jamk.fi

Web-sovelluksen kehittäminen testivetoisesti

Jarmo Syvälahti

Opinnäytetyö
Marraskuu 2020
Liiketalouden ala
Tradenomi (AMK), tietojenkäsittelyn tutkinto-ohjelma

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Syvälahti, Jarmo	Opinnäytetyö, AMK	Marraskuu 2020
	Sivumäärä 49	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Web-sovelluksen kehittäminen testivetoisesti

Tutkinto-ohjelma
Tietojenkäsittelyn tutkinto-ohjelma

Työn ohjaaja(t)
Kiviaho, Niko

Toimeksiantaja(t)

Tiivistelmä

Opinnäytetyön tavoitteena oli tutkia, kuinka testivetoisen kehityksen menetelmä soveltuu web-sovelluksen kehittämiseen. Web-sovellukset ovat nykyajan verkottuneessa maailmassa yhä suuremmassa roolissa. Testivetoisen kehityksen menetelmää on pidetty eräänä keinona lievittää ohjelmistojen monimutkaisuudesta johtuvia laatuongelmia.

Tutkimuksessa kehitettiin testivetoisesti esimerkkisovellus moderneilla web-teknologioilla. Se koostui verkkoselaimessa toimivasta asiakaspuolen sovelluksesta sekä Node.js-ympäristössä toimivasta palvelinpuolen sovelluksesta, joka tarjosi selainsovellukselle tietokantapalvelun RESTful-rajapinnan kautta. Selainsovelluksen testauksessa hyödynnettiin yksikkö- ja integraatiotestejä, palvelinpuolen sovellus testattiin yksinomaan integraatiotesteillä.

Sovelluksen toteutus onnistui suunniteltujen vaatimusten mukaisesti testivetoisen kehityksen menetelmää noudattaen. Testit kattoivat suurimman osan ohjelmakoodista, mutta kattavuus ei ollut aukotonta. Menetelmän hyödyntäminen hidasti kehitystyötä, mutta kehitystyön edetessä pidemmälle vähensi käsin testaamisen tarvetta, helpotti muutosten tekemistä ja vähensi yllättävien virheiden ilmaantumista.

Menetelmän näennäisestä yksinkertaisuudesta huolimatta sen omaksuminen koettiin haastavaksi. Testivetoisen kehityksen menetelmä todettiin kuitenkin soveltuvaksi web-sovelluksen kehittämiseen, mutta harkitessa sen noudattamista ohjelmistoprojekteissa tulee tarkkaan punnita sen tuomat hyödyt ja vaatimat panostukset.

Avainsanat (asiasanat)
Testivetoinen kehitys, web-sovelluskehitys, ohjelmistotestaus, Cypress, Express, Jest, React.js, Redux, Mongoose, Node.js, SuperTest, Testing Library, TypeScript

Muut tiedot (Salassa pidettävät liitteet)

jamk.fi

Description

Author(s)	Type of publication	Date
Syvälahti, Jarmo	Bachelor's thesis	November 2020
		Language of publication: Finnish
	Number of pages 49	Permission for web publication: x

Title of publication

Test-driven development of web application

Degree programme

Business Information Technology

Supervisor(s)

Kiviaho, Niko

Assigned by

Abstract

The objective of the thesis was to research how the method of test-driven development (TDD) can be applied to developing a web application. As the world gets more networked the role of web applications becomes crucial in our everyday lives. TDD has been suggested as one part of a solution to the quality issues that plague many of the software projects.

As a product of the research, a demo application was developed with test-driven method and modern web technologies. It consisted of a client-side application running in web browser and a server-side application running in Node.js environment. The purpose of the server-side app was to provide database access for the client-side app through RESTful-API. Testing of the client-side app included unit and integration tests, but the server-side app was tested solely with integration tests.

The application was implemented successfully according to specification adhering to TDD principles. The tests covered most of the application code, but some uncovered lines were observed. Applying the method of TDD increased the development time but it brought along some benefits as well later when the codebase was maturing. There was less of a need to manually test things, making changes to the codebase were easier and the occurrence of unexpected bugs was rare.

Despite the seeming simplicity of TDD the learning curve proved steeper than expected. All in all, the suitability of TDD for developing web applications was found sufficient. But when considering applying it for software projects the advantages and costs it brings need to be evaluated case-by-case.

Keywords/tags (subjects)

Test-driven development, web application development, software testing, Cypress, Express, Jest, React.js, Redux, Mongoose, Node.js, SuperTest, Testing Library, TypeScript

Miscellaneous (Confidential information)

Sisältö

Sanasto **4**

1 Johdanto **6**

2 Tutkimusasetelma **7**

2.1 Työn tavoitteet ja aikaisempi tutkimus aiheesta 7

2.2 Tutkimusmenetelmät 7

2.3 Rajaukset 8

2.4 Tutkimuskysymykset 8

3 Testaus osana ohjelmistotuotantoa **9**

3.1 Testaus ohjelmistotuotannon elinkaarimalleissa 9

3.1.1 Vesiputousmalli 9

3.1.2 Ketterät menetelmät 11

3.2 Testausautomaatio 12

3.3 Testaustasot 14

3.3.1 Yksikkötestaus 14

3.3.2 Integraatiotestaus 15

3.3.3 Järjestelmätestaus 15

3.3.4 Hyväksymistestaus 16

3.3.5 Staattinen analyysi 16

3.4 Testaustasojen välinen tasapaino 17

3.4.1 Testauspyramidi 17

3.4.2 Testauspokaali 18

3.5 Web-sovelluksen testauksen erityispiirteitä 19

4 Testivetoinen kehitys **20**

4.1 Testivetoisen kehityksen periaatteet 20

4.2 Testivetoisen kehityksen hyötyjä 22

4.3 Testivetoisen kehityksen rajoitteita ja kritiikkiä 22

4.4 Testivetoisen kehityksen tutkimuksesta 23

4.5 Testivetoiseen kehitykseen rinnastuvat menetelmät 24

4.5.1 Behavior-driven development 24

4.5.2 Acceptance test-driven development 24

5 Esimerkkisovelluksen toteutus 25

5.1 Sovelluksen kuvaus 25

5.2 Sovelluksen kehityksessä hyödynnetyt keskeiset teknologiat 26

5.3 Selainpuolen sovelluksen testivetoinen kehittäminen 29

5.3.1 React-komponentin yksikkötestaus 31

5.3.2 Redux-tilan yksikkötestaus 33

5.3.3 Sisäänkirjautumissivun integraatiotestaus 36

5.3.4 Testikattavuus 38

5.4 Palvelinpuolen sovelluksen testivetoinen kehittäminen 40

5.4.1 RESTful-rajapinnan reitin integraatiotestaus 40

5.4.2 Testikattavuus 42

5.5 Havainnot ja päätelmät 43

6 Pohdinta 45

Lähteet 47

Kuviot

Kuvio 1. Vesiputousmalli, jossa projektin vaiheet etenevät järjestyksessä vaiheesta toiseen. 10

Kuvio 2. Testauspyramidi, jossa yksikkötestien osuus on suurin, ja niitä täydentää integraatiotestit sekä järjestelmätestit. 17

Kuvio 3. Testauspokaali, jossa integraatiotesteillä on merkittävin rooli, ja niitä täydentävät yksikkö ja järjestelmätestit sekä staattinen analyysi. 18

Kuvio 4. Kuvaus testivetoisen kehityksen yhden iteraation vaiheista 21

Kuvio 5. EventItem React-komponentti 32

Kuvio 6. EventItem-komponentin testitapaus, jossa käyttäjä on annettu 33

Kuvio 7. Sisäänkirjautumisen hoitava toiminnallisuus toteutettuna Redux Toolkitilla 34

Kuvio 8. Testitapaus sisäänkirjautumiselle 35

Kuvio 9. Sisäänkirjautumissivua edustava komponentti 36

Kuvio 10. Sisäänkirjautumissivun testitapaus 37
Kuvio 11. Selainpuolen sovelluksen yksikkötestien testikattavuus 38
Kuvio 12. Selainpuolen sovelluksen Jestillä laskettu kokonaistestikattavuus 39
Kuvio 13. Selainpuolen sovelluksen integratiotestien kattavuus 39
Kuvio 14. Tapahtuman luontia vastaava reitti ja käsittelijäfunktio 41
Kuvio 15. Onnistuneen tapahtuman luomisen testitapaus 42
Kuvio 16. Palvelinpuolen sovelluksen testikattavuus 43

Sanasto

AJAX (Asynchronous JavaScript and XML)

Mahdollistaa web-sovelluksissa datan lähettämisen ja vastaanottamisen verkon yli asynkronisesti.

Asynkronisuus (asynchrony)

Ohjelman suoritus ei pysähdy odottamaan aikaa vieviä tapahtumia, vaan ne valmistuvat taustalla.

Autentikointi (authentication)

Prosessi, jolla varmistetaan palvelun tai järjestelmän käyttäjän identiteetti

Dokumenttioliomalli (DOM)

Tapa kuvata dokumentti, kuten HTML olioista muodostuvana puumaisena rakenteena, jonka elementtejä voi hakea ja muokata.

ECMAScript

JavaScript-ohjelmointikielen standardi

Funktio (function)

Aliohjelma, jota voidaan kutsua muualta koodista. Se voi ottaa vastaan parametreja ja voi palauttaa jonkin arvon.

Hypertext Markup Language (HTML)

Kuvauskieli, jolla voidaan kuvata hyperlinkkejä sisältävää tekstiä ja sen rakennetta.

HTTP-protokolla (Hypertext Transfer Protocol)

Verkkoselainten ja web-palvelinten väliseen tiedonsiirtoon kehitetty protokolla, joka toimii pyyntö-vastaus-periaatteella

JavaScript

Verkkoselainten tukema korkean tason dynaamisesti tyypitetty ohjelmointikieli

JavaScript Object Notation (JSON)

Tiedonsiirtoon tarkoitettu tekstidokumentin formaatti, jonka syntaksi on JavaScriptista johdettu.

Ohjelmointiympäristö (Development Environment)

Ohjelmakoodin kehittämiseen tarkoitettu ohjelmisto tai työkalujen kokoelma.

Komponentti (component)

Itsenäinen rakenteellinen tai toiminnallinen yksikkö, joita yhdistelemällä voidaan rakentaa suurempia kokonaisuuksia.

Moduuli (module)

Itsenäinen kokonaisuus, joka voi olla pienemmistä komponenteista rakennettu. Vastaa tavallisesti yhden tehtävän hoitamisesta, ja monimutkaisemman toiminnallisuuden saavuttamiseksi niitä voi kytkeä toisiinsa.

ODM (Object Document Mapping)

Hoitaa tietojen muuntamisen dokumenttipohjaisen tietokannan dokumenttien ja sovelluksen ymmärtämien olioiden välillä.

RESTful-rajapinta (Representational State Transfer)

Rajapinta, joka on toteutettu HTTP-protokollaan perustuvaa REST-arkkitehtuuria mukaillen

Väliohjelmisto (middleware)

Sovelluksen osien välillä toimiva komponentti, joka hoitaa jotain tehtävää tai toimii rajapintana osien välisessä kommunikaatiossa.

1 Johdanto

Muutamassa vuosikymmenessä tietotekniset laitteet ovat levinneet kaikkialle. Vielä 1970-luvulla harvat ihmiset käyttivät työssään tietokoneita, nykyään harvempi selviää päivästäkään työtä ilman tietoteknistä laitetta. Lähes jokaisessa laitteessa on jonkinlaista ohjelmistoa televisioista, autoihin ja kodinkoneisiin. Niinpä yhteiskuntamme on tullut hyvin riippuvaisiksi ohjelmistoista ja niiden toimivuudesta. Toimiessaan hyvin laitteet ja ohjelmistot helpottavat ihmisten elämää ja tuottavat taloudellista hyötyä, ja toimiessaan huonosti aiheuttavat suunnatonta turhautumista ja menetettyä työtä ja toimeentuloa. (Myers, Sandler & Badgett 2012, 1.)

Ohjelmistot ovat jo pitkään olleet niin monimutkaisia, ettei yksittäinen ohjelmoija pysty hallitsemaan koko ohjelmiston yksityiskohtia, ja tarkastettavia asioita ja mahdollisia lähteitä vikatilanteille on liikaa. 1960-luvun lopulla tälle ilmiölle annettiin nimi ”ohjelmistokriisi”. Ohjelmistoprojektien taipumuksena oli tuolloin ylittää budjettinsa ja aikataulunsa, ja niissä jouduttiin tinkimään laadullisten kriteerien tai vaatimusten täyttymisen osalta. (Kasurinen 2013, 10.)

Ohjelmistojen huonon laadun aiheuttamat ongelmat vaivaat ohjelmistoprojekteja yhä edelleen. Yrityksille ja yhteiskunnalle aiheutuu laatuongelmista valtavia tappioita. Pelkästään Yhdysvalloissa vuonna 2018 ohjelmistojen laatuongelmista aiheutuneet arvioidut kustannukset olivat noin 2,84 biljoonaa dollaria. Pääasiallisia kustannusten lähteitä olivat ohjelmistojen viat, vanhentuneiden järjestelmien aiheuttamat ongelmat, tekninen velka, vikojen etsiminen ja korjaaminen sekä vaikeuksiin ajautuneet tai peruuntuneet projektit. (The Cost of Poor Software Quality in the US: A 2018 Report)

Ongelmaa korjaamaan on kehitetty vuosien saatossa erilaisia ohjelmistotuotannon prosesseja, käytäntöjä ja malleja. Niissä keskeinen rooli on ohjelmistotestauksella ja sen menetelmillä. (Kasurinen 2013, 11.)

2 Tutkimusasetelma

2.1 Työn tavoitteet ja aikaisempi tutkimus aiheesta

Työn tavoitteena on tutkia, kuinka testivetoisen kehityksen menetelmää on mahdollista soveltaa web-sovelluksen kehitystyössä. Aihevalinta nousi tekijän omasta mielenkiinnosta ja halusta tutkia aihetta.

Testivetoisen kehityksen menetelmää on jonkin verran tuotu esiin eri yhteyksissä, esimerkiksi Helsingin yliopiston ohjelmoinnin peruskurssilla aiheeseen on lyhyt perehtyminen ohjelmien testausta käsittelevässä osiossa (ks. Ohjelmoinnin MOOC 2020: Johdatus ohjelmien testaamiseen). Mutta kovin laajaa ja yleistä menetelmän hyödyntäminen ei ole, ja vaikka menetelmää on tutkittu jonkin verran, tutkimusasetelmat ja mittarit ovat sen verran vaihtelevia, että tulokset eivät välttämättä ole kovin vertailukelpoisia (Dredge 2019).

2.2 Tutkimusmenetelmät

Kehittämistutkimuksessa lähtökohtana on tarve muutokselle, eli ratkaista jokin ongelma tai kehittää jotain paremmaksi, ja sen lopputuloksena syntyy jokin tuotos. Tutkimuksen kohteena on jokin prosessi, toiminta, asiantila tai tuote. Kehittämistutkimus on luonteeltaan monimenetelmäinen ja tutkimus pohjautuu johonkin teoriataustaan. Jotta kehittämistyö olisi tutkimusta, se edellyttää tutkimuksellista otetta. Kehittämistyö dokumentoidaan ja käytetään tieteellisiä menetelmiä tarkoituksena tuottaa uutta tietoa. (Kananen 2012, 19–21.)

Tutkimusmenetelmäksi valikoitui kehittämistutkimus, koska tavoitteena on tutkia ilmiötä käytännön kehittämistyön kautta. Kehittämistyönä toteutetaan esimerkkisovellus web-teknologioilla soveltaen testivetoisen kehityksen periaatteita.

Raportin teoriaosuudessa avataan yleisesti testaukseen, web-sovellusten testaukseen sekä testivetoiseen kehittämiseen liittyviä käsitteitä valitun aiheen kontekstissa.

Toteutusosiossa tutustutaan esimerkkisovelluksen kehittämisessä hyödynnettäviin työkaluihin ja teknologioihin, tarkastellaan koodiesimerkkien kautta tehtyjä ratkaisuja ja saavutettua testikattavuutta. Lopuksi arvioidaan menetelmän soveltamisesta havaittuja hyötyjä ja rajoitteita sekä kohdattuja ongelmia, ja vastataan asetettuihin tutkimuskysymyksiin.

2.3 Rajaukset

Työn ulkopuolelle on rajattu testauksen teorian ja käytäntöjen laajempi käsittely, esimerkiksi käsin testauksen menetelmiä työssä ei käsitellä juurikaan. Myöskään vaihtoehtoisia web-sovellusten kehittämisessä ja testivetoisessa kehityksessä hyödynnettäviä työkaluja ja teknologioita ei käydä läpi ja vertailla, koska niitä on paljon, ja periaatteet sekä toiminnallisuudet ovat osittain samoja. Tutkimuksellisen asetelman kannalta olisi myös hyödyllistä toteuttaa sama esimerkkisovellus ilman testivetoisen menetelmän noudattamista, ja verratta toteutuksia eri mittareilla, mutta se on jätetty pois toteutustavan työläyden vuoksi.

2.4 Tutkimuskysymykset

Tavoitteen ja rajausten pohjalta työlle hahmottui seuraavat tutkimuskysymykset:

- Mitä testivetoinen kehitys tarkoittaa, ja miten se liittyy ohjelmistotestaukseen?
- Kuinka testivetoisen kehityksen menetelmä soveltuu web-sovelluksen kehittämistyöhön?
- Millaisia hyötyjä ja rajoitteita testivetoisen kehityksen menetelmän noudattamisessa havaitaan?

3 Testaus osana ohjelmistotuotantoa

Ohjelmistotestaus on yksi ohjelmistotuotannon piiriin kuuluvista kokonaisuuksista. Testaus kuitenkin kattaa laajemman kokonaisuuden verrattuna esimerkiksi ohjelmointiin tai suunnitteluun. Ohjelmistotestauksen tarkoitus on taata, että kehitettävä ohjelmisto tai järjestelmä vastaa sitä, mitä siltä on haluttu, ja toteutetut ominaisuudet toimivat niin kuin on tarkoitettu. Tällöin puhutaan validaatiosta ja verifikaatiosta, eli tehdään oikeaa tuotetta ja että se on tehty oikein. (Kasurinen 2013, 10.)

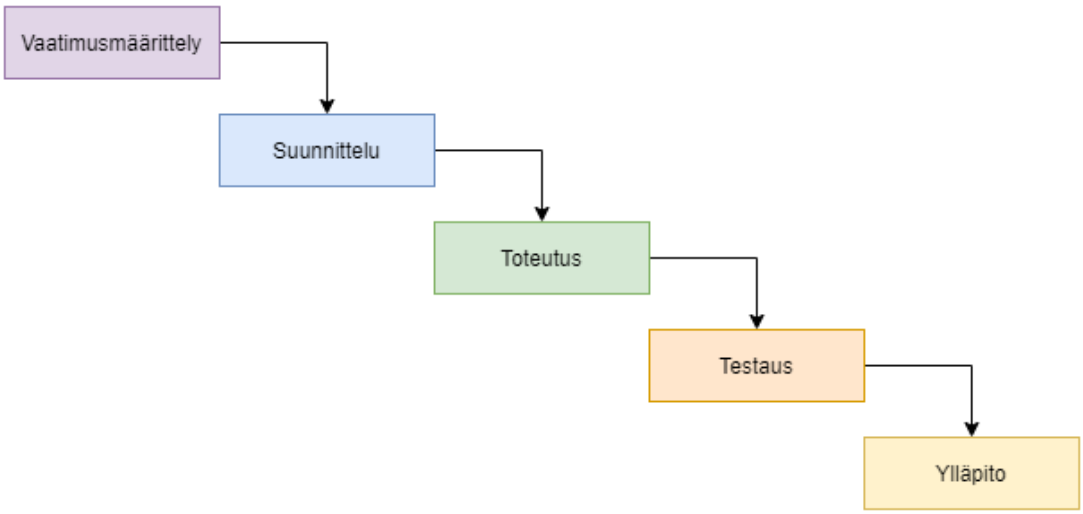
Ohjelmistotestaukseen pätee edelleen peukalosääntö, että puolet ajasta ja yli puolet kustannuksista kuluu kehitettävän ohjelmiston tai systeemin testaukseen (Myers, Sandler & Badgett 2012, ix). Kuitenkin virheen korjaaminen suunnitteluvaiheessa on vain murto-osa kustannuksista verrattuna saman virheen korjaaminen julkaisun jälkeen. Testaus tuottaakin kaikkein isoimman hyödyn kannattavuudella mitattuna. Hyvin testaavat yritykset pystyvät saamaan suuremman katteen tuotteillaan vertailtaessa huonosti testaaviin yrityksiin, ja eroa voi vielä korostaa vaikutukset yrityksen maineelle. (Kasurinen 2013, 12.)

3.1 Testaus ohjelmistotuotannon elinkaarimalleissa

Ohjelmistotuotannon elinkaarimallit kuvaavat sitä, kuinka ohjelmistoprojekti tulisi saattaa alusta loppuun ja miten eri työvaiheet prosessin aikana etenevät. Malleja on vuosikymmenten saatossa kehitetty useita erilaisia. Ne jakautuvat karkeasti suunnitelmalähtöisiin ja ketteriin menetelmiin. (Sami 2012.) Seuraavassa on käsitelty perinteistä vesiputousmallia sekä ketteriä menetelmiä kokonaisuutena.

3.1.1 Vesiputousmalli

Ohjelmistotuotannon perinteisessä mallissa, eli vesiputousmallissa ohjelmistoprojekti on jaettu järjestyksessä toisiaan seuraaviin vaiheisiin, ja ihannetilanteessa edellisiin vaiheisiin ei tarvitse enää myöhemmin palata (Kasurinen 2013, 12–14).



Kuvio 1. Vesiputousmalli, jossa projektin vaiheet etenevät järjestyksessä vaiheesta toiseen (Lynch 2019, muokattu).

Projekti alkaa vaatimusten määrittelyllä, missä koostetaan taustatutkimuksen aineiston pohjalta lista ohjelmiston ominaisuuksista. Sitä seuraa sovelluksen suunnittelu, ja sen pohjalta toteutetaan komponenttien ohjelmointityö ja koostaminen. Vasta ohjelmiston implementoinnin jälkeen tulee testauksen vuoro, missä varmistetaan ohjelmiston olevan toimiva, täyttävän vaatimukset ja toteutuksen vastaavan suunniteltua. Kun ohjelmisto on testattu, suoritetaan käyttöönotto, ja se siirtyy ylläpitovaiheeseen. (Mts. 12–14.)

Vesiputousmalli on yksinkertainen ja johdonmukainen, mutta käytännön ohjelmistoprojekteissa se on osoittautunut ajan saatossa monessa tapauksessa liian jäykäksi ja muutoksiin huonosti vastaavaksi toimintatavaksi. Ongelmia aiheuttaa se, että ohjelmiston käyttäjät eivät välttämättä osaa tarkkaan tunnistaa kaikkia vaatimuksia, ennen kuin pääsevät käyttämään sitä, ja toisaalta suunnittelijat eivät pysty varautumaan kaikkiin tuleviin pulmakohtiin etukäteen. (Lynch 2019.)

Vesiputousmalli ei taivu helposti iteratiivisuuteen ratkaistakseen edellä kuvattuja ongelmia, ja muutokset tulevat kalliiksi. Muutokset myös vaikuttavat projektin aikataulutukseen, ja joko ohjelmiston käyttöönotto viivästyy, tai muutosten toteuttamiseen

kuluva aika on pois myöhemmiltä vaiheilta, jolloin erityisesti testaukselle ei jää riittävästi aikaa, ja tämä näkyy ohjelmiston laatuongelmina. (Mts.)

3.1.2 Ketterät menetelmät

Internet-aika mahdollistaa ohjelmistojen viiveettömän jakelun, ja kuluttajat ovat tottuneet odottamaan mahdollisimman laadukasta jälkeä mahdollisimman nopeasti. Perinteiset ohjelmistotuotannon prosessit ovat osoittautuneet riittämättömäksi vastaamaan näihin odotuksiin. (Myers, Sandler & Badgett 2012, 175)

2000-luvun alussa joukko ohjelmistokehittäjiä kokoontui keskustelemaan nopeamman ja kevyemmän kehityksen tavoista, sekä onnistuneiden ja epäonnistuneiden projektien ominaispiirteistä. Sen tuloksena syntyi Agile Manifesto -nimeä kantava asiakirja, jossa määriteltiin joukko asiakas-, työntekijä-, ja muutoslähtöisiä periaatteita. Tämän voi ajatella lähtölaukauksena ketterän ohjelmistokehityksen liikkeelle. (Mts. 175.)

Toisaalta IBM on käyttänyt ketteriä tuotantomenetelmiä jo 1950-luvulla, ja yhden suosituimman ketterän kehityksen menetelmän Scrumin ensimmäinen version julkaistiin jo vuonna 1995 (Kasurinen 2013, 27).

Ketterän kehityksen alaisia menetelmiä on useita erilaisia, mutta yhteistä ja leimallista niille on, että asiakas ja asiakastyytyväisyys on keskiössä, kehitys on iteratiivista ja inkrementaalista sekä avointa muutoksille, ja lisäksi testaus on mukana isossa roolissa koko prosessin ajan. Ketterän kehityksen menetelmän omaksuminen ja käyttöönotto organisaatioissa voi osoittautua haastavaksi, koska ajattelu- ja toimintatavoiltaan ne eroavat selkeästi perinteisistä prosesseista. (Mts. 176–178.)

Ketterässä testauksessa kaikki projektissa mukana olevat toteuttavat sitä yhteistyössä, ja se vaatii hyvää viestintää ja yhteistoimintaa. Asiakas on testauksessakin keskeisesti osallisena, ja heti kun ohjelmistossa on jotain toiminnallisuuksia kehitettynä, niitä voidaan alkaa hyväksymistestata. Tästä syntyy jatkuva palautesilmukka, jossa saadun palautteen perusteella kehitettyjä ominaisuuksia voidaan parantaa tai

muuttaa, ja suunniteltuja priorisoida tai täsmentää uudelleen. Ketterän kehityksen menetelmissä eri testaustasojen ja testausautomaation hyödyntäminen on merkittävässä roolissa, jotta kehityssyklit pysyvät nopeana. (Mts. 178–179.)

Verrattaessa suunnitelmalähtöisiä menetelmiä ketteriin menetelmiin, niin ensin mainituilla kuten vesiputousmallilla lähtökohtana on varautua mahdollisimman monenlaisiin mahdollisesti vastaantuleviin ongelmatilanteisiin, kun taas ketterillä menetelmillä tavoitteena on reagoida nopeasti ilmenneisiin pulmakohtiin (mts. 27). Ketterillä menetelmillä voi olla haastavaa hallita isomman kokoluokan monimutkaisia projekteja, joissa tiimin koko on suurempi, ja näihin tapauksiin suunnitelmalähtöiset menetelmät voivat soveltua paremmin (mts. 150).

3.2 Testausautomaatio

Ohjelmistot tarvitsevat parannuksia, uusia ominaisuuksia ja virheiden korjausta, jotta ne säilyttäisivät käyttökelpoisuutensa. Jotta muutoksia olisi järkevää tehdä, niiden tulee olla kustannustehokkaita ja muutosten avulla pitäisi pystyä saavuttamaan haluttu lopputulos. (Subramaniam 2016, 1.)

Ohjelmistoihin tehdyillä muutoksilla voi usein olla sivuvaikutuksia: Muutos yhteen paikkaan voi rikkoa yhden tai useamman muun asian. Siksi tehtyjen muutosten jälkeen on aina paikallaan kysyä, toimiiko ohjelmisto edelleen. Ilman testausta kysymykseen on vaikea antaa kovin luotettavaa vastausta. (Mts. 2.)

Jokaisen tehdyn muutoksen vaikutusten käsin testaaminen on työlästä ja hidastaa kehitystyötä. Testausautomaation hyödyntämisellä voidaan korvata sellaista ohjelmiston regressioiden varalta usein toistuvaa käsin testaamista, joka on mahdollista automatisoida. (Mts. 2.)

Testausautomaatio ei poista tarvetta käsin tehtävälle testaukselle. Ideaalitilanteessa molempia hyödynnetään sellaisissa testaustoiminnoissa, mihin ne parhaiten soveltuvat. Käsin testaus on tehokkaampi menetelmä validaation osalta, eli saadaan käsitys siitä, tekeekö sovellus haluttuja asioita ja onko se käytettävä. Testausautomaatiolla

taas voidaan hoitaa verifikaatiota eli varmentaa, että ohjelmisto toimii niin kuin pitää. (Mts. 3.)

Testausautomaation hyödyntäminen ei ole mikään hopealuoti, vaan sen käyttöönotto vaatii huolellista harkintaa saavutettavien hyötyjen ja vaadittavien panostusten välillä. Testausautomaation toteuttaminen ei ole halpaa, vaan vaatii pidempää kehitysaikaa alkuvaiheessa, ja siitä saatavat hyödyt tulevat näkyviin vasta pidemmällä aikavälillä. (Five things you should evaluate before proceeding into test automation 2017.)

Jotta automaattisesta testistä olisi hyötyä verrattuna käsin suoritettuun testaamiseen, testin suorituskertoja pitäisi kertyä riittävästi, jolloin käsin testaamiseen kuluisi enemmän aikaa kuin automaattisen testin implementointiin ja ylläpitoon kuluva aika. Siksi automaattinen testaus vaatii huolellista suunnittelua ja valintoja, mitä tulisi automatisoida. (Mts.)

Testausautomaation hyödyntäminen vaatii myös vahvaa osaamista kehitystyöhön osallistuvilta henkilöiltä. Verrattuna käsin testaamiseen, automaattisessa testauksessa ohjelmistokehittäjät ovat keskeisemmässä roolissa, koska toisaalta ohjelmistokoodin tulee olla toteutettu testattavuuden ehdoilla, ja toisaalta automaattiset testitkin ovat osa tuotantokoodia, ja niiden kirjoittaminen vaatii ohjelmointiosaamista. Testausautomaation toteuttaminen jo olemassa olevaan järjestelmään on erityisen haastavaa, jos järjestelmä ei ole toteutettu testattavuus edellä. (Mts.)

Testausautomaatiota kannattaa hyödyntää erityisesti sellaisissa tapauksissa, kun pitää varmistaa, että muutokset eivät ole rikkoneet aiemmin toimineita osia. Se on enemmän laadunvalvonnan kuin testauksen työkalu, ja soveltuu huonommin uusien osien toimivuuden tutkimiseen. Testien jatkuva muuttelu ja ylläpito on kallista toimintaa. (Kasurinen 2013, 78–79.)

3.3 Testaustasot

Testausta voidaan tehdä eri tasoilla riippuen siitä, kuinka isoa osaa tai kokonaisuutta järjestelmästä testeissä tarkastellaan, ja nämä tasot ovat yksikkötestaus, integraatiotestaus, järjestelmätestaus ja hyväksyntätestaus (Kasurinen 2013, 50–51). Lisäksi joissain tapauksissa omaksi testaustasoksi voidaan määrittää staattinen analyysi (Sapegin 2019).

3.3.1 Yksikkötestaus

Yksikkötestaus on matalin testaustaso, ja yksikkötestit testaavat tavallisesti yksittäisiä toimintoja tai kokonaisuuksia, kuten funktiota, metodeja, komponentteja, moduuleja ja luokkia. Yksikkötesteissä pyritään eristämään testattavat asiat siten, ettei testien tulokseen ei vaikuta testattavan asian ulkopuoliset tekijät tai riippuvuudet. Jotta yksikkötestejä on mahdollista laatia järkevästi, tulee koodin osat olla keskenään löyhästi sidottu (loose coupling) ja koodin tulee olla rakenteeltaan modulaarista. (Öztürk 2020.)

Yksikkötestien etu on se, että ne ovat nopeita suorittaa. Ne usein ovat myös melko luotettavia ja johdonmukaisia siinä mielessä, että testattaessa pieniä yksiköitä kerrallaan, testien tuloksiin vaikuttavat ulkopuoliset tekijät on saatu eristettyä niistä pois. Keskittyminen pieniin palasiin tuo mukanaan myös sen edun, että testin epäonnistuessa, on virheen löytäminen usein helppoa, koska läpi käytäviä koodirivejä on rajallisesti. (Wacker 2015.)

Erityisesti yksikkötestauksessa hyödynnetään testattavan yksikön ulkopuolisien riippuvuuksien rajaamiseksi testitapauksen ulkopuolelle erilaisia testikomponentteja tai testitynkiä (mocks, stubs, shims) korvaamaan tai sijaistamaan ulkopuolisia riippuvuuksia (Kasurinen 2013, 52).

Ulkopuoliset riippuvuudet voivat tuoda testituloksiin ennakoimattomuutta, ne saattavat olla vielä kehittämättä tai kehityksen alla, tai niihin ei välttämättä ole testien si-

sältä pääsyä, tai ne hidastavat testien suorittamista (esimerkiksi autentikointia vaativat toimet tai verkon yli tapahtuvat kutsut). Testikomponenttien tai -tynkien hyödyntämisellä voidaan varmistaa, että esimerkiksi testattavan komponentin riippuvuus palauttaa aina tietyn, halutun arvon tai tilan, jotta testitapaukseen ei tule epätoivottua vaihtelua. (Paolini-Subramanya 2018.)

3.3.2 Integraatiotestaus

Integraatiotestauksella testataan kahden tai useamman yksikön tai moduulin yhteen kytketyn kokonaisuuden muodostamaa toimintaa. Siispä testaustaso on korkeampi kuin yksikkötestien, ja testit edustavat lähemmin sitä, miten asiat toimivat todellisessa suoritusympäristössä. (Öztürk 2020.)

Integraatiotestauksen etuna on, että sen avulla voidaan testata sellaisia asioita tai löytää sellaisia vikoja, mitkä ilmenevät vain useamman komponentin yhteen kytkemisessä ja toiminnassa, mitä ei voida yksikkötestauksella saavuttaa (Wacker 2015).

Yksiköiden tai moduulien integrointijärjestykselle voidaan tunnistaa neljä erilaista mallia. Alhaalta ylöspäin -mallissa (bottom up) aloitetaan integrointien tekeminen matalimman tason moduuleista. Ylhäältä alaspäin -mallissa (top down) on idea päinvastainen, eli aloitetaan ylimmän tason integraatioista. Voileipä-mallissa (sandwitch) yhdistetään molemmat edellä mainitut suunnat, ja kertarysäys-mallissa (big bang) kaikki komponentit integroidaan yhteen kerralla. Integraatiotesteissä voidaan myös joutua käyttämään sijaiskomponentteja (stubs), joiden ylläpitäminen voi olla työlästä. (Kasurinen 2013, 54–55.)

3.3.3 Järjestelmätestaus

Järjestelmätestauksessa tai päästä-päähän testauksessa (system testing, end-to-end testing, E2E) keskiössä on järjestelmän testaaminen kokonaisuutena, jossa kaikki sen osat ovat integroituna ja toiminnassa. Testit suoritetaan usein käyttäjän näkökulmasta, ja niissä ei pitäisi enää olla käytössä testi- tai sijaiskomponentteja. (Öztürk 2020.)

Verrattaessa yksikkö- ja integraatiotestaukseen, mitkä testaavat vain osaa järjestelmästä, järjestelmätestien etuna on, että niiden avulla saadaan todenmukaisin kuva järjestelmän toimivuudesta kokonaisuutena. Järjestelmätestien huonona puolena kuitenkin on, että ne ovat vaativimpia toteuttaa ja hitaimpia suorittaa. (Mts.)

Niissä voi myös todennäköisimmin esiintyä testiympäristöstä tai -työkaluista riippuen epäluotettavuutta, jolloin testi välillä menee läpi ja välillä ei, vaikka ohjelmaan ei olisi tehty muutoksia. Myös virheiden juurisyiden jäljittäminen voi viedä enemmän aikaa, koska epäonnistuneen testin taustalla voi olla monta yhteen kytkettyä osaa, ja virhe voi sijaita missä tahansa niistä tai niiden liitoksissa. Ja kun virhe on saatu korjattua, palaute testin läpimenosta tulee hitaammin. (Wacker 2015.)

3.3.4 Hyväksymistestaus

Hyväksymistestauksella (acceptance testing) voidaan todentaa sitä, täyttääkö kehitettävä järjestelmä kaikki sille asetetut vaatimukset esimerkiksi käyttäjätarinoiden näkökulmasta. Niissä näkökulma on myös koko järjestelmän tasolla ja käyttäjän perspektiivistä. (Pittet n.d.)

Hyväksymistestausta suoritetaan tavallisesti projektin loppuvaiheessa, jolloin järjestelmään ei ole odotettavissa enää merkittäviä muutoksia, ja siinä asiakas tai järjestelmän tilaaja on mukana varmistamassa, että heidän näkökulmastaan järjestelmälle asetetut vaatimukset ja laatu täyttyvät, jolloin järjestelmä voidaan ottaa käyttöön. Kun järjestelmätestauksessa testit ajetaan testiympäristössä, voi hyväksymistestauksessa olla käytössä järjestelmän kohdeympäristö. (Kasurinen 2013, 57.)

3.3.5 Staattinen analyysi

Staattisella analyysillä (static analysis) tarkoitetaan sellaista toimintaa, missä erilaisilla työkaluilla voidaan tarkastella tietotyyppien oikeellisuutta, sekä hyvien koodauskäytänteiden ja koodin ulkoasusääntöjen noudattamista, ja ne toimivat usein automaattisesti ohjelmointiympäristön taustalla huomauttaen tai korjaten virheitä ja

poikkeamia (Sapegin 2019). Staattisen analyysin työkalut vähentävät ohjelmistokehittäjien tarvetta keskittyä edellä mainittujen asioiden tarkastamiseen ja lisäävät koodin luotettavuutta (Dodds 2019).

3.4 Testaustasojen välinen tasapaino

3.4.1 Testauspyramidi

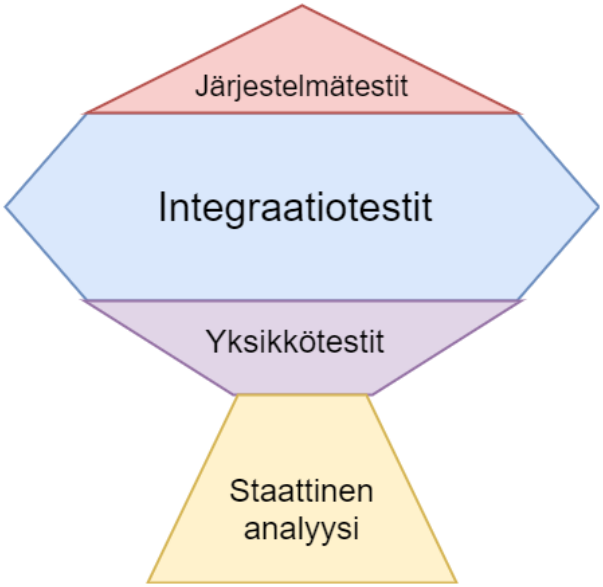
Testauspyramidin (testing pyramid) tarkoitus on havainnollistaa tavoiteltavaa automaattisten testien testaustasojen suhdetta (ks. kuvio 2). Siinä lähtökohta on, että suurin osa testeistä olisi yksikkötestejä, jotka ovat nopeita määrittää ja suorittaa. Niitä täydentävät integraatiotestit, joilla voi osin myös korvata järjestelmätestauksen tarvetta. Pyramidin huippu on varattu huolella valikoiduille järjestelmätesteille. (Fowler 2012.)

Kuvio 2. Testauspyramidi, jossa yksikkötestien osuus on suurin, ja niitä täydentää integraatiotestit sekä järjestelmätestit (Wacker 2015, muokattu).

Järjestelmätestien pientä osuutta yksikkötesteihin nähden perustellaan sillä, että järjestelmätestit ovat hitaita suorittaa, työläitä laatia ja ylläpitää, ja niihin liittyy epädeterministisyyden tuomaa epäluotettavuutta (mts.).

3.4.2 Testauspokaali

Tuoreempi malli testaustasojen väliselle tavoiteltavalle tasapainolle on testauspokaali (testing trophy, ks. kuvio 3). Se pohjautuu osittain automaattisen testauksen työkalujen kehittymisen tuomille hyödyille ja sille oletukselle, että korkeamman tason testit tuovat enemmän luotettavuutta sille, että ohjelmisto toimii kuten on tarkoitus. (Dodds 2019.)



Kuvio 3. Testauspokaali, jossa integraatiotesteillä on merkittävin rooli, ja niitä täydentävät yksikkö ja järjestelmätestit sekä staattinen analyysi (Dodds 2019, muokattu).

Pokaalin jalustan muodostaa staattiset analyysin työkalut, jotka ovat edullinen tapa vähentää virheitä ohjelmakoodissa. Suurin osuus testeistä tulisi olla integraatiotestejä, jotka ovat paras kompromissi luotettavuuden ja kustannustehokkuuden suhteen. (Mts.)

Mallissa yksikkötestit ovat pienemmässä roolissa verrattuna testauspyramidiin, koska yksikkötesteissä käytetään paljon testitynkiä (mocks), ja ne vähentävät varmuutta siitä, että komponentit toimivat silloinkin, kun ne on integroitu toimimaan keskenään (mts.).

3.5 Web-sovelluksen testauksen erityispiirteitä

Web-sovellus on web-teknologioilla toteutettu sovellus, joka tavallisesti koostuu käyttäjän verkkoselaimessa toimivasta selainpuolen sovelluksesta, joka kommunikoi verkon yli verkkopalvelimella pyörivän palvelinpuolen sovelluksen kanssa. Palvelinpuolen sovellus voi hoitaa tietojen tallentamisen ja noutamisen tietokannasta. Tyypillisiä web-sovelluksia ovat esimerkiksi erilaiset verkkokaupat, sähköpostisovellukset tai selaimessa toimivat tekstin- ja kuvankäsittelyohjelmat. (Gibb 2016.)

Web-sovelluksiin liittyvät erityispiirteet, kuten verkkoselaimet suoritusalustana ja verkon yli tapahtuva tiedonsiirto, aiheuttaa myös erityistarpeita niiden testaamiselle, jotta voidaan varmistua siitä, että sovellus on toimiva ja turvallinen käyttää (Vogels n.d.).

Toiminnallisuustestauksella (functionality testing) varmistetaan, että sovelluksen kaikki linkit, evästeet, ja lomakkeet toimivat, ja yhteydet tietokantaan ovat kunnossa. Käytettävyystestauksella (usability testing) arvioidaan sitä, että käyttäjän näkökulmasta sovellusta on intuitiivista ja luontevaa käyttää. Rajapintatestauksella (interface testing) varmennetaan, että kaikki selain- ja palvelinsovelluksen välillä olevat kutsut toimivat, ja että virhe- ja poikkeustilanteet on käsitelty asianmukaisesti. (Timotic 2018.)

Jotta voidaan olla varma, että sovellus toimii toivotulla tavalla eri tarjoajien verkkoselaimilla niin työpöytäkäytössä kuin mobiililaitteilla ja erilaisissa käyttöjärjestelmissä, tarvitaan yhteensopivuustestausta (compatibility testing). Suorituskykytestauksella (performance testing) mitataan sovelluksen toimivuutta hitaillakin verkkoyhteyksillä ja palvelinsovelluksen kykyä suoriutua normaalista sekä normaalia kovemmasta kuormituksesta. Myös tietoturvatestaus (security testing) on tärkeä osa web-sovellusten testausta, ja sillä pyritään löytämään sovelluksesta mahdollisia tietoturva-aukkoja, jotta pystyttäisiin ehkäisemään mahdolliset tietomurrot ja muut tietoturvaan liittyvät uhat ja riskit. (Mts.)

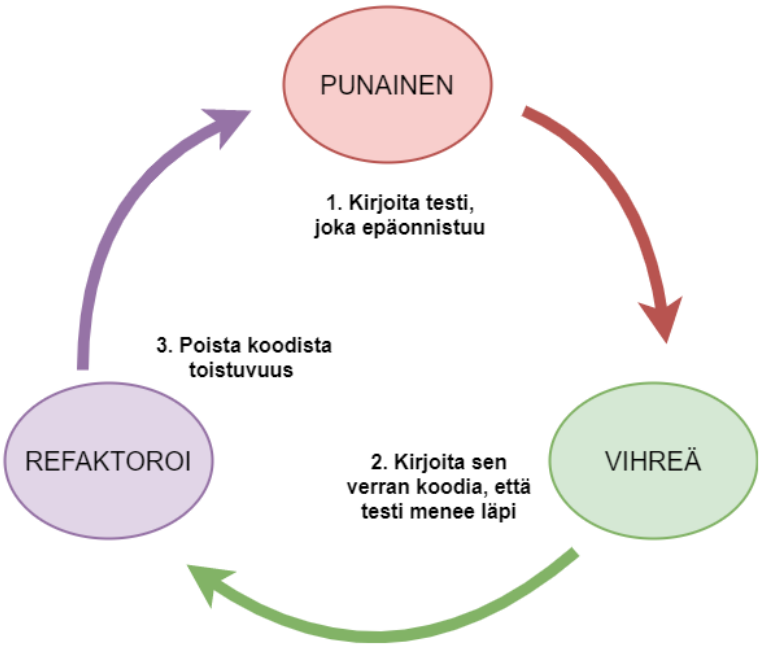
4 Testivetoinen kehitys

Testivetoisen kehityksen (Test-driven development, TDD) voi ajatella olevan lähtöisin 1960-luvulta, kun tietokoneita ohjelmoitiin vielä reikäkorteilla. Ohjelmoijilla oli rajoitetusti aikaa käytettävissä koneella, jolloin oli tärkeä käyttää aika mahdollisimman tehokkaasti. Tuolloin eräs käytäntö oli kirjoittaa ylös haluttu tulos, ja verrata sitä sitten koneen antamaan tulokseen. (Barber 2012.)

Testivetoinen kehitys on osa Extreme Programming (XP) -nimisen ketterän kehityksen metodologian käytäntöjä. Sen keskeinen kehittäjä ja tunnetuksi tekijä on yhdysvaltalainen ohjelmistoinsinööri Kent Beck. Testivetoista kehitystä harjoitettiin aluksi Smalltalk-nimiseen ohjelmointikieleen keskittyvän yhteisön parissa, mutta sen hyödyntäminen yleistyi Javalle julkaistujen automaattisen testauksen työkalujen kehityksen myötä. (Mts.)

4.1 Testivetoisen kehityksen periaatteet

Testivetoisen kehityksen tavoite on auttaa tuottamaan selkeää ja toimivaa koodia. Toisaalta se on keino hallita pelkoa, joka liittyy vaikeiden ongelmien ratkaisemiseen. Siinä tyypillinen työjärjestys – eli kehitetään ensin toiminnallisuus ja laaditaan sille myöhemmin testitapaukset – käännetään ikään kuin päälaelleen. Kehittäjä luo ja suorittaa itse koodille automaattisia yksikkötestejä koodin tuottamisen ja muokkaamisen lomassa. (Beck 2002, ix.)



Kuvio 4. Kuvaus testivetoisen kehityksen yhden iteraation vaiheista (Singh 2017, muokattu)

Testivetoisessa kehityksessä työskentely kulkee pienissä iteraatioissa, joiden vaiheet ovat punainen, vihreä ja refaktorointi (ks. kuvio 4). Punaisessa vaiheessa kirjoitetaan jotain kehittävän toiminnallisuuden osaa testaava automaattinen testi. Sen ei pitäisi mennä läpi tai välttämättä edes kääntyä, koska testattavaa toiminnallisuutta ei ole vielä toteutettu, ja siten on todennettavissa, että testin läpimeno vaatii muutosta. (Mts. 1–8.)

Vihreä vaihe edellyttää, että kirjoittaa vain sen verran koodia, että testi menee läpi. Ratkaisu voi olla tässä vaiheessa järjetönkin, esimerkiksi koodi palauttaa vain jonkin testin edellyttämän vakioarvon. (Mts. 1–8.)

Kun testi näyttää vihreää, siirrytään refaktorointivaiheeseen, jossa poistetaan koodista toistuvuus, jolla edellisessä vaiheessa saimme koodin toimimaan. Toistuvuus voi olla esimerkiksi saman lausekkeen tai logiikan kopioimista useaan eri paikkaan ohjelmakoodissa, tai se voi olla saman vakioarvon toistuminen testin ja testattavan koodin välillä. Iteraatioita toistetaan, kunnes kaikki toiminnallisuudet ovat toteutettu. (Mts. 1–8.)

Testivetoisen kehityksen idea on siis toisaalta tarvittaessa tarjota mahdollisuus ongelmien pilkkomiseen hyvin pieniksi askeliksi tai vaiheiksi, ja toisaalta ajaa kehitystä automaattisilla testeillä, jotka jatkuvasti varmentavat sitä, että koodin muutoksilla saamme aikaan haluttuja asioita, emmekä siinä sivussa riko jo toimivia asioita. (Mts. 9.)

4.2 Testivetoisen kehityksen hyötyjä

Testivetoisella kehityksellä voi olla monia potentiaalisia hyötyjä. Kehityksen aikana kertyvät testit luovat eräänlaisen turvaverkon, joka tarjoaa palautetta siitä, jos muutokset rikkovat asioita. Muutoksien tekeminen on tavallisesti sitä kalliimpaa, mitä myöhäisemmässä vaiheessa projektia niitä joutuu tekemään, mutta kattavat automaattiset testit helpottavat koodin refaktorointia ja muutoksien tekemistä pidemmällekin ehtineessä projektissa. (Llopis 2005.)

Testivetoisen kehityksen noudattaminen voi myös vaikuttaa ohjelmakoodin rakenteeseen ja modulaarisuuteen, koska kehittäjä joutuu miettimään, miten jotain komponenttia tai moduulia käytetään, eli millainen rajapinta sillä tulisi olla, ennen implementaation toteuttamista. Koodin testattavuus myös ohjaa parempaan modulaarisuuteen ja löyhempiin komponenttien välisiin riippuvuuksiin. (Mts.)

Muina hyötyinä voidaan pitää sitä, että testit ikään kuin luovat dokumentaatiota koodille, kuinka sitä tulisi käyttää. Myös jatkuvan ja välittömän palautteen saaminen ohjelmointityön kuluessa voi olla tärkeä subjektiivinen hyöty ohjelmistokehittäjän kannalta, ja se voi tehdä työn tekemisestä mielekkäämpää. (Singh 2017.)

4.3 Testivetoisen kehityksen rajoitteita ja kritiikkiä

Kaikkia ohjelmoitavia asioita ei voida helposti testata automaattisilla testeillä, kuten samanaikaisuutta tai tietoturvaan liittyviä asioita. Tämä asettaa selkeitä rajoituksia testivetoisen kehityksen hyödyntämiselle. (Beck 2002, xii.)

Ohjelmistokehittäjien keskuudessa on myös kritisoitu testivetoista kehitystä ja sen soveltamisessa on havaittu useita rajoitteita. Testivetoisen kehityksen omaksuminen ei välttämättä ole helppoa sen näennäisestä yksinkertaisuudesta huolimatta, ja voi vaatia paljon työtä ja vaivannäköä. (Disadvantages of Test Driven Development n.d.)

Testien kirjoittaminen ja ylläpitäminen vie aikaa, ja jos ohjelmiston vaatimuksiin tulee muutoksia, koodin lisäksi pitää muuttaa myös kaikki testit, joihin muutokset vaikuttavat. Ohjelmistokehityksessä tehtävät ratkaisut ovat aina kompromissi laadun, ajankäytön ja kustannusten kesken. (Li 2018).

Rajoitteena voi olla myös esimiehet tai muut tiimin jäsenet, jotka eivät ymmärrä tai koe tarpeellisena testivetoisen kehityksen pidemmän tähtäimen hyötyjä sen vaatimiin panostuksiin nähden. (Disadvantages of Test Driven Development.)

Testivetoisen kehityksen matalaan hyödyntämisasteeseen voi olla erilaisia syitä. Monilla kehittäjillä voi olla rajoittunut testausosaaminen. Myöskään testivetoisen kehityksen soveltamiseen ei välttämättä ole riittävästi hyviä oppaita ja resursseja saatavilla, missä olisi selkeitä malleja ja reseptejä hyvien käytäntöjen soveltamiseen. Yksi syy voi olla myös, ettei sitä juurikaan käsitellä yliopistoissa ja korkeakouluissa, ja sen takia käytännön kosketuspinta uusilla ohjelmistokehittäjillä on siihen heikko. (Arcuri 2018.)

4.4 Testivetoisen kehityksen tutkimuksesta

Jonkin verran tutkimuksia testivetoisen kehityksen mahdollisista hyödyistä ja rajoitteista on tehty, mutta tuloksista ei voi vetää kovin selkeitä johtopäätöksiä (Fucci 2016, 21). Oulun yliopistolle tehdyssä väitöskirjassa todettiin, että kokeneilla ohjelmistokehittäjillä testivetoisen kehityksen käytäntöjen noudattaminen ei tuonut parannuksia tuottavuudessa tai laadussa, mutta kokemattomat kehittäjät voivat hyötyä yksikkötestien toteuttamisesta testivetoisesti (mts. 93–94).

4.5 Testivetoiseen kehitykseen rinnastuvat menetelmät

4.5.1 Behavior-driven development

Behaviour-driven development -menetelmä (BDD) on kehitetty vastauksena testivetoisen kehityksen herättämiin pulmakohtiin, joihin etenkin sitä vasta opettelevat usein törmäävät. Tällaisia pulmia ovat: Mistä aloittaa, mitä testata ja mitä ei, miten nimetä testit, kuinka paljon testata kerralla, ja miten toimia tilanteessa, kun testit eivät mene läpi. (North 2006.)

BDD:n näkökulma on ajatella ja kirjoittaa testejä haluttuna käyttäytymisenä, eli kun tilanne on jonkinlainen, ja jotain tapahtuu, niin lopputuloksen pitäisi olla jotain. Tämä tarjoaa yhteisen kielen projektin eri aihealueiden asiantuntijoille ja sidosryhmille, ja on myös yhteneväinen sen muodon kanssa, miten ketterissä menetelmissä käyttäjätarinat ilmaistaan. Yhteys haluttuun käyttäytymiseen myös usein kertoo syyn, miksi jokin testi epäonnistuu: Joko on ilmennyt bugi, haluttu käyttäytyminen on siirtynyt toiseen paikkaan koodissa, tai käyttäytyminen ei ole enää toivottua. (Mts.)

4.5.2 Acceptance test-driven development

Acceptance test-driven development -menetelmä (ATDD) eroaa testivetoisesta kehityksestä siten, että siinä fokus on määritellä automaattisia hyväksyntätestejä ennen implementaation toteutusta, mitkä mittaavat sitä, täyttääkö toteutus vaatimukset käyttäjän näkökulmasta (Kairi 2019). ATDD ja BDD ovat idealtaan lähellä toisiaan, mutta ATDD keskittyy tarkemmin vaatimusten täyttämiseen, kun taas BDD keskittyy ominaisuuden haluttuun käyttäytymiseen (Unadkat 2019).

5 Esimerkkisovelluksen toteutus

5.1 Sovelluksen kuvaus

Jotta kehitettävä esimerkkisovellus olisi riittävän edustava web-sovelluksena, sen tulisi täyttää tietyt tekniset vaatimukset. Sovelluksen tulisi koostua verkkoselaimessa toimivasta selainpuolen sovelluksesta, jossa on graafinen käyttöliittymä, ja tietokantaa käyttävästä palvelinpuolen sovelluksesta, joka tarjoaa RESTful-rajapinnan selainpuolen sovellukselle tietojen hakemiseen, tallentamiseen ja muokkaamiseen. Sovelluksessa tulisi olla käyttäjien hallinta ja sisäänkirjautuminen, koska se on keskeinen toiminnallisuus monessa web-sovelluksessa. Lisäksi sovelluksen tulisi toimia Single Page App -periaatteen mukaisesti, eli sivulta toiselle siirtyminen ei aiheuta sivun uudelleen lataamista palvelimelta, vaan siirtyminen on sulavaa, ja palvelimelta haetaan vain tarvittava tieto JSON-muotoisena.

Esimerkkisovelluksen aiheeksi valikoitui tapahtumalistaus-sovellus, jossa sivuston vierailijat voivat selailla tapahtumia, tarkastella niiden kuvauksia sekä järjestää ja suodattaa niitä. Kirjautuneet käyttäjät voivat luoda omia tapahtumia, sekä muokata ja poistaa niitä.

Sovellukselle hahmottui seuraavat vaatimukset:

- Tapahtumia voi selata listalta, jossa näkyy otsikko, kategoria ja päivämäärä
- Tapahtumia voi järjestää uusimmasta vanhimpaan ja vanhimmasta uusimpaan
- Tapahtumia voi suodattaa otsikon ja kategorian mukaan
- Tapahtuman voi klikata auki, jolloin siitä näkyvät tarkemmat tiedot
- Sovellukseen voi rekisteröityä
- Rekisteröitynyt käyttäjä voi kirjautua sisään
- Kirjautunut käyttäjä voi luoda uuden tapahtuman
- Kirjautunut käyttäjä voi muokata lisäämäänsä tapahtumaa tai poistaa sen.

5.2 Sovelluksen kehityksessä hyödynnetyt keskeiset teknologiat

Selainpuolen sovellus toteutettiin React.js -käyttöliittymäkirjastolla, reititys React Router -kirjastolla, ja tilanhallinta sovellukselle Redux-kirjastolla. React-komponenttien kehittämiseen eristyksissä apuna oli Storybook. Palvelinpuolen sovelluksessa alustana oli Node.js, jonka päällä toimi web-palvelimen kehitystä helpottava Express-kirjasto ja tietokantayhteyksien toteuttamista avustava Mongoose-kirjasto. Testauksessa hyödynnettiin Jest, Testing Library, SuperTest ja Cypress -kirjastoja. Molemmat sovellukset kirjoitettiin TypeScript-ohjelmointikielellä. Seuraavassa on esitelty lyhyesti sovelluksen kehityksessä hyödynnettävät keskeiset työkalut ja teknologiat.

TypeScript

TypeScript on selaimissa natiivisti toimivan JavaScript-ohjelmointikielen laajennos, joka lisää siihen mm. staattisen tietotyyppien tarkistuksen ja uusimpien ECMAScript-standardien esittämiä ominaisuuksia. Se kääntyy tavalliseksi JavaScriptiksi, joten sitä voi käyttää kaikkien sellaisten sovellusten kehittämisessä, jotka pyörivät JavaScriptiä tukevissa ympäristöissä kuten selaimet tai Node.js. Monille JavaScriptillä kehitetyille kirjastoille on saatavilla tyyppimääritystiedostot, jotka mahdollistavat TypeScriptiä tukevissa ohjelmointiympäristöissä automaattiset koodintäydennykset, upotetut dokumentaatiot ja tyyppitarkistukset. (TypeScript n.d.)

React.js

React.js tai React on käyttöliittymäkirjasto uudelleenkäytettävien käyttöliittymäkomponenttien kehitykseen. Käyttöliittymä rakennetaan komponenteista koostamalla. Komponenteilla voi olla oma tila (state), ja ne voivat välittää tietoa alemman tason komponenteille Props-oliossa. React-komponenttien kehityksessä hyödynnetään tavallisesti HTML:a muistuttavaa JSX-syntaksia, joka on JavaScriptin laajennos. Reactin uusissa versioissa on mahdollista määrittää komponentit täysin JavaScriptin funktiosyntaksilla, ja komponenteille tuodaan esimerkiksi tilaan tai komponenttien elinkaareen (lifecycle event) liittyvät toiminnot Hooks-rajapinnan avulla. (React n.d.)

React Router

React Router -kirjasto mahdollistaa React-sovelluksissa reitityksen, jolloin selaimen osoitepalkissa näkyvän osoitepolun perusteella voidaan sovelluksessa näyttää tietty näkymä tai tiettyjä käyttöliittymäkomponentteja. Tämä mahdollistaa myös sen, että käyttäjä voi tallentaa tietyn polun kirjanmerkkeihinsä, tai mennä eteen tai taaksepäin navigaatiohistoriassa. Linkin klikkaamisen tai muun tapahtuman pohjalta on mahdollista muuttaa osoitepolkua ja sitä kautta muuttaa näkymää. Kirjasto tarjoaa sovelluksen käyttöön reitityksen mahdollistavat React-komponentit ja joitain hyödyllisiä Hookseja. (React Router n.d.)

Redux

Redux on Flux-arkkitehtuurin inspiroima tilanhallintakirjasto. Siinä on yksi yhtenäinen ja keskitetty tila (store), jolle voidaan tehdä pyyntöjä toiminnoilla (action). Toimintoihin vastaavat reducer-funktiot, jotka eivät suoraan muokkaa tilaa, vaan palauttavat toiminnon perusteella uuden version tilasta. (Redux n.d.)

Reduxin toimintaperiaate mahdollistaa tilan muutosten ennakoitavuuden, helpon virheiden jäljittämisen ja muutosten kumoamisen tai toistamisen. Se toimii periaatteessa minkä tahansa käyttöliittymäkerroksen kanssa, mutta on muodostunut suosituksi ratkaisuksi React-sovellusten tilanhallintaan. Redux tukee väliohjelmistoja (middleware), joilla voi laajentaa sen toimintoja. Näistä usein käytössä on jokin asynkroniset toiminnot mahdollistava väliohjelmisto. Redux Toolkit -kirjasto sujuvoittaa Reduxin käyttöä ja tarjoaa tiukemman raamin tilanhallinnan arkkitehtuurin toteuttamiselle. React Redux -kirjaston avulla React-komponenteista on helpompi lukea tilaa ja antaa käskyjä sille. (Mts.)

Storybook

Storybook on työkalu, jonka avulla voi mm. React-komponentteja kehittää erillään ilman että niitä tarvitsee koostaa osana komponenttipuita. Komponentille luodaan stories-päätteinen tiedosto, jossa on määritykset, esim. mitä propseja sille tulee. Niiden pohjalta Storybook koostaa kokoelman komponenteista, ja niitä pääsee tarkastelemaan verkkoselaimessa. Selaimen näkymän sivupalkissa on lista komponenteista,

ja niitä klikkaamalla komponentti piirtyy päänäkymään, josta sitä voi käsin testailla. (Storybook n.d.)

Node.js

Node.js on JavaScriptin tapahtumapohjainen ja asynkroninen suoritusympäristö, joka on rakennettu Chrome-verkkoselaimen V8 JavaScript -moottorin päälle. Se mahdollistaa komentorivisovellusten tai web-palvelinsovellusten kehittämisen JavaScriptillä. (Node.js n.d.)

Express

Express on sovelluskehys palvelinpuolen web-sovellusten ja web-rajapintojen kehittämiseen Node.js:n päälle. Perusperiaate siinä on määrittää reitit, mihin polkuihin on mahdollista vastata. Reiteille luodaan middleware-funktiot, jotka ottavat vastaan pyynnön, lähettävät jonkinlaisen vastauksen, ja käsittelevät poikkeukset. Middleware-funktioita voi myös ketjuttaa niin, että esimerkiksi pyyntöä käsitellään kaikille reiteille yhteisesti ennen spesifiä reitille tulevaa middleware-funktiota, tai vaikkapa poikkeustilanteiden käsittelyn voi ulkoistaa erilliselle middleware-funktiolle. (Express n.d.)

Mongoose

Mongoose on Node.js:n päällä toimiva MongoDB-tietokantakyselyjen toteuttamista abstraktoiva ODM-kirjasto. Sen avulla voi luoda skeemoja, jotka pakottavat MongoDB:hen tallennettavat dokumentit ennalta määritetyn muotoiseksi. Skeemalle voi määrittää validointisääntöjä, ja skeemasta luodun mallin metodeilla hoituvat dokumenttien luonti, haku ja muokkaaminen. (Mongoose n.d.)

Jest

Jest on JavaScript-koodin testaamiseen tarkoitettu testikehys, jota voi käyttää sekä selainpuolen että palvelinpuolen sovellusten testaukseen. Sen etuina on vähän konfigurointia vaativa käyttöönotto, rinnakkain suoritettavien testien nopeus, ja valekomponenttien (mock) helppo määrittäminen. (Jest n.d.)

Testing Library

Testing Library nimen alla on kokoelma kirjastoja, jotka tarjoavat eri alustoille käyttöliittymien testaukseen tarkoitettuja työkaluja. Niissä yhdistävä periaate on, että implementaation yksityiskohtien testaamisen sijaan testien tulisi kuvastaa sitä, miten käyttäjät todellisuudessa etsivät käyttöliittymästä elementtejä ja vuorovaikuttavat niiden kanssa, jotta testit lisäisivät luottamusta sovelluksen toimivuudesta. (Testing Library n.d.)

Cypress

Cypress on web-sovellusten käyttöliittymätason yksikkö, integraatio- ja järjestelmätestaukseen tarkoitettu testityökalu. Testit pyörivät verkkoselaimessa, mikä on lähimpänä web-pohjaisen asiakassovellusten todellista suoritusympäristöä. Testit kirjoitetaan ketjuttamalla komentoja, vähän kuin käyttäjä askel askeleelta suorittaisi verkkosivulla jonkin operaation. Cypress automaattisesti odottaa asynkronisten tapahtumien kuten verkkokutsujen vastausta ja sivujen latautumista, ennen kuin se jatkaa testin suorittamista. (Cypress n.d.)

Cypress pitää sisällään interaktiivisen testiajurin, jossa testien suorittaminen näkyy selainikkunassa vaihe vaiheelta, ja eri vaiheisiin voi palata tarkastelemaan verkkosivun tilaa kullakin hetkellä. Epäonnistuneista testeistä on myös mahdollista saada tallennettua automaattisesti ruutukaappauksen tai videokuvaa. (Mts.)

SuperTest

SuperTest on HTTP-kutsujen testausta helpottava apukirjasto. Se abstraktoi HTTP-kutsujen testauksen helposti luettavaan muotoon, ja sen avulla voi tehdä olettamia esimerkiksi HTTP-vastauksen statuskoodista ja palautetun sisällön tyypistä. (SuperTest n.d.)

5.3 Selainpuolen sovelluksen testivetoinen kehittäminen

Selainpuolen sovellus toteutettiin alhaalta ylöspäin periaatteella aloittamalla vaatimusten pohjalta johdettavien käyttöliittymäkomponenttien ohjelmoinnilla testivetoisesti. Testit luotiin React Testing Librarylla ja Jestillä. Niiden avulla yksinkertaisten

komponenttien, jotka ottavat vastaan Propsina arvoja ja tapahtumakäsittelijäfunktioita, testaaminen osoittautui melko suoraviivaiseksi. Komponentteja ei tarvinnut välttämättä koostaa sovelluksen juurikomponentissa, koska Storybookilla niitä pystyi kehittämään erillään.

Kun sovellukseen oli ajankohtaista lisätä tilaan ja palvelimella toimivaan RESTful-rajapintaan liittyviä kutsuja, toteutettiin ne toiminnallisuus kerrallaan Redux Toolkitilla, sen mukana tulevalla Redux Thunkilla ja Axios-kirjastolla. Redux Thunk mahdollistaa asynkroniset toiminnot, jolloin palvelimelle pyyntöjen tekeminen onnistuu Reduxin sisältä. Axios taas yksinkertaistaa palvelimelle tehtävien asynkronisten AJAX-kutsujen toteuttamista.

Redux-tilan siivujen (slice) testaaminen vaikutti aluksi hankalalta, mutta siihen löytyi kuitenkin luontevalta ja melko yksinkertaiselta tuntuva ratkaisu. Kun Axios-kirjastosta loi Jestillä valekomponentin, ja joka testissä alusti Store-olion configureStore-funktiolla, niin kaikkien toimintojen kehittäminen testivetoisesti onnistui.

Toinen hankala vaihe oli miettiä, kuinka Reduxin kanssa integroidut, sen Hookeja käyttävät React-komponentit olisivat järkevintä toteuttaa ja testata. Ratkaisu, johon päädyttiin, oli koostaa käyttöliittymäkomponentit sovelluksen sivuja kuvastavissa integraatiokomponenteissa, joissa kullakin sivulla näkyville komponenteille on välitetty Propseina niiden tarvitsemat asiat valittuna Redux-tilasta. Tapahtumakäsittelijä-funktioissa hoidettiin Redux-toimintojen lähettäminen (dispatch) ja navigointiin liittyvät uudelleenohjaukset (redirect).

Integraatiotestit olisi ollut mahdollista toteuttaa myös Jestillä, mutta lomakkeiden täyttäminen sillä vaikutti vähän työläältä, ja testityökaluksi integraatiotesteihin valikoitui Cypress. Se osoittautui soveltuvaksi työkaluksi testivetoiseen kehittämiseen, ja sen avulla oli suhteellisen yksinkertaista toteuttaa testit kaikille vaadituille ominaisuuksille siten, kuinka käyttäjä ne vaihe vaiheelta suorittaisi tai näkisi. Huono puoli sen käyttämisessä oli, että testien ajaminen sillä oli hitaampaa kuin Jestillä, vaikka verkkopyynnöt eivät menneet todelliselle palvelimelle vaan oli korvattu tyngillä, ja

esimerkiksi sovellukseen kirjautuminen testattiin vain kertaalleen käyttöliittymän kautta, kuten Cypressin ohjeistuksessa suositellaan (ks. Cypress n.d.).

Järjestelmätason testausta, eli lähinnä vain selainsovelluksesta verkon yli oikealle palvelimelle menevien pyyntöjen testausta ei nähty tarpeelliseksi automatisoida, vaan hoidettiin käsin.

5.3.1 React-komponentin yksikkötestaus

Tarkastellaan esimerkkinä sovelluksen React-komponentista ja sen testauksesta tapahtumalistan yksittäistä tapahtumaa edustavaa EventItem-komponenttia (ks. kuvio 5). Se on määritetty funktiomuodossa. Props-tyypissä on määritetty komponentin Props-oliossa syötteenä ottamat asiat, ja se sisältää tapahtuman tiedot Event-tyypin oliossa, tapahtuman klikkauksen ja muokkausnappulan tapahtumakäsittelijät, sekä käyttäjän User-tyypin oliossa. Käyttäjän tilalla voidaan myös antaa null-arvo, jos käyttäjää ei ole.

EventItem-komponentti näyttää tapahtumasta tietoja, ja jos käyttäjä on annettu ja tapahtuma on annetun käyttäjän luoma, niin tapahtuman yhteydessä näytetään muokkaa-painike. Muokkaa-nappulan painamisesta aktivoituvan onClick-tapahtuman leviäminen (propagation) ylempään div-elementtiin rekisteröityyn onClick-tapahtumaan on haluttu estää kutsumalla event-olion stopPropagation-metodia.

```
type Props = {
  event: Event
  onClick: (eventId: string) => void
  onEdit: (eventId: string) => void
  user: User | null
}

const EventItem = ({ event, onClick, onEdit, user }: Props) => {
  const { title, date, category, id } = event

  const handleEdit = (e: React.FormEvent<HTMLButtonElement>) => {
    e.stopPropagation()
    onEdit(event.id)
  }

  const editButton = (
    <button className="btn-small" onClick={handleEdit}>
      Edit
    </button>
  )

  return (
    <div className="card margin-bottom-small" onClick={() => onClick(id)}>
      <div className="card-body">
        <h4 className="card-title">{title}</h4>
        <h5 className="card-subtitle">{date}</h5>
        <p className="card-text">{category}</p>
        {user && user.id === event.user ? editButton : null}
      </div>
    </div>
  )
}
```

Kuvio 5. EventItem React-komponentti

EventItem-komponentin testitapauksessa, jossa testataan tilannetta, missä käyttäjä on annettu, määritellään ensin Props-oliona annettavat syötteet (ks. kuvio 6). Testi käyttää erikseen määritettyä testidataa tapahtuman ja käyttäjän osalta. Tapahtuma-käsitteljät on annettu Jestin mock-funktiona. React Testing Library -kirjaston render-metodilla komponentti lisätään selaimen DOM-ympäristöä simuloivaan JSDOM-ympäristöön. Metodin palauttamana saadaan myös destructuring-syntaksin avulla käyttöön getByText-kysely, jolla voidaan etsiä dokumentista elementtiä merkkijonon perusteella.

```
test('has edit button if users own event', () => {
  const [event] = events
  const handleClick = jest.fn()
  const handleEdit = jest.fn()
  const { getByText } = render(
    <EventItem
      event={event}
      onClick={handleClick}
      onEdit={handleEdit}
      user={user}
    />
  )

  fireEvent.click(getByText('edit', { exact: false }))

  expect(handleEdit).toHaveBeenCalledTimes(1)
  expect(handleEdit).toHaveBeenCalledWith(event.id)
  expect(handleClick).toHaveBeenCalledTimes(0)
})
```

Kuvio 6. EventItem-komponentin testitapaus, jossa käyttäjä on annettu

Testissä aktivoidaan React Testing Library -kirjaston tarjoaman FireEvent-olion click-metodin avulla parametrina annetun elementin onClick-tapahtuma. Tässä tapauksessa halutaan aktivoida muokkaa-tapahtuma, ja nappia vastaava elementti löydetään em. getByText-kyselyn avulla, minkä parametriksi annetaan napin tekstisisältö.

Jestin expect-funktiolla voidaan arvoja testata ketjuttamalla erilaisilla matcher-funktioilla. Tässä tapauksessa testaamme, että muokkaa-tapahtuman tapahtumakäsittelijää on kutsuttu kerran tapahtuman tunnisteen arvolla. Testataan myös, ettei ympäröivän div-elementin tapahtumakäsittelijää ole kutsuttu.

5.3.2 Redux-tilan yksikkötestaus

Seuraavaksi tarkastellaan kirjautuneen käyttäjän osuutta Redux-tilasta hoitavan kokonaisuuden toimintaa ja yksikkötestaamista (ks. kuvio 7). Esimerkkiä varten userSlice Redux-siivua on yksinkertaistettu sisältämään vain onnistuneen sisäänkirjautumisen tilan päivityksen. Ensin on määritetty State-tyyppi, jolla on kentät kirjautuneelle käyttäjälle ja kirjautumisen latauksen tilan osoittavalle muuttujalle. InitialState-muuttujaan tallennetaan Reducerin lähtökohtainen tilanne.

Redux Toolkitin CreateAsyncThunk-funktiolla luodaan asynkroninen toiminto (thunk action creator). Se ottaa parametrina Credentials-olion, joka sisältää käyttäjänimen ja salasanan. Se kutsuu userService-olion login-metodia, joka hoitaa RESTful-rajapintaan tehtävän HTTP-pyynnön. Operaation onnistuttua palautetaan tiedot käyttäjästä. Jos se epäonnistuu, käsitellään poikkeus.

```
type State = {
  user: User | null
  loading: Loading
}

const initialState: State = {
  user: null,
  loading: 'idle',
}

export const login = createAsyncThunk<
  User,
  Credentials,
  { rejectValue: Error }
>('user/login', async (credentials, thunkAPI) => {
  try {
    const data = await userService.login(credentials)
    return data
  } catch (e) {
    return thunkAPI.rejectWithValue(e.message)
  }
})

const userSlice = createSlice({
  name: 'user',
  initialState,
  reducers: {},
  extraReducers: (builder) => {
    builder.addCase(login.fulfilled, (state, { payload }) => {
      state.loading = 'success'
      state.user = payload
      toast(`Successfully logged in with ${payload.email}!`)
    })
  }
})
```

Kuvio 7. Sisäänkirjautumisen hoitava toiminnallisuus toteutettuna Redux Toolkitilla

Redux Toolkitin createSlice-funktiolla saadaan luotua toiminnot, jotka päivittävät itse tilaa. Sille on annettu lähtökohtainen tila initialState-muuttujassa. Reducers-kenttä on tässä esimerkissä tyhjä olio, mutta sovelluksessa siinä olisi uloskirjautumista ja käyttäjän hakemista selaimen paikallisesta varastosta koskevaa logiikkaa. ExtraReducers-kenttä on määritetty ”builder callback” -tyylillä, missä on tapaus sisäänkirjautu-

mistoiminnon onnistuneelle tilalle. Sille annetun takaisinkutsufunktion sisällä asetetaan tilan latautuminen onnistuneeksi ja käyttäjän tiedot user-kenttään, sekä lähetetään toast-funktiolla notifikaatio onnistuneesta kirjautumisesta. Redux Toolkitin ominaisuutena Reducer-funktioissa voidaan käyttää tilaa suoraa muuttavaa koodia, mutta todellisuudessa tilasta palautetaan uusi versio.

```
test('logs in user', async () => {
  const store = setup()

  const response = {
    data: user,
  }

  const endState = {
    user,
    loading: 'success',
  }

  mockedAxios.post.mockResolvedValue(response)

  await store.dispatch(login(credentials))

  expect(store.getState().user).toEqual(endState)
})
```

Kuvio 8. Testitapaus sisäänkirjautumiselle

Sisäänkirjautumista testaavassa testitapauksessa alustetaan ensin setup-funktiossa Redux store-olio, jossa sovelluksen tila sijaitsee (ks. kuvio 8). Response-muuttujassa on Jestillä sijaistetun axios-moduulin palauttama vastaus, ja endState-muuttujassa on haluttu lopputulos operaation jälkeen. MockResolvedValue-metodin avulla saadaan sijaistettu komponentti palauttamaan määritetyllä asynkronisella metodilla haluttu arvo. Varastoa pyydetään suorittamaan dispatch-metodilla asynkroninen login-toiminto, joka saa kirjautumistiedot parametrina. Lopuksi verrataan store-olion tilaa ja haluttua lopputulosta keskenään.

5.3.3 Sisäänkirjautumissivun integraatiotestaus

Lopuksi katsotaan sovelluksen sisäänkirjautumissivua edustavaa komponenttia ja sisäänkirjautumisen integraatiotestausta. LoginPage-komponentti koostaa sisäänkirjautumissivulla näytettävät elementit ja tarjoaa sivulla toimiville React-komponenteille syötteenä niiden tarvitsemat asiat (ks. kuvio 9). Komponentin alussa haetaan useTypedSelector-hookilla Redux-tilasta käyttäjän lataustilanne, useDispatch-hookilla viittaus dispatch-metodiin, sekä useHistory-hookilla viittaus React Routerin history-olioon.

```
const LoginPage = () => {
  const { loading } = useTypedSelector((state) => state.user)
  const dispatch = useDispatch()
  const history = useHistory()

  useEffect(() => {
    if (loading === 'success') {
      history.push('/')
    }
  }, [loading, history])

  const handleSubmit = (data: Credentials) => {
    dispatch(login(data))
  }

  return (
    <div className="col">
      <h2>Log in</h2>
      <LoginForm onSubmit={handleSubmit} />
      <LoadingIndicator loading={loading === 'pending'} />
      <div className="margin-top-large">
        <Link to="/signup">Sign Up instead...</Link>
      </div>
    </div>
  )
}
```

Kuvio 9. Sisäänkirjautumissivua edustava komponentti

UseEffect-hookin sisällä uudelleenohjataan käyttäjä sovelluksen etusivulle, kun kirjautuminen onnistuu. HandleSubmit-tapahtumakäsittelijässä kirjautumislomakkeen tiedot lähetetään login-toiminnolla Redux-tilan käsiteltäväksi. Komponentti palauttaa sivun otsikon, kirjautumislomake-komponentin, lataamiskuvakkeen ja linkin käyttäjätilin luomissivulle.

```
it('successfully logs in and logs out', () => {
  const {email, password} = credentials

  cy.server()
  cy.route({
    method: 'POST',
    url: '/api/login',
    response: user,
    delay: 100
  })

  cy.visit('/')
  cy.findByText('Login').click({force: true})
  cy.findByLabelText('email-input')
    .type(email)
  cy.findByLabelText('password-input')
    .type(password)
  cy.findByText('Submit')
    .click()
  cy.findByRole('progressbar')
  cy.findByText('Logged in', {exact: false})
  cy.findByText('Log out')
    .click({force: true})
  cy.findByText('Logged out', {exact: false})
  cy.findByText('Login')
})
```

Kuvio 10. Sisäänkirjautumissivun testitapaus

Onnistunutta sisäänkirjautumista testaava testitapaus kuten muutkin selainpuolen sovelluksen integraatiotestit toteutettiin Cypress-testityökalulla (ks. kuvio 10). Testissä näkyvä globaali muuttuja cy sisältää Cypressin komennot. Testin alussa server ja route -komennoilla on oikea palvelimelle tehtävä HTTP-kutsu korvattu Cypressin hoitamalla tyngällä.

Testi etenee suoraviivaisesti ylhäältä alas, kun Cypress käskyttää käyttöliittymää koodissa näkyvien komentojen mukaisesti. Se myös odottaa asynkronisten ja viiveellä tapahtuvien tapahtumien valmistumista automaattisesti, tarkemmin yrittää niitä niin kauan uudelleen, kunnes komento menee läpi tai asetettu aikakatkaisu saavutetaan. Jos aikakatkaisuun päädytään, testi epäonnistuu. Testissä näkyvät findByText ja findByLabelText -komennot ovat Cypress Testing Librarystä peräisin, ja niiden avulla voi sivulta etsiä elementtejä tekstisisällön ja saavutettavuuteen liittyvän ARIA Label -attribuutin perusteella.

5.3.4 Testikattavuus

Selainpuolen sovellukseen toteutettiin Jestillä yhteensä 44 yksikkötestiä kymmenelle eri komponentille (ks. kuvio 11). Raporttiin otettiin mukaan vain ne hakemistot ja tiedostot, joille yksikkötestausta toteutettiin, eli Components-hakemiston sisällä olevat käyttöliittymäkomponentit sekä Slices-hakemiston sisällä olevat Redux-siivut. Testattujen komponenttien testikattavuus oli hyvää tasoa, mutta mukaan oli kuitenkin päässyt testaamatonta koodia.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	95.86	92.31	93.33	95.76	
components/EventDetails	100	100	100	100	
EventDetails.tsx	100	100	100	100	
components/EventForm	100	100	100	100	
EventForm.tsx	100	100	100	100	
components/EventItem	100	100	100	100	
EventItem.tsx	100	100	100	100	
components/EventList	100	100	100	100	
EventList.tsx	100	100	100	100	
components/LoadingIndicator	100	100	100	100	
LoadingIndicator.tsx	100	100	100	100	
components/LoginForm	100	50	100	100	
LoginForm.tsx	100	50	100	100	33,47
components/Navigation	100	100	100	100	
Navigation.tsx	100	100	100	100	
components/SignupForm	100	100	100	100	
SignupForm.tsx	100	100	100	100	
slices	93.33	80	87.88	93.27	
eventsSlice.ts	95.45	100	86.36	95.38	94,97,100
userSlice.ts	89.74	0	90.91	89.74	55,56,57,58

```
Test Suites: 10 passed, 10 total
Tests:       44 passed, 44 total
Snapshots:   0 total
Time:        5.668s
Ran all test suites.
```

Kuvio 11. Selainpuolen sovelluksen yksikkötestien testikattavuus

Jos testikattavuusluvut lasketaan koko sovelluksen osalta, ei se anna täysin todellista kuvaa, koska integraatiotestit toteutettiin eri työkalulla, eikä ne näy Jestillä tuotetussa raportissa (ks. kuvio 12). Mukana on myös tiedostoja, joita ei lähtökohtaisesti haluttu testata, kuten Storybookia varten luodut tiedostot.

```
---------------------------------|----------|----------|----------|----------|-------------------|
File                             |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
---------------------------------|----------|----------|----------|----------|-------------------|
All files                        |    54.91 |    41.03 |    49.62 |    56.86 |                   |
---------------------------------|----------|----------|----------|----------|-------------------|

Test Suites: 10 passed, 10 total
Tests:       44 passed, 44 total
```

Kuvio 12. Selainpuolen sovelluksen Jestillä laskettu kokonaistestikattavuus

Cypressilla toteutettuja integraatiotestejä oli yhteensä 13 kappaletta viiteen eri kokonaisuuteen jaettuna, ja niiden kattavuus oli myös hyvää tasoa (ks. kuvio 13). Ensisijainen tarkoitus oli testata niillä App-komponenttiin sijoitettua reititystä sekä Pages-hakemiston kokonaisten sivujen integraatiosta vastaavien komponenttien logiikkaa. Kattavuusraportin mukaan tuli kuitenkin epäsuorasti testattua myös suuri osa käyttöliittymäkomponenttien ja Redux-siivujen sisällöistä, jolloin integraatiotesteissä oli huomattavaa päällekkäisyyttä yksikkötestien kanssa.

All files

84.81% Statements 296/349 **58.12%** Branches 68/117 **80%** Functions 96/120 **84.38%** Lines 281/333

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

File		Statements		Branches		Functions		Lines	
src		40.98%	25/61	23.08%	9/39	37.5%	9/24	36.84%	21/57
src/components/EventDetails		100%	2/2	100%	0/0	100%	1/1	100%	2/2
src/components/EventForm		100%	8/8	60%	6/10	100%	4/4	100%	7/7
src/components/EventItem		100%	8/8	100%	4/4	100%	3/3	100%	8/8
src/components/EventList		62.96%	17/27	66.67%	8/12	41.67%	5/12	64%	16/25
src/components/LoadingIndicator		100%	4/4	100%	2/2	100%	1/1	100%	4/4
src/components/LoginForm		100%	5/5	50%	2/4	100%	2/2	100%	5/5
src/components/Navigation		100%	2/2	100%	4/4	100%	1/1	100%	2/2
src/components/SignupForm		100%	8/8	50%	3/6	100%	3/3	100%	8/8
src/pages		97.26%	71/73	91.67%	22/24	100%	26/26	96.92%	63/65
src/services		97.62%	41/42	100%	2/2	100%	9/9	97.62%	41/42
src/slices		96.19%	101/105	60%	6/10	93.94%	31/33	96.15%	100/104
src/types		100%	1/1	100%	0/0	100%	0/0	100%	1/1
src/utils		100%	3/3	100%	0/0	100%	1/1	100%	3/3

Kuvio 13. Selainpuolen sovelluksen integratiotestien kattavuus

5.4 Palvelinpuolen sovelluksen testivetoinen kehittäminen

Palvelinpuolen sovelluksen tarjoamaa RESTful-rajapintaa kehitettiin testivetoisesti integraatiotestien ajamana. Ensin toteutettiin reitit tapahtumien hakemiseen, luomiseen ja muokkaamiseen, sitten käyttäjätilin luomiseen ja sisäänkirjautumiseen, ja lopuksi tapahtumien luomiselle ja muokkaamiselle lisättiin autentikaatio. Autentikaatiostrategia oli salasanapohjainen, ja se toimi JSON Web Tokeneiden avulla.

Testaus toteutettiin Jestillä ja SuperTestillä, ja niissä tehtiin testattaviin reitteihin pyyntöjä, ja tarkasteltiin lähetettyä vastausta ja tietokannan tilaa. Testitapauksissa onnistuneen operaation lisäksi varmennettiin myös pyyntöjen validaation ja autentikaatioon liittyvät seikat. Testeissä oli käytössä todellinen MongoDB-tietokanta, mutta MongoDB-Memory-Server-kirjaston avulla se pyöri käyttömuistissa, mikä nopeutti testien suorittamista. Ainakin näin pienessä sovelluksessa testien suoritusnopeus pysyi riittävänä.

5.4.1 RESTful-rajapinnan reitin integraatiotestaus

Esimerkkinä RESTful-rajapinnan reitin integraatiotestauksesta tarkastellaan tapahtuman luomiseen tarkoitettua reittiä (ks. kuvio 14). Aiemmin koodissa luotuun Router-tyypin eventRouter-olioon määritetään reitin juureen POST-pyyntöön vastaava käsittelijä. Pyynnön rungosta destrukturointi-syntaksilla otetaan kentät muuttujiin. Käyttäjä autentikoidaan authenticateUser-apufunktiolla. Jos se menee läpi, luodaan uusi Mongoosella määritetyn Event-skeeman mukainen tapahtuma. Se tallennetaan tietokantaan, ja tallennettu tapahtuma lähetetään vastauksena. Jos jotain menee pieleen, välitetään poikkeus seuraavalle, poikkeustilanteita käsittelevälle middleware-funktiolle.

```
eventRouter.post('/', async (req, res, next) => {
  const { title, date, description, category } = req.body

  try {
    const user = await authenticateUser(req)

    const event = new Event({
      title,
      description,
      date,
      category,
      user: user.id,
    })

    const savedEvent = await event.save()
    res.status(201).json(savedEvent)
  } catch (e) {
    next(e)
  }
})
```

Kuvio 14. Tapahtuman luontia vastaava reitti ja käsittelijäfunktio

Reittien integraatiotestejä varten jokaista testisarjaa kohden käynnistetään MongoDB-Memory-Server, ja jokaista testitapausta ennen alustetaan tietokanta haluttuun lähtötilanteeseen. Onnistunutta tapahtuman luontia testaavassa tapauksessa tehdään pyyntö SuperTestin avulla testattavaan reittiin (ks. kuvio 15). Pyyntöön on liitetty Authorization-ylätunnukseen pätevä JSON Web Token mukaan. Vastauksesta tarkastetaan, että statuskoodi on oikea, ja että vastauksen sisällön tyyppi on JSON-muotoinen. Olettamissa tarkastetaan, että tietokantaan tapahtumien listaan on lisätty uusi tapahtuma, ja että vastauksena saadun tapahtuman user-kentän sisältämä tunnus (id) on sama kuin käyttäjän, joka tapahtuman lisäsi.

```
test('create event', async () => {
  const [userInDb] = await getUsersInDb()
  const token = createToken(userInDb)
  const res = await req(app)
    .post('/api/events')
    .set({ Authorization: `bearer ${token}` })
    .send(newEvent)
    .expect(201)
    .expect('Content-Type', /application\/json/)

  const eventsInDbAfter = await getEventsInDb()
  const titles = eventsInDbAfter.map((event) => event.title)

  expect(eventsInDbAfter.length).toBe(events.length + 1)
  expect(titles).toContain(newEvent.title)
  expect(res.body.user).toEqual(userInDb.id)
})
```

Kuvio 15. Onnistuneen tapahtuman luomisen testitapaus

5.4.2 Testikattavuus

Palvelinpuolen sovelluksen reittejä testaavia integraatiotestejä oli yhteensä 17 kappaletta kolmeen kokonaisuuteen jaettuna, ja ne kattoivat valtaosan toiminnoista, vaikkakin yksittäisiä rivejä jäi testaamatta (ks. kuvio 16). Esimerkiksi EventRouter-komponentin riveillä 22–24 on koodia, joka käsittelee käyttäjän autentikaatiotilanteessa haaraa, jossa JSON Web Token on validi, mutta sen sisältämän tunnisteen mukaista käyttäjää ei löydy tietokannasta, ja tätä ei kateta missään testissä.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	94.04	82.35	86.49	94.15	
src	95	100	0	95	
app.ts	95	100	0	95	22
src/middleware	90.91	90	100	90.91	
errorHandler.ts	90.91	90	100	90.91	22
src/models	100	100	100	100	
event.ts	100	100	100	100	
user.ts	100	100	100	100	
src/routers	97.27	93.75	100	97.06	
eventRouter.ts	95	90	100	94.44	22-24
loginRouter.ts	100	100	100	100	
signupRouter.ts	100	100	100	100	
src/tests/testData	100	100	100	100	
event.testData.ts	100	100	100	100	
user.testData.ts	100	100	100	100	
src/tests/testUtils	95.24	100	87.5	95	
TestDbConnection.ts	90.91	100	75	90.91	15
hashPasswords.ts	100	100	100	100	
src/utils	74.07	33.33	50	76	
config.ts	100	100	100	100	
dbConnection.ts	63.64	25	33.33	63.64	11-21
mongooseConnectionOptions.ts	100	100	100	100	
requestLogger.ts	70	50	66.67	75	9-10

```
Test Suites: 3 passed, 3 total
Tests:       17 passed, 17 total
Snapshots:   0 total
Time:        10.606 s, estimated 15 s
Ran all test suites.
```

Kuvio 16. Palvelinpuolen sovelluksen testikattavuus

5.5 Havainnot ja päätelmät

Sovelluksen kehittäminen testivetoisesti oli menetelmän opetteluvaiheessa huomattavasti hitaampaa kuin jos testit olisi toteuttanut jälkeenpäin tai olisi turvauduttu vain käsin testaukseen. Sovelluksen kehittämiseen ilman testivetoista menetelmää oli arvioitu kuluvan noin 60 tuntia. Testivetoisen kehityksen menetelmää noudattamalla siihen kului kuitenkin kaikkinensa 147 tuntia. Kehitystä hidasti entisestään TypeScriptin käyttö, mistä tekijällä ei ollut TDD:n lisäksi juurikaan kokemusta. TDD-syklit olivat aluksi pitkiä, eikä vihreälle päästy kovin nopeasti. Menetelmän omaksumisessa auttaisikin varmasti, jos sen opettelun aloittaisi jollain vieläkin yksinkertaisemmalla kehitystyöllä, esimerkiksi jonkin apukirjaston tai komentorivisovelluksen ohjelmoinnilla.

Ennakko-odotus oli, ettei näin yksinkertaiselle sovellukselle olisi merkitseviä hyötyjä nähtävissä testivetoisen kehityksen menetelmän noudattamisesta. Selkeitä etuja pystyi kuitenkin toteamaan. Käsin testausta ei tullut tehtyä läheskään siinä määrin

kuin tavallisesti, koska sille ei ollut tarvetta automaattisten testien varmentaessa sen, että lisätyt toiminnot toimivat oletetusti, ja olemassa olevat eivät rikkoutuneet muutosten myötä. Vastaan tuli myös vain pari sellaista yllättävää vikaa, joita automaattiset testit eivät kattaneet. Käsin testaamiselle löytyi kuitenkin vielä sijansa esimerkiksi järjestelmätason testaamisessa ja sovelluksen ulkoasua ja tyylejä määritettäessä.

Mitä pidemmälle sovelluksen kehittäminen eteni, sitä enemmän automaattisista testeistä vaikutti olevan hyötyä, ja sitä työläämpää taas käsin testaaminen olisi ollut. Erityisesti integraatiotestit auttoivat suuresti sovelluksen rakenteen refaktoroinnissa, kun jokaisen muutoksen jälkeen pystyi nopeasti varmistamaan, että kaikki edelleen toimi. Muutokset koodin rakenteessa tai toteutuksen yksityiskohdissa ei odotetusti aiheuttanut muutoksia testeihin, mutta jos rajapintoihin tai ominaisuuksien toimintaperiaatteisiin tuli muutoksia, niin oli usein myös päivitettävä testit, ja tästä aiheutui lisätyötä.

Testivetoisen kehityksen noudattaminen vaikutti selkeästi myös sovelluksen rakenteeseen ainakin siten, että tehtyjen ratkaisujen tuli olla testattavia. Jos testausstrategiassa olisi suosittu suuremmalla painotuksella yksikkötestaamista integraatiotestien sijaan, rakenne olisi pitänyt olla vieläkin modulaarisempi ja abstraktiotasoja olisi tullut todennäköiseti lisää.

Kokemattomuus testivetoisen kehityksen noudattamisessa johti myös kehityksen edetessä kohtaamaan hankalia kysymyksiä ja suunnitteluvalintoja, mihin ei välttämättä löytynyt itsestäänselvää tai yksiselitteisesti parasta ratkaisua. Duplikaation poistaminen ohjelmakoodin ja testin kesken oli ajoittain vaikeaa. Esimerkiksi jos käyttöliittymässä näytetään jokin virheviesti, ja testissä todennetaan virheviestin näkyminen viestin sisällön perusteella, niin viestin muuttaminen rikkoisi testin, mutta viestin abstraktointi vakioksi aiheuttaisi lisää riippuvuuksia. Myös duplikaatio testien välillä pohditutti, esimerkiksi lomakkeen toiminnan testaaminen sekä yksikkötestissä omana komponenttinaan että integraatiotestissä, jossa tarkastellaan, että täytetyn lomakkeen tiedot voi lähettää, ja ne lisätään listalle.

Ylipäätään kokemattomuudesta johtuvana ongelmana oli usein se, että tiesi, miten jonkin ominaisuuden voisi toteuttaa, mutta ei ollut selvää, miten sille voi kirjoittaa testit, mitä asioita testeissä pitäisi varmentaa, miten ohjelmakoodista saisi testattavaa, ja mitä asioita tulisi korvata testikomponenteilla. Erityisesti sivuvaikutuksia aiheuttavaa koodia oli yleisesti vaikeampi testata.

Vastauksena tutkimuskysymyksiin testivetoisen kehityksen menetelmä soveltuu web-sovelluksen kehittämiseen, eikä sen noudattamisessa havaittu mitään ylitsepääsemättömiä rajoitteita esimerkkisovelluksen kehittämisessä valituilla teknologioilla. Hyödynnetyt testityökalut tukivat riittävällä tavalla testien toteuttamista ja suorittamista testivetoisen kehityksen syklin mukaisesti, kunhan testien suoritusnopeus saatiin pysymään riittävän nopeana. Kehittämällä web-sovellus testivetoisen kehityksen menetelmää noudattaen saavutettiin selkeitä laadullisia hyötyjä, mutta erityisesti menetelmän opetteluvaiheessa testattavan koodin toteuttamiseen sekä automaattisten testitapausten luomiseen ja ylläpitämiseen kului huomattavasti aikaa.

6 Pohdinta

Opinnäytetyön tavoitteena oli selvittää testivetoisen kehityksen menetelmän soveltuvuutta web-sovelluksen kehittämisessä. Tutkimus onnistui pääpiirteissään sille asetettujen tavoitteiden mukaisesti, ja sen tuloksena saatiin vastauksia asetettuihin tutkimuskysymyksiin. Kehitystutkimus valittuna tutkimusmenetelmänä soveltui ilmiön tutkimiseen.

Teoreettinen pohja ja käsitteiden määrittely kattoi tutkittavan ilmiökentän, ja lähdemateriaalina käytettiin aihepiirin perusteoksia sekä monipuolisesti alan asiantuntijoiden laatimia verkkojulkaisuja ja blogikirjoituksia ajantasaisen ja luotettavan tietopohjan kartuttamiseksi. Teoriaosuus olisi voinut olla tarkemmin rajattu käsittelemään pelkästään testivetoisen kehittämiseen ja web-sovelluksiin liittyviä käsitteitä, mutta

siinä olisi menetetty osa kontekstista, johon tieto asettuu. Myös testivetoisesta menetelmästä kertyneen tutkimustiedon esittely laajemmin olisi voinut olla hyödyllistä aineistosta tehtyjen havaintojen vertailtavuuden kannalta.

Kokemattomuus testivetoisen kehityksen menetelmän soveltamisessa on varmasti vaikuttanut aineistona toimineen esimerkkisovelluksen kehittämisessä tehtyihin ratkaisuihin. Samoin ohjelmointityöhön liittyvä luovuus ja ohjelmointiongelmien moninaiset ratkaisutavat tarkoittavat sitä, että samoilla vaatimuksilla toinen tekijä olisi voinut päätyä erilaisiin ratkaisuihin.

Johtopäätöksiin voi vaikuttaa myös se, että vertailun mahdollistavaa aineistoa eli esimerkkisovellusta toteutettuna ilman testivetoista menetelmää ei ollut käytettävissä. Myös esimerkkisovellus itsessään voi olla puutteellinen siinä mielessä, että se ei välttämättä vastaa kaikkia todellisille sovelluksille asetettuja vaatimuksia ja rajoitteita. Sovelluksen yksinkertaisuudella ja tiimikontekstin puuttumisella voi olla merkitsevä vaikutus tuloksiin. Käytössä ei myöskään ollut ajankäytön ja testikattavuuden lisäksi muita selkeitä mittareita, joilla tulosten vertailtavuutta ja merkitsevyyttä olisi voinut mahdollisesti parantaa.

Kehittämistutkimuksen luoteeseen kuuluu, että se ei välttämättä tarjoa kovin yleistettäviä ja laajasti hyödynnettäviä tuloksia. Tutkimuksen aineisto kuitenkin tarjoaa yhdenlaisen referenssitoteutuksen siitä, miten web-sovellus voidaan kehittää testivetoisesti.

Tutkimuksen pohjalta nousi esiin selkeitä jatkokehittämismahdollisuuksia. Web-sovelluksen kehittäminen testivetoisella menetelmällä todellisten käyttäjien tarpeiden pohjalta tiimiprojektina voisi tarjota luotettavampaa tietoa menetelmän käytännön eduista ja rajoitteista. Esimerkkisovelluksen testauksessa hyödynnettiin sekä yksikkö- että integraatiotestausta, ja näiden testitasojen tarkempi vertailu aiheen kontekstissa tarjoaisi myös aihetta jatkotutkimukselle.

Lähteet

Arcuri, D. 2018. Observations on the testing culture of Test Driven Development. Viitattu 25.3.2020. https://www.freecodecamp.org/news/8-observations-on-test-driven-development-a9b5144f868/.

Barber, D. 2012. Why Test-driven Development. Viitattu 25.3.2020. http://derekbarber.ca/blog/2012/03/27/why-test-driven-development/.

Beck, K. 2002. Test driven development: By example. Boston, Mass.: Addison-Wesley.

Cypress. N.d. Virallinen dokumentaatio. Viitattu 14.7.2020. https://www.cypress.io/.

Disadvantages of Test Driven Development. N. d. Viitattu 25.3.2020. https://stackoverflow.com/questions/64333/disadvantages-of-test-driven-development.

Dodds, K. C. 2019. Write tests. Not too many. Mostly integration. Viitattu 30.3.2020. https://kentcdodds.com/blog/write-tests.

Dredge, A. 2019. Test Driven Development (TDD) Research. Viitattu 14.4.2020. https://www.pluralsight.com/guides/test-driven-development-research.

Express. N.d. Virallinen dokumentaatio. Viitattu 14.7.2020. http://expressjs.com/.

Five things you should evaluate before proceeding into test automation. 2017. VALA Group Oy:n sivuilla oleva blogikirjoitus. Viitattu 6.4.2020. https://www.valagroup.com/fi/2017/07/five-things-you-should-evaluate-before-proceeding-test-automation/.

Fowler, M. 2012. TestPyramid. Viitattu 2.4.2020. https://martinfowler.com/bliki/TestPyramid.html.

Fucci, D. 2016. The role of process conformance and developers' skills in the context of test-driven development. Doctoral Dissertation. University of Oulu, Faculty of Information Technology and Electrical Engineering. Viitattu 8.4.2020. http://urn.fi/urn:isbn:9789526211657.

Gibb, R. 2016. What is a Web Application? Viitattu 3.4.2020. https://blog.stackpath.com/web-application/.

Jest. N.d. Virallinen dokumentaatio. Viitattu 14.7.2020. https://jestjs.io/.

Kairi, T. 2019. (Acceptance) Test-Driven Development: An Introduction. Viitattu 26.3.2020. https://www.eficode.com/blog/tdd-atdd.

Kananen, J. 2012. Kehittämistutkimus opinnäytetyönä: Kehittämistutkimuksen kirjoittamisen käytännön opas. Jyväskylä: Jyväskylän ammattikorkeakoulu.

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

Li, C. 2018. Why TDD is Bad (and How to Improve Your Process). Viitattu 25.3.2020. https://medium.com/@charleeli/why-tdd-is-bad-and-how-to-improve-your-process-d4b867274255.

Llopis, N. 2005. Stepping Through the Looking Glass: Test-Driven Game Development (Part 1). Blogikirjoitus. Viitattu 25.3.2020. http://gamesfromwithin.com/stepping-through-the-looking-glass-test-driven-game-development-part-1.

Lynch, W. 2019. What is the Problems of Waterfall Model? Viitattu 31.3.2020. https://medium.com/@warren2lynch/what-is-the-problems-of-waterfall-model-38de858f1058.

Mongoose. N.d. Virallinen dokumentaatio. Viitattu 14.7.2020. https://mongoosejs.com/.

Myers, G. J., Sandler, C. & Badgett, T. 2012. The art of software testing. 3rd ed. Hoboken, N.J.: John Wiley & Sons.

Node.js. N.d. Virallinen dokumentaatio. Viitattu 13.7.2020. https://nodejs.org/en/.

North, D. 2006. Introducing BDD. Viitattu 26.3.2020. https://dannorth.net/introducing-bdd/.

Ohjelmoinnin MOOC 2020: Johdatus ohjelmien testaamiseen. N.d. Helsingin yliopisto. Viitattu 14.4.2020. https://ohjelmointi-20.mooc.fi/osa-6/3-johdatus-ohjelmien-testaamiseen.

Paolini-Subramanya, M. 2018. Mocks? Stubs? Or Shims? Viitattu 30.3.2020. https://hackernoon.com/mocks-stubs-or-shims-f22164422020.

Pittet, S. N.d. The different types of software testing. Viitattu 30.3.2020. https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing.

React. N.d. Virallinen dokumentaatio. Viitattu 11.5.2020. https://reactjs.org/.

React Router. N.d. Virallinen dokumentaatio. Viitattu 13.7.2020. https://reactrouter.com/.

Redux. N.d. Virallinen dokumentaatio. Viitattu 13.7.2020. https://redux.js.org/.

Sami, M. 2012. Software Development Life Cycle Models and Methodologies. Viitattu 15.4.2020. https://melsatar.blog/2012/03/15/software-development-life-cycle-models-and-methodologies/.

Sapegin, A. 2019. Modern React testing, part 1: best practices. Viitattu 30.3.2020. https://blog.sapegin.me/all/react-testing-1-best-practices/.

Singh, N. 2017. Test-driven development might seem like twice the work — but you should do it anyway. Viitattu 25.3.2020. https://www.freecodecamp.org/news/isnt-tdd-test-driven-development-twice-the-work-why-should-you-care-4ddcabeb3df9/.

Subramaniam, V. 2016. Test-driving JavaScript applications: Rapid, confident, maintainable code. Raleigh, NC: The Pragmatic Bookshelf.

Storybook. N.d. Virallinen dokumentaatio. Viitattu 17.7.2020. https://storybook.js.org/.

SuperTest. N.d. Virallinen dokumentaatio. Viitattu 14.7.2020. https://github.com/visionmedia/supertest.

Testing Library. N.d. Virallinen dokumentaatio. Viitattu 14.7.2020. https://testing-library.com/.

The Cost of Poor Software Quality in the US: A 2018 Report. 2018. Consortium for IT Software Quality (CISQ). Viitattu 7.4.2020. https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf.

Timotic, M. 2018. Web Application Testing: Step by Step Process to make it Right. Viitattu 3.4.2020. https://tms-outsource.com/blog/posts/web-application-testing/.

TypeScript. N.d. Virallinen dokumentaatio. Viitattu 13.7.2020. https://www.typescriptlang.org/index.html.

Unadkat, J. 2019. BDD vs TDD vs ATDD: Key Differences. Viitattu 26.3.2020. https://www.browserstack.com/guide/tdd-vs-bdd-vs-atdd.

Vogels, R. N.d. A 6-Step Guide to Web Application Testing. Viitattu 3.4.2020. https://usersnap.com/blog/web-application-testing/.

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

Öztürk, M. 2020. Software Testing Process and Levels of Testing. Viitattu 30.3.2020. https://medium.com/swlh/software-testing-process-and-levels-of-testing-4274904ce655.