jamk

Automaatiotestauksen kehittäminen selainpohjaisessa sovelluksessa

Sami Luoma-aho

Opinnäytetyö, AMK
Elokuu 2022
Tietojenkäsittely ja tietoliikenne
Tieto- ja viestintätekniikan tutkinto-ohjelma

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Kuvailulehti

Luoma-aho, Sami

Automaatiotestauksen kehittäminen selainpohjaisessa sovelluksessa

Jyväskylä: Jyväskylän ammattikorkeakoulu. Elokuu 2022, 44 sivua

Tietojenkäsittely ja tietoliikenne. Tieto- ja viestintätekniikan tutkinto-ohjelma. Opinnäytetyö AMK.

Julkaisun kieli: suomi

Julkaisulupa avoimessa verkossa: Kyllä

Tiivistelmä

Opinnäytetyön toimeksiantajana toimi Pinja Digital Oy, joka on osa Pinja-konsernia. Pinjan yksi tuotteista Once by Pinja on toimitusketjun hallintajärjestelmä, joka on käytössä muun muassa energiantuotantolaitoksissa ja kiertotalousalan yrityksissä. Toimeksiantajalle on tarve automaatiotestauksen kehittämiselle kyseisen järjestelmän Oncenet-sovellukseen. Automaatiotestauksella pyritään osaltaan vaikuttamaan tuotteen laatuun.

Opinnäytetyön kehittämistehtävän tavoitteena oli luoda pohja Oncenet-sovelluksen automaatiotestausta varten Cypress-kirjastoa käyttäen sekä rakentaa keskeisimmät automaatiotestitapaukset regressiotestauksen tarpeisiin. Samalla tuli selvittää automaatiotestitapausten ylläpidettävyyteen vaikuttavia seikkoja ja sitä millaiset regressiotestitapaukset soveltuvat automaatiotestaukseen. Kehittämistehtävän aineistona toimivat kirjallisuuslähteet sekä verkkolähteet.

Kehittämistehtävä toteutettiin luomalla ensin testausympäristö toimeksiantajan virtualisointi- ja paikallisympäristöön. Tutkimuslähteitä apuna käyttäen luotiin pohja Cypress-automaatiotestaukselle ja tämän jälkeen automaatiotestejä regressiotestausta varten. Yksittäisten automaatiotestien riippumattomuus, yksinkertaisuus ja resilienssi muutoksille oli tärkeää testien ylläpidettävyyden kannalta. Lisäksi selvitettiin, mikä resilienssiin vaikuttaa. Sen sijaan yksiselitteistä vastausta siihen, millaiset regressiotestit tulisi automatisoida, ei löydetty.

Automaatiotestauksen kehittäminen selainpohjaiseen sovellukseen ei teknisesti ole vaativaa. Haastavampaa on luoda automaatiotestejä, joita myös vuosien päästä voidaan käyttää ja jotka ovat riittävän yksinkertaisia ylläpitää. Haastavaa on myös valita oikeat automatisoitavat testit ja luoda kokonaiskuva testauksesta, sisältäen testauksen eri tyypit ja tasot, jotta testauksen tavoitteissa päästään suunnitellulle tasolle.

Avainsanat (asiasanat)

Testaus, SaaS, testiautomaatio, regressiotestaus, Cypress

Muut tiedot (salassa pidettävät liitteet)

jamk

Description

Luoma-aho, Sami

Test automation development in browser-based application

Jyväskylä: JAMK University of Applied Sciences, August 2022, 44 pages.

Information and communication technologies. Degree Programme in Information and Communication Technology. Bachelor´s thesis.

Permission for open access publication: Yes

Language of publication: Finnish

Abstract

The thesis was assigned by Pinja Digital Oy which is part of Pinja concern. Once by Pinja - one of the products of Pinja is supply chain management system which is used by energy and circular economy companies. Pinja has need for automation testing development in Oncenet application which is part of the product in question. Automation testing is intended to enhance the quality efficiently.

The object of the development research was to create a base for automation testing in Oncenet application using Cypress testing tool and to build basic automation test cases for regression test purposes. In addition purpose was to search matters affecting sustainability of the automation tests and find out which kind of regression test cases are suitable for automation. The material of the development research formed from literature and network sources.

Development research was executed by creating first test environment in virtual and local environment. The base for Cypress-testing as well as automation tests for regression testing were created with help of research material. Independence, simplicity, and resilience for changes of the automation tests was important for sustainability of the tests. Also, the actors affecting to resilience was researched. However unambiguous answer for question what kind of regression tests should be automated was not found.

The development of the automation testing in browser-based application is not technically demanding. More challenging is to create automation tests which are usable years after, and which are simple enough to maintain. Challenging is also to choose right test for automation and create general view of the testing including different test types and levels so that planned testing goals are reached.

Keywords/tags (subjects)

Testing, SaaS, test automation, regression testing, Cypress

Miscellaneous (Confidential information)

Sisältö

Kuviot

Käsitteet

API

Application Programming Interface, ohjelmointirajapinta, jonka avulla eri ohjelmat voivat vaihtaa tietoja keskenään.

Assertointi

Testin toteuttavan tarkistuksen todentaminen.

Backend

Nimitys selainpohjaisen sovelluksen tai verkkosivuston palvelinpuolen ohjelmistosta.

CI/CD

Continuous integration, continuous delivery eli jatkuva integrointi ja julkaisu on toimintatapa, jossa ohjelmisto koostetaan, testataan ja julkaistaan automatisoidusti.

DOM

Document Object Model, dokumenttioliomalli on tapa kuvata esimerkiksi HTML dokumentin rakenne puumuodossa. Oliomallin olioita voidaan hakea ja muokata esimerkiksi Javascript-ohjelmointikielellä.

Frontend

Nimitys selainpohjaisen sovelluksen tai verkkosivuston selainpuolen ohjelmistosta.

JSON

JavaScript Object Notation on tiedonvälityksessä käytettävä tiedostomuoto.

On-premise

On-premise ohjelmisto on ohjelmisto, joka asennetaan ja jota suoritetaan käyttäjän koneella (vrt. SaaS).

SaaS

Software as a Service on ohjelmiston jakelumalli, jossa ohjelmisto tarjotaan loppukäyttäjälle internetin välityksellä. Käyttäjän koneelle ei tarvitse asentaa ohjelmistoa.

Testitynkä

Testitynkä (engl. mock tai stub.) on eräänlainen sijaiskomponentti, jota käytetään testauksessa silloin kun oikea komponentti ei ole vielä käytössä tai sitä ei jostain muusta syystä voida käyttää.

1 Johdanto

Ohjelmistotuotannossa on helppo tehdä virheitä. Ohjelmiston kehitystavasta riippuen, virheet voidaan huomata tai ne voivat päästä tuotantoon saakka, jossa asiakkaat törmäävät niihin. Riippuen asiakkaasta ja ohjelmiston kriittisyydestä, taloudellisesti tai muista syistä, virheen pääseminen tuotantoon voi olla kirjaimellisesti katastrofi tai vain epämukavuutta käyttäjälle. Vaikka täydelliseen virheettömyyteen ei ohjelmistoissakaan päästä, ohjelmistotestauksella pyritään vaikuttamaan siihen, ettei ohjelmiston tuotantoversioon saakka päädy hyväksyttävää enemmän virheitä.

1.1 Pinja Digital Oy

Opinnäytetyön toimeksiantajana on Pinja Digital Oy. Yritys kuuluu Pinja konserniin, joka työllistää Suomessa noin 500 henkilöä ja sen liikevaihto vuonna 2021 oli 42,7 miljoonaa euroa (Pinja, n.d.). Pinja, aiemmalta nimeltään Protacon Oy on perustettu vuonna 1990 (Protacon on nyt Pinja – teollisuuden uudistamisella ja digitalisaatiolla vahvaa kasvua, 2020).

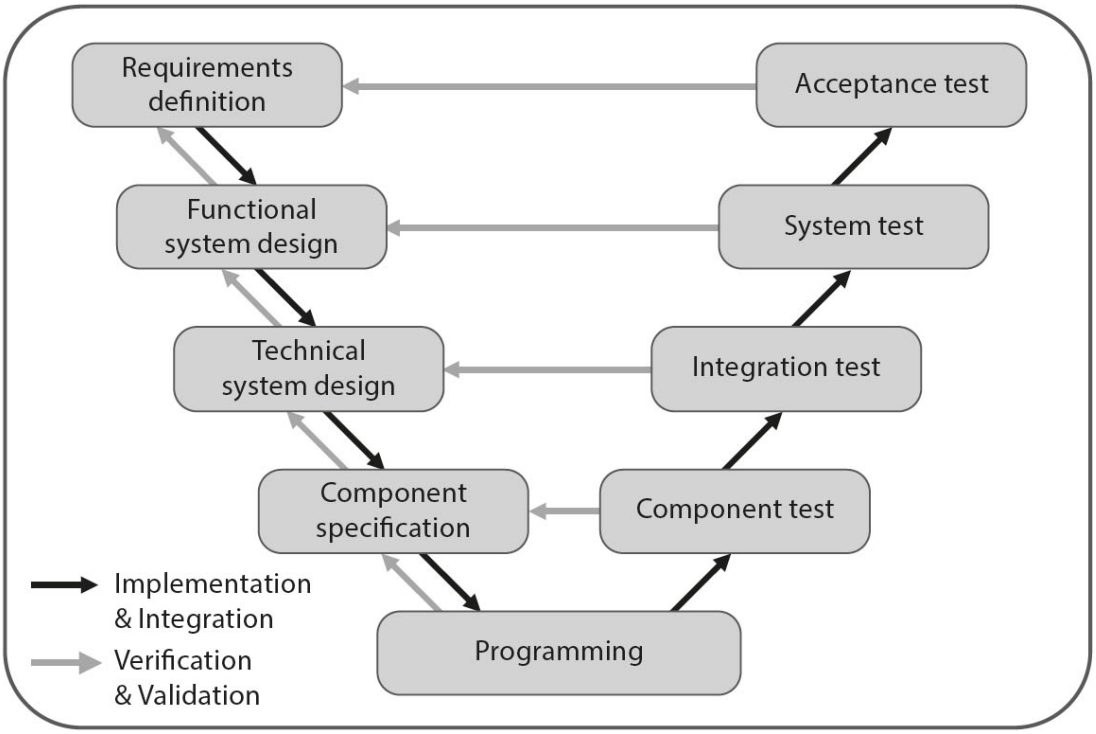
Työssäni paneudun Pinja Digital Oy:n selainpohjaisen Oncenet2-sovelluksen automaatiotestaukseen. Oncenet on osa Once by Pinja tuotetta, toimitusketjun hallintajärjestelmää, jota käytetään muun muassa energiantuotantolaitoksissa ja kiertotalousalan yrityksissä (Once by Pinja, n.d.). Järjestelmän käyttäjille kyse on laajoista tilaus- ja toimitusketjuista, joten materiaalivirtojen hallinta on niissä tavallisesti tärkeää, jopa kriittistä. Tämän vuoksi myös virheiden välttäminen ohjelmistossa on tärkeää. Työkaluna automaatiotestauksessa käytetään Cypress-testaustyökalua, jonka käyttöön toimeksiantaja on päätynyt. Aihe on kiinnostava henkilökohtaisesti ohjelmistokehittäjän näkökulmasta pelkästään jo testausautomaation yleistymisen myötä. On myös kiinnostavaa perehtyä testauksen teoreettiseen viitekehykseen hieman enemmän kehittääkseni omaa laatuajatteluani.

2 Ohjelmistotestaus

Erilaiset testauksen termit sekoittuvat helposti keskenään. Seuraavaksi käymme läpi testauksen tasoja, muutamia tärkeimpiä testausmenetelmiä ja testityyppejä. Lopuksi selvitämme mitä regressiotestauksella tarkoitetaan. Tasot kertovat enemmän testauksen laajuudesta ja kohteesta ja menetelmät eri tavoista testata.

2.1 Testauksen tasot

Järjestelmien kehityksessä laajalti tunnettu, alun perin saksalainen V-malli tunnetaan hyvin myös ohjelmistotestauksessa. Malli on kehittyneempi versio ns. vesiputousmallista ja se demonstroi hyvin ohjelmistokehityksen eri vaiheiden ja testauksen tasojen yhteyttä. (SDLC V-Model, n.d.) Hieman tekijästä riippuen esitettävä V-malli on joko kolme- tai neljätasoinen, joskus jopa viisitasoinen. Sekä Spillner & Linz (2021) että Kasurinen J. (2013, 51) esittävät V-mallin neljätasoisena (Kuvio 1). Kuviossa projektin etenemisen katsotaan kulkevan vasemmalta oikealle ja vasemmalla olevassa ”V:n haarassa” käydään läpi ohjelmistokehityksen vaiheita. Oikeanpuoleisessa haarassa on kuvattuna testauksen ohjelmistokehitykseen nivoutuvat tasot.



Kuvio 1. Ohjelmistotestauksen V-malli (Spillner & Linz 2021).

Testauksen tasoja ovat Kasurisen (2013, 51) mukaan yksikkötestaus, integraatiotestaus, systeemitestaus sekä hyväksymistestaus, jossa testaus tehdään jo kohdeympäristössä tai sitä simuloiden. Testit kattavat laajemman osan järjestelmästä kuvion tasoissa oikealle päin mentäessä.

Jokainen testauksen taso voi sisältää validointitestejä ja verifiointitestejä. Verifioinnissa varmistetaan, täyttääkö testattava järjestelmä vaatimusmäärittelyt kokonaan ja oikein. Siinä haetaan vastausta kysymykseen, rakennetaanko järjestelmä oikein. Validoinnissa toisaalta pyritään varmistamaan, että testattava järjestelmä toimii sille tehdyssä tarkoituksessa ja ratkaisee ongelman, jonka sen on suunniteltu tekevän. Toisin sanoen se vastaa kysymykseen, rakennetaanko oikeaa järjestelmää. (Spillner & Linz 2021.)

Vaikka testauksen jokainen taso voi sisältää molempia testejä, käytännössä jokainen testi testaa molempia näkökulmia. Painopiste kuitenkin muuttuu siten, että yksikkötestaus on pääasiassa keskittynyt verifiointiin ja hyväksyntätestaus validointiin. (Spillner & Linz 2021.)

Tasoilla olevat testit jakautuvat vielä kahteen eri perustyyppiin funktionaalisiin ja ei-funktionaalisiin testaustyyppeihin. Funktionaaliseen perustyyppiin kuuluvat testit, joissa testataan ohjelmiston toiminnallisia ominaisuuksia verraten niitä vaatimusmäärittelyyn. Ei-funktionaalisen perustyypin testeissä testataan esimerkiksi ohjelmiston suorituskykyä, luotettavuutta, skaalautuvuutta tai muita ei-toiminnallisia ominaisuuksia. (Hamilton 2022d.)

2.1.1 Yksikkötestaus

Yksikkötestaus on kenties ohjelmistokehittäjälle tutuin, koska juuri ohjelmistokehittäjä on usein testin kirjoittaja ja suorittaja. Yksikkötestissä, toiselta nimitykseltään moduulitestissä (Juvonen 2018, 27), testataan yhden moduulin, olion tai funktion toimintaa toteutuksen yhteydessä. Sen tarkoituksena on varmistaa, että toteutettu toiminto toimii ennalta oletetun mukaisesti, reagoi syötteisiin oikein ja kääntyy virheittä. (Kasurinen 2013, 51.)

Koska yksikkötestauksessa testataan vain yhtä komponenttia, ongelmana on usein, ettei yksi komponentti pysty toimimaan itsenäisesti. Tällöin on simuloitava muiden komponenttien toimintaa ja järjestelmän sisäistä liikennettä rakentamalla testikomponentteja, ns. testitynkiä (mock objects,

test studs). Testikomponenttien etuna on erilaisten virhetilanteiden helppo simulointi ja suurten järjestelmien tapauksessa, simulaation keveys verrattuna todelliseen. (Kasurinen 2013, 52.)

2.1.2 Intergraatiotestaus

Seuraavana testaustasona tulee integraatiotestaus. Siinä järjestelmän eri osia, joita yksikkötestauksessa testattiin erillisinä, aletaan sovittaa yhteen. Lopulta tarkoituksena on saada koko järjestelmä toimimaan kokonaisuutena. Integraatiotestauksen periaatteena on liittää yksi komponentti osaksi aiemmin testattua kokonaisuutta. Ei siis vielä testata koko järjestelmää, mutta selkeästi suurempia osia kuin yksikkötestauksessa. Myös integraatiotestauksessa käytetään testitynkiä, kunnes järjestelmän kaikki osat ovat lopulta toimivia. (Kasurinen 2013, 54.)

Integraatiotestausta käytetään vähentämään kokonaisen ohjelman testaustarvetta ja testien määrää. Samalla myös testaustehokkuus paranee ja testattaessa vain osaa koko järjestelmästä, virheiden löytyessä, niiden paikantaminenkin on helpompaa. Integraatiotestauksen tekeminen voi olla hankalaa strukturoimattomalle koodille, jossa ei ole erotettavissa testattavia osia. Useimmiten myös käyttöliittymät ovat vaikeita integraatiotestauksen kohteita. (Tevanlinna 2004, 7–10.)

2.1.3 Järjestelmätestaus

Järjestelmätestauksessa testataan nimensä mukaisesti kokonaista järjestelmää. Testauksen V-mallin kolmannen tason testaus laajentaa edelleen testauksen koodipohjaa integraatiotestauksesta. Integraatiotestauksessa tuotiin testaukseen mukaan uusia komponentteja ja osa korvattiin testitynglillä. Järjestelmätestauksessa ei ole mukana testitynkiä. Se ei myöskään rajoitu mihinkään erityiseen testaustapaan, vaan on yleisnimitys kokonaisen järjestelmän testaukselle. Testaustapa riippuukin paljolti projektista. Se voi toisaalta olla vaikkapa musta laatikko-, lasilaatikko-, käyttäjä- tai kuormitustestausta. Järjestelmätestauksessa testausympäristönä on testiympäristö. (Kasurinen 2013, 56–57.)

2.1.4 End-to-End -testaus

Niin sanottu End-to-end testaus (päästä päähän -testaus) ja järjestelmätestaus kulkevat käsi kädessä, mutta ne myös sekoitetaan keskenään helposti. End-to-end testaus kuvaa testauksen prosessin toimintatapaa ja keskittyy testausprosessin kulkuun. Siinä validoidaan ohjelmiston kulku

alusta loppuun, mukaan lukien rajapinnat ja siihen liitetyt järjestelmät, kun taas järjestelmätestauksessa ainoastaan testauksen kohteena oleva järjestelmä testataan (pp_pankaj 2019). End-to-end testauksessa tarkoituksena on siis testata koko ohjelmisto, riippuvuudet, datan yhtenäisyys ja ohjelmiston kommunikointi muiden järjestelmien, rajapintojen ja tietokantojen kanssa. End-to-end testauksessa käytetään tuotannon kaltaista dataa ja testausympäristöä, jotta päästään mahdollisimman lähelle tuotantoympäristöä. Käytännössä tämä voi tarkoittaa esimerkiksi erillisen testitietokannan käyttöä. (Hamilton 2022c.)

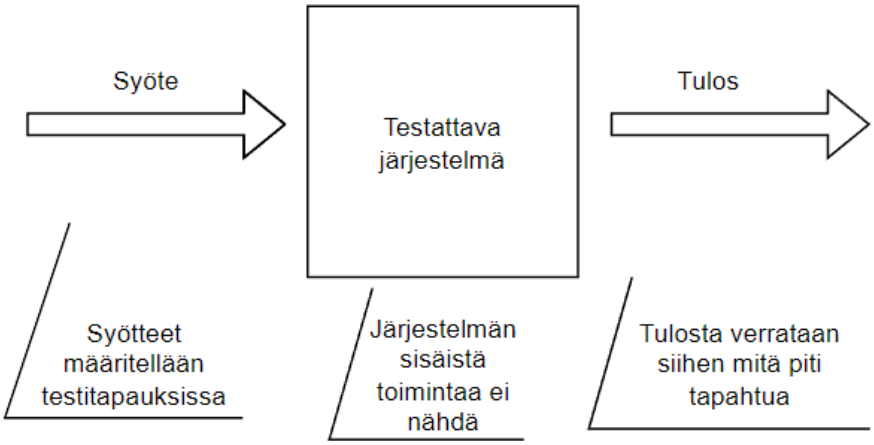
2.1.5 Hyväksyntätestaus

V-mallin viimeisen vaiheen, hyväksymistestauksen tavoitteena on osoittaa ohjelmiston vaatimuksenmukaisuus ja riittävän korkea laatu. Yhdessä järjestelmätestauksen kanssa hyväksyntätestaus antaa kuvan, voidaanko ohjelmisto julkaista loppukäyttäjälle (Pezzè & Young 2008, 417–418). Hyväksymistestaus on viimeinen testauksen työvaihe ja siinä ohjelmisto hyväksytään virallisesti. Mikäli aiemmat testausvaiheet on suoritettu asianmukaisesti, merkittäviä virheitä ei tulisi enää ilmaantua. Usein hyväksymistestaus tehdään muista testausvaiheista poiketen kohdeympäristössä tai vähintäänkin hyvin identtisessä ympäristössä. Ohjelmistoa ei tässä vaiheessa kuitenkaan ole vielä julkaistu ja seuraavaksi siirrytäänkin käyttöönottovaiheeseen. Tässä vaiheessa tehdään kevyt käyttöönottotestaus, jossa varmistetaan asetusten oikeellisuus ja eri käyttäjätasojen toiminta. Käyttöönottotestaus voi projektista riippuen sisältyä myös hyväksymistestaukseen. (Kasurinen 2013, 57; Juvonen 2018, 27–28.)

2.2 Testausmenetelmiä ja -tekniikoita

2.2.1 Mustalaatikkotestaus

Mustalaatikkotestauksessa (black box testing) ei olla kiinnostuneita järjestelmän toiminnasta kooditasolla. Järjestelmä nähdään nimensä mukaisesti mustana laatikkona, jolle annetaan testisyötteitä ja josta saadaan tuloksia (Kuvio 2). Mikäli tulokset vastaavat siihen, mitä järjestelmän tulisikin kyseisistä syötteistä tuottaa, voidaan ”mustan laatikon” katsoa toimivan niiltä osin oikein. Musta laatikko testaus rakentuu täysin järjestelmälle asetetuille määrityksille ja vaatimuksille (Hamilton 2022a). Menetelmä on perinteisin ja hyvin yksinkertainen testausmenetelmä. Sitä voidaan käyttää kaikissa testauksen vaiheissa, joissa on käytettävissä toimiva versio ohjelmasta. (Kasurinen 2013, 65–66; Juvonen 2018, 29.)

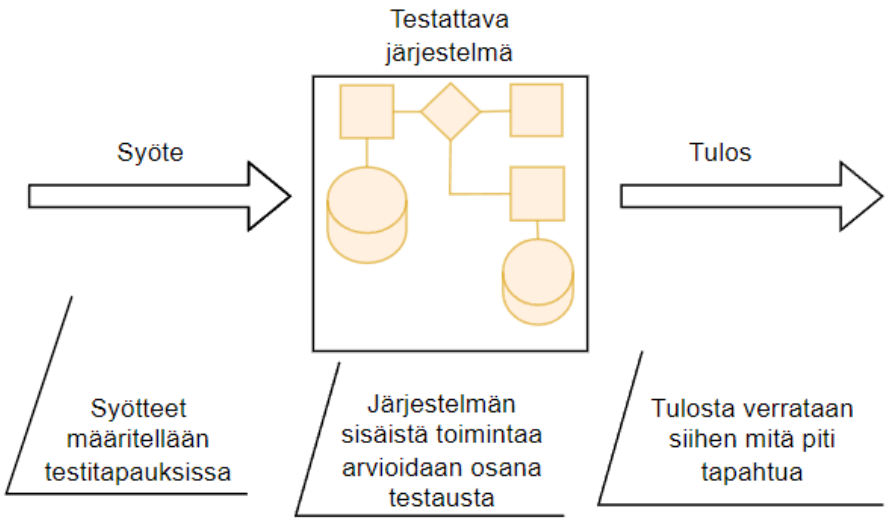


Kuvio 2. Mustalaatikkotestaus (Kasurinen 2013, 66, muokattu).

Syötteet ja tehtävät määritellään testitapauksissa. Tapauksia voivat olla esimerkiksi testit tiedostojen tallentamisesta, lomakkeiden täyttämisestä, eri sivujen avaaminen ja järjestelmän tuottaman datan arvojen tarkastaminen. Lisäksi tärkeää on yleensä luoda tapauksia, joissa testataan järjestelmän toimintaa eri käyttäjätasoilla. Testitapauksissa voidaan myös määritellä testejä, joissa testataan kuinka järjestelmä reagoi käyttäjän virheellisiin syötteisiin. Voidaan käyttää esimerkiksi juuri testaustarkoitukseen listaa hankalista käyttäjäsyötteistä (Big List of Naughty Strings 2021). Musta laatikko testauksessa onkin tärkeää testata monipuolisesti erilaisilla syötteillä, jotta kaikki mahdolliset syötteiden kombinaatiot tulevat testattua. Mikäli jokin syötteiden yhdistelmä tuottaa odottamattoman lopputuloksen, testaaja tekee asiasta ilmoituksen tai kirjaa poikkeaman testausraporttiin. (Kasurinen 2013, 66.)

2.2.2 Lasilaatikkotestaus

Lasilaatikkotestaus (tunnetaan myös nimillä white box testing, glass box testing ja clear box testing) täydentää edellä kuvattua musta laatikkotestausta. Lasilaatikkotestauksessa testaaja on kiinnostunut myös järjestelmän toiminnasta testauksen aikana (Kuvio 3) ja on selvillä siitä, mitä syötteelle järjestelmässä tapahtuu ennen tuloksen muodostumista. Virheen sattuessa testaaja osaa kooditasolla kertoa, missä virhe tapahtui. (Kasurinen 2013, 67–68.)



Kuvio 3. Lasilaatikkotestaus (Kasurinen 2013, 67, muokattu).

Lasilaatikkotestit ovatkin syvällisempiä mutta toisaalta asettavat testaajalle osaamisvaatimuksia. Testaajan tulee tuntea järjestelmä kooditasolla ja hänen tulee tuntea järjestelmän logiikka riittävän hyvin, jotta osaa arvioida toimiiko järjestelmä oikein. Tämä vaatii asiantuntemusta ohjelmoinnista, eikä välttämättä sovellu kokemattomalle testaajalle. Lasilaatikkotestauksessa ei myöskään havaita puutteita järjestelmän ominaisuuksissa tai vaatimusmäärittelyssä, joten menetelmää on hyvä täydentää muilla menetelmillä. Hyvänä puolena lasilaatikkotestauksessa on, että testaajat oppivat pakostakin tuntemaan järjestelmän ja saattavat havaita ongelmia, joita musta laatikko testauksessa ei havaittaisi. (Kasurinen 2013, 68.)

2.2.3 Savutestaus

Savutestauksesta (smoke testing) puhuttaessa kyseessä on tavallisesti ohjelmiston perustoiminnallisuuden varmistaminen. Se voidaan tehdä hyvin kevyellä testauskokoonpanolla ja yleensä se tehdään ensimmäisenä ennen varsinaista testausta. Tämä sen vuoksi, että varmistutaan järjestelmän perustoimivuudesta ennen testiympäristön pystytystä ja perusteellisempaa testausta. Savutestaus voidaan tehdä myös tuotannossa olevan järjestelmän päivityksen jälkeen. (Juvonen 2018, 30.)

2.2.4 Tutkiva testaus

Kaikissa testaustekniikoissa testitapauksia ei suunnitella etukäteen. Tutkiva testaus (exloratory testing) on testaamisen muoto, jolla pyritään täydentämään testaussuunnitelmaa ja muiden testausmenetelmien ja dokumentaation jättämiä aukkoja. Menetelmä pyrkii hyödyntämään ihmisten ymmärrystä siitä, missä virheitä usein esiintyy ja löytämää niitä. Dokumentaatiota voidaan tehdä ja testaus voi myös perustua olemassa oleviin malleihin, mutta se ei ole pakollista. (Kasurinen 2013, 74.)

Pyhäjärven (Pyhäjärvi 2018.) mukaan tutkiva testaus on testausta, joka keskittyy testaussuunnitelman ja testin suorittamisen jatkuvaan oppimiseen edellisessä testissä opitun pohjalta. Testaaja oppii ja arvioi järjestelmää suorittaessaan testiä. Tutkivassa testauksessa olennaista testien suorittajalle on aktiivinen ajattelu ja tutkivan testauksen keskiössä onkin itse testaaja, joka tekee työn ja oppii aktiivisesti parempia tapoja testata. (Pyhäjärvi 2018.)

2.3 Regressiotestaus

Regressiotestaus (regression testing) ei varsinaisesti ole oma erillinen menetelmänsä, vaan yleistermi, jolla kuvataan järjestelmän uudelleentestaamista. Se ei myöskään sijoitu millekään yksittäiselle testauksen tasolle, kuten yksikkötestaukseen tai integraatiotestaukseen. Kun järjestelmää kehitetään – siihen lisätään ominaisuuksia tai sen toiminnallisuuksia muutetaan, siitä poistetaan virheitä tai se siirretään uudelle alustalle, halutaan sen muiden toimintojen säilyvän ennallaan, muuttumattomana (nonregression). Regressiotestauksen tärkein tavoite varmentaa järjestelmän ja/tai sen osan toimivuus siihen tehdyn muutoksen jälkeen. Toisaalta regressiotestauksesta puhutaan myös, kun järjestelmästä halutaan varmentaa kaikkien toimintojen toimivuus esimerkiksi kehityshaaran yhdistämisen jälkeen. Regressiotestauksessa tehdään samoja testejä muutosten jälkeen, mikä täyttää yhden automaatiotestauksen vaatimuksista. Sen vuoksi automaatiotestien suorittaminen regressiotestauksessa on usein perusteltua. (Kasurinen 2013, 68–70.)

Järjestelmä-, hyväksyntä- ja regressiotestaus kaikki testaavat järjestelmää kokonaisena, niiden päämäärä vain on eri. Järjestelmätestaus (system testing) testaa järjestelmää verraten sitä järjestelmän vaatimusmäärittelyyn. Hyväksyntätestaus (acceptance testing) ei niinkään välitä järjestelmän määrittelyistä vaan siitä, toimiiko järjestelmä käyttäjän näkökulmasta ja regressiotestauksen

tehtävä, kuten jo aiemmin todettiin, on testata järjestelmän muuttumattomuutta. (Pezzè & Young 2008, 417–418.)

Ihanteellisessa tapauksessa muutoksen jälkeen koko järjestelmä testattaisiin uudelleen regression varalta, mutta koska se on harvoin järkevää kustannusten tai muiden resurssien rajallisuuden vuoksi, on tyydyttävä testaamaan järjestelmä vain osittain. On tehtävä kompromissi kulujen ja liiketoimintariskin välillä. Spillner & Linz (2021) mainitsee strategioita testitapausten valitsemiseksi tällaisissa tapauksissa:

- Tehdään vain korkean prioriteetin saaneet testit.
- Jätetään pois erityistapaukset
- Rajoitetaan testaus tiettyyn kokoonpanoon ja konfiguraatioon, esimerkiksi vain yksi kieliversio tai käyttöjärjestelmä.
- Rajataan testaus tiettyihin komponentteihin tai testitasoihin.

Haasteita regressiotestaukselle asettavat kompleksiset muutokset. Muutokset uudessa ohjelmistoversiossa voivat aiheuttaa muutoksia järjestelmän syötteissä tai tuloksissa. Tämä puolestaan voi johtaa siihen, että osa testeistä ei enää toimi odotetusti ja testiä ei läpäistä hyväksytysti tai testi rikkoutuu kokonaan. Ohjelmistomuutosten myötä osa testeistä voi ”vanhentua” jos niiden testaamat ominaisuudet poistetaan. Toisaalta eri syistä luodut tai muutetut testitapaukset voivat muuttua keskenään tarpeettomiksi, jos ne testaavat samaa ominaisuutta joko kokonaan tai osittain. Osaan edellä mainituista haasteista voidaan vastata laadukkailla, ylläpidettävillä testisarjoilla ja erityisesti hyvällä testidokumentaatiolla. (Pezzè & Young 2008, 427–428.)

3 Automaatiotestaus

Automaatiotestaus on testaustoiminnan muoto, jossa erillisiä automaatiotyövälineitä käytetään järjestelmän testaamiseen. Automaatiotestauksella voidaan vapauttaa testaajien työaikaa automatisoimalla toistuvien testien suorittaminen tarkoitusta varten rakennetulla automaatiolla. Tarkoitus ei ole kokonaan korvata manuaalitestausta. (Kasurinen 2013, 76.)

Automaatiotestauksella voidaan lisätä testauksen tehokkuutta ja testien kattavuutta sekä suoritusnopeutta. Manuaalitestausvaihtoehto voi olla joissain tapauksissa, varsinkin testin toistuessa usein, pitkästyttävää. Tästä voidaankin johtaa vastauksia kysymykseen, millaisia testejä sitten tulisi automatisoida? Hamilton (2022b) listaa neljä eri kriteeriä testitapauksille, joita tulisi ensisijaisesti automatisoida:

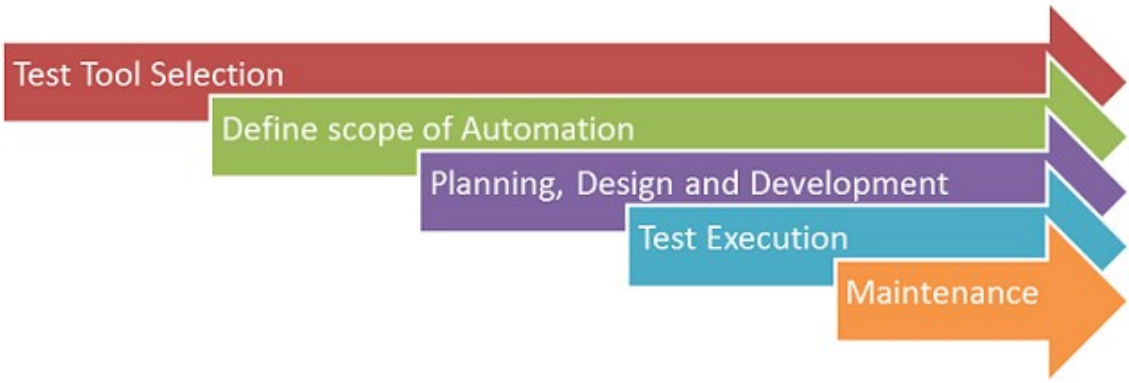
1. Korkean riskin, liiketoimintakriittiset testitapaukset
2. Testitapaukset, joita toistetaan
3. Testitapaukset, jotka ovat pitkäveteisiä tai vaikeita suorittaa manuaalisesti.
4. Aikaa vievät testitapaukset.

Samalla on tiettyjä tapauksia, joita ei tulisi automatisoida (Hamilton 2022b):

1. Testitapaukset, jotka on juuri luotu, eikä niitä ole suoritettu vielä manuaalisesti.
2. Testitapaukset, joiden vaatimukset muuttuvat usein.
3. Ad hoc -periaatteella suoritetut testitapaukset.

Kasurinen (2013) lähestyy automaatioon soveltuvia testitapauksia resurssien tarpeen, rahan sekä testitapausten kriittisyyden kautta. Testausautomaation rakentaminen vaatii aikaa ja rahaa, kun taas testien ajaminen on käytännössä ilmaista. Tämän vuoksi automatisoinnille soveliaimpia testitapauksia ovat ne, jotka pysyvät pitkään muuttumattomina ja joita toistetaan usein. Toinen kriteeri, varsinkin automatisoinnin priorisoinnille on Kasurisen (mt.) mukaan testitapausten kriittisyys. Kriittisyyteen tulisi ottaa kantaa testaussuunnitelmassa, jotta automatisoinnista saadaan paras mahdollinen hyöty.

Automaatiotestausprosessi voidaan jakaa viiteen osaan (Kuvio 4): Testaustyökalun valinta, testausautomaation laajuuden määritys, testien suunnittelu ja kehitys, testien suorittaminen ja testien ylläpito (Hamilton 2022b).



Kuvio 4. Automaatiotestausprosessin vaiheet (Hamilton 2022b).

3.1 Hyödyt ja riskit

Automatisointityökalut ja niiden ylläpito maksavat, samoin testaajien kouluttaminen. Testien automatisointi maksaakin ns. etupainotteisesti ja se tulisi nähdä tavallisena investointina. Hinta-hyöty analyysin avulla voidaan arvioida, kuinka suuri hyöty esimerkiksi regressiotestien automatisoinnista saadaan esimerkiksi henkilöstökulujen säästönä. Tämä luonnollisesti pätee vain, jos regressiotestejä voidaan ylipäätään automatisoida. (Spillner & Linz 2021, luku 7.2.)

3.2 Testaussuunnitelma

3.2.1 Automaatiotestauksen testitapausten laatiminen

Keskitymme tässä lähinnä funktionaalisiin, automaatiotestaukseen tarkoitettuihin testitapauksiin. Automaatiotestauksen testitapaukset poikkeavat hieman manuaalitestauksen testitapauksista, vaikkakin perusrunko on sama. Automaatiotestauksessa ns. mallirunkoon vaikuttaa kuitenkin paljon se, mikä automaatiotestauksen työkalu on käytössä ja millaisia erityisvaatimuksia se testitapaukselta vaatii. Perusrunko sisältää kuitenkin Owenin (2021) mukaan seuraavat osat:

- Testausta edeltävä tila tai tilan määrittely – Kuvaus kuinka sovellus asetetaan testattavaan tilaan
- Synkronisointi ja odotuslausekkeet – Sallitaan sovelluksen siirtyä haluttuun tilaan
- Testiaskeleet – Kuvaus siitä, mitä syötteitä sovellukselle annetaan ja kuinka se saavuttaa seuraavan tilan. Lisäksi myös kuinka sovellus palaa alkuperäiseen tilaan seuraavaa testiä varten.
- Kommentit – Selittävät automaation lähestymiskulman
- Virheenjäljitys (debugging) ja tulos -lausekkeet kuvaten, mihin tulokset tallennetaan

Automaatiotestien testitapauksissa keskitytään paljolti sovelluksen tilan vaihdoksiin ja datan muutoksiin. Sen vuoksi tärkeää, että yksi testitapaus sisältää vain yhden testin. (Owen 2021.)

3.2.2 Testitapausten valinta

Testitapausten tekeminen ja erityisesti automatisointi vaatii aikaa ja rahaa. Ja vaikka testiautomaation suorittaminen on edullista, automaatiotestien ylläpitäminen vaatii aikaa ja muita resursseja. Tämä vuoksi testitapausten valitsemiseen täytyy kiinnittää huomiota. Käytännössä myös projektin johto ja erilaiset asiakkaan vaatimukset asettavat sovelluksen testattaville osille erilaisia vaatimuksia. Testitapausten valintaan on Kasurisen (2013, 121–124) mukaan useita eri vaihtoehtoja ja lähestymistapoja. Yksinkertaisin lähestymistapa on tehdä riskianalyysi ohjelmiston määrittelyvaiheessa. Siinä pyritään analysoimaan ohjelman valmistumista, käytön kannattavuutta tai toimintaa uhkaavia riskejä. Haitallisimpiin riskeihin kohdistuvat testitapaukset valitaan ensimmäisenä.

Käytännössä testitapausten valinta voidaan kuitenkin jakaa kahteen pääsuuntaa, suunnitelmalähtöiseen ja riskilähtöiseen. Suunnitelmalähtöinen testitapausten valinnassa tarkoitus on saavuttaa riittävän korkea ohjelmiston laatu minimaalisilla kustannuksilla. Riskilähtöiseen lähestymistapaan verrattuna siinä oletetaan, että testausresursseja on käytettävissä riittävästi ja vähintäänkin kohtuullisesti. Testitapauksiksi valitaan ohjelman toimivuuden ja laatuvaatimusten kannalta oleelliset testitapaukset. (Kasurinen 2013, 121–123.)

Riskilähtöisessä testitapausten valinnassa pyritään minimoimaan viallisesta ohjelmistosta aiheutuvia kustannuksia ja valitsemaan testitapaukset sen perusteella. Toisin sanoen siinä pyritään poistamaan kaikista suurimmat ongelmat ja varmistamaan ohjelman päätoiminnallisuudet. Testaus pyritään tekemään mahdollisimman resurssitehokkaasti pienillä resursseilla. (Kasurinen 2013, 121–123.)

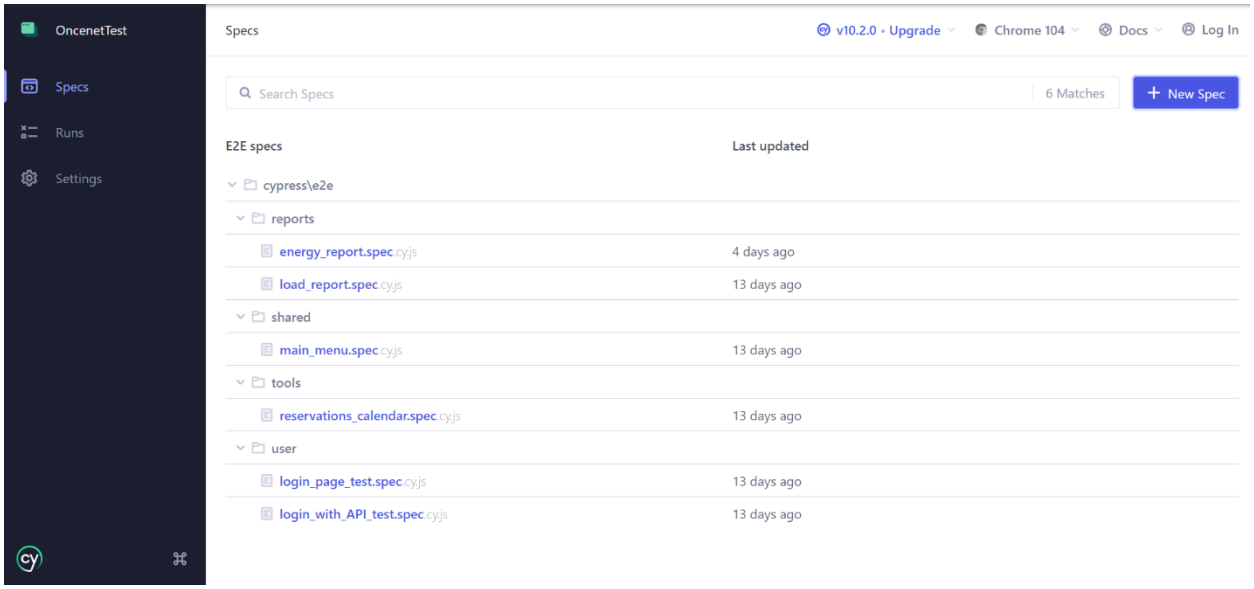
Molemmissa lähestymistavoissa törmätään kuitenkin kysymykseen, kuinka laajasti testataan? Testauksen laajuuteen vaikuttaa testiautomaation käyttö. Hamilton (2022b) luettelee automaatiotestauksen laajuuden määrittämistä helpottavia tekijöitä:

- Ominaisuuden liiketoimintakriittisyys
- Testiskenaariossa käsiteltävän tiedon määrä
- Toiminnallisuuden yleisyys / toistuvuus sovelluksessa
- Tekninen soveltuvuus
- Testitapauksen kompleksisuus
- Mahdollisuus käyttää samaa testitapausta useiden eri selainten testeissä.

3.3 Cypress-testauksen käytäntöjä

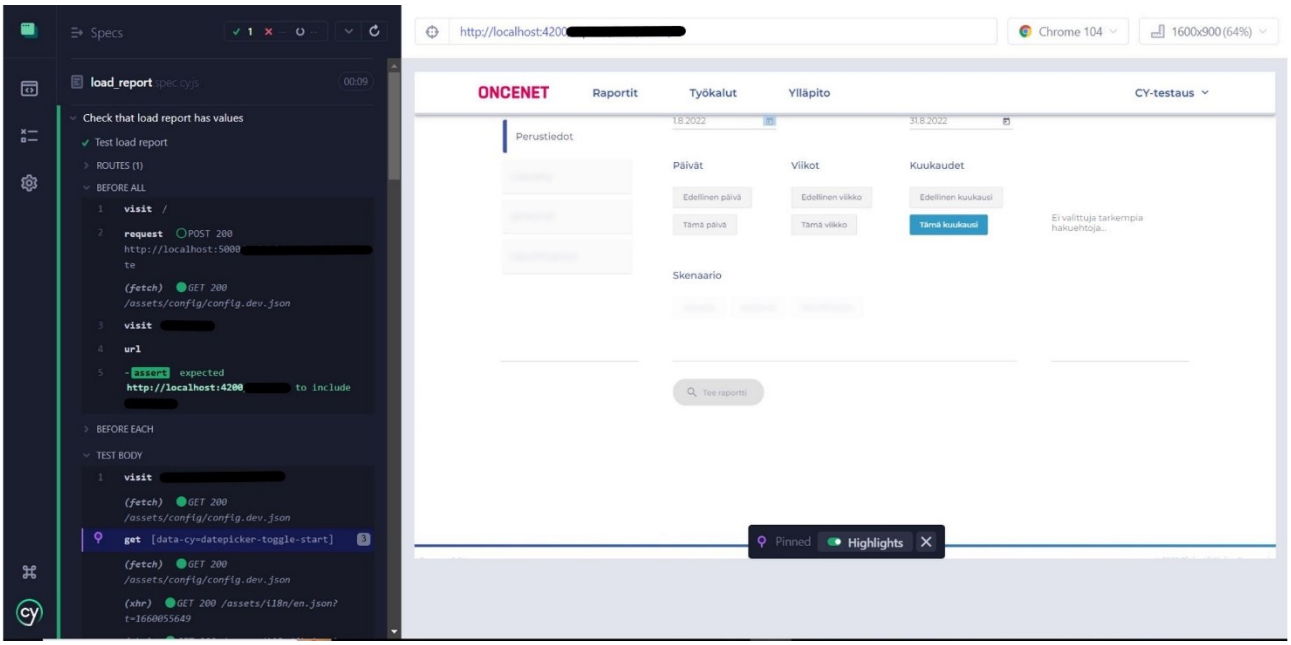
3.3.1 Cypress testauksen työkaluna

Cypress on testauksen käyttöliittymäpohjainen automatisointityökalu funktionaaliseen end-to-end testaukseen. Sen käyttö ei ole ohjelmointikielestä riippuvaa vaan soveltuu kaikille sovelluksille, jotka toimivat verkkoselaimessa. Cypress tukee verkkoselaimia kuten: Chrome, Firefox, Edge, Electron ja Brave. Automaatiotestien ohjelmointikielenä käytetään Javascriptiä. Ajatuksena Cypressissä on ollut koota end-to-end testauksen työkalut yhteen ja vähentää erillisten työkalujen asentamista. (Testing has been broken for too long n.d.)



Kuvio 5. Cypress käyttöliittymä.

Cypress-testejä voidaan suorittaa joko käyttöliittymän (Kuvio 5) kautta tai suoraan komentoriviltä. Jälkimmäistä käytetään erityisesti automatisoiduissa testiajoissa, kuten CI/CD mallissa (jatkuva integraatio, jatkuva toimitus). Ajettaessa testejä käyttöliittymän kautta Cypress myös näyttää testin etenemisen vaihe vaiheelta ja ottaa vaiheista kuvakaappauksia (Kuvio 6). Käyttöliittymässä on nähtävissä, missä vaiheessa testiä on tehty arvon assertointi (arvon tarkistus) ja onko se läpäisty hyväksytysti. Samoin käyttöliittymästä voidaan nähdä esimerkiksi, mistä kohdin sovellusta kyseistä arvoa on haettu.



Kuvio 6. Esimerkki Cypress-testin suorituksesta käyttöliittymän kautta.

3.3.2 Testien organisoiminen

Testien kirjoittamisen lisäksi Cypress-testauksessa on hyvä kiinnittää huomiota myös itse testauskansion rakenteeseen. Brian Mann ehdottaa (OKG! - Paul Dowman's tech events and interviews 2018) järjestelemään testit testattavan sivuston rakenteen mukaan, esimerkiksi artikkelit, käyttäjä, kirjoittaja ja jaetut. Tässä tapauksessa artikkelit kansioon sijoitetaan artikkeleihin liittyvät testit, käyttäjä kansioon sisäänkirjautumiseen ja rekisteröintiin liittyvät testit ja niin edelleen. Rakenne voi toki olla jokin muukin looginen sovelluksen rakenne, kuten päätoiminnallisuudet. Salasanojen ja käyttäjätunnusten hallinta olisi myös syytä miettiä etukäteen, jotteivat ne päädy versiohallin-

taan – vaikka käytettäisiinkin ”vain” testitunnuksia. Mikäli testauksessa käytetään muuttumattomia datatiedostoja, ne voidaan sijoittaa erilliseen kansioon (fixture) ja viitata cy.fixture() -funktiolla (fixture, n.d.).

Automaatiotestauksessa testien tulisi olla dynaamisia ja mahdollisimman yleiskäyttöisiä. Sen vuoksi erilaisten verkko-osoitteiden, käyttäjätunnusten, salasanojen ja muiden ”kovakoodattujen” arvojen kirjoittamista suoraan testiin tulisi välttää. Erilaisten muuttujien syöttämiseksi testiin voidaan käyttää ympäristömuuttujia. Cypressissä voidaan käyttää järjestelmän laajuisia (OS-level) tai Cypressin omia ympäristömuuttujia (Cypress.env). Jälkimmäisillä on se etu, että ne nollautuvat testien välillä, joten yksittäinen testi voidaan suorittaa täysin eristettynä. Cypress.env voidaan asettaa cypress.config -konfiguraatiotiedostossa, cypress.env.json -tiedostossa, komentoriviltä CYPRESS_ -alkuisella muuttujalla tai --env parametrillä ajettaessa ”cypress run” komentoriviltä. (Environment Variables, n.d.)

3.3.3 DOM-elementtien valinta

DOM (document objects model, dokumenttioliomalli) -elementtien valitseminen on keskeisessä roolissa Cypress -testien rakentamisessa. Valitsemalla käyttöliittymästä elementti ja suorittamalla sille toiminto saadaan sovelluksen tilaa muutettua. Lopuksi varmennetaan sovelluksen tila samasta tai toisesta elementistä. On siis olennaista valita oikea elementti, myös käyttöliittymämuutosten jälkeen tulevaisuudessa, kun testiautomaatiota suoritetaan seuraavaksi. Seuraavassa kuviossa (Kuvio 1) on esitetty eri tapoja elementtien valinnalle sen perusteella, kuinka kestäviä ne ovat (Best Practices, n.d.).

Selector	Recommended	Notes
`cy.get('button').click()`	⚠ Never	Worst - too generic, no context.
`cy.get('.btn.btn-large').click()`	⚠ Never	Bad. Coupled to styling. Highly subject to change.
`cy.get('#main').click()`	⚠ Sparingly	Better. But still coupled to styling or JS event listeners.
`cy.get('[name="submission"]').click()`	⚠ Sparingly	Coupled to the `name` attribute which has HTML semantics.
`cy.contains('Submit').click()`	✓ Depends	Much better. But still coupled to text content that may change.
`cy.get('[data-cy="submit"]').click()`	✓ Always	Best. Isolated from all changes.

Kuvio 7. Cypress.io DOM-elementtien valintatapoja (Best Practices, n.d.).

DOM-elementin valinta olisi siis parasta tehdä erillisen, testausta varten luodun id:n perusteella. Cypress ohjeistaa käyttämään data-alkuisia tunnisteita, kuten data-cy, data-test tai data-testid. Tapauskohtaisesti, toiseksi paras tapa olisi käyttää tekstisisältöön perustuvaa valintaa, joka on luonnollisesti altis sisällön muutoksille. Sen sijaan liian geneerisiä tai automaattiseen tyylittelyyn nojautuvia valintoja tulee välttää, koska ne eivät ole yksiselitteisiä tai ovat alttiita muutoksille, jolloin testi rikkoutuu helposti. (Best Practices, n.d..)

3.3.4 Testien riippumattomuus

Yksittäisen testin tulisi olla riippumaton toisten ennen tai jälkeen suoritettujen testien lopputuloksesta. Mikäli edellinen testi epäonnistuu, sillä ei tulisi olla vaikutusta testin tulokseen. Myöskään sillä, mihin tilaan edellinen testi sovelluksen jätti ei tulisi olla väliä. Tällä tarkoitetaan testien riippumattomuutta. Yksittäisen testin riippumattomuuden voi testata helposti muuttamalla Cypressin it-

komennon it.only -komennoksi, jolloin ainoastaan kyseinen testi ajetaan. Mikäli testi läpäistään hyväksytyksi, kyseinen testi on riippumattomuuden suhteen hyvä. (Best Practices, n.d..)

Yksi syy, miksi päädytään tilanteeseen, jossa testit eivät ole riippumattomia, on testitapausten liiallinen pilkkominen. Kuvio 8 osoittaa ongelman, jossa lomakkeen testaaminen on pilkottu neljään eri testiin, mutta vain ensimmäisessä siirrytään lomakkeelle (cy.visit). Vaikka yksittäisten testien tulee olla yksinkertaisia, liiallinen ja vääränlainen pilkkominen johtaa esimerkissä tilanteeseen, jossa kaikki ensimmäisen testin jälkeen tulevat testit ovat riippuvaisia siitä ja ensimmäisen testin epäonnistuessa, myös muut epäonnistuvat.

End-to-End Test | Component Test

```
// an example of what NOT TO DO
describe('my form', () => {
  it('visits the form', () => {
    cy.visit('/users/new')
  })

  it('requires first name', () => {
    cy.get('[data-testid="first-name"]').type('Johnny')
  })

  it('requires last name', () => {
    cy.get('[data-testid="last-name"]').type('Appleseed')
  })

  it('can submit a valid form', () => {
    cy.get('form').submit()
  })
})
```

Kuvio 8. Esimerkki testien liiallisesta pilkkomisesta (Best Practices, n.d.).

Tilanteen korjaamiseksi testejä tulisi koota suuremmaksi kokonaisuudeksi ja käyttää before tai beforeEach osioita toistuvalle koodille (tässä tapauksessa esimerkiksi cy.visit). Tilan palauttaminen tulisi siis tehdä pikemminkin ennen jokaista testiä before tai beforeEach komennoilla kuin jokaisen

testin jälkeen. Tähän yksinkertaisena selityksenä on, että testin keskeytyessä tila ei palautuisi ennalleen käyttäen esimerkiksi after tai afterEach komentoja, koska niitä ei suoritettaisi. (Best Practices, n.d..)

3.3.5 Mukautetut komennot

Cypress sisältää oman API rajapinnan, jonka avulla on mahdollista luoda omia ja korvata olemassa olevia komentoja. Oletussijainti mukautetuille komennoille on tiedosto sijainnissa *cypress/support/commands.js,* mutta mikäli mukautettuja komentoja on runsaasti, saattaa olla perusteltua jakaa komentoja useampaan tiedostoon ja määrittää (import) tiedostot index-tiedostossa. (Custom Commands, n.d..)

Suositeltava tapa Cypress -sivuston (Building Cypress Commands, n.d.) mukaan on luoda uusia komentoja sitä mukaan, kun komennot alkavat toistua kirjoitetuissa testeissä. Toisaalta mikäli sovelluksessa käytetään esimerkiksi *data-test* tunnisteita, tiedetään jo etukäteen, että kyseistä elementtiä tullaan hakemaan usein. Tämän vuoksi esimerkiksi seuraavanlaisen (Kuvio 9) mukautetun komennon kirjoittaminen heti alussa olisi perusteltua.

```
Cypress.Commands.add("getBySel", (selector, ...args) => {
  return cy.get(`[data-test=${selector}]`, ...args)
})
```

Kuvio 9. Mukautettu komento, jolla haetaan data-test tunnistetta DOM-puusta.

Toinen vastaava, usein käytetty mukautettu komento suorittaa sisäänkirjautumisen. Sisäänkirjautuminen voidaan tehdä joko ”perinteisellä” tavalla, kirjoittaen automaatiolla asianmukaisiin kenttiin käyttäjätunnus ja salasana tai käyttäen hyväksi sovelluksen API-rajapintaa. Jälkimmäistä tapaa suositellaan käytettäväksi nopeutensa vuoksi (OKG! - Paul Dowman's tech events and interviews 2018), mutta myös tavanomaisella kirjautumisella on paikkansa erityisesti sisäänkirjautumistesteissä.

4 Toteutus

4.1 Kehittämistyön menetelmä

Opinnäytetyö toteutettiin tutkimuksellisena kehittämistyönä. Tutkimuksellinen kehittämistyö saa alkunsa esimerkiksi organisaation kehitystarpeesta tai uudistumisen halusta. Tarkoituksena on ratkaista käytännön ongelma, tuottaa ideoita tai luoda käytäntöjä, tuotteita tai palveluita. Tutkimuksellisessa kehittämistyössä tehdään asioita käytännössä ja viedään niitä eteenpäin. (Ojasalo, Moilanen & Ritalahti, 2015 s. 18–19)

4.2 Kehittämistyön tarkoitus, tavoitteet ja tutkimuskysymykset

Vaikka toimeksiantajalla on käytössään testausta eri tasoilla, niin manuaali- kuin automaatiotestaustakin, on heillä tarve Oncenet-sovelluksen kokonaisvaltaisemmalle automaatiotestaukselle. Automaatiotestauksen laajentamisella pyritään vapauttamaan ohjelmistokehittäjien aikaa sovelluksen kehittämiseen, lisäämään testauskattavuutta ja toisaalta poistamaan päällekkäistä työtä.

Opinnäytetyön kehittämistehtävän tavoitteena on:

1. Luoda yrityksen kehittämälle selainpohjaiselle sovellukselle pohja automaatiotestaukseen Cypress-kirjastolla.
2. Rakentaa sovellukselle keskeisimmät automaatiotestitapaukset regressiotestauksen tarpeisiin.

Tutkimuskysymykset:

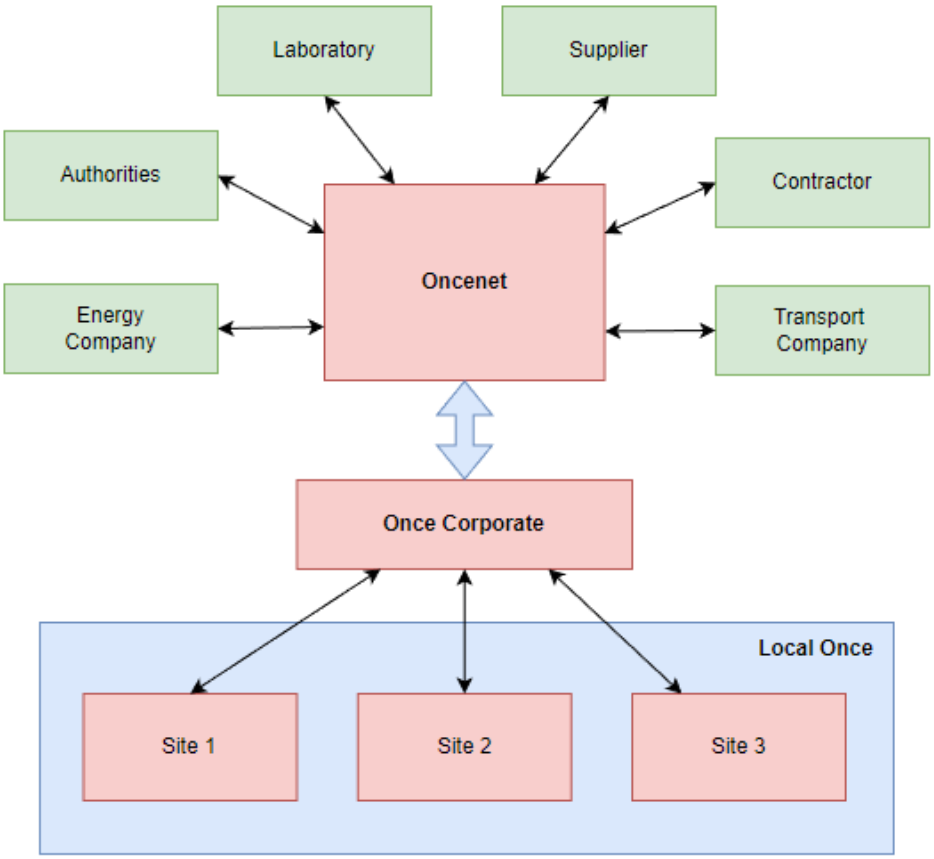
1. Kuinka automaatiotestitapaukset saadaan ylläpidettäviksi?
2. Millaisia regressiotestitapauksia tulisi testata automaatiolla?

Työn lopputuloksena yrityksellä ja sen ohjelmistokehittäjillä on käytössään Cypress-kirjastolla muodostettu sarja automaatiotestejä. Testien ajo tapahtuu joko manuaalisesti ja/tai jatkuvan integraation keinoin. Yrityksellä on jonkinasteinen kuva, kuinka rakentaa ylläpidettäviä testejä ja millaisia regressiotestitapauksia tulisi automatisoida.

4.3 Testattava sovellus ja järjestelmäkonfiguraatio

Once by Pinja koostuu kolmesta pääosasta: Once Local, Once Corporate ja Oncenet (Kuvio 10). Edellä mainituista Once Corporate ja Oncenet ovat ns. SaaS-palveluita ja Once Local on ns. on-premise palvelu eli sen käyttäminen vaatii ohjelmistoasennuksia. Tavallisesti Once Local -järjestelmään on liitetty myös erilaisia IO-laitteita, kuten vaakoja, kameroita tai mittareita. Once Corporate -järjestelmä toimii nimensä mukaisesti yritystasolla ja kokoaa yhteen eri Once Local -ympäristöjen tiedot. Once Corporate sisältää laitoksen henkilökunnalle näkyvät tiedot ja tuottaa raportteja mahdollistaen yritykselle eri tietojen muokkauksen ja syötön. Once Corporaten avulla mm. hallitaan tilaus-toimitusketjua, tehdään viranomaisraportointia ja inventoidaan varastoja.

Oncenetin rooli Once by Pinja järjestelmässä on toimia linkkinä Once Corporaten ja kolmansien osapuolien välillä. Oncenet tarjoaa käyttöliittymän muun muassa ulkopuolisille tavarantoimittajille, urakoitsijoille, kuljetusliikkeille, laboratorioille, viranomaisille ja energialaitoksille. Oncenet joko tarjoaa tai kerää tietoa. Tarjottava tieto voi olla esimerkiksi tieto toimitetun polttoaineen laadusta ja kerätty tieto puolestaan esimerkiksi polttoaineanalyysien tuloksia laboratoriolta.



Kuvio 10. Once by Pinja arkkitehtuuri.

Testattavan Oncenet sovelluksen frontend on toteutettu Angular- ja backend C#-ohjelmointikielellä. Tietokantana toimii SQL-tietokanta. Sovellus toimii SaaS-palvelumallin mukaisesti eli käyttäjän ei tarvitse asentaa koneelleen mitään, ja sovelluksen käyttö tapahtuu täysin verkon yli. Business logiikan rakenteella oli jonkin verran merkitystä testausympäristön rakentamisessa ja ylläpidossa. Cypress-testauksessa tällä ei juurikaan ollut vaikutusta.

4.4 Automaatiotestauksen rakentaminen

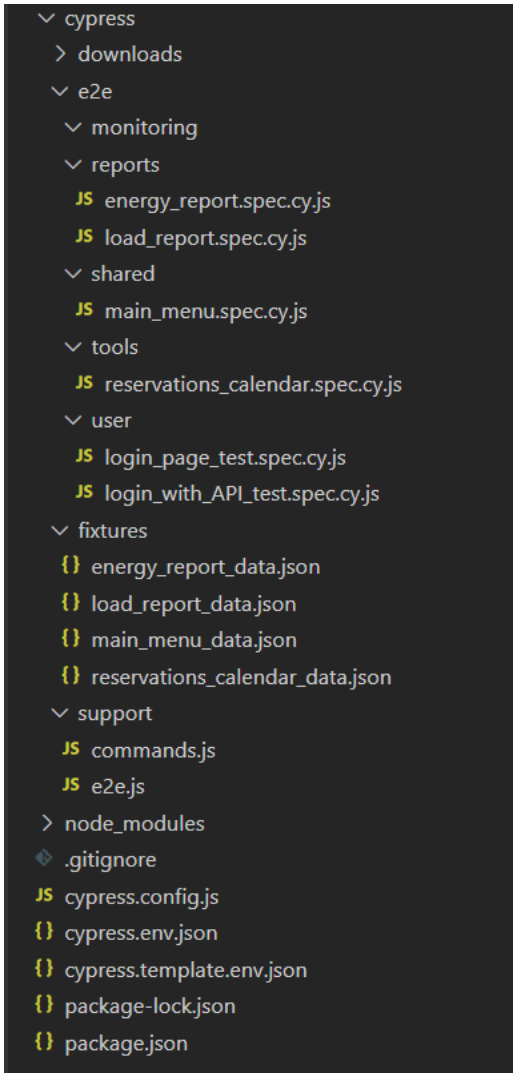
4.4.1 Kehitysympäristö ja testitietokanta

Testien kirjoittamista varten tuli pystyttää paikallinen kehitysympäristö. Tämä toteutettiin virtualisoimalla Linux-pohjainen virtuaalikone, johon asennettiin frontend valmiin asennusskriptin avulla. Backendin ohjelmistokoodi ajettiin lokaalisti C# sovelluksena omalla koneella toimeksiantajayrityksen asennusohjeiden avulla.

Tietokannan osalta täytyi toimia hieman toisin. Aluksi toimeksiantajan testitietokannasta otettiin kopio, joka sitten siirrettiin toiselle tietokantapalvelimelle ja nimettiin uudelleen. Tämän jälkeen myös tietokannan synonyymit täytyi poistaa ja luoda oikeilla nimillä uudelleen. Tietokanta luotiin tässä vaiheessa erillisenä, jotta testattava data pysyisi muuttumattomana. Tämä sen vuoksi, ettei toimeksiantajalla vielä ollut olemassa tarkempaa arkkitehtuuria automaatiotestaukselle. Myös olemassa olevaa testikantaa olisi voitu hyödyntää. Ainoa riski olemassa olevan testikannan käyttämisessä olisi ollut liian lähellä menneisyydessä olevien datan arvojen muuttuminen siinä vaiheessa, kun testikanta päivitetään tuotannosta. Jatkoa ajatellen testikannan käyttäminen saattaa ollakin helpoin vaihtoehto ja testejä kirjoittaessa tämä tulisi ottaa huomioon.

4.4.2 Testien organisointi

Vaikka alkuvaiheessa testejä ei ollutkaan lukumääräisesti monia, päätettiin heti alussa järjestellä testit omiin kansioihinsa toiminnon perusteella (Kuvio 11).



Kuvio 11. Cypress-testauksen kansiorakenne.

Tiedostorakenteen juureen luotiin *cypress.env.json* tiedosto muuttujille, erillinen versionhallintaan vietävä template kyseisestä tiedostosta sekä erillinen konfigurointitiedosto *cypress.config.js*. Rakenteeseen lisättiin oma *fixtures*-kansio ja sen alle testien kirjoituksen yhteydessä tiedostot staattiselle datalle. Lisäksi luotiin muutamia mukautettuja komentoja, joita etukäteen tiedettiin tarvittavan useamman kerran (Liite 6). Kyseisiä komentoja olivat lähinnä login-komennot ja data-cy-tunnistimen etsimiskomennot.

4.4.3 Sisäänkirjautumisen testaus

Automaatiotestien kirjoittaminen aloitettiin sisäänkirjautumisen (login) testaamisella. Cypress mahdollistaa kaksi toisistaan poikkeavaa tapaa sisäänkirjautumiselle, ohjelmallisen ja käyttöliittymään perustuvan. Tavallisesti testien yhteydessä sisään kirjautuessa tulisi suosia nopeampaa ohjelmallista sisään kirjausta, mutta koska sisäänkirjautuminen on usein ohjelman toiminnan kannalta kriittinen, halutaan se testata kerran käyttäjän tavoin käyttöliittymällä. Tämä vuoksi sisään kirjausta testatessa siis käytetään hieman eri komentoja kuin muissa testeissä, joissa käytetään ohjelmallista ASI-rajapintaa hyödyntävää sisään kirjausta.

Seuraavassa kuviossa (Kuvio 12) sama sisäänkirjautumistesti on kirjoitettu kolmella tavalla, joista jokainen toimi testin kirjoitushetkellä. Kahdessa ylimmässä on käytetty DOM-puun elementtejä ja niiden attribuutteja sekä luokkia. Erityisesti luokkaan (class) nojautuva elementin arvon tarkistaminen (assertointi) olisi todennäköisesti vaarassa muuttua tulevaisuudessa. Alimmassa testissä oikeat elementit DOM-puusta on haettu ohjelmakoodiin lisättyjen *data-cy*-tunnisteiden avulla. Ainoastaan *submit*-napin sijainti on haettu pelkästään elementillä ja sen attribuutilla. Tämä ei kyseissä testissä ole haitaksi niin kauaa, kuin sisään kirjaussivulle ei lisätä useampia nappeja. Testin kestävyyden vuoksi *submit*-nappiin tulisikin lisätä *data-cy*-tunniste.

```
describe('Login', () => {
    it('login manually', () => {
        cy.visit('/')
        cy.get('input[formcontrolname="username"]').type(Cypress.env('userName'))
        cy.get('input[formcontrolname="password"]').type(Cypress.env('userPassword'))
        cy.get('button[type="submit"]').click()
        cy.get('[class="mat-menu-trigger menu-item--container cursor-pointer ng-star-inserted"]').should('contain', Cypress.env('userName'))
    })
})

describe('Login', () => {
    it('login manually', () => {
        cy.visit('/')
        cy.get('[id="mat-input-0"]').type(Cypress.env('userName'))
        cy.get('[id="mat-input-1"]').type(Cypress.env('userPassword'))
        cy.get('button[type="submit"]').click()
        cy.get('[class="mat-menu-trigger menu-item--container cursor-pointer ng-star-inserted"]').should('contain', Cypress.env('userName'))
    })
})

describe('Login', () => {
    it('login manually', () => {
        cy.visit('/')
        cy.get('[data-cy="login-username"]').type(Cypress.env('userName'))
        cy.get('[data-cy="login-password"]').type(Cypress.env('userPassword'))
        cy.get('button[type="submit"]').click()
        cy.get('[data-cy="toolbar-username"]').should('contain', Cypress.env('userName'))
    })
})
```

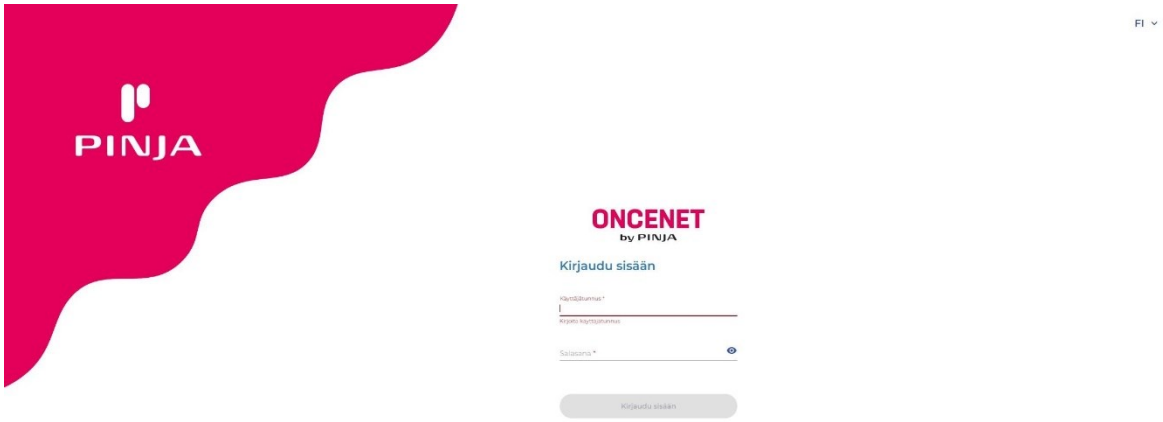
Kuvio 12. Kolme eri tapaa sisään kirjaustestille.

Varsinaisessa sisäänkirjautumistestissä (ks. Liite 1.) testi tehtiin Oncenetin sisäänkirjautumissivulle (Kuvio 13). Testissä pyrittiin käymään eri skenaariot läpi käyttäjän kirjautuessa tai yrittäessä kirjautua Oncenet:iin. Testissä käsiteltyjä skenaarioita olivat:

- väärä käyttäjätunnus,
- väärä salasana,
- tyhjä salasana,
- tyhjä käyttäjätunnus,
- tyhjä käyttäjätunnus ja salasana sekä
- onnistunut kirjautuminen ja uloskirjaus.

Testiin jäi joitain puutteita tunnisteiden suhteen, jotka tulisi korjata lisäämällä tarvittava *data-cy*-tunniste koodiin. Testi tarkistaa väärän salasanan tai käyttäjätunnuksen tapauksissa sovelluksen tuottaman virheilmoituksen. Tapauksissa, joissa joko käyttäjätunnus tai salasana on tyhjä merkkijono, kirjaudu sisään -nappi on poistettu käytöstä. Testi varmistaa tämän. Onnistuneessa kirjautumisessa ei erikseen ole kirjoitettu vahvistusta, koska *cy.login* on itse luotu mukautettu komento ja pitää sisällään jo yhden vahvistuksen. Uloskirjaaminen vahvistetaan url-osoitteesta.

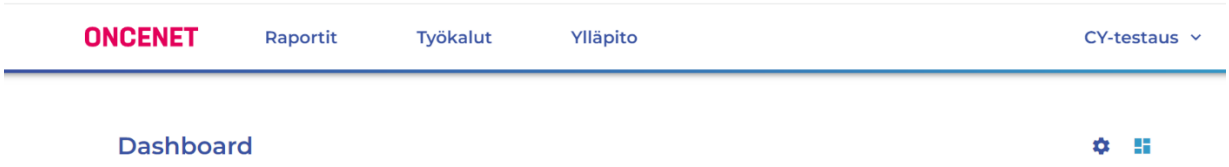
Ongelmana testin suorituksessa tai oikeastaan sen suorittamisessa useammin kuin kerran peräkkäin on väärien tunnusten ja salasanojen syöttäminen. Oncenet-sovellus nimittäin asettaa käyttäjätunnuksen hetkeksi käyttökieltoon liian monen virheellisen kirjautumisen jälkeen. Tavallisessa testauksessa, jossa testi ajetaan vain kerran tämä ei kuitenkaan ole ongelma.



Kuvio 13. Oncenet kirjautumissivu.

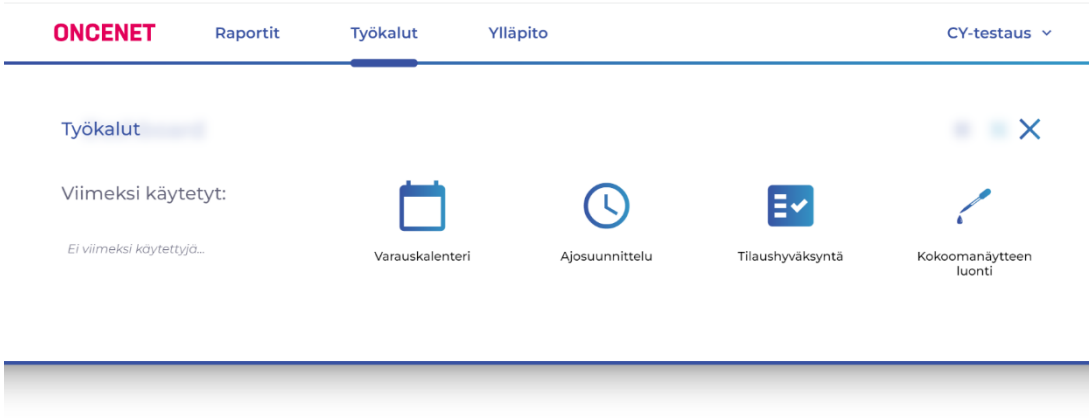
4.4.4 Päävalikon testaus

Kirjautumisen jälkeen Oncenet -sovelluksessa päädytään sivulle, jossa näkyy päävalikko (Kuvio 14). Päävalikon testauksessa testattiin valikoiden ja niiden alavaliokoiden (Kuvio 15) näkyvyyden testikäyttäjällä. Testaus oli tehtävä kyseisellä testikäyttäjällä, koska käyttäjän oikeudet vaikuttavat valikoiden näkyvyyteen. Tässä vaiheessa testausta ei kehitetty dynaamista käyttäjän ja käyttöoikeuksien luomista.



Kuvio 14. Päävalikko ja aloitussivu.

Päävalikon testauksessa testattavat valikot haettiin dynaamisesti fixtures-tiedostosta ja kaikkien tiedostossa olevien valikkonimien olemassaolo testattiin. Testissä käytettiin englanninkielistä käyttöliittymää, mutta samoin olisi voitu testata myös suomenkielinen käyttöliittymä.

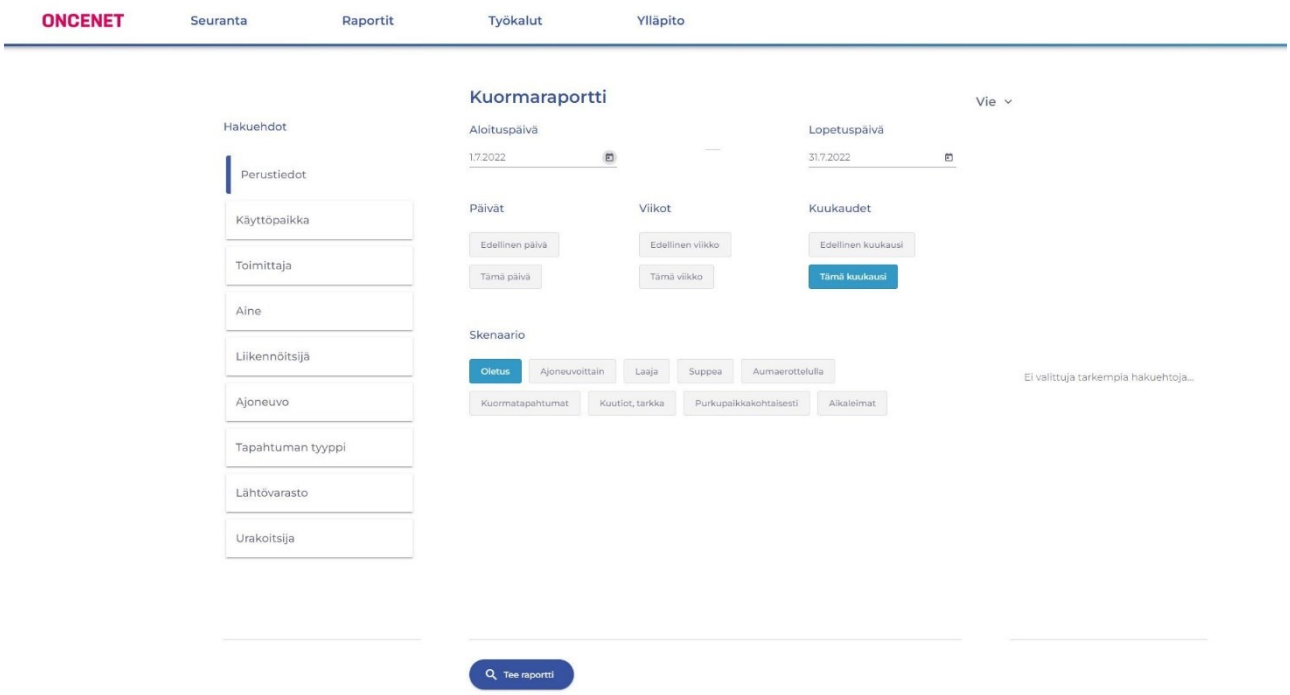


Kuvio 15. Työkalut-päävalikon alavalikko.

4.4.5 Raporttien testaus

Yksi Oncenet-sovelluksen keskeisistä toiminnoista ovat raportit. Tämän vuoksi raporteista luotiin kaksi automaatiotestiä, yksi energia- ja yksi kuormaraportista (Liite 2.). Molemmissa raporteissa on samanlainen suodatusvalikko raportin tulostuksessa, joten testeissä voitiin käyttää samanlaisia elementtien valintafunktioita. Raportin aloitus- ja lopetuspäivämäärien valinta tuli toteuttaa valitsemalla oikea päivämäärä Material-UI Date picker-elementistä. Päivämäärän kirjoittaminen oli elementissä estetty ja tämä pidensi hieman testiä.

Päivämääriin ja vasemmalla (Kuvio 16) oleviin hakuehtoihin annettiin fixtures-kansiossa olevan datatiedoston arvot. Testissä muodostetusta raportista vastaavasti etsittiin rivit, joiden arvoja verrattiin samassa tiedostossa oleviin arvoihin. Tällä tavoin raporttien testauksessa pyrittiin testaamaan raportin hakuehtojen toimivuutta, sitä tuottaako raportti dataa ja onko tuotettu raportti muuttunut arvojen osalta esimerkiksi kehityksen vuoksi. Tässä tapauksessa testattavia rivejä valittiin vain kaksi, mutta testattavia arvoja voi olla enemmänkin. Varsinaisessa testitietokannassa datan arvojen testaaminen on joidenkin raporttien kohdalla ongelmallista, koska datan arvot voivat muuttua myös menneiden arvojen osalta, jos dataa korjataan. Tätä tapahtuu, mutta mitä kauempaa menneisyydestä testidatan arvot haetaan, sitä epätodennäköisempiä muutokset ovat.

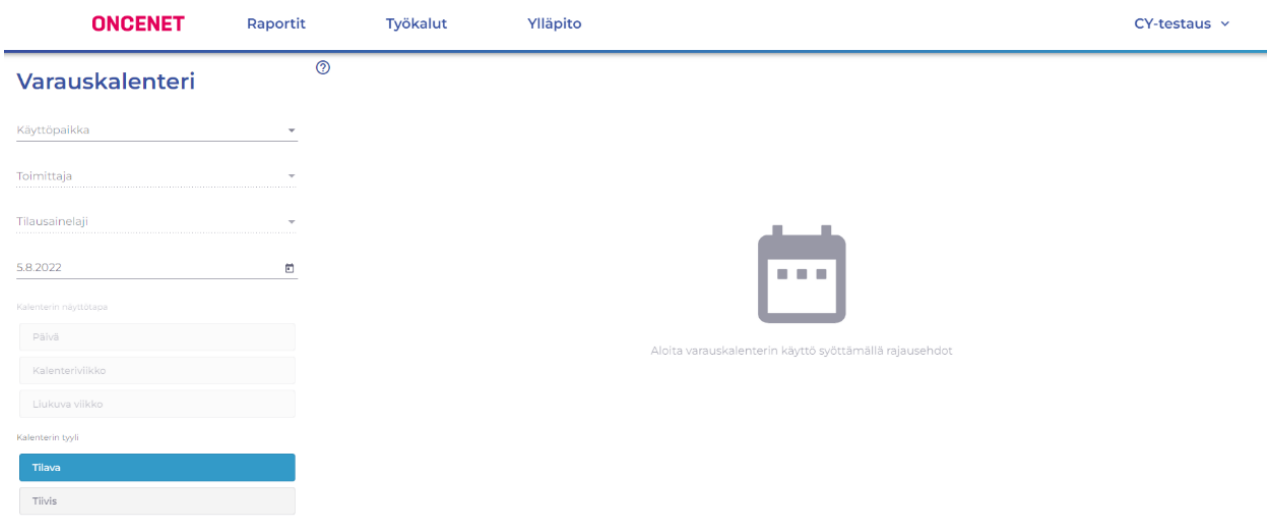


Kuvio 16. Kuormaraportin hakuehdot.

4.4.6 Varauskalenterin testaus

Toinen keskeisiä Oncenet-sovelluksessa käytettäviä ominaisuuksia on varauskalenteri (Kuvio 17), jossa käyttäjät näkevät ja varaavat kuormien purkuaikoja. Automaatiotestauksen osalta varauskalenterissa on useita eri toimintoja, joita testata. Luodussa automaatiotestissä keskityttiin kuitenkin

vain perustestiin, jossa pyrittiin testaamaan tietyn käyttöpaikan, toimittajan, tilausainelajin ja päivämäärän tuottamaan varauskalenteriin ja sen arvoihin. Testissä ei siis pyritty luomaan uutta tai muokkaamaan olemassa olevia varauksia.



Kuvio 17. Varauskalenterin suodatusvalikko.

Varauskalenterin suodatusvalikko osoittautui ongelmalliseksi automaatiotestiä ajettaessa. Tuntemattomasta syystä elementtien valinta toimi noin joka toisella testin ajokerralla halutusti, mutta muutoin kaikkia elementtejä ei löytynyt toivotusti, vaikka käytössä oli myös data-cy-tunnisteita. Todennäköisin syy liittyy Cypressin asynkroniseen suorittamiseen. Ongelman ratkaisuksi, joskaan ei kovinkaan ihanteelliseksi, saatiin ylimääräisten odotusaikojen (cy.wait) lisääminen testiin.

5 Tulokset

Opinnäytetyössä pyrittiin selvittämään, kuinka automaatiotestitapaukset saadaan ylläpidettäviksi. Ylläpidettävyyteen vaikuttavia tekijöitä käytiin läpi Cypress-testauksen käytäntöjä tutkittaessa. Tärkeimpiä esille nostettavia ominaisuuksia, jotka vaikuttavat ylläpidettävyyteen on testien selkeys ja riippumattomuus sekä elementtien resilienssi tulevaisuuden muutoksille. Myös testikansion rakenne sekä erilliset tiedostot konfigurointitiedoille ja ympäristömuuttujille auttavat testien ylläpitoa tulevaisuudessa.

Opinnäytetyössä haettiin vastausta siihen, millaisia regressiotestitapauksia tulisi testata automaatiolla. Tähän ei kuitenkaan saatu yksiselitteistä vastausta. Automaatiotestauksessa testitapausten valinnassa on strategisesti kaksi pääsuuntaa: riskilähtöinen ja suunnitelmalähtöinen. Strategian valinta siis vaikuttaa jonkin verran siihen mitä testitapauksia automatisoidaan. Käytännössä regressiotestitapausten automatisointiin vaikuttavat samat tekijät kuin muidenkin testien automatisointiin ja ensimmäisenä tulisi lähtökohtaisesti automatisoida seuraavat testitapaukset:

- Korkean riskin, liiketoimintakriittiset testitapaukset
- Testitapaukset, joita toistetaan
- Testitapaukset, jotka ovat pitkäveteisiä tai vaikeita suorittaa manuaalisesti.
- Aikaa vievät testitapaukset.

Lisäksi automatisointiin vaikuttavat esimerkiksi testitapauksen tekninen soveltuvuus ja niiden kompleksisuus.

Yksi opinnäytetyön kehittämistehtävän tavoitteista oli luoda toimeksiantajan kehittämälle selainpohjaiselle sovellukselle pohja automaatiotestaukseen käyttäen Cypress-kirjastoa. Kehittämistyössä luotiin alustava kansiorakenne (Kuvio 11), otettiin käyttöön ns. fixture-tiedostot ja ympäristömuuttujille oma tiedosto sekä luotiin mukautettuja komentoja peruskäyttöön.

Toisena kehittämistehtävän tavoitteena oli rakentaa sovellukselle keskeisimmät automaatiotestitapaukset regressiotestauksen tarpeisiin. Kehittämistehtävässä luotiin yhteensä viisi testitiedostoa, jotka testaavat kirjautumista, raporttien muodostamista ja varauskalenterin toimintaa. Testien pääasiallisena tarkoituksena oli varmistaa ominaisuuden toimivuus.

6 Pohdinta

Kehittämistyön tuloksena saatiin rakennettua toimeksiantajalle alku Cypress-perustaiselle End-to-End-automaatiotestaukselle Oncenet-sovellukseen. On kuitenkin syytä muistaa, että vaikka automatisointi nopeuttaa kehitysvaiheen testausta se ei korvaa manuaalista testausta kokonaan, pahimmillaan se voi luoda valheellisen turvallisuuden tunteen tuotteen laadusta. Kaikkia testitapauksia ei ole mahdollista tai järkevää automatisoida. Testiautomaation rakentaminen vie resursseja. Oncenet-sovelluksen osalta olisi kenties järkevää implementoida testiautomaation rakentaminen osaksi tuotekehitysprosessia, kuten toimeksiantajayrityksen sisällä jo osittain toimitaankin. Kokonaisuutena automaatiotestauksen käyttäminen tuo useita etuja ja vapauttaa kehittäjien aikaa sovelluksen kehitystyöhön, jos automaatiotestit on valittu oikein ja niiden merkitys testauksen kokonaisuudessa ymmärretään.

Tehdyssä kehittämistyössä luodut rakenteet ja tiedostot voivat muuttua erilaisten parannusten tai testausstrategian muuttumisen myötä. Esimerkiksi ns. fixture-tiedostojen käyttö voi muuttua, riippuen data-arvojen varmistustavasta tai siitä halutaanko sitä tehdä. Samoin dynaamisuuden lisääminen voi vaikuttaa fixture-kansiossa olevien json-tiedostojen rakenteeseen. Työssä lisättiin data-cy-tunnisteita testattavan sovelluksen koodiin. Se vaikutti yhdessä valmiiden testitapausten puutteen ja Oncenet-sovelluksen kompleksisuuden kanssa automaatiotestitapausten määrään kehittämistehtävän resurssien puitteissa. Tästä johtuen kirjoitettuja testejä on lukumääräisesti melko vähän ja ne ovat verraten yksinkertaisia. Toisaalta ne antavat kuitenkin hyvän pohjan jatkokehitykselle.

Automaatiotestien kehittämisen seuraava vaihe on mielestäni testauksen muuttaminen dynaamisemmaksi muun muassa testikäyttäjän luomisen ja poistamisen muodossa. Tämä tarkoittaa testikäyttäjän luomista testisarjaa suoritettaessa, sen käyttöoikeuksien muuttamista ja lopulta poistamista. Tämä ei tietenkään ole pakollista, mutta tällä tavoin myös päävalikon ja eri toimintojen näkyvyyttä testikäyttäjälle voidaan hallita paremmin.

Lähteet

Best Practices. N.d. Cypress.io -sivuston ohjeita. Viitattu 17.7.2022. https://docs.cypress.io/guides/references/best-practices#Selecting-Elements.

Big List of Naughty Strings. 2021. Github repositorio. Viitattu 19.6.2022. https://github.com/minimaxir/big-list-of-naughty-strings.

Building Cypress Commands. N.d. Cypress.io -sivuston ohjeita. Viitattu 19.7.2022. https://learn.cypress.io/advanced-cypress-concepts/building-the-right-cypress-commands.

Custom Commands. N.d. Cypress.io -sivuston ohjeita. Viitattu 18.7.2022. https://docs.cypress.io/api/cypress-api/custom-commands.

Environment Variables. N.d. Cypress.io -sivuston ohjeita. Viitattu 26.7.2022. https://docs.cypress.io/guides/guides/environment-variables.

Hamilton, T. 2022a. What is BLACK Box Testing? Techniques, Example & Types. Artikkeli Guru99-sivustolla. Viitattu 19.6.2022. https://www.guru99.com/black-box-testing.html.

Hamilton, T. 2022b. Automation Testing Tutorial: What is Automation Testing? Artikkeli Guru99-sivustolla. Viitattu 27.6.2022. https://www.guru99.com/automation-testing.html.

Hamilton, T. 2022c. END-To-END Testing Tutorial: What is E2E Testing with Example. Artikkeli Guru99-sivustolla. Viitattu 7.7.2022. https://www.guru99.com/end-to-end-testing.html.

Hamilton, T. 2022d. What is Functional Testing? Types & Examples (Complete Tutorial). Artikkeli Guru99-sivustolla. Viitattu 13.8.2022. https://www.guru99.com/functional-testing.html.

Juvonen, R. 2018. Ohjelmistoprojektin sudenkuopat ja miten ne vältetään. Helsinki: Books on Demand.

Kasurinen, J. 2013. Ohjelmistotestauksen käsikirja. Jyväskylä: Docendo.

Ojasalo, K., Moilanen, T. & Ritalahti, J. 2015. Kehittämistyön menetelmät – Uudenlaista osaamista liiketoimintaan. 3.–4. painos. Helsinki: Sanoma Pro Oy.

OKG! - Paul Dowman's tech events and interviews. 2018. Video. Brian Mann – I see your point, but… (Part 1). Julkaistu 5.3.2018. Viitattu 19.7.2022. https://www.youtube.com/watch?v=5XQOK0v_YRE.

Once by Pinja. N.d. Pinja konsernin verkkosivusto Once tuotteesta. Viitattu 31.5.2022. https://pinja.com/palvelut/bioenergia-ja-kiertotalous/once.

Owen, G. 2021. How to write test cases for both manual and automated tests. Artikkeli TechTarget -sivustolla. Viitattu 7.7.2022. https://www.techtarget.com/searchsoftwarequality/tip/How-to-write-test-cases-one-component-at-a-time.

Protacon on nyt Pinja – teollisuuden uudistamisella ja digitalisaatiolla vahvaa kasvua. 2020. Pinjan tiedote verkkosivuilla 26.3.2020. Viitattu 2.6.2022. https://pinja.com/uutiset/protacon-on-nyt-pinja-teollisuuden-uudistamisella-ja-digitalisaatiolla-vahvaa-kasvua.

SDLC V-Model. N.d. Ohjelmistokehityksen V-mallin selitys w3Scoolsin sivustolla. Viitattu 13.6.2022. https://www.w3schools.in/sdlc/v-model.

Pezzè, M. & Young, M. 2008. Software testing and analysis – Process, principles and techniques. Hoboken: John Wiley & Sons Inc.

Pinja. N.d. Pinjan verkkosivusto. Viitattu 12.8.2022. https://pinja.com/pinja.

pp_pankaj 2019. Difference between System Testing and End-to-end Testing. Artikkeli Geegsfor-Geegs -sivustolla. Viitattu 7.7.2022. https://www.geeksforgeeks.org/difference-between-system-testing-and-end-to-end-testing/.

Pyhäjärvi, M. 2018. What is Exploratory Testing? Artikkeli Medium -verkkolehdessä. Viitattu 26.6.2022. https://medium.com/@maaret.pyhajarvi/what-is-exploratory-testing-88d967060145.

Spillner, A. & Linz, T. 2021. Software Testing Foundations, 5th Edition: A Study Guide for the Certified Tester Exam. 5. uudistettu painos. Heidelberg: dpunkt.verlag GmbH. Viitattu 29.6.2022. https://janet.finna.fi, Skillsoft Books ITPro.

Testing has been broken for too long, n.d. Cypress.io työkalun verkkosivut. Viitattu 28.6.2022. https://www.cypress.io/how-it-works/.

Tevanlinna, A. 2004. Integrointitestaus. Ohjelmistojen testaus -kurssin luentokalvot. Helsingin Yliopisto, Tietojenkäsittelytieteen laitos. Viitattu 15.6.2022. https://www.cs.helsinki.fi/u/taina/ohte/s-2004/luennot/integrointitestaus.pdf.

Liitteet

Liite 1. login_page_test.spec.cy.js

```
describe('Test login page', () => {
  it('Wrong username', () => {
    cy.visit("/");
    cy.get('[data-cy="login-username"]').type('wrongUsername');
    cy.get('[data-cy="login-password"]').type(Cypress.env('userPassword'));
    cy.get('button[type="submit"]').click();
    cy.get('app-error.ng-tns-c240-2 > .mat-card').should('contain.text', 'Error
signing in, check username and password.');
  })
  it('Wrong password', () => {
    cy.visit("/");
    cy.get('[data-cy="login-username"]').type(Cypress.env('userName'));
    cy.get('[data-cy="login-password"]').type('password');
    cy.get('button[type="submit"]').click();
    cy.get('app-error.ng-tns-c240-2 > .mat-card').should('contain.text', 'Error
signing in, check username and password.');
  })
  it('Empty password', () => {
    cy.visit("/");
    cy.get('[data-cy="login-username"]').type(Cypress.env('userName'));
    cy.get('[data-cy="login-password"]').type(' ');
    cy.get('.login-form > .mat-focus-indicator').should('be.disabled');
  })
  it('Empty username', () => {
    cy.visit("/");
    cy.get('[data-cy="login-username"]').type(' ');
    cy.get('[data-cy="login-password"]').type(Cypress.env('userPassword'));
    cy.get('.login-form > .mat-focus-indicator').should('be.disabled');
  })
  it('Empty username and password', () => {
    cy.visit("/");
    cy.get('[data-cy="login-username"]').type(' ');
    cy.get('[data-cy="login-password"]').type(' ');
    cy.get('.login-form > .mat-focus-indicator').should('be.disabled');
  })
  it('Login and logout succesfully', () => {
    cy.login(Cypress.env('userName'), Cypress.env('userPassword'));
    cy.get('[data-cy="toolbar-username"] > .mat-icon').click();
    cy.getById('logout').click();
    cy.url().should("contain", "/[redacted]");
  })
})
```

Liite 2. energy_report.spec.cy.js

```
import data from "../../fixtures/energy_report_data.json";

describe("Check that load report has values", () => {
  before(() => {
    cy.login_with_API("fi");
  });

  beforeEach(() => {
    cy.visit("/[redacted]");
  });

  it("Test load report", () => {
    cy.visit("/[redacted]");
    cy.getById("datepicker-toggle-start")
      .first()
      .click()
      .then(() => {
        cy.get("[id^=mat-calendar-button]").click();
        cy.contains(data.date_range.start_year).click();
        cy.contains(data.date_range.start_month).click();
        cy.contains(data.date_range.start_day).click();
      });

    cy.contains("Energiaraportti").click();

    cy.getById("datepicker-toggle-end")
      .first()
      .click()
      .then(() => {
        cy.get("[id^=mat-calendar-button]").click();
        cy.contains(data.date_range.end_year).click();
        cy.contains(data.date_range.end_month).click();
        cy.contains(data.date_range.end_day).click();
      });
    cy.contains("Energiaraportti").click();
    cy.get("mat-action-list > button").contains("Käyttöpaikka").click();

    cy.get("app-report-filter")
      .contains("Käyttöpaikka")
      .closest("div")
      .within(() => {
        cy.getById("search-input-field").type("[redacted]");
      });
    cy.getById("filter-list").contains("[redacted]").click();
    cy.get("button").contains("Tee raportti").click();

    var list = data.load_code;
    for (let i in list) {
```

```
      cy.get("mat-tab-header").contains("Energiaraportti").click();
      cy.getById("data-row")
        .contains(i)
        .closest("tr")
        .within(() => {
          //cy.get('td').eq(11).contains('57,4')
          cy.get('[data-columnname="qnet_d"]')
            .invoke("text")
            .then((text) => {
              expect(text.replace(/\s/g, "")).equal(list[i].qnet_d);
            });
          cy.get('[data-columnname="net_weight_kg"]')
            .invoke("text")
            .then((text) => {
              expect(text.replace(/\s/g, "")).equal(list[i].net_weight_kg);
            });
          cy.get('[data-columnname="energy_mwh"]')
            .invoke("text")
            .then((text) => {
              expect(text.replace(/\s/g, "")).equal(list[i].energy_mwh);
            });
        });
    }
  });
});
```

Liite 3. load_report.spec.cy.js

```
import data from '../../fixtures/load_report_data.json'
describe("Check that load report has values", () => {
    before(() => {
        cy.login_with_API('fi');
    });
    beforeEach(() => {
        cy.visit("/[redacted]");
    });

    it("Test load report", () => {
        cy.visit('/[redacted]')
        cy.getById('datepicker-toggle-start').first().click()
            .then(() => {
                cy.get('[id^=mat-calendar-button]').click()
                cy.contains(data.date_range.start_year).click()
                cy.contains(data.date_range.start_month).click()
                cy.contains(data.date_range.start_day).click()
            })
        cy.getById('report-title').first().click()
        cy.getById('datepicker-toggle-end').first().click()
            .then(() => {
                cy.get('[id*="mat-calendar-button"]').click()
                cy.contains(data.date_range.end_year).click()
                cy.contains(data.date_range.end_month).click()
                cy.contains(data.date_range.end_day).click()
            })
        cy.getById('report-title').first().click()
        cy.get('button').contains('Tee raportti').click()

        var list = data.load_code
        for (let i in list) {
            cy.get('mat-tab-header').contains('Kuormaraportti').click()
            cy.getById('data-row').contains(i).closest('tr').within(() => {
                cy.get('[data-columnname="mar"]').invoke('text').then((text) => {
                    expect(text.replace(/\s/g, '')).equal(list[i].mar)
                })
                cy.get('[data-columnname="volume_m3"]')
                    .invoke('text').then((text) => {
                        expect(text.replace(/\s/g, '')).equal(list[i].volume_m3)
                    })
                cy.get('[data-columnname="net_weight_kg"]')
                   .invoke('text').then((text) => {
                    expect(text.replace(/\s/g, '')).equal(list[i].net_weight_kg)
                })
            })
        }
    });
});
```

Liite 4. main_menu.spec.cy.js

```
import data from '../../fixtures/main_menu_data.json'
describe("Check main menu items", () => {
    before(() => {
    });
    beforeEach(() => {
        cy.login_with_API('en');
        cy.visit("/");
    });

    it("Check main menu items exist for user " + Cypress.env("userName"), () => {
        cy.getById("menu-item")
            .each((e) => {
                expect(e.text()).to.be.oneOf(data.mainMenuList)
            });
    });

    it("Check sub-menu items of the user " + Cypress.env("userName"), () => {
        cy.getById("menu-item").parent().each((e) => {
            cy.get(e).then(() => {
                e.click()
                cy.getById('sub-menu-item-text')
                    .each((x) => {
                        expect(x.text()).is.oneOf(data.appMenuList)
                    });
            });
        });
    });
});
```

Liite 5. reservations_calendar.spec.cy.js

```
import data from '../../fixtures/reservations_calendar_data.json'
describe("Basic check for reservations calendar", () => {
    before(() => {
        cy.login_with_API('fi');
    });
    beforeEach(() => {
        cy.visit("/[redacted]");
    });
    it("Test reservations calendar from one date", () => {
        cy.visit('/[redacted]')
        cy.getById('mat-form-field-power-plant').within(() => {
            cy.wait(1500)
            cy.get('mat-select').trigger('click', { force: true })
        })
        cy.getById('mat-option-search-power-plant')
            .type('[redacted]').type('{enter}')
        cy.getById('mat-form-field-supplier').within(() => {
            cy.wait(750)
            cy.get('mat-select').trigger('click', { force: true })
        })
        cy.getById('mat-option-search-supplier').type('[redacted]').type('{enter}')
        cy.getById('mat-form-field-material-type').within(() => {
            cy.wait(750)
            cy.get('mat-select').trigger('click', { force: true })
        })
        cy.getById('mat-option-search-material-type')
            .type('Metsäpolttoaine').type('{enter}')
        cy.getById('mat-form-field-datepicker').within(() => {
            cy.get('mat-datepicker-toggle').trigger('click', { force: true })
        }).then(() => {
            cy.get('[id^=mat-calendar-button]').click()
            cy.contains(data.date.year).click()
            cy.contains(data.date.month).click()
            cy.contains(data.date.day).click()
        })
        cy.getById('calendar-view').within(() => {
            cy.contains('Kalenteriviikko').click()
        })
        var list = data.date.reservation
        for (let i in list) {
           cy.getById('calendar-event-cell')
            .contains(data.date.reservation[i])
            .should('have.text', data.date.reservation[i])
        }
    });
});
```

Liite 6. command.js

```
Cypress.Commands.add("login", (username, password) => {
    cy.visit("/");
    cy.get('[data-cy="login-username"]').type(username);
    cy.get('[data-cy="login-password"]').type(password);
    cy.get('button[type="submit"]').click();
    cy.get('[data-cy="toolbar-username"]').should(
      "contain",
      username
    );
    cy.url().should("contain", "[redacted]");
});

// Basic login command using API
Cypress.Commands.add("login_with_API", (language) => {
  cy.visit("/");
  cy.intercept("GET", `${Cypress.env("API_URL")}[redacted]`).as(
    "getUserModels"
  );

  cy.request("POST", `${Cypress.env("API_URL")}[redacted]`, {
    username: Cypress.env("userName"),
    password: Cypress.env("userPassword"),
  }).then((response) => {
    window.localStorage.setItem(
      "ngx-webstorage|token",
      '"' + response.body.token + '"'
    );
    window.localStorage.setItem(
      "ngx-webstorage|language",
      '"'+language+'"'
    );
  });
  cy.visit("/landing");
  cy.url().should("contain", "/[redacted]");
});

// Simple helper for data-cy selection
Cypress.Commands.add("getById", (id) => {
  cy.get(`[data-cy=${id}]`);
});
```