SAVONIA

OPINNÄYTETYÖ - **AMMATTIKORKEAKOULUTUTKINTO**
TEKNIIKAN JA LIIKENTEEN ALA

PHP-OHJELMISTON KÄYTTÖLIITTYMÄ-, KUORMITUS- JA YKSIKKÖTESTAUS

TEKIJÄ/T: Miika Niemi

SAVONIA-AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Tiivistelmä

Koulutusala Tekniikan ja liikenteen ala	
Koulutusohjelma Tietotekniikan koulutusohjelma	
Työn tekijä(t) Miika Niemi	
Työn nimi PHP-ohjelmiston käyttöliittymä-, kuormitus- ja yksikkötestaus	
Päiväys 20.4.2017	Sivumäärä/Liitteet 25/0
Ohjaaja(t) lehtori Jussi Koistinen, lehtori Sami Lahti	
Toimeksiantaja/Yhteistyökumppani(t) Finnish Net Solutions Oy	

Tiivistelmä

Työn aiheena oli Finnish Net Solutions Oy:n ylläpitämän eläinlääkäriohjelmiston yksikkö-, kuormitus- ja käyttöliittymätestien automatisointi. Työn tarkoitus oli toteuttaa ohjelmalliset testit niiltä osin, kuin oli aikataulun myötä mahdollista, ja tutkia mahdollisuuksia niiden automaattiseen suorittamiseen. Automaattisten testien tarkoituksena oli vähentää ohjelmiston testaukseen kuluvaa aikaa ja pitää yllä hyvä virheiden havaitsemistarkkuus.

Käyttöliittymä ja yksikkötestit toteutettiin Codeception-testauskehyksellä. Codeception valittiin, koska sama testauskehys soveltui käyttöliittymä- ja yksikkötesteihin. Myös hyvät dokumentaatiot ja asennuksen helppous puolsivat päätöstä. Kuormitustestauksen toteutukseen valittiin Gatling-työkalu, koska se oli helppo asentaa. Myös testien alkuun saamisen helppous johtivat tämän valintaan.

Työn prioriteetti oli käyttöliittymätestit. Käyttöliittymän toimintojen testauksesta haluttiin toteuttaa ohjelmallisesti mahdollisimman suuri osa, koska toimintojen manuaalinen testaus vie paljon aikaa. Yksikkötestauksen tavoite muuttui työn aikana, sillä niiden toteutukselle ei nähty tarvetta. Kuormitustestaus haluttiin toteuttaa ohjelmiston uudelle etusivulle, joka oltiin julkaisemassa pian työn aloittamisen jälkeen. Tarkoituksena oli testata, kuinka uusi etusivu tulisi suoriutumaan ohjelmistoon kohdistuvista käyttäjämääristä.

Työn lopputuloksena saatiin ohjelmalliset testit noin viidellekymmenelle prosentille käyttöliittymän toiminnoista. Käyttöliittymän toimintojen testaukseen kuluva aika väheni useita tunteja. Yksikkötestejä toteutettiin muutaman esimerkin verran niiden mahdollista myöhempää jatkamista varten. Kuormitustesti saatiin toteutettua vanhalle etusivulle. Uusi etusivu julkaistiin, ennen kuin sen kuormitustesti ehdittiin toteuttaa.

Avainsanat
Testaus, Codeception, Selenium

SAVONIA UNIVERSITY OF APPLIED SCIENCES

THESIS
Abstract

Field of Study Technology, Communication and Transport			
Degree Programme Degree Programme in Information Technology			
Author(s) Miika Niemi			
Title of Thesis GUI, Unit and Stress Testing of a PHP-software			
Date	20.4.2017	Pages/Appendices	26/0
Supervisor(s) Mr. Jussi Koistinen, Lecturer; Mr. Sami Lahti, Lecturer			
Client Organisation /Partners Finnish Net Solutions Oy			

Abstract

The purpose of this thesis was to automate the GUI, unit and stress testing of a veterinarian web application provided by Finnish Net Solutions Ltd. The aim was to implement automated tests as broadly as possible within a reasonable timeframe. The automated tests were supposed to reduce the time spent in testing the application while still finding possible defects atleast as well as manual testing.

UI and unit tests were implemented with the Codeception testing framework. Codeception was chosen because the same tool could be used for both test types. Also good documentation and ease of installation made the choice more obvious. Stress testing was carried out with Gatling, which is a stress testing tool for web applications. Gatling was an easy choice, because of the simple installation process. It was also very easy and fast to produce a skeleton for a stress test.

The greatest priority of this thesis was UI testing, because it is the most time consuming part of testing this particular application. This was why the goal was to automate as much of the UI tests as possible. The priority for unit testing changed in the process, because the company did not see much need for them. Stress testing was meant to be implemented on the new front page of the application, which was to be published soon after starting this thesis. The goal was to test how the new front page would work under a normal user load.

As the result for this thesis, about fifty percent of the tests for the UI's functions were automated. The time needed to test the UI dropped by several hours. A few unit tests were implemented to serve as an example, if unit testing was found to be useful in the future. A stress test was implemented for the old front page of the application. The new front page was published before a stress test could have been implemented for it.

Keywords
Testing, Codeception, Selenium

ESIPUHE

Kiitos Finnish Net Solutions Oy:lle opinnäytetyön aiheesta ja opastuksesta läpi projektin. Kiitos lehtori Jussi Koistiselle ohjauksesta ja insinööri Tanja Tikkaselle tuesta ja kärsivällisyydestä.

Kuopiossa 8.5.2017

Miika Niemi

SISÄLTÖ

1 JOHDANTO 7
2 OHJELMISTOTESTAUS 8
2.1 Käyttöliittymätestaus 8
2.2 Kuormitustestaus 8
2.3 Yksikkötestaus 8
2.4 Automatisointi 8
3 KÄYTETYT TYÖKALUT JA TEKNIIKAT 9
3.1 Codeception 9
3.2 Gatling 9
4 TOTEUTUS 10
4.1 Käyttöliittymätestit 10
4.1.1 Testien toteutus 11
4.1.2 Testien suorittaminen 15
4.2 Yksikkötestit 16
4.2.1 Testien pohjustus 16
4.2.2 Testien toteutus 17
4.2.3 Testien suorittaminen 17
4.3 Kuormitustestit 18
4.3.1 Testien pohjustus 18
4.3.2 Testien toteutus 19
4.3.3 Testien suorittaminen 22
5 YHTEENVETO 24
LÄHTEET 25

TERMIT JA LYHENTEET

Selenium Webdriver = Käyttöliittymätestaukseen tarkoitettu testaustyökalu, joka tekee suoria pyyntöjä selaimeen käyttäen selaimen natiivia automaatiotukea (Selenium Webdriver, 2016-09-04).

Scala (Scalable Language) = Yleiskäyttöinen, vuonna 2004 julkaistu ohjelmointikieli (Odersky, 2017).

Composer = Työkalu PHP-projektien riippuvaisuuksien hallintaan (Composer Documentation, 2017).

Jar (Java Archive) = Tiedosto, joka sisältää kokonaisen Java-applikaation luokka-, kuva- ja äänitiedostot koottuna yhteen, mahdollisesti pakattuun, tiedostoon (Rouse a, 2017).

YAML (YAML Aint Markup Language) = Merkintäkieli (markup language), joka on kevyempi vaihtoehto XML:lle tai JSON:lle (Vepsäläinen, 2015-01-11).

1 JOHDANTO

Tässä työssä tarkastellaan Finnish Net Solutions Oy:n (myöhemmin FNS) eläinlääkäriohjelmiston automaattista testaamista. Tuote on toiminnoiltaan laaja ja toimintojen manuaaliseen testaamiseen kuluu paljon aikaa. Ohjelmiston testaamisen avuksi FNS on tehnyt testausraportin, joka pitää sisällään kaikki käyttöliittymän testitapaukset. Raportin määrittämien testitapausten tarkistukseen ennen tämän työn aloittamista kului kahdelta työntekijältä yhteensä kahdesta kolmeen työpäivää. Lisäksi työssä tutkitaan myös eläinlääkäriohjelmiston automatisoitua yksikkö- ja kuormitustestausmahdollisuutta. Ohjelmistoon ei ole aiemmin toteutettu näistä kumpaakaan.

Provet-eläinlääkäriohjelmistot ovat eläinlääkäriohjelmistojen markkinajohtajia Suomessa (Provet, 2017). Provetista on olemassa eri versioita, mutta tässä työssä keskitytään yksityiseläinlääkäreille kohdistettuun Provet Net -ohjelmistoon.

Provet Net -järjestelmää ylläpitää Finnish Net Solutions Oy:n Kuopion toimiston kehitysosasto. Ohjelmistoon pyritään julkaisemaan uusi päivitys n. kuukauden välein. Päivityksissä tarjotaan korjauksia olemassa oleviin ominaisuuksiin sekä usein myös kokonaan uusia ominaisuuksia. Ohjelmiston käyttöliittymä ja sen toiminnot pyritään testaamaan läpikotaisesti ennen päivitysten julkaisua.

Työn tavoite on tutkia Provet Net -ohjelmiston yksikkö-, käyttöliittymä- ja kuormitustestauksen automatisoinnin mahdollisuutta ja toteuttaa se ainakin osittain. Jo alussa oli selvää, että vielä tämän työn lopussa automaattisten testien ajaminen tulisi olemaan ihmisen vastuulla. Testien automaattiseen ajamiseen vaadittavia järjestelmiä ei ollut vielä käytössä. Työn lopussa kuitenkin tarkastellaan mahdollisuuksia yksikkö- ja käyttöliittymätestien ajamisen automatisointiin esimerkiksi aina silloin, kun koodiin tehdään muutoksia.

Työn tavoitteena on myös merkittävästi helpottaa Provet Netin testausprosessia ja vähentää siihen kuluvaa aikaa. Tavoitteena on myös rakentaa testauspohja, jonka päälle on helppo rakentaa uusia testejä sen mukaan, kun ohjelmisto kehittyy.

2 OHJELMISTOTESTAUS

2.1 Käyttöliittymätestaus

Käyttöliittymätestaus on prosessi, jossa testataan, että tuotteen käyttöliittymä vastaa sen määrityksiä. Tämä tapahtuu normaalisti useiden eri testitapausten käytön myötä. (Rouse b, 2014)

Automatisoitu käyttöliittymätestaus on manuaalisten testitapausten automatisointia. Automaattiset testit ovat paljon tarkempia, nopeampia ja luotettavampia kuin manuaalinen testaus. Vaikka testauksen automatiointi voi viedä aikaa, on se ajan myötä kustannustehokkaampaa kuin manuaalinen testaus. (Urbonas, 2013-11-04.)

2.2 Kuormitustestaus

Kuormitustestauksessa keskitytään sovelluksen kestävyyteen, saatavuuteen ja luotettavuuteen äärimmäisissä olosuhteissa. Kuormitustestauksen tarkoitus on löytää sovelluksen ongelmia, jotka tulevat ilmi esimerkiksi vain suurissa käyttäjäkuormissa. Kuormitustesteissä yleisesti simuloidaan yhtä tai useampaa sovelluksen käyttötapausta erilaisissa kuormittavissa olosuhteissa. (Meier, Farre, Prashant, Barber, Rea. 2007.)

2.3 Yksikkötestaus

Yksikkötestaus tarkoittaa tiettyjen ohjelmakoodin funktioiden ja alueiden – yksiköiden – testaamista. Näin pystytään varmistamaan, että funktiot toimivat niin kuin odotetaan. Esimerkiksi jos funktiolle annetaan arvoja, voidaan olla varmoja, että funktio palauttaa niitä arvoja, joita pitäisikin ja käsittelee virheet niin kuin pitäisikin sellaisten sattuessa. (McFarlin. 2011.)

2.4 Automatisointi

Ohjelmistotestauksen automatisoinnissa käytetään hyväksi erikoistyökaluja, joilla kontrolloidaan testien suorittamista ja vertaillaan saatuja tuloksia odotettuihin tuloksiin (Tutorialspoint).

3 KÄYTETYT TYÖKALUT JA TEKNIIKAT

3.1 Codeception

Codeception on PHP-ohjelmistojen testaukseen tarkoitettu testauskehys. Sillä pystyy toteuttamaan sekä yksikkö-, käyttöliittymä- ja funktionaalisia testejä. Codeception pohjautuu PHPUnit-testauskehykseen. (Tutsplus)

Testit kirjoitetaan Codeceptionilla käyttäen Actor-luokkia. Jokaisella testityypillä on oma Actor-luokkansa, joka pitää sisällään sen testityypin testeissä käytössä olevat funktiot. Esimerkiksi käyttöliittymätesteissä, Actor-luokka pitää sisällään funktioita käyttöliittymässä olevien elementtien tarkistamiseen ja vuorovaikutukseen, kuten nappien klikkaamiseen tai lomakkeen kenttien täyttämiseen.

Codeceptionin Actor-luokkien toiminnallisuutta voidaan lisätä ottamalla käyttöön Codeceptionin mukana tulevia moduuleja. Tässä työssä käyttöön otettiin Assert ja Db -moduulit. Assert-moduuli lisäsi testeihin käytettäväksi ns. väittämäfunktioita, joilla voidaan tarkistaa jonkin true-false-väittämän paikkansapitävyys. Db-moduuli lisäsi testeihin funktiot tietokannan kanssa keskusteluun. Db-moduulilla oli mm. mahdollista lisätä testin ajaksi haluttua dataa tietokantaan tai tarkistaa, löytyykö jokin tietty data tietokannasta.

3.2 Gatling

Gatling on avoimen lähdekoodin kuormitustestauskehys, joka pohjautuu Scala-ohjelmointikieleen. Se on suunniteltu käytettäväksi erilaisten palveluiden suorituskyvyn mittaukseen ja analysointiin, mutta se soveltuu parhaiten web-ohjelmiin. (Wikipedia)

Kuormitustestin toteuttaminen Gatlingilla aloitetaan nauhoittamalla kuormitustestissä käytettävä testiskenaario Gatlingin Recorder-työkalulla. Testiskenaario pitää sisällään toimet, jotka halutaan testattavalla sivustolla tehdä. Recorder nauhoittaa kaikki testiskenaariossa lähetetyt http-pyynnöt ja luo niistä nauhoituksen jälkeen automaattisesti kuormitustestin Scala-ohjelmointikielellä. Luotuun testiin voidaan tämän jälkeen määrittää haluttu määrä virtuaalisia käyttäjiä, jotka toistavat testiskenaarion toimet testattavalla sivustolla.

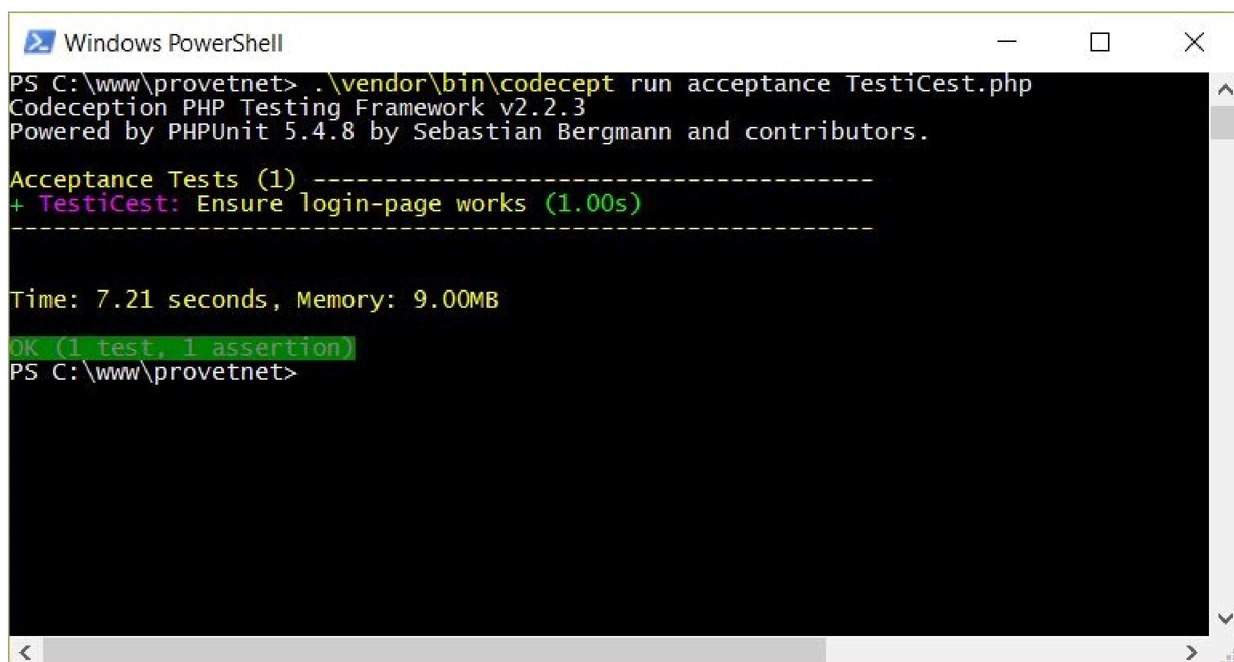
4 TOTEUTUS

4.1 Käyttöliittymätestit

Codeception asennettiin käyttämällä Composeria. Composer asensi myös kaikki tarvittavat paketit, joista Codeception on riippuvainen. Kun Codeception riippuvuuksineen oli asennettu, suoritettiin Codeceptionin bootstrap-toiminto, joka luo testien tiedostorakenteen. Aluksi Codeceptionin konfigurointitiedostosta laitettiin tietokanta-asetukset kohdilleen ja testattiin kuvan 1 mukaisella yksinkertaisella testillä, että ympäristö toimii. Perustesti tuotti kuvassa 2 nähtävän, helposti luettavan tuloksen komentoriville. Testissä siirryttiin ohjelmiston index.php-sivulle ja katsottiin, että sivu sisältää lomakkeen, jonka nimi on loginform.

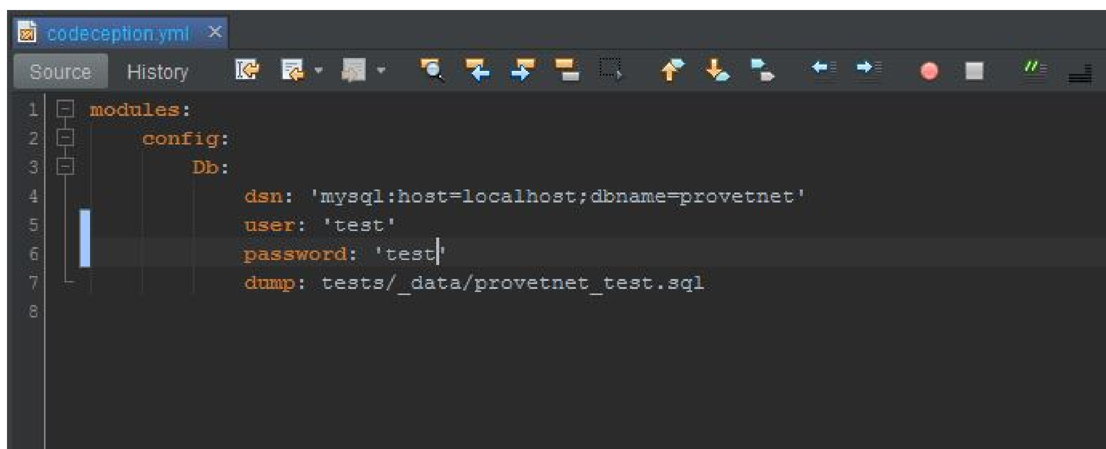
```
public function loginPageTest(AcceptanceTester $I) {
    $I->wantTo('Ensure login-page works');

    $I->amOnPage('index.php');
    $I->seeElement('form', ['name' => 'loginform']);
}
```

KUVA 1. Perustestin koodi (Niemi 2016-11-04.)



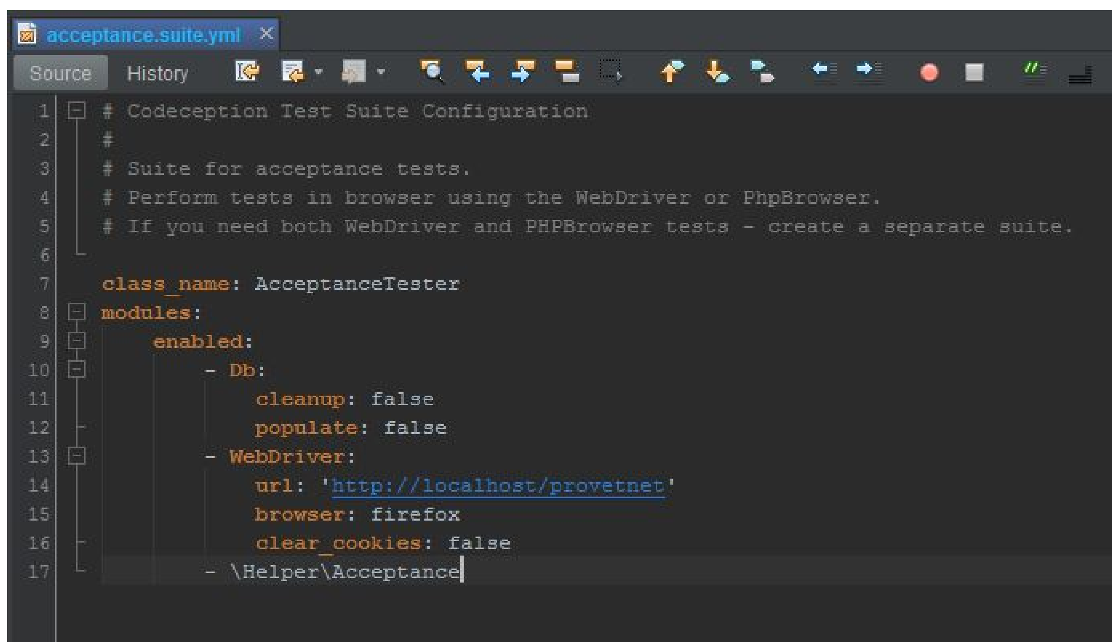
KUVA 2. Perustestin suoritus (Niemi 2016-11-04.)

Koska ohjelmistolla on monta kehittäjää, on myös mahdollisia testausasetuksia monia. Tiedostojuureen tehtiin yksi kaikille kehittäjille jaettava codeception.dist.yml-tiedosto, johon määritettiin yleispätevät kaikille yhteiset asetukset. Tämän lisäksi jokaisen kehittäjän projektiin voitiin määrittää omat asetukset käyttämällä codeception.yml-asetustiedostoa. Tähän tiedostoon voitiin mm. asettaa testitietokannan asetukset kuvan 3 mukaisesti, jotta Codeception sai yhteyden haluttuun tietokantaan.



KUVA 3. Codeception.yml asetustiedosto (Niemi 2016-11-04.)

Codeception rakentuu nk. toimijoista (Actors). Toimijat pitävät sisällään kaikki funktiot, jotka Codeception tarjoaa käytetylle testityypille. Käyttöliittymätesteissä käytettiin AcceptanceTester-toimijaa. Toimijan konfigurointitiedostossa voitiin kuvan 4 mukaisesti asettaa käyttöön tarvittavat lisämoduulit, kuten tietokanta- ja WebDriver-moduuli, joiden funktiot Codeception lisää käytettäväksi testeissä AcceptanceTester-luokan objektiin. Molemmat käytetyistä lisämoduleista olivat pakollisia, jotta testejä voitiin suorittaa.



KUVA 4. AcceptanceTester-toimijan asetukset (Niemi 2016-11-04.)

4.1.1 Testien toteutus

Testin tekeminen alkoi testitiedoston generoinnilla. Codeceptionissa on useita eri tiedostomalleja, mutta tämän työn tarpeisiin parhaiten sopi nk. Cest-malli. Cest-tiedostomalliin voi kirjoittaa useita testitapauksia, kun esimerkiksi yksi Cept-mallin mukainen tiedosto vastaa vain yhtä testitapausta. Tiedoston luominen tapahtui kuvan 5 mukaisesti Codeceptionin komentorivikomennolla, johon piti määrittää tiedostomalli, testin tyyppi ja tiedostonimi.

```
PS C:\www\provetnet> .\vendor\bin\codecept g:cest acceptance LoginCest
Test was created in C:\www\provetnet\tests\acceptance\LoginCest.php
PS C:\www\provetnet>
```

KUVA 5. Testitiedoston generointi (Niemi 2016-11-04.)

Komento luo Cest-muotoisen php-tiedoston, johon voidaan kirjoittaa halutut testitapaukset. Perustestin jälkeen ensimmäinen oikea testitapaus oli järjestelmän ajanvarauksen testaaminen. Testitapausta varten tehtiin uusi Cest-tiedosto. Tämä uusi tiedosto tulisi jatkossa pitämään sisällään kaikki ajanvaraukseen liittyvät testitapaukset. Kuvassa 6 nähdään ensimmäinen ajanvaraustesti.

```
    /**
     * @before doLogin
     * @group ajanvaraus
     *
     */
    public function tryToVarataAika(AcceptanceTester $I) {
        $I->wantTo('Luoda ajanvarauksen');

//        pistetään työvuoro tietokantaan jotta aina on työvuoro jolla testata
        $I->haveInDatabase('tyovuoro_taulu', array(
            'paiva' => date('Y-m-d'),
            'kayttaja_id' => 2,
            'tyovuoro_id' => 1
        ));

//        mennään ajanvaraukseen
        $I->openAjanvaraus();

//        valitaan varauksen tehnyt henkilö id:llä ja varataan aika
        $I->selectOption('#henkilot_1', '1');
        $I->wait(1);
        $I->click(AjanvarausPage::$viimeinenVapaaAika);
        $I->wait(1);

//        varauksen tallennus -sivulle tekemään varaus
        $I->switchToLastWindow();
        $this->tallennaVarausPage->lisaaAjanvaraus();
        $I->wait(1);
        $I->see('Testi Omistaja', AjanvarausPage::$varattuAikaRivi);
    }
```

KUVA 6. Ensimmäinen ajanvaraustesti (Niemi 2016-11-04.)

Ajanvaraustestin (kuva 6) ensimmäisillä riveillä määritettiin testiä ennen suoritettava doLogin-funktio @before-annotaatiolla sekä ryhmä, johon testitapaus kuuluu @group-annotaatiolla. Testitapauksen alussa ilmoitettiin, mitä testitapaus haluaa tehdä wantTo-funktiolla, joka tulostaa sille annetun tekstin testiä suoritettaessa komentoriville.

Jotta ohjelmistoon voitaisiin tehdä ajanvaraus, täytyi ohjelmiston simuloida työntekijän työvuoro, jolle ajanvarauksen voisi tehdä. Tämä toteutettiin käyttämällä Codeceptionin DB-modulin haveInDatabase-funktiota, jolla voitiin testin ajaksi laittaa tietokantaan tarvittavat tiedot. Työvuoroa ei lisätty käyttöliittymän kautta, koska jokaisen käyttöliittymätoiminnon testi halutaan eristää omilleen. Näin jos testissä tulee virhe, tiedetään missä toiminnossa virhe on.

Koska kuvassa 7 nähtävän ohjelmiston päävalikon toimintoja tultaisiin käyttämään useasti eri testeissä, päätettiin toiminnot tehdä helposti käytettäviksi Codeceptionin Helper-mallilla. Jokainen valikon nappi toteutettiin Helper-luokkaan funktiona, joka palauttaa halutun napin xpath-osoitteen. Tällaisella funktiorakenteella oli helpompi toteuttaa valikon dropdown-elementtien alustus, verrattuna siihen, että jokaisen valikon napin uniikki tunniste olisi kirjoitettu omaan staattiseen muuttujaansa. Tämä tarkoitti sitä, että testeissä nappeja käytettäisiin funktiokutsulla sen sijaan, että kutsuttaisiin staattista muuttujaa. Jotta nappien käyttö olisi vielä selkeämpää, toteutettiin AcceptanceTester-luokkaan funktiot, joilla voitiin avata haluttu toiminto päävalikosta. Näin päävalikon toiminnot olivat käytettävissä $I-muuttujassa jokaisessa testissä.

KUVA 7. Ohjelmiston päävalikko (Finnish Net Solutions 2016-11-04.)

Kuvassa 8 AcceptanceTester-luokkaan toteutettu openAjanvaraus-funktio, jossa kutsutaan päävalikon Helper-luokan ajanvarausnapin osoitteen palauttavaa funktiota.

```
/**
 * Avaa ajanvaraussivun
 */
public function openAjanvaraus() {
    $I = $this;

    $I->click(MegaMenu::MenuAjanvaraus());
    $I->wait(1);
    $I->switchToLastWindow();
}
```

KUVA 8. Ajanvarauksen avausfunktio (Niemi 2016-11-04.)

Kun käyttöliittymätestissä halutaan vaikuttaa johonkin käyttöliittymän elementtiin, tarvitsee testiä varten tietää kyseisen elementin uniikki tunniste. Yleensä tunnisteena käytetään elementin id-attribuuttia tai xpath-osoitetta. Jos useassa testissä tarvitaan samaa elementtiä, voi koodiin tulla helposti toistoa. Tämän ehkäisemiseksi käytettiin Codeceptionin sisälle rakennettua page object -

suunnittelumallia. Jokaisesta ohjelmiston testatusta sivusta tehtiin tällainen page object, johon kartoitettiin tarvittavat sivun elementit ja toteutettiin sivulla mahdollisesti käytettävät funktiot. Esimerkiksi kuvan 6 testitapauksessa käytettävän AjanvarausPage-page objectin osatoteutus on nähtävissä kuvassa 9. Näin joka kerta kun elementtiä käytettiin, viitattiin vain page objectin vastaavaan staattiseen muutujaan, joka piti sisällään elementin tunnisteen. Tämä lisäsi myös testien ylläpidettävyyttä esim. tilanteessa, jossa elementin tunniste muuttuu.

```php
<?php
namespace Page\Acceptance;

/**
 * Ajanvaraussivu
 */
class Ajanvaraus
{
    // include url of current page
    public static $URL = '';

    /**
     * Declare UI map for this page here. CSS or XPath allowed.
     * public static $usernameField = '#username';
     * public static $formSubmitButton = "#mainForm input[type=submit]";
     */

    public static $viimeinenVapaaAika = '(//*[@id="tiedot_1"]/div[@class="vapaarivi"])[last()]/div[2]/a';
    public static $varattuAikaRivi = '//*[@id="tiedot_1"]/div[@class="rivi"]';
    public static $varattuAikaLinkki = '//*[@id="tiedot_1"]/div[@class="rivi"]/div/b/a';
    public static $paivaInfoField = '#paivainfo';
    public static $paivitaSivuButton = '//input[@name="paivitasivu"]';
    public static $muokkaaAjanvarauspohjiaButton = '//img[@title="Muokkaa ajanvarauspohjia"]';
```

KUVA 9. Ajanvaraus-page object (Niemi 2016-11-04.)

Toinen kohta jossa huomattiin tulevan koodin toistoa, oli useassa testissä käytettävät yleiset funktiot. Ennen jokaista testiä, täytyi kirjautua ohjelmistoon, jolloin sama sisäänkirjautumiskoodi kirjoitettiin jokaisen testin alkuun. Tämä ratkaistiin kirjoittamalla AcceptanceTester-luokkaan sisäänkirjausfunktio, koska kaikki AcceptanceTester-luokan funktiot ovat käytössä testeissä. Tämän jälkeen sisäänkirjausfunktio asetettiin suoritettavaksi jokaisen testiluokan before-funktioon (kuva 10), jonka Codeception suorittaa ennen jokaista sen luokan testiä.

```php
public function _before(AcceptanceTester $I)
{
    $I->login('käyttäjänimi', 'salasana');
    $I->openKoodistot();
    $I->click(YllapitotyokalutMenu::$kutsutyypit);
    $I->waitForElement(Kutsutyyppihallinta::$grid, 3);
}
```

KUVA 10. Testiluokan before-funktio (Niemi 2016-11-04.)

Jos useampi testi suoritetaan peräkkäin, tarvitsee niistä vain ensimmäisessä kirjautua sisään. Edellä mainitulla tavalla sisäänkirjautuminen kuitenkin suoritettiin jokaisen testin alussa huolimatta siitä, oliko sisäänkirjautuminen edellisissä testeissä jo suoritettu. Tätä varten sisäänkirjausfunktioon täytyi lisätä Codeceptionin Session Snapshot -toiminto. Session Snapshot ottaa talteen testissä käytettävän session, jonka voi sitten myöhemmissä testeissä ottaa käyttöön. Kun toiminto lisättiin kuvassa 11

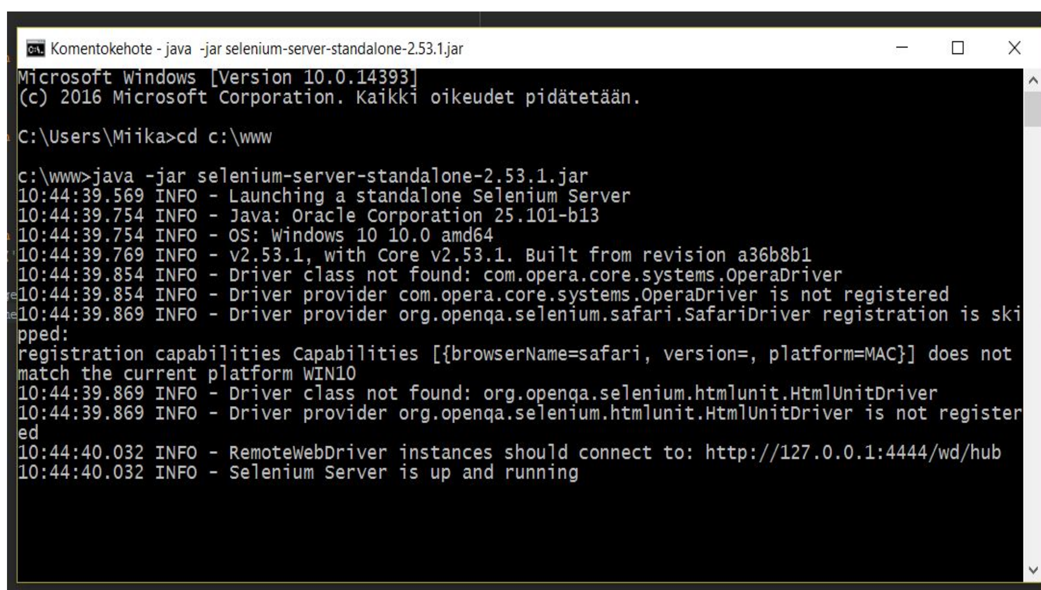
näkyvällä tavalla sisäänkirjautumiseen, tehtiin sisäänkirjaus vain ensimmäisessä testissä, vaikka testejä olisi suoritettu useita samalla kertaa.

```
/**
 * Yrittää kirjautua sisään annetuilla tunnuksilla. Tallentaa myös session snapshotin
 * eli sisäänkirjautuminen suoritetaan vain ensimmäistä testiä suoritettaessa vaikka kutsutaankin
 * ennen jokaista testiä.
 *
 * @param type $user käyttäjänimi
 * @param type $pass salasana
 */
public function login($user, $pass) {
    $I = $this;

//      Jos on jo logattu sisään, ei tehdä sitä uudestaan
    if($I->loadSessionSnapshot('login')) {
        return;
    }

    $I->wantTo('login with specified username and password');
    $I->amOnPage('index.php');
    $I->submitForm("//form[@name='loginform']", [
            'username' => $user,
            'password' => $pass
    ], 'do_login');
    // $I->fillField('username', 'testi');
    // $I->fillField('password', 'testi!');
    // $I->click('do_login');
    $I->saveSessionSnapshot('login');
}
```

KUVA 11. AcceptanceTester-luokan login-funktio (Niemi 2016-11-04.)

4.1.2 Testien suorittaminen

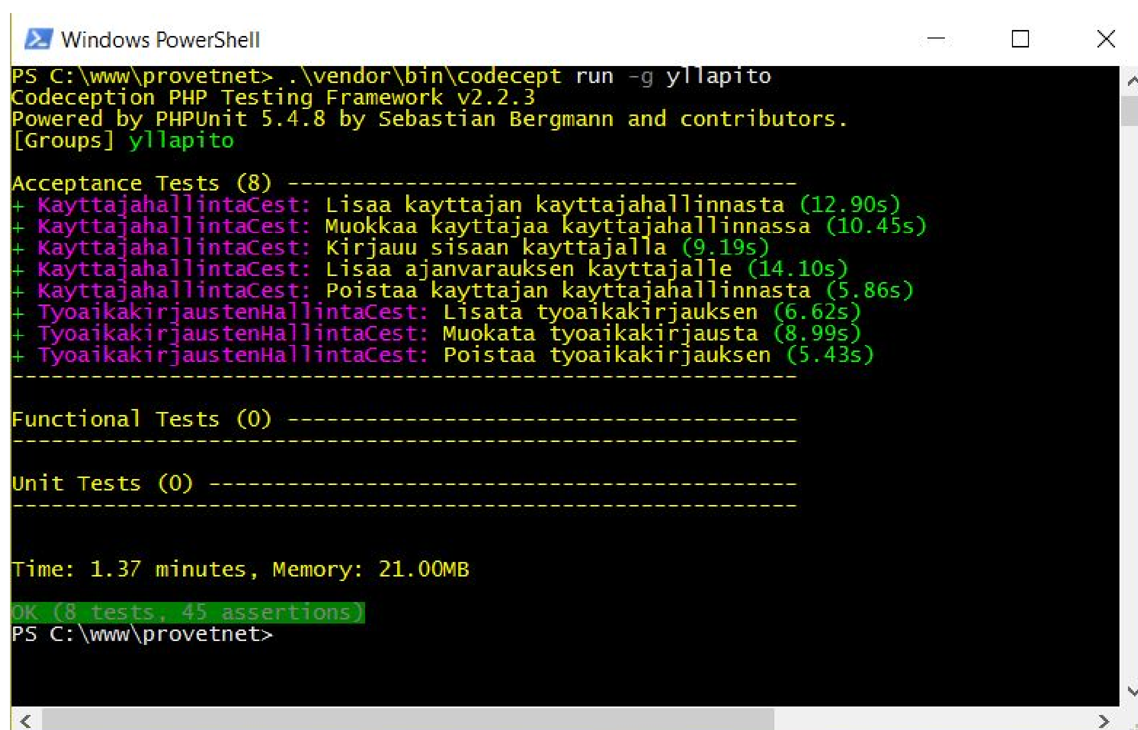
Ennen testien suorittamista, täytyi käynnistää Selenium WebDriverin käyttämä standalone palvelin, jonka avulla testien komennot välitetään selaimelle. Palvelin ladattiin Seleniumin internet-sivulta .jar-tyyppisenä pakettina. Palvelin käynnistettiin komentorivillä Java-komennolla kuvan 12 mukaisella tavalla.



KUVA 12. Selenium standalone palvelimen käynnistys (Niemi 2016-11-04.)

Tämän jälkeen testi voitiin ajaa komentorivikomennolla *codecept run.* Run-toiminnon parametreilla voitiin vaikuttaa siihen, mitä testejä ajettiin. Kerralla voitiin ajaa joko yksittäinen testi, testiryhmiä tai kaikki jonkin testityypin testit.

Tavallisesti testitilanteessa ohjelmiston käyttöliittymästä haluttiin testata jokin suurempi kokonaisuus kerralla. Joissain tapauksissa tämä tarkoitti usean eri testitapauksen suorittamista yhdellä kertaa. Jotta tämä olisi helppoa, testitapaukset jaettiin ryhmiin sen mukaan, mitä osaa käyttöliittymästä ne testaavat. Kuvassa 13 esimerkki ylläpito-testiryhmän suorittamisesta. Suorituksen jälkeen Codeception raportoi komentoriville jokaisen testitapauksen tuloksen.



KUVA 13. Testiryhmän suorittaminen (Niemi 2016-11-04.)

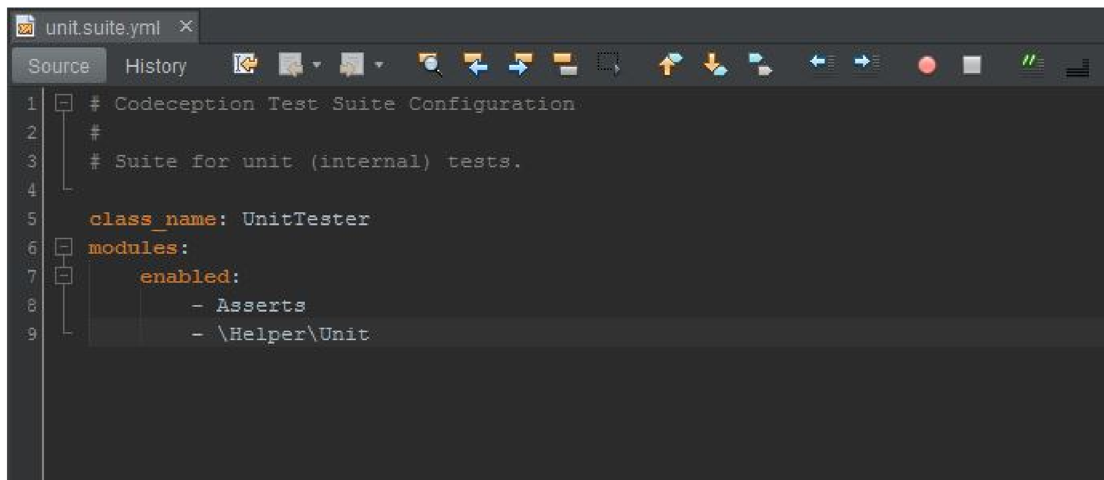
4.2 Yksikkötestit

Yksikkötestejä suunniteltaessa todettiin, että tarvetta laajalle yksikkötestaukselle ei sillä hetkellä ollut, joten yksikkötestien osa työssä päätettiin jättää pienemmäksi. Testattavan ohjelmiston apufunktiokirjastosta valittiin muutama yleisesti käytetty funktio, joille toteutettiin yksikkötestit esimerkin vuoksi.

4.2.1 Testien pohjustus

Koska yksikkö- ja käyttöliittymätestien toteutukseen käytettiin samaa työkalua ja lähes kaikki alkukonfiguroinnit oli jo tehty käyttöliittymätestien aloituksessa, ei yksikkötestejä varten tarvittu suuria alkutoimenpiteitä. Yksikkötesteissä käytettävä UnitTester-toimijan asetukset sopivat testien käyttöön

ns. out of the box eli asetuksia ei tarvinnut oletuksista muuttaa. Oletuksena UnitTester-toimijan toimintoihin lisättiin Asserts-moduulin ja Unit Helper-luokan sisällöt. Kuvassa 14 on nähtävissä UnitTester-toimijan konfigurointitiedosto.



KUVA 14. UnitTester-toimijan konfigurointitiedosto (Niemi 2016-11-04.)

4.2.2 Testien toteutus

Testeissä käytettiin samaa Cest-tiedostomallia kuin käyttöliitymätesteissäkin. Ensimmäiseksi testattavaksi funktioksi valittiin sähköpostiosoitteen validointifunktio. Funktio otti parametrina sisälleen tarkistettavan sähköpostiosoitteen ja tarkoitus oli palauttaa joko true tai false sen mukaan, oliko sähköpostiosoite validi vai ei. Funktion ollessa näin yksinkertainen myös sille testin kirjoittaminen oli triviaalia, kuten kuvasta 15 nähdään.

```
3  public function emailValidation(AcceptanceTester $I) {
4      $I->wantTo('Test email validation returns true on valid email address');
5
6      $email = 'test.email@email.com';
7      $result = validate_email($email);
8      $I->assertTrue($result);
9  }
```

KUVA 15. Sähköpostivalidoinnin yksikkötesti (Niemi 2016-11-04.)

Testin alussa muuttujaan alustettiin haluttu sähköpostiosoite ja sillä kutsuttiin testattavaa validointifunktiota. Codeceptionin Asserts-moduulin assertTrue-funktiolla tarkistettiin, että sähköpostiosoitteen ollessa validi funktio palauttaa arvon true. Seuraavaksi testattiin myös, että funktio palauttaa halutut arvot sekä tyhjällä parametrilla että epävalidilla sähköpostiosoitteella.

4.2.3 Testien suorittaminen

Testit suoritettiin hyvin samalla tavalla kuin käyttöliittymätestit, paitsi että testejä ei suoritettu internetselainta vasten. Tällöin tarvetta Selenium-palvelimelle ei ollut. Koska yksikkötestit olivat nopeita,

suoritettiin ne aina kaikki yhdellä kertaa kuvassa 16 esitetyllä tavalla. Codeception ilmoitti testien onnistumisesta ja epäonnistumisesta komentorivillä samalla tavalla kuin käyttöliittymätesteissä.

```
PS C:\www\provetnet> .\vendor\bin\codecept run unit
Codeception PHP Testing Framework v2.2.3
Powered by PHPUnit 5.4.8 by Sebastian Bergmann and contributors.

Unit Tests (2) ---------------------------------------------
+ ToolsGeneralCest: Test email validation returns true on valid email address (0.01s)
+ ToolsGeneralCest: Test email validation returns false on invalid email address (0.01s)
------------------------------------------------------------


Time: 397 ms, Memory: 13.00MB

OK (2 tests, 2 assertions)
PS C:\www\provetnet>
```

KUVA 16. Yksikkötestien suoritus (Niemi 2016-11-04.)

4.3 Kuormitustestit

Kuormitustestien tarkoituksena oli testata, kuinka ohjelmiston tuleva uusi etusivu käyttäytyy tavanomaisten käyttäjämäärien alla. Kun testit oli määrä aloittaa, ei uusi etusivu kuitenkaan ollut vielä siinä pisteessä, että siihen olisi voinut vielä kuormitustestejä toteuttaa. Päätettiin, että kuormitustesti toteutetaan ensin vanhalle etusivulle ja muokataan sitten sopimaan yhteen uuden etusivun kanssa, jos aikaa on.

4.3.1 Testien pohjustus

Kuormitustestien valmistelut aloitettiin asentamalla Gatling. Asennus onnistui helposti vain lataamalla ja purkamalla zip-paketti. Työkoneelle oli jo asennettu uusin JDK (Java Development Kit), joten sitä ei tarvinnut ottaa huomioon asennuksessa. Projektissa käytettävä tekstin enkoodaus ja muut tarvittavat asetukset asetettiin gatling.conf-tiedostoon.

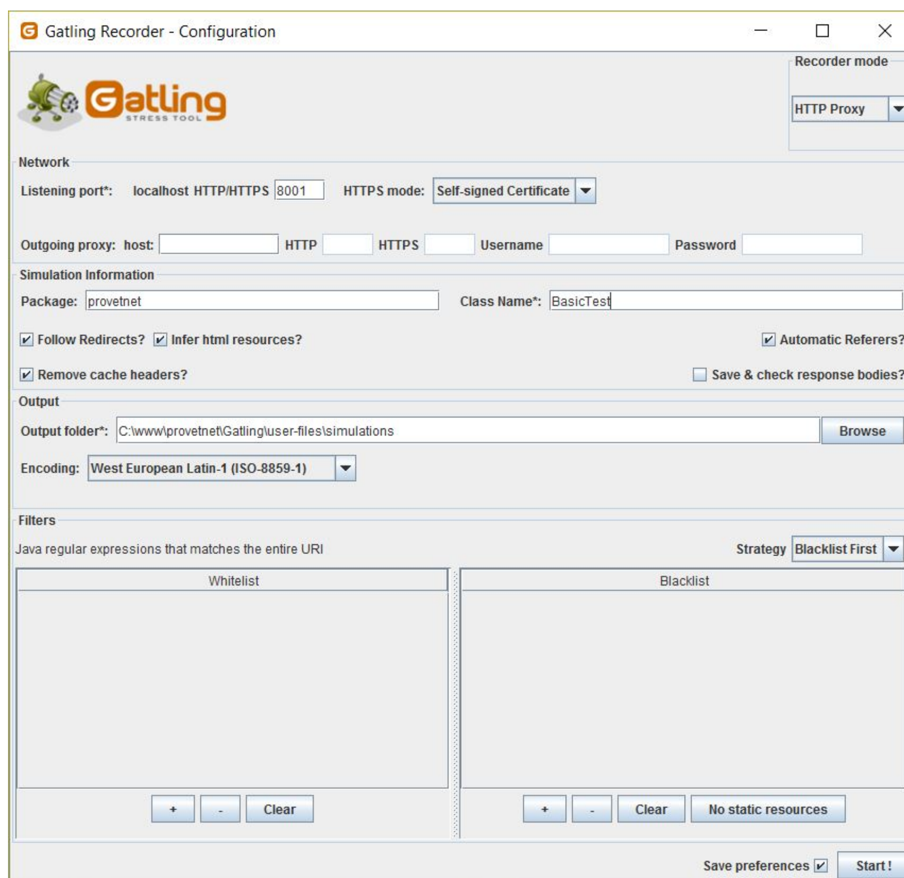
Kuormitustestiä varten suunniteltiin esimerkkiskenaario, mitä testi voisi noudattaa. Skenaarion tuli mukailla keskimääräisen käyttäjän toimintoja ohjelmistossa. Alun perin kehiteltiin yksi käyttäjäskenaario, jota testeissä käytettäisiin. Skeenariossa käyttäjä kirjautuisi sisään ohjelmistoon, etsisi hakutoiminnolla potilaan, kirjaisi potilaan sisään järjestelmään ja sen jälkeen kuittaisi potilaan kotiutetuksi. Tarkemmalla tarkastelulla tuli ilmi, että tämä skenaario ei tulisi toimimaan kuormitustesteissä, sillä jokaisen sisäänkirjattavan potilaan tulisi olla eri potilas. Jos kuormitustesteissä käytettäisiin esimerkiksi sataa yhtäaikaista käyttäjää, olisi eri potilaiden kirjoittaminen testeihin liian vaikeaa. Lopullisessa testiskenaariossa päädyttiin ratkaisuun, jossa pienempi osa testikäyttäjistä noudattaisivat edellämainittua skenaariota ja suurempi osa vain kirjautuisi sisään ohjelmistoon, ladaten näin ohjelmiston etusivun, ja pienen odottelun jälkeen kirjautuisivat ulos. Kuvassa 17 skenaario, johon päädyttiin.

```
SKENAARIOT

1. Sisäänkirjaajat
        -Käyttäjä kirjautuu sisään
        -Käyttäjä etsii järjestelmästä potilaan
        -Käyttäjä kirjaa potilaan sisään
        -Käyttäjä kirjaa potilaan ulos
        -Käyttäjä kirjautuu ulos järjestelmästä

2. Etusivun lataajat
        -Käyttäjä kirjautuu sisään
        -Käyttäjä odottaa etusivulla
        -Käyttäjä kirjautuu ulos
```

KUVA 17. Kuormitustestien skenaariosuunnitelma (Niemi 2016-11-04.)

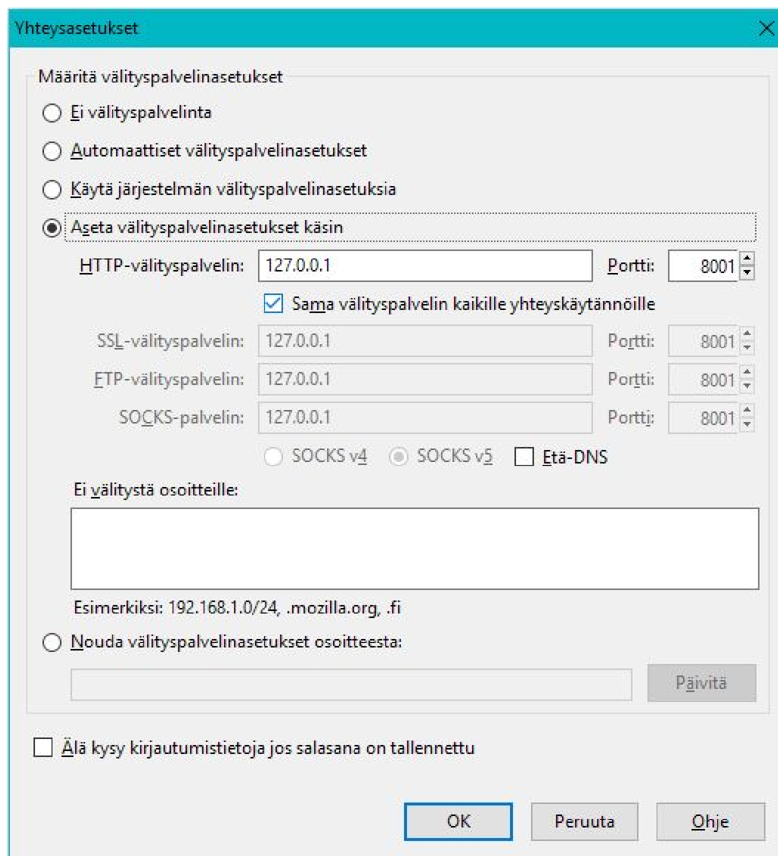
4.3.2 Testien toteutus

Testien luominen tapahtui nauhoittamalla simulaatio testiskenaarion osista Gatlingin Recorder-työkalulla. Työkalu käynnistettiin komentoriviltä komennolla: %GATLING_HOME%\bin\recorder.bat. Kun Recorder oli käynnissä, asetettiin nauhoittamiseen käytettävä internetselain kuuntelemaan samaa porttia, jota Recorder käyttää. Recorderissa oli tärkeää asettaa käytettävä merkistö, joka tässä tapauksessa oli ISO-8859-1. Muista asetuksista vaihdettiin simulaation paketti ja nimi. Yhteen pakettiin voi nauhoittaa useita simulaatioita. Kuvassa 18 nähdään Gatling Recorder -työkalu.



KUVA 18. Gatling Recorder (Gatling.io 2016.)

Jotta Recorder toimisi oikein, täytyi myös käytettävä selain asettaa käyttämään Recorderille määritettyä porttia. Kuvassa 19 nähdään tarvittavat selainasetukset.



KUVA 19. Mozilla Firefoxin asetukset simulaatioiden nauhoitukseen (Mozilla.org? 2016.)

Kun asetukset olivat kunnossa, aloitettiin testin nauhoittaminen painamalla Recorder-ikkunasta Start-nappia. Tämän jälkeen selaimessa suoritettiin määritetyn testiskenaarion mukaiset toiminnot. Kun toiminnot oli suoritettu, Recorder pysäytettiin ja Gatling loi automaattisesti suoritetuista toiminnoista simulaation (kuva 20).

```
val scn = scenario("Scenario Name") // A scenario is a chain of requests and pauses
  .exec(http("request_1")
    .get("/"))
  .pause(7) // Note that Gatling has recorded real time pauses
  .exec(http("request_2")
    .get("/computers?f=macbook"))
  .pause(2)
  .exec(http("request_3")
    .get("/computers/6"))
  .pause(3)
  .exec(http("request_4")
    .get("/"))
  .pause(2)
  .exec(http("request_5")
    .get("/computers?p=1"))
  .pause(670 milliseconds)
  .exec(http("request_6")
    .get("/computers?p=2"))
  .pause(629 milliseconds)
  .exec(http("request_7")
    .get("/computers?p=3"))
  .pause(734 milliseconds)
  .exec(http("request_8")
    .get("/computers?p=4"))
  .pause(5)
  .exec(http("request_9")
    .get("/computers/new"))
  .pause(1)
  .exec(http("request_10") // Here's an example of a POST request
    .post("/computers")
    .headers(headers_10)
    .formParam("name", "Beautiful Computer")
    .formParam("introduced", "2012-05-30")
    .formParam("discontinued", "")
    .formParam("company", "37"))

setUp(scn.inject(atOnceUsers(1)).protocols(httpConf))
```

KUVA 20. Gatling: esimerkkisimulaatio (Gatling.io 2016.)

Tämä nauhoitettu simulaatio simuloi yhden käyttäjän toimia testattavassa ohjelmistossa. Testiä tuli vielä muokata niin, että käyttäjiä olisi useita yhtäaikaisesti. Tämä tapahtui Gatlingin rampUsers-toiminnolla, jolla pystyttiin syöttämään asteittain simulaation mukaisia testikäyttäjiä ohjelmistoon.

Ensimmäisten testien jälkeen huomattiin, että määritelty testiskenaario oli vaikea toteuttaa. Kun testejä suoritetaan ja ensimmäinen testikäyttäjä aloittaa skenaarion kirjaamalla potilaan sisään, ei seuraavat käyttäjät voi kirjata samaa potilasta sisään. Tilanteessa päätettiin käyttää Gatlingin Feeder-toimintoa, jolla pystytään syöttämään testeihin dynaamista dataa. Käytännössä tämä tarkoitti potilaiden listaamista tekstitiedostoon JSON-muodossa, josta Gatling kävi järjestyksessä hakemassa eri potilaiden tiedot, joiden perusteella testit pystyttiin suorittamaan. Kuvassa 21 esimerkki Feeder-tiedoston sisällöstä.

```
[
    {
        "kayttaja_id":"1",
        "elain_id":"1",
        "sisaanKirjaus":"true"
    },
    {
        "kayttaja_id":"1",
        "elain_id":"2",
        "sisaanKirjaus":"true"
    }
]
```

KUVA 21. Feeder-esimerkkitiedosto (Niemi 2016-11-04.)

4.3.3 Testien suorittaminen

Kuormitustestien suorittaminen tapahtui kokonaan Windowsin komentoriviltä. Simulaation käynnistämiseksi täytyi navigoida Gatlingin tiedostojuureen ja käynnistää Gatling komennolla "%GATLING_HOME%\bin\gatling.bat". Tällä komennolla saatiin komentoriville lista testisimulaatioista, jotka voitiin käynnistää. Listasta valittiin aiemmin toteutettu simulaatio. Kuvassa 22 esimerkki komentorivinäkymästä testin ollessa käynnissä.

```
Simulation computerdatabase.BasicSimulation started...

================================================================================
2016-11-17 09:39:32                                           5s elapsed
---- Scenario Name -------------------------------------------------------------
[--------------------------------------------------------------------------]  0%
          waiting: 0      / active: 1      / done:0
---- Requests ------------------------------------------------------------------
> Global                                                   (OK=2      KO=0     )
> request_1                                                (OK=1      KO=0     )
> request_1 Redirect 1                                     (OK=1      KO=0     )
================================================================================


================================================================================
2016-11-17 09:39:37                                          10s elapsed
---- Scenario Name -------------------------------------------------------------
[--------------------------------------------------------------------------]  0%
          waiting: 0      / active: 1      / done:0
---- Requests ------------------------------------------------------------------
> Global                                                   (OK=3      KO=0     )
> request_1                                                (OK=1      KO=0     )
> request_1 Redirect 1                                     (OK=1      KO=0     )
> request_2                                                (OK=1      KO=0     )
================================================================================


================================================================================
2016-11-17 09:39:42                                          15s elapsed
---- Scenario Name -------------------------------------------------------------
[--------------------------------------------------------------------------]  0%
          waiting: 0      / active: 1      / done:0
---- Requests ------------------------------------------------------------------
> Global                                                   (OK=6      KO=0     )
> request_1                                                (OK=1      KO=0     )
> request_1 Redirect 1                                     (OK=1      KO=0     )
> request_2                                                (OK=1      KO=0     )
> request_3                                                (OK=1      KO=0     )
> request_4                                                (OK=1      KO=0     )
> request_4 Redirect 1                                     (OK=1      KO=0     )
================================================================================
```

KUVA 22. Komentorivi Gatling-simulaation aikana (Niemi 2016-11-04.)

Testisimulaation jälkeen Gatling muodosti tuloksistaan helppolukuiset tulossivut. Näiltä sivuilta katsottiin mm. se, kuinka pitkään keskimäärin pyynnöt odottivat vastauksia. Pyyntöjen kestosta voitiin

päätellä, kuinka kovasti määritellyt käyttäjämäärät rasittivat testattavaa järjestelmää. Kuviossa 1 nähdään erään simulaation pyyntöjen aikajakauma, jossa käyttäjiä syötettiin järjestelmään n. sata kappaletta eri aikoihin Gatlingin rampUsers-toiminnolla.

KUVIO 1. Testisimulaation pyyntöjen aikajakauma (Niemi 2016-11-04.)

Kuvasta voitiin päätellä, että järjestelmä oli melko kovassa rasituksessa. Suurin osa pyynnöistä odotti vastausta yli sekunnin, mikä näkyisi käyttäjälle järjestelmän hitautena latausaikojen pidentyessä. Positiivista tuloksissa kuitenkin oli, että kaikki pyynnöt saivat vastauksen eikä käyttäjille olisi näin tullut varsinaisia katkoksia järjestelmää käytettäessä.

Tuloksissa arveluttavaa oli kuitekin se, että tosielämässä järjestelmä voi käsitellä kaksinkertaisesti suurempia yhtäaikaisia käyttäjämääriä. Tämän perusteella arvioitiin, että toteutettu testisimulaatio ei ehkä vastaa järjestelmän keskimääräistä käyttötapausta. Tosielämässä käyttäjät todennäköisesti pitävät pidempiä taukoja järjestelmän käytössä, kuin testisimulaatiossa pidettiin. Esimerkiksi käyttäjä saattaa kirjata potilaan sisään järjestelmään, tehdä välissä toimenpiteitä ja vasta pidemmän tauon jälkeen kirjata potilaan ulos järjestelmästä. Tämä olisi pitänyt ottaa huomioon testiskenaariota suunniteltaessa, vaikkakin se olisi ollut hankalaa. Testiskenaarion oli pitänyt olla pitkäkestoisempi ja käyttäjien toimia olisi pitänyt jakaa pidemmälle aikavälille. Tällöin oltaisiin todennäköisesti saatu parempi kuva siitä, miten ohjelmiston etusivu tulisi käyttäytymään tosielämän kuormituksessa.

Kuormitustestin oltua tässä pisteessä, oli uusi etusivu jo todettu toimivaksi ja julkaistu. Näin ollen kuormitustesti ei ehtinyt ottaa kantaa päivitetyn etusivun toimintaan ja testin kehitys pysäytettiin.

5 YHTEENVETO

Työn tavoitteena oli toteuttaa pohja Provet Net -eläinlääkäriohjelmiston automaattiseen testaukseen. Ohjelmistoon säännöllisesti suoritettavista, ennalta määrätyistä, manuaalisista käyttöliittymätesteistä tuli automatisoida mahdollisimman suuri osa. Lisäksi ohjelmistoon haluttiin toteutettavan automaattiset yksikkö- ja kuormitustestiesimerkit, joiden avulla näitä voitaisiin jatkossa toteuttaa enemmän. Työn selvä päätavoite oli kuitenkin käyttöliittymätestien automatisointi.

Käyttöliittymä- ja yksikkötesteihin valittiin käytettäväksi PHPUnitiin pohjautuvaa Codeception-testauskehystä, koska samalla työkalulla saatiin toteutettua molemmat testit. Käyttöliittymätesteissä haastavinta oli toteuttaa testit valmiiseen ohjelmistoon, jota ei ollut alun perin suunniteltu automaattisia testejä varten. Joitain osia ohjelmiston käyttöliittymätesteistä ei voinut automatisoida ollenkaan, koska jotkin ohjelmistossa käytetyt vanhat komponentit eivät tukeneet ohjelmallista testausta. Haastavaa oli myös muistaa toteuttaa parhaita testauskäytäntöjä jokaisessa testitapauksessa, millä voisi vähentää koodin toistoa ja lisätä ylläpidettävyyttä. Käyttöliittymätesteistä jatkokehitykseen jäi testien suorittamisen automatisointi, joka todennäköisesti toteutetaan pian.

Yksikkötestejä aloitettaessa todettiin, ettei niillä välttämättä ole ohjelmiston testauksen kannalta suurta merkitystä ja pääpaino testien automatisoinnissa siirrettiin käyttöliittymätesteihin. Yksikkötesteissä päädyttiin säätämään testausympäristö valmiiksi ja toteuttamaan esimerkin vuoksi muutama testitapaus, jos myöhemmin tarve yksikkötestaamiseen syntyy. Välitöntä jatkokehitystarvetta yksikkötesteille ei ole.

Kuormitustesteillä haluttiin varmistaa, että ohjelmistoon julkaistava uusi etusivu toimisi kunnolla, kun siihen kohdistetaan suuri käyttäjäkuorma. Kuormitustestit päätettiin toteuttaa Gatling-testaustyökalulla. Työkalun asennus ja ensimmäisten testien toteuttaminen olivat helppoa. Ongelmaksi muodostui Scala-ohjelmointikieli, johon Gatling pohjautuu. Uuden ohjelmointikielen omaksuminen oli aluksi vaikeaa ja hidasti työkalun nauhoittamalla luotujen testien jalostusta. Haastavaa oli myös tyypillisen käyttäjäskenaarion suunnittelu. Toteutettu skenaario on luultavasti hieman raskaampi verrattuna oikeiden käyttäjien toimiin ohjelman sisällä. Valitettavasti toteutettu kuormitustesti ei valmistunut ennen uuden etusivun julkaisua. Tämä kuormitustesti voi kuitenkin tulevaisuudessa toimia esimerkkinä, jos kuormitustestejä päätetään toteuttaa lisää ohjelmiston eri osille.

Olen tyytyväinen siitä, miten testit päätettiin toteuttaa, enkä jälkikäteen ajateltuna muuttaisi mitään radikaalisti. Käyttöliittymätestit olisi voinut suunnitella paremmin automaattisesti suoritettaviksi. Tämän olisi voinut toteuttaa parantamalla testien virheidensietokykyä tilanteissa, jossa testi huomaa virheen.

Työ opetti paljon ohjelmistotestauksesta ja sen tärkeydestä. Projektien kasvaessa myös testien automatisoinnin tärkeys korostuu. Vaikka testien automatisointi vie aikaa, maksaa se itsensä takaisin hyvin nopeasti testausaikojen lyhentyessä merkittävästi ja ohjelmiston vakauden kasvaessa. Olen varma, että tarvitsen tulevaisuudessa työn aikana opittuja työkaluja ja käytäntöjä.

LÄHTEET

COMPOSER DOCUMENTATION 2017. Introduction. [Viitattu 2016-09-28]. Saatavissa: https://getcomposer.org/doc/00-intro.md

MCFARLIN, Tom. 2012-06-19. The Beginner's Guide to Unit Testing: What Is Unit Testing?. [Viitattu 2016-09-27]. Saatavissa: https://code.tutsplus.com/articles/the-beginners-guide-to-unit-testing-what-is-unit-testing--wp-25728

MEIER, J.D., FARRE, C., PRASHANT, B., BARBER, S., REA, DENNIS. 2007-09. Performance Testing Guidance for Web Applications. Chapter 18. [Viitattu 2017-05-08]. Saatavissa: https://msdn.microsoft.com/en-us/library/bb924374.aspx

ODERSKY, Martin 2017. What Is Scala?. [Viitattu 2017-04-24]. Saatavissa: https://www.scala-lang.org/what-is-scala.html

PERKINS, Andrew. 2014-01-02. Acceptance Testing With Codeception. [Viitattu 2016-09-28]. Saatavissa: https://code.tutsplus.com/tutorials/acceptance-testing-with-codeception--net-36337

PROVET 2017. [Viitattu 2017-04-01]. Saatavissa: http://www.provet.fi/

ROUSE, Margaret a. 2017-01. JAR file (Java ARchive). [Viitattu 2016-09-28]. Saatavissa: http://www.theserverside.com/definition/JAR-file-Java-ARchive

ROUSE, Margaret b. 2014-02. GUI testing (graphical user interface testing). [Viitattu 2017-04-24]. Saatavissa: http://whatis.techtarget.com/definition/GUI-testing-graphical-user-interface-testing

Scala (programmin language). 2016-09-24 (last updated). [Viitattu 2016-09-28]. Saatavissa: https://en.wikipedia.org/wiki/Scala_(programming_language)

SELENIUM WEBDRIVER. 2016-09-04 (last updated). [Viitattu 2016-09-28]. Saatavissa: http://www.seleniumhq.org/docs/03_webdriver.jsp

TUTORIALSPOINT 2017. Testing dictionary. [Viitattu 2017-04-24]. Saatavissa: https://www.tutorialspoint.com/software_testing_dictionary/automated_software_testing.htm

URBONAS, Rimvydas. 2013-11-04. Automated User Interface Testing. [Viitattu 2016-09-26.] Saatavissa: https://www.devbridge.com/articles/automated-user-interface-testing/

VEPSÄLÄINEN, Juho. 2015-01-11. What is YAML? What does it do?. [Viitattu 2016-09-28]. Saatavissa: https://www.quora.com/What-is-YAML-What-does-it-do