Opinnäytetyö (AMK)

Tieto- ja viestintätekniikka

2021

Eemeli Ranta

AUTOMAATIOTESTAUKSEN KEHITYS DIGIHELPPARI-SOVELLUKSESSA

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tieto- ja viestintätekniikka

2021 | 33 sivua

Eemeli Ranta

AUTOMAATIOTESTAUKSEN KEHITYS DIGIHELPPARI-SOVELLUKSESSA

Testaus on olennainen osa ohjelmiston kehitystä. Testauksen tarkoituksena on varmistaa, että ohjelmisto toimii määriteltyjen vaatimusten mukaisesti. Ohjelmistotestauksen avulla voidaan löytää ja korjata virheitä ennen kun ne vaikuttavat ohjelmiston loppukäyttäjiin.

Opinnäytetyön tavoite on edistää toimeksiantajan laadunvarmistusprosessia ja kehittää Digihelppari-sovellukseen mahdollisimman kattava automaatiotestiratkaisu. Työn päämääränä on luoda sovelluksen tärkeimmille ominaisuuksille automaatiotestitapauksia virheiden vähentämiseksi ja riskien minimoimiseksi.

Opinnäytetyössä tutustutaan yleisesti ohjelmistotestaukseen, sen menetelmiin ja hyötyihin. Lisäksi perehdytään tarkemmin automaatiotestaukseen ja esitellään automaatiotestien suunnittelemista sekä yksikkö- ja integrointitestien käyttöönottamista Digihelppari-projektissa.

Lopputuloksena Digihelppari-sovelluksen luotettavuutta ja käytettävyyttä parannettiin löytämällä ja korjaamalla virheitä hyödyntäen automaatiotestitapauksia. Työn kautta kehitettiin Digihelpparille kattava automaatiotestausratkaisu mahdollisen regression löytämiseen tulevaisuudessa.

ASIASANAT:

automaatiotestaus, ohjelmistotestaus, rajapinta, Python, Django

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information and Communications Technology

2021 | 33 pages

Eemeli Ranta

DEVELOPMENT OF AUTOMATION TESTING IN DIGIHELPPARI APPLICATION

Testing is an integral part of software development. The purpose of testing is to ensure that the software operates within the specified requirements. Software testing can be used to find and correct errors before they affect the end users of the software.

The purpose of the thesis is to improve the client's quality assurance process and to develop a comprehensive automation testing solution for Digihelppari application. The objective of this thesis is to create automation test cases for the most important features of the application to reduce errors and minimize risks.

This thesis introduces software testing in general, its methods and benefits, while keeping the main focus on automation testing. In addition, this thesis describes designing automation tests and developing unit and integration tests in Digihelppari.

As a result, the reliability and usability of Digihelppari was improved by finding and correcting errors utilizing automation test cases. Through this thesis, a comprehensive automation testing solution was developed for Digihelppari to find possible regression in the future.

KEYWORDS:

automation testing, software testing, API, Python, Django

SISÄLTÖ

KUVAT

KÄYTETYT LYHENTEET JA SANASTO

Git	Versionhallintajärjestelmä, jota käytetään muutosten seuraamiseen tiedostoissa.
HTTP	Protokolla, jota käytetään tiedonsiirtoon. Lyhenne sanoista Hypertext Transfer Protocol.
JSON	Muoto tallentaa ja siirtää tietoja. Lyhenne sanoista JavaScript Object Notation.
Komponentti	Itsenäinen ja uudelleenkäytettävä ohjelmiston yksikkö, josta voidaan rakentaa suurempia kokonaisuuksia.
Malli	Komponentti, joka on vastuussa järjestelmän datan hallinnasta.
Manager	Rajapinta sovelluksen mallin ja tietokannan välillä.
Moduuli	Itsenäinen ohjelmiston osa, josta voidaan koota erilaisia kokonaisuuksia. Moduuli voi sisältää useita eri komponentteja.
Ohjelmistokehys	Apuväline, jonka tarkoituksena on nopeuttaa ohjelmistojen kehitystä tarjoamalla valmiita ohjelmiston osia, jotta kehittäjän ei tarvitse kehittää niitä uudelleen.
Pull request	Mekanismi pyytää valmiin koodin yhdistämistä kehittäjän omasta haarasta päähaaraan.
Refaktoroida	Koodin muuttaminen tai parantaminen siten, että sen ulkoinen toiminnollisuus säilyy ennallaan.
Verkkokehys	Ohjelmistokehys, joka on suunniteltu tukemaan verkkosovellusten kehitystä.
XML	Merkintäkielen standardi, jota käytetään esimerkiksi tiedonvälitykseen ja tallentamiseen. Lyhenne sanoista Extensible Markup Language.

1 JOHDANTO

Jokainen ohjelmistokehitystiimi testaa tuotettaan, mutta toimitetuissa ohjelmistoissa on aina puutteita. Monet toimijat eivät kiinnitä paljon huomiota testaukseen, vaikka se on välttämätöntä laadukkaan tuotteen toimittamiseksi. Testaajat pyrkivät löytämään kaikki virheet ennen tuotteen julkaisua, mutta jopa parhaalla manuaalisella testausprosessilla virheet silti päätyvät ohjelmistoon tai ilmestyvät siihen jälkikäteen. Koska manuaalinen testaus vie paljon resursseja rahallisesti ja ajallisesti, automaatiotestien suosiminen edesauttaa tehokkuuden ja luotettavuuden parantamisessa, sillä niiden avulla testaus voidaan suorittaa nopeammin, kattavammin ja tarkemmin. Automaatiotestien avulla viat voidaan havaita välittömästi, jolloin niiden korjaus on yksinkertaisempaa ja halvempaa. [1]

Digihelppari on tarkoitettu tukemaan erityisesti nuorien neuropsykiatrisia häiriöitä omaavien henkilöiden itsenäisyyttä ja arjen hallintaa [2]. Tässä opinnäytetyössä suunnitellaan ja kehitetään automaatiotestejä Digihelppari-sovellukseen. Sovellus koostuu mobiilisovelluksesta ja taustajärjestelmästä. Tässä opinnäytetyössä keskitytään ainoastaan Digihelpparin taustajärjestelmään ja mobiilisovelluksen testaus sekä manuaaliset ja ei-toiminnalliset testit ovat rajattu tämän opinnäytetyön ulkopuolelle. Opinnäytetyön tarkoituksena on mahdollisimman monipuolisen automaatiotestauksen avulla vähentää taustajärjestelmän virheellisen toiminnan riskiä, löytää ja korjata virheitä rajapinnasta ja järjestelmän mallien toiminnollisuuksista testitapausten avulla.

Opinnäytetyössä käydään ensin läpi ohjelmistotestauksen yleistä teoriaa, jota voidaan soveltaa mihin tahansa ohjelmointikieleen. Luvussa 2 esitellään ohjelmistotestauksen hyötyjä, tasoja ja erilaisia testausmetodeita. Luvussa 3 esitellään lyhyesti REST-rajapinta arkkitehtuuria ja tärkeimmät opinnäytetyön soveltavassa osassa käytetyt teknologiat. Opinnäytetyön soveltavassa luvussa esitellään automaatiotestauksen käyttöönottamista Digihelppari-sovelluksen taustajärjestelmässä. Automaatiotestauksen käyttöönotto aloitetaan testien suunnittelusta ja testidatan valmistelemisesta, minkä jälkeen kehitetään yksikkö- ja integrointitestejä.

2 OHJELMISTOTESTAUS

2.1 Yleistä

Ohjelmistotestaus on tärkeä osa ohjelmistokehitysprosessia. Sen päämäärä ei ole etsiä kaikkia virheitä tai tehdä ohjelmistosta täydellistä. Kaikkia ohjelmiston virheitä tai vikoja on mahdotonta löytää, sillä testit voivat todistaa vain virheiden olemassaolon, ei niiden puuttumista. Testauksen ideana ei ole myöskään löytää kaikkea, mikä voisi mennä pieleen tai varmistaa vastaako ohjelmisto määrittelyä, sillä molemmat ovat mahdottomia toteuttaa. Sen sijaan ohjelmistotestauksen painopiste on riskin vähentäminen ennakoivasti etsimällä ja korjaamalla ongelmia, jotka vaikuttaisivat negatiivisesti ohjelmiston käyttäjään. [3]

Tärkein ohjelmistotestauksen hyöty on, että viat ja ongelmat voidaan ratkaista heti, ennen tuotteen julkaisua [4], milloin niiden korjaus on halvempaa ja helpompaa [5]. Vikojen löytäminen varhain auttaa kehittäjiä kirjoittamaan parempaa koodia, sillä saman virheen toistaminen uudestaan on epätodennäköisempää [5].

Testaus tuo itsessään testattavalle ohjelmistolle lisäarvoa ja luotettavuutta. Julkaisemalla hyvin testattuja ohjelmistoja rakennetaan vahvaa mainetta ohjelmiston kehittäjälle. Ohjelmiston loppukäyttäjät osaavat arvostaa korkealaatuisia ohjelmistoja joka johtaa vanhojen käyttäjien säilyttämiseen ja suositusten kautta uusien saamiseen. [4]

Koska kaiken kattava testaus on mahdotonta, riskin minimoimiseksi testaajan tulee keskittyä niihin alueisiin ohjelmistosta, millä on todennäköisimmin suurin vaikutus käyttäjän kokemukseen, ja valita mitkä alueen toiminnollisuudet testataan. Kun testatut toiminnollisuudet poikkeaa halutusta, viat kirjataan ja priorisoidaan vakavuuden perusteella korjattaviksi. [3]

2.2 Manuaali- ja automaatiotestaus

Ohjelmistotestitapaukset voidaan suorittaa joko manuaalisesti tai automatisoidusti.

Manuaalisessa testauksessa henkilö suorittaa testit ilman automaatiotyökalujen apua. Testaaja toimii ohjelmistossa loppukäyttäjän tavoin löytääkseen mahdollisimman paljon virheitä. Vaikka prosessi on aikaa vievää ja mahdollisten inhimillisten virheiden takia epätarkempaa, on sillä oma paikkansa ohjelmiston testauksessa, sillä kaiken testaaminen automatisoidusti voi olla mahdotonta tai kannattamatonta. [6]

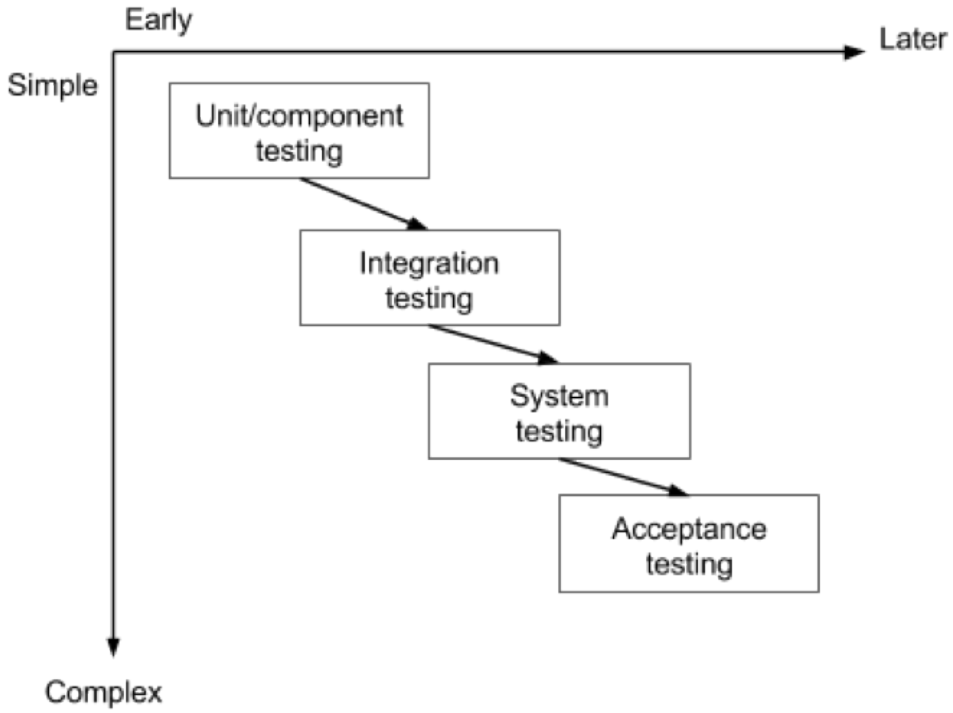
Tiettyyn pisteeseen asti on riittävää, että ohjelmisto testataan manuaalisesti niin, että ”se näyttää toimivan”. Laajemmissa ohjelmistoissa komponenttien välillä voi olla useita monimutkaisia vuorovaikutuksia, joiden testaaminen manuaalisesti muutoksen jälkeen ei ole tehokasta ajankäyttöä. Erityisesti, kun automaattiset testit voivat tehdä saman sekunneissa ja avustaa löytämään koodista virheen, joka aiheuttaa odottamatonta käyttäytymistä. [7]

Automaatiotestaus mahdollistaa toistuvien testien suorittamisen ilman manuaalista testaajaa [8]. Automaatiotesteissä suoritetaan ennalta määriteltyjä ohjelmiston toiminnallisuutta varmistavia testitapauksia. Tietokoneen suorittamana huolella suunnitellut ja kirjoitetut automatisoidut testitapaukset ovat erittäin luotettavia ja nopeita. Automaatiotestauksen tarkoitus on yksinkertaistaa testausprosessia ja tehdä siitä tehokkaampaa. Se nostaa tuottavuutta ja vähentää testauksen vaatimaa aikaa. [6]

Vaikka testitapaukset suoritetaankin automaattisesti, niiden kirjoittaminen ja automaatiotestausympäristön pystyttäminen vaatii jonkin verran manuaalista työtä [8], mikä luo korkeat alkukustannukset [6]. Automatisoidut testitapaukset auttavat kuitenkin pitkällä tähtäimellä säästämään ylläpidossa vähentämällä korjaustarvetta [6].

2.3 Testaustasot

Erityyppisten testien käyttöönotossa on olemassa looginen järjestys, jota tulisi noudattaa. Ohjelmistotestauksessa edetään yksinkertaisemmista komponenteista monimutkaisempiin ohjelmiston perusteellisen ja tehokkaan testauksen takaamiseksi. [9] (Kuva 1)



Kuva 1. Toiminnallisten testaustasojen järjestys ohjelmistokehityksessä. [9]

2.3.1 Yksikkötestaus

Yksikkötestauksessa testataan pienintä mahdollista ohjelmiston yksikköä eristyksissä kaikesta muusta koodista. Yksikkötestit selventävät mitä koodin on tarkoitus tehdä erittäin perusteellisella tasolla ja varmistavat että koodi toimii oikein. Yksikkötestit ovat lyhyitä koodin pätkiä, jotka testaavat muuta koodia ja ne kirjoitetaan usein samalla ohjelmointikielellä kuin testattava lähdekoodi. Kaikissa testeissä tulee olla jonkinlainen tarkistus eli väite, joka määrittää testin onnistumisen tai epäonnistumisen. Yksikkötestejä luodaan usein samaan aikaan kun ohjelmistoon lisätään uutta koodia, jolloin saadaan selville toimiiko uusi koodi oikein. [10]

2.3.2 Integrointitestaus

Kun erilliset komponentit on luotu ja yksikkötestattu hyväksytysti eristyksissä toisistaan, voidaan testata monen komponentin toimintaa yhdessä yhtenä integroituna moduulina. Tätä kutsutaan integrointitestaukseksi. Integrointitestaus keskittyy pääasiassa rajapintoihin ja datan kulkuun moduulien ja komponenttien välillä. Integrointitestauksen tarkoituksena on paljastaa viat eri komponenttien liitännöissä ja moduulien vuorovaikutuksessa. [11]

Erilliset moduulit integroidaan yksi kerrallaan oikeanlaisen toteutuksen ja yhteisen toiminnan varmistamiseksi. Virheet pyritään löytämään mahdollisimman aikaisessa vaiheessa vaivan ja kustannuksien säästämiseksi. Integrointitestaus ei tapahdu ohjelmistokehitysjakson lopussa, vaan sitä suoritetaan samanaikaisesti kehityksen kanssa. Tämän takia kaikki moduulit eivät välttämättä ole valmiina testattaviksi, ja ne voidaan testata sitä mukaa kun moduuleja saadaan valmiiksi. Integrointitestaus voidaan toteuttaa Big bang, Bottom-up tai Top-down-menetelmällä. [12]

Toisin kuin muissa menetelmissä, Big bang-menetelmässä integroidaan kaikki moduulit yhdellä kertaa osaksi järjestelmää toiminnan varmistamiseksi. Menetelmä vaatii kaikkien moduulien olevan valmiita testattavaksi, minkä takia se toimii paremmin pienemmissä järjestelmissä, jossa mahdollisten vikojen määrä on pienempi. Monimutkaisemmissa järjestelmissä virheiden löytäminen on hankalaa sillä kaikki moduulit testataan kerralla. [12]

Bottom-up-menetelmässä aloitetaan sovelluksen alhaisimman tason yksiköistä edeten korkeamman tason moduuleihin. Alemmista komponenteista aloittaminen tarkoittaa sitä, että korkeamman tason moduulit eivät ole vielä valmiita, minkä takia niitä joudutaan simuloimaan. Koska moduulit integroidaan yksi kerrallaan, virheiden löytäminen alemman tason moduuleista on nopeampaa, mutta korkeamman tason virheet voidaan löytää vasta myöhään testausprosessissa. [12]

Top-down-menetelmässä aloitetaan korkeamman tason moduuleista siirtyen yksitellen alemman tason komponentteihin. Kun alemman tason komponentti ei ole vielä valmis, sen toimintaa voidaan simuloida, jotta moduuli saadaan testattua. [12]

2.3.3 Järjestelmätestaus

Järjestelmätestaus on integrointitestauksen jälkeen suoritettava testaustaso, jossa testataan valmiin järjestelmän toiminnalliset ja ei-toiminnalliset osa-alueet yhtenä kokonaisuutena mustalaatikkotesteillä. Järjestelmätestausta tehdään jotta voidaan todeta, että järjestelmä täyttää sille ennalta määritellyt vaatimukset. Ohjelmiston vaatimukset ja odotukset tulee olla selkeitä ja dokumentoituja ennen järjestelmätestauksen aloittamista. Ohjelmiston testaajan on ymmärrettävä ohjelmiston reaaliaikainen käyttö. Testaajalla tulee olla selkeä kuva miten ohjelmistoa tullaan käyttämään ja minkälaisia ongelmia voidaan kohdata ohjelmistoa käytättäessä. Järjestelmätestauksen suorittaminen tuotantoympäristön kaltaisessa testausympäristössä voi auttaa ymmärtämään ohjelmistoa loppukäyttäjän näkökulmasta ja näin estää esiintyvät viat ennen niiden päätymistä tuotantoon. [13]

2.3.4 Hyväksymistestaus

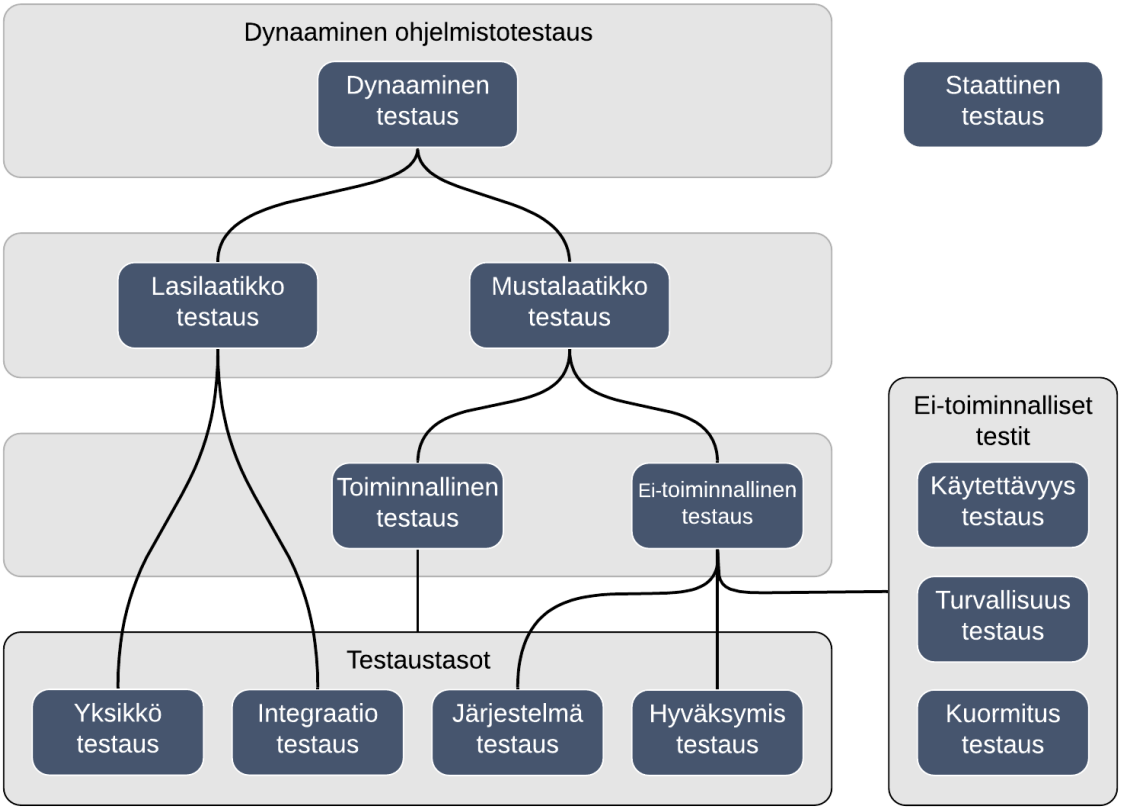
Kun kaikki muut testaustasot ovat läpäistyjä, aloitetaan hyväksymistestaus, jonka tarkoituksena on varmistaa onko ohjelmisto valmis toimitettavaksi. Tavallisesti ohjelmistoa kehittänyt organisaatio suorittaa kaikki aikaisemmat testaustasot, mutta hyväksymistestauksen suorittaa asiakas, joka arvioi täyttääkö ohjelmisto määritellyt vaatimukset. Hyväksymistestaus sisältää usein myös ohjelmiston lakisääteisten ja sopimuksellisien vaatimusten tarkistamista. Vaikka kaikki aikaisemmat toiminnalliset testit ovat hyväksytysti läpäistynä, ohjelmisto voi silti tulla hylätyksi, jos se ei täytä asiakkaan vaatimuksia. [14]

Ketterässä ohjelmistokehityksessä hyväksymistestaus suoritetaan usein joka kehitysjakson lopussa osana sen tuotosten esittelyä asiakkaalle [9].

2.4 Testausmenetelmät

Dynaamisten testausmenetelmien painopisteenä on ohjelmiston oikein toimimisen varmistaminen dynaamisilla muuttujilla, mikä voidaan toteuttaa vain ohjelmiston koodin suorittamisella. Näin voidaan tehokkaasti löytää virheitä, jotka ovat liian hankalia löytää ilman koodin ajamista. [15]

Dynaaminen testaus jaetaan usein lasilaatikko- ja mustalaatikkotestaukseen (Kuva 2), mutta näiden lisäksi on olemassa harmaalaatikkotestausmenetelmä, mikä tarjoaa sekä lasilaatikko- ja mustalaatikkotestauksen yhdistetyt edut. [16]



Kuva 2. Dynaaminen ohjelmistotestausmetodien suhteet.

2.4.1 Mustalaatikkotestaus

Mustalaatikkotestauksessa testattavaa ohjelmaa pidetään yhtenä kokonaisuutena ja sen sisäinen toiminta ja rakenne jätetään huomioimatta. Sen sijaan keskitytään tilanteisiin, missä ohjelma ei käyttäydy vaatimusten ja määrityksien mukaisesti vertailemalla annettua syötettä ja saatua ulostuloa ohjelmiston käyttäjän näkökulmasta. Käyttämällä mustalaatikkomenetelmää koodin sisäisestä rakenteesta ei tarvitse välittää, sillä vain ohjelmiston käyttäytymistä tarkastellaan, minkä takia menetelmä soveltuu hyvin toiminnallisten sekä ei-toiminnallisten vaatimusten testaamiseen (Kuva 2). [17]

2.4.2 Lasilaatikkotestaus

Lasilaatikkotestauksessa ohjelmiston sisäistä rakennetta, suunnittelua ja koodia testataan oikeanlaisen tiedonkulun varmistamiseksi sekä suunnittelun, käytettävyyden ja turvallisuuden parantamiseksi. Yksi lasilaatikkotestauksen perustavoitteista on tarkistella sovelluksen toiminnan kulkua vertaamalla ennalta määriteltyjä syötteitä odotettuihin ulostuloihin virheiden löytämiseksi. [18]

Jotta sovellusta voidaan testata lasilaatikkomenetelmällä, tulee testaajan ymmärtää sen lähdekoodi läpikotaisesti ja olla asiantunteva testattavan sovelluksen käytetyissä ohjelmointikielissä ja turvallisista koodikäytännöistä. Turvallisuus on yksi ohjelmistotestauksen ensisijaisista tavoitteista, ja testaajan on kyettävä löytämään tietoturvaongelmia ja estämään hyökkäyksiä haitallisilta käyttäjiltä. Sovelluksen tiedonkulkua ja rakennetta testataan kirjoittamalla lähdekoodia testaavaa koodia jokaiselle sovelluksen prosessille tai prosessisarjalle. Lasilaatikkotestit toteutetaan yleensä kehittäjän toimesta, sillä niiden kehittäminen vaatii koodin läheisen tuntemuksen. Yksikkö- ja integrointitestit ovat tyypillisesti toteutettu hyödyntäen lasilaatikkomenetelmää (Kuva 2). [18]

2.4.3 Harmaalaatikkotestaus

Yhdistämällä mustalaatikkotestauksen yksinkertaisen tekniikan ja lasilaatikkotestauksen tarkan testien kohdistuksen, harmaalaatikkotestaus tarjoaa etuja molemmista tekniikoista. Menetelmässä testattavan koodin toiminta on vain osittain tiedossa, mitä hyödynnetään testitapauksien suunnitellessa. Harmaalaatikkotestaus on hyödyllinen erityisesti integrointitestauksessa. Testitapaukset voidaan kohdistaa testaamaan korkean riskin alueita, mutta tapaukset suoritetaan kuitenkin käyttäjän näkökulmasta, mikä tekee testitapauksien kehittämisestä halvempaa. [16]

2.4.4 Toiminnallisuustestaus

Toiminnallisuustestauksessa varmistetaan ohjelmiston toimivuus annettujen vaatimusten mukaisesti. Testaus voidaan suorittaa kaikilla testaustasoilla (Kuva 2). Toiminnallisuustestauksen toteuttamista varten tarvitaan ohjelmiston hyväksyttävän toiminnan rajat määrittelevä dokumentti, jota käytetään §oppaana ohjelmistoa

testatessa. Dokumentin perusteella valmistellaan joukko testitapauksia ja testidataa. Toiminnallisuustestaus suoritetaan mustalaatikkomenetelmällä (Kuva 2) antamalla ohjelmistolle syöte ja varmistamalla, että saatu tulos vastaa ohjelmiston vaatimuksia oikeanlaisen toiminnan varmistamiseksi keskittymättä siihen, miten ohjelmisto päätyy saatuun lopputulokseen. [19]

2.4.5 Ei-toiminnallinen testaus

Ei-toiminnallisessa testauksessa testataan ohjelmiston ei-toiminnallisia ominaisuuksia, jotka eivät kuulu toiminnallisuustestaukseen kuten esimerkiksi ohjelmiston käytettävyys, nopeus, vakaus ja turvallisuus erilaisissa olosuhteissa (Kuva 2). Ei-toiminnallisten testien suorittaminen voidaan aloittaa vasta kun suurin osa toiminnallisista testeistä on suoritettu. Tässä testaustyypissä tarkastellaan käyttäytyykö ohjelmisto asiakkaan tai käyttäjän odotusten mukaisesti. Kun sovellus toimii käyttäjän odotusten mukaisesti, sujuvasti ja tehokkaasti erilaissa olosuhteissa, voidaan se todeta luotettavaksi. [20]

Ohjelmiston testausstrategia on suunniteltava oikein asianmukaisen testauksen varmistamiseksi. On olemassa erilaisia työkaluja, jotka helpottavat ei-toiminnallisten testien suorittamisessa ja automatisoinnissa. [20]

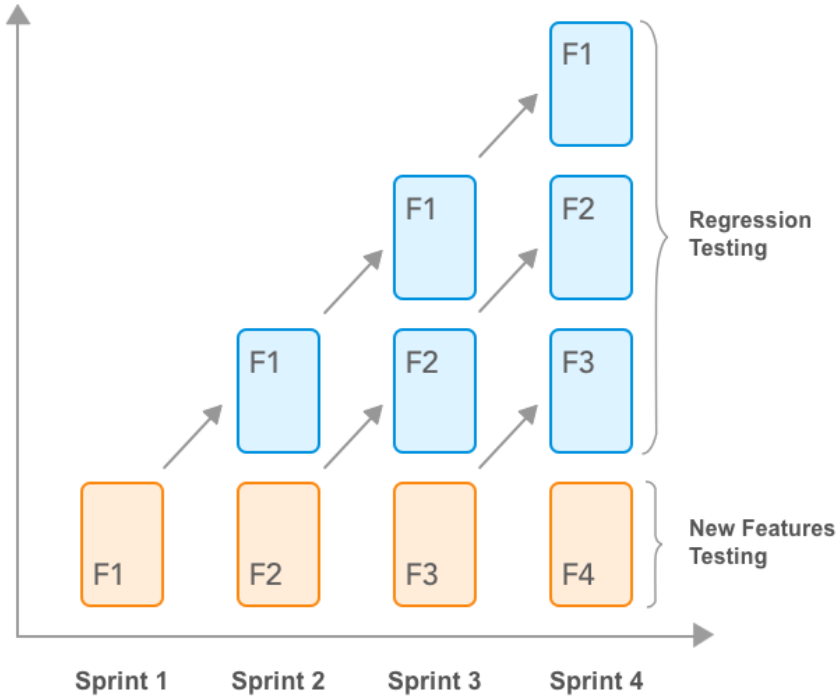
2.4.6 Tutkiva testaus

Tutkiva testaus on manuaalisen ohjelmistotestauksen lähestymistapa, jota kuvataan usein samanaikaiseksi oppimiseksi, testisuunnitteluksi ja -suoritukseksi. Tutkivassa testauksessa ohjelmistoa testataan sitä käytettäessä ilman etukäteen suunniteltuja testitapauksia. Menetelmässä luotetaan yksittäisen testaajan taitoon ja luovuuteen löytää vikoja, jotka eivät kuulu muiden testien rajoihin. Tutkiva testaus on erityisen hyödyllinen tilanteissa, joissa tarvitaan nopeaa palautetta tai halutaan löytää rajatapauksia, jotka voivat johtaa kriittisiin laatuvirheisiin. [21]

2.4.7 Regressiotestaus

Automaatiotestejä kirjoitetaan ohjelmiston oikeanlaisen toiminnan varmistamiseksi ja koodin alustavan valmistumisen jälkeen niistä tulee regressiotestejä (Kuva 3). Regressiotestit pitävät huolen, etteivät muutokset ohjelmiston koodissa aiheuta uusia virheitä eikä ohjelmiston aiemmin toteutetut ominaisuudet mene rikki. Vaikka koodi on

joskus toiminut, se ei tarkoita, että se tulisi aina toimimaan. Kun koodia refaktoroidaan, eli muutetaan koodin rakennetta selkeämmäksi tai yksinkertaisemmaksi ilman että koodin ulkoista toiminnallisuutta muutetaan, on aina olemassa riski, että aikaisemmin toimineet ominaisuudet hajoavat tai ohjelmistoon tulee uusia virheitä. [10]



Kuva 3. Regressiotestien muodostuminen ketterässä ohjelmistokehityksessä. [22]

Regressiotestaus on tehokkaampaa mustalaatikkomenetelmää käyttäen, sillä vaikka ohjelmiston sisäinen rakenne muuttuisi, sen ulkoisen toiminnan odotetaan pysyvän ennallaan. Kun ohjelmiston sisäistä toimintaa muutetaan, joudutaan melkein aina myös muuttamaan toimintaa testaavat lasilaatikkotestit. [10]

2.5 Testitapaukset

Testitapaus on joukko toimintoja, jotka suoritetaan ohjelmiston tietyn ominaisuuden tai toiminnallisuuden tarkistamiseksi. Testitapauksisiin kuuluu erilaisia testivaiheita, testidataa, alku- ja loppuväitteitä, jotka on kehitetty tietyn ohjelmiston tilanteen testaamiseen. [23]

Jokainen testitapaus tulisi ajaa omassa ympäristössään eristyksissä kaikista muista testeistä sen varmistamiseksi, että testitapaukset eivät vaikuta toisiinsa, mikä tarkoittaa

että testitapausten ajojärjestyksellä ei ole väliä. Testitapauksissa käytetään ennalta valmisteltua testidataa, jota hyödynnetään testattavan koodin suorittamisessa. Koska tiedetään, mitä koodin tulisi tehdä annetulla testidatalla, voidaan verrata odotettua tulosta saatuun tulokseen. Tätä kutsutaan väitteeksi, joka vaaditaan jokaisessa testitapauksessa. Jos tulokset ovat samat, eli väite on tosi, voidaan todeta koodin noudattavan vaatimuksia annetulla datalla.

Testitapauksen läpäisyn jälkeen ei kuitenkaan voida todeta, että koodissa ei ole virheitä, sillä testaus voi osoittaa vain virheiden läsnäolon, ei niiden olemattomuutta [15]. Onneksi ohjelmistotestauksen päämäärä ei ole kaikkien virheiden löytäminen, sillä se on mahdotonta, vaan riskin alentaminen, mikä voidaan toteuttaa testaamalla mahdollisimman kattavasti [10].

2.6 Kattavuus

Lausekattavuus on loistava, mutta ei täydellinen tapa selvittää, mitä osia koodista ei testata näyttämällä koodista jokaisen suorittamattoman rivin. Kun kattavuus on alle 100 %, lausekattavuutta käyttäen voidaan nähdä uusia tapoja kirjoittaa testejä mahdollisimman korkean kattavuuden saavuttamiseksi. Vaikka jokainen rivi koodista suoritetaan, koodi tai testit voivat silti olla viallisia. Lausekattavuus on hyvä lähtökohta ymmärtämään testientapausten täydellisyyttä, ne ovat hyödyllisiä vain, jos niitä käytetään ajattelun parantamiseen, ei korvaamiseen. [24]

Koodin kattavuuden mittaamiseen on olemassa myös muita menetelmiä kuten polkukattavuus, päätöskattavuus, ehtokattavuus ja moniehtokattavuus, mutta ne ovat rajattu tämän opinnäytetyön ulkopuolelle.

3 TEKNOLOGIAT

3.1 Ohjelmistorajapinta

Ohjelmistorajapinta on määritelmä tavoista, joilla eri ohjelmat voivat keskustella keskenään. Se määrittelee esimerkiksi minkä tyyppisiä pyyntöjä voidaan tehdä, mitä datamuotoja tulisi käyttää ja mitä käytäntöjä kuuluu noudattaa. Rajapinta helpottaa standardoitujen, uudelleenkäytettävien, helposti ymmärrettävien ja abstraktien komponenttien tekemistä. Abstraktisuus auttaa suojaamaan rajapinnan sisäistä toimintaa paljastamalla muille vain olennaiset vähimmäistiedot ja yksinkertaistamalla rajapinnan käyttöä. Rajapinnan käyttämiseksi sen sisäisestä toiminnasta ei tarvitse tietää. [25]

3.2 REST-arkkitehtuuri

Representational State Transfer (REST) on arkkitehtuurimalli ohjelmointirajapintojen toteuttamiseen käyttäen HTTP-protokollaa. RESTin tavoitteena on parantaa suorituskykyä, skaalautuvuutta, yksinkertaisuutta, muunneltavuutta, näkyvyyttä, siirrettävyyttä ja luotettavuutta ohjelmistorajapinnoissa. Nämä hyödylliset ei-toiminnalliset ominaisuudet voidaan saavuttaa noudattamalla REST-arkkitehtuurin kuutta ohjaavaa rajoitusta, jotka määrittelevät, miten rajapinta voi käsitellä ja vastata asiakkaan pyyntöihin. [26]

Yhtenäinen rajapinta

Yhtenäisyys on tärkeä ominaisuus REST-pohjaisissa rajapinnoissa. Rajapinnan resurssien osoitteiden, eli päätepisteiden, tulee olla loogisesti järjesteltyjä ja rajapinnan resurssien osoitteisiin lähetettyihin pyyntöihin vastataan yleensä JSON tai XML muodossa. [25]

Asiakas-palvelin

Jotta rajapinta ja sitä käyttävät sovellukset pysyvät mahdollisimman muunneltavina, tulee niiden pystyä kehittymään itsenäisesti, ilman että kumpikaan on riippuvainen toisesta. [25]

Tilattomuus

REST on tilaton, joka tarkoittaa, että palvelin ei tallenna mitään viimeisestä asiakkaan lähettämästä pyynnöstä ja kohtelee jokaista pyyntöä uutena. Mikäli asiakkaan sovellus vaatii tilan säilyttämistä, tulee jokaisessa asiakkaan pyynnössä olla kaikki pyynnön suorittamiseen tarvittavat tiedot. Esimerkiksi tilanteessa, jossa käyttäjän tulee pysyä sisään kirjautuneena, tulee todennustiedot lähettää jokaisessa pyynnössä. [25]

Välimuistin käyttö

REST-rajapintojen tulee tukea välimuistin käyttämistä. Datan tallentaminen välimuistiin parantaa suorituskykyä, sillä data on nopeammin asiakkaan käytettävissä ja tekee rajapinnasta skaalautuvamman vähentämällä palvelimen kuormaa. Jos välimuistiin on jo tallennettu asiakkaan pyytämä resurssi, voidaan pyyntöön vastata heti ilman, että se haetaan tietokannasta. Mikäli resurssi on tallennettu asiakkaan välimuistiin, ei resurssia tarvitse pyytää rajapinnasta. [25]

Kerroksittainen järjestelmä

Kerroksittaisen järjestelmätyylin käyttäminen sallii REST-rajapintojen muodostumisen useasta kerroksesta. Asiakas ei tiedä onko se yhteydessä suoraan loppupalvelimeen vai välittäjään. Välityspalvelimen tai kuormantasaajan asettaminen asiakkaan ja palvelimen väliin ei vaikuta heidän viestintään, eikä asiakkaan tai palvelimen koodia tarvitse päivittää. [26]

Pyydettävä koodi

Pyydettävä koodi on vapaaehtoinen rajoitus eikä se ole yleisesti käytössä. Rajoitus tarkoittaa, että rajapinta voi vaadittaessa palauttaa suoritettavaa koodia asiakkaalle. Asiakas lähettää pyynnön, johon palautetaan koodia, jota voidaan käyttää esimerkiksi käyttöliittymäkomponentin esittämiseen. Tämä yksinkertaistaa asiakkaan koodia vähentämällä siihen valmiiksi toteutettavien ominaisuuksien määrää. [25]

3.3 Python-ohjelmointikieli

Python on erittäin monipuolinen korkean tason dynaaminen ohjelmointikieli. Python on tulkattava kieli, eli kirjoitetut ohjelmat ovat välittömästi ajettavissa ilman, että niitä joutuu kääntämään. Tämä tekee siitä erittäin suositun ohjelmointikielen nopeassa sovelluskehityksessä, mutta hitaammin suoritettavan käännettyihin ohjelmistoihin verrattuna. Pythonissa on yksinkertainen syntaksi, mikä auttaa pitämään ohjelmakoodin selkeänä ja luettavana. Ohjelmointikieltä ylläpitää ja kehittää voittoa tavoittelematon Python Software Foundation. [27]

3.4 Django-verkkokehys

Avoimeen lähdekoodin perustuva Django on korkean tason Python-verkkokehys turvallisten ja ylläpidettävien verkkosivujen nopeaan ohjelmistokehitykseen. Djangoa voidaan käyttää melkein minkä tahansa tyyppisten verkkosivustojen kehittämiseen. Django sisältää tarvittavat työkalut melkein kaikkeen, mitä verkkosivujen kehittämisessä saatetaan tarvita. Kehittäjät voivat säästää aikaa ja välttyä monilta yleisiltä turvallisuusvirheitä hyödyntämällä Djangon tarjoamia työkaluja. [28]

3.5 Django REST Framework -työkalu

Django REST Framework on Djangon päälle rakennettu tehokas ja joustava työkalu verkkorajapintojen rakentamiseen. Django REST Framework vähentää rajapintojen kehittämiseen vaaditun koodin määrää, mikä yksinkertaistaa ja nopeuttaa kehittämistä. [29]

Django REST Frameworkin tarjoama *APIClient* tekee rajapinnan näkymien integrointitestauksesta yksinkertaista. Sillä voidaan simuloida rajapinnan käyttöä sisään kirjautuneena käyttäjänä ja testata koko näkymä lähetetystä pyynnöstä palautettuun vastaukseen asti.

3.6 Pytest-testaustyökalu

Pytest on avoimeen lähdekoodiin perustuva monipuolinen ja helppokäyttöinen Python-testaustyökalu, jonka avulla voidaan luoda pieniä tai monimutkaisia toiminnallisia testejä

sovelluksille ja kirjastoille. Pytestin fixtuureilla voidaan alustaa testi tunnettuun tilaan tuottamalla testidataa ja valmistelemalla objekteja testejä varten. [30]

Pytestille on tarjolla useita yhteisön tekemiä laajennusta, kuten esimerkiksi pytest-django, joka on luotu helpottamaan Django-sovellusten testausta tarjoamalla siihen hyödyllisiä työkaluja. Laajennuksen avulla testitapaukset, jotka vaativat tietokannan käyttöä merkataan *pytest.mark.django_db*-metodilla, joka varmistaa, että testeissä käytetty tietokanta nollataan automaattisesti jokaisen testitapauksen jälkeen. Tämä takaa testien riippumattomuuden, eli yksikään testi ei kykene vaikuttamaan toiseen testiin. Testien ajamisessa käytetään vakiona välimuistissa sijaitsevaa tietokantaa, joka mahdollistaa testien ajamisen erittäin nopeasti. [31]

3.7 Factory_boy-kirjasto

Factory_boy-kirjastolla määritellään helppokäyttöisiä tehtaita, joilla luodaan testejä varten tarvittavaa testidataa. Tehtaat tekevät testien kirjoittamisesta nopeampaa vähentämällä testidatan alustamiseen vaadittua työtä ja mahdollistavat monimutkaisten objektien hienomman hallinnan. Tehtaita käytetään korvaamaan staattisia ja ylläpidollisesti vaikeampia fixtuureita. [32]

Tehtaita käyttäessä riittää että ohjelmoija määrittelee vain testin kannalta tärkeät arvot. Mikäli ohjelmoija ei testejä kirjoittaessaan määrittele jollekin ominaisuudelle arvoa, tehdas täyttää sen ennalta määritellyllä arvolla. Ennalta määritelty arvo voi olla joko staattinen tai dynaaminen, eli arvo määritetään joka kerta, kun uusi objekti luodaan. [32]a

Korvaamaan tehtaiden staattisia arvoja tietynlaisissa ominaisuuksissa, kuten esimerkiksi käyttäjien etu- ja sukunimissä, voidaan hyödyntää Faker-kirjastoa. Fakeriä hyödyntäen voidaan luoda satunnaisia mutta realistisia arvoja, jotka näyttävät demoissa paremmilta ja voivat auttaa virheiden löytämisessä lisäämällä testeihin satunnaisen komponentin. [32]

Dynaamisia ominaisuuksien arvoja voidaan tuottaa *LazyAttribute*n ja *LazyFunction*n avulla. *LazyAttribute*lla ominaisuuden arvon laskemiseen voidaan käyttää objektin muita ominaisuuksia, esimerkiksi käyttäjän nimeä voitaisiin käyttää sähköpostiosoitteen luomiseen. *LazyFunction*in avulla voidaan ominaisuuden arvo hakea jostain toisesta funktiosta, esimerkiksi päivämäärä-tyyppisen ominaisuuden arvo voitaisiin täyttää Pythonin *datetime.now* metodin palauttamalla arvolla. Jotkut objektit ovat riippuvaisia

toisista objekteista. Toisesta objektista riippuvan objektin luomisessa käytetään alatehdasta eli *SubFactory*ä. Tehdas luo alatehtaan avulla toisen objektin ennen riippuvaisen objektin luomista ja luo suhteen luotujen objektien välille. [32]

4 AUTOMAATIOTESTIEN SUUNNITTELU JA TOTEUTUS

Opinnäytetyössä toteutettiin erilaisia automatisoituja yksikkö- ja integrointitestejä Digihelppari-sovelluksen REST-arkkitehtuuriin perustuvaan rajapintaan. Toteutetut testit testaavat sovelluksen toiminnallisuutta käyttäen lasilaatikko- ja harmaalaatikkomenetelmiä.

Digihelpparin-sovelluksen taustajärjestelmä on toteutettu Python-ohjelmointikieleen perustuvalla Djangolla. Testien suorittamisympäristönä toimii paikallinen kehitysympäristö, jossa on virtuaaliympäristöön asennettu Python 3.8 ja sovelluksen vaatimat Python-paketit.

Projektissa oli entuudestaan käytössä Trello, joka on hyvä projektinhallintatyökalu ohjelmistokehitystä varten. Trellossa pidettiin kirjaa vaadittavista testitapauksista ja mahdollistettiin automaatiotestien edistymisen seuranta. Projektin versionhallintajärjestelmänä toimi Git. Kun testit todettiin valmiiksi, niistä luotiin pull request ja ne yhdistettiin projektiin hyväksytyn vertaisarvioinnin jälkeen.

4.1 Testien suunnittelu

Sovelluksen kehitys oli ollut käynnissä jo pitkään ennen opinnäytetyön aloittamista, eikä sovelluksessa ollut opinnäytetyön aloittamisen aikaan lainkaan automaatiotestejä. Testien luominen päätettiin aloittaa mahdollisimman matalalta tasolta – kirjoittamalla yksikkötestit taustajärjestelmän malleille ja mallien metodeille. Yksikkötestien jälkeen rajapinnan eri päätepisteille luotiin testit, joilla varmistettiin, että kaikki dokumentoidut päätepisteet ovat olemassa eikä niistä havaita toiminnallisia virheitä. Käyttäjän tulee pystyä kirjautumaan sisään rajapinnan kautta ja kirjautuneena sisään hänen tulee pystyä luomaan, lukemaan, muokkaamaan ja poistamaan omia tietojaan. Päätepisteitä testataan myös virheellisillä arvoilla ja vastaukseksi odotetaan oikeanlaista virheilmoitusta. Luotuja yksikkö- ja integrointitestejä käytetään myös regression havaitsemiseen, mikäli koodia tullaan tulevaisuudessa muuttamaan.

Testien suorittamiseen tarvittiin testidataa, jota tuotettiin tehtailla. Factory_boyn avulla jokaiselle sovelluksen mallille luotiin tehdas. Ennen tehtaiden käyttöä ne kuitenkin testattiin mahdollisten virheiden löytämiseksi, sillä sovelluksen mallit sisältävät monimutkaisia vuorovaikutuksia. Mikäli tehtaissa, joita käytetään melkein jokaisessa

testissä, ilmenee virheitä, voivat myös tehtaita hyödyntävät testit olla virheellisiä, sillä ne käsittelevät virheellistä testidataa.

Projektin koodin lausekattavuus voidaan laskea pytest-cov -kirjastolla, jolla voidaan luoda testiajoista konfiguroitavissa olevia kattavuus raportteja. Raporteista selviää, mitä sovelluksen alueita ei testata tarpeeksi kattavasti.

4.2 Tehtaan luominen ja testaaminen

Automaatiotestauksen käytännön osuus aloitettiin luomalla jokaiselle projektin mallille tehtaat ja tehtaiden testit. Tehtaiden testejä käytettiin virheiden löytämiseen tehtaista niiden kehityksen aikana ja testien avulla tehdas voitiin todeta valmiiksi, kun testin suoritus läpäisi onnistuneesti ilman virheitä ja tehdas sisälsi tarvittavat ominaisuudet.

Esimerkkitehtaaksi esittelemään tehtaan sisältöä ja käyttöä valittiin kalenteritapahtumatehdas (Kuva 4). Tehtaassa nimetään kentät, jotka asetetaan ohjelmoijan täytettäviksi. Tapahtumalle voidaan määritellä käyttäjä, tapahtuman nimi, alkamis- ja loppumisaika sekä toistuvuuden määrä ja -taajuus. Jos käyttäjää ei ole tapahtumaa luodessa määritelty, alitehtaan avulla luodaan uusi käyttäjä ja asetetaan se tapahtuman käyttäjä-kenttään. Nimi-kenttä täytetään *Faker*in avulla, joka tässä tapauksessa palauttaa satunnaisen suomenkielisen sanan. Vakiona alkamisajaksi asetetaan tämänhetkinen aika ja lopetusaika asetetaan tunnin päähän tapahtuman alkamisajasta *LazyAttribute*n avulla.

```
class EventFactory(DjangoModelFactory):
    class Meta:
        model = Event

    user = SubFactory("api.tests.factories.UserFactory")
    name = Faker("word", locale="fi_FI")
    start = LazyFunction(datetime.now)
    end = LazyAttribute(lambda self: self.start + timedelta(hours=1))
    recurrence_count = 1
    frequency = ""
```

Kuva 4. Kalenteritapahtumatehdas.

Kuvassa 5 määritellään kalenteritapahtumatehtaalle testi. Testin alustuksessa kalenteritapahtumatehtaan *create*-metodilla luodaan uusi ja tallennetaan se tietokantaan. Metodiin annetaan parametreina vain testin kannalta tärkeät arvot:

tapahtuman nimi ja alkamisaika. Tapahtuman päättymisaikaa ei kuitenkaan määritellä, sillä se voidaan tuottaa automaattisesti tapahtuman alkamisajan perusteella tehtaassa. Tietokannasta haetaan juuri luotu kalenteritapahtuma ja varmistetaan, että sille on määriteltynä oikeanlaiset arvot.

```python
def test_event_factory():
    EventFactory.create(
        name="Test Event",
        start=datetime(year=2020, month=1, day=1, hour=12, minute=0),
    )

    event = Event.objects.first()
    assert event.name == "Test Event"
    assert event.start == datetime(year=2020, month=1, day=1, hour=12, minute=0)
    assert event.end == datetime(year=2020, month=1, day=1, hour=13, minute=0)
    assert event.user
```

Kuva 5. Kalenteritapahtumatehtaan testi.

Kuvassa 6 nähdään kuvan 5 testin suoritus, joka ei havainnut virheitä. Tehtaan voidaan nyt olettaa toimivan tarpeeksi hyvin, että sitä voidaan käyttää testeissä tuottamaan testidataa.

```
pytest factories/test_factories.py::test_event_factory -v
========================= test session starts ==========================
collected 1 item

factories\test_factories.py::test_event_factory PASSED           [100%]

========================== 1 passed in 0.98s ===========================
```

Kuva 6. Kalenteritapahtuma-tehtaan testin suorittaminen onnistuneesti.

4.3 Toistuvien kalenteritapahtumien yksikkötestaus

Yksi Digihelppari-sovelluksen tärkeimmistä ominaisuuksista on kalenteri, johon on mahdollista luoda yksittäisten tapahtumien lisäksi vuosittain, kuukausittain, viikoittain tai päivittäin toistuvia kalenteritapahtumia. Ominaisuus valittiin tässä esimerkiksi, sillä sovelluksen manuaalisen testauksen aikana toistuvien kalenteritapahtumien huomattiin sisältävän virheellisiä arvoja.

Toistuvien kalenteritapahtumien päivämäärien oikeanlaisuuden varmistamiseksi kehitettiin yksikkötestejä, jotka testaavat tapahtuman toistuvuuksien hakemista kaikilla mahdollisilla ajanmääreillä. Kuvassa 7 nähdään vuosittain toistuvan kalenteritapahtuman testi. Testi alustetaan luomalla vuosittain toistuva tapahtuma, joka tapahtuu yhteensä neljä kertaa. Alustuksen jälkeen haetaan kaikki tapahtuman toistuvuuksien päivämäärät *event.get_occurrences()* -metodilla. Testin ensimmäinen väite testaa, että metodin palauttama lista sisältää neljä arvoa – yksi jokaista tapahtuman toistuvuutta kohden. Ensimmäisen palautetun päivämäärän väitetään olevan sama kuin alkuperäisen tapahtuman päivämäärä ja jäljempien päivämäärien väitetään olevan vuosi aikaisemman päivämäärän jälkeen.

```
def test_event_occurrences_yearly():
    event = factories.EventFactory.create(
        start=datetime(year=2020, month=1, day=1),
        recurrence_count=4,
        frequency="YEARLY",
    )

    occurrences = event.get_occurrences()
    assert len(occurrences) == 4
    assert occurrences[0]["start"] == datetime(year=2020, month=1, day=1)
    assert occurrences[1]["start"] == datetime(year=2021, month=1, day=1)
    assert occurrences[2]["start"] == datetime(year=2022, month=1, day=1)
    assert occurrences[3]["start"] == datetime(year=2023, month=1, day=1)
```

Kuva 7. Toistuvan kalenteritapahtuman toistuvuuksien hakemisen testi.

Kuvassa 8 nähdään kuvan 7 ja muiden *event.get_occurrences*-metodia testaavien testien suoritus. Kaikki metodin testit läpäisivät onnistuneesti eikä tapahtumien päivämääristä löytynyt virheitä. On siis todennäköistä, että virhe sijaitsee jossain muualla kuin testit läpäisseessä metodissa, joten virheen etsintää jatketaan *Event*-mallin managerin metodista, joka käyttää juuri testattua *event.get_occurrences*-metodia.

```
============================= test session starts ==============================
collected 6 items

test_calendar.py::test_event_no_reoccurrences PASSED                     [ 16%]
test_calendar.py::test_event_occurrences_yearly PASSED                   [ 33%]
test_calendar.py::test_event_occurrences_monthly PASSED                  [ 50%]
test_calendar.py::test_event_occurrences_weekly PASSED                   [ 66%]
test_calendar.py::test_event_occurrences_daily PASSED                    [ 83%]
test_calendar.py::test_event_occurrences_filter PASSED                   [100%]

============================== 6 passed in 1.04s ===============================
```

Kuva 8. Toistuvan kalenteritapahtuman toistuvuuksien hakemisen testien suorittaminen onnistuneesti.

Kuvassa 9 suoritetaan *get_reoccurring_events*-metodin, joka testaa toistuvien kalenteritapahtumien hakemista ja todistaa metodin virheellisyyden. Testin suoritus keskeytettiin ensimmäiseen epäonnistuneeseen väitteeseen, jossa tapahtuman ensimmäisen toistokerran odotettiin olevan vuonna 2020, mutta palautetun päivämäärän vuosi olikin 2023.

```
_____________________________ test_event_get_reoccurring_events _____________________________

        events = Event.objects.get_reoccurring_events()
        assert len(events) == 4
>       assert events[0].start == datetime(year=2020, month=1, day=1)
E       AssertionError: assert datetime.datetime(2023, 1, 1, 0, 0) == datetime.datetime(2020, 1, 1, 0, 0)
E        +  where datetime.datetime(2023, 1, 1, 0, 0) = <Event: eero_ruohola's Event 1>.start
E        +  and   datetime.datetime(2020, 1, 1, 0, 0) = datetime(year=2020, month=1, day=1)

api\tests\test_calendar.py:113: AssertionError
```

Kuva 9. Toistuvien kalenteritapahtumien hakemisen testin suorittaminen epäonnistuneesti.

Tulostamalla kaikkien *get_reoccurring_events*-metodin (Kuva 10) palauttamien tapahtumien päivämäärät huomattiin kaikissa niissä olevan sama päivämäärä. Tulosteen ja metodin koodin tutkimisen perusteella pääteltiin virheen syy. Koodissa määritetään toistuvalle tapahtumalle muuttuja, mutta sen sijaan, että muuttujaan asetettaisiin kopio alkuperäisestä tapahtumasta, asetetaan siihen virheellisesti viite alkuperäiseen tapahtuma-objektiin. Tämä tarkoittaa, että kaikki toistuvaan tapahtumaan tarkoitetut muutokset tehdään sen sijaan alkuperäiseen tapahtumaan, mikä selittää miksi kaikissa metodin palauttamissa tapahtumissa on sama päivämäärä.

```python
def get_reoccurring_events(self, start: datetime = None, end: datetime = None) -> list:
    events = []
    for event in self:
        for occurrence in event.get_occurrences(start, end):
            occ_event = event
            occ_event.__dict__.update(occurrence)  # Copy everything from occurrence to occ_event
            events.append(occ_event)
    return events
```

Kuva 10. Toistuvien kalenteritapahtumien hakemisen metodi.

Kuva 10 metodin koodin virhe voidaan kuitenkin korjata yksinkertaisesti käyttämällä *copy*-funktiota. Funktion avulla luodaan uusi tapahtuma-objekti ilman, että se viittaisi alkuperäiseen tapahtuma-objektiin (Kuva 11). Näin uuteen, mutta sisällöllisesti samaan tapahtumaan voidaan asettaa uusia arvoja ilman, että muutokset vaikuttaisivat alkuperäiseen tapahtumaan. Korjauksen jälkeen kaikki testit läpäisivät onnistuneesti.

```python
for occurrence in event.get_occurrences(start, end):
    occ_event = copy(event)  # Copy original event
    occ_event.__dict__.update(occurrence)  # Copy everything from occurrence to occ_event
    events.append(occ_event)
```

Kuva 11. Korjattu toistuvien kalenteritapahtumien hakemisen metodin koodin pätkä.

4.4 Rajapinnan integrointitestaus

Opinnäytetyön aikana Digihelppari oli vielä kehityksen alla ja rajapintaan tehtiin muutos, joka sisälsi virheen. Virhe aiheutti yhden rajapinnan päätepisteen toimimattomuuden. Virhe olisi voitu huomata aiemmin, jos päätepisteille olisi ollut olemassa testit. Tämän takia osana opinnäytetyötä rajapinnan resurssien päätepisteille kirjoitettiin regressiotestit, jotta vastaavanlaiset virheet voidaan löytää mahdollisimman aikaisin ja korjata ne ennen niiden päätymistä tuotantoon.

Rajapinnan kautta mahdollistetaan käyttäjille pääsy heidän omiin resursseihinsa, joihin sisältyy esimerkiksi tallennetut kalenteritapahtumat, muistiinpanot ja käyttäjätiedot. Kuvassa 12 määritellään kalenteritapahtumien päätepisteen testi, joka alustetaan määrittelemällä testattavan päätepisteen osoite ja rajapinnan asiakas (*APIClient)*. Asiakas luodaan *get_client*-apufunktiolla, joka luo käyttäjätehtaan avulla käyttäjän ja kirjautuu sillä sisään uuteen *APIClient*tiin, ja palauttaa sen. Testissä määritellään myös rajapinnan kautta luotavalle resurssille vaaditut kentät sekä resurssin arvot, joita testissä halutaan päivittää.

```python
def test_event():
    url = "/api/event/"
    client = get_client()

    create_data = {"name": "Event", "start": datetime.now(), "end": datetime.now() + timedelta(hours=1)}
    update_data = {"name": "Updated name"}
    generic_endpoint_crud(client, url, create_data, update_data)
```

Kuva 12. Kalenteritapahtuman päätepisteen testi.

Suuri osa resurssien päätepisteistä on tarkoitettu käyttäjän resurssien lukemiseen, luomiseen, muokkaamiseen ja poistamiseen. Mustalaatikko näkökulmasta voi tuntua, että koska nämä resurssien päätepisteet toimivat samantapaisesti ja kaikkiin niihin voi käyttää samoja yleisiä metodeita, riittää, että testataan vain yksi samantapaisista päätepisteistä ajan säästämiseksi. Tämä on kuitenkin väärä ajattelutapa. Vaikka päätepisteet toimivat samantapaisesti ja yhden resurssin päätepiste toimisi, ei se tarkoita että kaikki päätepisteet toimisivat, sillä jokaisella päätepisteellä on oma koodinsa, missä voi piileksiä virheitä.

Päätepisteiden samanlaisuutta voitiin kuitenkin käyttää hyödyksi. Jokaiselle samankaltaisille päätepisteelle luotiin testit, jotka kaikki hyödyntävät samaa *generic_endpoint_crud*-apufunktiota (Kuva 13). Apufunktion avulla testattiin rajapinnasta resurssien lukemista, luomista, muokkaamista ja poistamista. Apufunktion avulla samankaltaisten testien kehittämisestä tuli yksinkertaisempaa ja nopeampaa, sillä sen avulla voitiin vähentää toistuvan koodin määrää testeistä.

```python
def generic_endpoint_crud(client, url, create_data, update_data):
    # Create an object
    endpoint_post(client, url, create_data)
    # Get all objects
    response_list = endpoint_get(client, url)
    # Get created object
    response_object = endpoint_get(client, response_list[0]["url"])
    # Update object
    endpoint_update(client, response_object["url"], update_data)
    # Delete object
    endpoint_object_delete(client, response_object["url"])
    # Endpoint is inaccessible when unauthenticated
    unauthenticated_endpoint_401(client, url)
```

Kuva 13. Yleisen päätepisteen apufunktio.

Kuvan 13 *generic_endpoint_crud*-apufunktiossa kutsutaan funktioita (Kuva 14), jotka lähettävät eri HTTP-metodeilla pyyntöjä päätepisteeseen. Nämä funktiot varmistavat, että saadun vastauksen tila on oikea, minkä jälkeen ne palauttavat saadun vastauksen datan jäsenneltynä sanakirjamuodossa, jotta sitä voidaan käyttää testin toisessa osassa. Apufunktion sisältämät funktiot testaavat päätepistettä ensin luomalla objektin rajapinnan kautta. Tämän jälkeen haettiin lista kaikista testissä määritellyistä objekteista, joita tiedetään löytyvän vain yksi, sillä testin alussa tietokanta on aina tyhjä. Tämän ainoan objektin tiedot haetaan ja objektin ennalta määritetty kenttä päivitetään. Päivitettävä kenttä ja kentän arvo määritellään jokaisen päätepisteen omassa testissään (Kuva 12). Seuraavaksi testataan luodun objektin poistamista. Lopuksi käyttäjä kirjataan ulos *ApiClient*istä ja varmistetaan, ettei testattavaa päätepistettä pysty käyttämään uloskirjautuneena.

```python
def endpoint_get(client: APIClient, url: str) -> dict:
    response = client.get(url)
    assert response.status_code == status.HTTP_200_OK
    return get_response_data(response)

def endpoint_post(client: APIClient, url: str, data) -> dict:
    response = client.post(url, data=data)
    assert response.status_code == status.HTTP_201_CREATED
    return get_response_data(response)

def get_response_data(response: HttpResponse) -> dict:
    return json.loads(response.content.decode("utf-8"))
```

Kuva 14. Rajapinnan päätepisteen testauksen funktiot GET- ja POST-metodeille sekä funktio palautetun datan purkamiseen sanakirjaksi.

Lukuisien samantapaisten päätepisteiden testien lisäksi taustajärjestelmään kehitettiin testejä myös käyttäjätietojen hallitsemiseen, kuten esimerkiksi uuden käyttäjän luomiseen, sisään kirjautumiseen ja salasanan vaihtamiseen. Lopuksi kaikkien rajapinnan automaatiotestien suoritus voidaan nähdä Kuva 15.

```
============================= test session starts ==============================
api\tests\api\test_email_reset_endpoint.py ....                          [ 16%]
api\tests\api\test_generic_endpoints_crud.py .............              [ 68%]
api\tests\api\test_location_area_endpoints.py ..                         [ 76%]
api\tests\api\test_system.py .                                           [ 80%]
api\tests\api\test_user_endpoints.py .....                               [100%]
============================== 25 passed in 2.34s ==============================
```

Kuva 15. Rajapinnan testien suoritus.

5 YHTEENVETO JA POHDINTA

Opinnäytetyön alkuperäisenä aiheena oli Full-stack testaus Digihelppari-sovelluksessa hyödyntäen testiautomaatiota, mikä olisi sisältänyt taustajärjestelmän testauksen lisäksi mobiilisovelluksen toiminnollisuus- ja käyttöliittymätestauksen. Kävi kuitenkin nopeasti ilmi, että aiheen laajuus oli liian suuri yhteen opinnäytetyöhön, joten aihe rajattiin Digihelpparin taustajärjestelmän toiminnalliseen automaatiotestaukseen.

Opinnäytetyön ensisijainen tavoite oli toteuttaa mahdollisimman kattava automaatiotestiratkaisu Digihelpparin taustajärjestelmälle. Opinnäytetyön aikana kirjoitettujen testien saavuttama lausekattavuus oli korkea 93 %. Vain pieni osa sovelluksen toiminnollisuuksista jäi täysin testaamatta. Automaatiotestejä olisi voitu kirjoittaa vielä kattavamminkin, mikäli opinnäytetyön soveltavaan osaan olisi käytetty enemmän aikaa.

Työn tuloksena automaatiotestit todettiin erittäin hyödyllisiksi osaksi ohjelmistokehitystä. Automaatiotestit auttoivat löytämään sovelluksen virheet aikaisemmin ja kasvattivat kehittäjien luottamusta koodin toimintaan. Lisäksi automaatiotestit vähensivät manuaaliseen testaukseen vaadittua aikaa, mikä nopeutti testausta ja jätti kehittäjille enemmän aikaa sovelluksen kehittämiseen. Opinnäytetyön aikana kehitetyt automaatiotestit tarjoavat monimutkaisten komponenttien testaamiseen esimerkkejä, joita voidaan käyttää pohjana sovelluksen automaatiotestien jatkokehityksessä ja mallina vastaavanlaisissa projekteissa.

Digihelpparin jatkokehityksessä tulisi ottaa testiautomaatio käyttöön. Testiautomaation avulla testit voidaan suorittaa automaattisesti esimerkiksi jokaisen pull requestin yhteydessä ennen sen yhdistämistä päähaaraan. Myös testivetoisen ohjelmistokehityksen käyttöönotto on suositeltavaa Digihelpparin jatkokehityksessä.

LÄHDELUETTELO

[1] Smartbear, ”What Is The Benefit of Test Automation and Why Should We Do It?,”. Saatavilla: https://smartbear.com/solutions/automated-testing/. [Haettu 27 4 2021].

[2] Gamu RY, ”Digihelppari,”. Saatavilla: https://gamu.fi/digihelppari/. [Haettu 6 2 2021].

[3] J. Sonmez, ”7 Common Types of Software Testing,”. Saatavilla: https://usersnap.com/blog/software-testing-basics/. [Haettu 14 7 2020].

[4] F. Ibiwoye, ”The essence of software testing,”. Saatavilla: https://medium.com/@femibiwoye/the-essence-of-software-testing-12d78c6a7beb. [Haettu 13 7 2020].

[5] Google, ”Behind the Open Source Browser Project,”. Saatavilla: https://www.google.com/googlebooks/chrome/. [Haettu 13 7 2020].

[6] T. Dolan, ”The Ultimate Guide to Software Testing,”. Saatavilla: https://www.globalapptesting.com/blog/the-ultimate-guide-to-software-testing-how. [Haettu 15 7 2020].

[7] Django, ”Writing your first Django app, part 5,”. Saatavilla: https://docs.djangoproject.com/en/3.0/intro/tutorial05/. [Haettu 20 5 2020].

[8] guru99, ”Automation Testing Vs. Manual Testing: What’s the Difference?,”. Saatavilla: https://www.guru99.com/difference-automated-vs-manual-testing.html. [Haettu 14 7 2020].

[9] U. Eriksson, ”Differences Between the Different Levels & Types of Testing,”. Saatavilla: https://reqtest.com/testing-blog/different-levels-of-testing/. [Haettu 6 1 2020].

[10] J. Sonmez, ”A Survival Guide to Test Driven Development and Unit Testing,”. Saatavilla: https://usersnap.com/blog/tdd-unit-testing/. [Haettu 6 1 2021].

[11] guru99, ”Integration Testing: What is, Types, Top Down & Bottom Up Example,”. Saatavilla: https://www.guru99.com/integration-testing.html. [Haettu 6 1 2021].

[12] softwaretestinghelp, ”What Is Integration Testing,”. Saatavilla: https://www.softwaretestinghelp.com/what-is-integration-testing/. [Haettu 10 1 2021].

[13] softwaretestinghelp, ”What Is System Testing,”. Saatavilla: https://www.softwaretestinghelp.com/system-testing/. [Haettu 15 1 2021].

[14] softwaretestingfundamentals, ”Acceptance Testing,”. Saatavilla: https://softwaretestingfundamentals.com/acceptance-testing/. [Haettu 6 1 2021].

[15] guru99, ”Static Testing vs Dynamic Testing: What's the Difference,”. Saatavilla: https://www.guru99.com/static-dynamic-testing.html. [Haettu 17 1 2021].

[16] A. Dobra, ”Gray Box Testing,”. Saatavilla: https://qabyexample.com/gray-box-testing-basics-advantages-more-tutorial/. [Haettu 13 2 2021].

[17] guru99, ”What is BLACK Box Testing? Techniques, Example & Types,”. Saatavilla: https://www.guru99.com/black-box-testing.html. [Haettu 16 1 2021].

[18] guru99, ”What is WHITE Box Testing? Techniques, Example & Types,”. Saatavilla: https://www.guru99.com/white-box-testing.html. [Haettu 16 1 2021].

[19] softwaretestinghelp, ”Functional Testing Vs Non-Functional Testing,”. Saatavilla: https://www.softwaretestinghelp.com/functional-testing-vs-non-functional-testing/. [Haettu 22 1 2021].

[20] softwaretestinghelp, ”A Complete Non-Functional Testing Guide For Beginners,”. Saatavilla: https://www.softwaretestinghelp.com/what-is-non-functional-testing/. [Haettu 22 1 2021].

[21] D. Parmar, ”Exploratory testing,”. Saatavilla: https://www.atlassian.com/continuous-delivery/software-testing/exploratory-testing. [Haettu 24 1 2021].

[22] H. Shah, ”8 Functional Testing Types Explained With Examples,”. Saatavilla: https://www.simform.com/functional-testing-types/. [Haettu 23 1 2021].

[23] guru99, ”How to Write Test Cases: Sample Template with Examples,”. Saatavilla: https://www.guru99.com/test-case.html. [Haettu 29 1 2021].

[24] N. Batchelder, ”Flaws in coverage measurement,”. Saatavilla: https://nedbatchelder.com/blog/200710/flaws_in_coverage_measurement.html. [Haettu 5 2 2021].

[25] restfulapi, ”REST Architectural Constraints,”. Saatavilla: https://restfulapi.net/. [Haettu 2021 5 10].

[26] Wikipedia, ”Representational state transfer,”. Saatavilla: https://en.wikipedia.org/wiki/Representational_state_transfer. [Haettu 16 5 2021].

[27] Wikipedia, ”Python (programming language),”. Saatavilla: https://en.wikipedia.org/wiki/Python_(programming_language). [Haettu 5 4 2021].

[28] Mozilla, ”Django introduction,”. Saatavilla: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction. [Haettu 5 4 2021].

[29] Encode OSS Ltd., ”Django REST Framework,”. Saatavilla: https://www.django-rest-framework.org/. [Haettu 2021 4 5].

[30] pytest, ”pytest: helps you write better programs,”. Saatavilla: https://docs.pytest.org/en/latest/index.html. [Haettu 6 2 2021].

[31] A. Pelme, ”pytest-django,”. Saatavilla: https://pytest-django.readthedocs.io/en/latest/helpers.html. [Haettu 17 4 20.21].

[32] factory_boy, ”factory_boy,”. Saatavilla: https://factoryboy.readthedocs.io/en/stable/. [Haettu 6 2 2021].