Katri Rasio

Testausstrategian laatiminen ja testausautomaation toteutus web-sovelluksen käyttöliittymälle

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

9.5.2022

Tiivistelmä

Tekijä:	Katri Rasio
Otsikko:	Testausstrategian laatiminen ja testausautomaation toteutus web-sovelluksen käyttöliittymälle
Sivumäärä:	41 sivua
Aika:	9.5.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Director Mika Vuorio Lehtori Juha Kämäri

Insinöörityön aiheena oli testausstrategian laatiminen ja testausautomaation toteuttaminen React-käyttöliittymäkirjaston avulla toteutetulle web-sovellukselle. Työn tavoitteena oli mahdollistaa painopisteen siirtyminen manuaalitestauksesta automaatiotestaukseen ja sitä myöten parantaa laatua ja käyttäjäkokemusta.

Projektissa otettiin käyttöön Jest- ja React Testing Library -työkalut. Sovelluksen käyttöliittymäkomponenteille ja apuluokille kirjoitettiin testejä, joilla varmistettiin testaukseen käytettyjen työkalujen toimivuus. Kirjoitetut testit määriteltiin suoritettavaksi osana jatkuvan integraation ja koonnin (CI/CD) käytäntöjä versionhallintaympäristössä. Testausstrategia lisättiin osaksi dokumentaatiota ja käytiin läpi kehitystiimin kanssa.

Työ alkoi testaukseen tarvittavien työkalujen valitsemisella, asentamisella ja määrityksillä. Testattaviksi valittiin ne sovelluksen osat, joiden automaattisesta testaamisesta olisi eniten hyötyä ja jotka edustaisivat sovellusta mahdollisimman laajasti, jotta testien kirjoittaminen olisi tulevaisuudessa mahdollisimman helppoa. Valituille tapauksille kirjoitettiin testit. Lopuksi toteutetut testit asetettiin suoritettaviksi aina lähdekoodin muuttuessa versionhallinnassa.

Testausstrategiassa määriteltiin testauksen periaatteet ja esimerkkitapauksia erityyppisistä testeistä lisättiin osaksi dokumentaatiota tulevan tekemisen tueksi.

Testausautomaatio toimi osana päivittäistä sovelluskehitystyötä. Käyttöönotetut työkalut madalsivat kynnystä kirjoittaa testejä, ja testien automatisointi paransi testaamisen laatua. Ohjelmavirheiden nopeampi havaitseminen helpotti virheiden korjaamista ja pienensi niiden korjaamisen kustannuksia.

Avainsanat: testausautomaatio, ohjelmistotestaus, Jest, React Testing Library

Abstract

Author:	Katri Rasio
Title:	Composing Testing Strategy and Implementing Test Automation in Web Application User Interface
Number of Pages:	41 pages
Date:	9 May 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Software Engineering
Supervisors:	Mika Vuorio, Director Juha Kämäri, Senior Lecturer

---

The aim of the study was to create a testing strategy and implement test automation for a web application built with React JavaScript library. The objective was to enable shifting from manual to automated testing and ultimately improve overall software quality and user experience.

Jest and React Testing Library tools for testing were implemented in the project. Tests were written for some of the user interface components and utility classes to verify the functionality of the selected testing tools. Tests written for the application were defined to be executed as part of the CI/CD (Continuous Integration and Compilation) procedures in a version control environment.

The study began with the selection, installation and configuration of the tools required for testing. The next phase was choosing what to test: the parts of the application that would benefit most from automated testing and those that would represent the application as broadly as possible were selected to make future test writing as easy as possible. The selected tests were written and set to run automatically whenever source code changed in the version control system.

The testing strategy was written to be part of the project documentation. It defined the principles of testing and provided examples of different types of test scenarios to support testing in the future.

The implemented test automation solution was part of the daily software development work. The implementation of testing tools lowered the threshold on test writing and automating the tests improved the quality of testing. Detecting bugs faster made it easier to fix them with lower costs.

Keywords: test automation, software testing, Jest, React Testing Library

Sisällys

Lyhenteet

1 Johdanto 1

2 Ohjelmistotestaus 2

2.1 Testaustasot 3
2.1.1 Yksikkötestaus 3
2.1.2 Integraatiotestaus 4
2.1.3 Järjestelmätestaus 5
2.1.4 Hyväksymistestaus 5
2.2 Ohjelmistotestauksen historia 6
2.3 Ohjelmistotestauksen kustannukset 7
2.4 Testivetoinen kehitys 9
2.5 Testausautomaatio 10
2.6 Haasteet 12

3 Projekti 13

3.1 Asiakas 13
3.2 Tuote 14
3.3 Testauksen lähtötilanne 18
3.4 Testausautomaatiotyökalut 19
3.5 Testausautomaation toteuttaminen 21
3.5.1 Testikirjaston konfigurointi 22
3.5.2 Testejä varten kirjoitetut apuluokat 23
3.5.3 Kirjoitetut testitapaukset 26
3.5.4 Testien ajaminen putkessa 29
3.6 Testausstrategia 30

4 Tulokset 31

4.1 Vaikutus kehitysprosessiin 33
4.2 Työkalujen toimivuus 34
4.3 Haasteet ja rajoitteet 35
4.4 Omat havainnot 36

5 Yhteenveto 37

Lyhenteet

CI/CD: *Continuous integration and Continuous Delivery/Deployment*. Automatisoitu järjestelmä, jossa ohjelmistotuote voidaan esimerkiksi koostaa, testata, toimittaa ja julkaista.

DoD: *Definition of Done.* Etenkin Scrum-ohjelmointiviitekehykseen liittyvä termi, jonka mukaan kehitystehtävä on valmis, kun se on julkaistavissa ja kun käsitys siitä, mitä tämä tarkoittaa, on yhteisesti hyväksytty.

DOM: *Document Object Model*. Dokumenttioliomallin avulla kuvataan esimerkiksi HTML-dokumentin rakenne puumaisena, jota käytetään ohjelmointirajapintana selaimessa esitettävien internetsivujen muokkaamiseen ohjelmallisesti.

E2E: *End-to-end testing.* Päästä päähän -testauksessa testataan sovellusta kokonaisuutena yleensä käyttäjän näkökulmasta, todellista vastaavassa ympäristössä.

GDPR: *General Data Protection Regulation.* EU:n yleisessä tietosuoja-asetuksessa yhtenäistetään EU:n jäsenvaltioiden tietosuojaa koskeva lainsäädäntö, jonka tarkoituksena on vahvistaa EU-alueella asuvien henkilöiden oikeuksia omiin henkilötietoihinsa ja yksinkertaistaa sääntely-ympäristöä liiketoiminnan kannalta.

HTML: *HyperText Markup Language.* Hypertekstin merkintäkieli on avoimen standardin merkintäkieli, jolla voidaan kuvata hyperlinkkejä sisältävää tekstiä rakenteellisena ja jota käytetään verkkosivuilla.

RegExp: *Regular Expression.* Säännöllinen lauseke on tietojenkäsittelyteoriassa käytetty lauseke merkkijonojen vertailuun.

SPA: *Single Page Application*. Web-sovellus, jossa vuorovaikutus käyttäjän kanssa tapahtuu nykyisen verkkosivun dynaamisella uudelleenkirjoittamisella sen sijaan, että ladattaisiin kokonaan uusia, erillisiä web-sivuja.

TDD: *Test-Driven Development*. Ohjelmointikehitysprosessi, jossa kirjoitetaan ensin testi, jossa kuvataan, mitä halutaan saavuttaa ja vasta sen jälkeen toteutetaan testissä määritelty logiikka.

1 Johdanto

Jokaisella on mielipiteitä ohjelmistojen laadusta. Digitalisaation myötä internet ja erilaiset tietotekniset sovellukset ovat osa jokaisen ihmisen arkea joko suoraan tai epäsuorasti. Kun ohjelmistotuote ei toimi, sillä voi olla vaikutuksia monen ihmisen elämään. Yksi tehokkaimmista keinoista parantaa ohjelmiston laatua on sovelluksen testaaminen ennen julkaisua.

Fintrafficin Liikennetilanne-palvelun käyttöliittymäuudistus aloitettiin alkuvuodesta 2021 ja vanhasta käyttöliittymästä päästiin luopumaan saman vuoden loppupuolella. Uudelle käyttöliittymälle ei kuitenkaan kirjoitettu testejä, vaan ennen julkaisua kehitystiimi ja asiakas testasivat sovellusta manuaalisesti käyttöliittymää käyttämällä.

Järjestelmällinen manuaalitestaus on hyvä lisä automaattiselle testaukselle, mutta mikäli se on käytössä ainoana testimetodina, kasvaa inhimillisten virheiden mahdollisuus, sillä ihminen ei kykene toistamaan operaatioita samanlaisella tarkkuudella kuin ohjelmallinen testaus. Ohjelmistotuotteen laajuuden ja monimutkaisuuden kasvaessa kokonaisuuden hahmottaminen tulee kehittäjälle koko ajan vaikeammaksi, jolloin virheellisen logiikan mahdollisuus kasvaa. Sama ongelma koskee myös kehittäjien suorittamaa manuaalista käyttöliittymätestausta, sillä mikäli kuva kokonaisuudesta ei vastaa todellisuutta, ei sitä myöskään testattaessa osata huomioida.

Liikennetilanteessa, kuten jokaisessa ohjelmistoprojektissa, on jatkuva tarve parantaa ohjelmiston laatua ja sitä myöten käyttäjien kokemusta palvelusta. Testausautomaation lisääminen osaksi projektia on yksi keino päästä lähemmäksi tätä tavoitetta.

Työn tavoitteena on luoda testausstrategia, jossa määritellään pelisäännöt sille, miten ja miksi testausta tehdään. Osana työtä projektissa otetaan käyttöön testauksen mahdollistava ympäristö ja määritellään testausautomaatio osaksi jatkuvan integraation ja koonnin (CI/CD) käytäntöjä versionhallintaympäristöön.

Osana työtä sovellukselle on myös tarkoituksena luoda joitakin esimerkinomaisia testikokonaisuuksia avuksi tulevaisuuden kehitystehtävissä.

2 Ohjelmistotestaus

Pattonin mukaan ohjelmavirheeseen liittyy kiinteästi ohjelmistotuotteen ja kehitystehtävien määrittelyt. Ohjelmavirhe voi tapahtua, kun sovelluksessa tapahtuu jotain, mitä määrityksen mukaan ei pitäisi tapahtua tai mitä ei ole määritelty, tai kun sovelluksessa ei tapahdu jotain, mitä määritysten mukaan pitäisi tapahtua. Ohjelmavirheeksi voidaan määritellä myös, mikäli sovellus on hankala, hidas tai muuten huono käyttää. Ohjelmointivirheen syy voi löytyä määrityksestä, suunnittelusta, koodista tai muusta syystä. [Patton, 2005.]

Ohjelmistotestauksen tarkoitus on löytää virheitä ohjelmistotuotteesta. Ohjelmavirheiden ja puutteellisen testaamisen vaikutukset voivat olla katastrofaalisia, kuten esimerkiksi Yhdysvaltojen Patriot-ohjuspuolustusjärjestelmän virhe, joka aiheutti 28 sotilaan kuoleman Saudi-Arabiassa vuonna 1991 [Patton, 2005].

On kuitenkin hyvä pitää mielessä, kuten Dijkstrakin [1970: 11] on todennut, että ohjelmistotestausta voidaan käyttää ohjelmavirheiden esiin tuomiseen, mutta ei todistuksena siitä, etteikö virheitä olisi.

Koskela [2007] huomauttaa, että vaikka ohjelmistokehityksessä on otettu edistysaskeleita, aiheuttaa ohjelmistojen laatu silti ongelmia. Syyksi laatuongelmille hän esittää varhaiset julkaisut, ohjelmistojen määrän kasvun ylipäätään ja alati kasvavan määrän uusia teknologioita.

Ohjelmistotestaukseen liittyy kiinteästi määrittelyt. Olennainen osa kehitystehtävää on valmiin määrittely (DoD, Definition of Done), joka on kehitystiimin yhteinen määrittely sille, mitä ominaisuuksia koodissa täytyy olla toteutettuna, jotta sen voidaan määritellä olevan valmis. Testeihin voi liittyä

myös omia määrityksiään, mutta usein pohjana on testattavan ominaisuuden kehitysvaiheen määrittelyt.

Testaamisen laatua voidaan mitata erilaisilla mittareilla riippuen testaustavasta, ohjelmistoprojektin ja testaamisen vaiheesta tai määrittelyistä. Yksikkö- tai integraatiotestien osalta yleisimmin käytetyistä mittareista on testikattavuus. Testikattavuuden käsitteen alle listataan usein joukko erilaisia kattavuusmittareita, sillä mittareilla voidaan mitata funktiokattavuutta, rivikattavuutta, haarakattavuutta, ehtokattavuutta ja polkukattavuutta. Käytännössä näitä tarkastellaan usein joukkona, joten on hyvä varmistaa, että testitapauksissa huomioidaan vaihtoehtoiset suorituspolut ja että testit käsittelevät testattavaa komponenttia kaikilla edellä mainituilla mittareilla tarkasteltuna mahdollisimman laajasti. [3.1. Ohjelmistojen testausta, 2016.]

2.1 Testaustasot

Ohjelmistoa voidaan testata eri tasoilla. Sovelluksesta voidaan testata pientä osaa, kuten esimerkiksi yksittäisen komponentin toimintaa tai koko sovellusta. Perinteisesti testaaminen jaotellaan yksikkötestaukseen, integraatiotestaukseen, järjestelmätestaukseen ja hyväksyntätestaukseen. Parhaimpiin testituloksiin päästään, kun testausta tapahtuu mahdollisimman monella tasolla.

2.1.1 Yksikkötestaus

Yksikkötestauksessa tutkitaan pienimpien, itsenäisesti toimivien koodiosioiden toimintaa ja määritetään, toimivatko ne määrityksien mukaisesti. Yksikkötestauksen katsotaan olevan arkkitehtuurillisesti matalimman tason testausta. Yleisesti katsotaan, että kehittäjän tulee laatia yksikkötestit komponentille ennen kuin kehitystehtävä voidaan määritellä valmiiksi [Elfriede, 2002]. Yksikkötestaus on sovellettavissa suurimpaan osaan ohjelmistoprojekteja.

Yksikkötesteillä voidaan varmistaa, että ohjelmisto on perusmäärittelyiden mukainen ennen integraatio- tai järjestelmätestausta. Hyvin toteutettu yksikkötestaus tukee myöhempiä, korkeamman tason testausvaiheita, mutta ylimalkaisesti kirjoitetut yksikkötestit eivät välttämättä ole hyödyllisiä. Yksikkötestejä suoritettaessa havaittu virhe on helpompi ja halvempi korjata kuin myöhemmin havaittu virhe, sillä ohjelmavirhe voidaan korjata hyvissä ajoin ennen julkaisua. [Elfriede, 2002.]

Yksikkötesteissä testattava komponentti eristetään ympäristöstään, jolloin tarvitaan erilaisia keinoja korvata ulkoisia riippuvuuksia [Özturk, 2020, Wacker, 2015]. Esimerkiksi sovelluksen suorittamien ulkoisten verkkokutsujen sijaan voidaan käyttää mock-toiminallisuutta, jossa ulkoisen verkkokutsun vastaus annetaankin staattisena sisältönä ja näin päästään testaamaan testattavan komponentin toimintaa ilman verkkoyhteyksiä.

On suosittua kirjoittaa yksikkötestit joko samaan aikaan koodin kanssa tai jopa ennen sitä. Testivetoisessa kehityksessä (TDD) kehitystehtävän määrittelemisen jälkeen kirjoitetaan määrityksiin vastaavat testityngät ennen varsinaista sovelluskehitystä.

2.1.2 Integraatiotestaus

Yksikkötestauksesta seuraavan tason testejä kutsutaan integraatiotesteiksi. Integraatiotestaukselle ei ole tarkkaa, yksimielistä määritelmää, mutta laajimman määritelmän mukaan se on yksikkötestejä laajempaa, mutta järjestelmätestejä suppeampaa testausta. Yleisimmin integraatiotesteissä testataan kahden tai useamman komponentin yhteistoimintaa. [Axelrod, 2018: 122.]

Integraatiotestauksessa tilanne on lähempänä todellista käyttötilannetta sovelluksessa, kun yksittäiset komponentit ovat vuorovaikutuksessa toistensa kanssa. Integraatiotesteissä löydettyjen virheiden alkuperän selvittäminen voi

olla hankalampaa kuin yksikkötestauksessa, sillä testattavan koodin määrä on suurempi. [Özturk, 2020.]

2.1.3 Järjestelmätestaus

Järjestelmätestaus, josta käytetään myös termiä päästä päähän -testaus (E2E), testaa sovellusta kokonaisuutena yleensä käyttäjän näkökulmasta todellisessa ympäristössä. Järjestelmätestauksessa ei tulisi käyttää sijais- tai testikomponentteja. [Özturk, 2020.]

Päästä päähän -testaus voi tarjota erittäin hyödyllisiä tuloksia, kun päästään lähelle käyttäjän kokemusta sovelluksesta, mutta E2E-testien konfiguroiminen voi olla hankalaa ja niiden suorittaminen hidasta. Käyttöliittymässä tehdyt testit voivat olla epäluotettavia eli testi voi onnistua tai epäonnistua, vaikka koodissa ei olisi tapahtunut mitään muutoksia. Syitä tähän voi olla monia: esimerkiksi ulkoiset riippuvuudet voivat lakata toimimasta tai niiden toiminnassa voi olla odottamattomia viiveitä [The Code Gang, 2017; End to end -testaus, 2022].

2.1.4 Hyväksymistestaus

Hyväksymistestauksen lähtöasetelma poikkeaa alempien tasojen testauksesta. Sen ideana on ohjelmavirheiden löytämisen sijaan varmistaa sovelluksen oikea toiminta sovelluksen käyttötapauksia kuvaavien käyttäjätarinoiden pohjalta. Käyttäjätarinoissa kuvataan käyttäjän näkökulmasta tehty toimenpide ja odotettu lopputulos. Hyväksyntätestauksen tarkoitus on varmistaa, että käyttäjän tarvitsemat toiminnot toimivat määritelmien mukaan. Hyväksymistestausta tehdään perinteisesti siinä vaiheessa, kun tuote tai kehitettävä ominaisuus on melko lailla valmis. Hyväksymistestaajina voivat toimia kehittäjien ja testaajien lisäksi myös sovelluksen loppukäyttäjät. [Özturk, 2020.]

Cimpermanin mukaan prosessia voidaan parantaa vielä sillä, ettei hyväksymistestausta jätetä suoritettavaksi vain kehitystyön loppuvaiheeseen

juuri ennen julkaisua ja että käyttäjät otetaan sovelluskehitysprosessin suunnittelu- ja järjestelmätestausvaiheisiin mukaan. Tällä tavoin on nopeampaa, helpompaa ja edullisempaa löytää virheellisiä oletuksia, vääriä määrittelyitä ja toimimattomia lähestymistapoja. [Cimperman, 2016.]

2.2 Ohjelmistotestauksen historia

Axelrod [2018: 30] esittää, että perinteisessä vesiputousmallissa, jossa ohjelmistotuote suunniteltiin tarkasti ja tehtiin käytännössä valmiiksi ennen julkaisua, testaaminen oli erillinen vaihe ennen julkaisua, joka suoritettiin kertaalleen, jolloin ei syntynyt tarvetta testausautomaation kehittämiselle. Vesiputousmallia on toteutettu eri projekteissa eri tavoin ja joissain tapauksissa yksikkötestausta suoritettiin jo osana toteuttamisvaihetta erillisen testausvaiheen lisäksi [SLDC Waterfall Model: The 6 phases you need to know about, 2019].

Vesiputousmallissa testaamisen yhteydessä havaitut, oletettavasti pienet ohjelmointivirheet korjattaisiin ennen julkaisua, sillä kun suunnittelutyö oli tehty tarkasti, ei suuria ohjelmavirheitä pitäisi syntyä. Käytännössä kuitenkin huomattiin, ettei edes vesiputousmalliin kuuluva huolellinen suunnittelu estänyt virheiden syntymistä. Ohjelmistoprojektien koon ja monimutkaisuuden kasvaessa tuli riittävän tarkasta suunnittelusta mahdotonta. [Axelrod, 2018: 30.]

Vesiputousmallista siirryttiin hiljalleen ketterään kehitykseen, jossa mittavasta etukäteissuunnittelusta luovuttiin, julkaisuista tehtiin pieniä ja niitä tehtiin nopealla tahdilla. Paradigman muutos merkitsi myös sitä, että testauskertojen määrä kasvoi sitä mukaa, kun julkaisujen määrä kasvoi.

Erilaisten ohjelmistotuotantoprosessien kanssa on käytetty erilaisia testausstrategioita, mutta ketterän kehityksen myötä on syntynyt tarve automaattiselle testaukselle, sillä ohjelmistoprojekteissa, joissa noudatetaan edes löyhästi ketterän kehityksen periaatteita, ohjelmistotuotteen julkaisut tehdään pienissä erissä tiheällä tahdilla. Julkaisutahdin tiivistymisen myötä

tarve testauksen muuttaminen manuaalisesta automaattiseksi oli tarpeen, sekä taloudellisista että laadullisista syistä.

2.3 Ohjelmistotestauksen kustannukset

Eri lähteiden mukaan ohjelmistotestauksen hinta vaihtelee 20-40 %:n välillä kehitystyön kokonaiskustannuksista, mutta hintaa on vaikeaa määrittää riippuen projektin ja sen testaamisen lähtötilanteesta. Perinteisesti testaamisen osa-alueista suurimman hintalapun saa testitapausten kirjoittaminen ja virheiden löytämiseen tähtäävien testaustyökalujen käyttöönotto. [How Much Does Software Testing Cost? 9 Proven Ways to Optimize it, 2021; Software Testing Accounts to What Percent of Development Cost?, 2021.]

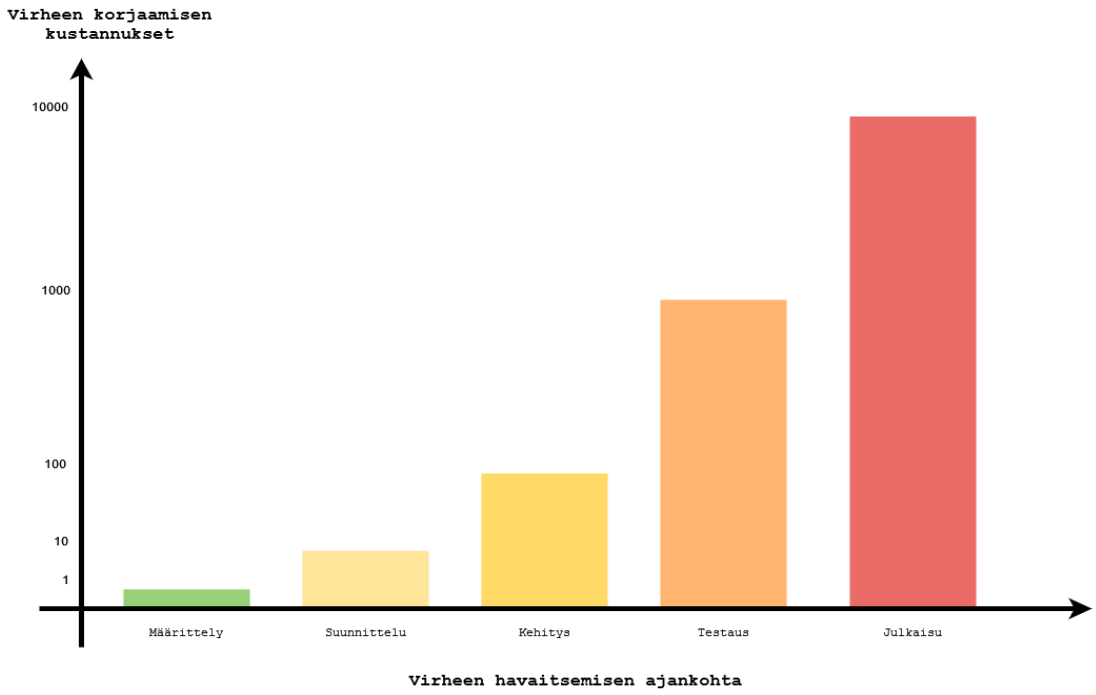
Testaukseen käytettyjen resurssien katsotaan olevan kustannustensa arvoisia, sillä pidemmällä aikavälillä tarkasteluna voidaan saada säästymään paitsi rahaa ja aikaa, myös palveluntarjoajan tai ohjelmoinnista vastaavan tahon maine.

Jos tarkastellaan testauksen hintaa yhdessä ohjelmavirheen aiheuttamien mahdollisten kustannusten kanssa, tulee kustannusten määrittelystä hankalampi yhtälö. Ohjelmavirhe voi olla luonteeltaan miltei mitätön tai erittäin kriittinen. Ohjelmavirheiden aiheuttamissa kustannuksissa voi olla kyse varallisuuden hupenemisen lisäksi esimerkiksi maineen menettämisestä, rikosoikeudellisesta vastuusta tai ihmishengistä.

Usein kustannukset ovat kuitenkin jonkinlainen yhdistelmä edellisistä, kuten teleyhtiö Elisan tapauksessa. Vuonna 2004 Elisalla tehtiin päätös useiden laskutusjärjestelmien yhdistämisestä yhdeksi kokonaisuudeksi ja muun muassa sadantuhannen virheellisen laskun lähettämisen myötä asiaa päädyttiin käsittelemään lopulta välimiesoikeudessa [Alkio, 2010].

Pattonin [2005] väitteen mukaan ohjelmavirheen hinta yleensä kasvaa sen myötä, mitä myöhäisemmässä vaiheessa ohjelmistoprojektia se löytyy [kuva 1]. Syitä tähän voi olla useita: esimerkiksi virheellisen logiikan päälle rakennettu

muu toiminnallisuus joudutaan myös korjaamaan, mikä merkitsee, että korjaaminen on monimutkaisempaa ja vie enemmän aikaa. Ohjelmavirhe on usein helpompi ja nopeampi korjata, mikäli sen aiheuttaneesta lähdekoodin muutoksesta on vähemmän aikaa.

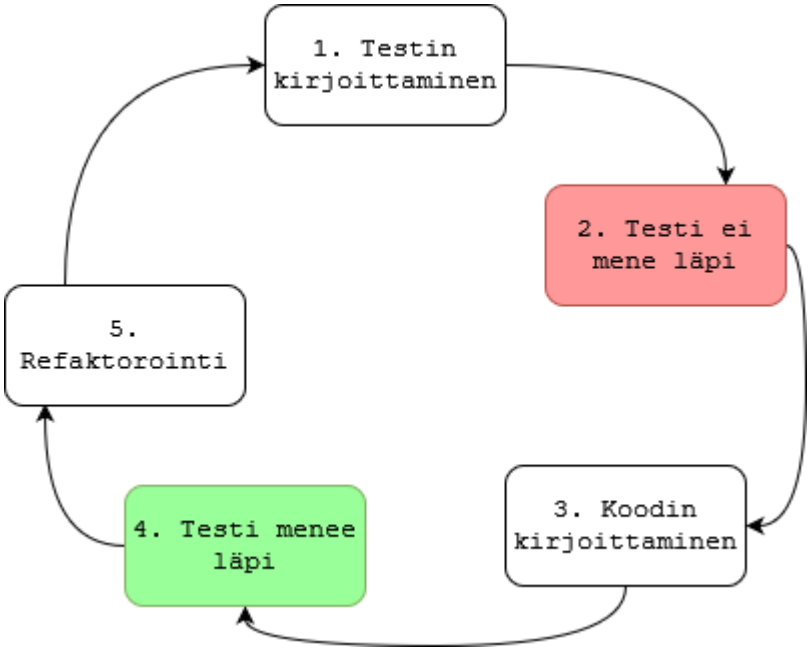


Kuva 1: Ohjelmavirheen löytymisen hinta [Patton, 2005, muokattu]

Testaamisen huomioiminen ohjelmistotuotantoprojektin varhaisessa vaiheessa tekee testaamisesta helpompaa ja kustannustehokkaampaa [Elfriede, 2002; How Much Does Software Testing cost? 9 Proven Ways to Optimize it, 2021]. Testauksen lisääminen projektiin jälkikäteen saattaa aiheuttaa koodin mittavaa uudelleenkirjoittamista, jotta testaaminen olisi ylipäätään mahdollista, sillä hyvin sovelluksessa toimiva komponentti ei välttämättä ole helposti tai kokonaisvaltaisesti testattavissa.

2.4 Testivetoinen kehitys

Sovelluskehitystä voidaan tehdä testivetoisesti. Erilaisia testivetoisia kehitysprosesseja löytyy useampia, mutta niiden pääasiallinen idea on siinä, että ennen ohjelmakoodin kirjoittamista kirjoitetaan testitapaukset, joissa määritellään, mitä halutaan saavuttaa ja vasta sitten ohjelmalogiikka. Koskelan määritelmässä testivetoiselle kehitykselle (TDD) kehotetaan kirjoittamaan koodia vain epäonnistuneen testin korjaamiseksi [Koskela, 2007].



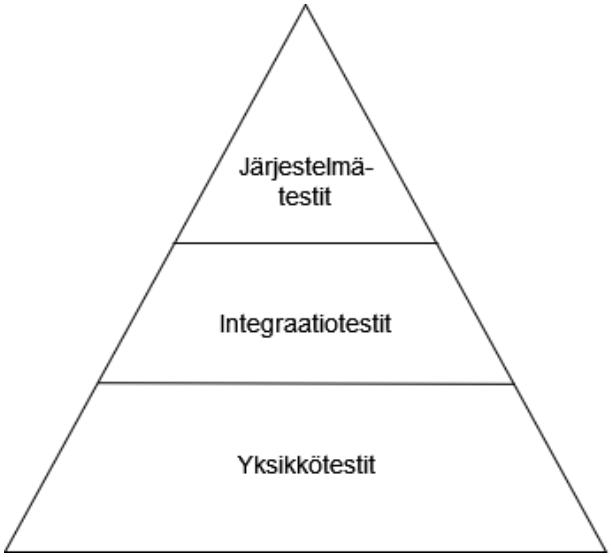
Kuva 2: Testivetoisen kehityksen prosessi pyörii kehää

Testivetoisen kehityksen kaaviossa [kuva 2] on kuvattu kehitysprosessi, jossa koodin refaktorointi, eli uudelleenjäsentely ilman toiminnallisuuden muutoksia on viimeisenä vaiheena ennen uuden kierroksen alkua. Refaktoroinnilla pyritään selkeyttämään koodia ja mahdollistamaan sen helpompi ylläpidettävyys. Testivetoisessa kehityksessä pienten muutosten jälkeen suoritetaan uudelleenkirjoitettua koodia koskeva testi ja varmistetaan, että sovelluksen toiminta ei ole muuttunut. Testivetoisen kehityksen eduiksi listataan pienempi määrä ohjelmavirheitä, paremmin suunniteltu ohjelmisto, parempilaatuinen

koodi sekä ohjelmavirheiden löytämisen ja korjaamisen helppous [Koskela, 2007].

2.5 Testausautomaatio

Toistuvien prosessien automatisointi alkaa olla tietotekniikan alalla ennemminkin sääntö kuin poikkeus, sillä asiantuntijoiden kallisarvoista aikaa on turha kuluttaa sellaiseen, mikä voidaan ulkoistaa automatisoiduksi prosessiksi. Yleisesti käytettyjen ketterän kehityksen menetelmien mukaisissa projekteissa pyritään tekemään usein pienikokoisia julkaisuja. Lähdekoodin muuttuessa testit on suoritettava aina uudelleen, mikä aiheuttaa suuren määrän testauskertoja ja luo näin ollen tarvetta automatisoinnille. [Bose, 2021; Axelrod, 2018: 16-17.]



Kuva 3: Alun perin Mike Cohnin esittelemän testiautomaation pyramidimallin mukaelma [Wacker, 2015]

Testausautomaatiota voidaan tarkastella testaustasojen kautta testiautomaation kuvassa 3 esitellyn pyramidimallin avulla. Pyramidimallin perusideana on, että testauksen perusta muodostuu yksikkötesteistä, jotka ovat kustannustehokkaampia tuottaa ja nopeampia suorittaa. Kokonaisuutta kasvatetaan integraatio- ja järjestelmätesteillä, jotka ovat sitä hitaampia

suorittaa ja kalliimpia laatia, mitä korkeammalle pyramidissa noustaan. Pyramidin muodolla havainnollistetaan myös tuotettujen testitapausten kappalemäärää: pohjalla olevia yksikkötestejä pitäisi olla eniten ja huipun järjestelmätestejä vähiten. [Bilski & Mandry 2020; Wacker, 2015.]

Wackerin [2015] mukaan teknologiayritys Googlen suositus olisi, että 70% testeistä olisi yksikkötestejä, 20% integraatiotestejä ja 10% järjestelmätestejä, kun taas Dijkstran [2014] mukaan tarkkoja osuuksia eritasoisille testeille on mahdotonta esittää, sillä sekä sovellus että kehittäjien osaamisen taso vaikuttavat kokonaisuuteen.

Vaikka mallia kritisoidaan siitä, että se on liiankin yksinkertainen, se voi silti olla käyttökelpoinen lähtökohta testaukselle [Vocke, 2018; Bilski & Mandry, 2020]. Vocken [2018] mukaan pyramidimallista kannattaa yksinkertaisuudestaan huolimatta omaksua ajatus siitä, että mitä ylemmäs pyramidissa noustaan, sitä vähemmän testejä kirjoitetaan, ja että testitapausten joukossa pitäisi olla sekä eristettyjä että kokonaisuuteen integroituja testejä.

Wacker [2015] kyseenalaistaa ylipäätään järjestelmätestauksen hyödyllisyyttä, sillä vaikka sillä voidaan emuloida käyttäjän toimia testattavassa sovelluksessa, ovat yksikkötestit kuitenkin usein käytännössä hyödyllisempiä. Wacker varoittaa myös muokkaamasta pyramidin painotuksia, sillä useimmissa tapauksissa kuvan 3 mukainen testipyramidi takaa testatessa vakaimman tuloksen.

Koska yksikkötestit ovat luonteeltaan eristettyjä ja niissä testataan kenties suurenkin ohjelmakokonaisuuden yksittäisiä osia, voi ohjelmavirheen paikantaminen ja korjaaminen olla nopeampaa. Single Page Application (SPA)-käyttöliittymäkirjastojen komponentteihin kirjoitetut testit voidaan käsittää alimman tason yksikkötesteiksi, mikäli niitä voidaan testata muusta käyttöliittymästä eristettyinä yksiköinään. Joskin määrittely yksikkö- ja integraatiotestien välillä on häilyvää [Vocke, 2018; Testing Overview, 2019].

Integraatiotesteillä voidaan saada useampaa kokonaisuutta koskeva ohjelmavirhe kiinni pienemmällä vaivalla kuin järjestelmätestauksen avulla.

Pyramidin alimman tason yksikkötestien ylläpitäminen on huomattavasti helpompaa kuin laajasti käyttäjän toimia imitoivien järjestelmätestitapausten, mikäli sovellukseen tehdään mittavia muutoksia [Vocke, 2018]. Toisaalta, mikäli ohjelmistotuotteen rakenne muuttuu, voivat pyramidin alimman tason yksikkötestit epäonnistua, vaikka kokonaisuus olisikin sama kuin ennen [Bilski & Mandry, 2020].

Eritasoiset testit kannattaa suorittaa erillisissä vaiheissa. Yksikkö- ja integraatiotestit voidaan suorittaa osana jatkuvan koonnin (CI)-järjestelmää lähdekoodin päivittyessä versionhallinnassa. Järjestelmätestit taas esimerkiksi ajastetaan tuotannon kaltaisessa ympäristössä ja hyödynnetäään niitä palvelun toiminnan monitoroinnissa. Näin saadaan hyödynnettyä eritasoisia testejä niiden ehdoilla. [Jung, 2021.]

2.6 Haasteet

Ohjelmistotuotannossa on ollut aiemmin vahvaa roolijakoa kehittäjiin ja testaajiin ja perinteisesti näiden ryhmien välinen kommunikaatio on ollut vähäistä. Kun kehitystehtävä on valmis, on se siirretty testaajille sen suuremmitta puheitta ja testaajat ovat palauttaneet vain testin lopputuloksen takaisin kehittäjille. [Perry & Rice, 2013.]

Rani [2020] esittää, että ketterän ohjelmistokehityksen menetelmien yleistyttyä testaajan rooli on muuttunut. Aiemmin vesiputousmallin mukaan toimineissa projekteissa testaajan tehtävä oli kertoa, menikö testi läpi vai ei, mutta ketterässä kehityksessä palautteenanto pitää olla nopeaa ja jatkuvaa.

Teknologioiden kehittyminen on myös muuttanut testaajien työskentelyä, sillä esimerkiksi nykyisin runsaasti käytetyt pilvipalvelut vaativat osaamisen jatkuvaa päivittämistä. Erilaisten päätelaitteiden ja verkkoselainten lukumäärä aiheuttaa oman haasteensa testaamiselle, sillä sovelluksen toiminta voi olla erilaista eri laitteilla. [Rani, 2020.]

Testiympäristön rakentamisessa oman haasteensa luo henkilötietojen käsittelyä koskeva EU:n yleinen tietosuoja-asetus (GDPR) [Rani, 2020]. Aiemmin testeissä tarvittava data saatettiin kopioida tuotannossa käytöstä olevasta datasta, mutta nykyisin oikeita henkilötietoja sisältävää dataa ei voi enää tallentaa ilman laissa määriteltyä perustetta [EU:n yleinen tietosuoja-asetus: II Luku, 5 artikla]. Asetusta noudattaessa testiaineistoja joudutaan generoimaan uusilla tavoilla.

Testaaminen näyttäytyy pintapuolisesti pelkkänä kulueränä, sillä se ei tuota suoraa taloudellista hyötyä. Shah [2021] esittää, että ohjelmistotestauksella on useita positiivisia mitattavia vaikutuksia, kuten projektin onnistuminen, nopeampi julkaisuaika tai asiakastyytyväisyys.

Sovelluksen toiminnan tai ulkoasun muuttuessa testien ylläpitäminen vaatii resursseja ja tulee huomioida kehitystehtävien työmääräarvioissa. Samojen testien toistuvan korjaamisen perustelu voi olla liiketaloudellisesta näkökulmasta haastavaa.

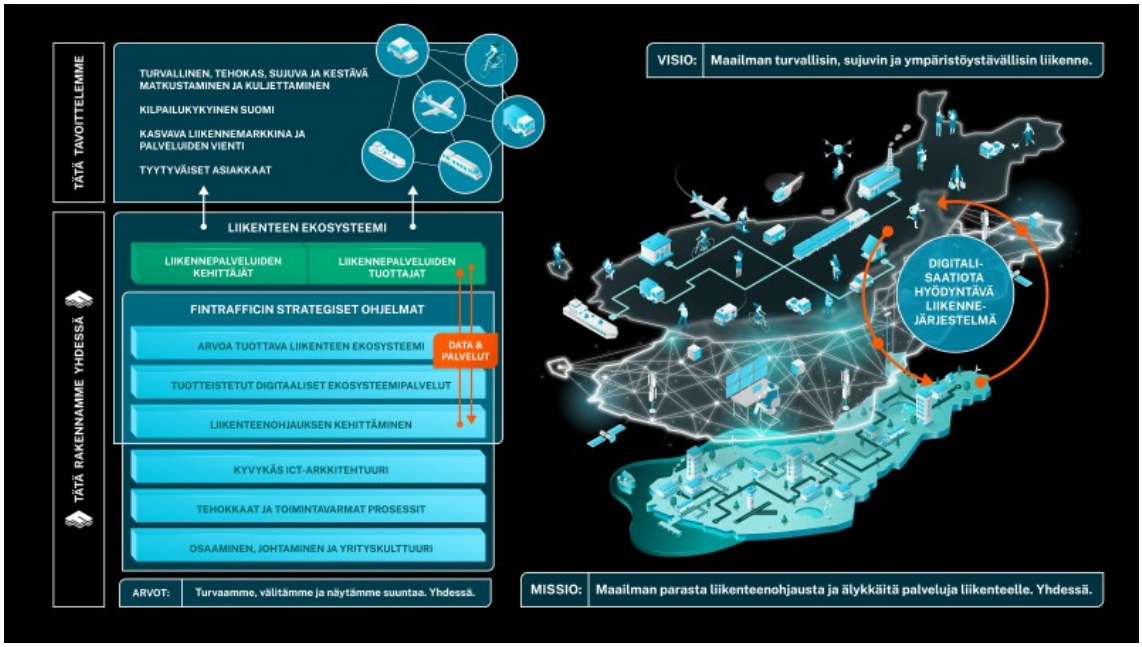
3 Projekti

Tässä dokumentissa kuvattu projekti on toteutettu vuoden 2022 tammi-huhtikuun aikana. Tavoitteena oli toteuttaa testausautomaatio osaksi Fintrafficin Liikennetilanne-verkkosivuston kehitysympäristöä ja laatia testausstrategia, jossa määritellään pelisäännöt testaamiselle.

3.1 Asiakas

Fintraffic on Liikenne- ja viestintäministeriön omistajaohjauksessa toimiva konserni, joka tarjoaa ja kehittää liikenteenohjauksen ja -hallinnan palveluita sekä tuottaa ja hallinnoi liikenteestä syntyvää dataa. Fintraffic tarjoaa Digitraffic-palvelun kautta ajantasaista liikennetietoa avoimena datana ja avoimista rajapinnoista. Konsernin liiketoiminta-alueita ovat lennonvarmistus, meri-, raide- ja tieliikenteenohjaus sekä liikenteen dataekosysteemi, joista jälkimmäistä

tehdään useiden liikennealan toimijoiden kanssa. [Fintraffic - Turvallista ja sujuvaa liikennettä 2021, Vuosikatsaus 2021, 2022.]



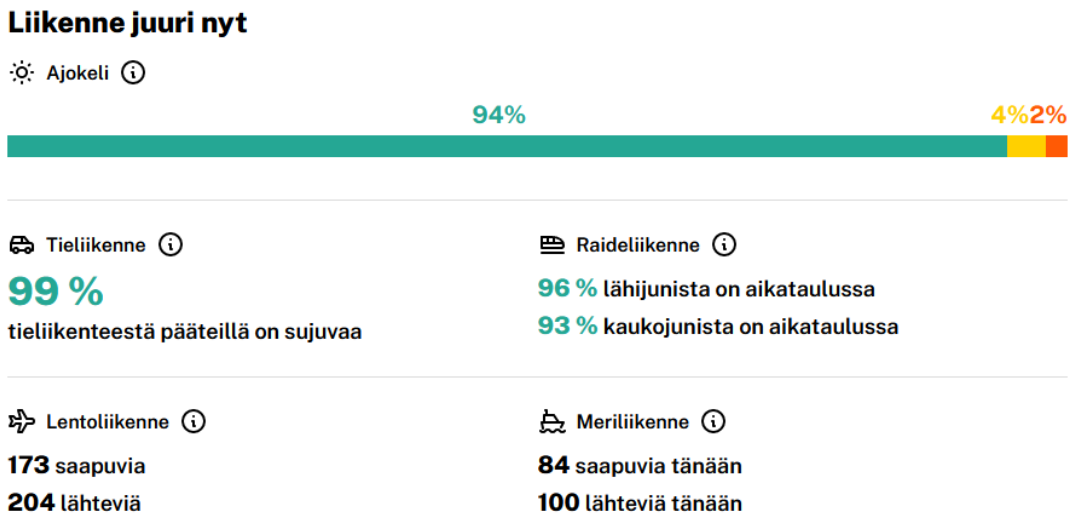
Kuva 4: Fintrafficin visio, missio ja arvot [Fintrafficin tarkoitus: yhteinen suuntamme, 2021]

Fintrafficin tavoitteena on hoitaa liikkuminen turvallisesti, sujuvasti ja tehokkaasti. Fintraffic yhdistää liikenteen toimialan infrastruktuuria liikennedataan ja liikenteen toimijoihin ja loppukäyttäjiin. Konsernin tarkoituksena on tukea niin yksityishenkilöiden liikkumista, turvallisuusviranomaisten toimintaa kuin liike-elämän tarpeita. [Kuva 4; Strategia 2022-2026, 2021; Fintrafficin tarkoitus: yhteinen suuntamme, 2021.]

3.2 Tuote

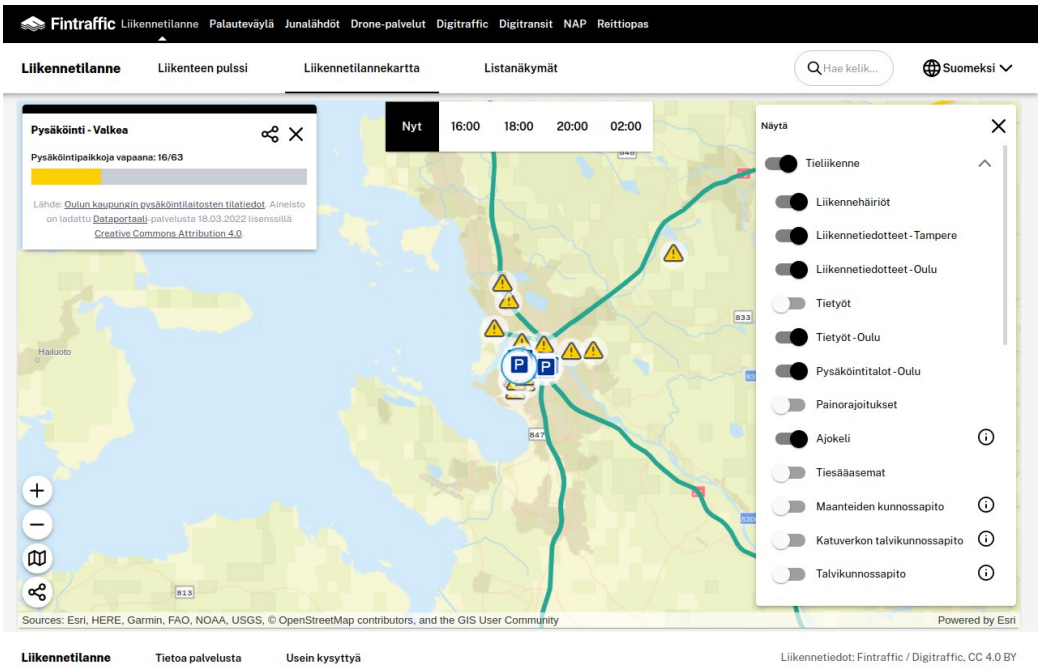
Liikennetilanne on Fintrafficin palvelu, joka tarjoaa käyttäjilleen ajankohtaista tietoa liikenteestä visuaalisessa muodossa. Sen käyttäjien joukkoon lukeutuvat niin autoilijat, joukkoliikenteen käyttäjät, media kuin erilaiset sidosryhmätkin. Palvelun sivukatseluiden määrä vuonna 2021 oli yli neljä miljoonaa, ja se kaksinkertaistui edellisestä vuodesta [Vuosikertomus 2021, 2022].

Liikennetilanne-palvelussa käytetään pääasiassa Fintrafficin, Väyläviraston sekä Traficomin tarjoamaa dataa. Palvelu tarjoaa tietoa tieliikenteen tilanteesta, ennusteista, sujuvuudesta, häiriöistä ja rajoituksista. Rautatie- ja lentoliikenteen osalta esitetään asemien ja satamien aikataulutietoja sekä vesillä liikkumisen varoituksia. Palvelusta löytyy kelikamerakuvia, sähköauton lataus- ja kaasutankkausasemaverkosto sekä kuntien tarjoamaa liikennedataa.



Kuva 5: Liikennetilanne: Liikenteen pulssi -näkymän avainluvut tarjoavat kuvan Suomen liikenteestä yhdellä silmäyksellä.

Verkkosovellus koostuu kolmesta eri päänäkymästä, joissa liikenteeseen liittyvää dataa tarjoillaan hiukan eri muodoissa. Liikenteen kokonaistilanteen saa omaksuttua nopeasti Liikenteen pulssi -näkymässä [kuva 5], jossa liikennedataa esitetään avainlukujen ja kevennetyn kartta- ja listaominaisuuden muodossa.



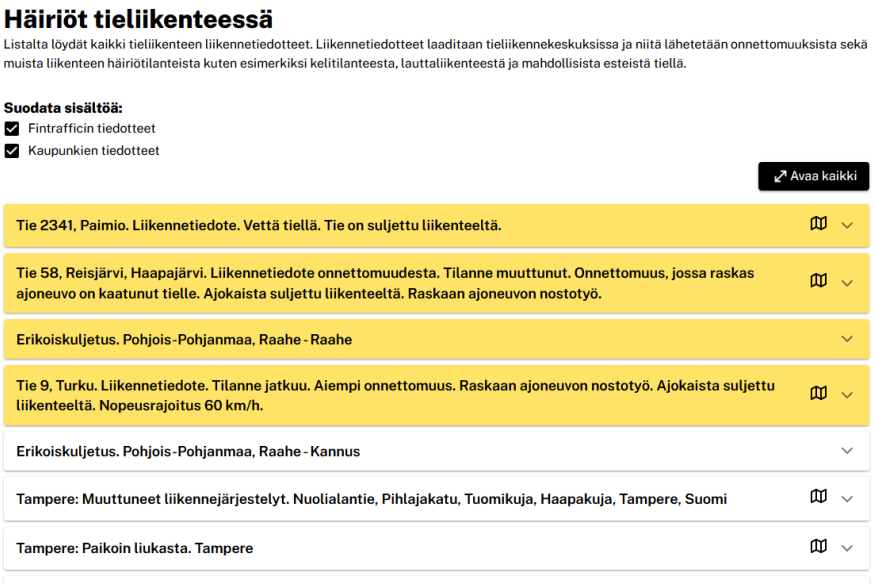
Kuva 6: Liikennetilanteen karttanäkymään voi valita haluamansa karttatasot näkyville

Pulssi-näkymän lisäksi tietoa tarjotaan kartalla [kuva 6], josta käyttäjä voi valita haluamansa datalähteet näkyviin. Lisäksi palvelusta löytyy listanäkymä [kuva 7], josta kymmenen eri otsikon takaa löytyy listamuotoista dataa joko yhdestä tai useammasta lähteestä.

Kuva 7: Listanäkymät-sivulta löytyvät otsikot

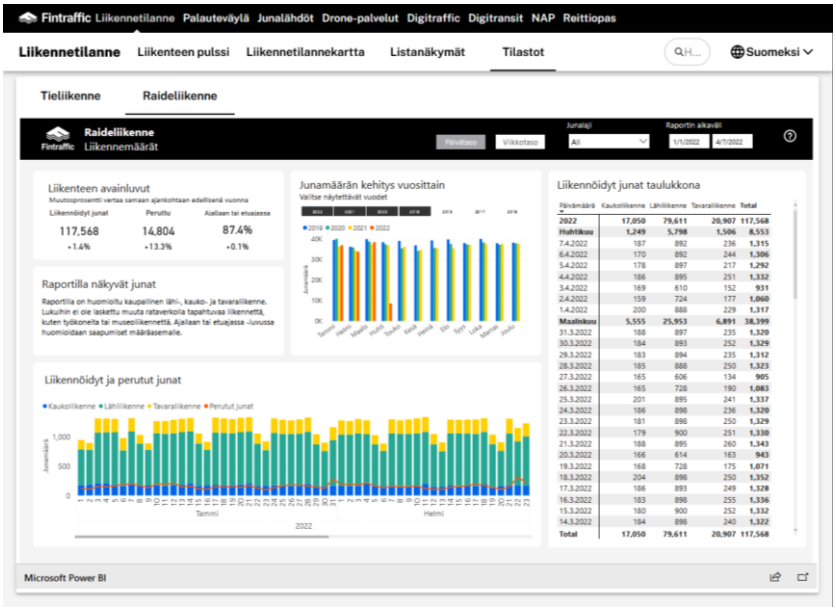
Yksi Fintrafficin tavoitteista vuodelle 2021 oli liikenteen dataekosysteemin toiminnan käynnistäminen ja osana sitä kaupunkien tilannekuvadatan jakaminen [Vuosikertomus 2021, 2022]. Liikennetilanne-palveluun on tuotu

kaupunkien omaa dataa liikennehäiriöistä kuten kuvassa 8 sekä tietöistä, kunnossapidosta ja pysäköintitilanteesta.



Kuva 8: Häiriöt tieliikenteessä- ja Tietyöt-listoilta löytyy Fintrafficin tiedotteiden lisäksi kaupunkien omia tiedotteita

Tilastot-näkymästä löytyy tie- ja raideliikenteen tilastotietoja, joiden tarkasteluajankohta ja muu sisältö ovat käyttäjän muokattavissa [kuva 9].



Kuva 9: Liikennedataa esitetään useilla eri tavoilla

Sovelluksen käyttöliittymä on uusittu hiljattain käyttämään uusia sovelluskirjastoja, jotta sivusto olisi modernimman näköinen ja kokonaisuutta olisi helpompi ylläpitää ja kehittää. Uusi versio sovelluksesta on toteutettu käyttäen React-käyttöliittymäkirjastoa, Material UI -käyttöliittymäkomponenttikirjastoa, OpenLayers-karttakirjastoa sekä Typescript-ohjelmointikieltä. Käyttöliittymäuudistuksen yhteydessä sovellukseen lisättiin myös Liikenteen pulssi ja Tilastot-näkymät, joista ensimmäinen toimii sovelluksen etusivuna. Käyttöliittymäuudistus on aloitettu 2020-2021 vuodenvaihteessa.

3.3 Testauksen lähtötilanne

Liikennetilanne-projekti siirtyi nykyiselle kehitystiimille kesken käyttöliittymäuudistuksen ja siinä vaiheessa testausta ja testausautomaatiota ei oltu huomioitu. Sovelluskehitys tiiminvaihtoon asti oli toteutettu varsin ripeällä aikataululla.

Ohjelmistoprojekteissa testien puuttumiseen voi olla monia syitä. Testit eivät tuota suoraa liiketaloudellista hyötyä, pikemminkin päinvastoin. Käytännössä myös usein projekteissa on aikataulupainetta, jolloin testaukselle ei tunnu jäävän aikaa, varsinkaan jos testaus ei ole ollut alusta alkaen osana määrittelyitä tai ohjelmointiprosessia. Yksi mahdollinen syy testien kirjoittamatta jättämiselle voi olla myös se, että projektissa käytetyt teknologiat ovat vielä suhteellisen tuoreita, eikä niiden testaamiseen käytetyt työkalut ole täysin vakiintuneita, jolloin osaaminen testaukseen käytettävien työkalujen osalta voi olla hajanaista.

Testien tarve huomattiin varsin varhaisessa vaiheessa, kun sovellukseen lisättiin lyhyessä ajassa rutkasti uutta ja muokattiin olemassa olevaa sisältöä. Muutoksista aiheutui odottamattomia ohjelmavirheitä, joita ei ollut huomattu manuaalisen käyttöliittymätestauksen aikana ja joiden tuotantoon pääsy olisi ollut mahdollista estää, mikäli sovellusta olisi testattu tai kehitystyötä olisi tehty testivetoisesti.

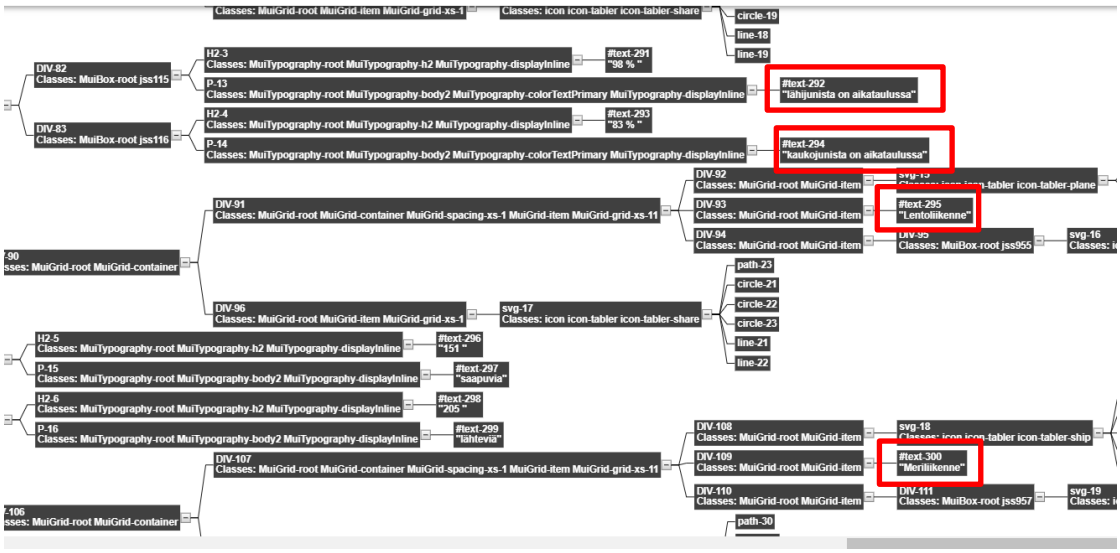
3.4 Testausautomaatiotyökalut

Sovelluksen koonti ja julkaisu tapahtuivat automatisoidussa ympäristössä, joka oli kytketty versionhallintaan. Kun versionhallintaan lisättiin uusi versio lähdekoodista, hoitui sovelluksen koonti automaattisesti, mutta julkaisun tekeminen edellytti kehittäjän toimia. Testausautomaatio haluttiin osaksi tätä automatisoitua prosessia.

Testaamisen työkaluiksi valikoituivat React-käyttöliittymäkirjaston suosittelemat React Testing Library ja Jest. Web-sovelluksen käyttöliittymätestaus ei välttämättä vaadi järjestelmätestausta, joka on usein turhan hidasta suorittaa etenkin automatisoidun koonnin yhteydessä. Valitut työkalut mahdollistavat yksikkö- ja integraatiotasoisen testaamisen myös käyttöliittymäkomponenteille.

React Testing Library on suosittu kirjasto komponenttien hahmontamiseen testausta varten. Kirjasto muodostaa testattavasta React-komponentista puumaisen dokumenttioliomallin (DOM-puu), aivan kuten verkkoselainkin tekee, ja tämän puun sisältöä voidaan tarkastella ja vertailla testeissä.

React Testing Library mahdollistaa myös tapahtumien simuloinnin fireEvent-funktioilla, mutta tämä ei täysin vastaa käyttäjän ja selaimen välistä interaktiota. Kyseisillä funktioilla onkin pääsääntöisesti tarkoitus testata, onko tapahtumakuuntelijana toimivaa funktiota kutsuttu, mutta varsinaiseen käyttäjäsimulointiin tämä on kehno keino. Laajempien käyttötapausten, joissa on useita vaiheita, simulointiin on suositeltavaa käyttää muita työkaluja. [Considerations for fireEvent | Testing Library, 2022.]



Kuva 10: Erittäin pieni osa Liikenteen Pulssi -näkymästä kuvattuna DOM-puuna

React Testing Libraryn testityökalujen avulla voidaan välttää runsaasti resursseja vaativa sovelluksen hahmontaminen verkkoselaimeen ja sen sijaan tarkastella DOM-puun solmujen sisältöjä.

Kuten kuvasta 10 voi huomata, on Liikennetilanteen kaltaisen verkkopalvelun yhdenkin alasivun DOM-puu kokoluokaltaan valtava, vaikka verkkosivulla näkyvä komponentti olisikin yksinkertainen [kuva 11]. Mikäli siitä halutaan tarkastella yhden ainoan solmun tekstisisältöä, on tarjolla olevia solmuvaihtoehtoja lukuisia.

Kuva 11: Kuvassa 10 merkittyjä DOM-puun solmut näkyvillä verkkoselaimessa

React Testing Libraryn suosittelema ratkaisu oikean solmun löytämiseen on asettaa tälle React-komponentiin data-testid-attribuutti, jonka avulla oikea solmu voidaan löytää. Verkkosivuilla käytetyn hyperlinkkejä sisältävän merkintäkielen (HTML) standardin versiossa 5 esitellyt ”data”-alkuiset HTML-elementtiin liitettävät attribuutit mahdollistavat HTML-standardissa määrittelemättömän, mutta sovelluksen lähdekoodissa määritellyn attribuutin käytön [Using data attributes, 2021].

React Testing Library mahdollistaa myös käyttäjän toimia muistuttavien toimenpiteiden suorittamisen, kuten esimerkiksi painikkeen painamisen, ja tämän toimenpiteen DOM-puuhun aiheuttamia muutoksia voidaan myös niin ikään testata.

Testit suoritetaan Jest-testausviitekehyksen avulla, joka on React-käyttöliittymäkirjaston suosittelema työkalu React-sovelluksen testaamiseen. Jest on alun perin Facebookin kehittämä ja nykyisin Metan ylläpitämä testaustyökalu. Jest on varsin yleisesti käytetty testaustyökalu, joka mahdollistaa testaamisen nopeasti ja turvallisesti suorittamalla testejä rinnakkaisina prosesseina. [Jest – Delightful JavaScript Testing, 2022.]

Testauksessa käytetty Jest-testausviitekehys suorittaa lähdekoodia JavaScriptinä, mutta koska kyseessä on TypeScriptillä koodattu React-sovellus, pitää ohjelmakoodi kääntää myös testausta varten JavaScriptiksi.

3.5 Testausautomaation toteuttaminen

Sovelluksen testiautomaation rakentaminen lähti Jest- sekä React Testing Library -testauskirjastojen asentamisesta ja näiden määrittelyistä. Jest-asetustiedostoon määritellään, kuinka testauskirjastoa halutaan käyttää, mikäli määritykset poikkeavat oletusarvoista. Vaikka Jest [2022] pyrkii olemaan käytettävissä minimaalisella konfiguroinnilla, piti asetustiedostoon tehdä joitakin määrittelyitä, ennen kuin testejä saatiin suoritettua.

Lähdekoodi vaati myös jossain määrin uudelleenkirjoittamista. Sovellukselle keskeisessä karttakomponentissa oli sen sisäistä logiikkaa, johon ei päässyt käsiksi DOM-puuta hahmontamalla ja sitä testaamalla. Toinen haaste karttakomponentin kanssa tuli siitä, kun kartan sisältämä komponentti hahmonnettiin testiä varten DOM-puuhun, pyrki tämä käynnistämään ulkoisia verkkokutsuja, mikä ei ollutkaan testiympäristössä mahdollista. Kun verkkokutsut toteutettiin testikirjaston mock-oliolla, oli seuraavan verkkokutsun virheet ratkaistavana. Tämä ketjutettujen ulkoisten verkkopyyntöjen virheiden vyyhti johti lopulta siihen, että karttakomponentin toiminnallisuutta, esimerkiksi kartan muokkaamiseen liittyviä funktioita, eriytettiin omaan apuluokkaansa. Karttakomponentin apuluokan funktiot kirjoitettiin staattisiksi, jotta niiden kutsuminen onnistuisi helposti myös testeissä.

3.5.1 Testikirjaston konfigurointi

Sovelluksen koodista löytyi sekä TypeScript- että JavaScript-ohjelmointikielellä kirjoitettua lähdekoodia, joista jälkimmäistä löytyi pääsääntöisesti sovelluksessa käytettyjen kirjastojen tiedostoista. TypeScript-ohjelmakoodin kääntäminen JavaScriptiksi vaati erillisen ts-jest-muuntimen käyttöönoton, mutta JavaScript-koodiin tätä samaa muunninta ei voinut käyttää.

Ratkaisu löytyi lisäämällä Jest-asetustiedostoon transform- sekä tranformIgnorePatterns-avainsanat.

```
  transformIgnorePatterns: [
    "/node_modules/(?!(ol|labelgun|mapbox-to-ol-
style|@arcgis|@esri|ol-mapbox-style)/).*/",
  ],
  transform: {
    "^.+\\.(ts|tsx|js|jsx)$": "ts-jest",
  },
```

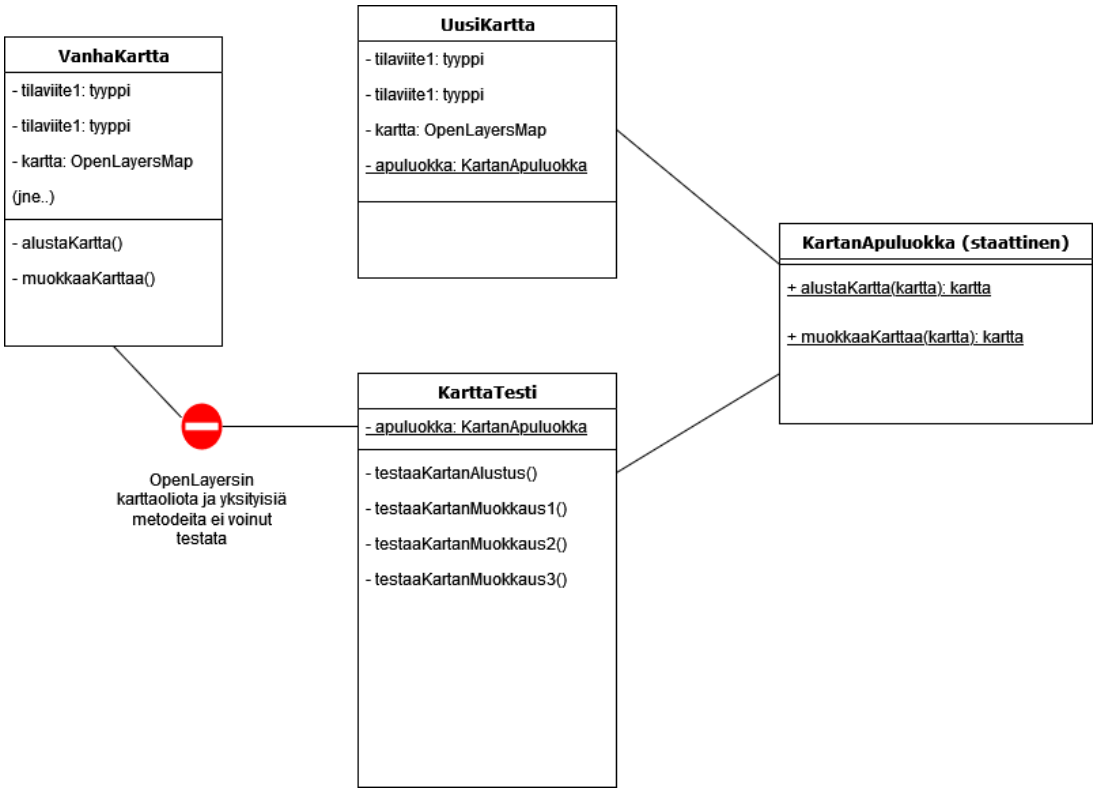
Esimerkkikoodi 1. Ote Jest-asennustiedoston määrityksistä

Transform- ja TransformIgnorePatterns-kohtien arvoksi määriteltiin säännöllinen lauseke (RegExp), johon etsitään vertaavuuksia projektin tiedostopuusta. TransformIgnorePatterns-kohdassa määritellyn mukaiset ulkoisten kirjastojen

lähdekoodit muunnettiin Babel-muuntimella ja muutoin Transform-kohdan määrittelyn mukaiset tiedostot taas ts-jest -muuntimella (esimerkkikoodi 1).

3.5.2 Testejä varten kirjoitetut apuluokat

Sovellukseen kirjoitettiin apuluokkia helpottamaan testien kirjoitusta.



Kuva 12: Hahmotelma karttakomponentin uusimisesta: Karttakomponentin logiikka eriytettiin omaan apuluokkaansa, sillä alkuperäinen karttakomponentti (VanhaKartta) ei ollut testattavissa.

Kaikki karttaluokan logiikka eriytettiin karttakomponentista, kuten kuvassa 12 on kuvattu. Karttakomponentin OpenLayersin karttaoliota ei ollut mahdollista hahmontaa testeissä sen tekemien useiden verkkokutsujen vuoksi, joten testien osalta päädyttiin ratkaisuun, että kaikki komponentin sisältämä logiikka, mikä oli eristettävissä komponentista, siirrettiin omaan apuluokkaansa [kuvassa 12 nimellä KartanApuluokka]. Hyödyntämällä apuluokkaa varsinaisen karttakomponentin ja testin välissä ikään kuin sovittimena saatiin kierrettyä

karttaolion verkkokutsujen aiheuttama testaamisen estävä ongelma ja näin ollen karttaan liittyvät operaatiot testatuksi.

```
static handleZoomToPoint(
  map: olMap | undefined,
  coords: Coordinate | undefined,
  wantedZoom?: number,
): olMap | undefined {
  if (map) {
    const view = map.getView();
    if (view) {
      const currentZoom = view.getZoom();
      if (currentZoom) {
        const zoom = currentZoom < 10 ? 10 : currentZoom;
        const maxZoom = map.getView().getMaxZoom();
        if (coords) {
          map.getView().animate({
            center: coords,
            duration: 500,
            zoom: zoom <= maxZoom ? zoom : undefined,
          });
        } else {
          map
            .getView()
            .animate({zoom: wantedZoom, duration: 500});
        }
      }
    }
  }
  return map;
}
```

Esimerkkikoodi 2. Kartan lähentämiseen ja loitontamiseen käytetty staattinen funktio apuluokasta

Esimerkiksi karttaan liittyvän apuluokan handleZoomToPoint-funktiolle, joka esitetään esimerkkikoodissa 2, annetaan parametrina testiluokassa luotu karttaolio sekä koordinaatit, joihin kartta halutaan keskittää, ja lisäksi valinnaisena parametrina voidaan antaa vielä haluttu lähennys- tai loitonnusarvo. Koordinaattipisteiden ja mahdollisen lähennys- tai loitonnusarvon perusteella muokataan karttaoliota. Lopulta se palautetaan takaisin testiluokkaan. Testissä funktion paluuarvosta voidaan tarkistaa, ovatko muutokset tapahtuneet oletetulla tavalla. Karttakomponentin apuluokan muissa metodeissa on vastaava toimintaperiaate.

```
<Provider store={store}>
  <Router history={history}>
    <I18nextProvider i18n={i18n}>
      <TestattavaKomponentti />
    </I18nextProvider>
  </Router>
</Provider>
```

Esimerkkikoodi 3. Ilman apuluokkaa jokainen testattava komponentti olisi pitänyt kuorruttaa sovelluksen vaatimilla tila-, reititys- ja lokalisointisolmuilla.

Koska sovelluksessa on käytetty Redux-tilanhallintaratkaisua sekä React Router-reitityskirjastoa, ei testattavaa komponenttia voi hahmontaa DOM-puuhun ilman niitä. Käytännössä tämä tarkoittaa, että yhden komponentin hahmontamiseksi jokaisessa testissä joudutaan testattavan komponentin lisäksi hahmontamaan tilanhallinnan ja reitityksen vaatimat komponentit, jotta testaaminen olisi ylipäätään mahdollista (esimerkkikoodi 3).

```
function render(
  ui,
  { preloadedState, store = realStore, ...renderOptions } = {},
) {
  function Wrapper({ children }) {
    return (
      <Provider store={realStore}>
        <Router history={history}>
          <I18nextProvider i18n={i18n}>{children}</I18nextProvider>
        </Router>
      </Provider>
    );
  }
  return rtlRender(ui, { wrapper: Wrapper, ...renderOptions });
}

export * from "@testing-library/react";
export { render };
```

Esimerkkikoodi 4. Ote testeissä hahmontamiseen käytetyn apuluokan lähdekoodista

Sovellukseen laadittiin testihahmontamisen apuluokka (esimerkkikoodi 4), jossa täydennetään React Testing Libraryn render-funktiota muokkaamalla sen paluuarvoksi alkuperäisen sisältönsä lisäksi myös tarvittavat tila-, reititys- ja lokalisointisolmut. Testihahmontamisen apuluokan muokattu render-funktio vähentää näin koodirivejä, sillä useissa testeissä tarvittava toiminnallisuus hoituu kutsulla tämän apuluokan render-funktioon, joka palauttaa myös

tarvittavat emokomponentit. Esimerkkikoodissa 5 on esitelty apuluokan käyttöä: Testikomponentin hahmontaminen onnistuu yhdellä rivillä, kun ilman apuluokkaa siihen olisi tarvinnut seitsemän riviä koodia. Apuluokkaa käyttämällä varmistettiin myös helpompi testien kirjoittaminen.

```
import { render } from "../../__testutil__/testUtils";

const component = render(<TestattavaKomponentti />);
```

Esimerkkikoodi 5. Testeissä hahmontamiseen käytetyn apuluokan render-funktion käyttö onnistuu napakasti.

Sovellukseen kirjoitettiin myös testihahmontamisen apuluokka, jossa täydennetään React Testing Libraryn render-funktiota muokkaamalla sen paluuarvoksi alkuperäisen sisältönsä lisäksi myös tarvittavat viitteet Redux-tilaolioon ja Router-reititysolioon. Tilaviite hoitui Redux Mock Store -kirjaston tarjoaman mock-olion avulla. Testihahmontamisen apuluokan muokattu render-funktio vähentää näin koodirivejä, sillä useissa testeissä tarvittava toiminnallisuus hoituu kutsulla tämän apuluokan render-funktioon, joka palauttaa myös tarvittavat emokomponentit.

Kolmannesta apuluokasta löytyy muutoin vastaavalla tavalla muokattu render-funktio, mutta ilman tilanhallinnan viitettä, sillä joidenkin komponenttien, esimerkiksi yksittäisen karttakohteen tietoja kertovien komponenttien kanssa on käytännöllisempää alustaa tila tarvittavine sisältöineen testitiedostossa.

3.5.3 Kirjoitetut testitapaukset

Testien kirjoittaminen jälkikäteen sovellukseen on paikoitellen hankalaa ja aikaa vievää. Tämän opinnäytetyön puitteissa ei ollut mahdollista kirjoittaa testejä koko sovelluksen laajuudelta, mutta testattavaksi valittiin sellaisia kohteita, joista arvioitiin olevan eniten hyötyä sovelluksen laadunparantamisen ja testausstrategian käyttöönoton kannalta.

Sovellukselle keskeisen karttakomponentin osalta testien tarve oli havaittu jo ennen opinnäytetyön aloittamista, sillä komponentti oli paisunut kooltaan melko

suureksi ja toiminnaltaan epäselväksi. Muilta osin testejä kirjoitettiin sellaisille komponenteille, joiden testaamisesta olisi eniten hyötyä tulevaisuuden kehitystehtävissä. Kirjoitetuilla testeillä pyrittiin myös löytämään mahdollisimman paljon sellaisia tilanteita, jotka vaativat testaustyökalujen asetusten muokkaamista tai mahdollisten lisäkirjastojen asentamisia. Nämä tapaukset huomioimalla kehitystiimi välttyisi aikaavieviltä konfiguraatiomuutoksilta tulevaisuudessa.

Kirjoitetut testit tarkastelevat pääsääntöisesti komponenttiin hahmonnettavan sisällön oikeellisuutta, mutta apuluokkien testit tarkastelevat funktioiden paluuarvoja. Käyttäjäinteraktioita muistuttavia tapahtumien simulointia ei testeissä käytetty.

Taulukossa 1 on listattu opinnäytetyön puitteissa kirjoitetut testit komponenteittain. Suurin määrä testejä löytyy kartan apuluokalle, jonka testejä on yhteensä 71 kappaletta. Kartan apuluokan korostuminen testimäärissä tarkasteltuna selittyy parilla seikalla paitsi että karttakomponentti on toiminnaltaan monipuolinen ja sisältää useita testattavia funktioita, on kutakin funktiota testattu useampia kertoja erilaisin parametrimäärittelyin. Esimerkkikoodin 2:n handleZoomToPoint-funktiota testataan useita kertoja: niin että kartan tai koordinaattien arvo on jätetty alustamatta sekä niin että haluttu zoom-parametrin arvo on 10 tai 12. Erilaisilla parametreilla tai parametrien yhdistelmillä saadaan varmistettua, että testeissä on huomioitu kaikki mahdolliset suorituspolut, mikä selittää sen, miksi 11 testatulle komponentille on yhteensä 129 testiä.

Taulukko 1: Opinnäytetyön puitteissa kirjoitettujen testien lukumäärä

Komponentin/apuluokan tyyppi	**Testitapauksia (kpl)**
Listanäkymien pääsivu	11
Listanäkymä: Lentokentät	10
Listanäkymä: Yhteenvetosivun liikennetiedotteet	8

Listanäkymässä esiintyvä solu: Liikennetiedotteet	6
Aikakäsittelyyn liittyvä apuluokka	7
Kartan käyttämä apuluokka	71
Ponnahdusikkuna: Oulun pysäköintitalot	3
Ponnahdusikkuna: Painorajoitukset	10
Lentoliikenteen yhteenveto	1
Liikenteen yhteenveto	1
Verkkosivun ylätunniste	1

Joissain komponenteissa data kulkee isäntäkomponentista lapsikomponenttiin funktionaalisen komponentin parametrina ja osassa komponentti pyytää sen keskitetystä tilasta. Esimerkiksi lentoliikenteen yhteenvetonäkymää tarjoava komponentti saa parametrina DOM-puuhun hahmonnettavat lentojen lukumäärät, joten testissä ei tarvita monimutkaista datan tai tilan alustusta. Esimerkkikoodissa 6 on käytetty render-funktion versiota, jossa tilan määrittely tapahtuu komponentin sijaan apuluokassa.

```
test("Arriving and departing renders right", () => {
    const component = render(
      <AirTrafficSynopsis arriving={"444"} departing={"12"} />,
    );
    expect(component.getByText("444")).toBeDefined();
    expect(component.getByText("saapuvia")).toBeDefined();
    expect(component.getByText("12")).toBeDefined();
    expect(component.getByText("lähteviä")).toBeDefined();
  });
```

Esimerkkikoodi 6. Lentojen yhteenvetonäkyvää testaava testitapaus

Esimerkkikoodissa 7 löytyy render-funktion tilatonta versiota käyttävä testi, jossa testidata alustetaan testiluokassa. Sieltä se tallennetaan keskitettyyn tilaan, josta se päätyy kaikkien DOM-puussa mallinnettavien komponenttien käyttöön.

```
test("Tampere announcement start and endtime is rendered", () => {
    const component = renderWithoutStore(
      <Provider store={store}>
        <TrafficAnnouncements trafficMode={"roadTraffic"} />
      </Provider>,
    );

    const expectedTimeStr = DateUtils.generateStartEndDateString(
      1620014400000,
      null,
      "klo",
    );
    const time = component.getByTestId("treStartEndTime");
    expect(time).toBeDefined();
    expect(time.textContent).toContain(expectedTimeStr);
  });
```

Esimerkkikoodi 7. Tieliikenteen tiedotteen ajankohdan hahmontamista testaava testitapaus

Testeissä päädyttiin käyttämään React Testing Libraryssä suositeltuja data-testid-attribuutteja (esimerkkikoodi 8), jotka määritellään testattavaan komponenttiin ja joilla varustetut solmut voidaan etsiä testissä getByTestId-funktiolla.

```
<div>
  <p data-testid=”esimerkki”> Teksti </p>
</div>
```

Esimerkkikoodi 8. Pseudokoodi, jossa käytetty data-testid-attribuuttia

Näin saadaan myös kierrettyä käyttöliittymäsovellusten testaukseen perinteisesti liittyvä haaste, joka muodostuu erilaisten päätelaitteiden ja näyttökokojen kanssa, kun testattava komponentti ja sieltä esittävä sisältö voi sijaita eri laitteilla eri kohdassa näyttöä, kuten esimerkiksi mobiililaitteen pystymallisella ja kannettavan tietokoneen vaakamallisella ruudulla.

3.5.4 Testien ajaminen putkessa

Projektin jatkuva integraatio ja koonti oli määritelty toimimaan versionhallintaympäristön CI/CD-putkessa. Projektiin jo aiemmin tehtyjen

määrittelyiden mukaan sovelluksen koonti tapahtuu aina, kun versionhallintaan viedään uutta lähdekoodia ja testien suorittaminen lisättiin osaksi tätä vaihetta.

Testien suorittaminen versionhallintaympäristössä aiheutti hankaluuksia sovelluksessa näkyvien aikaa kuvaavien merkkijonojen kanssa, sillä komentoja suorittava kone sijaitsee eri aikavyöhykkeellä kuin kehitystiimi. Sovelluksessa käsiteltävät muuttujat, jotka kuvaavat aikaa, ovat tyypiltään numeroita, ja ne muutetaan käyttöliittymää varten merkkijonoiksi kuten esimerkiksi 17.03.2022 klo 18:00 – 18.03.2022 klo 16:00. Mikäli testissä käytetään vertailuun valmista aikaa kuvaavaa merkkijonoa, kuten yllä oleva maaliskuinen aika, testi epäonnistuu, sillä komponentissa näkyvä alkuperäisestä numerosta muokattava aikaleimaa kuvaava merkkijono muodostetaan testin suorittamisen aikana sen aikavyöhykkeen perusteella, jossa kone sijaitsee. Testeissä tuleekin muodostaa myös vertailuarvo suoritusaikana.

3.6 Testausstrategia

Testausstrategiassa on tarkoitus määrittää sovelluskehityksen avuksi suunnitelma siitä, miten ja millaisilla pelisäännöillä sovelluksen testaus toteutetaan.

Testausstrategia laadittiin osaksi palvelun dokumentaatiota. Testausstrategian määrittelyn mukaan testien kirjoittaminen ja testauksesta huolehtiminen ovat osa kehitystehtävää. Mikäli tämä on jäänyt huomioimatta, ei kehitystehtävää voi merkitä valmiiksi.

Jos kehitystehtävässä lisätään sovellukseen uutta toiminnallisuutta, tulee tälle toiminallisuudelle kirjoittaa mahdollisimman kattavat testit. Mikäli tehtävässä taas muokataan vanhaa toiminnallisuutta, varmistetaan olemassa olevien testien toiminta uuden toteutuksen kanssa. Jos testejä ei vielä löydy, ne kirjoitetaan.

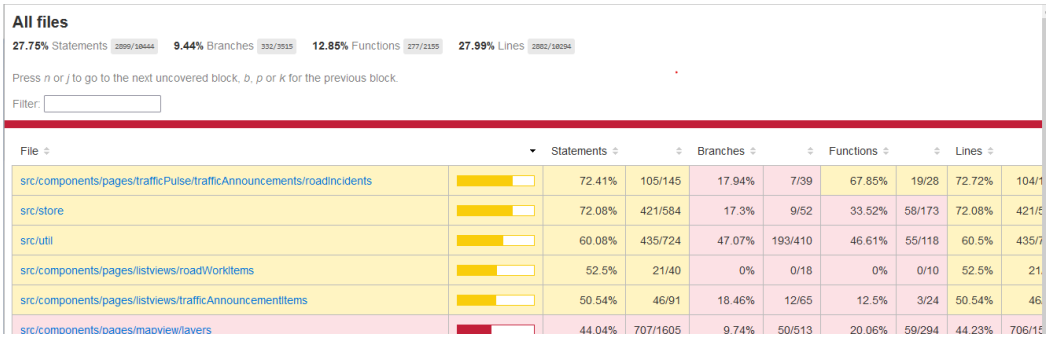
Testauksen tavoitteena on vähentää ohjelmavirheitä ja sitä myöten parantaa kokemusta laadusta. Osana strategiaa testaus on määritelty automaattiseksi vaiheeksi CI/CD-putkea, sillä testit suoritetaan jokaisella kerralla, kun versionhallintaan lisätään uutta koodia. Jos testejä ei läpäistä, ei sovelluksen koonti ja mahdollinen julkaisu onnistu. Tällä pyritään varmistamaan se, ettei testeillä löydettävää virheellistä ohjelmakoodia päädy julkaistavaksi.

Palvelun dokumentaatiosta löytyy myös käytännön vinkkejä testien kirjoittamisen tueksi ja esimerkiksi data-testid-attribuuttien käyttöä suositellaan vahvasti.

Testausstrategiaan kirjattiin tavoite 80 % testikattavuudesta, mutta koska kaikille komponenteille ei ole testejä kirjoitettu vielä, tulee tämä aluksi tulkita testattavan komponentin osalta. Sovelluksesta löytyy myös sellaisia osioita, joita testikattavuusvaatimus ei koske, kuten esimerkiksi avain-arvopareja tarjoilevat tiedostot, joten kirjattua tavoitetta kattavuusprosentista ei voi tulkita absoluuttisena.

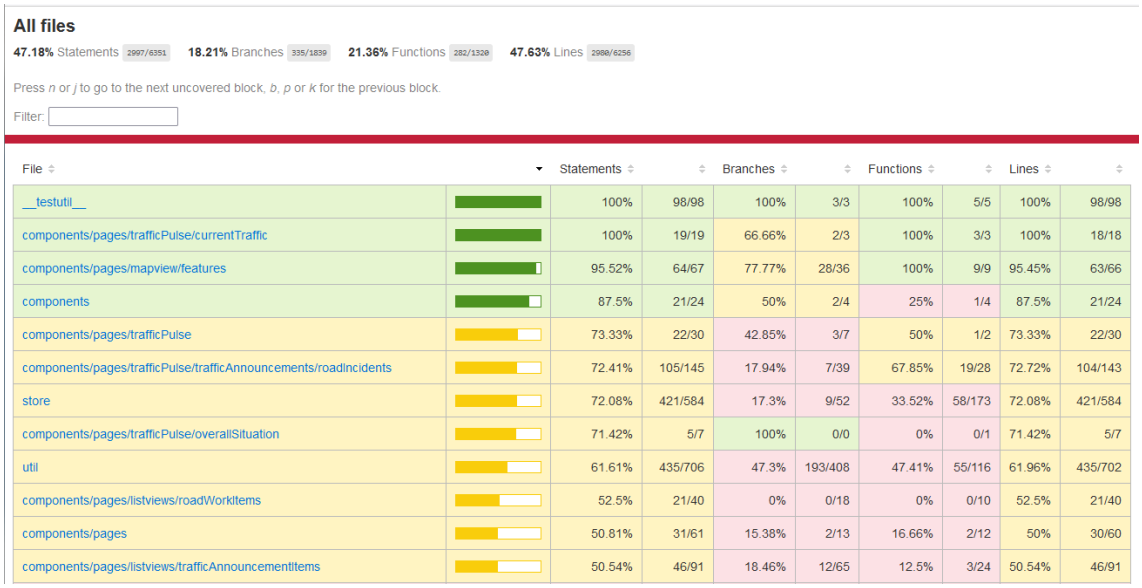
4 Tulokset

Testaamisen laadun arvioimiseen on lukuisia eri mittareita. Testaamisen arvioiminen kattavuusanalyysillä tarkasteltuna osoitti kattavuuden parantuneen testien määrän kasvettua.



Kuva 13: Ote testikattavuusraportista koko sovelluksen lähdekoodia vasten tarkasteltuna

Jest-testausviitekehys tarjoaa myös työkalun koodikattavuuden tarkistamiseen, joka tuottaa selaimessa nähtävän html-tiedoston. Kun kattavuusanalyysi suoritettiin koko sovelluksen laajuudella [kuva 13], päästiin koodiriveillä mitattuna miltei 28 prosentin kattavuuteen, mutta toisaalta päätöskattavuudella jäätiin alle kymmeneen prosenttiin. Kirjoitettuja testejä vasten tarkasteltuna [kuva 14] päästiin joidenkin komponenttien osalta vihreälle alueelle yli 80 prosenttiin.



All files

47.18% Statements 2997/6351 **18.21%** Branches 335/1839 **21.36%** Functions 282/1320 **47.63%** Lines 2980/6256

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

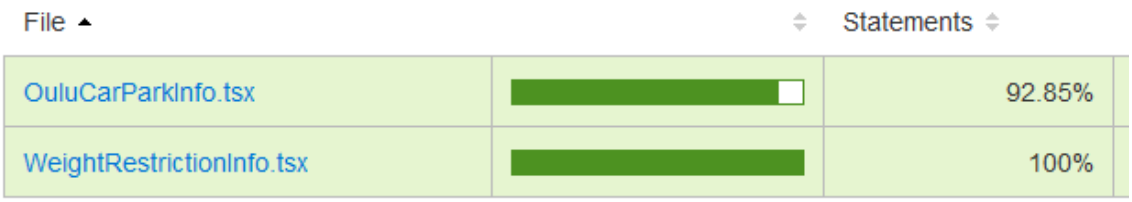
File	Statements		Branches		Functions		Lines	
__testutil__	100%	98/98	100%	3/3	100%	5/5	100%	98/98
components/pages/trafficPulse/currentTraffic	100%	19/19	66.66%	2/3	100%	3/3	100%	18/18
components/pages/mapview/features	95.52%	64/67	77.77%	28/36	100%	9/9	95.45%	63/66
components	87.5%	21/24	50%	2/4	25%	1/4	87.5%	21/24
components/pages/trafficPulse	73.33%	22/30	42.85%	3/7	50%	1/2	73.33%	22/30
components/pages/trafficPulse/trafficAnnouncements/roadIncidents	72.41%	105/145	17.94%	7/39	67.85%	19/28	72.72%	104/143
store	72.08%	421/584	17.3%	9/52	33.52%	58/173	72.08%	421/584
components/pages/trafficPulse/overallSituation	71.42%	5/7	100%	0/0	0%	0/1	71.42%	5/7
util	61.61%	435/706	47.3%	193/408	47.41%	55/116	61.96%	435/702
components/pages/listviews/roadWorkItems	52.5%	21/40	0%	0/18	0%	0/10	52.5%	21/40
components/pages	50.81%	31/61	15.38%	2/13	16.66%	2/12	50%	30/60
components/pages/listviews/trafficAnnouncementItems	50.54%	46/91	18.46%	12/65	12.5%	3/24	50.54%	46/91

Kuva 14: Kattavuusanalyysi kirjoitettujen testien suhteen

Kuvien 13 ja 14 listoilla näkyy myös sellaisia hakemistoja, kuten src/store, joiden alta löytyy paljon koodirivejä, mutta joiden testaaminen ei ole välttämättä järkevää ainakaan yksikkötesteinä. Redux-tilan testausta on suositeltu tehtävän integraatiotesteinä niin, että testissä alustetaan todellinen Redux-tila, johon tallennetaan mock-dataa ja tilan käsittelyä testataan käyttöliittymäkomponenttien kautta tarkasteltuna [Writing tests, 2022].

Opinnäytetyön puitteissa kirjoitettujen kartan ponnahdusikkunakomponenttien osalta testikattavuus oli erinomainen yli 90 prosentissa [kuva 15]. Kyseisten komponenttien testaaminen on kohtuullisen yksinkertaista, sillä niissä on pitkälti

staattista ja tekstimuotoista sisältöä.



Kuva 15: Ote testikattavuusraportista ponnahdusikkunakomponenttien osalta

Testaustyökaluista osoittautui olevan hyötyä myös sellaisissa tilanteissa, kun lähdekoodiin piti tehdä muutoksia sellaisille komponenteille, joiden perustana olevaa dataa ei ollut juuri sillä hetkellä saatavilla paikkatietojärjestelmästä. Esimerkiksi erikoiskuljetusten kartan ponnahdusikkuna- ja listanäkymäkomponentteihin tehtiin muutoksia hieman TDD-ideologiaa mukaillen: näille kirjoitettiin ensin testi, johon lisättiin manuaalisesti testidata ja haluttu lopputulos ja sen jälkeen tehtiin muutokset komponenttiin. Tällä keinolla on mahdollisuus nopeuttaa kehitystehtäviä pienten muutosten yhteydessä, mikäli kyseistä datalajia ei ole saatavilla.

4.1 Vaikutus kehitysprosessiin

Opinnäytetyönä tehty testausstrategia ja testauskirjastojen käyttöönotto mahdollistavat testaamisesta huolehtimisen myös tulevaisuudessa. Testausstrategian käyttöönoton myötä on odotettavissa testattavien komponenttien määrän kasvaminen, joka omalta osaltaan varmistaa parempaa laatua jatkossa. Osana työtä kirjoitetut testit, etenkin karttakomponentin osalta, tukevat mainiosti aiemmin pääasiallisena testausmetodina käytettyä järjestelmällistä manuaalitestausta.

Testit tarjoavat myös helpon keinon valmistautua siihen, jos lähdedatassa olisi poikkeavaa tai erikoista sisältöä. Testeissä voidaan tuottaa nopeasti ja helposti tällaista lähdedataa. Näin varmistetaan se, että datan muotoilussa käyttöliittymää varten tulee huomioitua myös sisällön erikoisemmat

reunatapaukset. Manuaalisessa testauksessa on jouduttu nojaamaan sellaiseen dataan, jota sillä hetkellä on ollut tarjolla, joten esiin ei ole välttämättä noussut ongelmia, mikäli data on ollut säännönmukaista ja virheetöntä.

Testausautomaatiolla suoritetut testit myös parantavat mahdollisuutta löytää virheet varhaisemmassa vaiheessa kuin ennen. Aiemmin järjestelmällistä manuaalitestausta tehtiin vasta siinä vaiheessa, kun julkaisua suunniteltiin. Nyt virheellinen testitulos käy ilmi jo siinä vaiheessa, kun kehittäjä lisää lähdekoodia versionhallintaan, mikä varsin todennäköisesti aikaistaa mahdollisen virheen löytämistä. Sitä myöten virheen korjaaminen on tehokkaampaa ja edullisempaa, kun lähdekoodiin tehdyt muutokset ovat vielä kehittäjän tuoreessa muistissa.

4.2 Työkalujen toimivuus

Jestin toimintavarmuus on erittäin hyvä, sillä se on varsin keveä toteutus testaamiseen. Huolimatta keveydestään joidenkin testijoukkojen suorittaminen vei useita sekunteja aikaa, ja automaattisena tämä aika keskimäärin tuplaantui. Valittu strategia suorittaa testit eristyksessä sivuuttaa haasteet ulkoisten tekijöiden, kuten epävakaiden verkkoyhteyksien, kanssa.

Virheitä CI/CD-putkessa ajetuissa testeissä ilmeni lähinnä aikakäsittelyn kanssa, sillä automaatiotestausta suorittava kone sijaitsee maantieteellisesti eri aikavyöhykkeellä. Kesäaikaan siirtyminen aiheutti myös vastaavantyyppisiä ongelmia joissain testeissä.

Testauksessa käytetyn kattavuusanalyysi ei anna välttämättä todellisuutta vastaavaa kuvaa siitä, miten laajaa testaus on, mikäli testit eivät ole laadultaan korkealuokkaisia. Kattavuusanalyysin tulokset saa näyttämään todellista paremmalta hahmontamalla testissä jonkin komponentin, mutta mikäli komponentin tekstisisältöjen vertailuja DOM-puusta löydetyn solmun ja odotetun sisällön välillä ei tehdä, ei testi todellisuudessa testaa mitään, vaikka se menisikin läpi ja kattavuusanalyysin mukaan kaikki rivit käytäisiin läpi. Tästä

syystä kattavuusanalyysi ei ole absoluuttisen hyvä testaamisen laadullinen mittari.

4.3 Haasteet ja rajoitteet

Uusien testauskirjastojen käyttöönotto ja testausstrategian omaksuminen koko kehitystiimin laajuudelta vie oman aikansa. Tämän helpottamiseksi testausstrategia esimerkkitesteineen käytiin läpi yhdessä tiimin kanssa. Koska kirjoitettujen testien määrä suhteessa sovelluksen laajuuteen on vielä pieni, voi ohjelmavirheitä päästä läpi pelkästään siitä syystä, ettei kaikkia komponentteja ole vielä testattu. Tämän vuoksi järjestelmällisestä manuaalitestauksesta ei voi vielä luopua.

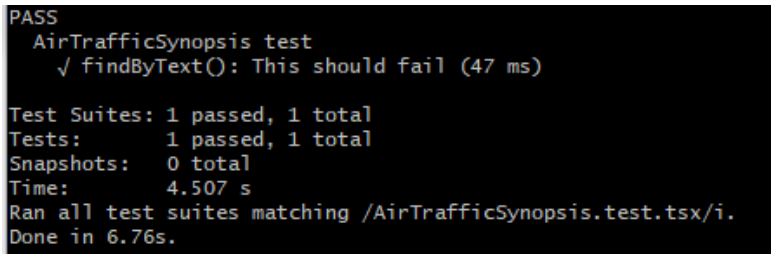
Testeissä jonkin verran haastetta aiheutti React Testing Library -testauskirjaston findByText-funtio. Kun etsittävä solmu DOM-puusta haetaan siihen hahmonnettavalla merkkijonolla kuten esimerkkikoodissa 9, ei sen tekstisisällön tarkistamiselle vaikuttaisi olevan enää tarvetta.

```
test("findByText(): This should fail", () => {
  const component = render(
    <AirTrafficSynopsis arriving={"444"} departing={"12"} />,
  );

  const foo = component.findByText("bar");
  expect(foo).toBeDefined();
});
```

Esimerkkikoodi 9. Testitapaus, jossa etsitään DOM-puusta solmua, jonka sisältönä olisi merkkijono ”bar”

Esimerkkikoodissa 9 siis etsitään DOM-puusta solmua, joka sisältäisi tekstisisällön ”bar”, jota ei todellisuudessa komponentista löydy. Huomionarvoista on kuitenkin se, että React Testing Libraryn findBy-alkuiset funktiot ovat asynkronisia, eli ne palauttavatkin toivotun solmun sijaan Promise-olion.



Kuva 16: Virheellisesti positiivinen testitulos liittyen esimerkkikoodiin 9.

Testaamisen alkuvaiheessa tämä aiheutti ongelmia, sillä testissä käytetty vertailu, toBeDefined, tarkisti vain, onko esimerkkikoodi 9:n foo-niminen komponentti ylipäätään määritelty ja koska muuttujaan oli tallennettu Promise-objekti, testi läpäistiin [kuva 16].

4.4 Omat havainnot

Testiautomaation rakentaminen edellytti paikoin melko syvällistä perehtymistä aiheeseen. Esimerkiksi karttakomponentin testaamiseen liittyvien ongelmien ratkaisu vaati lukuisia yrityksiä itse testien ja testiympäristön konfiguraation osalta. Valittu ratkaisu, jossa komponentin logiikka eriytettiin omaksi apuluokakseen, vaikuttaisi olevan tähän asti kokeilluista ratkaisuista kuitenkin käytännöllisin. Aiemmin monoliittisen karttakomponentin toimintaa oli ylipäätään hankala ymmärtää ja mahdollisten ohjelmavirheiden korjaaminen oli ongelmallista. Logiikan ja komponentin muun toiminnallisuuden ollessa erillään kokonaisuus on helpompi hahmottaa.

Testihahmontamisen apuluokkiin laadittu render-funktion täydennetty versio nopeutti testien kirjoittamista, kun uuden komponentin testiluokassa säästyi React Router- ja joissain komponenteissa Redux-solmujen määrittelyltä. Apuluokkamäärittelyt Redux-tilan kanssa ja ilman sitä palvelivat hyvin erilaisten komponenttien testausta.

React Testing Library vaikuttaa nopealla tutustumisella tarjoavan useita samankaltaisia funktioita vastaavuuksien etsimiseen, mutta joilla on kuitenkin

lähemmällä tarkastelulla hyvin erilaiset toimintaperiaatteet, kuten aiemmin mainituilla findBy-funktioilla. Tässä projektissa toimivien keinojen löydyttyä tuli testien kirjoittamisesta kuitenkin rutiininomaista.

5 Yhteenveto

Opinnäytetyön tavoitteena oli määritellä testausstrategia ja toteuttaa testausautomaatio osaksi projektia, jotta sovelluksen laatua voidaan parantaa.

Valitut teknologiat, React Testing Library ja Jest, soveltuivat hyvin käyttötarkoitukseensa sekä kehittämistyössä että automatisoituna osaksi versionhallinnan CI/CD-toimintoja. Esimerkinomaisesti kirjoitetut testit toimivat jatkossa tukena kehitystiimille tarjoten toimivia esimerkkejä erilaisiin tilanteisiin. Komponenttien hahmontamiseen liittyvissä testeissä käytetty data-testid-attribuutin käyttö osoittautui parhaaksi tavaksi löytää testattavasta komponentista oikea solmu sisällön vertailua varten. Keskittyminen yksikkö- ja integraatiotesteihin luo hyvän pohjan testaamisen laajentamiselle tulevaisuudessa.

Testausstrategiassa määritellyt pelisäännöt testaukselle ovat tavoitteissaan selkeät, joskin joissain tapauksissa määritelty käytäntö kirjoittaa puuttuvat testit kehitystiketin yhteydessä voivat hidastaa kehitystyötä. Käytäntö määriteltiin kuitenkin pyrkimyksenä kasvattaa sovelluksen testikattavuutta ilman erillisiä, pelkästään testaukseen liittyviä tehtäviä.

Jatkokehitysmahdollisuuksia testaamisen saralla riittää.

Nykyisellä toteutuksella opinnäytetyön puitteissa kirjoitettujenkin testien suorittaminen osana automaatiota vie kymmeniä sekunteja. Optimoimalla testikonfiguraatiota, itse testejä tai koontivaiheen testikomennon suoritusta rinnakkaisina prosesseina tätä saataisiin nopeutettua, mikä voi nopeuttaa sovelluskehittäjien työtä.

Käyttöliittymäkomponenttien testaamisen yksi haaste on testien ylläpito käyttöliittymän muuttuessa. Sovellukseen voidaan lisätä uusia komponentteja, muokata tai poistaa olemassa olevia taikka siihen voidaan tehdä rakenteellisia muutoksia. Näissä tilanteissa voisi käyttää apuna Jestin tukemaa snapshot-testausta, jonka ideana on, että testin yhteydessä tallennetaan tilannevedos komponentin palauttamasta sisällöstä ja tämä tallennetaan versionhallintaan lähdekoodin mukana. Testeissä verrataan kunkin suorituskerran tilannevedosta versionhallinnasta löytyvään, aiempaan tilannevedokseen. Mikäli ne eroavat toisistaan, testi epäonnistuu. Snapshot-testauksella voidaan varmistaa, ettei tahattomia käyttöliittymämuutoksia pääse testeistä läpi.

Testikokonaisuutta voisi laajentaa järjestelmätesteillä Cypress-kirjaston avulla. Selainympäristössä tapahtuvassa testauksessa käyttäjien toimien imitointi olisi helpompi toteuttaa esimerkiksi karttaan liittyvien operaatioiden osalta. Tämä laajentaisi testiympäristöä rutkasti nykyisestä, varsin eristetystä versiosta, sillä Cypress-kirjaston avulla testit voitaisiin suorittaa testikäyttöliittymää vasten, jolloin kokonaisuudessa olisivat mukana myös ulkoiset yhteydet paikkatietojärjestelmän tarjoamaan dataan. Käytännössä järjestelmätestauksen toteuttaminen vaatisi itsenäisen suoritusympäristön irrallaan nykyisestä CI/CD-putkesta esimerkiksi Docker-kontissa sekä muutoksia käytettyyn paikkatietojärjestelmään, sillä testeissä tarvittaisiin staattista dataa tai erillinen, järjestelmätestausta varten luotu tietovarasto.

Kehitysprosessin vieminen testivetoisempaan suuntaan olisi mielenkiintoista, kun kyseessä on jo tuotannossa oleva projekti. Käytännössä tämä voisi tarkoittaa esimerkiksi sitä, että uuden datalähteen lisääminen sovellukseen tehtäisiin miltei päinvastaisessa järjestyksessä kuin aiemmin: ensimmäiseksi kirjoitettaisiin esimerkiksi kartan ponnahdusikkunan testit ja toteutus, ja pala palalta integroitaisiin uusi sisältö aiempaan kokonaisuuteen.

Lähteet

3.1. Ohjelmistojen testausta — Ohjelmoinnin peruskurssi Y2, kurssimateriaali. 2016. Verkkoaineisto. Aalto-yliopisto. <http://www.cse.tkk.fi/fi/opinnot/CSE-A1121/2015/softwaretesting/testing.html>. Luettu 18.4.2022.

Alkio, Jyrki. 2010. Elisa ja IBM sotajalalla. Verkkoaineisto. Talouselämä. <https://www.talouselama.fi/uutiset/elisa-ja-ibm-sotajalalla/34cda1f9-fdfb-3c59-87a5-1aa2a87be16d>. Luettu 18.3.2022

Axelrod, Arnon. 2018. Complete Guide to Test Automation. E-kirja. APress.

Bilski Jacek; Mandry, Torsten. 2020. Tests Granularity. Verkkoaineisto. InnoQ. <https://www.innoq.com/en/blog/tests-granularity/>. Luettu 28.4.2022.

Bose, Shreya. 2021. Benefits of Automation Testing. Verkkoaineisto. BrowserStack. <https://www.browserstack.com/guide/benefits-of-automation-testing>. Luettu 8.4.2022.

Cimperman, Rob. 2006. UAT Defined: A Guide to Practical User Acceptance Testing. E-kirja. Addison-Wesley Professional.

The Code Gang. 2017. Flaky Tests – A War that Never Ends. Verkkoaineisto. Hackernoon. <https://hackernoon.com/flaky-tests-a-war-that-never-ends-9aa32fdef359>. Luettu 1.4.2022.

Considerations for fireEvent | Testing Library. 2022. Verkkoaineisto. Testing Library. <https://testing-library.com/docs/guide-events/>. Luettu 4.5.2022.

Dijkstra, Bas. 2014. The test automation pyramid. Verkkoaineisto. OnTestAutomation. <https://www.ontestautomation.com/the-test-automation-pyramid/>. Luettu 8.4.2022.

Dijkstra, Edsger W. 1970. Notes on Structured Programming. Verkkoaineisto. <https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF>. Luettu 18.3.2022. Technological University Eindhoven The Netherlands, Department of Mathematics.

Elfriede, Dustin. 2002. Effective Software Testing: 50 Specific Ways to Improve Your Testing. E-kirja. Addison-Wesley Professional.

End to end -testaus. 2022. Verkkoaineisto. Full Stack Open 2022. <https://fullstackopen.com/osa5/end_to_end_testaus>. Luettu 1.4.2022.

EU:n yleinen tietosuoja-asetus. 2016. 2016/679.

Fintraffic - Turvallista ja sujuvaa liikennettä. 2021. Verkkoaineisto. Fintraffic. <https://www.fintraffic.fi/fi/fintraffic-0>. Luettu 23.2.2022.

Fintrafficin tarkoitus: yhteinen suuntamme | Fintraffic. 2021. Verkkoaineisto. Fintraffic. <https://www.fintraffic.fi/fi/fintraffic/fintrafficin-tarkoitus-yhteinen-suuntamme>. Luettu 6.4.2022.

How Much Does Software Testing Cost? 9 Proven Ways to Optimize it. 2021. Verkkoaineisto. Simform. <https://www.simform.com/blog/software-testing-cost/>. Luettu 18.3.2022.

Jest – Delightful JavaScript Testing. 2022. Verkkoaineisto. Jest. <https://jestjs.io/>. Luettu 08.04.2022.

Jung, June. 2021. The path to production: how and where to segregate test environments. Verkkoaineisto. Circleci Blog. < https://circleci.com/blog/path-to-production-how-and-where-to-segregate-test-environments/#c-consent-modal https://circleci.com/blog/path-to-production-how-and-where-to-segregate-test-environments/#c-consent-modal >. Luettu 28.4.2022.

Koskela, Lasse. 2007. Test Driven: Practical TDD and Acceptance TDD for Java Developers. E-kirja. Manning Publications.

McCurdy, Nick. 2021. React Testing Library. Verkkoaineisto. Testing Library. <https://testing-library.com/docs/react-testing-library/intro/>. Luettu 1.4.2022.

Patton, Ron. 2005. Software Testing, Second Edition. E-kirja. Sams.

Perry, William; Rice, Randall. 2013. Surviving the Top Ten Challenges of Software Testing: A People-Oriented Approach. E-kirja. Pearson Education Inc.

Rani, Priya. 2020. Software Testing Challenges – 6 Modern QA Challenges. Verkkoaineisto. QA.world. <https://qa.world/software-testing-challenges/ >. Luettu 8.4.2022.

Software Testing Accounts to What Percent of Development Cost?. 2021. Verkkoaineisto. Intersog. <https://intersog.com/blog/software-testing-percent-of-software-development-costs/>. Luettu 18.3.2022.

Strategia 2022-2026 | Fintraffic. 2021. Verkkoaineisto. Fintraffic. <https://www.fintraffic.fi/fi/fintraffic/strategia-2022-2026>. Luettu 23.2.2022.

Testing overview. 2019. Verkkoaineisto. React.

<https://reactjs.org/docs/testing.html>. Luettu 28.4.2022.

Using data attributes. 2021. Verkkoaineisto. MDN. <https://developer.mozilla.org/en-US/docs/Learn/HTML/Howto/Use_data_attributes >. Luettu 08.04.2022.

Vocke, Ham. 2018. The Practical Test Pyramid. Verkkoaineisto. martinFowler.com. <https://martinfowler.com/articles/practical-test-pyramid.html>. Luettu 28.4.2022.

Vuosikatsaus 2021. 2022. Verkkoaineisto. Fintraffic. <https://www.fintraffic.fi/sites/default/files/2022-03/Fintraffic_Vuosikatsaus_2021.pdf>. Luettu 6.4.2022.

Wacker, Mike. 2015. Just Say No to More End-to-End Tests. Verkkoaineisto. Google Testing Blog. <https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html>. Luettu 8.4.2022.

Writing tests. 2022. Verkkoaineisto. Redux. <https://redux.js.org/usage/writing-tests >. Luettu 17.3.2022.

Zulqadar, Amna. 2019. SDLC Waterfall Model: The 6 phases you need to know about. Verkkoaineisto. Rezaid. <https://rezaid.co.uk/sdlc-waterfall-model/>. Luettu 1.4.2022.

Özturk, Miktad. 2020. Software Testing Process and Levels of Testing. Medium. <https://medium.com/swlh/software-testing-process-and-levels-of-testing-4274904ce655 >. Luettu 1.4.2022.