjestelmä, joka käyttää SQL; eli *Structured Query Language* -kieltä, ja lisää kieleen myös uusia toimintoja. PostgreSQL:llä voidaan tallettaa suuria määriä dataa helposti hallittavaan ja turvalliseen tietokantaan. [14.] Palvelimella PostgreSQL:ää käytetään useimmiten juuri suurien datamäärien tallettamiseen ja hallintaan.

3.5 Tietorakenne

PostgreSQL:n lisäksi palvelimella on välimuistia käyttävä tietorakenne Redis, jota käytetään pienemmän datamäärän väliaikaiseen tallettamiseen. Välimuisti on SQL-tietokantaa nopeampi säilytystapa, joten Redis-tietorakennetta käytetään, kun tuloksia halutaan käsitellä nopeammin. [15.] Välimuisti on kuitenkin huomattavan paljon pienempi kuin niin sanottu fyysinen muisti, joten Redis-tietorakenteisiin ei pysty säilömään yhtä paljoa dataa kuin PostgreSQL-tietokantaan.

4 Toimeksiantajan kriteerit

Koska kyseessä on yritykselle tehty toimeksianto, oli työn tilanneella yrityksellä luonnollisesti tiettyjä kriteereitä sekä toiveita niin itse työn etenemisen kuin lopputuloksenkin suhteen. Tässä tapauksessa kriteerit eivät olleet kovin tiukkoja tai tarkalleen määriteltyjä, mutta silti työn aikana huomioonotettavia. Kriteerit rajoittivat etenkin testaukseen käytettävien ohjelmien ja työkalujen valitsemista.

4.1 Aikataulu

Toimeksiantajan vaatimus työn valmistumisesta oli 2019 kevään aikana. Tarkkaa päivää projektin valmistumiselle ei asetettu, mutta arvioitu valmistumisaika asettui projektin alkaessa 2019 huhtikuun alkupuolelle. Aikataulua muutettiin projektin aikana tarvittaessa muiden kiireellisempien töiden suorittamiseksi.

4.2 Hinta

Testauksessa käytettävistä ohjelmista ja työkaluista haluttiin mahdollisimman avoimet, sekä sen myötä edulliset vaihtoehdot. Avointa lähdekoodia sekä edullisuutta painotettiin etenkin tulevaisuuden takia, sillä palvelinympäristössä odotettiin tulevan mahdollisia muutoksia yrityksen laajentuessa, ja sen seurauksena etenkään tässä vaiheessa ei haluttu sitoutua kalliisiin maksullisiin lisensseihin.

4.3 Testausohje

Toimeksiantaja toivoi heille toimitettavasta testausohjeistuksesta mahdollisimman selkeälukuisen, englanninkielisen ohjeen. Ohjeen tuli olla mahdollisimman kattava, sisältäen ohjeistuksen niin testaustyökalujen asentamiseen kuin käyttämiseenkin. Lisäksi ohjeistukseen lisättiin mahdollisia kohdattuja ongelmia, jotta niitä varten on ratkaisut dokumentoituna tulevaisuutta varten.

4.4 Rakennettavat testit

Toimeksiantajalle rakennettiin osana työtä valmiita testejä, jotka ovat valmiita ajettavaksi. Testejä rakennettiin niin yksikkö-, integraatio- kuin järjestelmätestauksiin. Pohjasta haluttiin saada mahdollisimman kattava, mutta pääpainona oli kuitenkin saada esimerkit kaikista kolmesta testauksen tasosta, joita voitaisiin käyttää yhdessä testausohjeen kanssa tulevaisuudessa testien rakentamiseksi.

5 Testausohjelmien ja -työkalujen vaihtoehtoja

Koska palvelinympäristössä on paljon erilaista testattavaa, tarvitaan testauksia varten monipuoliset työkalut. Laravel / Homestead -käyttöympäristö on rakennettu pitäen testaaminen mielessä, ja ympäristön mukana tuleekin niin testaukseen käytettäviä ohjelmia kuin myös esimerkkitestejä.

5.1 PHP

PHP-skriptien toiminnallisuus oli palvelinympäristön laajin ja tärkein testattava ominaisuus. Palvelimen toiminta perustuu lähes täysin PHP-skripteihin, joten on myös tärkeää, että skriptit myös toimivat odotetusti. Koska PHP on skriptauskielenä hyvin suosittu, löytyy myös testaustapoja sekä -ohjelmia paljon. Alla on lueteltu muutamia vaihtoehtoja, joita tutkin oikean työkalun löytämiseksi. Työkaluja on saatavilla monia muitakin, mutta etenkin alla mainituista työkaluista löytyi tietoa monista eri lähteistä.

5.1.1 PHPUnit

PHPUnit on hyvin tunnettu ja laajasti käytetty työkalu PHP-koodin testaamiseen. PHPUnit on avoimen lähdekoodin ohjelma, joten se on myös ilmainen. PHPUnitin heikkona puolena on eri testauksen tasojen puuttuminen, sillä ohjelma ei sisällä lainkaan järjestelmätestauksen mahdollisuutta. [22.] Näin ollen järjestelmätestaukset tulisi suorittaa eri työkalulla. Alla vielä listattuna PHPUnitin tärkeitä ominaisuuksia ja huomioonotettavia asioita.

- Ilmainen, avoimen lähdekoodin ohjelma
- Käyttö komentokehotteesta / terminaalista
- Laaja käyttö ja tuki
- Muokattavat testitulosviestit
- Laravel / Homestead -ympäristössä valmis tuki PHPUnitille
- Ei järjestelmätestausta

5.1.2 Selenium

Selenium on toinen laajalti tunnettu testaustyökalu, jota käytetään etenkin eri testien automatisointiin. Selenium toimii verkkoselaimen avulla käyttämällä selainta normaalin ihmiskäyttäjän tavoin, jolloin myös sivut tulee testattua niin sanotusti oikean käytön kannalta. Samalla tämä kuitenkin tarkoittaa, että koodin tarkempi yksikkö- ja integraatiotestaus jäävät testaamatta. [22.][23.][24.] Alla vielä listattuna Seleniumin tärkeitä sekä huomioonotettavia asioita.

- Ilmainen, avoimen lähdekoodin ohjelma
- Testaus käyttäjän näkökulmasta
- Automaattiset testit
- Ei yksikkö- tai integraatiotestausta
- Korkeampi käyttämisen aloituskynnys

5.1.3 Codeception

Codeception on jälleen maksuton testaustyökalu, joka yhdistää PHPUnitin yksikkötestauksen laajempaan integraatio- ja järjestelmätestaukseen [22]. Codeception onkin rakennettu PHPUnitin päälle, jolloin se sisältää kaikki PHPUnitin normaalit ominaisuudet, sekä sen lisäksi uusia ominaisuuksia integraatio-, järjestelmä- sekä hyväksyntätestauksen lisäämiseksi [25]. Lisäksi Codeception on integroitu toimivaksi Laravelin kanssa, jolloin toiminta toimeksiantajan palvelinympäristössä on varmempaa [23]. Alla vielä Codeceptionin tärkeitä sekä huomioonotettavia asioita.

- Maksuton
- Käyttö komentokehotteesta tai terminaalista
- Kaikki tarvittavat testauksen tasot
- Mahdollista integroida Seleniumin kanssa
- Dokumentointia sekä resursseja heikosti

5.1.4 Storyplayer

Storyplayer on jälleen avoimen lähdekoodin ilmainen ohjelma, jota voidaan käyttää sekä toiminnallisten että ei-toiminnallisten testien automatisointiin [22]. Ohjelma toimii niin sanottujen tarinoiden avulla. Tarinat ovat yleensä PHP-koodilla kirjoitettua testikoodeja, joissa ohjelma etenee tarinan omaisesti. Tarinaan voidaan lisätä toimintoja sekä tarkistuksia eri kohtiin, jolloin testien kirjoittaminen on huomattavan yksinkertaista [26]. Storyplayerissä ei kuitenkaan ole yksikkötestaukselle samanlaisia mahdollisuuksia kuin osalla aiemmista työkaluista. Alla vielä Storyplayerin tärkeitä ja huomioonotettavia asioita.

- Ilmainen, avoimen lähdekoodin ohjelma
- Testien lukeminen helppoa
- Ei yksikkötestausta

5.1.5 Laravel Dusk

Laravel Dusk on Laravelin kehittämä, helppokäyttöinen selainautomaation ja -testauksen työkalu. Työkalu toimii yhdessä PHPUnitin kanssa helpottaen käyttöä Homestead-ympäristössä, jossa PHPUnit on jo valmiiksi asennettuna. Vakiona Laravel Dusk käyttää selaintestien tekemiseen Chromea, mutta tämä on muutettavissa, mikäli testejä halutaan tehdä muulla selaimella. [27.] Alla vielä listattuna myös Laravel Duskin tärkeitä ja huomioonotettavia asioita.

- Ilmainen työkalu
- Laajentaa PHPUnitia järjestelmätesteihin
- Laravelin rakentama
- Laaja dokumentaatio

5.2 Nginx

Nginx WWW-palvelimen toimintaa ei tarvitse testata erikseen, sillä kyseessä on erittäin laajalti käytetty ja luotettava ohjelma. Ainoat heikot kohdat ovat ylläpidettävä sivusto ja Nginx:n konfiguraatio, joiden asetuksia saatetaan joutua muuttamaan. Tätä varten itse Nginx-palvelimesta löytyy komentokehotteessa ajettavia käskyjä, joten erillistä testausohjelmaa ei tarvita.

5.3 PostgreSQL-tietokanta

SQL-tietokannan toiminnallinen testaus tapahtuu yhdessä PHP:n testaamisen kanssa, sillä PHP-testeissä lähetetään API-kutsuja tietokannalle. Tällöin tietokantaa ei kuitenkaan tule testattua yksikkötestauksen tasolla, joten yksikkötestausta varten tulisi olla oma testausohjelma.

SQL-tietokantojen yksikkötestaus on melko harvinaista, mutta testauksesta ei missään nimessä ole haittaakaan. Testauksen harvinaisuuden seurauksena myös testausohjelmia on harvassa, ja ne ovat huonosti tuettuja. Nimenomaan PostgreSQL-tietokantojen testaamiseen löytyy kuitenkin pgTAP-niminen työkalu.

pgTAP on käytännössä laaja keräelmä tietokantafunktioita, joilla tietokantaa pystytään testaamaan yksikkötasolla. Itse testaaminen tapahtuu TAP; eli *Test Anything Protocol* -protokollaa käyttäviä testejä kirjoittamalla. Testit itsessään ovat .sql-tiedostoja, jotka sisältävät erilaisia käskyjä sekä vaihtoehtoisesti valintoja esimerkiksi tietokannan palauttamiseksi testien jälkeen. [28.]

6 Työkalujen ja ohjelmien valitseminen

Testaukseen käytettävien työkalujen ja ohjelmien valinta oli muiden kuin PHP:n testaamiseen käytettävien vaihtoehtojen osalta hyvin suoraviivaista. PHP:n testaustyökaluissa vaihtoehdot karsittiin hyvin tehokkaasti kahteen vaihtoehtoon, jotka olivat Codeception sekä Laravelissa valmiiksi asennettu Laravel Dusk. Koska molemmat pohjautuvat PHPUnitiin, tulee myös sen olla asennettuna, kuten se Homestead-ympäristössä on jo valmiiksi.

6.1 Tietokannat

PostgreSQL-tietokantojen testaamiseen ei löytynyt muuta yhtä kattavaa vaihtoehtoa kuin pgTAP, joten testaamiseen käytettävän työkalun valinta oli tässä tapauksessa helppoa. Lisäksi työkalulle voidaan miettiä vaihtoehtoa myös myöhemmin, sillä tietokantojen testaaminen on tässä vaiheessa jätetty taka-alalle. Tämä priorisointi johtuu pääosin siitä, että tietokannat ovat jo olemassa ja niihin tehtävien muutosten määrät ovat minimaalisia. Lisäksi toiminnallinen testaus tulee PHP-testien kautta, joten huomaamatta jäävien virheiden mahdollisuus vähenee huomattavasti.

6.2 Nginx

Kuten aiemmin mainittiinkin, on Nginx palveluna todettu jo hyvin toimivaksi, ja ainoa heikko kohta ovat ylläpidettävä sivusto sekä mahdollisesti muutetut konfiguraatiot. Nginx-työkalusta löytyy kuitenkin käskyjä, joilla palvelun toimiminen voidaan varmistaa. Myös Nginxin testaaminen on jätetty prioriteeteissa taka-alalle.

6.3 PHP

PHP:n yksikkö- ja integraatiotestaukseen valittiin käytettäväksi PHPUnit, sillä se toimii hyvin yhteen molempien järjestelmätestaukseen käytettävien työkalujen kanssa täydentäen testien kattavuutta. Järjestelmätestauksessa muut vaihtoehdot karsittiin tehokkaasti, mutta Codeceptionin ja Laravel Duskin välillä valinta oli vaikeampi.

Codeceptionin ja Laravel Duskin välillä valintaa vaikeutti se, että molemmat toimivat PHPUnitin kanssa yhteen saumattomasti ja tarjoavat hyvin samantasoisia testaustoimintoja. Pieniä eroja työkaluista toki löytyy, sillä Codeception käyttää esimerkiksi omia versioitaan joistain PHPUnitin tarjoamista ominaisuuksista. Yksi esimerkki näistä on PHPUnitin käyttämä Mockery, jolla voidaan korvata moduuleja yksikkö- ja integraatiotestausta varten.

Toinen huomattava ero on testien käyttämien moduulien konfigurointi. Codeceptionissa konfiguroinnit tehdään erillisten *unit.suite.yml*- ja *functional.suite.yml*-tiedostojen kautta, kun taas Laravelissa / PHPUnitissa moduulit asetetaan itse testitiedostojen alussa. Näin ollen Codeceptionissa moduulien konfigurointi on helpompaa, mikäli suuri osa testeistä käyttää samoja moduuleita, mutta sen sijaan Laravelissa / PHPUnitissa moduulien konfigurointi on joustavampaa testien välillä.

Lopulta päädyttiin valitsemaan Laravelin mukana tulevat työkalut muutamasta eri syystä. Pääsyitä valinnalle olivat työkalujen käyttöönoton helppous sekä erittäin varma toiminta palvelinympäristön kanssa. Lisäksi moduulien konfiguroinnista haluttiin joustavampaa testien välillä, eikä yhdelle yleiskonfiguraatiolle nähty tarvetta.

Laravelin omat työkalut on helpompi ottaa käyttöön, sillä niistä suurin osa tulee valmiina Laravelin kanssa jo asennettuna. Ainoa asennuksen vaativa käytetty työkalu oli Laravel Dusk, jota käytetään selaustesteihin. Tämä vaatii erillisen asennuksen, sillä Duskia ei suositella käytettävän tuotantoympäristössä, johtuen selainta käyttävien testausten mahdollisista tietoturvariskeistä. Laravel Duskin asentaminen on kuitenkin erittäin helppoa.

Helppokäyttöisyyteen vaikuttaa lisäksi se, että Laravelin työkaluilla rakennettavat testit ovat rakenteeltaan samankaltaisia kuin PHPUnitin testit, ja työkalut lisäävät pääasiassa lisätoimintoja PHPUnitin vakiotesteihin.

Myös toimintavarmuuden katsottiin olevan Laravelin työkaluilla parempi, sillä Laravelin mahdolliset muutokset on tällöin otettu myös testauksen työkaluissa huomioon nopeammin ja helpommin. Sen sijaan Codeceptionissa muutokset saattavat tulla myöhässä Laravelin päivittyessä, minkä katsottiin olevan mahdollinen riski tulevaisuudessa.

7 Testit ja niiden rakentaminen

Suurin osa testausympäristön rakentamista on yleensä itse testien rakentaminen. Tämä pitää paikkansa etenkin tilanteessa, jossa palvelinympäristö on jo rakennettu ja siihen on lisätty jälkikäteen lisää ominaisuuksia.

Paras vaihtoehto testien rakentamiselle olisi tehdä testi joko ennen itse koodin kirjoittamista tai ainakin mahdollisimman pian sen jälkeen. Tämä johtuu siitä, että olisi parempi määritellä testin perusteella, mitä koodin pitäisi tehdä, eikä toisinpäin. Mikäli testit rakennetaan pitkän aikaa koodin jälkeen, etenkin jos koodilla on jo monia siitä riippuvaisia muita koodeja, joudutaan testejä helposti muuttamaan koodiin sopivaksi. Tällöin tulee eteen tilanteita, joissa lopulliset testit eivät välttämättä täysin vastaa niille aluksi annettuja määritelmiä, vaan testejä on jouduttu muokkaamaan, jotta koodi läpäisisi sen.

Yksi hyvä esimerkki on tilanne, jossa jo tuotannossa olevassa koodissa on kirjoitusvirhe. Mikäli monet muut koodit tai ominaisuudet riippuvat kirjoitusvirheen sisältävästä koodista, saattaa kirjoitusvirheen korjaaminen aiheuttaa joidenkin ominaisuuksien hajoamisen. Tällöin voidaan joko alkaa korjaamaan kaikki muutkin ominaisuudet, joilla saattaa edelleen olla lisää riippuvuuksia, tai kirjoitusvirhe voidaan kiertää testissä. Näistä tilanteessa helpompi vaihtoehto on testin muuttaminen, mutta samalla vältetään tarkoituksella koodissa olevan virheen korjaamista.

7.1 pgTAP

pgTAP-työkalulle testien kirjoittaminen on hyvin yksinkertaista, mikäli testien kirjoittajalla on aiempaa kokemusta SQL-käskyistä. pgTAPin testit kirjoitetaan SQL-kielellä käyttämällä sekä val-

miina SQL:stä löytyviä että pgTAPin lisäämiä SQL-käskyjä. Työkalu vaatii kuitenkin ensin asennuksen tietokantaan, ennen kuin sen antamia toimintoja voidaan käyttää. Asennus tapahtuu seuraavasti:

- Lataa pgTAP-tiedostot sekä siirrä ne palvelimelle.
- Siirry pgTAPin kansioon:
    - cd pgtap
- Aja seuraavat käskyt järjestyksessä:
    - make
    - make install
    - make installcheck

Tämän jälkeen voidaan asentaa pg_prove-lisäosa, mikäli testit halutaan ajaa sen avulla. Asennus tapahtuu yksinkertaisesti käskyllä:

- cpan TAP::Parser::SourceHandler::pgTAP

Kun pgTAP ja pg_prove on asennettu, lisätään pgTAP vielä tietokantaan seuraavasti:

- Käynnistä psql-työkalu:
    - sudo -u postgres psql
- Yhdistä testaukseen käytettävään tietokantaan, tässä tapauksessa testing:
    - \c testing
- Luo tietokantaan pgTAP-laajennus:
    - CREATE EXTENSION IF NOT EXISTS pgtap;

Tietokantaan lisäämisen jälkeen pgTAP on käytettävissä tietokannan testaamiseksi.

Kuvassa 5 näkyy lyhyt, hyvin yksinkertainen pgTAP-testi, jossa tarkistetaan, että palvelimen tietokannasta löytyy taulukko *users*, *languages* ja *permissions* ja että näistä taulukoista löytyy sarake *id*. Lisäksi testi tarkistaa, että *id*-sarake on kyseisen taulukon *primary key* eli pääavain. Pääavain

on taulukossa sarake, jossa jokaisen rivin tai arvon täytyy olla uniikki. Pääavaimella voidaan helposti hakea tietty rivi, mikäli muut sarakkeet sisältävät duplikaattiarvoja.

```
-- Start transaction
BEGIN;

-- Plan the tests.
SELECT plan(9);

-- Run the tests.
SELECT has_table( 'users', 'Check that schema has table ''users'' with column ''ID'' as primary key' );
SELECT has_column( 'users', 'id', '' );
SELECT col_is_pk( 'users', 'id', '' );

SELECT has_table( 'languages', 'Check that schema has table ''languages'' with column ''ID'' as primary key' );
SELECT has_column( 'languages', 'id', '' );
SELECT col_is_pk( 'languages', 'id', '' );

SELECT has_table( 'permissions', 'Check that schema has table ''permissions'' with column ''ID'' as primary key' );
SELECT has_column( 'permissions', 'id', '' );
SELECT col_is_pk( 'permissions', 'id', '' );

-- Finish and clean up
SELECT * FROM finish();
ROLLBACK;

```

Kuva 5. Yksinkertainen pgTAP-testi

pgTAPissa testien tulostusta on mahdollista muokata tarkoitukseen sopivammaksi. Tulostuksen voi joko jäsentää antamalla testitiedoston alussa testille asetuksia, kuten *\pset format unaligned*, tai käyttämällä valmista jäsennystä. Toimeksiantajan tapauksessa ulostuloa ei lähdetty jäsentämään käsin, vaan testien jäsennykseen käytettiin pgTAPin omaa pg_prove-komentoa. Komennolla ulostulo saadaan helposti luettavaksi, ja testit pystytään ajamaan kätevästi yhdellä käskyllä. [28.]

Jäsennysvalintojen jälkeen testin alkamiskohta merkitään käskyllä *BEGIN*. Tämän jälkeen annetaan erillisten testien määrä käskyllä *SELECT plan(x)*, jossa x:n tilalle merkitään testien määrä. Kuvan 5 tapauksessa jokainen *SELECT*-käsky on oma testinsä, joten testien määräksi merkitään yhdeksän.

Itse testit ovat siis SQL-käskyjä, kuten *SELECT, INSERT* tai *RETURN*. Esimerkiksi *SELECT*-käsky valitsee sille annettujen argumenttien perusteella tietokannasta arvoja. Kuvassa 5 esimerkkinä käsky *SELECT has_table ('users')* valitsee tietokannasta taulukon, jonka nimi on *users*. Mikäli kyseistä taulukkoa ei löydy, käsky ilmoittaa, ettei kyseistä dataa löydy.

SQL-käskyjen lisäksi pgTAP-työkalussa on valmiita funktioita, jotka helpottavat testien rakentamista. Myös yllä mainittu *has_table()* on yksi pgTAPin valmiita funktioita. Funktio tarkistaa sille annettujen argumenttien mukaisesti, löytääkö se haluttua taulukkoa, ja palauttaa SQL-käskyn luettavissa olevaa dataa. Esimerkiksi *has_table()*-käskylle voidaan antaa argumentteja seuraavalla tavalla [28]:

```
SELECT has_table ( :schema, :table, :description );
```

Valinnoista voidaan jättää pois kaikki muut argumentit paitsi *:table* eli taulukon nimi. Muut argumentit ovat *:schema*, eli skeema, sekä *:description*, eli kuvaus. Skeema on SQL-tietokannan osa, johon haluttu taulukko kuuluu, ja kuvaus on vaihtoehtoinen tulostettava kuvaus testille. Kuvan 5 käskyssä esimerkiksi ei ole lainkaan *:schema*-argumenttia, sillä se olisi toimeksiantajan tapauksessa tarpeeton.

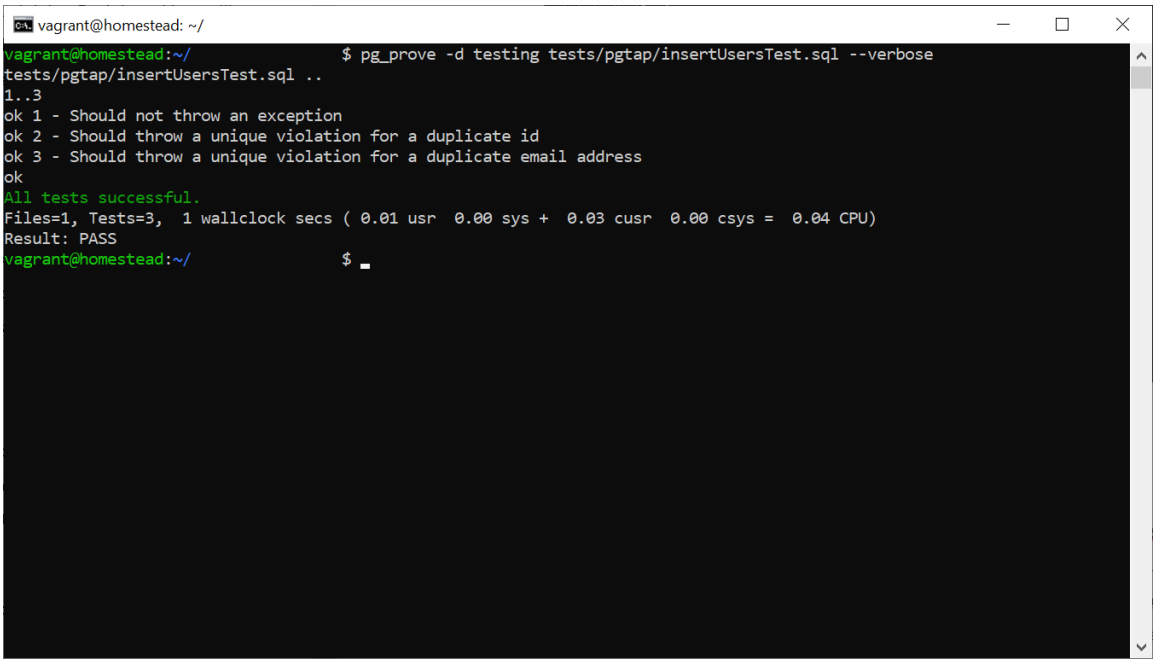
Testien jälkeen niiden päättyminen ilmoitetaan käskyllä *SELECT * FROM finish()*, jolloin työkalu näyttää testien tulokset. Tämän jälkeen voidaan vielä vaihtoehtoisesti asettaa ehto *ROLLBACK*, joka kumoaa kaikki testien aikana tehdyt muutokset. Tämä on kätevää etenkin, mikäli testataan datan asettamista tietokantaan. Testien aikana tietokantaan laitettua dataa ei välttämättä haluta säilyttää testien jälkeen.

Testin rakentamisen jälkeen testit voidaan ajaa oheisella käskyllä:

```
pg_prove -d *TESTITIETOKANTA* “POLKU TESTIKANSIOON”/*.sql
```

Käskyssä *TESTITIETOKANTA* tilalle tulee kirjoittaa testaukseen käytettävän tietokannan nimi, esimerkiksi *testing*. “POLKU TESTIKANSIOON” -tekstin tilalle tulee kirjoittaa polku kansioon, joka sisältää ajettavat testit. Polku tulee kirjoittaa ilman lainausmerkkejä.

Testit kannattaa kerätä mahdollisimman hyvin kasaan, ja esimerkiksi toimeksiantajan tapauksessa polku oli yksinkertaisesti *tests/pgtap/*. Yllä olevaan käskyyn voi lisätä perään myös vaihtoehdon *–verbose*, joka tulostaa testin tulokset sekä kuvaukset. Kuvassa 6 näkyy tuloste testistä, joka sijoittaa käyttäjät (*users*) -taulukkoon tietoja käskyllä *INSERT*. Testin tarkoituksena on varmistaa, että taulukkoon voi kirjoittaa ja että taulukko palauttaa virheen, mikäli sinne koetetaan asettaa id tai sähköpostiosoite, joka on jo olemassa toisella rivillä.



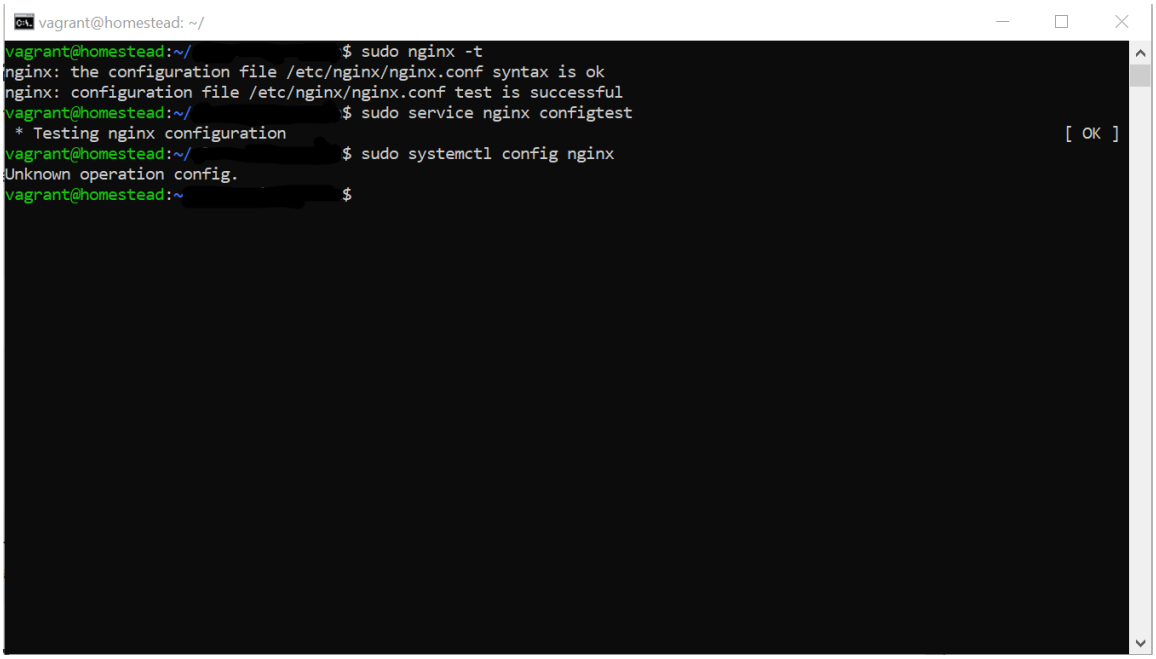
Kuva 6. insertUsersTest-testin tuloste

7.2 Nginx

Ensimmäisenä Nginx-palvelinta tarkistaessa kannattaa varmistaa se, että Nginx on taustalla päällä. Tämä käy käskyllä *service nginx status*, tai vaihtoehtoisesti *systemctl status nginx*. Käskyjen toimiminen saattaa vaihdella käyttöjärjestelmien kesken, mutta esimerkiksi Ubuntussa molemmat toimivat moitteetta. Käskyt palauttavat palvelun tiedot, ja kohdassa *Active:* lukee palvelun tilan mukaisesti joko *active (running)*, mikäli palvelu on päällä, tai *inactive (dead)*, mikäli palvelu ei ole päällä. Lisäksi käskyllä saa kätevästi tietoa palvelun viimeisimmistä tilanmuutoksista, kuten siitä, milloin palvelu on sammunut tai käynnistynyt.

Konfiguraatiota muuttaessa pystyy uuden konfiguraation oikeellisuuden tarkistamaan kätevästi useammalla eri käskyllä. Yksinkertaisin näistä on *nginx -t*, joka tarkistaa aluksi konfiguraation syntaksin ja sen jälkeen testaa konfiguraation Nginxin omalla testillä. Vaihtoehtoisesti konfiguraation testaamiseen voi käyttää myös käskyjä *service nginx configtest* tai *systemctl config nginx*. Nämä ajavat vain Nginxin testin, eivätkä tarkista konfiguraation syntaksia.

Kuvassa 7 näkyy kaikki konfiguraation testaamiseen käytettävät menetelmät Ubuntu-käyttöjärjestelmässä ajettuna. Käskyjen toiminta voi vaihdella käyttöjärjestelmittäin, esimerkiksi tässä tapauksessa viimeisin käskyistä eli *systemctl config nginx* ei toiminut.



Kuva 7. Nginx-konfiguraation testaus

7.3 PHP

Koska toimeksiantajan palvelinympäristö perustuu suurelta osin PHP-kieliseen koodiin, tulee myös suurin osa testeistä keskittymään niihin. Testejä priorisoidessa PHP-testit menivät myös muiden edelle, sillä PHP-koodit ovat sekä tärkein että todennäköisimmin vikoja aiheuttava osuus koko palvelinympäristöä. PHP on myös testattavista osuuksista ainoa, jossa yksikkö-, integraatio- ja järjestelmätestaukset ovat selkeästi erillään.

PHP:n testaamiseen jokaisella tasolla liittyy muutama tärkeä työkalu, kuten tehtaat (*factory*) sekä niin kutsutut jäljitelmät (*mock*), joita ilman testaaminen olisi huomattavan paljon työläämpää. Koska nämä työkalut ovat tärkeitä testaamisen tasosta riippumatta, kannattaa ne myös opetella hyvin ennen testien kirjoittamista.

7.3.1 Tehtaat

Tehtailla tarkoitetaan PHPUnitissa ja Laravelissa koodia, jota käytetään luomaan tietokantaan sisältöä. Näin tietokantaa käyttävissä testeissä voidaan luoda testin ajaksi tietokantaan dataa ja tarkistaa, että esimerkiksi tietokantaan vaikuttavat koodit toimivat oikein.

Tehtaan tekemiseen käytetään aiemminkin käytettyä php artisan -komentoa. Tällä kertaa komento on:

```
php artisan make:factory *TEHTAAN NIMI*
```

Käskyssä *TEHTAAN NIMI* -kohtaan kirjoitetaan luotavan koodin nimi. Nimen kannattaa yleensä kuvastaa, mihin tehdasta käytetään. Esimerkiksi käyttäjiä (*user*) luovalle tehtaalle kannattaa antaa nimeksi *UserFactory* tai vastaava nimi.

Tehtaiden kirjoittaminen on tehty Laravelissa hyvin yksinkertaiseksi, eikä valmiiksi luotuun tiedostoon tarvitse lisäillä itse paljoa. Pääasiassa lisättävää on taulukon tai mallin (*Model*) nimi, jota tehdas käyttää, sekä kentät ja tiedot, jotka tehtaan tulisi tietokantaan lisätä.

Laravelissa tehtaan tietojen luomiseen voi käyttää *Faker*-nimistä valmiiksi asennettua työkalua, joka luo tietokantaan keksittyjä tietoja käyttäjän antamien argumenttien mukaan. Esimerkiksi käyttäjiä sisältävään taulukkoon voi luoda keksityn etu- ja sukunimen sekä sähköpostin kuvan 8 mukaisella tehtaalla.

```php
<?php

use Faker\Generator as Faker;

$factory->define(App\User::class, function (Faker $faker) {
    return [
        'firstname' => $faker->firstName,
        'lastname' => $faker->lastName,
        'email' => $faker->safeEmail
    ];
});

```

Kuva 8. Käyttäjiä (user) luova tehdas

Kuvan koodiin on lisätty käsin ainoastaan rivit 7–9. Kaikki muu on luotu automaattisesti *php artisan* -komennolla.

Tehtaita voidaan käyttää testeissä luomaan niille määritettyä tietoa tietokantaan. Tehtaiden käyttäminen on tehty vielä niiden luomista yksinkertaisemmaksi. Alla olevassa käskyssä on esitetty tapa luoda kolme eri käyttäjää, joille kaikille on asetettu käsin sama etunimi. Myös muita luotavia tietoja voidaan yliajaa tai lisätä samalla tavalla, kuin oheisessa esimerkissä etunimelle on tehty.

```
$users = factory(App\User::class, 3)->create([
        'firstname' => 'Testi'
]);
```

Käskyssä kohtaan *App\User* kirjoitetaan luotavan mallin nimi, jonka jälkeen voidaan vaihtoehtoisesti erottaa pilkulla luotavien käyttäjien määrä. Lisäksi *create*-käskyn jälkeen voidaan vaihtoehtoisesti listata [] sulkujen sisään yliajettavat tai lisättävät tiedot; ylläolevassa esimerkissä kaikille kolmelle käyttäjälle yliajetaan etunimeksi 'Testi'. Yliajettavia tai lisättäviä tietoja voidaan listata pilkulla erotettuna niin monta kuin niitä halutaan olettaen, että kyseinen kenttä löytyy jo valmiiksi luotavasta mallista.

7.3.2 Jäljitelmät

Jäljitelmillä eli *mock*eilla tarkoitetaan aiemmin integraatiotestauksen yhteydessä mainittuja simuloitavia moduuleita. Laravelissa tähän voidaan käyttää niin kutsuttuja apureita (*Helpers*), jotka pohjautuvat *Mockery*-nimiseen työkaluun. Apureista on kuitenkin tehty hieman yksinkertaisempia luoda ja käyttää kuin alkuperäisen työkalun vastaavat ominaisuudet. Vaihtoehtoisesti jäljitelmien luontiin voidaan käyttää myös itse Mockery-työkalua.

Jäljitelmillä voidaan esimerkiksi estää koodin lähettämät tapahtumat (*event*), tai vaihtoehtoisesti ohjata lähetetty tapahtuma jäljiteltyyn koodiin. Tällöin moduulit saadaan erotettua toisistaan, mikä on etenkin yksikkötestauksessa tärkeää.

Jäljitelmän luominen on yksinkertaisimmillaan vain yksi lyhyt käsky, jolla kerrotaan testille, mikä ohjelman osa tai moduuli halutaan korvata jäljitelmällä. Esimerkiksi käyttäjämallin koodin voi korvata jäljitelmällä kuvassa 9 näkyvän koodin avulla. Itse jäljitelmän asennus on vain rivi 41, joten käsky on hyvin lyhyt.

```
37      public function setUp()
38      {
39          parent::setUp();
40
41          $this->user = Mock::mock(User::class);
42      }
```

Kuva 9. Jäljitelmän asetus User-luokalle

Kuvan 9 tapauksessa siis korvataan User.php-niminen malli jäljitelmällä, jolloin kaikki kyseiseen malliin testissä tehdyt kutsut tulevat jäljitelmälle. Tämän jälkeen jäljitelmälle voidaan antaa tiedoksi, mitä eri kutsuja sille odotetaan tulevan, sekä vaihtoehtoisesti, mitä jäljitelmän halutaan niihin vastaavan. Kuvassa 10 näkyy vielä, kuinka kuvassa 9 näkyvässä testissä on asetettu jäljitelmä odottamaan *getAttribute*- sekä *save*-toimintoja.

```
23          $this->user->shouldReceive('getAttribute')
24              ->andReturn(1);
25
26          $this->user->shouldReceive('save');
27
```

Kuva 10. Jäljitelmän asetetut odotukset

Jäljitelmän funktio *shouldReceive* tarkistaa, että jäljitelmä todellakin saa kyseiset kutsut, ja ilmoittaa testin lopussa, mikäli näin ei ole käynyt. Kuvassa 10 lisäksi jäljitelmää käsketään vastaamaan *getAttribute*-kutsuun 1, joka tarkoittaa myös *true* eli totta.

Jäljitelmillä voi tehdä testauksessa paljon muutakin, ja yllä oleva käyttö on vain yksi esimerkki. Mahdollisuuksia käytölle on kuitenkin niin useita käyttötarkoituksesta riippuen, ettei niiden avaaminen tässä olisi järkevää.

7.3.3 Yksikkötestaus

Yksikkötestaus PHP:ssa tapahtuu pääosin PHPUnitilla, mutta Laravelissa on joitain käteviä ominaisuuksia auttamaan testien rakentamisessa. Yksikkötestien rakentaminen käy helposti, sillä kaikki tarvittavat työkalut löytyvät Laravel / Homestead -pohjasta jo valmiiksi.

Kuten aina, testien rakentaminen aloitetaan luomalla testitiedosto. Tämän voi tehdä joko manuaalisesti tai php artisan -työkalulla. Työkalulla tiedoston luonti on kätevämpää, sillä testi menee automaattisesti testikansioon ja sisältää jo testien peruspohjan. Yksikkötestin luominen käy käskyllä:

```
php artisan make:test *TESTIN NIMI* –unit
```

Käskyssä ”TESTIN NIMI” tilalle laitetaan haluttu testitiedoston nimi ilman tähtiä. Kuvassa 11 näkyy käskyn luoman tiedoston sisältö.

```
1  <?php
2
3  namespace Tests\Unit;
4
5  use Tests\TestCase;
6  use Illuminate\Foundation\Testing\WithFaker;
7  use Illuminate\Foundation\Testing\RefreshDatabase;
8
9  class UserModelTest extends TestCase
10 {
11     /**
12      * A basic unit test example.
13      *
14      * @return void
15      */
16     public function testExample()
17     {
18         $this->assertTrue(true);
19     }
20 }
21
```

Kuva 11. UserModelTest-testitiedoston sisältö luonnin jälkeen

Kuten kuvassa 11 näkyy, php artisan luo automaattisesti pohjan testien rakentamiselle. Tiedostossa on määritelty jo valmiiksi nimiavaruus (*namespace*) Tests\Unit, joka määrittää testin kuuluvan yksikkötesteihin. Tämän jälkeen on määritelty käytettävät tiedostot sekä funktiot, jotka ovat vakiona kuvassa riveillä 5–7 näkyvät kohdat. Muut käytettävät tiedostot tai funktiot tulee lisätä itse käsin samaa *use*-käskyä käyttämällä. Viimeisenä näkyy testin luoma luokka (*class*), joka pohjautuu *TestCase*-tiedostoon. Kaikki testit tulee kirjoittaa kyseisen luokan sisälle, ja luokka sisältää valmiiksi muokattavan esimerkkitestin.

Koska yksikkötestit keskittyvät testaamaan yksittäisiä funktiota, ovat itse testit yleensä melko lyhyitä. Testejä tulee kuitenkin määrällisesti paljon, koska jokainen funktio tulisi testata mahdollisimman suuren kattavuuden nimissä. Koska osa funktioista luottaa muihin funktioihin tai moduuleihin, tulee myös aiemmin mainittuja jäljitelmiä suuri määrä. Kuvassa 12 näkyy esimerkkiä varten rakennettu testi, jossa testataan käyttäjämallista yksittäistä funktiota *profileFieldsValid()*. Funktiota halutaan testata sekä tilanteessa, jossa sen pitäisi mennä läpi, että tilanteissa, joissa sen odotetaan antavan virhe.

Testissä on muutettu automaattisesti mukaan tuleva *RefreshDatabase*-moduulin tilalle *DatabaseTransactions*-niminen moduuli, joka peruu automaattisesti testien aikana tehdyt muutokset. *DatabaseTransactions* ei kuitenkaan poista tietokannassa mahdollisesti valmiina olleita tietoja, mikä olikin suurin syy sen valintaan.

Itse testissä käytetään User-mallista löytyvää *profileFieldsValid()*-funktiota, jota käytetään tarkistamaan, että käyttäjältä löytyy varmasti kaikki pakolliset tiedot. Mallille pakolliseksi merkityt tiedot ovat tässä tapauksessa etunimi (*firstname*), sukunimi (*lastname*), sekä syntymävuosi (*birth_year*).

Koska funktio halutaan testata mahdollisimman tarkasti, rakentuu yksittäisen funktion testaaminen neljästä eri testistä:

1. Käyttäjältä löytyy kaikki tarvittavat tiedot.
2. Käyttäjän tiedoista puuttuu etunimi.
3. Käyttäjän tiedoista puuttuu sukunimi.
4. Käyttäjän tiedoista puuttuu syntymävuosi.

Näin funktion toimivuus saadaan testattua jokaiselle kentälle erikseen ja varmistutaan siitä, että funktio toimii jokaisessa oheisista tilanteista. Koska User-mallissa on monia muitakin funktioita ja kaikki halutaan testata mahdollisimman laajasti, tulee lopullisesta testitiedostosta huomattavan paljon suurempi.

```php
<?php

namespace Tests\Unit;

use Tests\TestCase;
use Illuminate\Foundation\Testing\WithFaker;
use Illuminate\Foundation\Testing\DatabaseTransactions;

use App\User;

class UserTest extends TestCase
{
    use DatabaseTransactions;

    public function testAllValid()
    {
        $this->user = factory(User::class)->create();
        $this->assertTrue($this->user->profileFieldsValid());
    }
    public function testFirstNameMissing()
    {
      $this->user = factory(User::class)->create([
        'firstname' => '',
      ]);
      $this->assertNotTrue($this->user->profileFieldsValid());
    }
    public function testLastNameMissing()
    {
      $this->user = factory(User::class)->create([
        'lastname' => '',
      ]);
      $this->assertNotTrue($this->user->profileFieldsValid());
    }
    public function testBirthYearMissing()
    {
      $this->user = factory(User::class)->create([
        'birth_year' => '',
      ]);
      $this->assertNotTrue($this->user->profileFieldsValid());
    }
    protected function setUp()
    {
        parent::setUp();
    }
    protected function tearDown()
    {
        parent::tearDown();
    }
}

```

Kuva 12. profileFieldsValid()-funktion testit

Kun kaikki testit on kirjoitettu, käy testien ajaminen yksinkertaisesti oheisella käskyllä:

```
phpunit *POLKU TESTEIHIN*/Unit/
```

Käskyssä *POLKU TESTEIHIN* tulee korvata testit sisältävän kansion polulla, jälleen ilman tähtiä. Yleensä polku on *tests/Unit/*, sillä php artisan -käsky luo testit automaattisesti oheiseen kansioon. Mikäli polkua testeihin ei lisätä, ajaa phpUnit kaikki vakiopolustaan löytämänsä testit, mukaan lukien kaikki integraatiotestit.

Mikäli testeistä haluaa lisätietoa, voi käskyn perään lisätä vielä vaihtoehtoisesti *--verbose*. Aina lisätietoa ei kuitenkaan ole, ja tämän takia testien nimet sekä mahdolliset tulostukset kannattaa tehdä mahdollisimman selviksi, mikäli jokin testi epäonnistuu.

7.3.4 Integraatiotestaus

Integraatiotestaus pohjautuu phpUnitissa samalle pohjalle kuin yksikkötestaus. Ainoat erot testeillä ovat erilliset juurikansiot sekä pieni ero testitiedoston luontiin käytettävässä käskyssä. Integraatiotestin pohjan saa luotua käskyllä:

```
php artisan make:test *TESTIN NIMI*
```

Kuten yksikkötesteissäkin, tulee *TESTIN NIMI* -kohta korvata luotavan testitiedoston nimellä. Käskyllä luotu testitiedosto on identtinen yksikkötestien luontiin, mutta Unit-kansion sijasta testitiedosto luodaan sen sijaan Feature-nimiseen kansioon. Kuvassa 13 näkyy käskyllä luotu testitiedosto *ControllerTest.php*.

```php
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Foundation\Testing\WithFaker;
use Illuminate\Foundation\Testing\RefreshDatabase;

class ControllerTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testExample()
    {
        $this->assertTrue(true);
    }
}
```

Kuva 13. ControllerTest-tiedoston pohja

Myös testeissä käytettävät komennot ovat pääasiassa samoja, mutta tällä kertaa jäljitelmien sijasta käytetään oikeita tiedostoja, jolloin varmistutaan tiedon kulkevan koodien välillä oikein. Tällöin saadaan selville, mikäli joidenkin koodien tai moduulien välisessä toiminnassa on virheitä.

Integraatiotestien kohteena on monesti erilaiset kontrollerit tai koodit, joilla hallitaan esimerkiksi käyttäjämalleja. Kuvassa 14 on esimerkki koodista, jonka avulla käyttäjän salasana voidaan vaihtaa konsolin kautta. Kyseinen koodi käyttää hallintaan User-mallia, ja käyttää esimerkiksi mallin *save()*-komentoa. Koska käytössä on oikea malli eikä jäljitelmä, on kyseessä integraatiotesti.

```php
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Foundation\Testing\WithFaker;
use Illuminate\Foundation\Testing\DatabaseTransactions;

use App\User;
use App\Console\Commands\ChangeUserPassword;

class ChangeUserPasswordTest extends TestCase
{
    use DatabaseTransactions;
    // Test with all correct info
    public function testSuccesfull()
    {
        $this->artisan('auth:change-password')
            ->expectsQuestion('Email', $this->user->email)
            ->expectsOutput('User found: id='.$this->user->id.', email="' .
            $this->user->email.'", firstname="'.$this->user->firstname . '"')
            ->expectsQuestion('Password', 'newPassword')
            ->expectsQuestion('Password again, must match with password', 'newPassword')
            ->expectsOutput('Saved.');
    }

    // Test with non-existent email address
    public function testNoUser()
    {
        $this->artisan('auth:change-password')
            ->expectsQuestion('Email', 'NoSuchEmail')
            ->expectsOutput('User not found');
    }

    // Test with wrong password, should keep asking for password again
    public function testPasswordsDontMatch()
    {
        $this->artisan('auth:change-password')
            ->expectsQuestion('Email', $this->user->email)
            ->expectsOutput('User found: id='.$this->user->id.', email="' .
            $this->user->email.'", firstname="'.$this->user->firstname . '"')
            ->expectsQuestion('Password', 'newPassword')
            ->expectsQuestion('Password again, must match with password', 'wrongPassword')
            ->expectsQuestion('Password again, must match with password', 'wrongPasswordAgain')
            ->expectsQuestion('Password again, must match with password', 'newPassword')
            ->expectsOutput('Saved.');
    }

    protected function setUp()
    {
        parent::setUp();

        $this->user = factory(User::class)->create();
        $this->controller = new ChangeUserPassword;
    }
}

```

Kuva 14. ChangeUserPassword-kontrollerin testi

Kuvan 14 tapauksessa on käytetty myös Laravelista löytyviä konsoliapplikaatioiden testaamiseen tarkoitettuja *artisan()*-käskyn funktioita, kuten *expectsQuestion()* sekä *expectsOutput()*. Näiden käskyjen avulla voidaan vastata koodin lähettämiin kyselyihin sekä tarkistaa, että koodi tulostaa halutun tuloksen.

7.3.5 Järjestelmätestaus

Järjestelmätestit koostuvat toimeksiantajan tapauksessa pääasiassa selaintesteistä, joita ajetaan Laravel Duskin avulla. Selaintesti tarkoittaa sitä, että itse testaustyökalu käyttää palvelimen sivustoja virtuaaliympäristössä ja ajaa siellä testeissä annettuja käskyjä. Näin saadaan testattua esimerkiksi kaikkien painikkeiden sekä lomakkeiden toiminta.

Koska testit luottavat Laravel Dusk -työkaluun, pitää se asentaa ensin. Työkalun asentaminen käy vaivattomasti, sillä kyseessä on Laravelin oma työkalu:

- Lisää Laravel Dusk Composer -työkalun vaatimuksiin.
    - composer require --dev laravel/dusk
- Aja Laravel Duskin asennustoiminto.
    - php artisan dusk:install

Ja näin työkalu on asennettu ja valmis ajettavaksi. Ensin kuitenkin täytyy rakentaa itse testit. Pohjan luonti käy jälleen kerran kätevästi php artisanin avulla:

```
php artisan dusk:make *TESTIN NIMI*
```

Kuten aiemminkin, pitää *TESTIN NIMI* kohtaan kirjoittaa halutun testin nimi, esimerkiksi LoginTest, mikäli testataan sivustolle kirjautumista. Kuvassa 15 näkyy oheisella käskyllä luotu testitiedosto.

```php
<?php

namespace Tests\Browser;

use Tests\DuskTestCase;
use Laravel\Dusk\Browser;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class LoginTest extends DuskTestCase
{
    /**
     * A Dusk test example.
     *
     * @return void
     */
    public function testExample()
    {
        $this->browse(function (Browser $browser) {
            $browser->visit('/')
                    ->assertSee('Laravel');
        });
    }
}

```

Kuva 15. php artisan -käskyllä luotu LoginTest

Testit koostuvat pääosin rivillä 20 olevan *assertSee()*-käskyn tyyppisistä käskyistä, joissa Laravel Dusk -työkalulle kerrotaan, mitä sen halutaan tekevän, tai mitä sen pitäisi milläkin hetkellä nähdä. Käskyjä on käytettävissä todella suuri määrä, ja oikeat käskyt kannattaa tarkistaa aina Laravel Duskin dokumentaatiosta.

Monessa tapauksessa yleisimpiä käskyjä ovat esimerkiksi *press()* sekä *type()*. Käskyistä ensimmäisellä käsketään työkalua painamaan sivulla näkyvää painiketta. Esimerkiksi *press('Rekisteröidy')* painaa nappia, jonka nimeksi on asetettu 'Rekisteröidy'. Jälkimmäinen käskyistä taas käskee työkalua kirjoittamaan tieto johonkin kenttään. Esimerkiksi *type('email', 'oma@sähköposti.fi')* kirjoittaa sivustolle 'email'-nimen omaavaan kenttään osoitteen 'oma@sähköposti.fi'. Näitä käskyjä antamalla pyritään kokeilemaan, että sivusto toimii oikein.

Kuvassa 16 näkyy yksinkertainen versio valmiista LoginTest-testistä. Testissä työkalu ohjataan aluksi etusivulle käskyllä *visit('/')*, ja sen jälkeen sitä käsketään painamaan painiketta, jonka nimenä on *a.btn.btn-primary.btn-lg.digits-login-btn*. Painikkeen nimi on tässä tapauksessa kirjoitettu mahdollisimman tarkasti, jotta työkalu painaa varmasti juuri oikeaa painiketta.

Painiketta painettuaan työkalu kirjoittaa sille annetut sähköpostiosoitteen sekä salasanan, ja painaa sen jälkeen *button.btn.btn-primary.pull-right*-nimen omaavaa painiketta, jolla sivustolle kirjaudutaan. Tämän jälkeen työkalu tarkistaa näkevänsä tervehdyksen, jonka asiakas näkisi kirjautuessaan. Mikäli tervehdystä ei näy, testi ilmoittaa epäonnistuneensa sekä ottaa kuvakaappauksen omasta näkymästään. Kuvakaappauksesta testaaja voi alkaa päätellä, missä vaiheessa asiat ovat menneet pieleen.

```php
<?php

namespace Tests\Browser;

use Tests\DuskTestCase;
use Laravel\Dusk\Browser;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class LoginTest extends DuskTestCase
{
    /**
     * A Dusk test example.
     *
     * @return void
     */
    public function testExample()
    {
        $this->browse(function (Browser $browser) {
            $browser->visit('/')
                    ->press('a.btn.btn-primary.btn-lg.digits-login-btn')
                    ->type('email', 'testing@example.test')
                    ->type('password', 'justTesting042019')
                    ->press('button.btn.btn-primary.pull-right')
                    ->assertSee('Hei!');
        });
    }
}
```

Kuva 16. Yksinkertainen LoginTest -testitiedosto

Kuvan 16 esimerkissä testaustietokantaan on luotu ennen itse testiä sen käyttämät tunnukset valmiiksi, jotta työkalu pystyy kirjautumaan sisään. Käyttäjätunnuksen voisi kuitenkin luoda samaisen User-tehtaan avulla kuin aiemmissakin testeissä, jolloin tiedot myös poistuvat testin jälkeen automaattisesti.

Itse testien ajaminen käy yhtä kätevästi kuin yksikkö- ja integraatiotestitkin, vaikka käsky onkin hieman eri:

```
php artisan dusk
```

Testien tulokset esitetään suurelta osin samalla lailla kuin phpUnitin omissa testeissä, sillä myös Laravel Dusk käyttää phpUnitia. Dusk kuitenkin ottaa testien epäonnistuessa kuvan sekä tallentaa sen selaintestien juureen. Kuvakaappauksesta näkee verkkosivun sellaisena, kuin se käyttäjälle näkyisi samassa tilanteessa. Näin virheen voi testata itse käsin tai päätellä kuvasta, minkä takia testi on epäonnistunut.

Erilaisia käskyjä sekä onnistumisen varmistuksia löytyy työkalusta todella suuri määrä, joten käyttötarkoitukseen sopivat vaihtoehdot kannattaa etsiä Laravel Duskin dokumentaatiosta.

8 Yhteenveto

Tavoitteena työllä oli tutkia toimeksiantajalle sopivimmat testauksen työkalut sekä luoda testausta varten pohja ja ohjeistus. Standardisoitujen työkalujen avulla testaamisesta saadaan varmempaa, eivätkä testien ajot tai tulokset heittele käyttäjien kesken. Lisäksi testauspohjalla sekä -ohjeistuksella saadaan toimeksiantajalle toimintavarmuutta sekä valmius testauksen jatkokehitykseen.

Tuloksena työstä toimeksiantajalle saatiin rakennettua testejä niin yksikkö-, integraatio- kuin järjestelmätestaukseen. Testejä on kuitenkin rakennettavana vielä suuri määrä ennen kunnollista kattavuutta, ja testejä tullaan rakentamaan lisää aina ympäristön kehittyessä ja muuttuessa. Täydellistä kattavuutta on lähes mahdoton saada, sillä testien tarve lisääntyy ajan myötä.

Itse testausohjeistus pohjautuu suurelta osin samoihin tietoihin kuin tämän työn 7. osio. Ohjeistus kuitenkin käy testaamiseen käytettävien työkalujen ominaisuuksia syvemmin, ja ohjeistus on tarkempaa sekä toimeksiantajalle suunnatumpaa. Ohjeistus saatiin työssä hyvään vaiheeseen, mutta vaatii vielä tarkempaa testauksen läpikäyntiä. Lisäksi myös ohjeistusta tullaan kehittämään käyttöympäristön kehittyessä.

Jälkikäteen projektin etenemistä katsoessa olisi ollut toivottavaa, että testausta olisi päästy rakentamaan myös toimeksiantajan tapauksessa aiemmin, kun käyttöympäristö oli vasta tekeillä. Tällöin testien rakentaminen olisi ollut huomattavan paljon helpompaa, ja todennäköisesti myös käyttöympäristöstä olisi tullut selkeämpi opetella.

Itse työhön olen tyytyväinen ja tunnen oppineeni työn ansiosta paljon uutta tietoa sekä taitoja, joista tulee varmasti olemaan hyötyä myös jatkossa. Työn alussa toivoisin opetelleeni toimeksiantajan palvelinympäristön tarkemmin, sillä sen opettelulle jäänyt aika pieneni lopussa huomattavasti, ja samalla testien rakentamisesta tuli työläämpää. Toivon myös, että testejä olisi saatu rakennettua enemmän kattavamman pohjan saamiseksi. Testien kehitystä kuitenkin jatketaan myös työn jälkeen, joten kattavuus tulee parantumaan tulevaisuudessa.

Lähteet

1 International Software Test Institute. Software Testing Revealed. Wollerau, Sveitsi; International Software Test Institute; 2019.

2 Tutorialspoint. What is Static Testing? Haettu 08.02.2019 osoitteesta https://www.tutorialspoint.com/software_testing_dictionary/static_testing.htm.

3 Tutorialspoint. What is Functional Testing? Haettu 08.02.2019 osoitteesta https://www.tutorialspoint.com/software_testing_dictionary/functional_testing.htm.

4 Software Testing Fundamentals. Black Box Testing. Haettu 08.02.2019 osoitteesta http://softwaretestingfundamentals.com/black-box-testing/.

5 Software Testing Fundamentals. White Box Testing. Haettu 08.02.2019 osoitteesta http://softwaretestingfundamentals.com/white-box-testing/.

6 Software Testing Fundamentals. Gray Box Testing. Haettu 08.02.2019 osoitteesta http://softwaretestingfundamentals.com/gray-box-testing/.

7 Ulf Eriksson. Differences between the different levels of tests. Haettu 05.02.2019 osoitteesta https://reqtest.com/testing-blog/differences-between-the-different-levels-of-tests/.

8 Software Testing Fundamentals. Unit Testing. Haettu 05.02.2019 osoitteesta http://softwaretestingfundamentals.com/unit-testing/.

9 Software Testing Fundamentals. Integration Testing. Haettu 05.02.2019 osoitteesta http://softwaretestingfundamentals.com/integration-testing/.

10 Guru99. Integration Testing. Haettu 05.02.2019 osoitteesta https://www.guru99.com/integration-testing.html#2.

11 STC Admin (2012, 01. lokakuuta). System Testing: What? Why? & How? Haettu 05.02.2019 sivustolta Software Testing Class, osoitteesta https://www.softwaretestingclass.com/system-testing-what-why-how/.

12 Software Testing Fundamentals. Acceptance Testing. Haettu 07.02.2019 osoitteesta http://softwaretestingfundamentals.com/acceptance-testing/.

13 HashiCorp Inc. Introduction to Vagrant. Haettu 08.02.2019 osoitteesta https://www.vagrantup.com/intro/index.html.

14 Taylor Otwell, Laravel.com. Laravel Homestead introduction. Haettu 08.02.2019 osoitteesta https://laravel.com/docs/5.7/homestead#introduction.

15 Linux.fi -wiki. Muokattu viimeksi 10. joulukuuta 2018. Haettu 01.02.2019 osoitteesta https://www.linux.fi/wiki/Ubuntu.

16 W3Techs. Web technologies of the year 2018. Haettu 01.02.2019 osoitteesta https://w3techs.com/blog/entry/web_technologies_of_the_year_2018.

17 NGINX Inc. What is NGINX? How different is it from Apache (for example)?. Haettu 01.02.2019 osoitteesta https://www.nginx.com/faq/what-is-nginx-how-different-is-it-from-e-g-apache/.

18 Wikipedia, Wikimedia Foundation. Nginx. Haettu 01.02.2019 osoitteesta https://fi.wikipedia.org/wiki/Nginx.

19 The PHP Group. PHP manual. Haettu 01.02.2019 osoitteesta http://fi2.php.net/manual/en/intro-whatis.php.

20 The PostgreSQL Global Development Group. Haettu 01.02.2019 osoitteesta https://www.postgresql.org/about/.

21 redis.io. Haettu 01.02.2019 osoitteesta https://redis.io/.

22 Software Testing Help, muokattu viimeksi 31.08.2018. Haettu 04.03.2019 osoitteesta https://www.softwaretestinghelp.com/php-testing-framework-tools/.

23 Hongkiat (HKDC), muokattu viimeksi 27.10.2018. Haettu 04.03.2019 osoitteesta https://www.hongkiat.com/blog/automated-php-test/.

24 SeleniumHQ. Haettu 04.03.2019 osoitteesta https://www.seleniumhq.org.

25 Codeception. Introduction. Haettu 05.03.2019 osoitteesta https://codeception.com/docs/01-Introduction.

26 DataSift. Understanding Stories. Haettu 05.03.2019 osoitteesta https://datasift.github.io/storyplayer/v2/learn/fundamentals/understanding-stories.html.

27 Taylor Otwell. Laravel Dusk. Haettu 07.03.2019 osoitteesta https://laravel.com/docs/5.8/dusk.

28 pgTAP. Haettu 10.03.2019 osoitteesta https://pgtap.org.

Kuvat:

Kansikuva - testbytes, haettu osoitteesta https://pixabay.com/illustrations/software-testing-service-762486/