jamk

Kestävien Cypress E2E-testien edellytykset sovelluskehityksessä

Rauli Rolig

Opinnäytetyö, AMK
Toukokuu 2025
Tietojenkäsittelyn tutkinto-ohjelma

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Kuvailulehti

Rolig, Rauli

Kestävien Cypress E2E-testien edellytykset sovelluskehityksessä

Jyväskylä: Jyväskylän ammattikorkeakoulu. Toukokuu 2025, 37 sivua

Tietojenkäsittelyn tutkinto-ohjelma. Opinnäytetyö AMK.

Julkaisun kieli: suomi

Julkaisulupa avoimessa verkossa: kyllä

Tiivistelmä

Web-sovellusten kehityksessä automaattinen testaaminen on keskeisessä osassa laadunvarmistusta. End-to-end-testauksen (E2E) avulla voidaan varmistaa, että sovellus toimii loppukäyttäjän näkökulmasta. Cypress on yksi suosituimmista E2E-testauksen työkaluista, mutta sen tehokas hyödyntäminen vaatii suunnittelua, jotta testit säilyvät vakaina ja ylläpidettävissä sovelluksen muutosten mukana.

Tutkimus toteutettiin kehittämistyönä, jonka toimeksiantajana oli Quux Oy. Yrityksessä tunnistettiin toistuvia ongelmia testien kestävyydessä, joiden perusteella aineisto kerättiin testaajan työympäristössä osallistuvan havainnoinnin keinoin. Tarkastelun kohteena olivat muun muassa selektorien käyttö, ajoitusten hallinta, testidatan hallinta ja toistuvan koodin uudelleenkäytettävyys.

Kehittämistyön tuloksena laadittiin ohjeistus, jonka avulla testien luotettavuutta voitiin parantaa huomattavasti. Selektorien vakiointi, dynaamisten odotusten hyödyntäminen sekä testikoodin modulaarisuus vähensivät testien rikkoutumista ja paransivat ylläpidettävyyttä. Ohjeistus sovellettiin käytäntöön osaksi testejä, mikä vahvisti sen toimivuutta.

Johtopäätöksenä havaittiin, että teknisten ratkaisujen avulla testaaja voi lisätä testien vakautta, mutta paras lopputulos saavutetaan yhteistyössä kehittäjien kanssa. Yhteisesti sovitut käytännöt ja ennakoiva testattavuuden huomiointi kehitystyössä parantavat testauksen laatua ja vähentävät ylläpidon määrää pitkällä aikavälillä.

Avainsanat (asiasanat)

Cypress, ohjelmistotestaus, testauskäytännöt, end-to-end-testaus

Muut tiedot (salassa pidettävät liitteet)

Ei salassa pidettäviä liitteitä.

jamk

Description

Rolig, Rauli

Requirements for sustainable Cypress E2E testing in software development

Jyväskylä: JAMK University of Applied Sciences, May 2025, 37 pages

Degree Programme in Energy and Environmental Technology. Bachelor´s thesis.

Permission for open access publication: Yes

Language of publication: Finnish

Abstract

Automated testing plays a key role in quality assurance in web application development. End-to-end (E2E) testing ensures that the application works as expected from the end-user's perspective. Cypress is one of the most popular tools for E2E testing, but its effective use requires careful planning to keep tests stable and maintainable as the application evolves.

The study was carried out as a development project commissioned by Quux Oy. The company had identified recurring issues with test reliability, and the material was collected through participatory observation in the tester's work environment. The focus areas included selector usage, timing control, test data management, and reusability of repetitive code.

As a result of the development process, a guide was created to significantly improve the reliability of tests. Standardized selectors, dynamic wait conditions, and modular test code reduced test failures and improved maintainability. The guide was applied in practice as part of the test implementation, confirming its effectiveness.

In conclusion, it was observed that testers can improve test stability through technical solutions, but the best results are achieved in collaboration with developers. Agreed practices and proactive consideration of testability during development enhance test quality and reduce maintenance needs in the long term.

Keywords/tags (subjects)

Cypress, software testing, testing practices, end-to-end testing

Miscellaneous (Confidential information)

No confidential infromation

Sisältö

Kuviot

1 Johdanto

Ohjelmistokehityksessä laadunvarmistus ei ole enää vain viimeinen tarkistus ennen julkaisua, vaan osa jatkuvaa kehitysprosessia. Automaattinen testaaminen on noussut keskeiseen asemaan erityisesti nykyaikaisten web-sovellusten kehityksessä. Kun ohjelmistoja rakennetaan ja päivitetään jatkuvasti pieninä kokonaisuuksina, testauksen rooli muuttuu jatkuvasti enemmän ennakoivaksi laadun varmistajaksi. End-to-end (E2E) -testaus jäljittelee loppukäyttäjää sovelluksessa varmistaakseen, että se toimii virheettömästi todellisia käyttötilanteita vastaavissa olosuhteissa.

Yksi suosituimmista työkaluista E2E-testaukseen on Cypress, joka mahdollistaa selaimessa tapahtuvan nopean ja luotettavan testauksen. Tässä opinnäytetyössä tutkitaan, miten Cypressin avulla voidaan rakentaa end-to-end-testejä, jotka kestävät käyttöliittymä muutoksia ja ovat helpommin ylläpidettäviä. Tavoitteena on kehittää ohjeistus, joka tukee tasalaatuisia ja kestäviä testaus käytäntöjä.

Tutkimus on toteutettu toimeksiantona Quux Oy:lle, joka on ohjelmistokehityksen ja digitaalisen liiketoiminnan asiantuntijayritys. Yrityksen kehitystyössä korostuvat asiakaskohtaiset ratkaisut, joiden käyttöliittymät ja toiminnallisuudet elävät projektien mukaan. Tällaisessa ympäristössä testien ylläpidettävyys ja kestävyys käyttöliittymä muutoksille vaatii erityistä huomiota. Jo pienet visuaaliset tai rakenteelliset muutokset voivat aiheuttaa testien hajoamista, mikä heikentää testauksen luotettavuutta.

Työn lähtökohtana on tarve ymmärtää, miten testien tekninen suunnittelu, kehitystiimin yhteistyö ja sisäiset käytännöt vaikuttavat testauksen kestävyyteen ja laatuun. Näitä näkökulmia yhdistämällä pyritään tukemaan automaattisen testauksen asemaa osana ohjelmistokehityksen kokonaisprosessia. Aihe on minulle myös henkilökohtaisesti merkittävä, sillä toimin yrityksen kehitystiimissä testaajana. Työssäni olen kohdannut toistuvia haasteita erityisesti testien kestävyydessä käyttöliittymän jatkuvan muutoksen takia, joka herätti tarpeen etsiä luotettavampia ja ylläpidettävämpiä ratkaisuja E2E-testaukseen.

E2E-testauksen laadulla ja kestävyydellä on tärkeä rooli myös kestävän kehityksen näkökulmasta. Kun testit kestävät käyttöliittymän muutoksia ja vaativat vähemmän ylläpitoa, voidaan vähentää

ylimääräistä työtä ja ehkäistä tuotantokatkoksia. Tämä tehostaa ajankäyttöä ja teknisten resurssien hyödyntämistä. Pitkällä aikavälillä kestävä testausprosessi tukee ohjelmistokehityksen jatkuvuutta ja vastuullista toimintaa.

2 Tutkimusasetelma

2.1 Tutkimuksen tavoite, rajaus ja tehtävät

Tämän opinnäytetyön tavoitteena on laatia ohjeistus kestävästä E2E-testauksesta Cypressin avulla. Ohjeistus on suunnattu testaajille ja ohjelmistokehittäjille, ja sen tarkoituksena on vahvistaa testauksen luotettavuutta, selkeyttä ja ylläpidettävyyttä osana yrityksen projektin ohjelmistokehitysprosessia. Työ pyrkii vastaamaan käytännön tarpeeseen kehittää testaus käytäntöjä, jotka tukevat web-sovellusten kehitystä muuttuvassa ympäristössä. E2E-testauksen kestävyyttä lähestytään teknisten ratkaisujen lisäksi myös testauksen käytäntöjen ja kehitystiimien yhteistyön näkökulmasta.

Opinnäytetyön konkreettisena tavoitteena on tunnistaa ja analysoida niitä ilmiöitä ja teknisiä käytäntöjä, jotka vaikuttavat E2E-testien hajoamiseen tai epäluotettavuuteen sovelluksen elinkaaren aikana. Näiden pohjalta tuotetaan selkeä, sovellettava ja käytännönläheinen ohjeistus, joka tukee testaajia testien kirjoittamisessa, päivittämisessä ja ylläpidossa. Ohjeistus toimii samalla osana laajempaa laadunvarmistusta, jolla pyritään lisäämään yrityksen automaattisen testauksen roolia.

Tutkimus rajautuu web-sovellusten frontendin E2E-testaukseen Cypress-testikirjastoa käyttäen. Vaikka työssä käsitellään myös muita testausmenetelmiä ja -kirjastoja vertailun ja taustan näkökulmasta, kuten Playwrightia ja Seleniumia, niiden tekninen käyttö ei kuulu työn varsinaiseen sisältöön. Lisäksi työ rajattiin niihin teknisiin ratkaisuihin, jotka ovat suoraan hallittavissa testaajan roolilla.

2.2 Kehittämistehtävät

End-to-end-testauksen toteuttaminen muuttuvissa web-sovelluksissa on haastavaa, sillä muutokset rikkovat helposti testejä ja heikentävät niiden ylläpidettävyyttä. Vaikka Cypress tarjoaa moder-

neja ja kehittyneitä ominaisuuksia, sen käyttö ei yksinään takaa testien kestävyyttä. Tämän kehittämistyön lähtökohtana oli tunnistaa ja ratkoa ongelmakohtia, joihin ohjeistuksen avulla voidaan vaikuttaa.

Kehittämistehtävät muotoiltu kysymysmuotoon:

1. Mitkä ovat yleisimmät syyt siihen, että Cypress-testit rikkoutuvat sovellus muutosten yhteydessä?
2. Miten testikoodia voidaan kirjoittaa siten, että se kestää sovelluksen muutoksia ja säilyy selkeänä sekä ylläpidettävänä?
3. Miten sovelluskehityksessä tehtävät ratkaisut voivat tukea end-to-end-testien vakautta ja ennustettavuutta?

2.3 Tutkimusmenetelmä

Tämän opinnäytetyön tutkimusmenetelmänä käytettiin tutkimuksellista kehittämistyötä, joka soveltuu erityisesti työelämästä nousevien käytännön ongelmien ratkaisemiseen. Kehittämistyön taustalla on toimeksiantajan havainto siitä, että end-to-end-testauksen kestävyydessä on puutteita, jotka hankaloittavat testiautomaation hyödyntämistä muuttuvassa sovellusympäristössä. Kehittämistyölle on yleistä, että siinä luodaan osittain tai kokonaan uusi tuote, joka voi olla esimerkiksi ohjelmisto, tietojärjestelmäratkaisu tai muu vastaava (Ojasalo, Moilanen & Ritalahti 2015, 19).

Tutkimuksellinen kehittämistyö etenee vaiheittain, ja sen toteuttamiseen liittyy järjestelmällinen, analyyttinen ja kriittinen lähestymistapa. Vaikka kehittämisprosessi voidaan jäsentää eri vaiheisiin, ne eivät käytännössä aina erotu selkeästi toisistaan, vaan työskentely voi edellyttää joustavaa siirtymistä vaiheesta toiseen. Kehittämistyön tavoitteena on aina konkreettinen lopputulos tai muutos, kuten uusi tuote tai menetelmän parantaminen. (Ojasalo ym. 2015, 23–24.)

Tutkimusaineisto perustuu työelämässä kertyneeseen kokemukseen ja testaus prosessin aikana syntyneisiin havaintoihin. Tällainen aineisto on Güntherin ja Hasasen (n.d.) määritelmän mukaan luonnollista aineistoa, jota tutkija ei ole tuottanut itse tutkimusta varten, vaan se on syntynyt osana normaalia toimintaa. Havaintoaineistoa analysoidaan kehittämistyön tueksi, jotta nykytila ja siihen liittyvät ongelmat voidaan hahmottaa.

Opinnäytetyössä kehittämistyö eteni käytännössä seuraavasti. Ensin analysoitiin testauksen nykytilaa osallistuvan havainnoinnin keinoin testaajan omassa työssä. Havaintojen perusteella tunnistettiin toistuvat ongelmat, kuten selektorien haavoittuvuus, ajoitusongelmat, testidatan hallinta ja koodin toistuvuus. Näihin etsittiin ratkaisuja hyödyntämällä ajankohtaista dokumentaatiota, testauskirjallisuutta ja alan käytäntöjä. Ratkaisujen pohjalta laadittiin ohjeistus kestävän E2E-testauksen tueksi, jota sovellettiin käytännössä testikoodiin. Soveltamisen vaikutuksia arvioitiin erityisesti testien vakauden näkökulmasta. Työn ulkopuolelle jäi kuitenkin ohjeistuksen pitkäaikainen vaikutusten arviointi sovellusmuutosten yhteydessä, sillä varsinaiseen tuotantosovellukseen ei voitu tehdä tarkoituksellisia muutoksia tätä työtä varten.

2.4 Tiedonhaun kuvaus

Tiedonhaku toteutettiin hyödyntämällä valikoituja tieteellisiä julkaisuja ja asiantuntijoiden blogikirjoituksia ja käytännön sovelluskehityksestä saatuja kokemuksia. Koska opinnäytetyö käsittelee end-to-end-testauksen kestävyyttä Cypress-ympäristössä, tiedonkeruu kohdistui erityisesti testiautomaation vakauteen, ylläpidettävyyteen ja kehitystyön käytäntöihin.

Hakusanoina käytettiin muun muassa E2E testing, test automation, Cypress testing best practices, testing best practices ja frontend testing. Näiden avulla löydettiin julkaisuja, jotka käsittelivät testauksen teknisiä ratkaisuja ja käytännön toimintatapoja. Tieteelliset lähteet muodostavat työn teoreettisen perustan.

Käytännön näkökulma perustui tekijän omaan kokemukseen testiautomaation toteuttamisesta sekä olemassa olevan testikoodin tarkasteluun. Tämä tarjosi olennaista ymmärrystä siitä, millaisia haasteita E2E-testauksessa kohdataan käytännössä ja mitkä tekijät vaikuttavat testikoodin kestävyyteen.

Lähteiden valinnassa painotettiin luotettavuutta, ajankohtaisuutta ja aiheeseen soveltuvuutta. Erityisesti 2020-luvun julkaisut sekä nykyiset työkalun käytännöt ja suositukset olivat tiedonhaun kohteena.

3 Web-sovellus

End-to-end-testaus kohdistuu sovelluksen toimintaan käyttäjän näkökulmasta, ja siksi on tärkeää ymmärtää, miten sovellus on teknisesti rakennettu. Web-sovelluksissa käyttäjän toiminnot tapahtuvat frontendissä, mutta niillä on usein vaikutuksia backendissä ja sitä kautta koko järjestelmän toimintaan. DOM-rakenne taas muodostaa teknisen pohjan käyttöliittymän elementeille, joita automatisoidut testit käsittelevät.

3.1 Frontend ja backend

Ohjelmistokehityksessä frontend ja backend kuvaavat sovelluksen kahta olennaista osiota, joilla on toisistaan erilaiset, mutta läheiset roolit. Frontend on sovelluksen käyttäjälle näkyvää osaa, johon kuuluu käyttöliittymä (UI). Siinä toteutuu kaikki ne visuaaliset ja vuorovaikutteiset elementit, joiden kanssa käyttäjä toimii. Näihin kuuluvat esimerkiksi valikot, painikkeet, lomakkeet ja tekstisisältö, jotka rakennetaan teknologioilla kuten HTML, CSS ja JavaScript. (Gallinelli 2021)

Backendillä tarkoitetaan puolestaan sovelluksen näkymätöntä osaa, joka huolehtii liiketoiminta logiikasta, tietojen käsittelystä ja tallennuksesta sekä yhteyksistä ulkoisiin palveluihin. Kun käyttäjä tekee toimenpiteen frontendissä, lähetetään siitä pyyntö backendille, joka prosessoi sen, tekee tarvittavat tietokantaoperaatiot ja palauttaa vastauksen käyttöliittymälle. Backend rakentuu yleensä ohjelmointikielistä kuten Python, Ruby tai Java, ja hyödyntää tietokantajärjestelmiä kuten SQL tai NoSQL. (What’s the Difference Between Frontend and Backend in Application Development? n.d.)

Esimerkiksi käyttäjän kirjautuessa sovellukseen frontendin kautta syötetään tunnukset lomakkeelle, jonka jälkeen tiedot välitetään backendille tarkistettavaksi. Testauksessa tarkastetaan, että oikea pyyntö lähtee, backend palauttaa onnistumisviestin ja käyttäjälle avautuu oikea näkymä. E2E-testi simuloi tämän käyttäjäpolun alusta loppuun.

3.2 Document Object Model (DOM)

Document Object Model eli DOM on keskeinen käsite web-kehityksessä. DOM on verkkosivun hierarkkinen esitystapa, joka mahdollistaa dynaamisen vuorovaikutuksen verkkosivujen sisällön

kanssa. Kun selain lataa verkkosivun, se käsittelee HTML:n, CSS:n ja JavaScriptin, ja muuttaa ne puumaiseksi rakenteeksi, joka kuvaa dokumentin kaikki osat, kuten elementit ja attribuutit. DOM mahdollistaa sen, että JavaScript voi dynaamisesti muokata verkkosivun elementtejä, päivittää sisältöä, käsitellä käyttäjän toimintoja ja soveltaa visuaalisia muutoksia reaaliaikaisesti. (What is the DOM (Document Object Model) in Web Development? 2025)

Cypress hyödyntää DOM-rakennetta etsiessään testattavia elementtejä. Esimerkiksi komento cy.get('["#login-button"]') hakee DOM-puusta id tunnisteen perusteella painikkeen, jota käyttäjä normaalisti painaisi. Mikäli DOM muuttuu, joten testi ei huomioi tätä, testi epäonnistuu, vaikka sovellus toimisi odotetusti.

4 Sovellustestaus

4.1 Testauksen suunnittelu

Testisuunnitelma on dokumentti, joka määrittelee testauksen laajuuden, lähestymistavan, resurssit, työkalut, aikataulut ja ympäristön. Hyvin laadittu suunnitelma varmistaa kattavan testauksen, vähentää riskejä ja tukee tavoitteita liiketoiminnassa. Tärkeitä osia ovat muun muassa testauksen rajaukset, vastuuhenkilöt, poistumiskriteerit ja virheiden hallinta. (Son 2024)

Testauksen suunnittelu rakentuu vaiheittain etenevästä prosessista. Ensimmäiseksi määritellään julkaistavan version testauksen laajuus ja aikataulu. Tämän jälkeen määritetään testauksen tavoitteet, kuten virheiden havaitseminen. Tuotokset voivat sisältää muun muassa testisuunnitelman, testitapaukset, virheraportit ja testilokit. Testausstrategiassa määritetään testitasot esim. yksikkö- tai integraatiotestaus, testityypit esim. manuaalinen tai automatisoitu, sekä tarvittava ympäristö ja testidata. (Son 2025)

Testit voidaan suunnitella eri näkökulmista, kuten käyttäjäpolkujen, ominaisuuksien, lokien tai poikkeustilanteiden perusteella. Esimerkiksi SFDIPOT-menetelmä (structure, function, data, interfaces, platform, operations, time) huomioi rakenteen, toiminnan, datan, rajapinnat, alustan, toiminnallisuuden ja ajan testauksen suunnittelussa. Näiden avulla voidaan varmistaa, että testaus on kattavaa ja kohdistus osuu kriittisiin kohtiin. (Son 2024)

4.2 Manuaalitestaus

Manuaalinen testaus on ohjelmistojen laadunvarmistukseen kuuluva menetelmä, jossa testaaja käy läpi sovelluksen toimintoja itse ilman automatisoituja testaustyökaluja. Tarkoituksena on löytää ohjelmistosta virheitä vertaamalla sen toimintaa määriteltyihin vaatimuksiin tai käyttäjän odotuksiin. (Nguyen 2025; Berga 2024a)

Manuaalisen testauksen vahvuus on sen joustavuus. Testaaja voi muuttaa lähestymistapaa havaintojensa perusteella, mikä mahdollistaa testien suuntaamisen uusiin ongelma-alueisiin nopeasti. Lisäksi ihmisen kyky huomata visuaalisia virheitä tai käytettävyyteen liittyviä poikkeuksia on sellainen, jota automaattiset työkalut eivät voi korvata. (Berga 2024a; Nguyen 2025)

Toisaalta manuaalisessa testauksessa on myös haasteita. Koska kaikki testausvaiheet suoritetaan käsin, testaus vie usein paljon aikaa ja resursseja. Toistuvien testien suorittaminen manuaalisesti on työlästä, ja se voi johtaa siihen, että eri testaajat saavat hieman erilaisia tuloksia samasta testistä. (Nguyen 2025)

Parhaassa tapauksessa ohjelmistotestaus yhdistää sekä manuaalisen että automatisoidun lähestymistavan. Manuaalinen testaus täydentää automatisoitua testausta tarjoamalla mahdollisuuden arvioida käytettävyyttä ja monimutkaisia toimintoja. (Berga 2024a)

4.3 Automaatiotestaus

Automaatiotestaus on ohjelmistotestauksen menetelmä, jossa testitapaukset suoritetaan automaattisesti hyödyntäen ohjelmistotyökaluja ja testiskriptejä. Sen tavoitteena on lisätä testauksen tehokkuutta, vähentää manuaalista työtä ja parantaa testien tarkkuutta. Automaatiotestauksessa voidaan esimerkiksi simuloida käyttäjän toimintoja, kuten painikkeiden painamista, tietojen syöttämistä ja sivujen välillä siirtymistä. Tämän ansiosta testit voidaan suorittaa nopeasti, toistettavasti ja rinnakkain useissa käyttöympäristöissä ja eri datalla. Menetelmä soveltuu erityisesti sellaisiin toistuviin ja ennakoitaviin testitapauksiin kuten regressiotestaukseen. Hyvin suunnitellut testit voidaan myös uudelleenkäyttää ja integroida osaksi ohjelmistokehityksen jatkuvaa testausprosessia. (GAT Staff Writers 2024)

4.4 Yksikkötestaus

Yksikkötestaus on ohjelmistotestauksen menetelmä, jossa tarkastellaan ohjelmiston pienimpiä testattavia yksiköitä, kuten yksittäisiä funktioita tai metodeja täysin erillään muusta järjestelmästä. Testauksen tavoitteena on varmistaa, että yksiköt toimivat oikein valmiiksi määrätyillä syötteillä ja palauttavat odotettuja tuloksia. Testit suoritetaan automaattisesti aina koodi muutosten yhteydessä, joten ne mahdollistavat nopean virheiden löytämisen. (What is Unit Testing? n.d.)

Yksikkötestauksen vahvuuksia ovat sen nopeus, eristäminen ja toistettavuus. Testit suoritetaan ilman ulkoisia riippuvuuksia, ja niiden yhteydessä käytetään usein yksinkertaistettuja korvaavia rakenteita, jotka jäljittelevät muita järjestelmän osia ja palauttavat ennalta määriteltyjä arvoja. Tämä tekee testeistä ennustettavia ja luotettavia. (da Costa 2021, luku 2; Berga 2024b) Hyvin suunnitellut yksikkötestit tukevat myös turvallista koodin muokkausta ja toimivat dokumentaationa, sillä ne kertovat, miten minkäkin yksikön pitäisi toimia (Berga 2024b).

4.5 Integraatiotestaus

Integraatiotestaus on ohjelmistotestauksen menetelmä, jossa testataan eri moduulien tai komponenttien välistä yhteistoimintaa osana laajempaa järjestelmää. Tavoitteena on varmistaa, että ohjelmiston osat toimivat yhdessä oikein, ja että tiedonsiirto, API-kutsut ja moduulien välinen kommunikaatio toimivat suunnitellusti. Testaus auttaa tunnistamaan esimerkiksi käyttöliittymän, taustajärjestelmän ja tietokannan välisiä yhteensopivuusongelmia sekä virheitä ulkoisten palveluiden integroinnissa. (What is Integration Testing 2025)

Yleisesti käytettyjä lähestymistapoja ovat Big Bang -testaus, jossa kaikki moduulit yhdistetään ja testataan samanaikaisesti, sekä inkrementaalinen testaus, jossa moduulit integroidaan vaiheittain. Inkrementaalinen testaus voidaan toteuttaa ylhäältä alas, alhaalta ylös tai hybridi mallina, ja se mahdollistaa virheiden varhaisen havaitsemisen ja helpottaa vian paikantamista. (Katalon Team 2025)

Testipyramidissa integraatiotestit sijoittuvat yksikkötestien ja end-to-end-testien väliin. Ne auttavat tunnistamaan ongelmia, joita ei yksittäisiä komponentteja testattaessa havaita, mutta jotka voivat aiheuttaa järjestelmätason vikoja. (What is Integration Testing 2025)

4.6 End-to-end-testaus (E2E)

End-to-end-testaus (E2E) on ohjelmistotestauksen menetelmä, jossa sovellusta testataan, kuten käyttäjä sen kokisi, eli yleisesti käyttöliittymän kautta. Tarkoituksena on varmistaa, että kaikki järjestelmän osat toimivat yhdessä oikein ja että järjestelmä käyttäytyy odotetulla tavalla todellisissa käyttötilanteissa. E2E-testeissä simuloidaan käyttäjän toimintaa sovelluksessa ilman, että testissä hyödynnetään tietoa sen sisäisestä toiminnasta. (da Costa 2021, luku 10) Toisin kuin yksikkö- ja integraatiotestit, E2E-testit paljastavat virheitä, jotka ilmenevät koko järjestelmän toiminnasta käyttäjän näkökulmasta (da Costa 2021, luku 2).

E2E-testauksessa simuloidaan käyttäjän todellista käyttökokemusta erilaisissa tilanteissa. Esimerkiksi rekisteröitymislomakkeen testaaminen, johon kuuluu vaiheita kuten käyttäjätietojen syöttäminen, lomakkeen lähettäminen ja järjestelmän reagoinnin tarkistaminen, kuten käyttäjän tallentaminen tietokantaan ja vahvistussähköpostin lähettäminen. Näin E2E-testit varmistetaan, että järjestelmän kaikki komponentit, kuten frontend, backend ja tietokannat, toimivat saumattomasti yhdessä. (Schmitt 2024)

E2E-testauksen hyödyt voivat olla merkittäviä. Sillä voidaan parantaa sovelluksen laatua havaitsemalla virheet ennen kuin ne saavuttavat käyttäjän. Sillä myös vähennetään ohjelmiston julkaisun riskejä, sillä se auttaa tunnistamaan ja korjaamaan ongelmat kriittisissä käyttäjäpoluissa. Lisäksi E2E-testauksen avulla voidaan varmistaa, että järjestelmän suorituskyky vastaa käyttäjien odotuksia, jolla voidaan lisätä luottamusta ohjelmistoon ja parantaa käyttäjäkokemusta. (Dilmegani 2025)

Vaikka E2E-testaus tarjoaa merkittäviä etuja, se tuo mukanaan myös haasteita. Testien suunnittelu ja ylläpito voivat olla aikaa vieviä, koska ne edellyttävät syvällistä ymmärrystä sovelluksesta ja sen käyttäjien odotuksista. Lisäksi testien ajaminen voi olla hidasta ja resursseja kuluttavaa, sillä ne simuloivat kokonaisia käyttöskenaarioita. Näiden takia E2E-testit yleensä kohdistetaan sovelluksen tärkeimpiin toimintoihin ja käyttäjäpolkuihin. (da Costa 2021, luku 10)

4.7 E2E-työkalut

Selenium (2004), Cypress (2014) ja Playwright (2019) ovat yhdet kolme suosituinta selainautomaation testauskehystä. Selenium on pitkään ollut markkinoiden laajimmin käytetty kehys erityisesti moniselaintuen ja monikielisen tuen ansiosta. Cypress on noussut suosituksi yksinkertaisuutensa ja nopean kehityssyklinsä ansiosta, kun taas Playwright on saanut nopeasti asemaa modernina vaihtoehtona, joka yhdistää laajan selaintuen ja tehokkaat ominaisuudet. (Top Test Automation Frameworks in 2025 2024) Esitettyjen vertailtavien kehysten valinta perustuu niiden yleiseen käyttöön, yhteisötukeen ja soveltuvuuteen testauksessa, mutta tärkeimpänä tutkittavana on Cypress.

Selenium

Selenium on 2004 vuodesta alkaen kehitetty selainautomaatiotyökalu, joka mahdollistaa web-sovellusten testaamisen eri selaimilla, käyttöjärjestelmillä ja ohjelmointikielillä. Se tukee muun muassa Javaa, Pythonia, JavaScriptiä ja C#:a, ja sen voi integroida yleisiin testaus- ja CI/CD-kehyksiin, kuten JUnit ja Jenkins. (What is Selenium? A Complete Introduction 2023)

Selenium koostuu kolmesta pääosasta, joita ovat WebDriverista, Gridistä ja IDE:stä. WebDriver on tärkein komponentti. Se ohjaa selainta selainkohtaisen driverin kautta, mutta ei sisällä testi logiikkaa tai raportointia, vaan tarvitsee tuekseen testikehyksen. (Selenium Documentation: Components Overview 2022) Grid mahdollistaa testien ajamisen rinnakkain eri ympäristöissä, kun taas IDE auttaa luomaan testejä ilman koodia nauhoittamalla käyttäjän toimia selaimessa (What is Selenium? A Complete Introduction 2023).

Selenium soveltuu useisiin testityyppeihin, kuten toiminnalliseen testaukseen, regressiotestaukseen ja end-to-end-testaamiseen, ja se on laajasti käytetty työkalu testausautomaatiossa (What is Selenium? A Complete Introduction 2023).

Playwright

Playwright on Microsoftin kehittämä selainautomaatiotyökalu, joka mahdollistaa modernien web-sovellusten end-to-end-testauksen. Se tukee Chromium-, WebKit- ja Firefox-selainmoottoreita ja

toimii eri käyttöjärjestelmissä. Playwrightia voi käyttää muun muassa JavaScriptillä, TypeScriptillä, Pythonilla, Javalla ja .NET:llä. (What Is Playwright: A Tutorial on How to Use Playwright 2024.)

Testien vakauden parantamiseksi Playwrightilla on automaattinen odotus ominaisuus, joka varmistaa, että elementit ovat valmiita ennen etenemistä. Tämä vähentää testien epäonnistumisia ja poistaa välttämättömyyden manuaalisille odotuksille. (Playwright enables reliable end-to-end testing for modern web apps n.d.)

Testit suoritetaan omissa selain konteksteissaan, joissa ei jaeta istuntoja tai välimuisteja. Tämä varmistaa testien eristyksestä ja mahdollistaa rinnakkain testien ajon. (Playwright enables reliable end-to-end testing for modern web apps n.d.)

Playwrightilla on työkaluja, kuten Codegen ja Trace Viewer. Codegen tallentaa käyttäjän toiminnot selaimesta koodiksi, ja Trace Viewer esittää epäonnistuneiden testien kulun visuaalisesti muun muassa testin etenemis tallenteena ja reaaliaikaisia DOM-tilannekuvia. (Playwright enables reliable end-to-end testing for modern web apps n.d.)

Cypress

Cypress on myös moderni selainautomaatiotyökalu, joka on suunniteltu erityisesti nykyaikaisten web-sovellusten testaamiseen. Se toimii suoraan selaimessa ja mahdollistaa end-to-end- ja komponenttitestauksen yhdellä työkalulla. (Test. Automate. Accelerate. n.d.)

Cypressin arkkitehtuuri eroaa perinteisistä työkaluista, kuten Seleniumista. Se ei käytä WebDriveriä, vaan suorittaa testit samassa tapahtumasilmukassa kuin itse sovellus. Tämä mahdollistaa suoran pääsyn DOM-elementteihin ja tarjoaa vakaan ja nopean testauksen ilman verkkoviiveitä. (How Cypress Works n.d.)

Yksi Cypressin olennaisista ominaisuuksista on automaattinen odotus, jonka avulla ei tarvita välttämättömiä erillisiä keinotekoisia viiveitä. Cypress odottaa automaattisesti, että elementit ovat näkyvissä ja valmiita ennen seuraavan komennon suorittamista. (Why Cypress 2025) Odotuksia voi

myös säätää erikseen timeout-parametrilla, esimerkiksi cy.get('[data-testid="element"]', {timeout:10000}), jolloin testi odottaa määritellyn ajan ennen virheilmoitusta (Introduction to Cypress App 2025).

Testien kirjoittamista helpottavat esimerkiksi alias-mekanismi, jolla voidaan nimetä API-kutsuja tai muita toimintoja ja odottaa niitä myöhemmin testissä, käyttäen komentoja kuten cy.intercept() ja cy.wait('@alias') (Variables and Aliases 2025). Cypress tukee myös omien komentojen luomista, joita voidaan tallentaa commands-tiedostoon ja käyttää uudelleen eri testeissä (Custom Commands 2025). Lisäksi cy.session() mahdollistaa käyttäjäsession tallentamisen testien välillä, jolloin samoja kirjautumisvaiheita ei tarvitse toistaa jokaisessa testissä (Session 2025).

Cypress tarjoaa myös tehokkaat debuggaus ominaisuudet. Testin kulkua voidaan tarkastella vaiheittain time travel -toiminnon avulla. Testien ajon aikana voidaan käyttää myös selaimen kehitystyökaluja. Lisäksi kaikista tapahtumista otetaan tilannekuva, joka helpottaa virheiden löytämistä. (Why Cypress 2025)

Cypressin lisäpalveluita ovat muun muassa Cypress Cloud, jonka avulla testisuorituksia voidaan tallentaa ja toistaa ja UI Coverage -toiminto, joka tarjoaa visuaalisen näkymän sovelluksen testikattavuudesta. Lisäksi Cypress Accessibility on ratkaisu saavutettavuustestaukseen. (Why Cypress 2025)

Cypress isoimpia rajoitteita ovat, että se ei tue usean selaimen rinnakkaista hallintaa yksittäisessä testissä. Lisäksi tuki rajoittuu moderneihin selaimiin, kuten Chromeen, Firefoxiin ja Edgeen, kun taas Safari ja Internet Explorer eivät ole täysin tuettuja. (Raji 2024)

5 Toteutus

5.1 Aineistonkeruu ja tutkijan rooli

Aineistoa keräsin omassa työympäristössäni, jossa toimin testaajana ohjelmistokehitystiimissä. Koska yrityksessä ei ole käytössä dokumentoituja testauskäytäntöjä, tiedonkeruuni perustui osallistuvaan havainnointiin, jossa analysoin testauksen toteutumista omassa työssäni.

Kirjallista aineistoa hain erityisesti Cypressin virallisesta dokumentaatiosta, jota hyödynsin työn teknisen taustan rakentamiseen. Dokumentaatio tarjosi ajantasaisia ohjeita muun muassa testien rakenteeseen, ajoituksen hallintaan ja selektori käytäntöihin liittyen. Sen lisäksi käytin myös muita alan lähteitä, jotka täydensivät ymmärrystäni end-to-end-testauksen käytännöistä.

Kaikkia työssä esiintyviä ongelmia en kuitenkaan ratkaissut pelkästään kirjallisten lähteiden avulla. Tällöin ratkaisut perustuivat omaan asiantuntemukseeni osana testausprosessia. Näin muodostui aineisto, jossa yhdistyivät dokumentoitu tieto, alan yleinen tietämys sekä arjen kokemuksellinen ymmärrys. Näitä käytin vastatakseni tutkimuskysymyksiin ja kehittääkseni ohjeistusta, joka tukee kestävien E2E-testien toteutusta Cypress-ympäristössä.

5.2 Havainnot testauksen nykytilasta

Ennen ohjeistuksen laatimista oli tarpeen selventää testauksen nykytilaa organisaatiossa. Tarkastelu keskittyi tunnistamaan ne ongelmat, jotka heikentävät testien kestävyyttä ja vaikeuttavat automaation jatkuvaa hyödyntämistä. Yrityksessä ei ole käytössä kirjallista ohjeistusta testaukseen, eikä testaus ole vielä täysin sulautunut kehitystiimin yhteisiin toimintamalleihin.

Ongelmat havaittiin osallistuvan havainnoinnin keinoin, tutkijan päivittäisessä työssä testaajana. Havainnointi toteutettiin järjestelmällisesti kiinnittämällä huomiota toistuviin testien hajoamisiin, koodikäytäntöihin ja yhteistyön toimivuuteen testauksen näkökulmasta. Havainnot kirjattiin työn edetessä muistiin ja niitä verrattiin kirjallisuudessa esitettyihin testauksen parhaita käytäntöjä koskeviin suosituksiin. Ongelmat luokiteltiin niiden esiintyvyyden ja vaikutusten perusteella, ja valikoiduista havainnoista muodostui pohja ohjeistuksen kehittämiselle.

Selektorien puuttuminen ja kieliriippuvuus

Yksi testien jatkuva rikkoutumisen syy liittyi vakioitujen selektorien puuttumiseen. Sovelluksessa ei ole yleisesti käytössä data-testid-attribuutteja tai muita vastaavia tunnisteita testaukseen. Tämän takia testien kirjoittaminen vaatii usein DOM-rakenteessa liikkumista esimerkiksi CSS-luokkien tai tekstisisällön avulla. Nämä valitsijat ovat kuitenkin herkkiä muutoksille, jos esimerkiksi painikkeen tekstiä muutetaan, testi ei enää löydä elementtiä.

Sovelluksen monikielisyys tuo myös oman lisähaasteensa. Tekstisisältöön perustuvat valitsijat toimivat vain tietyllä kieliasetuksilla, jolloin testien uudelleenkäytettävyys ja ylläpidettävyys kärsii merkittävästi. Lisäksi käyttöliittymää muokataan usein asiakkaan toiveiden mukaan, mikä tekee epäselvistä selektoreista vieläkin ongelmallisimpia.

Kommunikaation puute kehitystiimin kanssa

Toinen merkittävä haaste liittyy testaajien ja sovelluskehittäjien väliseen kommunikaatioon. Testit kaatuvat usein siksi, että käyttöliittymää, API-kutsuja tai reittejä on muutettu ilman, että muutoksista on tiedotettu testaajalle. Myös uusista ominaisuuksista saatava tieto on usein puutteellista. Ominaisuuksia on saatettu jo toteuttaa tai julkaista ilman, että testaaja on saanut riittävästi kontekstia niiden testaamista varten. Tämä johtaa siihen, että testit jäävät reaktiivisiksi. Ongelmat havaitaan vasta jälkikäteen, eikä testausta voida suunnitella ajoissa osaksi kehitystyötä.

Tämä viestinnän puute aiheuttaa testien rikkoutumista ja hidastaa palauteprosessia. Kun tiedonkulku ei ole systemaattista, testit eivät pysy sovelluksen muutosten mukana, eikä testaus voi tukea sen kehitystä ennakoivasti.

Epävakaat odotukset

Testien epäonnistumisia aiheuttaa usein se, että sovelluksen näkymät tai elementit eivät ehdi latautua riittävän nopeasti ennen testin seuraavaa komentoa. Tämä on erityisen ongelmallista tilanteissa, joissa sovelluksen toiminnallisuus on kunnossa, mutta testin rakenne ei osaa odottaa tarpeeksi sovelluksen tilan valmistumista. Vaikka joissain tapauksissa on jäänyt käyttöön cy.wait()-komento, niiden kovakoodattu aikaperusteisuus ei riitä käsittelemään tilanteita luotettavasti.

Sovelluksen kehityksessä priorisoidaan toiminnallisuutta nopeuden sijaan, mikä tarkoittaa, että komponentit voivat ilmestyä näkyviin hitaammin tai eri tahdissa. Tämä aiheuttaa testien epäluotettavuutta ja testit voivat epäonnistua satunnaisesti ilman varsinaista logiikka virhettä. Esimerkiksi jos testissä odotetaan kolme sekuntia, mutta elementti latautuu sattumalta vasta 4 sekunnin kohdalla, testi kaatuu ilman, että sovelluksessa olisi mikään varsinaisesti rikki.

Testidatan hallinta ja ympäristön epävarmuus

Testien suorittaminen edellyttää yleensä tiettyä vakio alkutilaa tai valmiiksi olemassa olevaa dataa. Yrityksessä on kuitenkin useita käyttöliittymiä, joissa testin luoma data jää pysyvästi näkyviin, eikä sitä voi poistaa käyttöliittymän kautta. Vaikka joissain näkymissä on “poista”-toiminto, se ei välttämättä toimi odotetusti tai siivoa dataa kaikkialta. Tämä johtaa siihen, että on testejä, joita voi käytännössä ajaa vain kerran, minkä jälkeen tietokanta on manuaalisesti tyhjennettävä ennen seuraavaa testiä.

Tällainen ympäristön epävarmuus rikkoo E2E-testauksen olennaisimpia periaatteita, kuten toistettavuutta, eristystä ja ennakoitavuutta. Testejä ei voida ajaa luotettavasti, mikä heikentää testauksen ylläpidettävyyttä ja kestävyyttä.

Toistuvan testikoodin aiheuttama ylläpito

Testauksen arjessa havaittu haaste liittyy testikoodin toistuvuuteen. Useat testit sisältävät samoja vaiheita. Erityisesti kirjautumiseen liittyviä toimintoja, jotka on kirjoitettu erikseen jokaiseen testi tiedostoon. Tämä tarkoittaa, että jos kirjautumiseen tehdään muutos, se on korjattava jokaisesta testistä erikseen. Tällainen rakenne tekee testikoodista työlään ylläpitää, sillä pienikin muutos voi edellyttää korjauksia kymmeniin eri tiedostoihin.

Toistuvuus ongelma ei ole vain kirjautumisessa, vaan koskee myös muita testikoodin osia, kuten tietyn alkutilan valmistelua. Kun sama logiikka on kopioitu useaan tiedostoon ilman yleistä ratkaisua, testien muokkaaminen ja virheiden löytäminen vaikeutuu huomattavasti.

Tällainen rakenne estää testien hallittua kehittämistä ja hidastaa kehitystyötä. Testit eivät ole tällöin joustavia, mikä heikentää automatisoinnin hyötyjä erityisesti muuttuvassa sovellusympäristössä.

5.3 Kestävän end-to-end-testauksen periaatteet ja suositukset

End-to-end-testauksen tavoitteena on varmistaa, että sovellus toimii loppukäyttäjän näkökulmasta tarkasteltuna. Testien ylläpidettävyys ja pitkäikäisyys voivat kuitenkin kärsiä merkittävästi, ellei kestävyyteen liittyviä kysymyksiä huomioida jo suunnitteluvaiheessa. Kestävä testaus tarkoittaa, että testit säilyttävät toimintakykynsä sovelluksen kehittyessä ilman jatkuvaa manuaalista ylläpitoa.

Selektorit

Testien selektorit on tärkeä end-to-end-testien kestävyyden kannalta. Takuya Suemura (2024) tuo esiin, että usein käytetyt id- ja class-attribuutit ovat osa sovelluksen sisäistä HTML-rakennetta. Ne voivat muuttua kehitystyön edetessä ilman, että käyttäjän näkökulmasta mikään muuttuu. Tämän vuoksi niihin perustuvat valinnat tekevät testikoodista haavoittuvaa ja lisäävät ylläpidon määrää.

Cypressin virallinen ohjeistus suosittelee käyttämään testaukseen tarkoitettuja attribuutteja, kuten data-testid tai data-cy, ja varoittaa nimenomaan class-luokkien ja id-tunnisteiden käytöstä testien selektoreina, koska ne muuttuvat usein visuaalisten tai rakenteellisten päivitysten yhteydessä. (Cypress Best Practices 2025)

Kommunikaatio

Vaikka työkalut ja prosessit ovat tärkeitä ohjelmistokehityksessä, kehittäjien ja testaajien välinen yhteistyö perustuu lopulta toimivaan vuorovaikutukseen. Tamara Vasilevskan (2022) mukaan monet haasteet syntyvät siitä, että roolit toimivat liian erillään ja vuorovaikutus rajoittuu. Vasilevska korostaa, ettei testaajien tehtävä ole arvostella kehittäjien työtä, vaan varmistaa laadukas lopputulos. Kun roolit nähdään toisiaan täydentävinä eikä vastakkaisina, paranee yhteistyö ja koko ohjelmistoprosessin laatu (Vasilevska 2022).

Toimiva yhteistyö edellyttää testaajien osallistumista jo kehityksen alkuvaiheessa, kuten vaatimusten määrittelyssä ja suunnittelussa. Tämä auttaa tunnistamaan testattavat kohdat ajoissa ja lisää

yhteistä ymmärrystä sovelluksen tarkoituksesta ja käyttötapauksista. Kiranin (2023) mukaan säännölliset yhteiset palaverit, viestintäkanavat ja testitapausten katselmoinnit ovat keinoja, joilla yhteistyötä voidaan syventää. Tällainen yhteistyö lisää luottamusta ja parantaa testien laatua ja kattavuutta. Esimerkiksi kehittäjien lisäämät käyttöliittymäelementtien tunnisteet tukevat testien automatisointia, kun taas testaajat tuovat loppukäyttäjän näkökulmaa (Kiran 2023).

Epävakaat odotukset

Kuten kappaleessa 4.6 todettiin, Cypress odottaa automaattisesti, että elementit ovat näkyvissä ja valmiita ennen seuraavan komennon suorittamista. Cypressin dokumentaation mukaan testien epävakaus liittyy kuitenkin usein siihen, että ne odottavat sovelluksen tilan muuttumista epäluotettavalla tavalla. Yleinen virhe on käyttää satunnaisia viiveitä, kuten cy.wait(5000), joita dokumentaatio kuvaa tarpeettomiksi lähes kaikissa tilanteissa. Esimerkiksi cy.request() ja cy.visit() -komennot odottavat automaattisesti vastausta palvelimelta tai sivun latautumista, joten lisäviiveen käyttö ei tuo lisäarvoa vaan hidastaa testausta ja voi aiheuttaa epäluotettavaa käyttäytymistä. (Cypress Best Practices 2025)

Dokumentaatio suosittelee automaattisten odotusten käyttöä, jolloin testit etenevät vasta, kun jokin ehto on varmasti täyttynyt. Esimerkiksi aliaksen odottaminen cy.wait('@getUsers') tai elementin näkyvyyden tarkistaminen cy.get('[data-testid="element"]').should('be.visible') ovat tapoja varmistaa, että sovellus on oikeassa tilassa ennen seuraavaa askelta. (Cypress Best Practices 2025; Introduction to Cypress App 2025)

Testidatan hallinta

Luotettava end-to-end-testaus vaatii hallittua ja ennustettavaa testidataa. Runchevan (2022) mukaan testien tulisi olla riippumattomia aiemmista ajoista, mikä edellyttää testiympäristön palauttamista puhtaaseen tilaan ennen jokaista suoritusta. Hän kertoo, että näin vältetään tilanne, jossa testitulokset vääristyvät aikaisemmin luodun datan vaikutuksesta. Runcheva painottaa, että testien on oltava uudelleenkäytettäviä ja testaus ei saa olla sidottu aiempien testien lopputilaan.

Johnston (2024) mielestä olisi hyvä, että erityisesti testausta aloitettaessa kannattaa suosia tapaa, jossa testidata luodaan testin suorituksen aikana. Hänen mukaansa on tärkeää käyttää yksilöllisiä ja tunnistettavia arvoja, jotta vältetään virheelliset positiiviset tulokset, joita voi syntyä. Vaikka tämä lähestymistapa voi johtaa testien pidentymiseen. (Johnston 2024)

Toistuvan testikoodin uudelleenkäytettävyys

Kuten kappaleessa 4.6 esiteltiin, Cypress tukee toistuvan testikoodin eriyttämistä omiksi komennoiksi. Tilanteissa, joissa samoja toimintoja tarvitaan useissa testeissä, toiminnot tulisi siirtää erillisiin, uudelleenkäytettäviin komentoihin. Cypress tarjoaa tähän tarkoitukseen mahdollisuuden luoda niin kutsuttuja custom commands -komentoja, jotka kirjoitetaan commands -tiedostoon ja ovat käytettävissä kaikissa testitiedostoissa. (da Costa 2021, luku 11)

Myös Cypressin cy.session()-ominaisuus on kuvattu teoria osuudessa (ks. luku 4.6), ja se mahdollistaa kirjautumisen sessiotiedon säilyttämisen testien välillä ilman, että prosessi toistetaan jokaisessa testissä. Tämä nopeuttaa testien suoritusta ja vähentää toistosta johtuvaa epävakautta. (Cypress Best Practices 2025)

5.4 Ohjeistuksen rakentaminen

Tässä opinnäytetyössä laadittiin ohjeistus (ks. Liite 1), jonka tavoitteena on tukea kestävän end-to-end-testauksen toteuttamista Cypressin avulla. Ohjeistus on kirjoitettu englannin kielellä, jotta sitä voidaan hyödyntää suoraan osana yrityksen testausta.

Ohjeistuksen (ks. Liite 1) sisältö pohjautuu kahteen päälähteeseen. Ensimmäisenä oli yritykseen tehtyyn testauksen nykytilan katsaukseen ja toisena tekijän omiin kokemuksiin testaajana yrityksessä. Ne suurimmat ongelmat, jotka katsauksessa nousivat toistuvasti esiin, kuten selektorien puutteet ja toistuvan koodin aiheuttama ylläpito, käsitellään ohjeistuksessa suoraan, ja niihin on etsitty konkreettisia ratkaisuja.

Lisäksi ohjeistukseen (ks. Liite 1) sisällytettiin neuvoja, jotka eivät välttämättä nousseet esiin katsauksessa, mutta ovat muodostuneet tärkeiksi havainnoiksi testauksen arjessa. Näitä ovat esimerkiksi muistutukset testikoodin kommentoinnista sekä lintterin käytöstä testikoodin yhdenmukaisuuden varmistamiseksi. Tällaiset lisäykset tekevät ohjeistuksesta paitsi ongelmiin vastaavan, mutta myös ennakoivan työkalun, joka edistää laadukasta ja ylläpidettävää testauskäytäntöä yrityksessä.

Sisällössä painotettiin käytettävyyttä, roolikohtaista hyödyllisyyttä ja selkeyttä. Ohjeistus on jaettu kahteen osioon, joista toinen on suunnattu testaajille ja toinen kehittäjille. Tämä jako vastaa havaittuihin tarpeisiin ja tukee tiimien välistä yhteistyötä. Testaajien osuus keskittyy testien kirjoittamiseen, ajamiseen ja ylläpitoon, kun taas kehittäjille suunnattu osio korostaa testauksen mahdollistamista. (ks. Liite 1)

5.5 Ohjeistuksen soveltaminen käytännössä

Kaikkia ohjeistuksen kohtia ei ollut mahdollista soveltaa työn laajuuden ja roolijaon vuoksi, käytännön työssä keskityttiin niihin osa-alueisiin, jotka olivat suoraan testaajan hallittavissa. Kaikki sovelluskoodin rakenteelliset muutokset, kuten DOM-rakenteen uudelleenjärjestely, eivät kuuluneet testaajan vastuulle. Samoin viestintään ja tiimityöhön liittyvät kehitysehdotukset jäivät ohjeistuksen konkreettisen soveltamisen ulkopuolelle. Osaan esimerkeistä sovelluskehittäjät lisäsivät tunnisteita työtä varten. Soveltaminen kohdistui siten niihin teknisiin ratkaisuihin, jotka oli mahdollista toteuttaa testikoodissa suoraan ja joilla oli konkreettinen vaikutus testien vakauteen ja ylläpidettävyyteen. Tutkimuksen havainnollistamiseksi esimerkki koodin pätkät sisältävät normaalia laajempia ja suomenkielisiä kommentteja, mutta nämä eivät vastaa todellista kommentointikäytäntöä.

Selektorien parantaminen

Testikoodin kannalta merkittävä haavoittuvuus liittyi tapaan, jolla elementtejä valittiin käyttöliittymästä. Erityisen ongelmallisia olivat selektorit, jotka perustuivat näkyvään tekstiin, visuaalisiin luokkiin tai elementtien sijaintiin DOM-rakenteessa.

Kuviossa 1 alkuperäinen tapa oli klikata painiketta tekstin "Työmaan osoite" perusteella ja löytää siihen liittyvä painike DOM-hierarkian avulla. Tämä korvattiin käyttämällä yksinkertaista ja helposti luettavaa data-testid="add-address-button"-tunnistetta, joka parantaa testin kestävyyttä rakenne- tai kieli muutosten yhteydessä.

```
// Aiempi tapa: etsitään napin otsikko ja klikataan sen perusteella
cy.iframe().contains("Työmaan osoite").parent().find("button").click();

// Uusi tapa: käytetään selkeämpää tunnistetta (data-testid)
cy.iframe().find('[data-testid="add-address-button"]').click();
```

Kuvio 1. Selektorin parannus tekstistä tunnisteeseen

Kuviossa 2 painiketta painettiin indeksin mukaan .eq(2), mikä on epäselvää ja haavoittuvaa. Tämä korvattiin määritellyllä data-testid="confirm-location-button"-tunnisteella, joka tekee pelkästään testin luettavammaksi ja kestävämmäksi.

```
// Aiempi tapa: klikataan kolmatta kuvaketta Google Maps -dialogissa sijainnin vahvistamiseksi
cy.iframe()
  .find("dialog#google-map-form-dialog i")
  .eq(2)
  .should("be.visible")
  .click();

// Uusi tapa: käytetään suoraan määriteltyä tunnistetta (data-testid)
cy.iframe().find('[data-testid="confirm-location-button"]').click();
```

Kuvio 2. Indeksihaku korvattu yksilöivällä tunnisteella

DOM-hierarkian riippuvuuden vähentäminen

Kuvion 3 tapauksessa alkuperäinen koodi käytti toistuvia parent()-ketjuja DOM-rakenteen navigointiin. Tällaiset valinnat rikkoutuvat helposti, mikäli käyttöliittymää muokataan. Vaikka vanhassa tavassa oli käytössä data-testid-attribuutti, se oli liian yleistetty arvo “EditIcon”, joita on käytössä monia. Uuden tavan avulla nämä korvattiin data-testid-attribuuteilla, jolloin valinta ei enää riipu

elementin sijainnista tai hierarkiasta. Samalla kovakoodatut arvot, kuten type("5"), korvattiin testimuuttujilla testValues.price, mikä parantaa testin uudelleenkäytettävyyttä ja selkeyttää sen logiikkaa.

```
// Aiempi tapa: etsitään varausrivi DOM-rakenteen perusteella ja klikataan muokkauspainiketta
cy.iframe()
  .contains(heap)
  .parent()
  .parent()
  .parent()
  .parent()
  .find('[data-testid="EditIcon"]')
  .click({ force: true });

// Aiempi tapa: odotetaan kiinteä aika lomakkeen päivitykseen
cy.wait(1000);

// Aiempi tapa: etsitään varausrivi DOM-rakenteen perusteella ja
// kirjoitetaan hinta kenttään kovakoodatulla numerolla
cy.iframe()
  .contains(heap)
  .parent()
  .parent()
  .parent()
  .parent()
  .find("input[id='grid-input-4']")
  .type("5");

// Uusi tapa: klikataan muokkauspainiketta yksilöidyllä tunnisteella
cy.iframe().find(`[data-testid="edit-button-${heap}"]`).click();

// Uusi tapa: tarkistetaan että hintakenttä on näkyvissä ennen syöttöä
cy.iframe().find(`[data-testid="price-button-${heap}"]`).should("be.visible");

// Uusi tapa: syötetään testiarvo muuttujasta kovakoodauksen sijaan
cy.iframe()
  .find(`[data-testid="price-button-${heap}"]`)
  .type(`${testValues.price}`);
```

Kuvio 3. DOM-navigointi ja kovakoodaus korvattu selkeillä tunnisteilla ja muuttujilla

Ajoituksen hallinta

Aiemmin testeissä käytettiin useassa kohtaa kiinteitä viiveitä, kuten cy.wait(5000) (ks. Kuvio 4). Tämä tapa ei ole luotettava, koska odotusaika on mahdollisesti liian pitkä tai liian lyhyt. Uudemmassa ratkaisussa kuviossa 4 käytetään cy.get(..., {timeout: 10000}) -syntaksia, jolloin testi odottaa dynaamisesti ainakin 10 sekuntia, että elementti ilmestyy. Tämä tekee testeistä mahdollisesti nopeampia ja ympäristömuutoksia paremmin kestäviä.

```
// Aiempi tapa: odotetaan kiinteä aika sivun latautumista varten
cy.wait(5000);

// Aiempi tapa: syötetään kenttään työmaan nimi
cy.iframe().find("#sites").type(site);

// Uusi tapa: Annetaan elementille enemmän aikaa latautua tarvittaessa,
// jonka jälkeen syötetään työmaan nimi.
// Mahdollistetaan myös nopeampi eteneminen jos kenttä löytyy aiemmin
cy.iframe().find("#sites", { timeout: 10000 }).type(site);
```

Kuvio 4. Kiinteä viive korvattu dynaamisella odotuksella

Toistuvan logiikan refaktorointi

Alkuperäinen kirjautumis- ja kontekstinvalintaprosessi toteutettiin kokonaisuudessaan beforeEach-lohkossa. Tämä sisälsi evästeiden tallentamisen ja uudelleenasettamisen, manuaalisen kirjautumisen sekä asiakas kontekstin valinnan. Vaikka ratkaisu oli teknisesti toimiva, se oli pitkä, vaikeasti ylläpidettävä ja altis rikkoutumaan, mikäli sovellusta muutettaisiin näiltä osin lainkaan. (ks. Kuvio 5)

```
// Aiempi tapa: tallennetaan evästeet muuttujaan ja käytetään niitä kirjautumisen ohittamiseen
let savedCookies = [];

// Käyttäjätunnukset haetaan ympäristömuuttujista
// @ts-ignore
const user = Cypress.env("users").admin;

beforeEach(() => {
  // Aiempi tapa: jos evästeet on jo tallennettu, käytetään niitä ja ohitetaan kirjautuminen
  if (savedCookies.length > 0) {
    // Asetetaan tallennetut evästeet ennen testin alkua
    savedCookies.forEach((cookie) => {
      cy.setCookie(cookie.name, cookie.value);
    });
    cy.log("Cookies set, skipping login.");

    // Siirrytään suoraan kontekstivalintasivulle
    cy.visit("/cui/pages/choose-context/");

    // Valitaan asiakaskonteksti
    cy.get(`@ [data-customer-name="${context}"]`).click();

    // Varmistetaan että iframe latautui
    cy.frameLoaded("#bonitaframe");
    cy.get("@ #bonitaframe").should("exist");
  } else {
    // Aiempi tapa: suoritetaan kirjautuminen, jos evästeitä ei ole vielä tallennettu
    cy.visit("/cui/pages/login/");

    // Kirjaudutaan käyttäjätunnuksilla
    cy.get("@ input[id='inp-usr']").type(user.username);
    cy.get("@ input[id='inp-psw']").type(user.password);
    cy.get("@ input[id='inp-go']").click();

    // Siirrytään kontekstivalintaan kirjautumisen jälkeen
    cy.get("@ input[id='btn-context']").click();

    // Valitaan asiakaskonteksti
    cy.get(`@ [data-customer-name="${context}"]`).click();

    // Varmistetaan että iframe latautui
    cy.frameLoaded("#bonitaframe");
    cy.get("@ #bonitaframe").should("exist");

    // Tallennetaan kaikki evästeet myöhempää käyttöä varten
    cy.getCookies().then((cookies) => {
      savedCookies = cookies;
      cy.log("All cookies saved.");
    });
  }
});
```

Kuvio 5. Kirjautuminen ja kontekstivalinta toteutettu beforeEach-rakenteessa jokaisessa testissä

Tämä kokonaisuus (ks. Kuvio 5) purettiin kahdeksi erilliseksi Cypress-komennoksi commands.ts tiedostoon. Kuvio 6 kirjautuminen kapseloitiin cy.login()-metodiksi, jossa käytetään cy.session-toimintoa estämään turhaa toistoa eri testien välillä. Lisäksi siihen liitettiin cy.intercept, jolla varmistetaan, että backend vastaa onnistuneesti. Asiakas kontekstin valinta puolestaan eriytettiin omaksi cy.context()-komennoksi, jossa hoidettiin sivulla navigointi, asiakasvalinta sekä iframe-latauksen varmistus.

```
/*************** LOGIN *****************/
Cypress.Commands.add("login", () => {  new *
  const user = Cypress.env("users").admin;

  // Uusi tapa: kirjautuminen toteutettu session-muistiinpanolla,
  // jotta samaa kirjautumista ei toisteta joka testissä
  cy.session([user.username, user.password], () => {
    // Kuunnellaan kirjautumispyyntöä ja annetaan sille alias
    cy.intercept("POST", `${Cypress.config("baseUrl")}/acm/sess/init`).as(
      "login",
    );

    // Siirrytään kirjautumissivulle
    cy.visit("/cui/pages/login/");

    // Syötetään kirjautumistiedot
    cy.get("input[id='inp-usr']").type(user.username);
    cy.get("input[id='inp-psw']").type(user.password);
    cy.get("input[id='inp-go']").click();

    // Varmistetaan, että kirjautuminen onnistui (HTTP 200)
    cy.wait("@login").its("response.statusCode").should("eq", 200);
  });
});

/*************** GO TO CHOOSE CONTEXT  *****************/
Cypress.Commands.add("context", () => {  new *
  const context = Cypress.env("context");

  // Uusi tapa: kontekstin valinta kapseloitu
  // omaan komentoon testin toistettavuuden ja selkeyden vuoksi
  cy.visit("/cui/pages/choose-context/");

  // Valitaan asiakaskonteksti yksilöidyllä tunnisteella
  cy.get(`[data-customer-name="${context}"]`).click();

  // Varmistetaan että iframe latautuu
  cy.frameLoaded("#bonitaframe");
  cy.get("#bonitaframe").should("exist");
});
```

Kuvio 6. Kirjautuminen ja konteksti kapseloitu erillisiksi Cypress-komennoiksi commands.ts tiedostoon

Näitä komentoja kutsutaan nyt yksinkertaisesti ennen jokaista testiä, mikä tekee testikoodista huomattavasti selkeämmän ja vähemmän herkän rikkoutumiselle. (ks. Kuvio 7)

```
beforeEach(() => {
  cy.login();
  cy.context();
});
```

Kuvio 7. Komentojen käyttö testeissä

6 Tulokset

Ohjeistuksen käytännön soveltaminen testikoodiin toi esille konkreettisia parannuksia Cypress-pohjaisten end-to-end-testien kestävyyteen, rakenteeseen ja ylläpidettävyyteen. Soveltaminen kohdistui niihin ohjeistuksen kohtiin, jotka olivat suoraan testaajan hallittavissa ilman itsenäisiä muutoksia sovelluskoodiin. Vaikka ohjeistuksen yhteistyötä vaativat osat jäivät työn rajauksen ulkopuolelle, tekniset toimet osoittautuivat itsessään kannattaviksi.

Ensimmäisenä keskeisenä tuloksena havaittiin, että testien haavoittuvuus DOM-rakenteen muutoksille väheni merkittävästi, kun selektorit muutettiin käyttämään uniikkeja data-testid-attribuutteja. Näiden avulla testit eivät enää rikkoudu pienistä käyttöliittymän muutoksista tai visuaalisista uudelleenjärjestelyistä, koska testi löytää elementit aina. Esimerkiksi testit, joissa aiemmin käytettiin tekstipohjaisia tai indeksipohjaisia valintoja, selkeytyivät käyttämään saman tapaa elementin etsimiseen.

Toisena keskeisenä havaintona oli testien ajoituksen hallinnan parantuminen. Kovakoodattujen viiveiden, kuten cy.wait(5000), korvaaminen dynaamisilla odotuksilla tai muilla ratkaisuilla paransi testien varmuutta ja vähensi mahdollisia turhia odotuksia. Tämä toi testikoodiin teknistä kestävyyttä ja suorituskykyä, kun testit kykenevät reagoimaan esim. käyttöliittymän tai API-kutsujen todelliseen tilaan.

Lisäksi testikoodin modulaarisuus kehittyi selkeästi. Kirjautumislogiikan ja asiakaskontekstin valinnan siirtäminen omiksi Cypress-komennoiksi lyhensi yksittäisten testien pituutta ja paransi niiden

luettavuutta. Myös cy.session()-toiminnon käyttöönotto selkeytti testien rakennetta ja mukautui paremmin Cypressin toimintalogiikkaan verrattuna aiempaan itse toteutettuun ratkaisuun. Uuden toteutuksen etuna on erityisesti se, että muutokset voidaan tehdä yhdessä tiedostossa ilman, että jokaista yksittäistä testiä tai tiedostoa tarvitsee erikseen muokata. Tämä vähentää ylläpitotyötä ja nopeuttaa reagointia sovelluksen muutoksiin.

Yksittäisten teknisten parannusten lisäksi ohjeistuksen soveltaminen auttoi muodostamaan yhtenäisen ja johdonmukaisen tavan kirjoittaa testejä. Selektorien käyttö ja testikoodin rakenne yhdenmukaistuivat, mikä helpottaa uusien testaajien perehdyttämistä.

7 Pohdinta

7.1 Johtopäätökset

Työn perusteella voidaan todeta, että testaaja voi vaikuttaa testauksen laatuun ja toimivuuteen myös ilman pääsyä sovelluskoodiin tai kehityskäytäntöihin, kunhan käytössä on selkeä ja huolellisesti suunniteltu toimintamalli. Parhaat tulokset kuitenkin saadaan silloin, kun ohjeistus otetaan huomioon jo sovelluksen kehitysvaiheessa. Tällöin kehittäjät voivat rakentaa testauksen kannalta suunniteltuja ratkaisuja ja koko kehitystyö hyötyy ennakoivasta testattavuuden suunnittelusta.

Samalla työ nosti esiin olennaisen asian testien sietokyvystä. Ei ole mahdollista tehdä testeistä mahdollisimman kestäviä ilman kehittäjien ja testaajien yhteistyötä. Sovelluksen rakenteelliset ratkaisut, kuten merkityksellinen DOM, selkeät tunnisteet ja dokumentoidut muutokset luovat perustan sille, kuinka helposti ja luotettavasti testejä voidaan kirjoittaa ja ylläpitää. Mikäli sovelluksessa on puutteita näissä, testikoodi jää varmasti haavoittuvaksi käyttöliittymän muutoksille. Työn tulokset siis vahvistavat käsitystä siitä, että laadukas E2E-testaus ei ole vain tekninen tehtävä, vaan osa kehitystyön kokonaisuutta.

Yrityksen näkökulmasta ohjeistuksen hyödyntäminen tuo useita selkeitä hyötyjä. Kun testikoodi on rakennettu kestäväksi ja modulaariseksi, sen ylläpito vie vähemmän aikaa ja aiheuttaa vähemmän häiriöitä muulle kehitykselle. Tämä vähentää testien jatkuvaa korjaustarvetta, parantaa testauksen

luotettavuutta ja nopeuttaa kehitystyön etenemistä. Näiden avulla voidaan pitkällä aikavälillä saavuttaa myös taloudellista hyötyä, kun manuaalista työtä ja virheiden aiheuttamia kustannuksia voidaan vähentää.

Lisäksi yhdenmukaiset käytännöt testeissä tekevät niistä helpommin luettavaa ja koulutettavaa, joka mahdollistaa uuden osaamisen siirtymistä tiimiin. Ohjeistus voi siis olla arvokas tukimateriaali uusien ominaisuuksien kehittämisessä, kun testauksen laatu varmistetaan jo kehitystyön alkuvaiheessa.

7.2 Luotettavuus ja eettisyys

Tässä opinnäytetyössä on pyritty noudattamaan Tutkimuseettisen neuvottelukunnan (TENK) ja suomalaisen tiedeyhteisön ohjeistusta hyvästä tieteellisestä käytännöstä (HTK). Tutkimuksen kaikissa vaiheissa on pyritty rehellisyyteen, huolellisuuteen ja avoimuuteen. Lähteiden valinnassa painotettiin ajankohtaisuutta ja luotettavuutta, ja viittaustekniikassa on noudatettu Jyväskylän ammattikorkeakoulun raportointiohjeistusta.

Kehittämistyössä on olennaista noudattaa hyvää eettistä käytäntöä, mikä tarkoittaa rehellisyyttä, huolellisuutta ja tarkkuutta kaikissa työn vaiheissa. Lisäksi on huomioitava mahdolliset oikeudelliset näkökulmat ja sopimukselliset seikat, kuten toimeksiantajan linjaukset ja käytännöt. Myös tiedonhankintaan ja menetelmien käyttöön liittyvät eettiset kysymykset on arvioitava huolellisesti. Erityistä huomiota on kiinnitettävä myös lähteiden asianmukaiseen merkitsemiseen ja tiedon alkuperän kunnioittamiseen. (Ojasalo, Moilanen & Ritalahti 2015, 48–49.)

Tutkimus toteutettiin kehittämistyönä, jossa tekijä osallistui itse tutkittavaan toimintaan testaajana ohjelmistokehitystiimissä. Tämä osallistuva rooli mahdollisti yksityiskohtaisen ja käytännönläheisen analyysin, mutta samalla se asetti rajoituksia arvioinnin objektiivisuudelle. Koska aineisto perustui tutkijan omaan havaintoon ja testikoodin muokkaamiseen, tulkinnat ovat pakostakin sidoksissa yksilölliseen tulkintaan. Tätä rajoitetta on pyritty tasapainottamaan avoimella raportoinnilla.

Vuoren (n.d.) mukaan eettiset kysymykset ovat laadullisessa tutkimuksessa läsnä kaikissa tutkimuksen vaiheissa. Eikä vain tutkittavien suojelemisessa, vaan myös siinä, miten tutkija tuottaa, tulkitsee ja jakaa tietoa. Hän korostaa, että tutkijan on arvioitava toimintaansa myös silloin, kun eettiset ratkaisut eivät ole selkeitä. Tässä työssä on pyritty noudattamaan tätä kertomalla avoimesti niistä ratkaisuista, joita työn edetessä tehtiin esimerkiksi, että tutkimuksen havainnollistamista varten osa testikoodin esimerkeistä on kommentoitu tavanomaista laajemmin ja suomenkielisesti, vaikka tämä ei vastaa normaalia tapaa.

Tutkimuksessa ei käsitelty henkilötietoja, salassa pidettäviä aineistoja tai arkaluontoista tietoa. Kaikki aineisto perustui tutkijan havaintoihin, avoimeen dokumentaatioon ja testikoodin analyysiin.

Mahdollisena haasteena voidaan pitää sitä, että kehittämistyö kohdistui yhteen sovellukseen ja testausympäristöön, vaikka ohjeistus oli tarkoitettu yrityksen omaan käyttöön. Se silti rajoittaa tulosten yleistettävyyttä. Lisäksi luotettavuuden kannalta keskeinen haaste liittyy ohjeistuksen validointiin. Tässä työssä ohjeistuksen pitkäaikaista toimivuutta ei arvioitu, koska sovellukseen ei voitu tehdä muutoksia vain tutkimusta varten. Tästä syystä ohjeistus perustuu tutkijan havaintoihin kokemuksen kautta ja kirjallisuuteen, ei erilliseen soveltuvuuden arviointiin. Tämä rajoittaa lopputuotoksen käytännön oikeellisuuden arviointia.

7.3 Jatkokehitysehdotukset

Tässä opinnäytetyössä laadittu ohjeistus keskittyi erityisesti testikoodin kestävyyden parantamiseen ja sen käytännön soveltamiseen. Työn aikana kuitenkin havaittiin, että testauksen kehittäminen ulottuu laajemmalle.

Ensisijainen kehittämiskohde liittyy ohjeistuksen käyttöönottoon. Jotta ohjeistus ei jäisi kertaluonteiseksi dokumentiksi, se tulisi liittää osaksi yrityksen sisäistä dokumentaatiota, esimerkiksi versionhallintajärjestelmän wikiosioon. Ylläpidosta voidaan sopia tiimin kesken, jotta ohje pysyy ajan tasalla ja mukautuu kehitystyön muuttuviin tarpeisiin.

Samalla testauksen roolia pitäisi vahvistaa osana ohjelmistokehitystä siten, että testit nähdään kiinteänä osana suunnittelua ja kehitystä, eikä erillisenä vaiheena. Tämä vaatii yhteistä ymmärrystä testauksen merkityksestä, jatkuvaa keskustelua testaajien ja kehittäjien välillä. Testaamisen käytäntöjä voidaan kehittää esimerkiksi sopimalla yhteisiä periaatteita selektorien käytöstä ja huomioimalla testattavuus jo suunnittelussa.

Lisäksi on suositeltavaa arvioida käytössä olevien työkalujen soveltuvuutta muuttuvassa kehitysympäristössä. Vaikka Cypress on nykyisessä ympäristössä toimiva ratkaisu, voi esimerkiksi Playwrightin tai muiden E2E-työkalujen käyttöönotto nousta ajankohtaiseksi, mikäli vaatimukset selainyhteensopivuuden, suorituskyvyn tai mobiilitestauksen osalta kasvavat.

Lähteet

Berga, K. 2024a. What is Manual Testing? Learn the Step-by-Step Process, Types & Techniques. TestDevLab blogi. Viitattu 12.4.2025. https://www.testdevlab.com/blog/manual-testing.

Berga, K. 2024b. The Ultimate Guide to Unit Testing: Benefits, Challenges, and Best Practices. TestDevLab blogi. Viitattu 13.4.2025. https://www.testdevlab.com/blog/the-ultimate-guide-to-unit-testing.

Best Practices. 2025. Cypress dokumentaatio. Viitattu 14.5.2025. https://docs.cypress.io/app/core-concepts/best-practices.

Custom Commands. 2025. Cypress dokumentaatio. Viitattu 26.5.2025. https://docs.cypress.io/api/cypress-api/custom-commands.

da Costa, L. 2021. Testing JavaScript Applications. Shelter Island, NY: Manning Publications. Viitattu 20.4.2025.

Dilmegani, C. 2025. 7 End-to-End Testing Best Practices in 2025. AIMultiple artikkeli. Viitattu 20.4.2025. https://research.aimultiple.com/end-to-end-testing-best-practices/.

Gallinelli, N. 2021. Front End vs. Back End Development. Flatiron School blogi. Viitattu 10.4.2025. https://flatironschool.com/blog/front-end-vs-back-end-development/.

GAT Staff Writers. 2024. What is Automation Testing? – Everything You Need To Know. Global App Testing blogi. Viitattu 12.4.2025. https://www.globalapptesting.com/blog/what-is-automation-testing.

Günther, K. & Hasanen, K. N.d. Tutkimuksen suunnittelu. Teoksessa: Laadullisen tutkimuksen verkkokäsikirja. Tampere: Yhteiskuntatieteellinen tietoarkisto (FSD), Tampereen yliopisto. Viitattu 27.3.2025. https://www.fsd.tuni.fi/fi/palvelut/menetelmaopetus/kvali/laadullisen-tutkimuksen-prosessi/tutkimuksen-suunnittelu/.

How Cypress Works. N.d. Cypress.io. Viitattu 20.4.2025. https://www.cypress.io/how-it-works.

Introduction to Cypress App. 2025. Cypress dokumentaatio. Viitattu 26.5.2025. https://docs.cypress.io/app/core-concepts/introduction-to-cypress.

Johnston, D. 2024. How to Get Started Testing a React codebase that has no tests. GitHub sivusto. Viitattu 14.5.2025. https://blacksheepcode.com/posts/how_to_get_started_testing.

Katalon Team. 2025. What is Integration Testing? Definition, How-to, Examples. Katalon blogi. Viitattu 19.4.2025. https://katalon.com/resources-center/blog/integration-testing.

Kiran, G. 2023. How to Improve Collaboration Between Your Developers and Testers. Tavant blogi. Viitattu 15.5.2025. https://tavant.com/blog/how-to-improve-collaboration-between-your-developers-and-testers/.

Nguyen, H. 2025. Manual Testing: Full Tutorial (With Examples). Katalon blogi. Viitattu 12.4.2025. https://katalon.com/resources-center/blog/manual-testing.

Ojasalo, K., Moilanen, T. & Ritalahti, J. 2015. Kehittämistyön menetelmät: Uudenlaista osaamista liiketoimintaan. E-kirja Ellibsis Library -palvelussa. 4. p. Helsinki: Sanoma Pro. Luettavissa jamk:n tunnuksilla. Viitattu 19.5.2025. https://www.ellibslibrary.com/.

Playwright enables reliable end-to-end testing for modern web apps. N.d. Playwright etusivu. Viitattu 20.4.2025. https://playwright.dev/.

Raji, T. 2024. Cypress Vs Selenium: Which Testing Tool Is Right For You? Keploy blogi. Viitattu 20.4.2025. https://keploy.io/blog/community/cypress-vs-selenium-which-testing-tool-is-right-for-you.

Runcheva, N. 2022. 10 Guiding Principles for Effective E2E Test Automation in CI/CD. TestDevLab blogi. Viitattu 14.5.2025. https://www.testdevlab.com/blog/10-guiding-principles-for-effective-e2e-test-automation-in-ci-cd.

Schmitt, J. 2024. What is E2E? A Guide to End-to-End Testing. CircleCI blogi. Viitattu 20.4.2025. https://circleci.com/blog/what-is-end-to-end-testing/.

Selenium Documentation: Components Overview. 2022. Selenium dokumentaatio. Viitattu 20.4.2025. https://www.selenium.dev/documentation/overview/components/.

Session. 2025. Cypress dokumentaatio. Viitattu 26.5.2025. https://docs.cypress.io/api/commands/session.

Son, H. 2024. Test Planning: A Step-by-Step Guide for Software Testing Success. TestRail blogi. Viitattu 30.5.2025. https://www.testrail.com/blog/test-planning-guide/.

Son, H. 2025. How To Create A Test Plan (Steps, Examples, & Template). TestRail blogi. Viitattu 30.5.2025. https://www.testrail.com/blog/create-a-test-plan/.

Suemura, T. 2024. Best Practices for Element Locating in E2E Testing. Autify blogi. Viitattu 14.5.2025. https://autify.com/blog/best-practices-for-element-locating-in-e2e-testing.

Test. Automate. Accelerate. N.d. Cypress etusivu. Viitattu 20.4.2025. https://www.cypress.io/.

Top Test Automation Frameworks in 2025. 2024. SauceLabs blogi. Viitattu 30.5.2025. https://saucelabs.com/resources/blog/top-test-automation-frameworks-in-2023.

Variables and Aliases. 2025. Cypress dokumentaatio. Viitattu 26.5.2025. https://docs.cypress.io/app/core-concepts/variables-and-aliases.

Vasilevska, T. 2022. Effective Collaboration Between Software Developers and QA Engineers. TestDevLab blogi. Viitattu 14.5.2025. https://www.testdevlab.com/blog/collaboration-between-software-developers-and-qa-engineers.

Vuori, J. N.d. Tutkimusetiikka ihmistieteissä. Teoksessa: Laadullisen tutkimuksen verkkokäsikirja. Tampere: Yhteiskuntatieteellinen tietoarkisto (FSD), Tampereen yliopisto. Viitattu 27.3.2024. https://www.fsd.tuni.fi/fi/palvelut/menetelmaopetus/kvali/tutkimusetiikka/tutkimusetiikka-ihmistieteissa/.

What is Integration Testing. 2025. BrowserStack opas. Viitattu 19.4.2025. https://www.browserstack.com/guide/integration-testing.

What Is Playwright: A Tutorial on How to Use Playwright. 2024. LambdaTest opas. Viitattu 20.4.2025. https://www.lambdatest.com/playwright.

What is Selenium? A Complete Guide on Selenium Testing. 2023. LambdaTest opas. Viitattu 20.4.2025. https://www.lambdatest.com/selenium.

What is the DOM (Document Object Model) in Web Development? 2025. BrowserStack opas. Viitattu 10.4.2025. https://www.browserstack.com/guide/dom-in-web-development.

What is Unit Testing? N.d. Amazon Web Services opas. Viitattu 13.4.2025. https://aws.amazon.com/what-is/unit-testing/.

What's the Difference Between Frontend and Backend in Application Development? N.d. Amazon Web Services opas. Viitattu 10.4.2025. https://aws.amazon.com/compare/the-difference-between-frontend-and-backend.

Why Cypress. 2025. Cypress dokumentaatio. Viitattu 20.5.2025. https://docs.cypress.io/app/get-started/why-cypress.

Liitteet

Liite 1. Ohjeistus testaukseen (englanniksi)

Guidelines for Sustainable End-to-End Testing with Cypress

This guide is designed to support sustainable end-to-end (E2E) testing practices within the company. It is based on practical issues observed in the organization. The goal is to offer clear and repeatable approaches that improve the reliability, maintainability, and usefulness of tests.

1. Selector Usage

- Use attributes specifically intended for testing, such as data-testid, to identify elements. Example: cy.get('[data-testid="addOrder-button"]').
- Avoid using CSS classes, IDs, or text content as selectors, as they can change when the UI changes.
- Ask developers to add these attributes to key interactive elements like buttons, input fields, and dropdowns.

2. Timing and Wait Conditions

- Avoid using fixed wait times (cy.wait(3000)) unless absolutely necessary.
- Use conditions such as cy.get(...).should('be.visible') or cy.get(..., {timeout:5000}) to wait for elements to appear in the DOM.
- Utilize cy.intercept() and cy.wait('@alias') for monitoring and verifying API calls. Example: cy.intercept('GET', '/api/addorder').as('addOrder'); cy.wait('@addOrder');

3. Test Independence and Test Data Management

- Use unique identifiers (example: random generators) to distinguish test data, such as test-site-123456, to avoid collisions and false positives.
- Ensure that tests clean up after themselves if they create persistent data, for example, by sending a delete request to the API.
- Each test should be runnable independently and in any order without relying on other tests.
- Use beforeEach and afterEach blocks for preparing and cleaning up tests.
- Avoid hardcoding information like usernames or passwords in your tests. Use Cypress.env() to access them from environment variables.

4. Reusability of Test Code

- Encapsulate repetitive logic, such as login, into custom commands (cy.login()), and add them to cypress/support/commands.ts.
- Avoid hardcoded values. Use variables or test-specific configurations instead.

5. Test Readability and Clean Code

- Write a clear and descriptive title for each test: it('add a new site').
- Keep the test steps in a logical, sequential order.
- Add short comments to clarify what each step does and why. Example: //Press to open a site, there is longer wait time for random slowness.
- Use a linter (example: ESLint and Prettier) to maintain consistent and clean test code.
- When appropriate, write a short README or a comment block at the beginning of the test file explaining the test's purpose and dependencies.

6. Communication and Collaboration

- Ask for clarification when needed: If a new feature or change is not documented or its functionality is unclear, ask the developer or product owner for clarification.
- Document important changes from a testing perspective: If the application undergoes significant structural or data logic changes, log them in a GitHub Issue.
- Follow development progress actively: Stay informed about upcoming changes in time.

7. Error Handling and Feedback in Tests

- Write assertions in a way that failed tests provide meaningful error messages. Example: should('contain', 'Invalid password').

Developer Support for Testing

- **Semantic and structured DOM elements:** Build the UI using logical and consistent HTML. Use semantic elements and avoid overly nested or meaningless wrappers.
- **Test-friendly selectors:** Add data-testid attributes to critical elements, especially those involved in key user flows. Example: data-testid="addOrder-btn".
- **Version control and documentation of changes:** If UI structures, routes, or APIs change, log the changes clearly in GitHub Issues and inform testers.
- **Proactive communication:** Consider testing early during development and ensure that selectors and structure are planned from the start.
- **Collaborative test planning:** Involve testers when designing complex UI or interactive features. Early collaboration helps avoid testing bottlenecks and supports better selector strategies.