Automaatiotestit Katalon Studiolla

Ammattikorkeakoulututkinnon opinnäytetyö

Riihimäen kampus, Tieto- ja viestintätekniikka

Kevät, 2019

Eemeli Meriläinen

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tieto- ja viestintätekniikka
Riihimäki

Tekijä Eemeli Meriläinen **Vuosi** 2019

Työn nimi Automaatiotestit Katalon Studiolla

Työn ohjaaja/t Petri Kuittinen

TIIVISTELMÄ

Tämän opinnäytetyön tavoitteena oli luoda automatisoituja testejä käyttäen Katalon Studio testausalustaa. Opinnäytetyössä käydään läpi eri menetelmiä, joilla testejä voi luoda Katalon Studiossa.

Opinnäytetyössä käydään läpi testausta yleisellä tasolla. Lisäksi käydään läpi ohjelmistotestauksen eri vaiheita ja osa-alueita. Katalon Studiosta käydään läpi tärkeimpiä toimintoja ja ominaisuuksia.

Soveltavasssa osuudessa kerrotaan miten automatisoitu testi saadaan kirjoitettua Katalon Studion avulla. Testin kirjoittamisen jälkeen se ajetaan ja tarkastellaan testin antamia tuloksia.

Avainsanat Katalon Studio, ohjelmistotestaus

Sivut 23 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Information and Communication Technology
Riihimäki

Author Eemeli Meriläinen **Year** 2019

Subject Test automation with Katalon Studio

Supervisors Petri Kuittinen

ABSTRACT

The goal of this thesis project was to create fully automated tests using Katalon Studio automation framework. This thesis goes over different testing methods that can be used to create tests in Katalon Studio.

Testing is explained at a general level in the theory section. This part also covers the different phases and sectors of software testing. Katalon Studio's most important features and functions will be examined in closer detail.

The practical part of the thesis discusses the different methods that can be used in Katalon Studio to create automation tests. After writing the test it will be executed and the results can be examined.

Keywords Katalon Studio, software testing

Pages 23 pages

SISÄLLYS

1 JOHDANTO

Automaatiotestaus on sovelluksen tai sovellusten automaattista testaamista. Ennen testien automatisointia jouduttiin manuaalisesti etsimään sovelluksista virheitä, joka on työlästä ja aikaa vievää. Testauksen automatisointi parantaa testauksen nopeutta, vähentää inhimillisiä virheitä ja parantaa tuottavuutta pitemmällä aikavälillä.

Opinnäytetyön tavoitteena on määrittää, kuinka Katalon Studio soveltuu websovellusten testaamiseen ja minkälaisia ominaisuuksia se sisältää. Opinnäytetyön teoriaosissa käydään yleisesti läpi testausautomaatiota. Työssä käydään läpi testauksen automatisaation hyödyt, testauksen eri vaiheita ja eri testausmenetelmiä.

Testauksen vaiheet ovat vaatimusanalyysi, testisuunnitelma, testicasen suunnittelu ja kehitys, testiympäristön luonti, testien ajaminen ja tulosten tarkastelu. Opinnäytetyön soveltava osio keskittyy testicasen kirjoittamiseen ja testien ajamiseen.

Katalon Studio on ilmainen automaatiotestausratkaisu, jota kehittää Katalon LLC. Ohjelmisto on rakennettu ilmaisten open-source automaatio frameworkkien Seleniumin ja Appiumin päälle. Siinä on integroitu kehitysympäristö, joka mahdollistaa API-, web- ja mobiilitestauksen. Katalon Studion ensimmäinen julkinen julkaisu oli syyskuussa 2016. (Katalon Studio, n.d.)

2 TESTAUKSEN AUTOMATISOINNIN PERUSTEET

Testiautomaatio voidaan jakaa viiteen perusosaan, jotka ovat nimeltään vaatimusanalyysi, testisuunnitelma, testausympäristön luominen, test casejen suunnittelu ja kehitys ja lopuksi testien suorittaminen ja viimeistely. (ForgeAhead, 2017)

Ensimmäinen vaihe on vaatimusanalyysi. Tässä vaiheessa testaustiimi tutkii mitä vaatimuksia testattava ohjelmisto sisältää. Tämän pohjalta saadaan tehtyä testisuunnitelma. Ohjelmisto saattaa sisältää molempia käytännöllisiä ja ei-käytännöllisiä vaatimuksia. Mahdollisia automaattisen testauksen tuomia etuja punnitaan ja lasketaan pitkän tähtäimen investoinnin kannattavuus. (ForgeAhead, 2017)

Toinen vaihe on testisuunnitelma. Automaatio on kriittisessä osassa testisuunnitelmaa laadittaessa. Tässä vaiheessa tehdään yksityiskohtainen analyysi testivaatimuksista, jotta löydetään testit, jotka sopivat automaatiotesteiksi manuaalisen testauksen sijaan. Oikea automatisointitestaustyökalu valitaan ja aletaan rekrytoimaan työkalun asiantuntijoita, jotta saadaan testit kirjoitettua. Luodaan yksityiskohtainen automatisoitu testausprosessi, joka sisältää automatisoitujen testisuunnittelu- ja kehitys standardien kehittämisen. On myös tärkeää suunnitella testaajaryhmän kouluttamista automatisoituun testausprosessiin, testisuunnitelmaan, testien kehittämiseen ja testien ajamiseen. Lisäksi luodaan automaatiosuunnitelmat yksikkö-, integraatio-, järjestelmä- ja käyttäjätestejä varten. (ForgeAhead, 2017)

Kolmannessa vaiheessa luodaan testausympäristö, joka on luotava ennen testien kehittämistä. Testausympäristö sisältää teknisen ympäristön, joka sisältää resursseja, kuten testauspaikan, laitteiston ja vaadittavat ohjelmistot, jotka ovat tarpeen automaattisten testien kehittämiseksi ja niiden suorittamiseksi. Testausautomaatiota varten onkin tärkeää, että automaatiotestaustyökalu on yhteensopiva sovelluksen kanssa ja ristiriitojen sattuessa on keksittävä ideoita ja ratkaisuja ristiriidan ratkaisemiseksi. On myös tärkeää, että tekninen ympäristö on yleisessä toimintavalmiudessa. Testausympäristöä tukevan laitteiston on oltava riittävä varmistamaan, että suorituskykyongelmia ei tule jatkossakaan vastaan. Testausympäristön suunnittelussa on myös otettava huomioon kuormitustestit. (ForgeAhead, 2017)

Neljäs vaihe on test casejen suunnittelu ja kehitys. Tässä vaiheessa määritellään suoritettavien testien määrä ja testausmenetelmä. Myös suunnittelustandardit on määriteltävä ja niitä on noudatettava. Testimenettelyn suunnittelu edellyttää, että luodaan järkeviä testimenettelyryhmiä ja määritellään tapa nimetä koko sarjaprosessi. Kun testimenettely on määritelty, jokainen menettely tunnistetaan sitten joko automaattiseksi tai manuaaliseksi testiksi. Tässä vaiheessa on ratkaisevan tärkeää tunnistaa lopulliset automaatioehdokkaat. (ForgeAhead, 2017)

Kun suunnittelu on saatu päätökseen, testin kehittäminen perustuu suunnitteluun. Automaattisten testien on oltava uudelleenkäytettäviä ja ylläpidettäviä. Tämän varmistamiseksi testin kehittämisstandardit on määriteltävä selkeästi ja niitä on noudatettava. Testisuunnittelu- ja kehitystoiminta on yleensä iteratiivista ja inkrementaalista. Nämä testimenetelmät luodaan, kun integrointitestaus on käynnissä, ja tavoitteena on ottaa ne uudelleen käyttöön järjestelmän testausvaiheessa. (ForgeAhead, 2017)

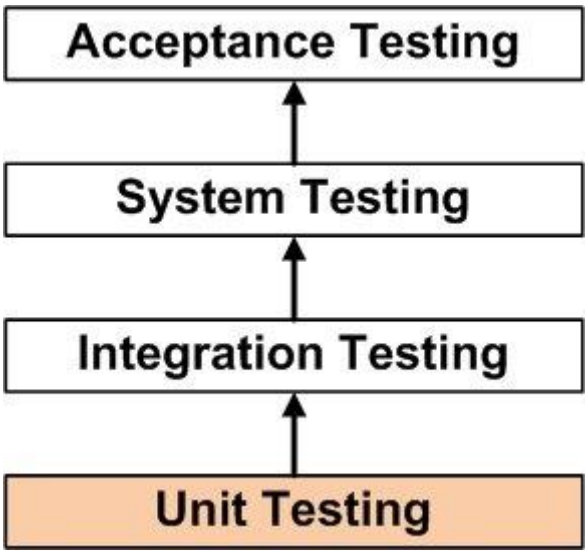
Viides ja viimeinen vaihe on testien suorittaminen ja viimeistely. Testien suoritusvaiheessa on tärkeää varmistaa, että testit pystytään suorittamaan aikataulun mukaisesti. Vikailmoitusten dokumentointi ja seuranta helpottuu suurelta osin, jos käytössä on automatisoitu vianseurantatyökalu. Automaattisen seurantatyökalun käyttäminen on hyödyllistä, koska se auttaa välttämään häiriöitä, joita ei huomattaisi manuaalisesti, säästää aikaa ja on tarkempi. Testiautomaatiotyökaluja käytetään myös testien yleisen etenemisen seurantaan ja niitä käytetään johdon tilannekatsauksen tuottamiseen. (ForgeAhead, 2017)

Automatisoidut testityökalut antavat raportteja testin kattavuudesta, edistymisestä ja testien laadusta. Automaattisen testin suorittamisen jälkeen testiohjelman revisiot voidaan lopulta tehdä. Koko automaattinen testausprosessi kerää erilaisia testituloksia. Näitä tuloksia ei käytetä vain edistymisen mittaamiseen, mutta todennetaan myös testin kattavuus ja varmistetaan laadukas ohjelmisto. Tässä vaiheessa tiedoja tallennetaan, jotta voidaan mitata automaation hyötyjä koko testin elinkaaren ajan. Voidaan luoda myös kysely, jossa kerätään yleinen palaute käytettävyydestä, suorituskyvystä, tehokkuudesta ja testien ROI-asteesta. (ForgeAhead, 2017)

3 TESTAUSMENETELMÄT

Testausmenetelmät ovat strategioita ja lähestymistapoja, joita käytetään valitun tuotteen testaamiseen. Sillä varmistetaan, että tuote on tarkoituksenmukainen. Testausmenetelmissä yleensä testataan, että tuote toimii sen määrittelyiden mukaisesti ja sillä ei ole ei-toivottuja sivuvaikutuksia, kun sitä käytetään sen suunnitteluparametrien ulkopuolella. Ja ei-toivotuissa tapauksissa tuote hajoaa tai epäonnistuu turvallisesti. Koska ohjelmistosovellukset ovat yhä monimutkaisempia ja riippuvaisia toisistaan. On tärkeämpää kuin koskaan testata niiden toimivuus niille sopivalla menetelmällä, jotta ohjelmistot täyttävät niiltä vaaditun käytettävyyden ja turvallisuuden vaatimukset. (Inflectra, 2018)

Toiminnallinen testaus on black-box testaus tyyppi, joka perustaa testicaset testattavien ohjelmistokomponentien spesifikaatioihin. Toiminnot testataan syöttämällä ne sisääntuloon ja tutkimalla ulostuloa. Sisäistä ohjelmistorakennetta tarkastellaan harvoin, toisin kuin white-box testauksessa. Toiminnallinen testaus kuvaa yleensä miten järjestelmä toimii. Toiminnallinen testaus testaa jonkin tietyn osan järjestelmän toiminnoista. (Inflectra, 2018)



Kuva 1. Toiminnallisen testauksen järjestys (Software Testing Help, (2019).

3.1 **Yksikkötestaus**

Yksikkötesteissä testataan sovelluksen pienempää osaa eli yksikköä. Yksikkö voi tietyissä tapauksissä olla myös koko moduuli, mutta se on yleisemmin vain yksittäinen toiminto tai menettely. Objektikohtaisessa ohjelmoinnissa yksikkö on usein koko käyttöliittymä, kuten luokka. Yksikkötestit ovat lyhyitä koodin pätkiä, jotka on kehittänyt ohjelmoijat tai white-box testaajat kehittämisprosessin aikana. Ne muodostavat perustan komponenttitestaukselle. (Software Testing Help, (2019).

Parhaassa tapauksessa kukin test case on riippumaton muista caseista. Ohjelmistokehittäjät kirjoittavat ja ohjaavat tyypillisesti yksikkötestejä sen varmistamiseksi, että koodi täyttää sille asetetut vaatimukset ja käyttäytyy suunnitellusti. Koska joillakin luokilla voi olla viittauksia muihin luokkiin, luokan testaaminen voi usein mennä toisen luokan testaukseen. Yleinen esimerkki tästä on tietokannasta riippuvaiset luokat. Testatakseen luokan, testaaja kirjoittaa koodin, joka toimii vuorovaikutuksessa tietokannan kanssa. Näin ei saisi tehdä, koska yksikkötesti ei yleensä saa mennä oman luokkansa rajojen ulkopuolelle, koska tämä voi aiheuttaa suorituskykyongelmia yksikkötestipaketille. Rajojen ylittäminen muuttaa yksikkötestit integrointitesteiksi. Kun tällaiset testcaset epäonnistuvat on epäselvää, mikä komponentti aiheuttaa vian. (Unit testing, n.d.)

Yksikkötestaus on yleensä automatisoitu, mutta se voidaan silti suorittaa manuaalisesti. Yksikkötestauksessa molemmat lähestymistavat ovat hyviä vaihtoehtoja. Yksikkötestauksen tavoitteena on eristää yksikkö ja validoida sen virheettömyys. Manuaalinen lähestymistapa yksikkötestaukseen voi käyttää step-by-step dokumenttia testien luomiseksi. Automaatio on kuitenkin erittäin tehokas vaihtoehto myös yksikkötesteissä ja antaa lukuisia etuja manuaaliseen testaukseen nähden. Jos manuaalista testiä ei suunnitella huolellisesti se saatetaan suorittaa vahingossa integrointitestinä, johon liittyy monia ohjelmistokomponentteja. Jos testin tekee huolimattomuuttaan integraatiotestinä siinä estetään useimpien,

jos ei kaikkien, yksikkötestausta varten asetettujen tavoitteiden saavuttaminen. (Unit testing, n.d.)

Yksikkötestauksen tavoitteena on eristää kukin ohjelman osa ja testata yksittäisten osien toimivuus. Yksikkötesti asettaa koodille tarkat vaatimukset, jotka koodin on täytettävä. Tämän seurauksena se tarjoaa useita etuja muihin testausmenetelmiin nähden. Yksikkötestaus löytää ongelmat kehityskaaren alkuvaiheessa. Tämä sisältää sekä ohjelmoijan toteutuksesta johtuvat virheet, sekä koodin puuttuvat osat. Testisarjojen tarkempi kirjoittaminen pakottaa kirjoittajan miettimään tuloja, lähtöjä ja virheitä, ja siten määrittelemään entistä tarkemmin yksikön haluttu käyttäytyminen. Virheen löytämisen kustannukset ennen koodauksen alkua tai kun koodia kirjoitetaan ensimmäisen kerran, ovat huomattavasti pienemmät kuin virheen havaitsemisen, tunnistamisen ja korjaamisen kustannukset myöhemmin. Julkaistu koodi voi myös aiheuttaa kalliita ongelmia ohjelmiston loppukäyttäjille. Koodi voi olla mahdotonta tai vaikeaa testata, jos se on huonosti kirjoitettu, joten yksikkötestaus voi pakottaa kehittäjät kirjoittamaan funktioita ja objekteja erilaisilla tavoilla. (Unit testing, n.d.)

3.1.1 Yksikkötestauksen esimerkki

```
public class MyUnit {

    public String concatenate(String one, String two){
        return one + two;
    }
}
```

Kuva 2. Testattava luokka (Jenkov, 2014)

```
import org.junit.Test;
import static org.junit.Assert.*;

public class MyUnitTest {

    @Test
    public void testConcatenate() {
        MyUnit myUnit = new MyUnit();

        String result = myUnit.concatenate("one", "two");

        assertEquals("onetwo", result);

    }
}
```

Kuva 3. Yksikkötesti (Jenkov, 2014)

Yllä oleva yksikkötesti, testaa concatenate()-metodia. TestConcatenate()-metodin sisällä luodaan MyUnit, jonka concatenate()-metodia kutsutaan kahdella stringillä (”one” ja ”two”). AssertEquals()-metodi tekee kaiken varsinaisen testauksen. Tämä metodi vertaa concatenate()-metodin ulostuloa ja vertaa sitä odotuksenmukaiseen ulostuloon. Jos ulostulo ja odotettu ulostulo täsmäävät,

testi menee läpi normaalisti. Jos ulostulot poikkeavat toisistaan, testi on löytänyt virheen ja se pysähtyy virheen löytyessä. (Jenkov, 2014)

3.2 **Integraatiotestaus**

Integraatitestauksessa testataan integroitua tai yhdistettyä yksikköä ja testataan näitä yksikköjä kokonaisena moduulina tai suurempana kokonaisuutena. Tämän testauksen päätehtävä tai tavoite on testata moduulien välisiä rajapintoja. Integrointitestaus suoritetaan normaalisti yksikkötestauksen jälkeen. Kun kaikki yksittäiset yksiköt on luotu ja testattu, yhdistämme nämä yksikön testatut moduulit ja aloitamme integraatiotestauksen. Kun moduulit on testattu yksikkökohtaisesti sen jälkeen ne integroidaan yksitellen, kunnes kaikki moduulit on integroitu. Tällä tavalla saadaan tarkistettua moduulien yhteensopivuus ja toimivatko ne oikein. Tässä on ymmärrettävä, että integraatiotestaus ei tapahdu kehityksen lopussa, vaan se suoritetaan samanaikaisesti ohjelmistokehityksen kanssa. Joten lähes kaikissa tapauksissa kaikki moduulit eivät ole todellisuudessa käytettävissä testattaviksi, vaan niitä voidaan testata sitä mukaa kun yksikköjä saadaan koottua isommiksi kokonaisuuksiksi. Integraatiotestauksen erilaisia tyyppejä ovat Big Bang, sandwich, Top Down, ja Bottom Up. (Software Testing Help, 2018)

Big bang -lähestymistavassa suurin osa kehitetyistä moduuleista on kytketty yhteen muodostaen koko ohjelmistojärjestelmän tai suurimman osan järjestelmästä, jota voidaan sen jälkeen käyttää integraatiotestaukseen. Tämä lähestymistapa on erittäin tehokas säästämään aikaa integraatiotestauksessa. Jos testcaseja ja niiden tuloksia ei kuitenkaan tallenneta kunnolla, koko integraatioprosessi on monimutkaisempi ja voi estää testausryhmää saavuttamasta integrointitestauksen tavoitetta. Big Bang -menetelmä soveltuu hyvin pienempiin järjestelmiin, joissa mahdollisten vikojen määrä on huomattavasti pienempi. Koska koko järjestelmä testataan yhdellä kertaa on huomattavasti vaikeampi vikatilanteen sattuessa määrittää mistä moduulista vikatilanne johtuu. (Software Testing Help, 2018)

Bottom Up on lähestymistapa, jossa alimman tason komponentit testataan ensin ja joita sen jälkeen hyödynnetään korkeamman tason komponenttien testaamiseen. Prosessi toistetaan, kunnes hierarkian ylimmät komponentitkin on testattu. Kaikki pohjimmaisen tai matalimman tason moduulit ja toiminnot integroidaan ja testatataan. Alemman tason integroitujen moduulien integraatiotestauksen jälkeen muodostetaan seuraava moduulien taso, jota voidaan käyttää integraatiotestaukseen. Tämä lähestymistapa on hyödyllinen vain silloin, kun kaikki tai useimmat saman kehitystason moduulit ovat valmiita. Tämä menetelmä auttaa myös määrittämään kehittyneiden ohjelmistojen tasot ja helpottaa testauksen etenemisen ilmoittamista prosentteina. (Software Testing Help, 2018)

Top Down-lähestymistavassa testaus alkaa ylimmältä moduulilta ja etenee vähitellen kohti alempia moduuleja. Vain ylin moduuli testataan yksittäin. Tämän jälkeen alemmat moduulit integroidaan yksi kerrallaan. Prosessi toistetaan, kunnes

kaikki moduulit on integroitu ja testattu. Testaus alkaa ylimmästä moduulista ja alemmat moduulit integroidaan yksitellen. Tällöin alemmat moduulit eivät ole tosiasiassa vielä käytettävissä testaukseen. Jotta voimme testata ylimmän moduulin täytyy kehittää dummy koodinpätkä, joka hyväksyy tulot ja pyynnöt ylimmältä moduulilta ja palauttaa tulokset ja vastaukset. Tällä tavalla, vaikka alempia moduuleja ei ole olemassa, pystymme testaamaan ylimmän moduulin. Käytännön skenaarioissa tälläisten dummy koodinpätkien käyttäytyminen ei ole niin yksinkertaista kuin voisi kuvitella. Monimutkaisten moduulien ja arkkitehtuurien vuoksi testeihin liittyy suurimman osan ajasta monimutkaisia logiikoita, kuten yhteyden muodostaminen tietokantaan. Tämän seurauksena dummy koodin luominen muuttuu yhtä monimutkaiseksi ja aikaa vieväksi kuin todellisen moduulin ohjelmointi. (Software Testing Help, 2018)

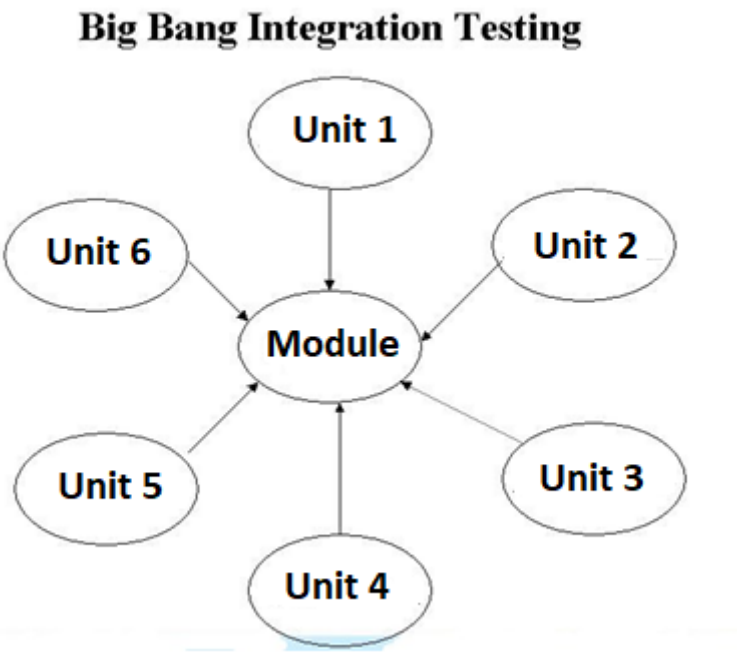
Sandwich on lähestymistapa, jossa yhdistyvät sekä Top Down että Bottom Up -periaatteiden piirteet. Kun testataan suuria ohjelmia, kuten käyttöjärjestelmiä, on oltava lisää tekniikoita, jotka ovat tehokkaita ja lisäävät testien luotettavuutta. Sandwich-testillä on tässä erittäin tärkeä rooli, jossa molemmat Top Down ja Bottom Up -testaukset aloitetaan samanaikaisesti. Integrointi alkaa keskimmäisistä moduuleista ja siirtyy samanaikaisesti ylös- ja alaspäin. Koska molemmat lähestymistavat alkavat samanaikaisesti, tämä tekniikka on monimutkainen ja vaatii enemmän henkilöstöä, sekä tiettyjä taitoja ja ei ole tämän takia kovin kustannustehokas. (Software Testing Help, 2018)

3.2.1 Integraatiotestauksen esimerkki

Integraatiotestauksen esimerkkinä toimii Big Bang -menetelmällä toteutettu web-kauppa. Kuvitellaan tilanne, jossa yrityksessä kuusi kehittäjää ohjelmoivat web-kauppa projektin eri osa-alueita. Esimerkiksi.

1. Käyttäjän rekisteröinti ja kirjautuminen
2. Tuoteluettelo / kauppa
3. Ostoskori
4. Laskutus
5. Laskutus portaalien toiminta.
6. Toimitus ja paketinseuranta

Kun kehittäjät saavat osuutensa ohjelmoitua he yksikkötestaavat sen virheiden varalta ja korjaavat mahdolliset virheet. Tämän jälkeen kaikki kuusi yksikköä liitetään yhteen ja testataan, että ne toimivat yhdessä (integraatiotestaus). Kaikki kuusi yksikköä testataan samanaikaisesti, tätä kutsutaan Big Bang -menetelmäksi. (TRY QA, n.d.)



Kuva 4. Big Bang esimerkki

3.3 Järjestelmätestaus

Järjestelmätestaus on black-box testauksen alle kuuluva menetelmä. Se on vaihe ohjelmistojen testauksessa, jossa testataan kokonaista sovellusta / järjestelmää. Järjestelmätestauksen painopiste on arvioida koko järjestelmän vaatimustenmukaisuutta sille asetettujen vaatimusten mukaisesti. Järjestelmätestaus auttaa hyväksymään ja tarkistamaan sovelluksen liiketoiminnalliset, toiminnalliset, tekniset vaatimukset katsottaessa arkkitehtuuria kokonaisuudessaan. (Techopedia, n.d.)

Järjestelmätestauksen laajuus ei rajoitu pelkästään järjestelmän suunnitteluun vaan myös järjestelmän toimintaan ja siltä odotettuun käyttäytymiseen. Ohjelmistojen testausjakson mukaisesti järjestelmän testaus suoritetaan ennen hyväksymistestausta ja integraatiotestauksen jälkeen. Käyttäjille tai testaajille annetaan kaikille tietyt tehtävät, joita heidän tulee testata järjestelmän testausvaiheessa. Järjestelmätestaus on tärkeä etappi testauksessa sillä siinä testataan ensimmäistä kertaa koko järjestelmän toimivuus kokonaisuudessan. Järjestelmätestauksen aikana tehdään myös arviointi siitä kuinka järjestelmä vastaa sille asetettuja toiminnallisia vaatimuksia. Toimintavaatimusten ja sovellusarkkitehtuurin validointi, todentaminen ja testaus tehdään järjestelmätestausvaiheen aikana. Järjestelmätestaus antaa käyttäjille tehokkaan ympäristön, joka muistuttaa enemmän tai vähemmän elävää tuotantoympäristöä. Tästä johtuen, kun tämänkaltainen testaus tehdään, saadaan luotettavampia ja tehokkaampia tuloksia. (Techopedia, n.d.)

3.3.1 Järjestelmätestauksen esimerkki

Järjestelmä testataan kokonaisuudessaan, kun kaikki moduulit on saatu valmiiksi. Järjestelmästä voidaan testata erittäin laajasti eri osa-alueita, joihin lukeutuvat mm.

1. Regressiotestaus - pitää huolen, että mikään muutos projektissa ei vaikuta negatiivisesti muihin projektin osiin.
2. Käytettävyystestit - testataan kuinka hyvä käytettävyys ohjelmalla on loppukäyttäjän näkökulmasta.
3. Rasitustestit - testataan miten ohjelma kestää rasitusta ja miten se käyttäytyy rasituksen alaisena
4. Palautumistestaus - testataan miten ohjelma palautuu virheen tai kaatumisen jälkeen. Demonstroidaan, että ohjelma on luotettava.

Järjestelmäntestaus tyyppejä on yli 50 erilaista. Testejä, joita jokin tietty projekti käyttää valitaan ajan, testin relevanttiuden, saatavilla olevien resurssien ja testausbudjetin mukaan. (Guru99, n.d.)

3.4 **Hyväksymistestaus**

Hyväksymistestaus on menetelmä, jolla selvitetään täyttääkö ohjelmisto eritelmän tai sopimuksen vaatimukset. Se voi sisältää kemiallisia testejä, fyysisiä testejä tai suorituskykytestejä. Ohjelmistotestauksessa ISTQB (International Software Testing Qualifications Board) määrittelee hyväksymistestauksen seuraavasti: käyttäjän tarpeiden, vaatimusten ja liiketoimintaprosessien muodollinen testaus sen määrittämiseksi, täyttääkö järjestelmä sille asetetut kriteerit. Se antaa myös käyttäjälle, asiakkaille tai muulle valtuutetulle yksikölle mahdollisuuden päättää hyväksyykö se järjestelmän. Hyväksymistestausta kutsutaan myös käyttäjän hyväksymistestaukseksi (UAT). (Acceptance testing, n.d.)

Hyväksymistestit on mahdollisesti suoritettava useita kertoja, koska kaikki testitcaset eivät välttämättä toteudu yhden testin iteraation aikana. Hyväksymistestipaketti suoritetaan käyttämällä ennalta määritettyjä hyväksyntätestimenettelyjä, joilla ohjataan testaajia siihen mitä dataa käytetään, mitä vaiheita seurataan ja minkälaista tulosta testeiltä odotetaan. Varsinaiset tulokset säilytetään ja niitä verrataan odotettuihin tuloksiin. Jos todelliset tulokset vastaavat kunkin testitapauksen odotettuja tuloksia, testicase on mennyt hyväksytysti läpi. Jos ei-hyväksyttyjen testien määrä ei riko projektin ennalta määrättyä kynnysarvoa, testit voidaan myös tämän jälkeen katsoa menneen hyväksytysti läpi. Jos ei-hyväksytyt testit ylittävät kynnysarvon järjestelmä voidaan joko hylätä tai hyväksyä olosuhteissa, joista sponsorit ja valmistaja ovat aiemmin sopineet. (Acceptance testing, n.d.)

Onnistuneilta testeiltä odotettu tulos sisältää seuraavat vaiheet. Testicaset suoritetaan käyttäen ennalta määritettyjä tietoja. Testien todelliset tulokset tallen-

netaan. Todellisia tuloksia ja odotettuja tuloksia verrataan ja testitulokset määritetään, joko hyväksytyksi tai ei-hyväksytyksi. Tavoitteena on varmistaa, että kehitetty tuote täyttää sekä toiminnalliset että ei-toiminnalliset vaatimukset. Hyväksymistestauksen tarkoituksena on, että kun testaus on valmis ja hyväksymiskriteerit täyttyvät. Sponsorit ja valmistaja voivat tämän jälkeen sopia tuotteen kehityksestä ja jatkotoimenpiteistä. (Acceptance testing, n.d.)

3.4.1 Käyttäjän hyväksymistestaus (UAT)

Käyttäjän hyväksymistestaus koostuu prosessista, jolla varmistetaan, että ohjelmisto toimii käyttäjällä. Se ei ole järjestelmätestausta vaan se varmistaa, että ohjelmisto toimii käyttäjien joka päiväisessä käytössä. Ohjelmistotoimittajat viittaavat tähän usein betatestauksena. Tämän testauksen tulisi suorittaa henkilö, joka mieluiten tietää testattavasti aiheesta jo entuudestaan, sekä on ohjelmiston omistaja tai asiakas. Testaushenkilö antaa yhteenvedon havainnoista, jotka vahvistavat, että voidaan jatkaa testauksen jälkeen. Ohjelmistokehityksessä UAT on yhtenä hankkeen viimeisistä vaiheista ja tapahtuu usein ennen kuin asiakas tai kuluttajat hyväksyvät uuden ohjelmiston. Järjestelmän käyttäjät suorittavat testit sen mukaan, mitä tapahtuisi todellisissa käytössä. On tärkeää, että testaajalle annetut materiaalit ovat samankaltaisia kuin loppukäyttäjällä. Testaajille on annettava todellisia tilanteita, kuten esimerkiksi kolme yleisintä tehtävää, joita heidän edustamansa loppukäyttäjät tulevat toteuttamaan. UAT toimii lopullisena varmistuksena liiketoiminnallisesta toiminnasta ja järjestelmän moitteettomasta toiminnasta, jäljittelemällä todellisia käyttöolosuhteita. Jos ohjelmisto toimii vaatimusten mukaisesti ja ilman normaalia käyttöä koskevia ongelmia, voidaan kohtuullisesti odottaa samaa vakauden tasoa tuotannossa. Käyttäjätestit, joita yleensä suorittavat asiakkaat tai loppukäyttäjät, eivät yleensä keskity tunnistamaan yksinkertaisia kosmeettisia ongelmia, kuten oikeinkirjoitusvirheitä, eikä virheitä, kuten ohjelmiston kaatumisia. Testaajat ja kehittäjät tunnistavat ja korjaavat nämä ongelmat aikaisemman yksikkötestauksen, integraatiotestauksen ja järjestelmätestausvaiheiden aikana. (Acceptance testing, n.d.)

UAT tulee suorittaa testiskenaarioita vasten. Testiskenaariot poikkeavat yleensä toiminnallisista testicaseista, koska ne edustavat pelaajaa tai loppukäyttäjää. Testausskenaarion laajuus takaa, että painopiste on käyttökokemuksella eikä teknillisillä tai järjestelmällisillä yksityiskohdilla. Se pysyy poissa yksinkertaisista klikkaustestivaiheista, jotta käyttäjien käyttäytymiseen saadaan vaihtelua. Teollisuudessa yhteinen UAT on tehtaan hyväksymistätesti (FAT). Tämä testi suoritetaan ennen laitteen asennusta. Suurimman osan ajasta testaajat eivät vain tarkista, että laitteisto täyttää vaatimukset, vaan myös, että laite on täysin toimiva ja turvallinen. FAT sisältää yleensä valmiuden tarkistuksen, sopimuksen vaatimusten mukaisuuden tarkastuksen, todisteen toiminnallisuudesta (joko simulaation tai tavanomaisen toimintatestin avulla) ja lopputarkastuksen. Näiden testien tulokset antavat asiakkaille luottamuksen siitä, miten järjestelmä toimii tuotannossa. Järjestelmän hyväksymiseen voi myös liittyä oikeudellisia tai sopimusvelvoitteita. (Acceptance testing, n.d.)

3.4.2 Hyväksymistestauksen esimerkki

Hyväksymistestauksen paras esimerkki on betatestaus, jossa osallisena ovat yleensä loppukäyttäjät. Betatestauksessa ohjelmasta julkaistaan rajoitettu versio käyttäjien käyttöön, jonka avulla saadaan testattua tuotetta todellisessa käyttöympäristössä. Betatestauksessa on otettava huomioon.

- Testauksen päämäärä - kaikki asiat, jotka täytyy saada testattua
- Aikataulutus – Jokaisen syklin ja vaiheen kesto
- Betatestisuunnitelma
- Testauksen lähetymistapa, jota käyttäjät noudattavat
- Työkalut, joilla kerätään bugeja ja palautetta ja mitataan tuottavuutta
- Palkinto tai kannuste testiin osallistujille
- Milloin ja miten tämä testaus päätetään

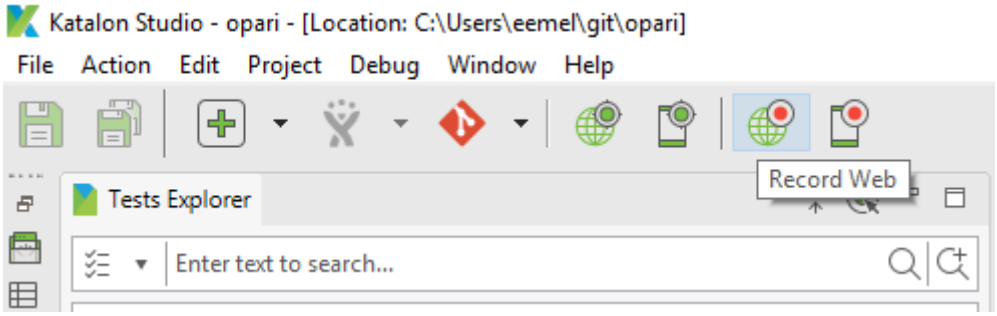
4 SOVELTAVA PROJEKTI

4.1 Testien luominen

Katalon Studiolla voidaan luoda testejä kolmella eri tavalla. Record Web toiminnolla, Manual modella tai Script Modella. Testien kirjoittamisen aikana voidaan käyttää jokaista eri tapaa samassa testissä. Jokaisessa testien kirjoitustavassa täytyy ensin luoda uusi projekti valitse yläpalkista file>new>project. Tämän jälkeen luodaan uusi Test Suite samasta polusta kuin uusi projekti tai vasemmanpuoleisesta sivupalkista. Test Suite sitoo kaikki valitut test caset yhteen paikkaan, jotta voidaan helposti suorittaa ne kaikki kerralla. Test Caset lisätään Test Suiteen, joka ajaa ne kaikki läpi.

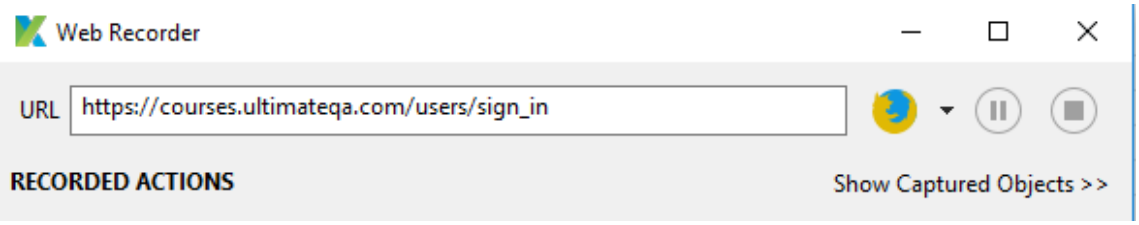
4.1.1 Record Web

Testin kirjoittamisen aloittamiseksi valitaan ylävalikosta Record Web-painike.

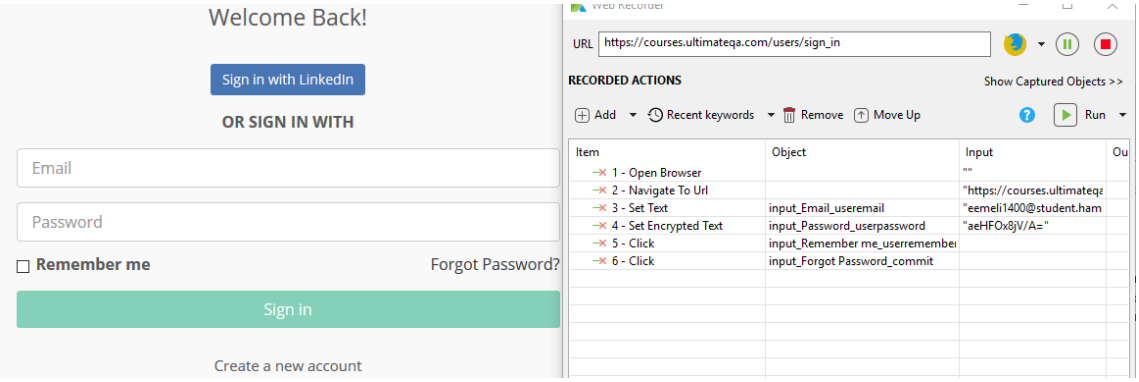


Kuva 5. Record Web

Näissä testeissä käytettiin https://www.ultimateqa.com/ sivustoa, joka antaa laajat mahdollisuudet testata web-sivun erilaisia ominaisuuksia. Kirjoitetaan testissä käytettävä url-osoite Web Recorderin palkkiin ja valitaan oikealta selain, jota halutaan käyttää testeissä. Klikataan valittua selaimen kuvaa, joka avaa url-osoitteen ja lisää testiin Open Browser lausekkeen.

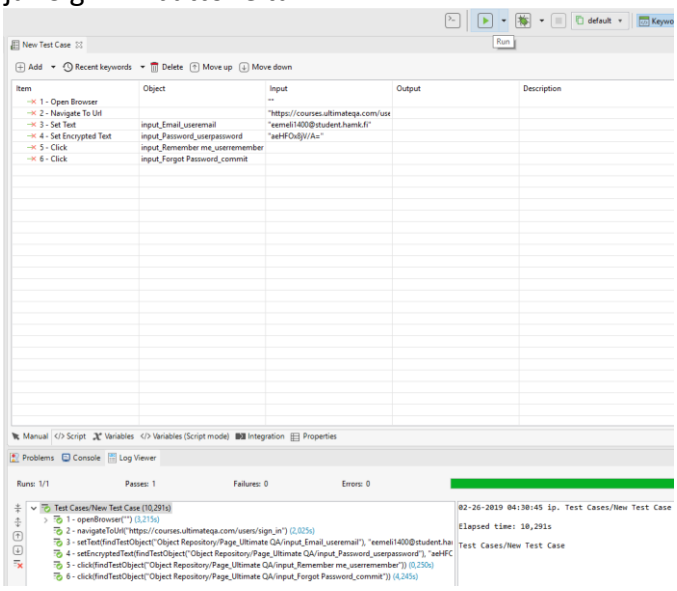


Kuva 6. Web Recorder URL



Kuva 7. Nauhoitettu kirjautumisprosessi

Yllä olevassa kuvassa on nauhoitettu normaali kirjautumisprosessi web-sivustolle. Ensimmäisessä ja toisessa stepissä testi avaa selaimen ja navigoi oikeaan url-osoitteeseen. Kolmannessa ja neljännessä stepissä kirjoitetaan kenttiin käyttäjänimi ja salasana. Viidennessä ja kuudennessa stepissä klikataan ”Remember me” ja ”Sign in” buttoneita.

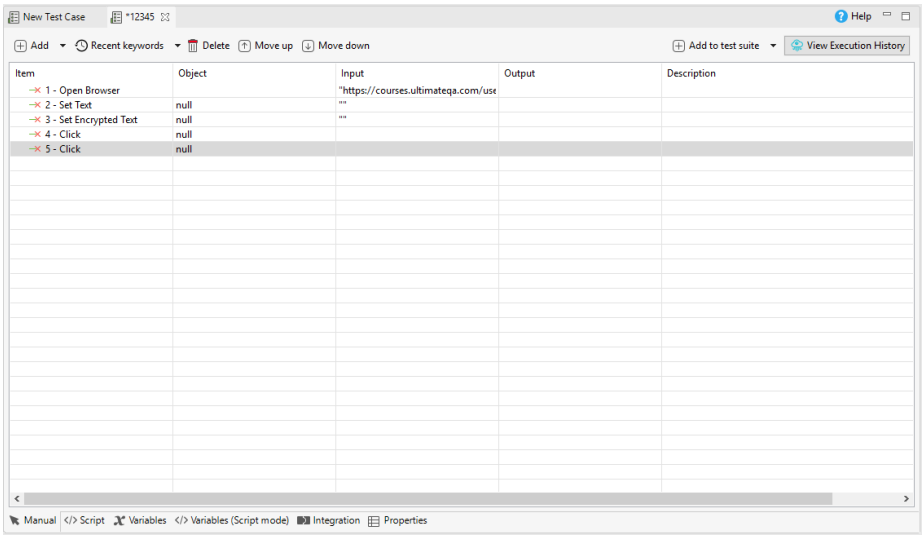


Kuva 8. Onnistunut testi

Luotu testi ajetaan ylävalikon Run-painikkeesta. Alhaalla näkyy vaihe vaiheelta testin eteneminen. Jos testi syystä tai toisesta on viallinen, se pysähtyy sen stepin kohdalla, josta ei päästy etenemään ja antaa virheilmoituksen. Hyväksytysti läpi mennyt testi on esitetty yllä olevassa kuvassa. Kyseinen test case olisi nyt valmis lisättäväksi Test Suiteen, jossa testataan koko sivuston eri ominaisuuksia. Testiin voidaan mahdollisesti myös lisätä muut sign in-sivulla sijaitsevat painikkeet, jotta saadaan koko sign-in sivu yhteen test caseen.

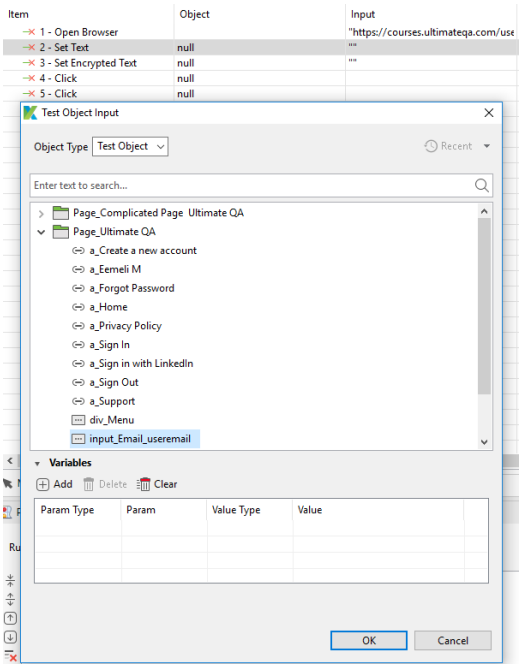
4.1.2 Manual Mode

Manual Modella testejä kirjoittaessa täytyy olla jonkin verran tietoa mitä Katalon Studion keywordit tekevät. Manual modea käytettäessä täytyy myös tallentaa elementit Object Spy toiminnolla object repositoryyn. Object Spy toiminnon käyttäminen on käyty läpi luvussa 4.2.



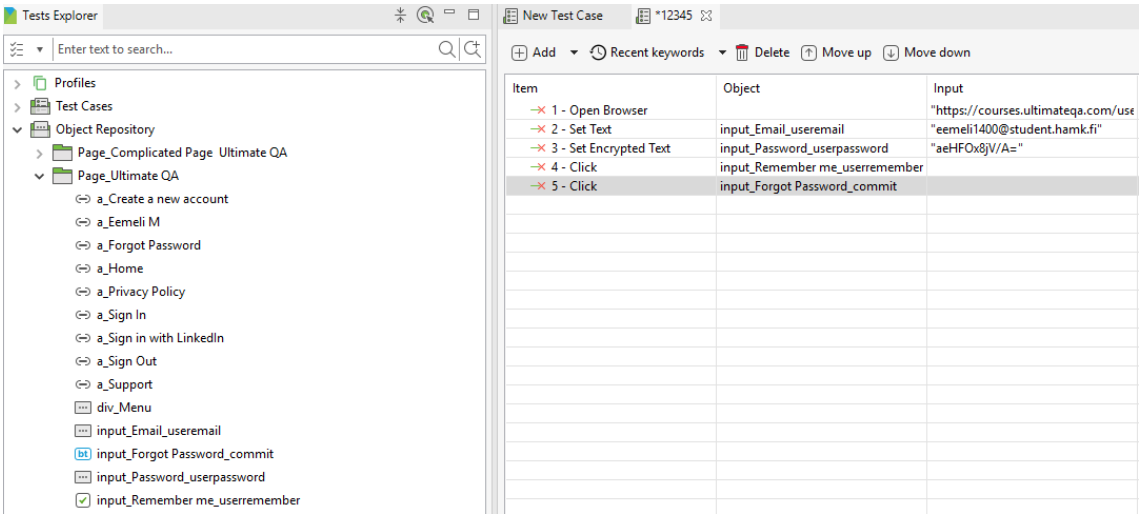
Kuva 9. Manual Mode näkymä

Manual Modessa voidaan lisätä steppejä testiin vasemman yläreunan Add-painikkeella. Saman testin kirjoitus, kuin aiemassa Record Web osuudessa vaatii ainoastaan viisi steppiä, koska voimme lisätä url:n Open Browser lausekkeen kohdalle. Lisätään testiin ensin tarvittavat stepit, jotka ovat Open Browser, Set Text, Set Encrypted Text ja kaksi Click toimintoa.



Kuva 10. Test Object Input

Klikataan ensin Set Text kentästä, joka avaa Test Object Input -valikon. Valikko sisältää kaikki Object Repositoryyn tallennetut objektit. Kyseiseen kenttään haluamme tässä tapauksessa input_Email_useremail vaihtoehdon, joka on kirjautumis-sivun email kenttä. Tämän jälkeen input sarakkeeseen lisätään, käyttäjätunnus tai email, jonka halutaan testin syöttävän tuohon kenttään. Set Encrypted Text kohdassa valitaan samalla tavalla input_Password_userpassword vaihtoehto, joka on kirjautumis-sivun salasana kenttä. Tämän jälkeen taas input kenttään haluamasi salasana. Encrypted Text toiminto enkryptaa salasanat automaattisesti. Jos ei haluta, että salasanoja enkryptataan valitaan salasana kenttään normaali Set Text.



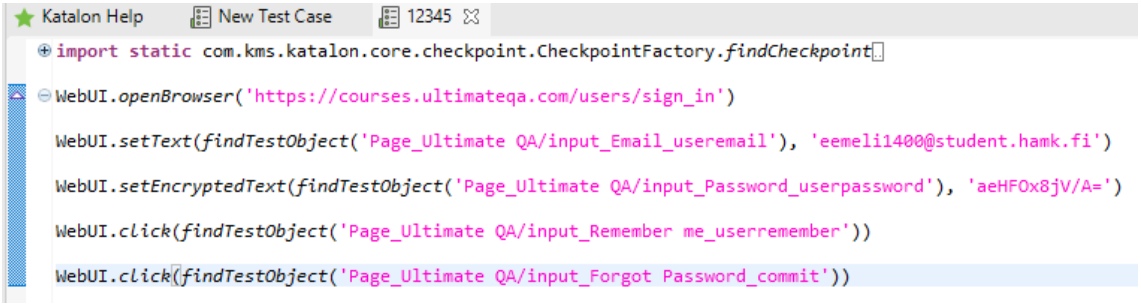
Kuva 11. Valmis testi

Viimeisiin Click toimintoihin valitaan Object Repositorystä input_Remember me_userremember ja input_Forgot Password_commit. Tämän Web-sivun login

painikkeen nimi on jostain syystä Forgot Password, joka on hiukan kyseenalainen nimi login painikkeelle. Jos web-sivun painikkeet ja elementit on nimetty huonosti tai hämmentävästi, voi testauksessa ilmetä sekaannusta ja ongelmia myöhemmin. Tästä syystä kannattaa miettiä tarkkaan miten nimeää kunkin elementin web-sivua rakentaessaan.

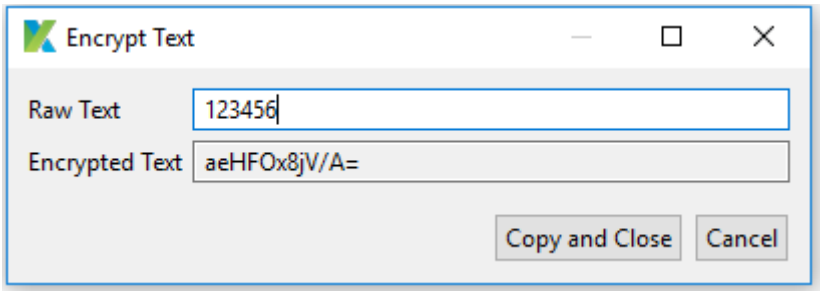
4.1.3 Script Mode

Script Mode on puhtaasti tekstipohjainen skriptaus vaihtoehto testien kirjoittamiselle. Kuten Manual Modessa myös Script Modessa täytyy luoda object repository Object Spy toiminnolla ennen testien kirjoittamista. Käytettäessä pelkkää script modea testien luomiseen täytyy olla myös laaja tietämys eri keywordeista, jota katalon studio käyttää.



Kuva 12. Script Mode

Script Modeen päästään valitsemalla alapalkista script-välilehti. Jokainen koodirivi alkaa WebUI keywordilla, joka on Katalonin sisäänrakennettu keyword. Ensimmäisenä avataan web-osoite, jota halutaan testata ja lisätään se openBrowser toiminnon yhteyteen. Seuraavissa kohdissa haetaan Object Repositoryyn tallennetuista objekteista Email- ja Password-kentät, joihin syötetään kirjautumistiedot. Skriptaustilassa salasanat täytyy enkryptata manuaalisesti. Valitaan yläpalkista Help>Encrypt Text, kirjoita kenttään haluamasi salasana jolloin Katalon enkryptaa sen automaattisesti. Kopioi enkryptatty teksti, jonka jälkeen se toimii normaalin salasanasi mukaisesti testeissä.



Kuva 13. Encrypt Text

Viimeisen kahden rivin click toimintoihin haetaan Object Repositorystä ”Remember me” ja ”Login” painikkeet.

4.2 Testien ajaminen ja tulokset

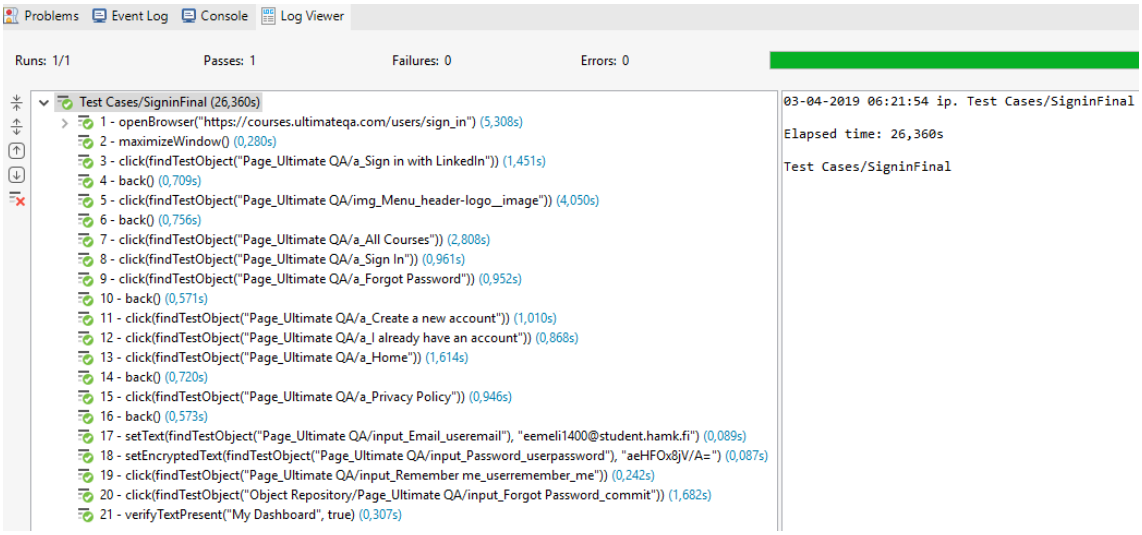
SigninFinal

```
import static com.kms.katalon.core.checkpoint.CheckpointFactory.findCheckpoint
WebUI.openBrowser('https://courses.ultimateqa.com/users/sign_in')
WebUI.maximizeWindow()
WebUI.click(findTestObject('Page_Ultimate QA/a_Sign in with LinkedIn'))
WebUI.back()
WebUI.click(findTestObject('Page_Ultimate QA/img_Menu_header-logo__image'))
WebUI.back()
WebUI.click(findTestObject('Page_Ultimate QA/a_All Courses'))
WebUI.click(findTestObject('Page_Ultimate QA/a_Sign In'))
WebUI.click(findTestObject('Page_Ultimate QA/a_Forgot Password'))
WebUI.back()
WebUI.click(findTestObject('Page_Ultimate QA/a_Create a new account'))
WebUI.click(findTestObject('Page_Ultimate QA/a_I already have an account'))
WebUI.click(findTestObject('Page_Ultimate QA/a_Home'))
WebUI.back()
//WebUI.click(findTestObject('Page_Ultimate QA/a_Support'))
//WebUI.back()
WebUI.click(findTestObject('Page_Ultimate QA/a_Privacy Policy'))
WebUI.back()
WebUI.setText(findTestObject('Page_Ultimate QA/input_Email_useremail'), 'eemeli1400@student.hamk.fi')
WebUI.setEncryptedText(findTestObject('Page_Ultimate QA/input_Password_userpassword'), 'aeHFOx8jV/A=')
WebUI.click(findTestObject('Page_Ultimate QA/input_Remember me_userremember_me'))
WebUI.click(findTestObject('Object Repository/Page_Ultimate QA/input_Forgot Password_commit'))
WebUI.verifyTextPresent("My Dashboard",true)
```

Kuva 14. Lopullinen testi

Yllä olevassa testissä testataan koko kirjautumissivun toiminnot. Testi on kirjoitettu yhdistelemällä aikaisemmin esitettyjä testien kirjoitusmenetelmiä, mutta pääasiassa Record ja Script Modeja käyttäen. Testistä on kommentoitu pois rivi, joka painaa sivulla olevaa Support-painikketta, koska se avaa kaksi eri pop-up -ikkunaa. Pop-up -ikkunat ovat erittäin hankalia tapauksia automaatiotestauksessa. Jos jokin painike avaa niitä kaksi kappaletta, pelkästään sen koodirivin saaminen toimivaksi on erittäin työlästä. Riippuen millä tekniikalla pop-up on toteutettu niiden sulkeminen voi vaihdella erittäin vaikeasta suhteellisen yksinkertaiseen. Viimeisellä rivillä varmistetaan kirjautumisen onnistuminen. Testi etsii sivulta tekstin My Dashboard, joka tulee ainoastaan näkyviin, kun kirjautuminen on onnistunut.

Testit ajetaan valitsemalla ylävalikosta Run-valikosta haluamasi selain, jolla haluat testin suorittaa. Halutessasi voit ajaa testejä myös headless-tilassa Katalon Studiolla. Headless-tila tarkoittaa, että Katalon ei näytä fyysistä selainta ajaessaan testejä vaan suorittaa testit ilman selaimen graafista käyttöliittymää.

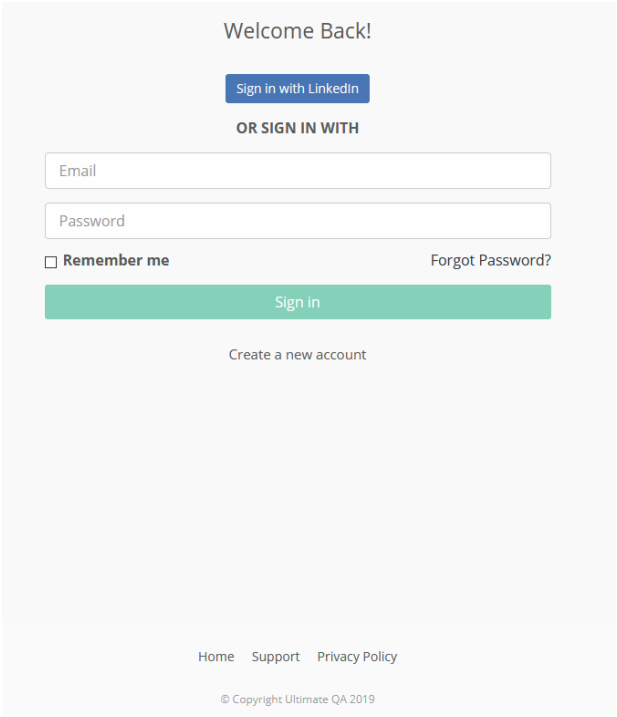


Kuva 15. Onnistunut testi

Katalon ajaa testin vaihe vaiheelta, jonka ansiosta virheen sattuessa tiedot tarkalleen missä kohtaa testi antoi virheen. Tämän ansiosta tiedetään tarkalleen missä kohtaa sivustolla on virhe. Logitiedoista näkyy tarkalleen, kuinka kauan jokainen vaihe testissä kesti ja vaikka testi menisi hyväksytysti läpi jos jokin vaihe kesti erityisen kauan voi sivustollasi silti olla koodivirheitä. Katalon odottaa jokaisen vaiheen välissä, että sivu on latautunut ennen kuin se siirtyy seuraavaan vaiheeseen.

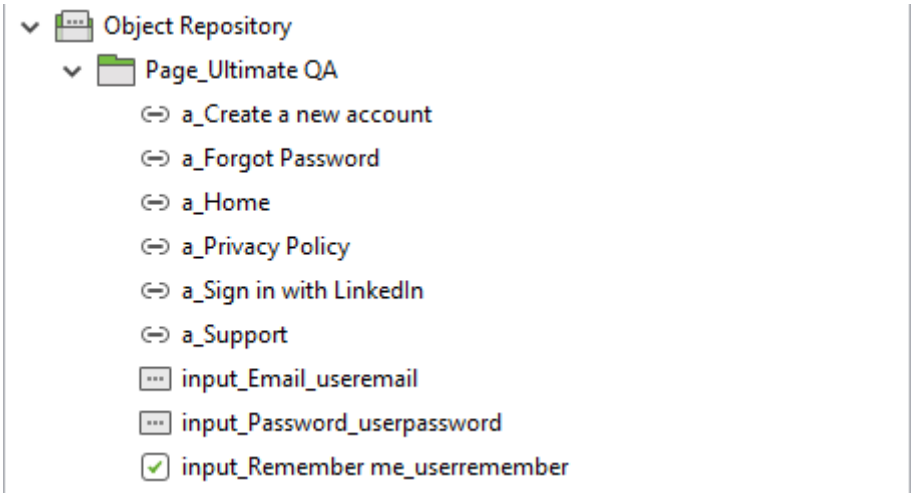
4.3 **Object Repository**

Katalon tallentaa kaikki objektit Object Repositoryyn, jotta se pystyy käyttämään niitä testeissä. Voidaan myös manuaalisesti tallentaa elementtejä repositoryyn käyttämällä object spy toimintoa. Object Spy -toiminto on erittäin tärkeä työkalu, jos halutaan tehdä kaikki testit täysin manuaalisesti käyttämättä, Web Record toimintoa. Normaalien testien kirjoittamisessa käytettävä Web Record tallentaa myös kaikki käytetyt objektit.



Kuva 16. Esimerkki sivusto Object Repositoryn tallentamista elementeistä

Katalonilla voidaan tallentaa kaikki testeihin tarvittavat elementit sivulta, jota aiotaan testata. Katalon tallentaa ne Object Repositoryyn, joka löytyy vasemman puoleisesta sivupalkista. Ensimmäisenä on listattu normaalit button elementit, joita klikattaessa aukeaa mahdollisesti, joko pop-up -ikkuna tai täysin eri url-osoite. Sen jälkeen on kaksi tekstikenttää joihin syötetään tässä tapauksessa normaalisti käyttäjätunnus ja salasana. Tämän jälkeen on vielä yksi radio button, joka muistaa käyttäjätunnuksen ja salasanan.



Kuva 17. Katalon Studion Object Repository

Katalon hakee objektit testejä varten Object Repositorystä, jotta testit pysyvät toimivina, vaikka sivuston rakenne muuttuisi. Testi etsii testattavan elementin

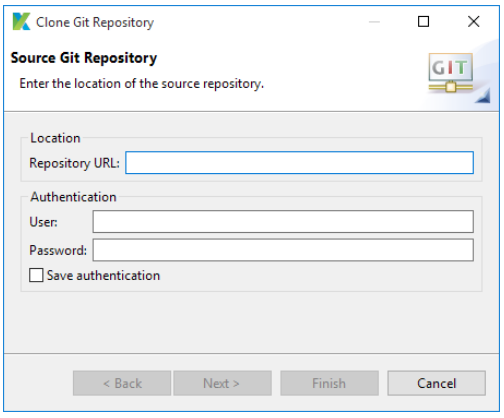
sivuston lähdekoodista tämä seurauksena, vaikka sivuston rakenne ja painikkeiden paikat muuttuvat testi ei rikkoudu ainakaan tämän seurauksena. Katalon käyttää automaattisesti salasanakentissä enkryptattua tekstiä. Jos jostain syystä ei haluta käyttää enkryptattuja salasanoja voi vaihtaa setEncryptedText lauseen tilalle normaalin setText. Katalon voi myös enkryptata tekstiä manuaalisesti yläpalkin Help valikon kohdasta Encrypt Text.

```
WebUI.openBrowser('')

WebUI.navigateToUrl('https://courses.ultimateqa.com/users/sign_in')

WebUI.setText(findTestObject('Object Repository/Page_Ultimate QA/input_Email_useremail'), 'eemeli1400@student@hamk.fi')

WebUI.setEncryptedText(findTestObject('Object Repository/Page_Ultimate QA/input_Password_userpassword'), 'aeHFOx8jV/A=')

WebUI.click(findTestObject('Object Repository/Page_Ultimate QA/input_Remember me_userremember'))

WebUI.click(findTestObject('Object Repository/Page_Ultimate QA/a_Forgot Password'))

WebUI.click(findTestObject('null'))

WebUI.click(findTestObject('Object Repository/Page_Ultimate QA/a_Sign in with LinkedIn'))

WebUI.click(findTestObject('Object Repository/Page_Ultimate QA/a_Create a new account'))

WebUI.click(findTestObject('Object Repository/Page_Ultimate QA/a_Home'))

WebUI.click(findTestObject('Object Repository/Page_Ultimate QA/a_Support'))

WebUI.click(findTestObject('Object Repository/Page_Ultimate QA/a_Privacy Policy'))
```

Kuva 18. Esimerkki Object Repositoryn käytöstä testeissä

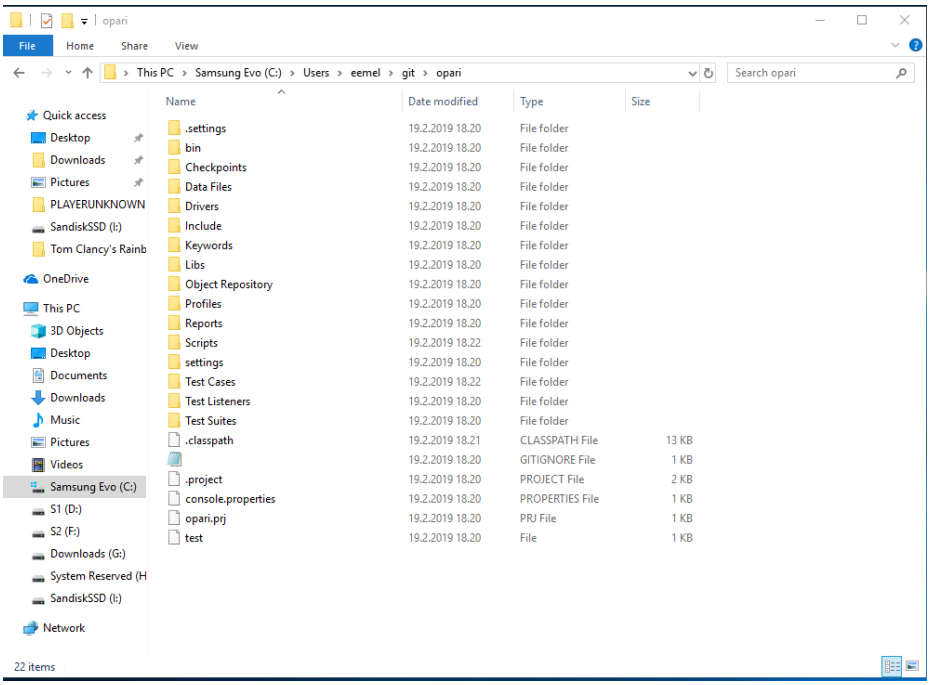
4.4 Katalon Studio GIT-Integraatio

Katalon studiossa testien jakaminen onnistuu sisäänrakennetun Git jakamistoiminnon kautta. Katalon GIT integraatio voidaan aktivoida asetuksista: Window > Katalon Studio Preferences > Katalon > Git.
Lisäasetuksia löytyy tarvittaessa: Window > Preferences > Team > Git. Aktivoinnin jälkeen Git-painike tulee aktiiviseksi ja voidaa alavalikosta kloonata projekti. Valitaan Git valikosta Clone project, jonka jälkeen avautuu seuraavanlainen ikkuna.



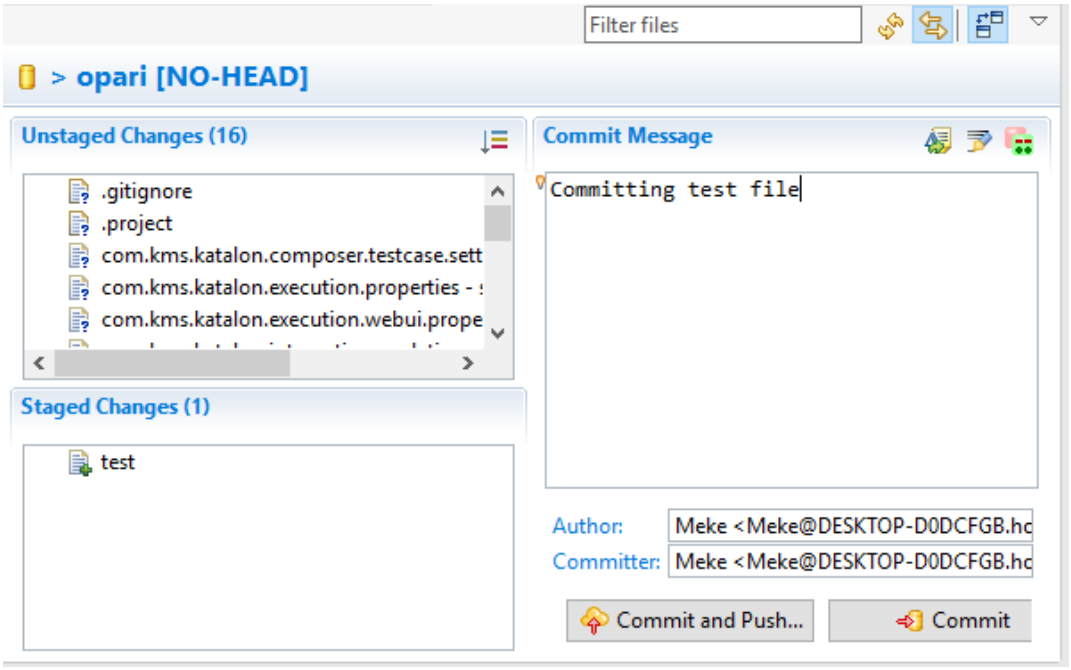
Kuva 19. Projektin Kloonaus

Source Git Repository ikkunaan lisätään oman Git repositoryn URL ja sen käyttäjätunnus ja salasana. Tässä tapauksessa minun oma github url oli esimerkiksi https://github.com/Mege1/opari.
Kloonamisen jälkeen valitaan Git-valikosta Share Project, jotta Katalon saadaan keskustelemaan Git:n kanssa. Katalon kopioi projektin määritettyyn polkuun ja luo .gitignore tiedoston, joka kertoo git:lle mitkä tiedostot se jättää huomioimatta. Gitignore tiedostoon voidaan lisätä haluttuja projektin osia, jotta Katalon osaa automaattisesti jättää ne osat huomioimatta, joita ei haluta jakaa. Jos projekti ei kopioitunut automaattisesti määritettyyn sijaintiin se voidaan kopioida manuaalisesti.



Kuva 20. Polku johon git integroitu projekti kloonautuu

Commit-painikkeella nähdään kaikki projektiin liittyvät tiedostot ja voidaan valita tiedostot, jotka halutaan ajaa local branchiin. Local branch on henkilökohtainen git-hakemisto johon voidaan ensin ajaa muutokset commit toiminnolla. Remote branch tarkoittaa github sivustolla sijaitsevaa projekti hakemistoa, jonka projektiryhmälaiset näkevät. Manage Branches kohdasta voidaan luoda uusia brancheja, vaihtaa nykyistä branchia ja poistaa brancheja. Siirretään tiedostot, jotka halutaan kommitoida staged changes laatikkoon. Alla olevassa kuvassa on otettu projektin test case ”test” ja valittu se kommitoitavaksi. Voidaan joko kommitoida tiedostot tai valita commit and push, joka samaan aikaan ajaa muutokset remote branchiin.



Kuva 21. Projektiin tehtyjen muutosten kommitointi

Fetch hakee kaikki muutokset, joita kuka tahansa projektihenkilö on tehnyt projektiin ja ajanut ne push-toiminnolla remote branchiin. Kannattaa aina ennen projektin jatkamista käyttää fetch-toimintoa, jotta saadaan kaikki uusimmat muutokset, joita projektiin on tehty. Show History näyttää kaikki muutokset, jotka haettiin fetch-toiminnolla. Pull-toiminto hakee muutokset, jotka on tehty remote branchi:ssä ja tuo ne nykyiseen branchiin. Pull ja fetch toiminnon ero on siinä, että fetch hakee kaikki muutokset ja pull toiminnolla voidaan valita mistä branchistä ja mitä muutoksia halutaan tuoda. Push ajaa tehdyt muutokset local branchistä remote branchiin. Kuten jo aiemmin mainittiin push voidaan tehdä myös Kommitoinnin yhteydessä. Jos halutaan käyttää commit-toimintoa vain ajaakseen tehdyt muutokset ensin local branchiin tarvitaan push toimintoa myöhemmin, kun halutaan muutokset kaikkien saataville.

5 YHTEENVETO

Olen tyytyväinen opinnäytetyössä saavutettuun lopputulokseen. Projekti eteni hiukan aikataulusta jäljessä, mutta loppujen lopuksi se saatiin päätökseen. Projektin aikana opin paljon uusia asioita testauksesta, automaatiosta ja Katalon Studiosta. Katalon Studio oli ennestään jonkin verran tuttu työkalu, mutta tämän työn mukana opin paljon uusia ominaisuuksia Katalonista. Opinnäytetyö oli aiheeltaan mielenkiintoinen ja antoi minulle hyviä valmiuksia palata testauksen pariin mahdollisesti tulevaisuudessa. Tämän projektin pohjalta on hyvä lähteä syventämään tietoja ja taitoja automaatiotestauksesta.

Tietotekniikan jatkuva kasvu yhteiskunnassa tietää sitä, että testaukselle ja automaatiolle on tulevaisuudessa paljon kysyntää. Kaikki uudet web-sivut ja ohjelmistot on testattava ennen niiden vapauttamista markkinoille, joko automatisoidusti tai manuaalisesti. Testauksen automatisointi säästää pitkällä tähtäimellä yritykseltä ylimääräisiä työtunteja ja sitä kautta vähentää kuluja.

Kaikkia ongelmia en saanut selvitettyä testejä kirjoittaessa. Esimerkiksi pop-up -ikkunoiden käyttäytyminen oli erittäin haastava osa-alue. Erityisesti se, että kaikki pop-upit ovat erilaisia ja vaativat erilaisen ratkaisun toimiakseen hankaloittaa niiden kanssa työskentelyä. Opinnäytetyö myös poikkesi jonkinverran alkuperäisistä suunnitelmista, mutta kaiken kaikkiaan sain mielestäni tärkeimmät osat työhön sisällytettyä. Katalon Studiolla oli mielestäni mukava kirjoittaa testejä ja sillä pääsee lähes kuka tahansa alkuun automaatiotestauksessa. Muihin automaatiotyökaluihin nähden Katalon on käyttäjäystävällinen ja ei vaadi muita lisäosia toimiakseen. Suosittelen Katalon Studiota kaikille testauksesta kiinnostuneille. Testien eri kirjoitusmenetelmien ansiosta se sopii niin aloittelijoille, kuin kokeneemmille testaajillekkin.

LÄHTEET

Acceptance testing (n.d.). Haettu 10.2.2019 osoitteesta https://en.wikipedia.org/wiki/Acceptance_testing

Guru99 (n.d.). What is System Testing? Haettu 4.5.2019 osoitteesta https://www.guru99.com/system-testing.html

Jenkov, Jakob (2014). A simple unit test. Haettu 3.5.2019 osoitteesta http://tutorials.jenkov.com/java-unit-testing/simple-test.html

Katalon Studio (n.d.). Haettu 20.4.2019 osoitteesta https://en.wikipedia.org/wiki/Katalon_Studio

Software Testing Fundamentals (STF) (n.d.). Unit Testing. Haettu 1.2.2019 osoitteesta http://softwaretestingfundamentals.com/unit-testing/

Software Testing Help (2019) What is Integration Testing. Haettu 29.1.2019 osoitteesta https://www.softwaretestinghelp.com/what-is-integration-testing/

Inflectra (2018) Software Testing Methodologies. Haettu 3.2.2019 osoitteesta https://www.inflectra.com/ideas/topic/testing-methodologies.aspx

Techopedia (n.d.). System Testing. Haettu 2.2.2019 osoitteesta https://www.techopedia.com/definition/22445/system-testing

ForgeAhead (2017) Test Automation Across the Phases of Test Cycle. Haettu 28.1.2019 osoitteesta https://www.forgeahead.io/blogs/test-automation-across-the-phases-of-test-cycle-2/

TRY QA (n.d.). What is Integration testing? Haettu 5.5.2019 osoitteesta http://tryqa.com/what-is-integration-testing/

Unit testing (n.d.). Haettu 4.2.2019 osoitteesta https://en.wikipedia.org/wiki/Unit_testing