Tampereen ammattikorkeakoulu

Päästä päähän -testaus osana jatkuvan julkaisemisen laadunvarmistusta

Tapaus MyRoidu

Miika Mandelin

OPINNÄYTETYÖ
Elokuu 2022

Tietotekniikka
Ohjelmistotekniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikka
Ohjelmistotekniikka

MANDELIN, MIIKA:
Päästä päähän -testaus osana jatkuvan julkaisemisen laadunvarmistusta
Tapaus MyRoidu

Opinnäytetyö 37 sivua
Elokuu 2022

Opinnäytetyön tavoitteena oli web-automaatioon perustuvan testauskehyksen käyttö kattavan testausohjelman luomiseen. Työssä kerrotaan käytetystä kehyksestä ja jatkuvasta julkaisemisesta, mutta työn pääpaino oli toimivan ja vaatimukset täyttävän testausohjelman toteuttaminen. Valmis testausohjelma tuli osaksi jatkuvan julkaisemisen prosessia, ja sillä oli tarkoitus nostaa työn tilaajan tuotteen laadunvarmistusta.

Työn tilaajana oli Roidu Oy Ltd. Roidu on asiakas- ja henkilöstökokemuksen asiantuntijayritys ja auttaa asiakkaitaan kokemustiedolla johtamisessa. Roidun tarjoama MyRoidu online-palvelu sisältää kattavan kirjon ominaisuuksia, joilla asiakaskokemustiedon analysointi ja hyödyntäminen tehdään mahdollisimman helpoksi ja nopeaksi. Työssä toteutettu testausohjelma automatisoi MyRoidu:n testaamisen käyttäjän näkökulmasta. Tämä prosessi oli aikaisemmin manuaalista työtä, joten testaaminen nopeutui ja määritellyt toiminnallisuudet tulee testattua aina, kun uutta toiminnallisuutta ollaan viemässä loppukäyttäjälle.

Työtä varten määriteltiin testattavaksi sellaiset toiminnallisuudet, joiden toiminnat ovat tärkeitä Roidun asiakkaiden tyytyväisyyden takaamiseksi. Näistä muodostunut testiohjelma suorittaa yhteensä 393 testiä kolmessa minuutissa, kattaen kolme eri selainmoottoria. Testausohjelmaa tullaan jatkokehittämään lisäämällä siihen uusia testauskohteita, jolloin saadaan laajempi testikattavuus MyRoiduun ja varmuus siitä, että Roidun asiakkaille tarjotaan aina laadukasta ja toimivaa tuotetta.

Asiasanat: päästä päähän -testaus, jatkuva julkaiseminen, laadunvarmistus

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in ICT Engineering
Software Engineering

MANDELIN, MIIKA:
End-to-End Testing as Part of Continuous Deployment Quality Assurance Case MyRoidu

Bachelor's thesis 37 pages
August 2022

This bachelor’s thesis describes the usage of a web automation testing framework as a tool to create a comprehensive testing program. The thesis covers theory on subjects such as the framework used for the project and continuous integration, but the main focus is on creating a fully functional testing program that fulfills the given requirements. The finished test program will be part of a continuous integration process at the client company and it will increase the quality of the company’s product.

The client of the work for this thesis is Roidu Oy Ltd. Roidu is an expert company in customer and personnel experience and helps its customers to utilize that in management. Roidu offers the MyRoidu online service with a variety of features, which make using and analyzing customer experience as easy and fast as possible. The test program created in this thesis automates the testing of MyRoidu from the user’s perspective. This process used to be manual labor, so the speed of testing increased significantly, and defined functionalities will be tested every time before a new feature is deployed to the end user.

Functionalities whose functions are important to guarantee the satisfaction of Roidu’s customers were selected for the test cases of the test program. As a result, the test program runs a total of 393 tests in three minutes, covering three different web browser engines. Further development of the test program will extend the test cases to include more functionalities of MyRoidu, bringing certainty that Roidu’s customers will always be offered a high quality and functional product.

Key words: end-to-end testing, continuous delivery, quality assurance

SISÄLLYS

LYHENTEET JA TERMIT

Back End	Kommunikoi käyttöliittymän kanssa ja tarkoittaa sivuston palvelimella suoritettavaa koodia.
Botti	Tietokoneohjelma, joka suorittaa sille määriteltyjä tehtäviä automaattisesti.
Bugi	Ohjelmiston virheellinen toiminta.
CAPTCHA	Testi, joka erottelee automatisoidut ohjelmat oikeista käyttäjistä. (Completely Automated Public Turing test to tell Computers and Humans Apart).
DOM	Dokumenttioliomalli (Document Object Model). Mahdollistaa web-dokumentin rakenteen, tyylin ja sisällön muokkaamisen.
Front End	Koodi, joka suoritetaan käyttöliittymässä, esim. nettiselaimessa.
HTML	Hypertekstin merkintäkieli (Hypertext Markup Language).
JavaScript	Ohjelmointikieli, jota käytetään web-ohjelmointiin. Mahdollistaa dynaamisten web-sivujen luomisen.
Node.JS	Node.JS on sovelluskehys, joka mahdollistaa JavaScriptin käytön.
Refaktorointi	Prosessi, jossa lähdekoodia muutetaan siten, että sen toiminnallisuus säilyy, mutta rakenne muuttuu.
Skripti	Skripti voi olla ohjelma, mutta yleisimmin sillä viitataan sarjaan komentoja, joita toinen ohjelma käyttää.
Staging	Kehitysympäristön ja tuotantoympäristön välillä oleva välitila, jossa kehitysympäristön muutokset testataan toimiviksi tuotantoympäristöstä kloonatussa versiossa.
Trello	Projektin ja työtehtävien hallintaan sekä seuraamiseen tarkoitettu visuaalinen työkalu.
Yksikkötestaus	Testaamisen ja laadunvarmistuksen menetelmä, jossa lähdekoodin osat testataan yhdessä tai erikseen.

1 JOHDANTO

Opinnäytetyön toimeksiantaja on Roidu Oy Ltd. Roidu on asiakas- ja henkilöstökokemuksen asiantuntijayritys ja auttaa asiakkaitaan kokemustiedolla johtamisessa. Roidun jatkuvan mittaamisen palvelut tuottavat täsmällistä ja reaaliaikaista kokemustietoa toteuttamalla kaikki mielipidekyselyt lähtökohtaisesti modernisti sähköisten tiedonkeruukanavien kautta ja sähköistä raportointia hyödyntäen. Näin tuotetaan ajantasaisia ja selkeitä raportteja tiedolla johtamisen tueksi. Tämän opinnäytetyön kannalta oleellisin Roidun tarjoama palvelu on MyRoidu. MyRoidu-palvelu tekee asiakas- ja henkilöstökyselyiden tekemisen yksinkertaiseksi, mahdollistaa helpon laitehallinnan sekä esittää kerätyn datan selkeillä kuvaajilla.

MyRoidun tarjoama online-palvelu sisältää kattavan kirjon ominaisuuksia, joilla asiakaskokemustiedon analysointi ja hyödyntäminen tehdään mahdollisimman helpoksi ja nopeaksi. Palvelussa voi lähettää kyselyitä tekstiviestitse ja sähköpostitse tai julkaista palautelaitteilla, verkkosivuilla ja sovelluksissa. Tulokset näkyvät reaaliaikaisesti selkeinä kuvaajina ja mittareina. Tuloksia voi suodattaa ja vertailla monipuolisesti parilla hiiren klikkauksella esimerkiksi toimipisteittäin, ajanjaksoittain tai muilla halutuilla parametreillä. Raportit ovat saatavilla PDF-, Excel- ja Powerpoint-tiedostoina halutuilta aikaväleiltä, laitteilta, toimipisteiltä tai palveluilta. Sähköpostiraportit kiinnostavista tiedoista saadaan niin halutessa valituin määrävälein automaattisesti suoraan sähköpostiin. Palvelussa voi myös luoda automaattisia hälytyksiä halutunlaisista vastauksista ja hälytykset voidaan tiketöidä automaattisesti, jotta niihin reagointia voidaan seurata.

MyRoidu-palvelua kehitetään jatkuvasti paremmaksi tuottaen käyttäjille uusia ominaisuuksia ja parantaen olemassa olevien ominaisuuksien toiminnallisuuksia. Muutokset MyRoiduun tuodaan käyttämällä jatkuvaa julkaisemista. Käyttäjä siis pystyy käyttämään palvelua häiriöittä, ja sivun päivittämisen jälkeen sivulle on saattanut ilmestyä uusi ominaisuus tai jonkin napin toiminnallisuus on voinut muuttua. Jatkuvassa julkaisemisessa on kuitenkin riski, että käyttäjän siirtyessä uuteen näkymään tai painaessa nappia sivusto voi ilmoittaa virheestä. Pahimmillaan koko sivusto tulee käyttökelvottomaksi, jos tuotantoversioon päätyy bugi.

Yleisesti testausvaiheessa käytetään yksikkötestejä varmistamaan toimintojen toiminnallisuus. Yksikkötestauksessa testattaville funktioille kuitenkin annetaan vain parametrit ja verrataan lopputulosta oletettuun tulokseen. Käyttäjän toimintaa ei oteta huomioon. Tuomalla päästä päähän -testaus osaksi jatkuvan julkaisemisen tuotantoputkea voidaan bugit pysäyttää tehokkaammin ennen kuin ne päätyvät tuotantoversioon. Päästä päähän -testaus imitoi käyttäjän toimintaa sivustolla. Sillä voidaan esimerkiksi suorittaa hiiren klikkaus elementin päällä ja syöttää tekstiä tekstikenttään.

Yksikkötestauksen ja päästä päähän -testauksen eroja voidaan havainnollistaa yksinkertaisen valintaruudun testaamisen kautta. Yksikkötestauksella voidaan testata, että valintaruudun taustalla olevalle funktiolle voidaan antaa tosi tai epätosi arvo ja arvo muuttuu. Päivittämällä yksi käyttöliittymän kehyksen monista kirjastoista saatetaan kuitenkin rikkoa valintaruudun todellinen toiminta. Tästä oli esimerkkitapaus alkuvuodesta MyRoidu-palvelussa. Sivusto toimi näennäisesti normaalisti ja virheilmoitusta ei tullut. Käyttäjä ei kuitenkaan voinut painaa valintaruutua, eikä valinta näin ollen tullut aktiiviseksi. Päästä päähän -testauksella voidaan navigoida eri sivuille ja testata käyttäjän näkökulmasta toimintaa. Jos valintaruudun toiminnan tarkastus olisi edellisessä esimerkissä ollut määriteltynä osaksi testejä, sen toimimattomuus olisi huomattu ennen ongelman tuotantoversioon päätymistä.

Opinnäytetyön projektissa testauskohteiksi määriteltiin MyRoidun kriittisimmät palvelut, joiden toiminta jokaisen julkaisun yhteydessä halutaan varmistaa. Näitä ovat

- kyselyn luominen
- yleisimpien kysymyselementtien lisääminen kyselyyn
- kyselyn julkaiseminen
- kyselyyn vastaaminen
- tulosten näkyminen raportointisivulla.

Projektin päästä päähän -testaamisen kehykseksi valittiin Playwright, koska se on tuttu ja ollut käytössä jo aikaisemmin Roidulla. Playwrightillä on luotu Roidulle testaustyökalu osaksi kyselyiden räätälöintiä. Työkalulla voitiin avata parametrina anneutu julkaisulinkki, navigoida kysely läpi antaen vastauksia ja ottaen kuvakaappauksia jokaiselta sivulta. Työkalu oli luotu suorittamaan vain edellä mainittuja asioita. Tämän lisäksi sillä ei voi antaa vastauksia kyselyyn, jonka ulkoasu on lomake, joten vanhaa työkalua ei voinut hyödyntää muun kuin kokemuksen kautta. Tässä työssä luotiin huomattavasti laajempi testaustyökalu. Uusi Työkalu kykenee kirjautumaan MyRoiduun, luomaan automaattisesti uuden kyselyn, julkaisemaan sen, käyttämään julkaisulinkkiä itsenäisesti kyselyyn vastaamisen avaamiseen, antamaan vastaukset kyselyihin, joiden ulkoasut ovat oletus ja lomake. Kyselyyn tulleiden vastauksien tarkastaminen onnistuu myös.

Raportissa kerrotaan yleisesti päästä päähän -testauksesta ja työhön valikoituneesta testauskehyksestä sekä jatkuvasta julkaisemisesta. Työn pääpaino oli kuitenkin toimivan ja vaatimukset täyttävän testausohjelman toteuttaminen, joten tätä prosessia on käyty läpi laajemmin vaihe vaiheelta. Työvaiheet ovat kronologisessa järjestyksessä, koska jokaisen vaiheen toteuttaminen oli toisistaan riippuvaisia. Raportin lopussa on pohdinta koko työstä.

2 PÄÄSTÄ PÄÄHÄN -TESTAUS

2.1 Yleistä

Internet-sivuja on kahdenlaisia, Staattisia sekä dynaamisia. Staattiselle internet-sivulle on ominaista se, että sen sisältö ei muutu, ellei sen taustalla olevaa tiedostoa itsessään muuteta. Dynaamiselle internet-sivulle ominaista on sen sisällön luominen joka kerta, kun sen sisältö käsitellään. Tämän ansiosta dynaamisella internet-sivulla voi olla sellaisia ominaisuuksia, mitä staattisella ei voi olla, kuten sivuston reagointi käyttäjän antamaan syötteeseen ja sivuston räätälöinti erikseen jokaista käyttäjää varten. Dynaaminen internet-sivu voi esittää dataa, jota haetaan käyttäjän toimesta erilaisilla hakusanoilla tietokannasta. Tämän lisäksi käyttäjä voisi esimerkiksi määritellä käyttäjätunnuksensa asetuksissa mielenkiintonsa kohteita tai käyttöliittymän kielen. Sivu voi käyttää näitä tietoja määritellessään mitä ja minkälaista tietoa se esittää käyttäjälle. (Connolly & Begg 2002.)

Kuten useimmat modernit internet-sivut, tämän projektin kohteena oleva MyRoidu on dynaaminen. MyRoidussa voidaan luoda kyselyitä, julkaista niitä ja tarkastella kyselyyn tulleita vastauksia. Tietokantaan voidaan siis lisätä ja siellä olevaa tietoa voidaan hakea sekä muokata. Sisällön julkaisemisessa staattiselle internet-sivulle riittää, että tarkastetaan sivun ulkonäön ja sisällön olevan oikein. Dynaamisella internet-sivulla on tärkeää testata, että siellä olevat toiminnallisuudet, vanhat sekä uudet, toimivat oikein aina kun julkaistaan uutta toiminnallisuutta tai muokataan vanhaa.

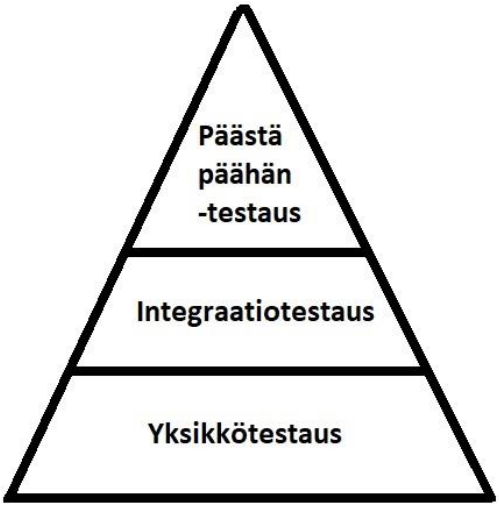
Ohjelmistotekniikassa ohjelmiston testaamiseen käytetään laajasti yksikkötestejä. Yksikkötestit kuitenkin testaavat vain pieniä osia ohjelmasta, ja niitä kirjoitetaan jokaiselle funktiolle erikseen. Näille testeille annetaan parametrit, ja kohteena olevan funktion toiminta testataan. Päästä päähän -testaus tarjoaa mahdollisuuden testata kerralla koko ohjelma. Testauksen aikana voidaan varmistaa kaikkien komponenttien toiminta yksitellen ja niiden vaikutus muihin komponentteihin. Päästä päähän -testaus imitoi ohjelmiston käyttäjää ja soveltuu erittäin hyvin web-kehityksen laadunvarmistukseen, koska päästä päähän -testauksella

voidaan tarkastaa, onko elementti näkyvillä selaimessa, ja elementistä riippuen, voiko sitä klikata tai syöttää siihen tekstiä. Käyttöliittymän testaamisen lisäksi testataan myös varsinainen toiminnallisuus. Käyttäjä voi esimerkiksi painaa käyttöliittymässä nappia, joka lisää sivulle uuden elementin. Päästä päähän -testaamisella voidaan tarkastaa, että pyyntö lähetettiin ja sille tuli onnistumisesta ilmoittava vastaus. Tämä varmistaa sen, että käyttöliittymän taustalla olevan napin funktio ja siihen liittyvät vuorovaikutukset toimivat niin kuin pitääkin.

On olemassa kahden tyyppistä päästä päähän -testaamista. Näitä ovat pystysuora (vertikaalinen) ja vaakasuuntainen (horisontaalinen) testaaminen. Vaakasuuntainen on näistä kahdesta yleisimmin tunnettu ja käytetty lähestymistapa päästä päähän -testauksessa. Vaakasuuntainen testaaminen tuo varmuutta järjestelmään käyttämällä käyttäjän näkökulmaa. Siinä selvitetään, voiko käyttäjä käyttää järjestelmää ilman, että bugeja tai virhetilanteita tapahtuu. (Gillis 2018.)

Pystysuora päästä päähän -testaaminen viittaa toisistaan erillisiin komponentteihin. Tässä testaaminen tapahtuu yksi kerrallaan hierarkkisessa järjestyksessä. Kaikki järjestelmän tai tuotteen komponentit testataan alusta loppuun, jotta niiden laatu voidaan varmistaa. Pystysuoraa testaamista käytetään usein sellaisten monimutkaisten laskentajärjestelmien kriittisten komponenttien testaamiseen, joilla ei ole käyttöliittymää tai jotka eivät koske käyttäjiä. (Gillis 2018.)

Kuviossa 1 esitetään, miten päästä päähän -testaaminen sijoittuu hierarkkisessa järjestyksessä muihin käytettyihin testeihin. Kuvio esittää visuaalisesti erilaisten testien tarpeen määrällisesti. Yksikkötestaus on pyramidin alin ja isoin lohko. Yksikkötestejä tulisikin olla määrällisesti eniten. Päästä päähän -testaaminen on tärkeätä, mutta hitaampaa verrattuna yksikkötestaamiseen, koska sillä voidaan testata koko järjestelmää. Siksi päästä päähän -testaaminen on pyramidin huipulla ja parhaan hyödyn siitä saa, kun sitä käytetään tärkeimpien toimintojen testaamiseen.



KUVIO 1. Testauskolmio.

Tozzi (2021) listaa viideksi parhaaksi JavaScript-testiautomaatiokehykseksi vuodelle 2021 seuraavat kehykset:

- WebdriverIO
- Cypress
- TestCafe
- Playwright
- Puppeteer.

2.2 Playwright

Tässä projektissa käytetty päästä päähän -testauksen kehys Playwright tarjoaa mahdollisuuden testata toimivuutta kolmella eri selaimella, joita ovat Chromium, Safari, Google Chrome ja Firefox. Playwrightin käyttämät funktiot ovat promise-kutsuja, joten asynkronisen funktion sisällä lähetetyn kutsun, kuten esimerkiksi getProperty-pyynnön vastausta voidaan jäädä odottamaan käyttämällä await-operaattoria kutsun edessä ennen siirtymistä seuraavaan komentoon. Ilman erillistä määrittämistä kutsujen onnistumista odotetaan 30 sekuntia ennen kuin aikakatkaisu tapahtuu, mutta maksimiodotusajan pystyy määrittelemään jokaiselle kutsulle erikseen. Ohjelma ilmoittaa testin epäonnistuneen, jos aikakatkaisu tapahtuu. Testin käynnistyessä ohjelma avaa uuden selaimen halutulla selainmoot-

torilla ja navigoi annetulle nettisivulle. Playwright voi todentaa määritellyn elementin näkyvyyden ja yrittää käyttää sitä. (Playwright enables reliable end-to-end… 2022.)

Playwrightin pääsäännölliset käyttökohteet ovat testiautomaatio moderneissa web-sovelluksissa, selainten välinen testaus, kuvakaappausten ottaminen ja datan kerääminen nettisivuilta. Automaatiolla voidaan varmistaa, että käyttäjille näytettävät ominaisuudet ja toiminnallisuudet käyttäytyvät niille tarkoitetulla tavalla. Selainten välisellä testauksella voidaan varmistaa, että web-sovellukset toimivat useilla selaimilla ja niiden moottoreilla. Kuvakaappaukset voidaan joko tallentaa myöhempää tarkastelua varten tai käyttää niitä ohjelman ajon aikana visuaalisessa vertailussa. Kerätyllä datalla voidaan suorittaa myöhemmin jäljitystä tai analyysejä. (Checkly 2022.)

Kyseistä päästä päähän -testauksen kehystä päädyttiin käyttämään tässä projektissa, koska se oli jo entuudestaan tuttu. Kehyksen käyttö oli valikoitunut käyttöön jo aikaisemmassa projektissa, jossa kehitettiin kyselyiden ulkoasun kustomointiin aputyökalu, joka osaa navigoida parametrinä annettuun nettiosoitteeseen ja vastata dynaamisesti kysymyksiin sekä edetä seuraavalle sivulle, vaikka nettiosoite ei muuttuisi ja elementit saattaisivat olla joka kerta erilaiset.

3 JATKUVA JULKAISEMINEN

Suomenkielisesti puhutaan jatkuvasta julkaisemisesta. Yleisesti ottaen se tarkoittaa prosessia, jossa palveluun tai ohjelmistoon voidaan tuoda uusia versiopäivityksiä ilman, että siitä syntyisi katkoksia palvelussa. Suomenkielisestä termistä ei kuitenkaan ilman asiayhteyttä tiedetä suoraan mitä sillä tarkoitetaan, koska englanniksi termi saa kaksi käännöstä ja molempia käytetään ohjelmistotekniikassa. Jatkuva julkaiseminen voi tarkoittaa englanninkielisistä termiä ”continuous deployment” tai ”continuous delivery”. Vaikka nämä molemmat tarkoittavat lähes samaa asiaa, niissä on ero.

Molemmissa prosesseissa jokainen asennusvaihe on automaattinen. Suurin ero tulee siitä, että Continuous deployment -mallissa kaikki muutokset julkaistaan automaattisesti. Tästä muodostuu uusi julkaistu versio. Continuous delivery -mallissa on tarkoitus saattaa muutokset valmiiksi julkaisua varten ja ihminen tekee päätöksen tuotantoon viemisestä. Continuous deployment -malli mahdollistaa nopeamman ja tehokkaamman versiojulkaisun, mutta tuotantoon saattaa päätyä virheitä, jos automatisoituja testejä ei ole tehty kunnolla. Continuous delivery -malli on helpommin hallittavissa, koska virheitä voidaan korjata ennen julkaisemista. (Huotarinen 2016.)

Jatkuvan julkaisemisen prosessissa käytetään jatkuvaa integraatiota. Jatkuvassa integraatiossa yksittäisen kehittäjän tekemät muutokset havaitaan ja niille suoritetaan automatisoidut testit. Tämän tarkoitus on tunnistaa mahdollisten ongelmien esiintyminen uusissa muutoksissa ja estää virheiden pääseminen eteenpäin. Kun muutokset on validoitu, ne yhdistetään koodikantaan. (Laster 2017.)

Jatkuva julkaiseminen ja jatkuva integraatio ovat Safen (2021) mukaan jatkuvan julkaisemisen tuotantoputken alemman tason käsitteitä. Jatkuvan julkaisemisen tuotantoputki sisältää kaikkiaan neljä käsitettä. Näitä ovat jatkuva tutkiminen, jatkuva integraatio, jatkuva julkaiseminen ja julkaisu tarpeen mukaan. Jatkuvan julkaisemisen tuotantoputki esittää kaikki ne vaiheet, joita tarvitaan uuden ominaisuuden viennissä ideasta valmiiksi julkaisuksi, joka voidaan julkaista. (Safe 2021.)

4 TOTEUTUS

4.1 Vaatimukset

Vaatimuksia määriteltäessä haluttiin varmistaa, että ohjelmasta saataisiin mahdollisimman yleiskäyttöinen ja hyödyllinen, tietoturvaa unohtamatta. Ohjelman tulisi kattaa kaikki MyRoidun käyttäjien yleisimmin käytetyt ominaisuudet ja toiminnallisuudet, jotta voidaan varmistua siitä, että MyRoidun käyttäjäkokemus on mahdollisimman hyvä. Seuraavassa kappaleessa on esitetty miten tähän tavoitteeseen päästään.

Ohjelmalle pitää voida antaa parametrina URL-osoite, jolla määritellään, mitä sivua testataan. Tämä mahdollistaa staging-version testaamisen lisäksi paikallisten ja tuotantoon vietyjen versioiden testaamisen. Tietoturvasyistä ohjelman käyttämiä käyttäjätunnuksia ei saa lukea suoraan tiedostosta vaan ne tulee ajaa skriptillä ympäristömuuttujiin. Ohjelman tulee kirjautua MyRoidu-portaaliin kolmella eri selaimella, luoda kysely, julkaista siitä kaksi versiota erilaisilla rakenteilla, vastata molempiin versioihin viisi kertaa per selain ja varmistaa vastausten oikeellisuus raportointisivulla.

4.2 Työvaiheet

Tässä luvussa kuvataan työvaiheet ja miten ne toteutettiin. Projektin suunnitteluvaiheessa määriteltiin halutut testauskohteet ja suunnitelman perusteella pilkottiin koko projekti pienemmiksi osasuorituksiksi, joilla projektia edistetään järkevästi ja johdonmukaisesti. Tätä kokonaisuutta hallittiin käyttämällä Trelloa, joka on visuaalinen työkalu projektien hallintaan, jossa työtehtävät esitetään kortteina.

Projektin vaiheet etenivät samassa järjestyksessä kuin ne on esitetty tämän luvun alaluvuissa. Kaikki vaiheet ovat taakse päin riippuvaisia toisistaan, koska kyselyyn tulleita vastauksia ei voida validoida ennen kuin kyselyyn on vastattu ja kyselyyn ei voida vastata ennen kuin se on julkaistu ja niin edelleen.

4.2.1 Projektin alustaminen

Projekti perustettiin Bitbucketiin luomalla sinne oma repositorio projektia varten. Kun projektille oli luotu repositorio, se kloonattiin paikalliseen ympäristöön eli työtietokoneelle. Repositorion luomisen yhteydessä valittiin, että projektiin sisällytetään readme-tiedosto. Kyseinen tiedosto kertoo, mistä projektissa on kyse ja miten se saadaan käyttöön mille tahansa tietokoneelle. Tiedoston sisältöä päivitettiin projektin aikana ja se viimeisteltiin projektin lopussa. Käyttämällä Git-versionhallintaa jokainen uusi ominaisuus ja readme-tiedoston versio lisättiin paikallisesta kehitysympäristöstä repositorioon.

Seuraavaksi piti asentaa Playwrightin test-kirjasto, joka sisältää Playwrightin perustoiminnallisuuksien lisäksi test-lohkojen käytön. Asentamiselle löytyy komento ”npm i @playwright/test”, joka ajetaan terminaalissa repositorion hakemistossa. Oletuksena on, että NodeJS on jo asennettuna tietokoneelle. Asennuksen yhteydessä asentuu kolme selainta, joita ovat: Chromium, Safari ja Firefox. Playwright tukee myös Microsoft Edgeä, mutta se pitäisi asentaa manuaalisesti, ja koska projektissa sille ei ollut tarvetta, sitä ei asennettu.

Huomioitavaa projektissa oli, että projektin luonteen takia testausohjelman toiminen kokonaisuuden kannalta vaatii sitä, että testit ajetaan järjestyksessä sen sijaan, että ne ajettaisiin asynkronisesti kaikki yhtä aikaa rinnakkain. Tämän lisäksi Playwrightin test-lohkot itsessään sekä tavalliset komennot ovat promise-funktioita. Jos koodissa yritetään edetä heti, kun pyyntö on lähetetty selaimelle sen sijaan, että odotettaisiin vastausta ja edettäisiin vasta sen jälkeen, saattaa ilmetä odottamattomia tilanteita tai olemattoman muuttujan käsittelyä, joka johtaa virheeseen.

Promise-funktioissa voidaan käyttää await-operaattoria ennen kutsua, jos niitä käytetään asynkronisen funktion sisällä. Tämä tarkoittaa sitä, että koodissa ei edetä seuraavalle riville ennen kuin on saatu vastaus pyyntöön. Testilohkon eteen ei kuitenkaan voi lisätä kyseistä await-operaattoria, mutta test-kirjastosta löytyy oma metodinsa tälle. Metodilla ”test.describe.serial” voidaan luoda yksi iso

testilohko, jonka sisälle voidaan luoda uusia testilohkoja, ja niitä ajetaan vuoron perään järjestyksessä. Jokainen tällainen uusi testilohko luodaan mallilla

```
test('lohkon nimi', async () => {
        koodia
        koodia
});
```

Käytettäessä yhtä isoa testilohkoa tulee huomioida se, että jos lohkon missä tahansa suoritusvaiheessa ilmenee virhe, loppuja testejä ei ajeta vaan kyseessä olevan selainmoottorin osalta testit lopetetaan ja terminaalissa ilmoitetaan virheestä. Tässä projektissa kuitenkin suoritetaan samat testit kolmeen kertaan eri selainmoottoreilla, joten jos kaksi menee läpi ja yksi epäonnistuu, virhe johtuu todennäköisesti prosessoinnin resurssipulasta tai internet-yhteyden kuormittumisesta. Virhe voi kuitenkin johtua siitäkin, että testauksen kohde ei vain toimi kyseisellä selainmoottorilla. Olisi siis hyvä ajaa testit uudestaan, jos virhe on ilmaantunut, ja ryhtyä korjaustoimenpiteisiin, mikäli samassa kohdassa ilmenee ongelma uudelleen.

4.2.2 Käyttäjätunnukset

Testejä on tarkoitus pääsääntöisesti ajaa MyRoidun staging-versiota vasten, joten stagingille lisättiin tunnukset, joita ohjelma käyttää. Näillä tunnuksilla ei voi kirjautua varsinaiseen MyRoiduun, joten jos testiohjelma halutaan ajaa siellä, täytyy sitä varten luoda omat käyttäjätunnukset ja asettaa ne ympäristömuuttujiin.

Tietokantaan lisättiin kaksi yritystä, ”E2E company X” ja ”E2E company Y”. Tämän lisäksi tietokantaan lisättiin käyttäjiä erilaisilla käyttöoikeuksilla ja kukin niistä liitettiin jompaankumpaan yritykseen, jotta voidaan testata tiettyjä jakamis- ja pääsyoikeuksia ja niiden toiminnan virheettömyyttä.

Yritykselle X tehtiin superadmin, pääkäyttäjä, tavallinen käyttäjä kyselyn muokkausoikeuksilla, tavallinen käyttäjä raportin katseluoikeuksilla ja tavallinen käyttäjä laitehallinnan oikeuksilla. Yritykselle Y tehtiin pääkäyttäjä, tavallinen käyttäjä kyselyn muokkausoikeuksilla ja tavallinen käyttäjä raportin katseluoikeuksilla.

Käyttäjätunnukset asetettiin suoraan ympäristömuuttujiin, kun projekti vietiin tuotantoputkeen, mutta tämän lisäksi on olemassa skriptitiedosto, josta tunnukset voidaan viedä ympäristömuuttujiin, jos ohjelmaa haluaa ajaa paikallisesti. Tiedosto lisättiin .gitignore-tiedostoon, jotta se ei vahingossa päätyisi repositorioon ja sitä kautta vääriin käsiin. Käyttäjätunnukset saadaan käyttöön suorittamalla terminaalissa projektin hakemistossa komento ”*source ./setup-local-env.sh*” ennen ohjelman käynnistämistä.

4.2.3 Kirjautuminen

Huomattavana haasteena tässä projektissa oli MyRoiduun kirjautuminen automaattisesti, koska kirjautumisen yhteydessä on käytössä CAPTCHA, jonka tarkoitus on nimenomaan estää automaattiset kirjautumiset.

CAPTCHA-testi on luotu erottelemaan automaattiset ohjelmat oikeista käyttäjistä ja se on akronyymi sanoista "Completely Automated Public Turing test to tell Computers and Humans Apart." Useimmat internetin käyttäjät ovatkin törmänneet joskus CAPTCHAan selatessaan internetiä, koska se on yleinen työkalu taistelussa botteja vastaan. (How CAPTCHAs work… 2022.)

CAPTCHAn automaattiseen ohitukseen on olemassa ratkaisu, ja sen tarjoaa 2Captcha maksullisena palveluna. 2Captcha on ihmisvoimalla toimiva kuva- ja CAPTCHA-tunnistuspalvelu. Sen päätarkoituksena on ratkaista CAPTCHAt nopeasti ja tarkasti ihmisten toimesta. Palvelun käyttäjälle tarjotaan API-avain, jonka avulla käyttäjä voi lähettää kuvan tai CAPTCHA-haasteen HTTP-kutsuna POST-metodia käyttäen ja saa vastauksena ratkaisun, jolla voi ohittaa haasteen ja jatkaa sisäänkirjautumisprosessissa. Palvelu mainostaa prosessin kestoksi 10–40 sekuntia.

MyRoidun paikallisen kehityksen versiossa on kuitenkin käytössä reCaptchan (käyttää edistyneitä riskianalyysin tekniikoita eroittelemaan ihmiset boteista) testiversio. Tämä eroaa tavallisesta CAPTCHAsta siten, että siihen riittää pelkkä valintaruudun klikkaaminen ja testissä itsessään on varoitusteksti. Varoitus ilmoittaa

testin olevan vain testauskäyttöön. Playwrightillä pystyy aktivoimaan kyseisen testin elementtikahvan ja kaivamaan sieltä esille elementin id:n eli tunnistetiedon. Tällä tunnistetiedolla voidaan suorittaa hiiren painallus valintaruudun päällä, jolloin kirjautuminen on mahdollista. Projektissa päädyttiin tuomaan sama CAPTCHA myös staging-version sisäänkirjautumiseen, mutta koska kyseessä on vain testiversio eikä oikea CAPTCHA, herää huoli tietoturvasta. Tietoturvaa kasvatettiin lisäämällä staging-versioon IP-suodatus. Tämä estää ulkopuolisten eli määrittelemättömistä IP-osoitteista sivustolle pyrkivien pääsyn edes yrittämään kirjautumista sivustolle.

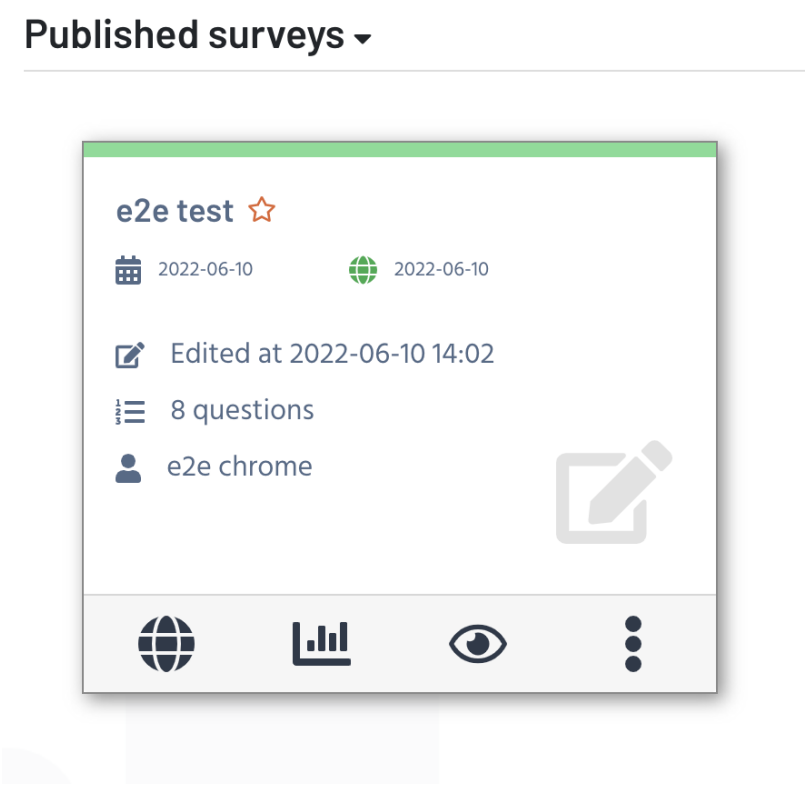
Testausohjelma navigoi parametrina annetulle kirjautumissivustolle, syöttää käyttäjätunnukset, jotka se saa turvallisesti suoraan ympäristömuuttujista. Tämän jälkeen se suorittaa hiiren painalluksen CAPTCHAn päällä, jolloin ”kirjaudu sisään” -nappi aktivoituu. Aktivoitumisen jälkeen ohjelma painaa ”kirjaudu sisään” -nappia. Tämän jälkeen ohjelmalla on kirjautumisessa käyttämänsä käyttäjätunnuksen mukaiset käyttöoikeudet MyRoidun staging-versioon, joka on tuotantoversiota vastaava.

4.2.4 Kyselyn poistaminen

Ohjelman tulee poistaa aikaisemmin luotu kysely, jos sellainen on olemassa, jotta tietokanta ei roskaannu turhista kyselyistä. Kyselyn voisi poistaa joko ennen uuden kyselyn luomista tai kaikkien testien suorittamisen jälkeen. Valitsin testin poistamisen ohjelman alussa, koska se mahdollistaa testien jälkeen testikyselyn manuaalisen tarkastamisen tarvittaessa.

Testikäyttäjälle saattaa olla vahingossa tai tarkoituksella jaettuja muita kyselyitä, joten testiohjelman täytyy käydä läpi kaikki näkyvillä olevat kyselyt ja tarkastaa niiden nimet. Testikyselyn nimi on aina "e2e test", joten kaikki kyselyt, joiden nimessä kyseinen teksti esiintyy, poistetaan. Komento *”const surveyCards = await page.$$(’.card-title’)”* tallentaa kaikki elementit, joilla on luokkatunnus ”card-title” surveyCards-nimiseen muuttujaan. Tämän muuttujan avulla ohjelma tietää, montako kyselyä käyttäjällä on näkyvissä, ja ne voidaan kyselyiden lukumäärän mukaan käydä läpi silmukassa tarkastaen niiden nimet ja poistaa halutut kyselyt.

Käyttäjän käytettävissä olevat kyselyt näytetään surveys-sivulla kortteina. Korttien rakenne aiheuttaa ongelmia automaatiolle; suoralla tekstillä elementtikahvan kaappaaminen ei ole hyvä tapa, koska käyttäjän asetuksista riippuen järjestelmäkieli MyRoidussa voi olla suomi tai englanti. Playwright sallii elementtitunnisteiden ja luokkanimien käytön, joten niillä saadaan luotua järjestelmäkielestä riippumaton toiminto. Kuvassa 1 esitetään korttinäkymä kyselyt-sivulla.



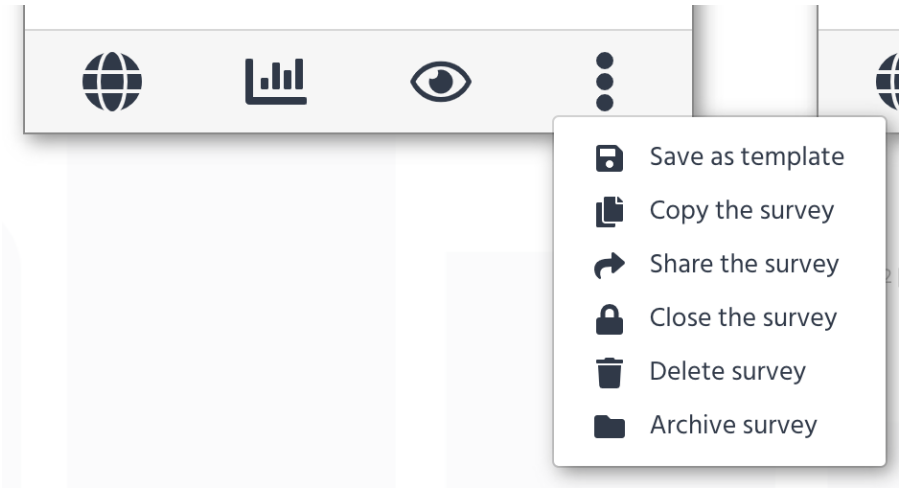
KUVA 1. Kyselyn kortti.

Ensin pitää hakea kaikki title-elementit, koska niistä saadaan element.innerText() komennolla kyselyn nimi. Kun haluttu title löytyy, pitää sille hakea ympäröivä elementti. Tälle elementille pitää hakea vielä sen ympäröivä elementti. Tällä elementillä päästään käsiksi halutun elementin alaviitteeseen, jota tarvitaan, jotta automaatio osaa painaa oikean elementin delete-nappia. Kuvassa 2 näkyy prosessi halutun kyselyn poistamiselle.

```
const surveyCards = await page.$$(".card-title");
for (let y=0; y<surveyCards.length; y++) {
  const surveyName = await surveyCards[y].innerText();
  if (surveyName.includes('e2e test')) {
    const cardBodyElement = await surveyCards[y].getProperty('parentNode');
    const cardElement = await cardBodyElement.getProperty('parentNode');
    const surveyCardID = await page.evaluate(el => el.id, cardElement);
    await page.click(`[ID='${surveyCardID}'] .card-footer .row .col.footer-other .dropdown .dropdown-toggle .fa-icon path`);
    await page.click(':text("Delete survey"):visible');
    await page.click('button:has-text("OK")');
  }
}
```

KUVA 2. Kortin poistaminen.

Kyselyn poistaminen korttinäkymässä tapahtuu painamalla kuvassa 1 ja 3 näkyvää kolmea allekkaista palloa. Painalluksen jälkeen tulee esille kuvan 3 mukainen toimintovalikko. Tässä valikossa tulee painaa kyselyn poistamista, minkä jälkeen tulee vielä esille varmistusmodaali, jossa tulee painaa ”OK”, jotta kysely poistetaan. Tämä ilmenee kuvassa 2 kolmen eri *click*-komennon suorittamisella sen jälkeen, kun elementin kahva on saatu.



KUVA 3. Toimintovalikko.

4.2.5 Kyselyn luominen

Kyselyn luomisessa haasteena oli mm. automatisoidun suorituksen liian nopea eteneminen. Piti osata odottaa oikeissa kohdissa waitfor-komennoilla, jotta elementtejä pystyi ylipäätään painamaan, koska DOM ei ollut vielä ehtinyt päivittyä. DOM tarkoittaa dokumenttioliomallia (Document Object Model) ja se mahdollis-

taa web-dokumentin rakenteen, tyylin ja sisällön muokkaamisen. Kysymyselementin tallennuksen jälkeen, jos ei odottanut ja painoi ”uusi sivu” -nappia, sivusto ilmoitti virheestä, koska tallennuksen jälkeen kyselyn versionumero muuttuu ja uuden sivun lisääminen tunnistaa eriävän version.

Front end päivittää *version*-nimistä muuttujaa ja lähettää kutsun back endiin, jotta uusi versio kyselystä tallentuisi tietokantaan. Kun ”uusi sivu” -nappia painetaan, lähtee back end -kutsu, jossa on mukana kyselyn versio, joka on tässä tapauksessa juuri päivittynyt versionumero. Koska edellinen kutsu ei ole vielä palautunut back endistä, tietokanta luulee edelleen, että kyselyn versio on sama kuin ennen kumpaakaan edellistä kutsua. Tämä aiheuttaa virhetilanteen, ja selain ilmoittaa versioiden ristiriidasta erillisellä modaalilla, eivätkä viimeisimmän kutsun muutokset jää voimaan.

Tallentamattomalla elementillä on luokkatunnus ”sketch”, ja se katoaa, kun elementti on tallentunut ja nettiselain ymmärtää tilanteen. Playwrightiltä löytyy toiminto *waitForSelector()*, jolle voi antaa parametriksi kohteena olevan elementin luokan ja tila-arvon ”detached”. Tämän jälkeen se jää odottamaan enintään käytössä olevaan kutsun aikakatkaisuarvoon asti, että DOMista ei löydy waitForSelector -toiminnon valitsijalle annettua luokkatunnusta. Kun kyseinen ”sketch”-luokka on kadonnut tarkasteltavasta elementistä, on turvallista jatkaa, eikä virhettä tule.

Kaikille elementeille ei ole kuitenkaan annettu MyRoidussa tunnistettavaa luokkaa, vaan käytössä saattaa olla vain geneerinen luokkatunnus. Esimerkiksi ”row-control row” esiintyi muutamassa paikassa. Joissakin ympäröivissä elementeissä oli myös peräkkäin nappeja tai valintaruutuja, ja näistä näkyi ulospäin vain peräkkäin olevia luokkatunnuksia, joten piti löytää vaihtoehtoisia ratkaisuja. Ratkaisin ongelman käyttämällä mm. CSS-valitsijoita *right-of(:text=”size”)* ja *:nth-match(.choices-row [type=”text”], 1)*. Näillä valitsijoilla pystyy valitsemaan elementin tietyn tekstin vierestä tai valitsemaan esimerkiksi järjestyksessä toisen elementin, jolla on näkymässä luokkatunnus ”row-control row”. Tälläisten valitsijoiden käyttö kuitenkin on kyseenalaista, koska testattavan kohteen jatkokehityksessä teksti saattaa muuttua tai samalle luokkatunnukselle saattaa tulla uusi elementti, joka voi muuttaa järjestysnumeroa riippuen uuden elementin sijainnista.

Tämä tulee ottaa huomioon jatkokehityksessä tai muokata testausohjelmaa, kun muutoksia ilmenee. Ongelmaa voi hallita antamalla mahdollisimman tarkasti valitsijalle parametrit. Tästä esimerkkinä toimii kuva 4, jossa valitsijalle on kerrottu, että sen tulee valita kolmas elementti, jolla on ”matrixID”:n mukainen ID-arvo ja luokkatunnuksina ”highcharts-stack-labels”, ”highcharts-label” sekä elementtitunnus ”text”.

```
await expect(await page.locator(`:nth-match([ID='${matrixID}'] .highcharts-stack-labels .highcharts-label text, 3)`)).toContainText("5");
```

KUVA 4. CSS-valitsijan tarkempi määrittely.

Kyselyn sivujen luominen on testiohjelmassa lohkottu siten, että jokainen erilainen kysymyselementti ja samalla uusi sivu (yhdelle sivulle voisi lisätä useita kysymyksiä, mutta projektissa on tarkoituksenmukaista luoda uusi sivu per kysymys) on oma testilohkonsa. Tällä tavalla testien ajon aikana ilmenevät poikkeustilanteet ja niistä johtuvat virheet viittaavat suoraan sinne, missä ongelma oli, ja mahdolliset korjaustoimenpiteet on helpompi kohdistaa.

Ohjelmaa varten määriteltiin kahdeksan tärkeintä ja yleisimmin käytössä olevaa kysymystyyppiä, joiden toimintavarmuus halutaan varmistaa. Näitä ovat

- NPS
- hymynaama
- valinta yhdellä valinnalla
- valinta monella valinnalla, sisältäen kaksi käyttäjän määrittelemää vaihtoehtoa sekä vakiovaihtoehdot ”ei koske minua” ja ”joku muu, mikä?”
- teksti, pieni
- teksti, iso
- valintaruutu
- matriisi.

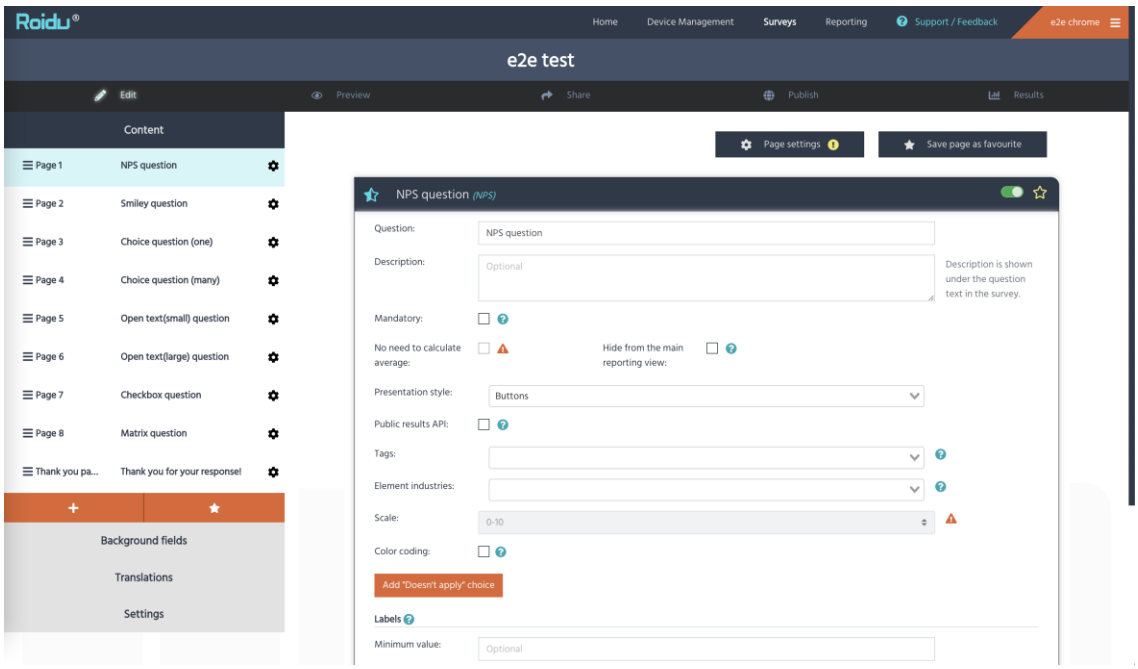
Kysymysten luomisessa esiintyi paljon samoja komentoja peräkkäin, joten oli kannattavaa tehdä apufunktioita, joissa on kyseiset komentoryppäät. Näitä apufunktioita hyödyntämällä testilohkot eivät paisu tarpeettoman isoiksi ja niitä on helpompi lukea, korjata ja jatkokehittää. Luotuja apufunktioita ovat seuraavat:

- newPageNewQuestion()

Funktio painaa nappia, jolla lisätään uusi sivu, odottaa, että uusi tyhjä sivu on näkyvissä, painaa uuden elementin lisäämis nappia ja valitsee auenneesta modaalista kysymyselementin.

- waitNewQuestionElement()
  Funktio odottaa, että kysymyselementti on lisätty kyselyyn ja auennut.
- saveElement()
  Funktio painaa elementintallennusnappia ja jää odottamaan, että elementistä poistuu ”unsaved” -luokkatunnus.
- addNextButton()
  Funktio painaa aktiivisena olevan elementin asetukset-ikonia, odottaa, että asetukset sivu latautuu, klikkaa valintaruutua, jossa määritellään, näkyykö sivulla nappia, josta pääsee kyselyssä eteenpäin. (osa kysymyselementeistä on sellaisia, että seuraavalle sivulle siirtyminen aktivoituu heti, kun vastaus on annettu, joten ”seuraava” -nappia ei tarvita.) ja painaa asetussivulla tallenna-nappia. Tämän jälkeen funktio odottaa, että sivulta poistuu latausta indikoiva pyörivä ikoni ja vaihtaa sen jälkeen näkymäksi kysymyselementin.

Kuvassa 5 esitetään, miltä näyttää kyselyn käyttöliittymä MyRoidussa. Kuvassa testiohjelma on luonut kyselyn, avannut sen ja lisännyt siihen kahdeksan kysymystä. Kuvan yläreunassa näkyvästä navigointipalkista voidaan siirtyä muille MyRoidun tarjoamille alueille. Mustalla taustalla oleva navigointipalkki on kyselykohtainen ja sillä voidaan siirtyä kaikkiin näkymiin, jotka liittyvät kyseessä olevaan kyselyyn.



KUVA 5. Käyttöliittymä.

4.2.6 Kyselyn julkaiseminen

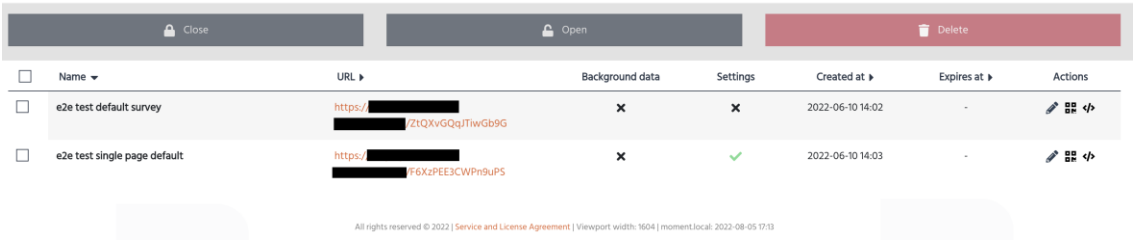
Kyselyn luotuaan ohjelma siirtyy navigointipalkin kautta julkaisemaan kyselystä kaksi eri versiota. Ensimmäinen on tavallinen monisivuinen, jota voidaan käyttää esimerkiksi tablettilaitteissa. Toinen versio on upotusrakennetta käyttävä yksisivuinen lomakemuodossa oleva kysely. Tällainen voidaan esimerkiksi upottaa asiakkaan nettisivuille.

Ensimmäistä julkaisua varten ohjelma hakee elementtikahvan ”uusi linkki” -napille, josta aukeaa uusi modaali. Tässä näkymässä ohjelma antaa vain julkaisulinkille nimen ja painaa tallenna-nappia, koska mitään erityisasetuksia tähän ei tarvita. Toista julkaisua varten ohjelma avaa modaalin uudelleen ja nimeää julkaisulinkin. Tämän jälkeen ohjelma aktivoi ”lisäasetukset” -välilehden ja klikkaa ”käytä upotuskyselyn rakennetta” -valintaruutua. Modaaliin ilmestyy pudotusvalikko, joka ohjelman tulee avata ja valita sieltä yksisivuinen lomake. Tämän jälkeen ohjelma käy painamassa modaalin tallenna-nappia.

Kyselyn julkaisulinkin luonnissa tuotetaan uniikki URL-osoite, jota käyttämällä voi vastata vain siihen kyselyyn, jolle linkki on luotu. Pääsääntöisesti kaikki vastaukset millä tahansa kyselyn julkaisulinkillä ohjautuu kyseiselle kyselylle, mutta niitä

voidaan eritellä asettamalla julkaisulinkin asetuksissa taustatietoja. Taustatietojen avulla voidaan tarkastella raportointisivulla tietylle taustatiedolle tulleita vastauksia tai koko kyselyyn tulleita vastauksia. Vastaavasti kyselyn rakenne eli ulkonäkö riippuu siitä, miten se on määritelty asetuksissa, eli lomakemuotoiseen kyselyyn ei pääse vastaamaan käyttämällä oletusrakenteen julkaisulinkkiä.

Julkaisulinkit luotuaan ohjelma poimii kuvassa 6 näkyvästä taulukosta molemmat julkaisulinkit ja tallentaa ne muuttujaan, josta niitä voidaan käyttää, kun ohjelma siirtyy vastaamaan kyselyyn. Julkaisulinkit saadaan talteen ottamalla ensin taulukon elementtikahva muuttujaan. Tämän jälkeen käydään läpi kaikki elementit silmukassa. Jokaisella kierroksella otetaan elementin teksti muuttujaan komennolla ”*const answerLink = await URLS[x].innerText()”* ja tehdään tarkastelu answerLink-muuttujalle. Jos teksti sisältää osamerkkijonon ”http”, teksti siistitään mahdollisista ylimääräisistä välilyönneistä yms. *trim()*-komennolla ja otetaan talteen muuttujaan myöhempää käyttöä varten.



KUVA 6. Julkaisulinkkien taulukko.

4.2.7 Kyselyyn vastaaminen

Kyselyyn tulee voida vastata jokaisen kysymyksen osalta, koska asetuksissa saattaa olla asetettuna etenemisehdoksi vastauksen antaminen, joten jos kysymykseen ei voi vastata, kyselyssä ei voi edetä ja se on näin ollen rikki. Projektia varten määriteltiin, että kumpaankin julkaisulinkkiin ohjelma käy antamassa viisi kertaa vastauksen. Tämä halutaan toistaa kolmella tuetulla selaimella. Playwright tukee asynkronista ohjelman ajoa jokaiselle selaimelle. Tämä tarkoittaa sitä, että jokaista testattavaa selainta kohden testiohjelma käynnistää erillisen prosessin. Prosessorin tehoja säästeltäessä jokainen selain voitaisiin testata järjestyksessä vuorotellen, mutta se lisäisi testiohjelman ajoaikaa.

Ohjelman tulee siis käydä kolmesti kymmenen kertaa kyselyyn vastaaminen läpi. Tämä luo omat haasteensa, koska siihen liittyy http-kutsujen odotus ja uuden välilehden avaamisen ja sulkemisen aiheuttamat prosessointiteho-resurssitarpeet sekä äkillinen piikki back endin kutsujen lähetyksessä ja vastaanottamisessa. Osa testeistä saattaisi epäonnistua ihan puhtaasti sen takia, että Playwrightin aikakatkaisu tulee väliin, koska http-kutsu ei mene läpi tai vastausta ei tule tarpeeksi nopeasti. Tästä tultiin siihen tulokseen, että ohjelman kannattaa käydä kyselyyn vastaamiset läpi julkaisulinkki kerrallaan edeten. Tällä tavalla kaikki testit suoritetaan siedettävän nopeasti ilman merkittävää suorituskykyongelmaa. Ohjelma siis käynnistää samanaikaisesti kaikki kolme selainta ja rupeaa etenemään jokaisessa selaimessa omaan tahtiinsa. Alla kuvataan testien etenemistä yksittäisessä selaimessa.

Ohjelma käy ensin tavallisen monisivuisen julkaisulinkin vastaamiset läpi. Julkaisulinkkejä on kaksi ja molempiin on määritelty viisi vastauskertaa. Tämän lisäksi on määritelty, että kyselyyn vastaamista testataan myös tuetulla mobiiliresoluutiolla. Ohjelmaan on kovakoodattu silmukka, joka toistuu viisi kertaa. Silmukan neljännellä kierroksella selaimen resoluutioksi asetetaan tuetun mobiiliresoluution vaakatasoresoluutio ja viidennellä kierroksella vastaava, mutta pystytasoresoluutio. Jokaisella kierroksella kyselyyn vastaaminen on samanlaista. Ainoa muuttunut elementti on kyselyn ulkonäkö. Tällä saadaan selville, karkaavatko vastausnapit ulos näkymästä. Kuvassa 7 näkyy tavallisen monisivuisen kyselyrakenteen vastaamiseen käytetty silmukka.

```
for (let x=1; x<6; x++) {
  if (x === 4) {
    platform = 'mobile landscape'
  } else if (x === 5) {
    platform = 'mobile portrait'
  }
  test(`default > open answer link [${x}/5] ${platform}`, async () => {
    await defaultOpenAnswerLink(x);
  });
  test(`default > answer page 1 [${x}/5] ${platform}`, async () => {
    await defaultAnswerPage1();
  });
  test(`default > answer page 2 [${x}/5] ${platform}`, async () => {
    await defaultAnswerPage2();
  });
  test(`default > answer page 3 [${x}/5] ${platform}`, async () => {
    await defaultAnswerPage3();
  });
  test(`default > answer page 4 [${x}/5] ${platform}`, async () => {
    await defaultAnswerPage4();
  });
  test(`default > answer page 5 [${x}/5] ${platform}`, async () => {
    await defaultAnswerPage5();
  });
  test(`default > answer page 6 [${x}/5] ${platform}`, async () => {
    await defaultAnswerPage6();
  });
  test(`default > answer page 7 [${x}/5] ${platform}`, async () => {
    await defaultAnswerPage7();
  });
  test(`default > answer page 8 [${x}/5] ${platform}`, async () => {
    await defaultAnswerPage8();
  });
  test(`default > thank you page [${x}/5] ${platform}`, async () => {
    await defaultLastPage();
  });
}
```

KUVA 7. Monisivuisen kyselyn vastaussilmukka.

Kuten kuvassa 7 ilmenee, silmukan alussa tarkastetaan, tuleeko vaihtaa resoluutiota. Indeksi-luku viedään *defaultOpenAnswerLink()*-funktiolle, ja muutos tehdään tarpeen mukaan. Silmukassa käytetyt funktiot hoitavat jokainen omaa sivuansa niitä vastaavien testilohkojen sisällä. Tämä tarkoittaa sitä, että jos jollakin sivulla vastauksen antaminen ei onnistu, siitä tulee virhe ja virheilmoitus osoittaa suoraan paikan, jossa virhe sijaitsee. Testilohkoissa on mukana myös indeksi

-muuttujan arvo ja ”platform”-muuttuja, joka kertoo, onko kyse tavallisesta vai mobiiliresoluutiosta. Indeksin arvo kertoo, monennellako kierroksella oltiin virheen sattuessa. Tämän perusteella voidaan päätellä, että jos ongelma ilmenee esimerkiksi kolmannella kierroksella, ongelma todennäköisesti ei ole itse kyselyssä. Sen sijaan, jos virheilmoituksessa indeksin arvo on 1, tarkoittaa se erittäin todennäköisesti sitä, että kysely on rikki.

Ohjelman avatessa julkaisulinkin se jää odottamaan, että kyselyn ensimmäisellä sivulla oleva Roidu-logoanimaatio katoaa ennen kuin se edes yrittää kysymyselementin hakemista. Tälle on oma testilohkonsa, jotta virheilmoituksesta saadaan suoraan selville, aukesiko kysely ollenkaan. Tämän jälkeen jokaisessa kysymyksen vastausfunktiossa odotetaan, että haluttu kysymyselementti on valittavissa. Tähän käytetään *waitForSelector()*-funktiota, jolle annetaan halutut luokkatunnukset. Sen jälkeen yritetään suorittaa vastauksen antamista joko kirjoittamalla tekstikenttään tai painamalla valintanappia.

Kysymykseen vastaamisen jälkeen on muutamaa elementtiä lukuun ottamatta oletusarvoista, että kyselyssä siirrytään eteenpäin eli seuraavalle sivulle. Poikkeuksia ovat valintaruutu- ja matriisityypin kysymykset. Näissä tulee antaa ensin valinta ja sen jälkeen painaa erikseen vielä ”seuraava” -nappia, jolloin päästään kyselyssä eteenpäin. Jokaisessa vastausfunktiossa on siispä viimeisenä komentona seuraavan sivun elementin odotuskomento. Jos testilohko epäonnistuu, se viittaa oikealle sivulle sen sijaan, että ohjelma yrittäisi jo seuraavaa testilohkoa ja ilmoittaisi virheestä siellä. Kuvassa 8 esitetään matriisityypin kysymys monisivuisessa julkaisulinkissä. Kuvassa näkyy tilanne, jossa on annettu vastaukset jokaiselle riville, mutta ”Next” -nappia ei ole painettu. Kysely ei etene ennen kuin nappia on painettu. Harmaa etenemispalkki kuvan alareunassa esittää kuinka pitkällä kyselyyn vastaamisessa kokonaisuudessa ollaan.

KUVA 8. Matriisityypin kysymys monisivuisessa kyselyssä.

Ohjelman käytyä vastaamassa viisi kertaa tavalliseen monisivuiseen julkaisulinkkiin se siirtyy vastaamaan yksisivuiseen lomakkeeseen. Koska kaikki kysymykset ovat samalla sivulla, ohjelman täytyy hakea ensin kohteena olevan kysymyselementin kahva ja sieltä hakea kohteena oleva vastausnappi. Tämä ei itsessään aiheuta sen isompaa ongelmaa, koska kysymyselementit ovat ennalta määriteltyjä. Kysymyselementin haussa voidaan käyttää valitsijalla myös suoria tekstihakuja, koska ohjelma on itse kirjoittanut kysymykset ja näin ollen on tiedossa, mitä sivulla pitäisi olla. Kuvassa 9 esitetään esimerkki tekstikysymyselementtiin vastaamisesta.

```
async function formAnswerPage5 () {
  const smallTextInputLabel = await newPage.$$(".MRS-label:has-text('Open text(small) question')");
  const smallTextInputContainer = await smallTextInputLabel[0].getProperty('parentNode');
  const smallTextInputElement = await smallTextInputContainer.getProperty('parentNode');
  const smallTextInputID = await newPage.evaluate(el => el.getAttribute('data-mrs-id'), smallTextInputElement);
  await newPage.click(`[data-mrs-id='${smallTextInputID}'] .MRS-question-input`);
  await newPage.type(`[data-mrs-id='${smallTextInputID}'] .MRS-question-input`, 'e2e test small text input (form)');
}
```

KUVA 9. Tekstikysymyselementtiin vastaaminen.

Kuvassa 9 nähdään, että elementtikahva haetaan käyttämällä tekstiä. Valitsijan parametrinä on luokkatunnus ja ”*:has-text(’Open text(small) question’)*”. Tämän

jälkeen päästään hakemaan ylin kysymyselementtitaso hakemalla kaksi kertaa ympäröivä elementti. Tätä kautta saadaan koko kysymyselementin uniikki ID-tunnus, jota voidaan käyttää valitsijassa ja näin varmistaa oikeaan kysymykseen vastaaminen.

Ohjelma ei siis oleta pääsevänsä seuraavalle sivulle vastatessaan lomakerakennetta käyttävässä kyselyssä. Jokaisessa kysymyselementissä pyritään antamaan vastaus, ja jos virhettä ei ilmaannu, edetään seuraavaan kysymyselementtiin. Kun kaikki vastaukset on annettu, siirrytään vastausten validointiin, jotta tiedetään, että kaikki vastausten antamiset ovat rekisteröityneet. Kuvassa 10 esitetään osa lomakerakennetta käyttävästä yksisivuisesta kyselystä. Asiakkaille toimitetuissa kyselyissä on huoliteltu ja räätälöity ulkoasu, mutta testiohjelma käyttää rujoa pohjaversiota, koska visuaalisesta miellyttävyydestä ei tule ohjelmalle lisäarvoa.

KUVA 10. Lomakerakennetta käyttävä yksisivuinen kysely.

4.2.8 Vastausten validointi

Julkaisulinkit on avattu uusiin välilehtiin ja suljettu vastaamisen jälkeen, joten ohjelma on edelleen julkaisulinkin-luontisivulla. Tästä siirrytään raportointisivulle erillisellä testilohkolla navigointipalkin kautta, ja samalla varmistuu navigointipalkin toiminta. Raportointisivulla näkyvät kaikki kyselyt, jotka ovat käytössä olevalle käyttäjälle tarkoitettuja. Näitä ovat käyttäjän itse luomat ja hänelle jaetut kyselyt. Kuten kyselyn poistamisessa, tässäkin näkymässä kyselyt näkyvät kortteina. Tässä vaiheessa kuitenkin on tiedossa, että tarkastettavan kyselyn nimellä löytyy vain yksi kysely, koska ohjelman alussa poistettiin kaikki tämän nimen omaavat kyselyt ja luotiin vain yksi. Valitsijalle voidaan siis antaa suora tekstihaku ”*.card-title:has-text('e2e test')*”, jolla saadaan elementtikahva oikealle kyselylle. Korttia painetaan ja jäädään odottamaan, että lataamista indikoiva pyörivä ikoni katoaa.

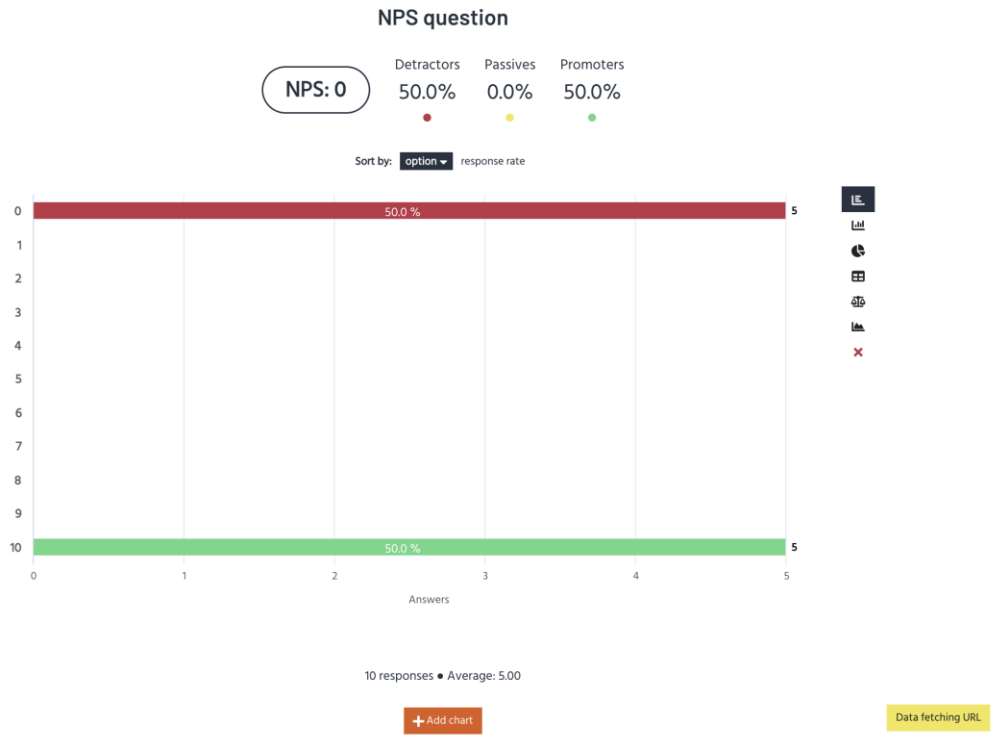
Kyselyn raportointisivulla ladataan oletusarvoisesti vain ensimmäisen kyselyn vastauksista tulokset ja niiden muodostama kuvaaja. Jotta ohjelma pääsee tarkastamaan jokaisen kysymyksen vastaukset ja niiden oikeellisuuden, tulee kaikki kuvaajat ladata ennen tarkastusta. Käyttäjän selatessa sivua alas päin sivusto lähettää automaattisesti http-kutsuja kuvaajien lataamiselle, kun kysymys tulee kohdalle. Huomioitavaa siis on se, että kaikki kuvaajat tulee ladata ennen kuin vastauksia päästään tarkastamaan. Playwrightillä on olemassa komento *waitForResponse()*, jota käyttämällä voitaisiin ensin automaattisesti selata niin alas kuin mahdollista ja odottaa, että *waitForResponse()* palauttaa halutun arvon. Testeihin käytetään kuitenkin kolmea eri selainta, ja Firefox ei tue tätä komentoa. Ratkaisu tähän on kuvan 11 mukainen.

```
test('load answers', async () => {
  for (let x=0; x<7; x++) {
    // Playwright does not support disabling caches. Routing kind of does it
    // especially Firefox had problems with waitForResponse
    await page.route('**', route => route.continue());
    await Promise.all([
      page.waitForResponse(res => res.url().includes('/results') && res.status() === 200),
      page.evaluate(() => window.scrollTo(0, document.body.scrollHeight))
    ]);
  }
});
```

KUVA 11. Kuvaajien lataaminen.

Koska oli ennalta tiedossa, että kysymyksiä oli 8 ja ensimmäinen kuvaaja latautuu automaattisesti sivulle siirryttäessä, voitiin kovakoodata silmukkaan seitsemän kierrosta. Silmukassa asetetaan uusi reititys jokaiselle kutsulle, minkä jälkeen tehdään uusi promise-kutsu selaimelle, joka tarkoittaa sitä, että ohjelmassa ei edetä ennen kuin ohjelma on saanut kutsulle vastauksen. Käytännössä kuvassa 11 näkyvä lupaus jää odottamaan vastausta, jonka URL-osoitteessa on ”/results”, ja jonka tila on koodi 200, ja rullaa sivua eteenpäin laukaisten uuden-kuvaajanlataus-kutsun. *scrollTo()*-komento rullaa arvosta nolla eli sivun yläreunasta scrollHeight-muuttujan arvoon asti. Tämä muuttuja on käytännössä selaimen ikkunan alareunan sijainnin numeerinen arvo.

Kun kaikki kuvaajat ovat ladattuina, päästään tarkastamaan kyselyn vastaanottamia vastauksia. Ohjelmassa on määritelty, mitä vastauksia annetaan ja kuinka monta kertaa, joten Playwrightistä löytyvä *expect*-funktio on erittäin hyödyllinen. Jokaiselle eri kysymyselementille on omat testauslohkonsa, joissa tätä funktiota hyödynnetään kohdistamalla se oikeaan elementtiin ja olettamalla sen tekstisisältö. Kuvassa 12 esitetään NPS-kysymystyypin vastauskuvaaja raportointisivulla ja kuvassa 13 esimerkki vastauskuvaajan käsittelystä testiohjelmassa.



KUVA 12. NPS-kysymystyypin vastauskuvaaja.

```
test('validate nps question', async () => {
  await page.waitForSelector("h2:has-text('NPS question')");
  const npsTitle = await page.$$("h2:has-text('NPS question')");
  const npsElement = await npsTitle[0].getProperty('parentNode');
  const npsElementContainer = await npsElement.getProperty('parentNode');
  const npsID = await page.evaluate(el => el.id, npsElementContainer);

  await expect(await page.locator(`[ID='${npsID}'] .numbers`).innerText()).toBe("10 responses ● Average: 5.00");
  const npsBars = await page.$$(`[ID='${npsID}'] .hc-datalabel-right`);
  await expect(npsBars.length).toBe(2);
  await expect(await page.locator(`[ID='${npsID}'] .highcharts-data-label-color-0 .hc-datalabel-right`).innerText()).toBe("5");
  await expect(await page.locator(`[ID='${npsID}'] .highcharts-data-label-color-10 .hc-datalabel-right`).innerText()).toBe("5");
});
```

KUVA 13. NPS-kysymystyypin vastauskuvaajan käsittely.

Kuvassa 13 näkyvässä NPS-kysymystyypin vastaustentarkastus-testilohkossa odotetaan ensin, että sivulta löytyy HTML-elementti ”h2”, jolla on tekstinä ”NPS question”. Jos elementti löytyy sivulta, otetaan sen elementtikahva muuttujaan käyttämällä valitsijalla samaa kohdistinta, jolla elementti löytyi. Tämän jälkeen haetaan kuvaajaelementin ylin taso hakemalla otsikkoelementin ympäröivä elementti ja vielä otsikkoelementin ympäröivä elementti. Ylimmän tason elementtiä käyttämällä saadaan selvitettyä elementin uniikki ID-arvo.

Raportointisivu käyttää highcharts-kirjastoa kuvaajien piirtämiseen. Jokaisella kuvaajalla on paljon samoja geneerisiä luokkatunnuksia, joten ID-arvon selvittäminen on tärkeää, jotta voidaan tarkastella oikeasta kuvaajasta haluttuja tekstiarvoja. Kysymystyyppien vastausvaihtoehdoilla on laskennallisesti erilaisia numeroarvoja, joita käytetään esimerkiksi keskiarvojen laskentaan. Vastausten todentaminen on käytännössä mahdotonta, jos tarkastelu tehdään väärälle elementille, koska annetut vastaukset ovat eriarvoisia ja näin ollen kuvaajilla on erilaisia keskiarvoja.

NPS-kysymystyypille ohjelma on yrittänyt antaa yhteensä 10 kertaa vastauksen, joista 5 kertaa arvolle 0 ja 5 kertaa arvolle 10. Ensimmäisenä tarkastettavana asiana todennetaan, että kuvaajan alareunassa näkyvä teksti on ”10 responses ● Average: 5.00”. Tällä varmistutaan siitä, että kysymykseen on tullut oikea määrä vastauksia ja että keskiarvon laskeminen vastauksille toimii. *Expect*-funktiolla on metodi *toBe*. Tässä tarkastuksen toteutuksessa on annettu *expect*-funktiolle valitsija, joka hakee elementin, jolla on sama ID-tunnus kuin aikaisemmin

selvitetty ID ja luokkatunnuksena ”numbers”. Tästä valitsijan löytämästä elementistä otetaan tekstisisältö käyttämällä *innerText*-metodia ja tekstisisällön odotetaan olevan ”10 responses ● Average: 5.00” käyttämällä *toBe*-metodia. Jos elementissä oleva teksti ei täsmää odotettuun tekstiin, testilohko lopettaa suorituksen ja ilmoittaa virheestä.

Seuraavaksi tarkastetaan kuvaajassa näkyvien palkkien määrä ja niiden esittämien vastausten määrä. Vastauspalkkeja pitäisi olla kaksi ja molemmissa tulisi olla viisi vastausta, jos mitään ei ole mennyt pieleen vastauksia annettaessa ja raportointisivu toimii kuten pitääkin. Jos testiohjelma on päässyt vastausten validointiin asti ja virhe ilmentyy tässä vaiheessa, voidaan olettaa, että vastausten esittämisessä on ongelmia. Testauslohko ottaa taulukkona muuttujaan kaikki elementit, jotka se löytää käyttämällä valitsijassa elementin ID-arvoa ja luokkatunnusta ”hc-datalabel-right”. Tämän jälkeen tarkastetaan, että taulukon pituus on kaksi, koska taulukossa ei pitäisi olla enempää eikä vähempää elementtejä. Viimeiseksi lohkossa testataan, että vastausvaihtoehtojen 0 ja 10 kohdalla molemmissa palkeissa on arvo 5.

Jokainen kysymystyyppi kuvaajineen on erilainen, joten yhtä ja samaa testiä ei voida hyödyntää. Yllä on esitetty NPS-kysymystyypin testaus. Esimerkiksi matriisityypin kysymyksessä testataan ensin vastausten määrä ja niiden keskiarvo. Matriisi-kysymystyypissä kuitenkin piirrettyjä vastauspalkkeja tulee olla kolme. Kuvaajalla on myös vastaussuodatin, jota käyttämällä voidaan määritellä mitä vastauksia kuvaajassa esitetään. Testilohko aktivoi suodattimia ja tarkastaa muuttuiko, kuvaaja suodatusten mukaisesti.

5 POHDINTA

Ohjelmistokehityksessä ylläpidettävyyden ja jatkokehittämisen kannalta toimintoja kannattaa pilkkoa uudelleenkäytettäviin komponentteihin, joita voi hyödyntää useassa paikassa. Yksikkötestit eivät kuitenkaan ota huomioon komponenttien ristiin käyttöä, jolloin syntyy tilanne, jossa komponentti itsessään voi toimia, mutta toisen komponentin käyttämänä ilmaantuu odottamaton häiriö, joka voi pahimmillaan estää laajemman kokonaisuuden käytön. Esimerkiksi kokonainen navigaatiopalkin tarjoama välilehti saattaa lakata toimimasta. Päästä päähän -testaus on tehokas työkalu toimintavarmuuden lisäämiseen. Sillä voidaan imitoida käyttäjän toimintaa sivustolla ja tarkastaa erilaisten elementtien näkyvyys ja toiminta niin käyttäjän päässä kuin back endin päässä.

Tämä projekti syntyi, koska haluttiin parantaa laadunvarmistusta ja tarjota käyttäjille laadukkaampaa tuotetta. Yhtenä esimerkkinä todellisesta tarpeesta on tapaus, jossa kirjaston päivityksen jälkeen yhdellä välilehdellä ei voinut enää suorittaa valintaa valintaruudussa, vaikka sivu toimi muuten täysin normaalisti. Virhe ehti olla tuotantoversiossa viikkoja, koska kyseessä oli harvoin käytetty toiminto. Vaikka kyseessä ei ollut varsinaisesti kriittinen toiminto, tämä olisi huomattu jo ennen versiopäivityksen julkaisemista, jos sen toiminta olisi testattu. Tässä projektissa katettiin ensisijaisesti MyRoidun tärkeimmät ominaisuudet ja jatkokehityksessä tullaan lisäämään testattavia kohteita. Näitä kohteita tulevat olemaan ainakin uuden käyttäjän luominen käyttöliittymästä ja vastauksista muodostetun raportin lataaminen tulokset-sivulta. Jatkoa varten tullaan myös keskustelemaan siitä, että kuinka kauan tuotantoputken automatisoidut testit saavat kestää ja sen perusteella voidaan mahdollisesti lisätä tämän projektin testausohjelmaan myös harvemmin käytettyjä toiminnallisuuksia.

Projektissa luotu ohjelma ajaa kolmella selaimella yhteensä 393 testiä kolmessa minuutissa. Jos päätettäisiin ajaa testit vain yhdellä selaimella, suoritettuja testejä olisi silloin 131 kappaletta ja niiden ajoaika olisi kaksi minuuttia. Minuutin lisäinvestoinnilla saadaan siis huomattavasti laajempi kattavuus. Bonuksena tätä projektia päästiin hyödyntämään jo ennen kuin se oli valmis. Eräällä työntekijällä oli työtehtävänä kyselyn luomiseen liittyvän koodin refaktorointi samaan aikaan, kun

projektissa oli valmiina MyRoiduun kirjautuminen ja uuden kyselyn luominen. Refaktoroinnin aikana hän pystyi automaattisesti tarkastamaan, toimiiko kaikki niin kuin pitääkin.

Projekti eteni suunnitelmien mukaisesti vaihe vaiheelta pysyen sille annetussa aikataulussa. Isoin ongelma projektin aikana oli vastausten validointi osuudessa kuvaajien lataaminen. Osuuden ensimmäinen testilohko vierittää kuvaruutua alaspäin laukaisten jokaisen kysymyksen kohdalla pyynnön kuvaajan lataamiselle. Palautuneet vastaukset back endiltä pitää pystyä erottelemaan toisistaan, koska jos yksikin vastaus ilmoittaisi virheestä, vastauksia ei voisi tarkastaa. Välimuistin tyhjentäminen olisi ollut hyvä ratkaisu, mutta kolmesta käytetystä selaimesta Firefox ei tue kunnolla siihen tarkoitettua komentoa. Tähän löytyi kuitenkin ratkaisu vastauksen reitityksestä.

Kaiken kaikkiaan projektissa toteutui vaatimukset täyttävä, päästä päähän -testausta hyödyntävä ohjelma, joka liitettiin osaksi MyRoidun jatkuvan julkaisemisen tuotantoputkea. MyRoidun tärkeimmät ja keskeisimmät toiminnallisuudet testataan päästä päähän. Esimerkkinä sisäänkirjautuminen. Käyttäjä pystyy syöttämään käyttäjätunnuksensa käyttöliittymässä. Tämän jälkeen käyttäjätunnusten olemassaolo tietokannassa tarkastetaan ja käyttäjä päästetään MyRoiduun sisälle, jos tunnukset olivat oikein eli sisäänkirjautuminen toimii. Ohjelmaa käyttämällä varmistutaan siitä, että Roidun asiakkaille tarjotaan aina mahdollisimman laadukasta tuotetta.

LÄHTEET

Checkly. 2022. Basics playwright intro. Verkkosivu. Viitattu 13.8.2022. https://www.checklyhq.com/learn/headless/basics-playwright-intro/

Connolly, T., Begg, C. 2002. DATABASE SYSTEMS. A Practical Approach to Design, Implementation, and Management. 3. painos. Boston: Addison-Wesley.

Gillis, S. 2018. end-to-end testing. TechTarget. Verkkosivu. Viitattu 13.8.2022. https://www.techtarget.com/searchsoftwarequality/definition/End-to-end-testing#:~:text=End%2Dto%2Dend%20testing%20is,optimally%20under%20real%2Dworld%20scenarios

How CAPTCHAs work | What does CAPTCHA mean? 2022. Cloudflare. Verkkosivu. Viitattu 13.8.2022. https://www.cloudflare.com/learning/bots/how-captchas-work/

Huotarinen, J. 2016. Tiesitkö jatkuvan julkaisun olevan jo arkipäivää? GOFORE. Verkkosivu. Viitattu 13.8.2022. https://gofore.com/tiesitko-jatkuvan-julkaisun-olevan-jo-arkipaivaa/#:~:text=Jatkuva%20julkaisu%20(eng.,palveluun%20tapahtuvat%20katkottomasti%20ja%20nopeasti

Laster, B. 2017. Continuous Integration vs. Continuous Delivery vs. Continuous Deployment. E-kirja. Sebastopol: O'Reilly Media. Viitattu 23.8.2022. Vaatii käyttöoikeuden. https://learning.oreilly.com/library/view/continuous-integration-vs/9781492028918/

Playwright enables reliable end-to-end testing for modern web apps. 2022. Microsoft. Verkkosivu. Viitattu 13.8.2022. https://playwright.dev/

Safe. 2021. Continuous Delivery Pipeline. Verkkosivu. Viitattu 24.8.2022. https://www.scaledagileframework.com/continuous-delivery-pipeline/

Tozzi, C. 2021. Top 5 JavaScript Test Automation Frameworks in 2021. SAUCELABS. Verkkosivu. Viitattu 13.8.2022. https://saucelabs.com/blog/top-5-javascript-test-automation-frameworks-in-2021