Nico Ala-Könni

Verkko-ohjelmointirajapinnan testauksen automatisointi

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikan tutkinto-ohjelma

Insinöörityö

18.5.2022

Tiivistelmä

Tekijä:	Nico Ala-Könni
Otsikko:	Verkko-ohjelmointirajapinnan testauksen automatisointi
Sivumäärä:	46 sivua
Aika:	18.5.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaajat:	Lehtori Antti Laiho Lehtori Miikka Mäki-Uuro

Insinöörityön tarkoitus oli tutkia verkko-ohjelmointirajapintoja, niiden testausta ja testauksen automatisointia. Työssä selvitettiin aihetta ensin lähdeaineiston avulla tutkimalla, mitä ohjelmointirajapinnat ovat, millaisia hyötyjä ja heikkouksia niihin liittyy ja miten niitä luodaan ja käytetään. Sen jälkeen tutkittiin ohjelmointirajapintoihin, niiden testaukseen ja testauksen automatisointiin liittyviä teknologioita ja työkaluja. Viimeiseksi selvitettiin ohjelmointirajapintojen testauksen suunnittelua ja toteutusta sekä automatisointia.

Rajapinnan testaus on keskeinen vaihe ohjelmiston koko elinkaaren hallintaa. Ohjelmointirajapinnoissa, niiden testauksessa ja automatisoinnissa voidaan hyödyntää useita eri teknologioita ja työkaluja. Testaussuunnitelman ja testitapausten määrittely ovat keskeinen osa testausprosessin hallintaa. Manuaalinen testaus on edellytys testauksen automatisoinnille. Oikein toteutettuna testauksen automatisointi tuo monia hyötyjä.

Työssä toteutettiin kootun tietoperustan pohjalta myös tapaustutkimus, jossa suunniteltiin ja toteutettiin verkko-ohjelmointirajapinnan testaus ja testauksen automatisointi. Testattava ohjelmointirajapinta oli yksinkertainen GraphQL-rajapinta verkkosovelluksesta, jossa pystyttiin luomaan reseptejä, kommentoimaan niitä ja antamaan niistä arvosteluja. Rajapinta oli ohjelmoitu JavaScript-ohjelmointikielellä, ja se käytti MongoDB-tietokantaa. Rajapinnan manuaalinen testaus tehtiin Postman-testaussovelluksella, testit luotiin testi- ja prosessiautomaation Robot Framework -ohjelmistolla ja testien automatisointiin käytettiin Jenkins-automaatiopalvelin.

Insinöörityön lopputuloksena päähavaintoja olivat testaussuunnitelman ja testitapausten määrittelyn hyödyllisyys ja tärkeys testauksen toteutukselle, testauksen mukanaolon tarpeellisuus jo heti ohjelmakehityksessä sekä testauksessa löydettyjen virheiden korjaaminen ja testitapausten uudelleen testaaminen.

Avainsanat: ohjelmointirajapinta, API, testien automatisointi

Abstract

Author:	Nico Ala-Könni
Title:	Automation of application programming interface tests
Number of Pages:	46 pages
Date:	18 May 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Game Applications
Supervisors:	Antti Laiho, Senior Lecturer
	Miikka Mäki-Uuro, Senior Lecturer

The purpose of this thesis was to study application programming interfaces (API), their testing and automation of tests. First, this study, it explored the topic through theory by examining what APIs are, what benefits, and weaknesses are associated with them, and how APIs are created and used. Second, the technologies and tools related to APIs, their testing and automation of tests were studied. In the last theoretical part, the planning and implementation of API testing and automation of tests were examined.

API testing is a key step in software lifecycle management. Several different technologies and tools can be utilized in APIs, their testing and automation. Defining a test plan and test cases is a key part of managing the testing process. Manual testing is a prerequisite for automating testing. Properly implemented, automation of testing brings many benefits.

The practical part of this thesis included a case study which examined the testing of the API and the automation of testing. The data of the theoretical part was used in the case study. In the case study, testing and automation of testing were planned and implemented. The API to be tested was a simple GraphQL API from a web application that was able to create, comment on, and review recipes. The API was programmed with JavaScript programming language, and it used the MongoDB database. Manual testing of the API was done with Postman API platform, tests were created with automation framework called Robot Framework, and Jenkins automation server was used to automate the tests.

The main findings of this thesis study were the usefulness and importance of the planning and the definition of test cases for the implementation of testing, the need to involve testing already in application development, and the correction of found errors and retesting of test cases.

Keywords: Application programming interface, API, automated testing

Sisällys

Lyhenteet

API: *Application programming interface*. Ohjelmointirajapinta.

HTTP: *Hypertext Transfer Protocol*. Hypertekstin siirtoprotokolla.

DB: *Database*. Tietokanta.

XML: Extensible Markup Language. Merkintäkieli.

JSON: JavaScript Object Notation. Tiedostomuoto.

CRUD: *Create, Read, Update, Delete*. Luo, lue, päivitä ja poista. Perustoiminnot rajapinnan tiedon käsittelyä varten.

SOAP: *Simple Object Access Protocol*. Tiedonsiirtoprotokolla.

REST: *Representational State Transfer*. Ohjelmistoarkkitehtuurityyli.

URI: *Uniform Resource Identifier*. Internetissä olevan resurssin yksilöivä tunniste.

VM: *Virtual Machine*. Virtuaalikone.

E2E: *End to end testing*. Päästä päähän -testaus.

1 Johdanto

Insinöörityössä tutkitaan verkko-ohjelmointirajapinnan (application programming interface, API) testausta ja sen automatisointia. Työ jakautuu kahteen osioon. Ensimmäisessä osiossa selvitetään ohjelmointirajapintoja, niiden testausta ja automatisointia lähdeaineistojen kautta. Toisessa osiossa toteutetaan ohjelmointirajapinnan testaus ja testauksen automatisointi tapaustutkimuksen avulla.

Lähdeaineistojen avulla tutkitaan ensin yleisesti, mitä ohjelmointirajapinnat ovat, mitkä ovat niiden hyötyjä ja heikkouksia ja mitä keskeisiä asioita liittyy niiden luomiseen ja käyttämiseen. Sen jälkeen tutkitaan ohjelmistorajapintoihin, niiden testaukseen ja automatisointiin liittyviä teknologioita ja työkaluja. Sitä seuraavassa osuudessa selvitetään ohjelmistorajapintojen testausta ja sen suunnittelua ja toteutusta. Viimeisessä alkuosuuden luvussa selvitetään testauksen automatisointia.

Työssä tutkitaan rajapinnan testausta ja automatisointia myös tapaustutkimuksen avulla. Esimerkkinä käytetään itse tehtyä verkkosovellusta, jossa käyttäjä pystyy luomaan reseptejä, kommentoimaan niitä ja antamaan niistä arvosteluja. Sovellus on tehty JavaScript-ohjelmointikielellä ja se käyttää MongoDB-tietokantaa. Testattava ohjelmointirajapinta, jota sovellus käyttää, on tehty GraphQL:llä. Rajapinnan testaus tehdään Robot Framework -ohjelmistolla ja testien automatisointi Jenkins-automaatiopalvelimella. Tapaustutkimuksessa käsitellään rajapinnan päästä päähän (E2E) -testausta, eikä siinä käsitellä yksikkö- tai muun tason testauksen automatisointia.

Tutkittava aihe on tärkeä, koska verkkosovellusten suosio ja rajapintojen määrä on kasvanut viime vuosikymmeninä merkittävästi. Verkkosovellusten ja rajapintojen toimivuuden vaatimus on siten myös merkittävä. Luotettavuutta lisätään ohjelmien ja rajapintojen testauksella. Testien automatisointi helpottaa ohjelmiston virheiden hallintaa ja on sen vuoksi suositeltavaa erityisesti isoissa ohjelmissa.

2 Ohjelmointirajapinta

Ohjelmointirajapinta tarkoittaa eri ohjelmien tai sovellusten välistä yhtymäkohtaa, joka mahdollistaa tietojen siirron niiden välillä (1). Rajapinnan avulla ohjelmistot voivat olla vuorovaikutuksessa toistensa kanssa. Datarajapinnaksi kutsutaan rajapintaa, jossa siirretään pelkkää tietoa. Toiminnalliseksi rajapinnaksi kutsutaan rajapintaa, joka esimerkiksi mahdollistaa järjestelmän tietojen muuttamisen. (2.)

Ohjelmointirajapintoja on neljää eri perustyyppiä: yksityinen/suljettu, julkinen/avoin, kumppani/rajattu ja yhdistelmä. Yksityinen rajapinta julkaistaan vain yrityksen sisäisesti, avoin rajapinta on avoin kenelle tahansa, kumppanirajapintaa voivat käyttää vain määritetyt osapuolet ja yhdistelmärajapinnat yhdistävät useita rajapintoja toisiinsa. (3.)

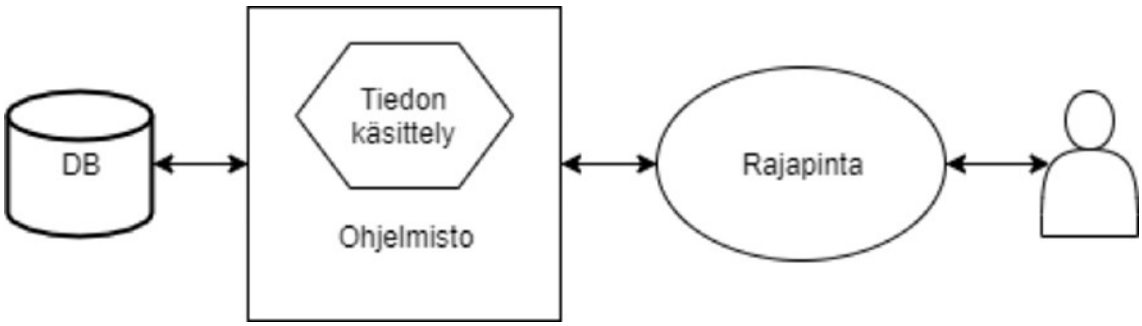
Avoimessa rajapinnassa sen ominaisuudet, kuten kuvaukset, dokumentit, testiaineistot ja muut ominaisuudet, ovat julkisia ja käytettävissä vapaasti ilman ehtoja ja lisenssimaksuja. On kuitenkin mahdollista, että saadakseen palvelun tietosisältöä laajemmin siitä voi joutua maksamaan. Rajapinnan avoimuus, maksuttomuus ja mahdollisuus käyttää erilaisia teknologioita tiedon haussa helpottavat ja mahdollistavat rajapinnan käytön esim. muiden sovellusten tekemisessä ja testaamisessa. (2.)

Rajapinnat voi luokitella lisäksi paikallisiksi, verkko- tai sovellusrajapinnoiksi. Paikallinen rajapinta antaa käyttöjärjestelmä- (OS) tai ohjelmistopalveluita sovelluksille. Verkko-ohjelmointirajapintoja käytetään esimerkiksi verkkosivuilla, ja niitä voi käyttää HTTP-protokollalla. Sovellusrajapinnat perustuvat RPC-teknologiaan. (3.)

Kuvassa 1 on esitetty ohjelmointirajapinta graafisesti:

- DB (database) tarkoittaa palvelun tietokantaa. Käyttöoikeus tietosisältöön voi olla rajattu, vaikka rajapinta olisi avoin, tai tieto voi olla kaikille avoin. Hyvä esimerkki käyttäjiltä rajatusta tiedosta on potilastietojärjestelmä, jossa potilaan tiedot ovat luottamuksellisia, vaikka sen rajapinta voi olla avoin (2).

- Tiedonkäsittely ja ohjelmisto. Ohjelmisto saa pyynnön tiettyyn tiedonkäsittelyyn, kuten hakuun, muokkaamiseen tai poistoon, tekee oikean komennon pyynnöstä ja lähettää tietokannalle.
- Rajapinta. Rajapinta ottaa vastaan käyttäjän tai sovelluksen tiedonkäsittelypyynnön ja lähettää sen ohjelmistolle sekä palauttaa vastauksen käyttäjälle.
- Käyttäjä tai sovellus voi avoimessa rajapinnassa olla kuka tahansa, mutta suljetussa rajapinnassa käyttäjät on rajattu, jolloin käyttöoikeus todennetaan käyttäjälle määritellyn roolin pohjalta (3).



Kuva 1. Ohjelmointirajapinta (3).

Rajapintojen tarkoituksena on tehdä ohjelmistojen tietojen käsittelystä yksinkertaisempaa ja laajentaa käyttömahdollisuuksia. Rajapintojen avulla niitä käyttävien sovellusten ja muiden käyttäjien ei tarvitse päästä käsittelemään suoraan palvelun tietokantaa. (3.)

2.1 Ohjelmointirajapinnan hyötyjä ja heikkouksia

Ohjelmointirajapintojen hyötyjä

Ohjelmointirajapintojen avulla käyttäjät saavat tietoa rajapintaa tarjoavan palvelun ominaisuuksista, toiminnallisuuksista ja käytöstä. Käyttäjien ei tarvitse tietää tai tuntea itse palvelun tietosisältöä tai toimintaa, vaan he pystyvät ymmärtämään sen toimintaa rajapinnan ominaisuuksien avulla. (2.)

Avoimen rajapinnan hyötyjä ovat sen avoimuus, käyttöönotettavuus ja testattavuus muille käyttäjille. Riittävän tarkka dokumentointi mahdollistaa rajapinnan sujuvan käytön ja hyödyntämisen. Käyttöönoton voi tehdä itsenäisesti ilman

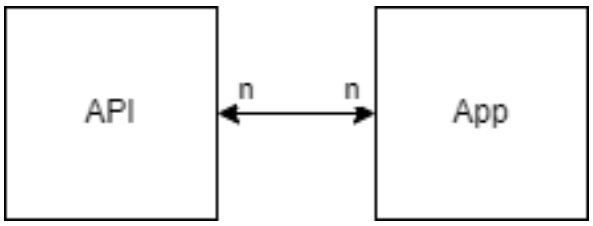
palvelun edustajaa, ja palvelun testiaineisto on saatavissa. Testauksen voi tehdä avoimella pääsyllä tuotanto- tai testijärjestelmään, tai testijärjestelmän voi ladata itselleen. (2.)

Avoimet rajapinnat parantavat ohjelmistojen ja palveluiden laatua ja toimintaa, sillä ne vähentävät kehitystyöhön käytettyä aikaa ja kustannuksia, ja ne vähentävät virheiden mahdollisuuksia. Rajapintoja käytetään esimerkiksi mobiilisovelluksissa, pilvipalveluissa ja esineiden internetissä (Internet of Things, IoT). Ne mahdollistavat käyttäjille paremman asiakaskokemuksen, ja ne luovat yrityksille uusia mahdollisuuksia kaupallistamiseen. (3.)

Avoimen rajapinnan avulla yritys antaa muille mahdollisuuden hyödyntää sovellustietojaan, mikä parantaa yrityksen brändin näkyvyyttä ja siitä tietoisuutta (3). Avoimenkaan rajapinnan ei silti tarvitse mahdollistaa pääsyä itse palvelun tietosisältöön, vaan siihen voi olla pääsy vain rajatuilla henkilöillä. (2.) Lisäksi rajapintojen avulla yritys voi valvoa ja hallita turvallisesti palvelunsa käyttäjiä ja tietoja (3).

Verkkopohjaisten palveluiden kehitys vie paljon aikaa. Avoimien rajapintojen ja niiden kautta muiden sovellusten tietojen hyödyntäminen nopeuttaa ja monipuolistaa palveluiden kehitystä ja antaa niiden kehittämiselle uusia mahdollisuuksia. (4.) Avoimen rajapinnan kautta yritykset pystyvät yhdistämään oman palvelunsa muiden yritysten palveluiden kanssa (3). Palvelun kehittäjä voi muokata avoimen rajapinnan kautta saatua dataa edelleen (4). Se antaa yrityksille uusia kaupallistamis- ja yhteistyömahdollisuuksia (3).

Usea sovellus (app) voi käyttää samaa rajapintaa (API), ja yksi sovellus voi käyttää useita eri rajapintoja, kuten on esitetty kuvassa 2. Rajapinnat antavat mahdollisuuden erottaa ohjelma pienempiin osiin ja yhdistää eri ohjelmia, mikä selkeyttää ja yksinkertaistaa ohjelmia.



Kuva 2. Sovellusten ja rajapintojen välinen yhteys (3).

Internet on tällä hetkellä ohjelmointirajapintojen tärkein lähde ja mahdollistaja. Esimerkiksi Facebook, Google ja Yahoo julkaisevat rajapintoja ja kannustavat kehittäjiä hyödyntämään omien palveluidensa sivustoja ja sovelluksia sekä tarjoavat helpon pääsyn verkkoresursseihin. (3.)

Ohjelmointirajapintojen heikkouksia

Ohjelmointirajapinnan ymmärtäminen ja ominaisuuksien tulkitseminen vaativat ohjelmointiosaamista, vaikka ohjelmointirajapinnan dokumentointi olisi tehty mahdollisimman selkokieliseksi ja ymmärrettäväksi. Ilman ohjelmointiosaamista rajapintaa ei pysty hyödyntämään ideaalisti ja helposti. (2; 5, s. 24.)

Kun ohjelmointirajapinta on verkossa, se on altis haitallisille käyttäjille. Sen vuoksi rajapinnasta on tehtävä mahdollisimman turvallinen, ettei etenkään henkilötietoja pääse ulkopuolisille tahoille. Ohjelmointirajapinnan välillä kulkevia tietoja ei salata oletuksena, vaan se täytyy osata tehdä rajapintaan tietoisesti. (6.)

Koska avoimen rajapinnan kautta verkkopalvelun dokumentit, testaukset ja muut ominaisuudet ovat avoimesti kaikkien saatavissa, voi kuka vain, myös kilpailija, tehdä palvelun ominaisuuksien pohjalta oman palvelunsa (2). Tämä voidaan nähdä ongelmana, mutta toisaalta se voi olla myös mahdollisuus, koska muut käyttäjät pääsevät kehitetyn sovelluksen pohjalta kehittämään sitä edelleen ja uusi palvelu voi myös olla alkuperäisen palvelun kehittäjän käytettävissä.

Rajapinnan, erityisesti avoimen rajapinnan, ylläpito, dokumentointi ja testaus vievät resursseja ja aiheuttavat kustannuksia. Rajapinnan päivityksen yhteydessä dokumentaation ja testausaineiston on pysyttävä ajan tasalla. Lisäksi

rajapinnan ohjelmiston ja tiedonkäsittelyn on pysyttävä kevyinä, sillä palvelun-tarjoajan kapasiteetin tulee pystyä käsittelemään tulevat pyynnöt nopeasti isoinakin määrinä.

2.2 Ohjelmointirajapinnan luominen ja käyttö

Rajapinta koostuu ohjelmistosta, joka hoitaa tiedonkäsittelyn tietokannan kanssa, ja rajapinnasta, joka hoitaa kommunikoinnin ulos ja sisään. Rajapinta luodaan tapauskohtaisesti sovitulla ohjelmointikielellä. Ohjelmointikielen valinta perustuu esimerkiksi haluttuihin teknologioihin ja ohjelmoijan osaamiseen. Ohjelmointirajapintoja voidaan rakentaa esimerkiksi seuraavilla kielillä:

- C#
- Java
- PHP
- Python
- JavaScript.

C# on moderni olio-ohjelmointikieli, jota käytetään verkkopalveluiden ja Windows-sovellusten tuottamiseen (7). Javaa käytetään paljon ohjelmisto- ja mobiilikehityksessä sekä peleissä (8). PHP:tä käytetään erityisesti web-sovelluskehityksessä. Python on ystävällinen uusille käyttäjille ja sopii pieniin ja keskikokoisiin projekteihin. (7.) JavaScript on moderni ohjelmointikieli, jota käytetään muun muassa www-sivujen ja rajapintojen luomisessa sen helppokäyttöisyyden ja sen antamien mahdollisuuksien vuoksi (9).

Verkkopalveluja toteutetaan nykyisin yhä useammin www-palveluna, joka käyttää HTTP-protokollaa palvelun eri osien tiedonsiirrossa (10). Tiedonkäsittely tapahtuu käyttäjän tekemällä HTTP-pyynnöllä, johon palvelimen rajapinta lähettää pääosin pyynnön mallin mukaisen vastauksen, minkä jälkeen palvelin sulkee yhteyden (11, s. 7–8). XML- ja JSON-muotoja suositaan vastauksen muotona, koska niidentyyppiset tiedostot on helppo muokata muille sovelluksille ja ohjelmille. (12.)

HTTP-palvelupyyntö rakentuu eri osista, kuten metodista, URIsta, otsikkotiedosta ja viestirungosta:

- metodi: GET, POST, PUT, PATCH, DELETE
- URI: rajapinnan kutsun osoite, joka saattaa sisältää parametrejä
- otsikkotiedot: kutsun lisätiedot
- viestirunko: sisältää luotavan tai muokattavan tiedon (11).

Rajapinnan vastaus sisältää HTTP-statuskoodin. 1xx-alkuiset statuskoodit tarkoittavat informatiivisia, 2xx-alkuiset suoritettuja, 3xx-alkuiset uudelleenohjattavia, 4xx-alkuiset käyttäjän virheitä, 5xx-alkuiset palvelimen virheitä ja 9xx-alkuiset patentoituja virheitä. Eri tilanteille, tiloille ja statuksille on eri koodit. (13.) Vastaus saattaa sisältää myös tietoa viestirungossa, riippuen käytetystä metodista ja ohjelmistosta. Yleisesti GET palauttaa aina haetun tiedon, mutta myös POST, PUT ja PATCH voivat palauttaa lisätyn tai muokatun tiedon vastauksessa.

Rajapinnat ovat datan käsittelyä varten. Rajapinnan perustoimintoja voi kutsua lyhyesti sanalla CRUD, joka koostuu englanninkielisistä sanoista create, read, update ja delete eli suomeksi luo, lue, päivitä ja poista. Luomisella tarkoitetaan uuden tiedon lisäämistä tietokantaan, lue-toiminto noutaa haettavan tiedon, mutta ei muuta sitä, päivittämistoiminto päivittää haettavia tietoja ja poistamistoiminnolla poistetaan halutut tiedot (14).

Rajapinnan suunnittelu ja tietorakenteiden ja parametrien määrittely on tärkeää, jotta rajapinnan kutsu on yhteensopiva rajapinnan tietorakenteen kanssa (3). CRUD toimii hyvänä muistisääntönä tarpeellisista perustoiminnoista käyttökelpoisten verkkosovellusten rakentamisessa. CRUD vastaa HTTP-menetelmiä POST, GET, PUT ja DELETE. (14.)

Rajapinnasta tehdyn dokumentaation tulee olla riittävän kattavaa ja sisältää rajapinnan päätepisteet, toiminnot ja parametrit (2). Dokumentaatio kuuluu avoimen rajapinnan määritelmään, mutta sen on hyvä olla myös muissa rajapinnoissa. Dokumentaation luonti onnistuu rajapinnan ohjelmoinnin kanssa samaan aikaan, ja se helpottaa ylläpitoa. Rajapinnan dokumentointiin on useita

tapoja. Esimerkiksi Swagger on yksi työkalu rajapinnan suunnitteluun ja dokumentointiin.

3 Ohjelmointirajapintojen ja testauksen teknologiat

Ohjelmointirajapinnoissa, niiden testauksessa ja automatisoinnissa voidaan hyödyntää useita eri teknologioita ja työkaluja. Rajapinnan teknologiat voivat auttaa esimerkiksi rajapinnan dokumentoinnissa ja kehityksessä. Testauksen työkalut auttavat testitapausten luonnissa, suorittamisessa sekä testauksen automatisoinnissa ja rajapinnan toimivuuden valvonnassa (15, s. 18). Automatisoinnin työkalut auttavat testien säännöllisessä ajossa ilman testaajan vuorovaikutusta (5, s. 32).

3.1 Ohjelmointirajapinnan teknologiat

Verkkorajapintojen tiedonsiirtoa standardoidaan erilaisilla protokollilla (12). Protokollia voidaan kutsua myös tiedonsiirron viitekehyksiksi tai ohjelmistoarkkitehtuurityyleiksi. Yleisimmät verkkopalveluprotokollat ovat

- SOAP (Simple Object Access Protocol)
- REST (Representational State Transfer).

SOAP ja REST ovat erilaisia tekniikoita, ja niillä on erilaiset käyttötarkoitukset, minkä vuoksi niitä ei voi suoraan verrata toisiinsa. SOAP on enemmänkin protokolla, kun taas REST on arkkitehtoninen tyyli, joka voi hyödyntää SOAP-protokollaa samalla tavalla kuin HTTP:tä. (16.).

SOAP on XML-pohjainen tiedonsiirtoprotokolla. Se on alustasta ja kielestä riippumaton. (17.) SOAPilla on SSL-tuki ja sisäänrakennettu WS-Security-standardi, joiden avulla se pyrkii tietojen turvalliseen lähettämiseen. SOAPia käytetään, kun sovellukselta toivotaan korkeampaa turvallisuustasoa, luotettavaa viestintää, viestintää vanhojen järjestelmien kanssa tai ACID-yhteensopivuutta (Atomicity, Consistency, Isolation, Durability). ACID on konservatiivisempi tietojen johdonmukaisuusmalli, minkä vuoksi sitä suositaan yleensä taloudellisia tai muuten arkaluontoisia tapahtumia käsiteltäessä. (16.)

REST perustuu URIiin (Uniform Resource Indentifier) ja HTTP-protokollaan. Tiedonsiirron HTTP-protokollien ansiosta REST on yksinkertainen (16). REST-rajapinnan kanssa voi XML:n lisäksi kommunikoida myös muun muassa JSON-, CSV-, RSS- ja HTML-tietomuodoilla (18). RESTin etuja ovat muun muassa pienempi määrä suojausvaatimuksia, selaimen yhteensopivuus ja skaalautuvuus (16). REST on yksinkertainen toteuttaa ja integroida verkkopalveluihin (3).

Viime vuosina ovat yleistyneet myös

- gRPC (Google Remote Procedure Call)
- GraphQL (Graph Query Language) (16).

gRPC (Google Remote Procedure Call) on Googlen kehittämä avoimen lähdekoodin järjestelmä, joka käyttää HTTP/2:ta. Sitä käytetään yleisesti palveluiden yhdistämiseen mikropalveluarkkitehtuurissa ja mobiililaitteiden yhdistämisessä taustapalveluihin. gRPC:n etuja ovat JSON:a kevyemmät viestit, korkea suorituskyky, sisäänrakennettu koodin luominen ja tuki useammille yhteysvaihtoehdoille, kuten datan suoratoistolle. (16.)

GraphQL on kyselykieli, joka tukee lukemista, kirjoittamista ja tietojen muutosten tilaamista. GraphQL-palvelimia on saatavana mm. JavaScript-, Python-, C++- ja muille kielille. GraphQL kommunikoi HTTP:n avulla ja käyttää JSON-tietomuotoa. Merkittävä etu GrahpQL:ssä on, että siinä voi määrittää useita eri tietoja, jotka halutaan palauttaa palvelimelta yhdellä API-kutsulla. (16.) GraphQL-teknologialla tieto siirtyy nopeammin ja tehokkaammin kuin REST-mallilla. GraphQL on myös järjestelmistä riippumattomampi kuin REST. (19.)

Rajapintojen kehityksen apuna voi käyttää eri kuvauskieliä, jotka auttavat rajapinnan kuvaustiedoston luonnissa. Kuvaustiedosto sisältää rajapinnan käyttömahdollisuuksista kuvauksen, joka on sekä koneen että ihmisen luettavissa. Kuvaustiedosto on eri työkalujen käytettävissä esimerkiksi testaamiseen ja dokumentointiin, ja tällöin helpottaa ja nopeuttaa rajapinnan kehitystyötä. (20.)

3.2 Testauksen teknologiat

Rajapintojen testaus on tärkeä ohjelmiston kehittämisen vaihe. Testaustyökalujen käyttö edesauttaa testauksen varmentamista, sen nopeampaa toteuttamista, sen saamista kustannustehokkaammaksi ja testauksen tekemistä ilman syvää teknistä osaamista. Työkalut tarjoavat ohjelmistokirjastoja tai käyttöliittymiä HTTP-kutsujen kanssa toimimiselle. (15, s. 19.)

Ohjelmointirajapintojen testaamiseen on olemassa useita erilaisia työkaluja, esimerkiksi

- Robot Framework (robotframework.org)
- Postman (postman.com)
- Wireshark (wireshark.org).

Robot Framework on avoimen lähdekoodin ohjelmisto, jota voidaan käyttää testi- ja prosessiautomaatiossa. Se kehitettiin alun perin Nokia Networksissa. Robot Frameworkiä käytetään erilaisten laitteiden, ohjelmistojärjestelmien ja protokollien testaamiseen graafisten käyttöliittymien, rajapintojen ja useiden muiden käyttöliittymien kautta. Se on kirjoitettu Python-ohjelmointikielellä, ja sen ominaisuuksia voidaan laajentaa Pythonilla, Javalla tai monilla muilla ohjelmointikielillä. Robot Frameworkilla on yksinkertainen syntaksi, joka käyttää avainsanoja (keyword). Sen ympärillä on laaja verkosto kirjastoja ja työkaluja, joita kehitetään erillisinä projekteina. Robot Framework voidaan integroida muiden työkalujen kanssa tehokkaiden ja joustavien automaatioratkaisujen luomiseksi. (21.)

Postman on rajapinnan testausta helpottava sovellus, jolla voidaan suorittaa HTTP-kutsuja rajapintoihin graafisesta käyttöliittymästä (22). Se on hyvä työkalu manuaaliseen testaukseen. Postmanissa on kattavasti työkaluja, jotka nopeuttavat testausta, suunnittelua ja dokumentointia. (23.) Sen avulla voi luoda eri ympäristöjä, joihin voi tallentaa usein käytettyjä muuttujia eri testiympäristöjä varten, kuten URL-muuttujan, jolloin olemassa olevia kutsuja voi suorittaa eri ympäristöissä. Postmanissa voi ryhmitellä kutsuja, mikä auttaa testien organisoinnissa, ja tiedostoja voi myös viedä ja tuoda Postmanista. (22.)

Wireshark on laajasti käytetty sovellus verkkoprotokollan analysointiin. Se on suosituin kaltaisensa ohjelmisto, joka tarjoaa paljon erilaisia ominaisuuksia. Wiresharkilla voi kaapata oman koneen verkkotapahtumat ja vuorovaikuttaa niiden kanssa. (24.)

3.3 Testauksen automatisoinnin teknologiat

Automaatiotestaustyökalut ovat ohjelmistokehyksiä tai sovelluksia, jotka auttavat testausten toteutuksessa ja ajavat niitä automaattisesti. Manuaalisen testauksen automatisointi mahdollistaa sen, että testauksia voidaan tehdä ajastetusti, systemaattisesti ja säännöllisesti ilman testaajan itsensä aktiivista vuorovaikutusta muuten kuin alussa automatisoinnin määrittelyssä, testauksessa ja suorittamisessa. Testausautomaation hyödyllisyyden ja ohjelmiston laadun pysyvyyden varmistamiseksi tulee automaatiotestaus saada osaksi rajapinnan jatkuvaa integraatiota ja kehittämistä, jolloin virheitä pystytään ehkäisemään ja havaitsemaan nopeammin. (25.)

Jatkuvan integraation työkaluja ovat esimerkiksi

- Jenkins
- GitLab CI/CD.

Jenkins on itsenäinen, avoimen lähdekoodin automaatiopalvelin, jota voidaan käyttää kaikenlaisten ohjelmistojen rakentamiseen, testaamiseen ja toimittamiseen tai käyttöönottoon liittyvien tehtävien automatisoimiseen (26). Jenkins on helposti laajennettavissa oleva tuote, jonka toimintoja voidaan laajentaa asentamalla lisäosia. Jenkins sisältää yli 1500 laajennusta, jotka mahdollistavat Sen automatisoinnin käytännössä minkä vain ohjelmistotekniikan kanssa. (27.) Jenkins on rakennettu Java-virtuaalikoneelle (Java Virtual Machine, JVM), ja se voidaan asentaa alkuperäisten järjestelmäpakettien kautta, Dockerin kautta tai jopa suorittaa itsenäisesti millä tahansa koneella, johon on asennettu Java Runtime Environment (JRE) (26).

Virtuaalikoneita (Virtual Machines, VM) voidaan ja kannattaa hyödyntää testien automatisoidussa ajamisessa. Virtuaaliteknologiat mahdollistavat erillisen

käyttöjärjestelmän käytön nykyisen käyttöjärjestelmän sisällä. Virtuaalikoneita ei tarvitse asentaa tai alustaa manuaalisesti omalle koneelle joka kerta ympäristöä tehdessä, vaan ne voidaan asettaa virtuaalikoneen luonnin yhteydessä.

GitLab CI/CD on ohjelmistokehityksen työkalu. GitLab CI (Continuous Integration) -palvelu rakentaa ja testaa ohjelmistoa aina, kun kehittäjä syöttää koodia sovellukseen. GitLab CD (Continuous Deployment) on ohjelmistopalvelu, joka asettaa tehdyt koodinmuutokset tuotantoon päivittäin. (28.) GitLab CI/CD:tä käytetään vikojen ja virheiden havaitsemiseen koodin integroinnissa ja varmistamaan, että kaikki tuotantoon käytetty koodi noudattaa sovellukselle määritettyjä koodistandardeja. GitLab CI/CD voi automaattisesti rakentaa, testata, ottaa käyttöön (deploy) ja valvoa sovelluksia. (29.)

4 Ohjelmointirajapinnan testaus

Rajapinnan testauksen tavoitteena on toiminnallisuuden varmistaminen ja virheiden etsiminen. Se on keskeinen vaihe ohjelmistokehittämistä. (5, s. 116.) Ohjelmistorajapinnat on tärkeää testata ennen julkaisua, mutta myös julkaisun jälkeen. Se varmistaa, että tiedonjako toimii halutulla tavalla ja kaikilla alustoilla ja ettei se aiheuta sovellusongelmia tai tietojen korruptoitumista. Rajapinnan testaus on yleensä osa sovellusten elinkaaren hallintaa (application lifecycle management, ALM). (3.)

Avoin rajapinta edellyttää, että rajapinta on myös muiden kuin sen kehittäjien testattavissa tai siitä tehty testiaineisto on saatavilla. Rajapinnan testattavuuden voi tehdä antamalla avoin pääsy tuotanto- tai testijärjestelmään tai lataamalla testijärjestelmä käyttäjälle omaan käyttöön. (2.)

Ohjelmiston testaus on prosessi tai prosessien sarja, joka on suunniteltu varmistamaan, että ohjelmiston koodi tekee sen, mitä se on suunniteltu tekemään, ja päinvastoin, ettei se tee mitään tahatonta. Ohjelmiston tulee olla ennustettavissa ja johdonmukainen, eikä se saa tuottaa käyttäjille yllätyksiä. (30, s. 123.) Täysin täydellistä ja kattavaa testausta on kuitenkin mahdotonta tehdä, sillä se edellyttäisi kaikkien ohjelmistoon ja testaukseen vaikuttavien asioiden

huomioimista, mikä on mahdotonta. Testausta käytetään siten vain virheiden olemassaolon varmistamisessa ja löytämisessä, ei virheiden puuttumisen osoittamisessa. (30, s. 5–6.)

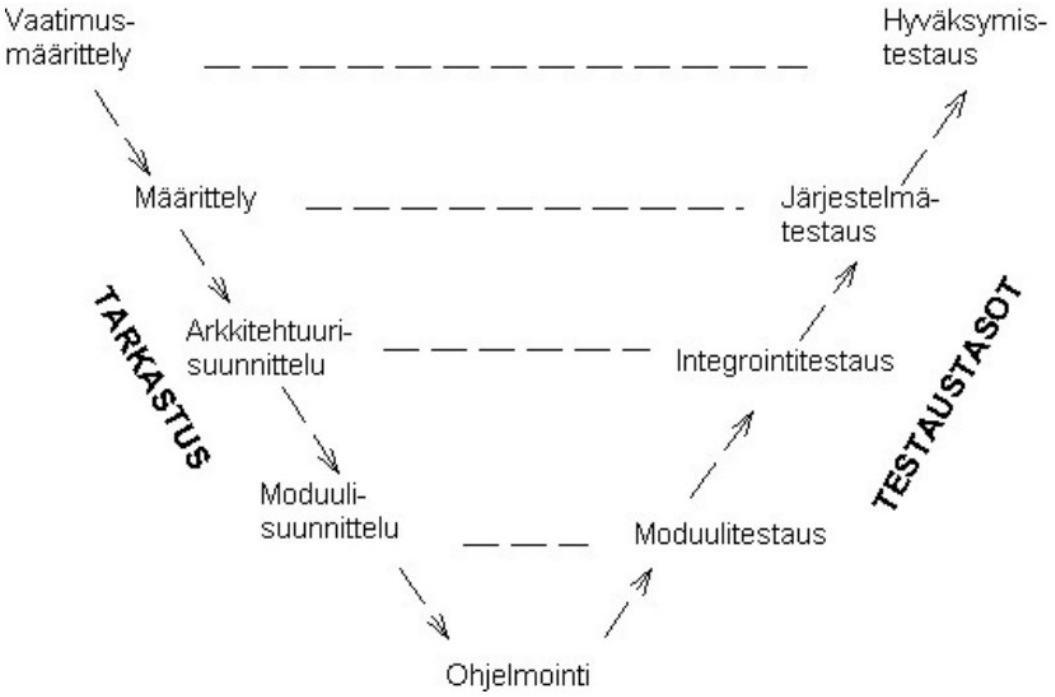
4.1 Testaustasot

Ohjelmistotestausta voi toteuttaa ja tarkastella erilaisilla tasoilla ja menetelmillä. Niiden määrittely etukäteen tarkentaa testauksen kohdetta ja tarkoitusta sekä selkeyttää testaussuunnitelman ja itse testauksen tekemistä. Testaustasojen määrittely vähentää tarpeetonta testausta ja varmistaa suurten virheluokkien löytämisen (30, s. 128).

Erilaisia testaustasoja voi kuvata esimerkiksi niin sanotulla V-mallilla, jossa erilaisia testausvaiheita toteutetaan ohjelmiston eri kehitys- ja rakentamisvaiheissa. Vaiheet on esitetty kuvassa 3. Kuvan alareunassa, kolmion kärjessä, ohjelmiston taso, ja siten myös testauksen taso, ovat hyvin konkreettisia ja rajattuja kokonaisuuksia. Kun kuvan kolmiota nousee ylöspäin, muuttuvat ohjelmiston taso ja sen mukaisesti myös testauksen taso abstraktimmaksi, ylätason ja isomman kokonaisuuden testaamiseksi. (31, s. 39–40.) Eri testaustasot voivat olla osittain päällekkäisiä ajallisesti muiden testausprosessien kanssa (30, s. 128).

Kuvassa 3 esitetyt V-mallin mukaiset testausvaiheet ovat

- yksikkötestaus (tai moduuli- tai komponenttitestaus)
- integrointitestaus
- järjestelmätestaus
- hyväksymistestaus (31, s. 40).



Kuva 3. Ohjelmistotestauksen V-malli (31, s. 40; 34, s. 13).

Yksikkötestauksessa (tai moduuli- tai komponenttitestaus) testataan ohjelmakooditasolla ohjelman, luokan tai komponentin pienimpiä osia, kuten yksittäisiä funktioita (31, s. 42). Yksikkötestauksessa verrataan yksittäisen moduulin toimintaa johonkin moduulia määrittelevään toiminnalliseen määritykseen tai rajapintamääritykseen. Koska yksikkötestauksessa testataan vain pientä osaa kerrallaan, pystytään testauksen kokonaisuutta hallitsemaan paremmin, virheet pystytään paikantamaan ja korjaamaan tarkasti ja rinnakkaisia testauksia pystytään toteuttamaan samanaikaisesti. (30, s. 91.) Yksikkötestausten tekeminen tuo luotettavuutta kokonaistestaukseen, sillä seuraavalla testaustasolla voidaan luottaa yksittäisten moduulien toimivuuteen (31, s. 41 ja 47). Selkeä ohjelmistokoodin rakenteen jäsentely ja dokumentointi auttavat ohjelmistokoodin testausta.

Integraatiotestauksessa (integrointitestaus) varmistetaan ohjelman testattujen moduulien yhteentoimivuutta ja niiden rajapintojen vuorovaikutusta (31, s. 50). Integraatiotestaus määritellään arkkitehtuurisuunnitelman pohjalta. Testauksessa voi yhdistää kaikki moduulit kerralla testattavaksi tai lisätä moduulien

määrää vähitellen, mikä on suositeltavampi testaustapa. (31, s. 52–53.) Testauksessa voi käyttää esimerkiksi GitLab CI/CD -työkalua. Integraatiotestausta ei välttämättä tehdä erillisenä testausvaiheena, vaan se voi olla osa yksikkötestausta (30, s. 129).

Järjestelmätestauksessa (system testing) testataan koko järjestelmää tai ohjelmaa yhtenä kokonaisuutena, joka sisältää ohjelmiston, laitteet ja tietokannat, ja verrataan niiden toimivuutta sen alkuperäisiin dokumentoituihin ja mitattavissa oleviin tavoitteisiin. Testausta ei siten tehdä teknisten vaatimusmäärittelyjen lähtökohdista, vaan loppukäyttäjän ja ohjelmiston alkuperäisen tavoitteen lähtökohdista. Ulkoisten vaatimusten dokumentointi ja ohjelmointikielelle muuttaminen ovat ohjelmistokehityksessä yleensä virhealttein vaihe, ja se on sen vuoksi tärkeintä myös testata. (30, s. 130.) Järjestelmätestauksessa voidaan testata esimerkiksi käsiteltävän tieto- tai pyyntömäärän volyymia, kuormituksia, ohjelmiston turvallisuutta, suorituskykyä, käytettävyyttä, resurssien (esim. muistin ja levytilan) käyttöä, konfiguraatioita (yhteenliittymiä), asennettavuutta, virheestä toipumista sekä luotettavuutta ja saavutettavuutta. (30, s. 133–142; 31, s. 70 ja 73.)

Hyväksymistestaus (acceptance testing) tehdään ennen ohjelmiston käyttöönottoa (31, s. 61). Testauksessa varmistetaan, että ohjelmisto täyttää sille asetut alkuperäiset vaatimukset ja loppukäyttäjien nykyiset tarpeet. Hyväksymistestauksen tekee yleensä ohjelman asiakas tai loppukäyttäjä. (30, s. 144.) Alfatestaukseksi kutsutaan hyväksymistestausta, joka tehdään toimittajan tiloissa, kun taas beetatestaus tehdään asiakkaan omissa tiloissa ja suoritusympäristössä. Asiakkaan tai loppukäyttäjän testausmallin päättää asiakas itse, eikä se välttämättä pohjaudu olemassa oleviin määrittelydokumentteihin. Asiakkaan ja loppukäyttäjän näkökulmasta ohjelmiston käytettävyys on tärkeä testattava asia. Havaitut virheet analysoidaan ja priorisoidaan, ja niiden korjauksesta ja kehittämistoimista sovitaan asiakkaan ja toimittajan välillä. (31, s. 61 ja 64.)

Osana ohjelman käyttöönoton testauksia ovat myös regressio- ja asennustestaukset, vaikka niitä ei katsotakaan osaksi V-mallin mukaista ohjelmistokehityksen testausta. Regressiotestaus suoritetaan sen jälkeen, kun ohjelmaan on

tehty toiminnallinen parannus tai korjaus. Sen tarkoituksena on selvittää, onko muutos heikentänyt ohjelman muita näkökohtia. Se tehdään yleensä suorittamalla uudelleen jokin ohjelman testitapausten osajoukko. Regressiotestaus on tärkeää, koska muutokset ja virheenkorjaukset ovat yleensä paljon alttiimpia virheille kuin alkuperäinen ohjelmakoodi. (30, s. 147.) Asennustestaus puolestaan suoritetaan järjestelmän asennuksen jälkeen. Sen tarkoituksena ei ole löytää enää ohjelmistovirheitä, vaan asennuksen aikana ilmeneviä virheitä. Asennuksessa voi tapahtua virheitä esimerkiksi käyttäjän tekeminä väärinä valintoina, tiedostojen ja kirjastojen käytettävyydessä, laitteistojen toimivuudessa ja verkkoyhteyksissä. (30, s. 145.)

Päästä päähän -testaus (E2E) on tekniikka, joka testaa koko ohjelmistotuotteen alusta loppuun asti varmistaakseen, että sovellus toimii odotetulla tavalla. Päätarkoitus on testata loppukäyttäjän kokemusta simuloimalla todellista käyttäjän skenaariota ja validoimalla testattava sovellus ja sen komponentit integroinnin ja tietojen eheyden kannalta. (32.)

4.2 Testausmenetelmät

Ohjelmistoja voidaan testata erilaisilla menetelmillä. Perinteinen tapa luokitella testausmenetelmiä on jakaa ne mustalaatikkotestaukseen (black-box testing) ja lasilaatikkotestaukseen (white-box testing) (30, s. 9).

Mustalaatikkotestaus on käyttäjän käyttäytymiseen perustuva testaus. Sitä voidaan kutsua myös dataohjatuksi- tai panos-/tuotosohjatuksi testaukseksi. Se on toiminnallinen testaus, jossa ei välitetä ohjelmiston sisäisestä toiminnasta, koodista tai rakenteesta, vaan sen sijaan hyödynnetään yksinomaan ohjelmiston vaatimusmäärittelyjä ja käyttötapausten kuvauksia ja selvitetään, mistä määrittelystä ohjelmisto ei suoriudu. (30, s. 9 ja 223.)

Lasilaatikkotestaus on rakenteellinen testaus, joka testaa ohjelmiston sisäistä rakennetta. Sitä voidaan kutsua myös logiikkaohjatuksi testaukseksi. Testaus hyödyntää ohjelmiston sisäistä toimintalogiikka, minkä vuoksi se edellyttää ohjelmakoodin käyttämistä. Testauksessa pyritään käymään läpi eri

suorituspolkuja ja siten varmistamaan erilaisten käyttötilanteiden toimivuus. (30, s. 11 ja 225.)

Suositeltavaa on käyttää testauksessa sekä mustalaatikko- että lasilaatikkotestausta. Jos testauksessa käytetään sekä mustalaatikko- että lasilaatikkostestausta, kutsutaan testausta hybriditestausstrategiaksi. (30, s. 44.)

Testausmenetelmiä voidaan luokitella myös esimerkiksi staattiseen ja dynaamiseen testaukseen. Staattisessa testauksessa etsitään ja ehkäistään ohjelmistokoodin virheitä katselmoimalla ja analysoimalla suunnitelmia ja vaatimusmäärittelyjä. Testaus pyritään tekemään ja virheisiin puuttumaan mahdollisimman varhaisessa vaiheessa ohjelmiston kehitystä. Dynaamisessa testauksessa puolestaan käytetään ohjelmistokoodia hyödyksi ja etsitään olemassa olevia vaatimustenmukaisuuden virheitä ohjelmistokoodissa. (31, s. 95 ja 105.)

Tilanteen ja tarpeen mukaan testausmenetelmiä voidaan luokitella myös funktionaalisiin (toiminnallisiin) ja ei-funktionaalisiin (ei-toiminnallisiin). Funktionaalisessa testauksessa testataan ohjelmiston toiminnallisuutta vaatimusten mukaisella tavalla. Ei-funktionaalisessa testauksessa testataan ohjelmiston ominaisuuksia, kuten suorituskykyä, tehokkuutta, luotettavuutta tai käytettävyyttä. (31, s. 70 ja 72.)

5 Testauksen suunnittelu ja toteutus

Ohjelmiston testaus kannattaa suunnitella ja toteuttaa samanaikaisesti ohjelmiston määrittelyn ja kehittämisen kanssa (31, s. 177). Testauksella todennetaan ohjelmiston vaatimusten, määrittelyjen ja prosessien mukaisuus, ja siten mitataan ohjelmiston laadullista onnistumista (33). Testauksen merkittävimpiä vaaranpaikkoja ovat, että testaukselle ei ole resursoitu riittävästi osaavia henkilöitä ja työaikaa tai testauksessa käytetään epäsopivaa testausmenetelmää. Siksi testaussuunnitelman tekeminen on keskeinen osa testausprosessin hallintaa. (30, s. 145–146.)

5.1 Testaussuunnitelman laatiminen

Testaussuunnitelmassa määritellään testauksen toteutus, resurssit ja aikataulu. Testaussuunnitelmasta tulee tehdä kirjallinen dokumentti. Suunnitteluvaiheessa kirjattuja asioita seurataan ja raportoidaan testauksen edetessä, ja tarvittaessa niihin puututaan ja niitä tarkennetaan testauksen ohjauksen yhteydessä.

Hyvässä testaussuunnitelmassa käsitellään muun muassa seuraavia asioita:

- johdanto
- testauksen kohde
- testauksen tavoitteet
- testausstrategia ja -periaatteet
- testausympäristö
- lopputulokset
- resurssit ja vastuutahot
- aikataulu
- riskit. (31, s. 176; 33; 30, s. 146–147.)

Johdanto on lyhyt kuvaus testauksen kohteesta, käytetyistä menetelmistä ja keskeisistä dokumenteista ja standardeista (31, s. 225; 30 s. 146).

Testauksen kohteessa rajataan tarkemmin, mitä ohjelmistoa, sen ominaisuuksia tai versiota suunnitelma kattaa ja mitä se ei kata. Kohteen kuvauksessa voi olla viittauksia keskeisiin dokumentteihin, kuten ohjelmiston määrittelyyn, suunnitelmaan ja käyttöohjeeseen (31, s. 226). Kaikki ohjelmiston kohteet eivät ole välttämättä aina helposti testattavissa, esimerkiksi silloin, jos rajapinnat eivät ole käytettävissä tai testiympäristön rakentaminen ei ole mahdollista (31, s. 226, 274). Tällaiset rajaukset on hyvä kirjoittaa perusteluineen suunnitelmaan näkyviin.

Testauksen tavoitteissa määritellään tarkemmin, mitä testauksella ja eri testausvaiheilla halutaan saavutettavan, esimerkiksi luovutettavan ohjelmiston vaatimustenmukaisuus, laatutaso, asiakastyytyväisyys, kustannushyöty ja niin edelleen (31, s. 73 ja 226).

Testausstrategiassa ja -periaatteissa kuvataan testausmalli, testauksen toimenpiteitä, testaustekniikoita, työkaluja, testauksen kattavuuden mittausta ja testauksen edistymisestä raportointia. Lisäksi määritellään kriteereitä testauksen aloitukseen, hyväksymiseen, hylkäykseen ja keskeytykseen. (30, s. 146; 31, s. 226–227; 33.)

Erillisen testausmallin teko on suositeltavaa etenkin monimutkaisten ja laajojen ohjelmistojen testauksessa. Testausmalli kuvaa ohjelmiston käyttäytymistä, rakennetta ja ympäristöä, ja se nostaa esiin keskeisimmät testattavat ominaisuudet. Testausmallin teossa voi hyödyntää esimerkiksi UML (Unified Modeling Language) -kaavioita. (34, s. 2 ja 9.) UML on standardoitu mallinnuskieli, joka auttaa ohjelmiston määrittelyn ja toimintojen visualisoinnissa, rakentamisessa ja dokumentoinnissa (35).

Testitapauksia voidaan määritellä suunnitelmassa, tai niistä voi olla hyvä tehdä erillinen liite, sillä testitapauksia on hyvä täydentää ja tarkentaa testauksen yhteydessä (34, s. 22). Suuressa testausprojektissa tarvitaan systemaattisia menetelmiä testitapausten tunnistamiseen, kirjoittamiseen ja tallentamiseen.

Tärkeää on myös määritellä mekanismit havaittujen virheiden raportoimiseksi, korjausten etenemisen seuraamiseksi ja korjausten lisäämiseksi järjestelmään (30, s. 147).

Testausympäristössä määritellään tekninen ympäristö, jossa testaus tehdään tai tulee tehdä, esimerkiksi millä laitteistolla, ohjelmilla, tietoliikenneyhteyksillä, työkaluilla, ohjelmakirjastoilla, käyttöoikeustasoilla. Testaustyökalujen osalta on tunnistettava myös, kuka ne hankkii ja miten, ja milloin niitä käytetään. (31, s. 228; 30, s. 146.)

Lopputuloksissa kuvataan, mitä konkreettisia lopputuloksia testauksesta tulee saada ja miten ne jatkokäsitellään. Kuvataan esimerkiksi, miten havaintoihin puututaan, miten muutospyynnöt kirjataan ja miten niitä hallitaan, mitä dokumentteja tulee tehdä ja minne ne tallennetaan. Testauksen päätyttyä on hyvä tehdä myös loppuraportti ja arviointi testauksen onnistumisesta. (33.)

Resursseissa ja vastuutahoissa on tunnistettu testaukseen tarvittavat henkilöt ja henkilöryhmät. Resursoinnin voi tehdä esimerkiksi vaiheittain tai osa-alueittain: testauksen hallintaan, suunnitteluun, suorittamiseen, valvontaan, testiympäristön valmisteluun sekä suunnitelman ja lopputulosten hyväksyntään liittyen. On resursoitava myös henkilöt, jotka korjaavat havaitut virheet. (30, s. 146; 31 s. 229.)

Aikataulussa testaus ja sen eri osa-alueet vaiheistetaan ja vaiheisiin ja työtehtäviin kuluva aika jaetaan kalenteriin (30 s. 146). Vaiheiden aikataulutuksessa tulee huomioida niiden mahdolliset riippuvuudet toisistaan. (34, s. 29). Aikataulutuksessa tulee huomioida myös muun muassa mahdollinen laitteiston konfigurointiin tarvittava aika (30, s. 146).

Riskeissä tunnistetaan asioita, jotka voivat vaikuttaa testauksen toteuttamiseen tai sen tavoitteiden toteutumiseen. Keskeisimmille riskeille tulee määritellä ennaltaehkäiseviä hallintakeinoja ja toimintamalleja riskin toteutumisen varalle. (31 s. 229; 33.)

5.2 Testitapausten määrittely

Onnistunut ohjelmiston testaus perustuu testitapausten tunnistamiseen ja määrittelyyn. Testitapauksella tarkoitetaan ohjelmiston toiminnallisuutta, jonka tulisi toimia etukäteen määritellyllä ja vaatimusten mukaisella tavalla. Testijoukoksi tai testiajoksi kutsutaan usean testitapauksen ryhmää, jossa testitapauksen tulos voi toimia syötteenä seuraavalle testille. (31, s. 9–10 ja 107.) Testituloksen käyttöä seuraavassa testissä on hyvä kuitenkin olla varovainen, jotta testejä olisi helppo ajaa tarvittaessa myös yksittäin. Testeille voi tarvittaessa asettaa aloitus- tai lopetustoiminnot, joilla voidaan esimerkiksi alustaa ja nollata sovelluksessa käytettyjä tietoja.

Testitapausten suunnittelu ja rakentaminen tulisi aloittaa heti ohjelmiston tuotannon alkuvaiheessa, jolloin virheitä pystytään korjaamaan heti määrittely- ja rakentamisvaiheissa. Testitapaukset tulee suunnitella ja valita, ja niistä tulee raportoida huolellisesti. Eri testaustasoille ja -menetelmille määritellään erilaisia

testitapauksia. (31, s. 25–26 ja 93.) Testaussuunnitelman ja testitapausten määrittely auttavat merkityksellisimpien testitapausten tunnistamisessa ja siten varmistavat, että testaus priorisoituu oikeisiin asioihin (33).

Testitapausten suunnittelu vaatii testaajalta osaamista, rajapinnan tuntemista ja myös luovuutta, jolloin testaaja esimerkiksi tunnistaa, missä kohdin virheitä todennäköisimmin sattuu. Eri testauksen vaiheet on yleisesti jaettu myös eri henkilöille. Lisäksi testaajan tulee toimia järjestelmällisesti ja yksityiskohtaisesti. Testaajan osaamista ja asiantuntemusta on hyvä käyttää myös ohjelman suunnittelu- ja rakentamisvaiheissa. (31, s. 17; 34, s. 7.)

5.3 Manuaalinen testaus

Manuaalisessa testauksessa testaaja itse testaa ohjelmiston toimivuuden testaussuunnitelman mukaisesti suorittamalla määriteltyjä testitapauksia (25). Testaustulosten luotettavuuden varmistamiseksi testauksen tulee olla etukäteen harkittua, kattavaa, dokumentoitua ja suunnitelmallista. (33.)

Manuaalitestaus vie usein enemmän aikaa ja resursseja kuin automatisoitu testaus. Siinä on kuitenkin omat hyötynsä. Manuaalisessa testauksessa testaaja pystyy esimerkiksi helpommin havaitsemaan visuaalisia ohjelmistovirheitä ja tarkistamaan pieniä ohjelmistossa tehtyjä muutoksia. (25.)

6 Testien automatisointi

Testauksen automatisointi on järkevää pidemmällä tähtäimellä, mutta se edellyttää, että testaus on tehty ensin manuaalisesti (25). Toisaalta testauksen automatisointi ei ole hyödyllistä, jos testaus tehdään ohjelmaan kertaluonteisesti tai käyttöön otettava ohjelma on käyttäjälle uusi (31, s. 205; 33).

Testien automatisoinnilla tarkoitetaan manuaalisesti määriteltyjen testitapausten automatisointia eli testien suorittamista testausohjelmalla koneellisesti niin, että testauksia voidaan toteuttaa ilman manuaalista työvaihetta. Testauksen automatisointi edellyttää, että olemassa on testitapaukset ja niiden odotetut

vaatimus- ja määrittelydokumentteihin perustuvat tulokset. Automatisointi edellyttää myös erillistä testausympäristöä, jossa olevan testitietokannan avulla testitapauksia voidaan suorittaa ja toistaa. (25.)

Automatisoidussa testauksessa testejä voidaan helposti toistaa, minkä ansiosta testauksesta saadaan kattavampaa, nopeampaa ja kustannustehokkaampaa (5, s. xvii). Testauksen automatisoinnilla tuotetaan ja suoritetaan useita testitapauksia järjestelmällisesti ja toistettavasti. Testit on hyvä automatisoida käynnistymään tietyin väliajoin, mikä pitää testauksen osaltaan kattavana ja systemaattisena. Myös vanhoja testejä on hyvä ylläpitää, jolloin niitä voidaan tarvittaessa hyödyntää uudelleen ohjelmiston eri kehitysvaiheissa. (5, s. 24.)

On hyvä osata huomioida, että testausautomaatio (test automation) tarkoittaa eri asiaa kuin automatisoitu testaus (automated testing). Testausautomaatiolla viitataan automatisoinnin hallintaan, seurantaan ja suorittamiseen. (25.)

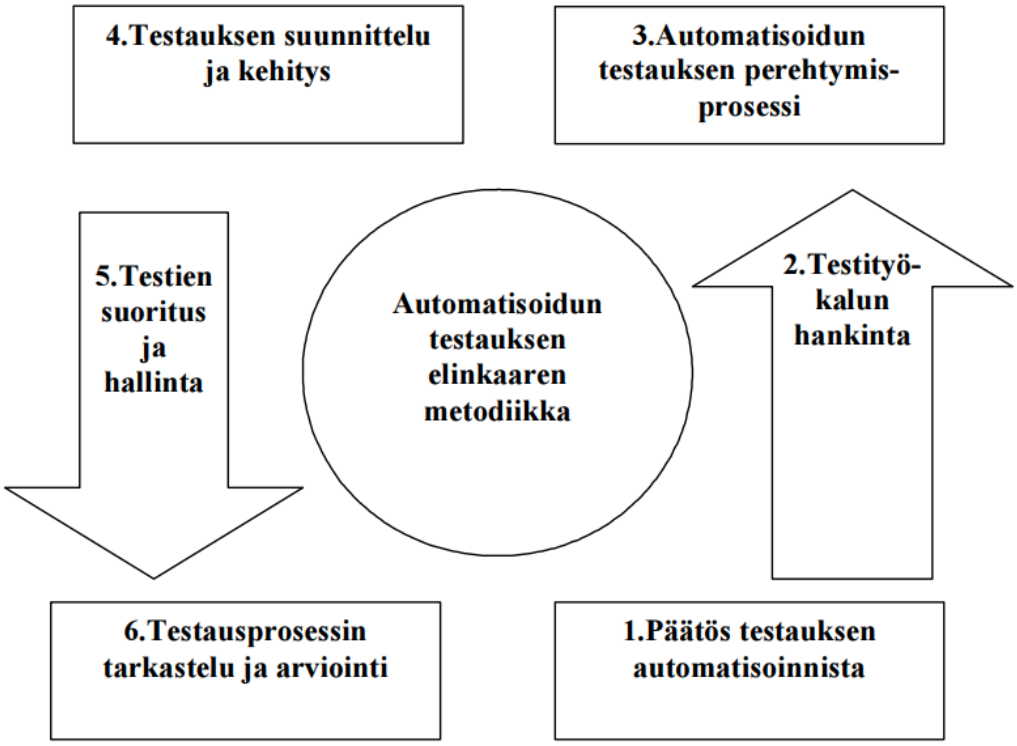
Jatkuva integraatio (Continuous Integration, CI) on olennainen ohjelmistokehitysmenetelmä, joka on huomioitava myös testausten automatisoinnissa. Jatkuva integraatio tarkoittaa, että aina kun ohjelmistoon tehdään ohjelmakoodin muutos, päivittyy koodin käännös myös testausautomaatioon ja jatkuvan integraation työkalu testaa ohjelmiston uusilla asetuksilla automaattisesti. Jatkuvan integraation työkalujen käyttö automaatiotestauksessa parantaa ja nopeuttaa virheiden löytymistä muutostilanteissa ja varmistaa ohjelmiston laatutason. (36.)

6.1 Automatisoidun testauksen elinkaari

Kuvassa 4 on esitetty automatisoidun testauksen elinkaari. Elinkaari on syklinen prosessi, joka voidaan suorittaa uudelleen prosessin viimeisen vaiheen päätyttyä (5, s. 9). Prosessi on kuusivaiheinen. Vaiheet ovat

- päätös testauksen automatisoinnista
- testityökalun hankinta
- automatisoidun testauksen perehtymisprosessi
- testauksen suunnittelu ja kehitys
- testien suoritus ja hallinta

- testausprosessin tarkastelu ja arviointi. (5, s. 9.)



Kuva 4. Automatisoidun testauksen elinkaaren vaiheet (5, s. 9).

Ensimmäisessä vaiheessa analysoidaan manuaalisen testauksen nykytilanne, muun muassa testauskäytäntö ja siihen liittyvät kehitystarpeet. Nykytilanneanalyysin lisäksi päätökseen automatisoinnista vaikuttavat automatisoinnille asetut odotukset, edut verrattuna manuaaliseen testaukseen, johdon ja esimiesten tuki ja automatisoinnissa käytettävien työkalujen vertailu ja arviointi. Mikäli kokonaisanalyysin perusteella voidaan todeta, että manuaalinen testausjärjestelmä ei ole riittävän tehokas, se päätetään automatisoida. (5, s. 10 ja 29.)

Toisessa vaiheessa hankitaan automatisoinnin testityökalu. Hankinta edellyttää johdon tuen, organisaation systeemiympäristön arvioinnin ja erilaisten automatisoinnin työkalujen arvioinnin. Kun potentiaaliset työkaluehdokkaat on tunnistettu, tutkitaan niitä tarkemmin ja ne pisteytetään ja arvioidaan. Työkalujen vertailtavia ominaisuuksia voivat olla esimerkiksi käytettävyys, toimivuus ja

suorituskyky. Hankinnassa tulee noudattaa myös organisaatiossa määriteltyä hankintaprosessia, jossa voidaan esimerkiksi määritellä, että tarjoukset tulee pyytää vähintään kolmelta eri tarjoajalta. Ehdokkaiden vertailun pohjalta tehdään päätös hankittavasta työkalusta ja käynnistetään pilotointivaihe. Työkalun hankinnassa on huomioitava työkalun yhteensopivuus organisaation systeemiympäristön, kuten alustojen, ohjelmistojen ja laitteistojen, kanssa ja muistettava, että työkalun hankinta vie aikaa, joten sille tulee varata riittävästi resursseja. (5, s. 12 ja 14.)

Kolmannessa vaiheessa perehdytään testausprosessiin. Testausprosessissa tarkennetaan organisaation testauskäytännön ja sen kehityskohteiden vaatimuksia ja tutkitaan niiden soveltuvuutta valittuun testaustyökaluun käytännössä. Testausprosessille määritellään tavoiteltavat päämäärät, testausstrategiat ja mittaaminen. (5, s. 12.)

Neljännessä vaiheessa tehdään testaussuunnitelma. Siinä on samanlaisia asioita kuin on luvussa 7.1, jossa selvitetään testaussuunnitelman määrittelyä. Erityisesti testauksen automatisoinnissa, ja sitä myöten testauksen kehittämisessä, testaussuunnitelmassa on hyvä olla ainakin seuraavia asioita: testauksen suunnittelu (esimerkiksi aikataulu ja resurssit), suorittaminen, dokumentointi, kehittäminen, mittarit ja valvonta sekä kokoonpanon hallinta, virheen jäljitys, työkalun valinta ja siihen perehtyminen, roolit ja vastuut. (5, s. 13).

Viidennessä vaiheessa suoritetaan testit suunniteltujen testaustasojen ja -menetelmien mukaisesti. Testit ja testien tulokset on tärkeää dokumentoida ja mittaroida, sillä niiden avulla pystytään seuraamaan testausprosessin laatua. Testauksessa noudatetaan testaussuunnitelmassa määriteltyä kriteeristöä testauksen aloittamisesta, hyväksymisestä, hylkäämisestä ja keskeyttämisestä. Testauskokonaisuuden hallinnassa yksi vaikeimpia osa-alueita on testattavan järjestelmän vaatimus- ja määrittelydokumenttien muutosten hallinta. (5, s. 13–14.)

Kuudennessa vaiheessa tarkastellaan ja arvioidaan suoritettua testausta ja saatuja tuloksia. Arvionnissa käytetään suunnitteluvaiheessa määriteltyjä mittareita,

ja niiden avulla voidaan todeta alkuperäisten suunnitelmien toteutuminen. Arviointituloksia on hyvä hyödyntää seuraavalla testikierroksella. (5, s. 14.)

6.2 Hyötyjä ja heikkouksia

Oikein toteutettuna testauksen automatisointi tuo monia hyötyjä. Etuja ovat muun muassa testattavan ohjelman saaminen luotettavaksi, testauksen laadun parantuminen ja testauksen manuaalisen työmäärän väheneminen. (5, s. 37.)

Automatisoidut testaukset tuovat merkittävää lisähyötyä ympäristöissä, joissa on tarpeen testata usein päivittyviä ohjelmia säännöllisesti. Tällöin regressiotestauksia voidaan ajaa olemassa olevia testejä hyödyntäen. Testejä voidaan automatisoituina ajaa entistä useammin, helpommin ja lyhyemmässä ajassa. Automatisointi lyhentää siten ohjelmiin käytettyä testausaikaa. Testien automatisointi mahdollistaa myös sellaisten testien suorittamisen, jotka olisivat manuaalisesti vaikeita tai mahdottomia suorittaa. (5, s. 38–41.)

Testauksen automatisoinnissa on myös heikkouksia. Testaukseen voi kohdistua odotuksia, jotka ovat epärealistisia. Automatisointi ei vastaa asetettuja odotuksia esimerkiksi silloin, jos testaajat eivät ole kokemukseensa perustuen tyytyväisiä testaukseen tai testaustyökaluun, testaukset on organisoitu huonosti, dokumentointi on puutteellista tai testit eivät löydä virheitä odotetusti. Jos testit eivät löydä virheitä, ei se välttämättä kuitenkaan johdu niiden huonosta suunnittelusta. (5, s. 32.)

Automatisoiduille testaukselle on ominaista, että pääosin virheet löytyvät ensimmäisen testauskerran aikana, ja sen jälkeen löytyvät virheet aiheutuvat enimmäkseen ohjelmaan myöhemmin tehdyistä korjauksista. Testaaja saattaa silloin kokea, ettei automatisoinnista ole riittävästi hyötyä. Voi myös olla, että automaattisen testauksen virheettömyys antaa testaajalle väärän mielikuvan ohjelman vaatimustenmukaisesta toiminnasta. On kuitenkin muistettava, että testitkin voivat olla virheellisiä ja niitä tulee päivittää ohjelmiston kehittämisen myötä. Automaattisia testejä tulee siis jatkuvasti ylläpitää, ja ylläpito voi olla joskus vaikeaa. Lisäksi myös itse testaustyöväline voi toimia virheellisesti. (5, s. 49.)

7 Ohjelmointirajapinnan testaus ja automatisointi

Insinöörityön tapaustutkimuksessa suunniteltiin ja toteutettiin verkko-ohjelmointirajapinnan testaus ja testauksen automatisointi. Testattava ohjelmointirajapinta on yksinkertainen ja pieni GraphQL-rajapinta verkkosovelluksesta, jossa käyttäjät voivat luoda reseptejä, kommentoida niitä ja antaa niistä arvosteluja. Sovelluksen rajapinta on ohjelmoitu JavaScript-ohjelmointikielellä, ja se käyttää MongoDB-tietokantaa. Rajapinnan manuaalinen testaus tehtiin Postmanilla, testit luotiin Robot Frameworkillä ja testien automatisointiin käytettiin Jenkinsiä.

Tapaustutkimuksessa tehty testaus on osa hyväksymistestausta. Se toteutettiin mustalaatikkotestausmenetelmällä, joka on käyttäjän käyttäytymiseen perustuvaa testausta. Tapaustutkimuksessa tehdyt testit ovat osa CI/CD:tä, eli jatkuvaa integrointia ja julkaisua.

Tapaustutkimus toteutettiin vaiheittain. Ensin selvitettiin testattavan sovelluksen GraphQL-rajapinta, sitten laadittiin testaussuunnitelma, suunniteltiin ja laadittiin lista testitapauksista, tehtiin manuaalinen testaus Postmanilla, toteutettiin suunnitelmaan ja testitapauksiin perustuva testaus Robot Frameworkillä ja lopuksi testauksen automatisointi Jenkinsillä. Nämä vaiheet selvitetään insinöörityön luvuissa 7.1–7.4. Kunkin luvun lopussa selvitetään, millaisia havaintoja vaiheen toteutuksessa huomattiin tapaustutkimusta tehdessä.

Tapaustutkimuksessa tutkitut vaiheet olivat keskeisiä ohjelman testauksessa ja testauksen automatisoinnissa. Testaussuunnitelman avulla pyrittiin varmistamaan, että testaus ja sen automatisointi tehtiin riittävän kattavasti ja systemaattisesti. Testattavan rajapinnan kuvaus ja testattavan kokonaisuuden ymmärtäminen olivat tärkeitä testauksen toteuttamiseksi ja testitapausten luontia varten. Testitapausten avulla voitiin testata sovelluksen vaatimusten ja määrittelyjen mukainen toiminta. Testauksen manuaalisessa toteuttamisessa sovellettiin testaussuunnitelmaa ja testitapauksia. Manuaalinen testaus oli tärkeä vaihe, sillä sen pohjalta määriteltiin robot-testien toiminta. Testauksen automatisoinnin avulla voitiin ajastaa testausten suorittaminen säännölliseksi ja systemaattiseksi

ilman testaajan erillistä vuorovaikutusta. Siitä on merkittävä hyöty esimerkiksi ohjelmakehityksessä.

7.1 GraphQL-rajapinta

Testattava ohjelmointirajapinta oli yksinkertainen ja pieni GraphQL-rajapinta verkkosovelluksesta, jossa käyttäjät voivat luoda reseptejä, kommentoida niitä ja antaa niistä arvosteluja. Rajapinta oli ohjelmoitu JavaScript-ohjelmointikielellä, ja se käytti MongoDB-tietokantaa.

Testattava verkkorajapinta oli kehittämisvaiheessa ja perustyypiltään tällöin yksityinen, koska sitä ei ollut vielä jaettu tai julkaistu muille käyttäjille. Rajapintaa testattiin tapaustutkimuksessa omalla koneella. Kun verkkosovellus olisi testattu ja se olisi valmis, olisi se mahdollista julkaista. Julkaisun jälkeen sovellus voisi olla avoin.

Rajapinnassa oli CRUDin perustoiminnot, eli rajapinnan kautta pystyy luomaan ja lukemaan reseptejä, kommentteja ja arvosteluja, päivittämään reseptejä ja arvosteluja sekä poistamaan reseptejä, kommentteja ja arvosteluja. Kaikki toiminnot paitsi lukeminen vaativat kirjautumaan käyttäjän tekemillä tunnuksilla, jotta saa tokenin kutsuihin.

GraphQL:ssä voi tehdä kolme erilaista operaatiota: query, mutation ja subscription. Query-operaatiolla haetaan sovelluksesta tietoa, mutation-operaatiolla tietoja muutetaan, päivitetään tai poistetaan, subscription-operaatio on tilaustoiminto, joka voi muuttaa tulosta ajan myötä. Saatavilla olevat pääoperaatiot, niiden tyypit, kentät ja relaatiot on määritelty GraphQL-skeemassa. GraphQL:ssä skeemassa voi käyttää kolmea erilaista tyyppiä: objektityyppi (object type), skalaarityyppi (scalar type) ja lueteltu tyyppi (enum).

Rajapintaan tulleet kyselyt kohdistetaan GraphQL:ssä HTTP POST -tyyppisinä samaan URI-osoitteeseen. Rajapinnan vastaukset tulevat JSON-muodossa. Vastaukset tulevat samaan rakenteeseen, kuin missä kysely on. Jos vastauksesta löytyy errors-kenttä, se tarkoittaa, että kyselyn suorittamisessa on ollut

virhe. GraphQL:ssä virheet eivät välttämättä tule esiin suoraan HTTP-status-koodista. Jos palautteessa on ilmennyt virhe, se täytyy varmistaa erikseen joka kerta, koska rajapintapyynnön palautus näyttää usein positiiviselta. Mahdollinen virhe täytyy löytää palautuksesta erikseen, toisin kuin REST, jossa virhe palauttaa vain virheen.

7.2 Testauksen suunnittelu

Testaussuunnitelman ja testitapausten määrittely auttavat varmistamaan, että testaus priorisoituu oikeisiin asioihin.

Testaussuunnitelma

Testaussuunnitelmassa sovellettiin insinöörityön luvussa 5.1 esitettyä testaussuunnitelman rakennetta ja sisältöä. Testaussuunnitelma oli taulukon 1 mukainen.

Taulukko 1. Tapaustutkimuksen testaussuunnitelma.

TESTAUSSUUNNITELMA:	
Johdanto:	GraphQL-ohjelmointirajapinnan testaus Postmanilla ja Robot Frameworkillä sekä testauksen automatisointi Jenkinsillä.
Testauksen kohde:	Testauksen kohde on JavaScript-ohjelmointikielellä tehty sovelluksen rajapinta, jossa käyttäjät voivat luoda reseptejä, kommentoida niitä ja antaa niistä arvosteluja. Testauksessa käytettävä keskeisin dokumentti on kuvaus GraphQL-rajapinnasta. Testattavan sovelluksen vaatimus- ja määrittelydokumentit sekä käyttöohje eivät poikkeuksellisesti ole tässä aineistossa mukana.
Testauksen tavoitteet:	Testauksen tavoitteena on varmistaa sovelluksen vaatimustenmukaisuus ja laatutaso.

Testausstrategia ja -periaatteet:	Testausmalli: hyväksymistestaus. Testausmenetelmä: mustalaatikkotestaus, eli käyttäjän käyttäytymiseen perustuva testaus. Erillistä testausmallia tai UML-kaaviota ei tehdä, sillä testattava sovellus on pieni. Testauksen toimenpiteitä: manuaalinen testaus, testien luonti, testien automatisointi. Testitapaukset liitetään testaussuunnitelman liitteeksi. Testitapauksia täydennetään tarvittaessa testauksen edetessä. Työkaluja: Postman, Robot Framework, Jenkins. Testauksen kattavuus: Testaus kattaa rajapinnan perustoiminnallisuudet käyttäjän näkökulmasta. Kriteerit: - aloitukseen: testiympäristö on käytettävissä - hyväksymiseen: testitapaukset suoritetaan Jenkinsissä.
Testausympäristö:	Omalla tietokoneella.
Lopputulokset:	Testausprojektin loppuraportti on tämä tapaustutkimus.
Resurssit ja vastuutahot:	Testaus ja automatisointi suoritetaan osana insinöörityön tapaustutkimusta.
Aikataulu:	Vaiheet: 1. testaussuunnitelman ja testitapausten määrittely 2. manuaalinen testaus ja testien luominen 3. testien automatisointi. Riippuvuudet: Manuaalinen testaus ja testien luonti tehdään ennen testauksen automatisointia.
Riskejä ja niiden hallintakeinoja:	- Testausta ei tehdä osana ohjelmistokehitystä. Hallintakeino: Testaus ja virheiden korjaukset tehdään ennen ohjelman käyttöönottoa. - Testaukselle ei ole riittävästi resursseja. Hallintakeino: Resurssien riittävyyttä seurataan testauksen aikana ja muita työtehtäviä priorisoidaan tilanteen mukaan.

Testaussuunnitelman laatiminen oli hyödyllistä, sillä sen avulla pystyi selventämään muun muassa testauksen kohteen, menetelmät ja rajoitteet. Testaussuunnitelma toimii hyvin myös informatiivisena työkaluna.

Haasteellista suunnitelman teossa oli, että siinä piti saada testauksen kokonaisuus hahmotettua jo ennen varsinaisen testauksen käynnistymistä. On kuitenkin hyvä muistaa, että suunnitelmaa voi usein tarvittaessa päivittää ja tarkentaa testausprojektin edetessä.

Testitapausten määrittely

Testattavan rajapinnan testitapauksia määriteltiin osana testaussuunnitelmaa. Testitapauksilla pyrittiin tunnistamaan rajapinnan toiminnallisuuksia, joiden pitäisi toimia ohjelman vaatimusten ja määritysten mukaisesti.

Testitapauksista on määritelty seuraavat tiedot:

- ID: Testitapauksen yksilöivä tunniste, jolloin esimerkiksi niihin viittaaminen on helpompaa.
- Testitapaus: Rajapinnan yksittäinen toiminnallisuus, jota testattiin.
- Priorisointi: Testitapauksen merkittävyys ohjelmakokonaisuudessa. Priorisoinnin asteikko voi vaihdella. Tässä tapaustutkimuksessa käytettiin asteikkoa 1–3. 1 tarkoittaa kriittistä toiminnallisuutta, jonka on ehdottomasti toimittava ohjelmassa. 2 tarkoittaa merkittävää, mutta ei kriittistä toiminnallisuutta. 3 tarkoittaa tapausta, jossa virheiden olemassaolo ei ole kriittistä ohjelman toiminnalle.
- Positiivisten (pos.) ja negatiivisten (neg.) testitapausten luokittelu. Positiivisen skenaarion testitapaus varmistaa ohjelman toiminnan odotetulla tavalla, negatiivisen skenaarion testitapauksella testataan käyttäjän odottamatonta käyttäytymistä ja virheellisiä syötteitä.
- Odotettu tulos: Testitapauksen oletettu toiminnallisuus.
- Tulos: Testin tulokseksi kirjoitettiin hyväksytty tai hylätty. Tieto täytettiin vasta testauksen jälkeen.

Testitapaukset olivat seuraavat:

- "As a user I want to create a new recipe". (1, pos.). Odotettu tulos: Haetaan token kirjautumalla, lisätään uusi resepti rajapintakutsulla, varmistetaan reseptin pätevyys kutsun palautuksesta.
- “As a user I want to delete my recipe”. (2, pos.). Odotettu tulos: Haetaan token kirjautumalla, poistetaan ennestään luotu resepti rajapintakutsulla, varmistetaan reseptin poisto uudella kutsulla.
- “As a visitor I can’t create a recipe”. (1, neg.). Odotettu tulos: Tehdään rajapintakutsu reseptin luomiseen, varmistetaan virheilmoitus kutsun palautuksesta.

- “As a user I can’t delete other’s recipe”. (3, neg.). Odotettu tulos: Haetaan token kirjautumalla, tehdään rajapintakutsu reseptin poistamiseen, varmistetaan virheilmoitus kutsun palautuksesta.
- “As a user I want to modify my recipe”. (1, pos.). Odotettu tulos: Haetaan token kirjautumalla, muokataan ennestään luotua reseptiä rajapintakutsulla, varmistetaan reseptin muokkaus kutsun palautuksesta.
- “As a user I want to comment on recipe”. (1, pos.). Odotettu tulos: Haetaan token kirjautumalla, luodaan kommentti ennestään luotuun reseptiin rajapintakutsulla, varmistetaan kommentin luonti kutsun palautuksesta.
- “As a user I want to rate a recipe”. (1, pos.). Odotettu tulos: Haetaan token kirjautumalla, luodaan arvostelu ennestään luotuun reseptiin rajapintakutsulla, varmistetaan arvostelun luonti kutsun palautuksesta.
- “As a user I want to delete my comment”. (1, pos.). Odotettu tulos: Haetaan token kirjautumalla, poistetaan ennestään luotu resepti rajapintakutsulla, varmistetaan reseptin poisto uudella kutsulla.
- “As a user I can’t delete other’s comment”. (2, neg). Odotettu tulos: Haetaan token kirjautumalla, tehdään rajapintakutsu kommentin poistamiseen, varmistetaan virheilmoitus kutsun palautuksesta.
- “As an admin I want to delete other’s comment”. (2, pos). Odotettu tulos: Haetaan token kirjautumalla, tehdään rajapintakutsu kommentin poistamiseen, varmistetaan kommentin poisto kutsun palautuksesta.
- “As an admin I want to delete other’s recipe”. (2, pos.). Odotettu tulos: Haetaan token kirjautumalla, poistetaan toisen käyttäjän ennestään luoma resepti rajapintakutsulla, varmistetaan reseptin poisto uudella kutsulla.
- “As a user I want to remove my rating”. (1. pos.). Odotettu tulos: Haetaan token kirjautumalla, poistetaan ennestään luotu arviointi rajapintakutsulla, varmistetaan arvioinnin poisto uudella kutsulla.

Testitapausten suunnittelu oli tärkeä osa testausprosessia. Testitapausten määrittely vaatii rajapinnan tuntemista ja osaamista, joiden avulla pystyy tunnistamaan asioita, joissa virheitä tulee helpoiten ja useimmiten. Testitapausten määrittely vaatii myös järjestelmällisyyttä ja luovuutta, joiden avulla pystyy täydentämään testitapauksiin asioita, joita esimerkiksi käyttäjällä saattaisi tulla vastaan ohjelman eri käyttövaiheissa.

Koska testitapaukset oli määritelty jo etukäteen, se helpotti testauksen toteutusta, virheiden todentamista ja mahdollista korjausta sekä uudelleen testausta

korjauksen jälkeen. Testitapausten käsittelyä helpottaisi, jos testitapaukset olisi esitetty taulukossa eikä listana, niin kuin ne on esitetty tässä tapaustutkimuksessa. Testitapausten tuloksia on hankala, ja oikeastaan turha, pitää erikseen manuaalisesti päivitettyinä, sillä testin tulos saattaa muuttua päivittäin riippuen esimerkiksi rajapinnan kehityksestä, ja ne löytyvät Robot Frameworkin luomasta raportista.

Haastavaa testitapausten kirjoittamisessa oli määritellä testit riittävän kattaviksi ja tunnistaa ohjelman käyttöä estävät merkittävimmät virhetilanteet. Kaikkein oleellisimmat toiminnallisuudet olivat tässä tapaustutkimuksessa kuitenkin selkeitä, sillä ohjelma oli pieni ja yksinkertainen. Ohjelmistokehityksessä on hyvä luoda testitapauksia jo kehitysvaiheessa. Testitapausten kirjoittamisessa kannattaa hyödyntää myös ohjelman vaatimus- ja määrittelydokumentteja sekä käyttöohjeita, mutta niitä ei ollut tämän tapaustutkimuksen ohjelmasta olemassa.

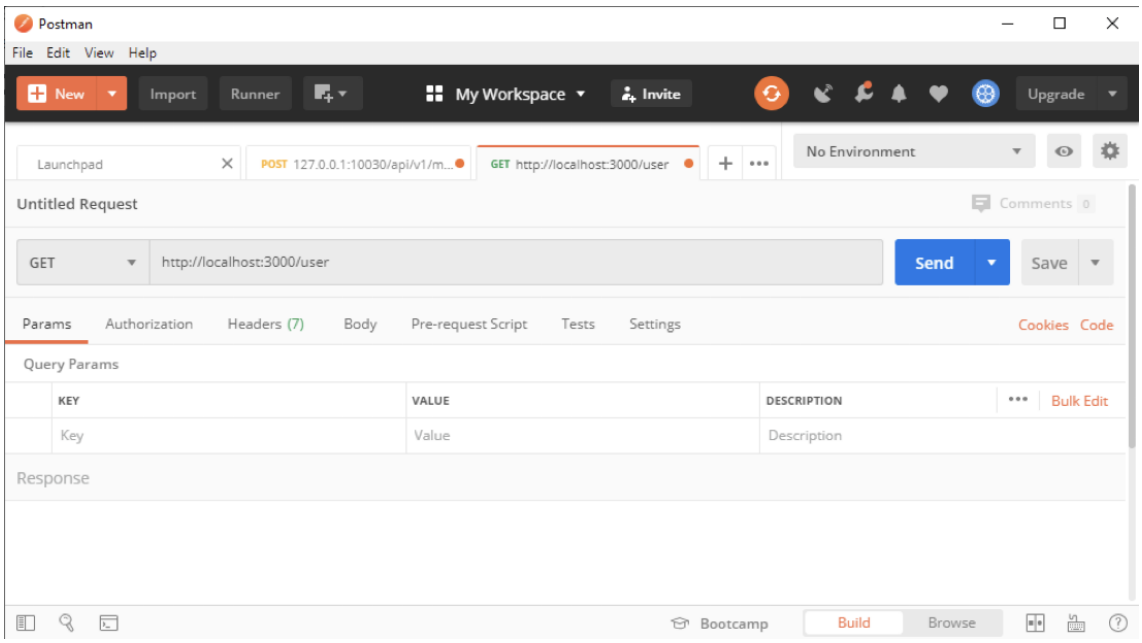
7.3 Testaus ja testien luonti

Manuaalinen testaus tehtiin Postmanillä ja testien luonti Robot Frameworkillä testitapaus kerrallaan. Manuaalista testausta tehtiin robot-testien luonnin yhteydessä rinnakkain helpottamaan testien kirjoittamista.

Manuaalinen testaus Postmanilla

Rajapinnan manuaalinen testaus tehtiin Postmanilla, jolla suoritettiin rajapintaan HTTP-kutsuja. Postman valittiin testauksen apuvälineeksi sen helppokäyttöisyyden vuoksi. Manuaalisella testauksella varmistettiin testitapausten toimivuus ja se, miten kutsut luodaan testitapauksissa.

Manuaalisessa testauksessa kutsu muodostettiin Postman-työkaluun käsin testitapauksen mukaan. GraphQL-pyynnöt ovat pääosin POST-tyyppisiä, mutta myös GET-kutsut voivat toimia rajapinnasta riippuen. Kuvassa 5 on esitetty GET-tyyppisen HTTP-palvelupyynnön rakentuminen Postmanissa; pyynnöllä haetaan lista kaikista palvelun käyttäjistä.



Kuva 5: Rajapinnan kutsu Postmanista (34).

Rajapintakutsut palauttavat JSON-mallisen rakenteen. Esimerkkikoodissa 1 näkyy GraphQL-rajapintaan tehdyn kutsun palautuksen rakenne.

```
{
  "data": { ... },
  "errors": [ ... ]
}
```

Esimerkkikoodi 1. Kutsun palautuksen rakenne GraphQL-rajapinnasta.

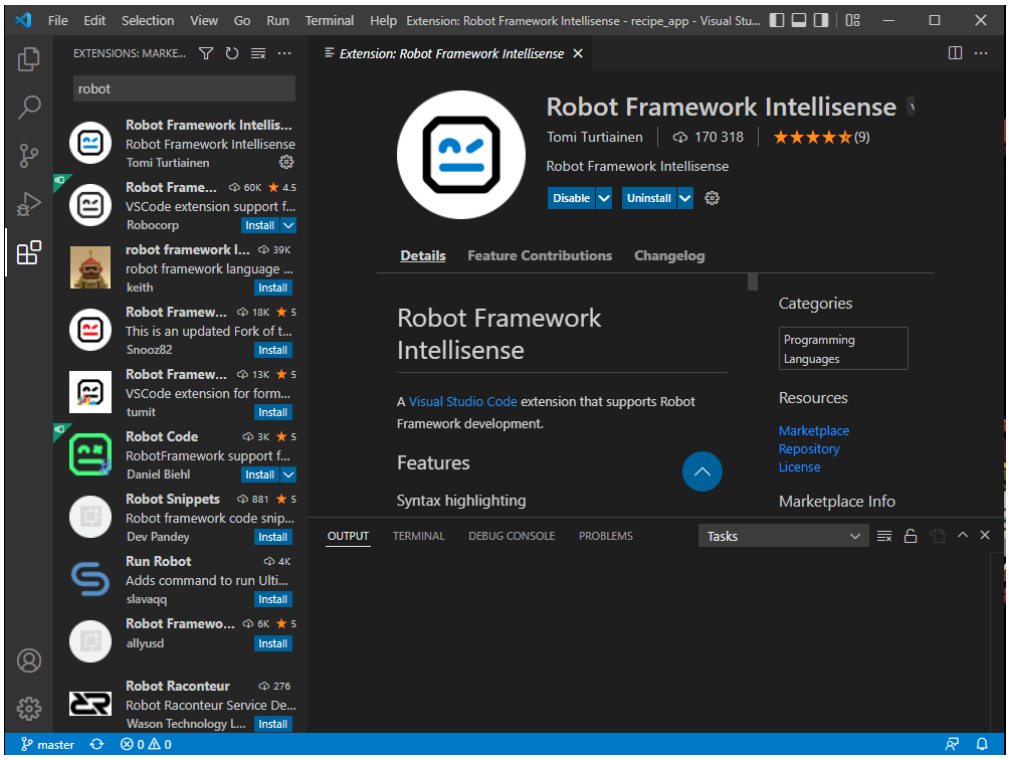
Tärkeää manuaalisessa testauksessa, kuten myös testauksen automatisoinnin seuraavissa vaiheissa, on se, että kun jokin testitapaus ei mene testauksesta läpi, se täytyy aina testata uudelleen ohjelmakoodin korjauksen tai muokkauksen jälkeen, ettei ohjelmaan jäisi aikaisemmin testattua tai uutta virhettä. GraphQL-rajapinnan kanssa on muistettava pitää kutsut POST-tyyppisinä, vaikka olisi tekemässä muokkaus- tai poistotoimintoa.

Testien luonti Robot Frameworkillä

Testit luotiin Robot Frameworkiin testitapauksittain manuaalisen testauksen yhteydessä. Robot Framework valittiin, koska se oli ennestään tuttu ja

yksinkertainen käyttää. Robot-testit luodaan omaan tiedostoon, jotka ovat helppoja ajaa ja myöhemmin automatisoida.

Testien kirjoittamisessa käytettiin Robot Frameworkin Visual Studio Code -editoria (integrated development environment, IDE), johon sai lisättyä eri laajennuksia (extension) Robot Frameworkille ja muille mahdollisesti tarvittaville työkaluille, kuten Pythonille. Kuvassa 6 on näkymä Robot Frameworkin lisäosan latauksesta Visual Studio Codeen.



Kuva 6. Näkymä Visual Studio Coden lisäosavalikosta (35).

Testitapaukset on mahdollista ryhmitellä testijoukoiksi (test suite) esimerkiksi testattavan toiminnon tai osion mukaan. Testitapaukset voi ajaa testijoukkona samasta tiedostosta, mutta myös yksittäisinä testeinä. Testijoukossa yksi testin tulos voi olla syötteenä seuraavalle testille. Tätä kannattaa kuitenkin välttää, jotta testit toimivat myös yksitellen. Testidatan alustamista ja siistimistä käsitellään sivulla 37: miten se toimii käyttäen teardown- ja setup-vaihtoehtoja.

Testien luontiin Robot Frameworkissä käytetään .robot-päätteisiä tiedostoja. Testien ja avainsanojen syntaksi on Pythonin kaltaista, eli rivit eivät pääty puolipisteeseen ja sisennykset ovat tärkeitä. Robot Frameworkissa on muistettava käyttää kahta tai useampaa välilyöntiä avainsanojen ja muuttujien välissä. Testien ja avainsanojen syntaksi on helposti ymmärrettävää ja suoraviivaista.

Esimerkkikoodissa 2 on esitetty Robot Frameworkin neljä eri osiota:

- Settings-kohdan alle listataan asetukset. Asetuksissa voidaan tuoda muuttujia, resurssitiedostoja ja kirjastoja. Asetuksissa voidaan myös määrittää metadata testisarjalle ja testeille.
- Variables-kohdan alle listataan muuttujia. Muuttujat voi myös lisätä omaan Python-tiedostoon (.py) ja lisätä sen joko asetuksissa tai testien ajokomennossa.
- Test Cases -kohdan alle lisättiin itse testitapaukset, jotka koostuvat avainsanoista.
- Keywords-kohdan alle lisätään avainsanat, jotka ovat helposti verrattavissa funktioihin tai metodeihin. Pythonilla voi tarvittaessa tehdä funktioita, joita voi käyttää Robot Frameworkissa.

```
*** Settings ***
Documentation     Test Suite for recipe endpoints.
Library           SeleniumLibrary
Library           RequestsLibrary
Library           JSONLibrary
Library           Collections
Resource          CommonKeywords.robot
Variables         Variables.py
Force Tags        recipe_test

*** Variables ***
${TEST_VAR}        Example variable

*** Test Cases ***
User Creates Recipe
    [Documentation]    As a user I want to create a recipe.
    [Tag]    not-ready
    [Teardown]    Remove recipe    ${RECIPE_JSON}
    ${token} =  Fetch Token  ${USR}  ${PWD}
    Log    ${token}
    ${response} =  Create Recipe  ${token}  ${RECIPE_JSON}
    Verify Recipe Created Successfully  ${response}  ${RECIPE_JSON}

# Testitapauksia luodaan tähän lisää ylemmän testin tapaan

*** Keywords ***
Verify Status Code Successful
    [Arguments]    ${status}
    Should Be Equal  ${status}  200

# Avainsanoja luodaan tänne lisää
```

Esimerkkikoodi 2. Robot Frameworkin neljä eri osiota.

Testien alustaminen ja siivoaminen onnistuu tarvittaessa suorittamalla yksi tai useampia avainsanoja setup- tai teardown-vaihtoehdoilla. Teardownissa määritetään testin lopettava toiminto, jossa voidaan esimerkiksi muokata testissä käytetty data alkuperäisiin arvoihin tai poistaa testissä luotua dataa. Setup suoritetaan taas testin alussa, jossa voidaan alustaa ympäristö testiä varten.

Robot-testit suoritetaan terminaalista ajamalla käynnistyskomento. Komennossa käytetään eri vaihtoehtoja (options) rajaamaan, mitä testejä ajetaan, lisätään muuttujia ja määritetään, mihin palautus tulee, kuten

- -v (variable, muuttujia)
- -i (include, mitä testejä ajetaan)
- -e (exclude, mitä testejä ei ajeta)
- -t (test, valitaan ajettava testi nimen avulla)
- -o (output, mihin testitulos tallennetaan).

Esimerkkikoodissa 3 on esitetty käynnistyskomento, jolla voi ajaa osan testeistä. Käynnistyskomennossa muuttujan voi antaa suoraan tai Python-tiedostossa esimerkkikoodin 3 tapaan. Hyvä tapa hyödyntää erillistä Python-tiedostoa on käyttää sitä muuttujille, jotka on tarkoitettu tietylle testiympäristölle.

```
robot -i recipe_test -e not-ready -v USR:”username” -v PWD:”password”
-v env_variables.py recipe_app/tests/
```

Esimerkkikoodi 3. Käynnistyskomento Robot Frameworkissä.

Esimerkkikoodin 3 käynnistyskomento ajaa kaikki testit, joilla on tagi “recipe_test”, mutta ei “not-ready”. Komennossa lisätään sisäänkirjautumista varten muuttujat USR ja PWD, joilla saadaan token rajapinnan kutsuja varten sekä env_variables.py-tiedosto testiympäristön muuttujia varten. Komennossa mainitaan viimeisenä testien lokaatio.

Rajapinnan testausta ei voi seurata samalla tavalla kuin käyttöliittymän testausta, koska rajapintaa testataan kutsumalla sitä eikä sille ole omaa käyttöliittymää. Testin etenemistä voi tosin seurata terminaalista pääpiirteittäin tai Robot Frameworkin luomasta lokista.

Robot Framework oli helppo ottaa käyttöön, ja verkosta löytyi hyvä dokumentaatio. Testejä oli helppo työstää Robot Frameworkin yksinkertaisen ja helppolukuisen syntaksin ja lokien kanssa. Robot Frameworkin avainsanojen ja muuttujien väliin kirjoitettaviin välilyönteihin tottui melko nopeasti. Testien luonnissa oli tärkeää nimetä testitapaukset ja avainsanat selkeästi ja ymmärrettävästi, koska ne näkyivät myös lokissa. Tapaustutkimuksen testit olivat yksinkertaisia, eivätkä ne kaivanneet erillisiä funktioita. Monimutkaisempiin tapauksiin sai käytettyä valmiiksi tehtyjä avainsanoja kirjastoista.

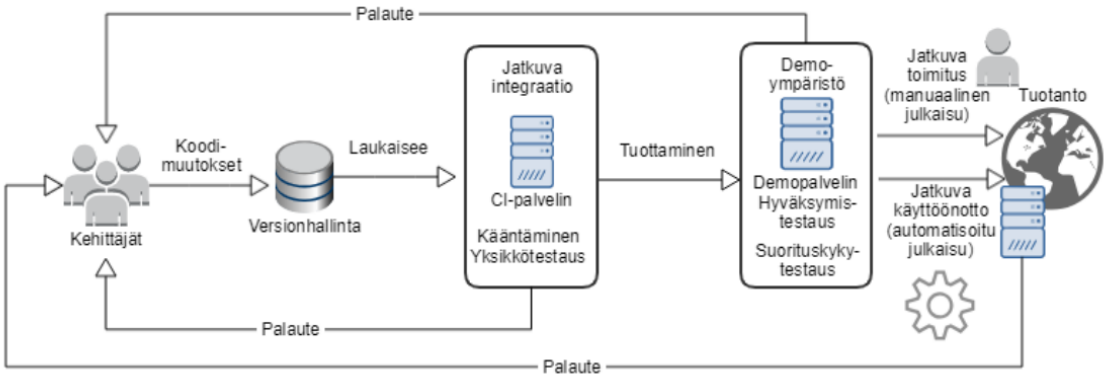
Tässä tapaustutkimuksessa testaus tehtiin myöhäisessä vaiheessa rajapinnan kehitystä. Ihannetapauksessa testausta olisi tehty jo aikaisemmin. Tällöin virheet olisi ollut mahdollista huomata nopeammin, ja ne olisi pystytty korjaamaan heti ohjelmakehityksen yhteydessä, mikä säästäisi isommassa projektissa aikaa ja resursseja. Rajapinnan testauksessa on mahdollista tehdä kutsuja liian

nopeasti peräkkäin, jolloin kutsut eivät välttämättä mene läpi ja kutsu saattaa palauttaa virheilmoituksen.

7.4 Testauksen automatisointi Jenkinsillä

Robot Frameworkillä luodut testit automatisoitiin seuraavaksi Jenkinsillä. Se on avoimen lähdekoodin laajasti käytetty jatkuvan integroinnin työkalu, jonka saa ladattua sen omilta sivuilta. Ladattu tiedosto on itsenäinen Java-ohjelma, joka on helppo ottaa käyttöön millä vain laitteella. Jenkins on hyvin laajennettavissa monien lisäosien ansiosta. Jos Robot-testit ovat Git-versiohallintajärjestelmässä, asentamalla sille laajennuksen Jenkinsistä voi ajaa testien uusimman version. Tapaustutkimuksen testit olivat kuitenkin omalla koneella ja ajettiin koneella olevaa ympäristöä vasten.

Kuvassa 7 on selkeästi kuvattu jatkuvan integraation, toimituksen ja käyttöönoton välinen suhde, jota kohti tapaustutkimuksessa pyritään. Tapaustutkimuksessa tehdyt testit kuuluvat hyväksymistestaukseen, joka tehdään ennen julkaisua.



Kuva 7. Jatkuva integraatio, toimitus ja käyttöönotto (39, s. 2; 40, s. 6).

Jenkins käyttää Java-versioita 8 tai 11 ja sen voi asentaa joko palveluna tai .war-tiedostona (Web Application Resource). Tapaustutkimuksessa käytettiin .war-tiedostoa, joka käynnistetään terminaalista esimerkkikoodin 4 mukaan.

```
C:\sijainti\>Java –jar Jenkins.war
```

Esimerkkikoodi 4. Jenkinsin käynnistyskomento.

Jenkinsiin lisättiin vapaamuotoisia (freestyle project) tehtäviä (job), jotka konfiguroitiin ajamaan eri testausalueiden tehtäviä. Näin pienessä projektissa tämä ei olisi kuitenkaan tarpeen. Jenkins vaati testien ajoon Robot Framework -laajennuksen.

Jobin konfigurointi jakautuu seuraaviin osioihin:

- Yleinen (General), sisältää tehtävän kuvauksen ja muita määrittelyjä, kuten tehtävien samanaikaisen ajon.
- Lähdekoodin hallinta (Source Code Management), sisältää tässä tapauksessa testien sijainnin.
- Tehtävän ajon laukaisijat (Build Triggers), sisältää määrittelyt, milloin tehtävä ajetaan, kuten kellonajan tai tapahtuman.
- Ympäristön rakentaminen (Build Environment), sisältää testiympäristön hallintaan liittyviä asetuksia.
- Rakentaminen (Build), sisältää käynnistyskomennon.
- Ajon jälkeiset tapahtumat (Post-build Actions), sisältää tapahtumat, joita tehdään buildin jälkeen.

Jenkinsin hyviä puolia olivat sen maksuttomuus, helppo asentaminen ja graafinen käyttöliittymä. Lisäksi Jenkinsissä oli mahdollista käyttää useita erilaisia liitännäisiä. Tehtävien (job) konfigurointi saattaa olla hankalaa ensimmäistä kertaa sitä tekevälle. Testien suorittamisen jälkeen sai raportit ja lokit, joista oli helppo varmistaa testien kulun ja mahdolliset virheet.

Tässä tapaustutkimuksessa testit olivat osa CI/CD:tä eli jatkuvaa integrointia ja julkaisua, vaikka rajapinta oli pieni. Jatkuva integrointi on erityisen hyödyllistä isompien rajapintojen testausautomatisoinnissa, jolloin virheitä havaitaan ja niitä voi korjata nopeammin. Manuaalisia testejä on silti tarpeen tehdä vielä automatisoinnin jälkeenkin.

8 Yhteenveto

Insinöörityössä tutkittiin verkko-ohjelmointirajapintoja, niiden testausta ja testauksen automatisointia. Asiaa tutkittiin ensin lähdeaineiston ja sitten tapaustutkimuksen avulla.

Ohjelmointirajapinta on ohjelmien ja sovellusten välinen yhtymäkohta, joka mahdollistaa niiden välisen tiedonsiirron. Ohjelmointirajapintojen avulla käyttäjät saavat tietoa rajapintaa tarjoavan palvelun ominaisuuksista ja toiminnallisuuksista ja niiden käytöstä, mikä hyödyttää ohjelmistojen käyttöä ja jatkokehitystä. Avoimen verkkorajapinnan hyötyjä ovat sen avoimuus, käyttöönotettavuus ja testattavuus muille käyttäjille, vaikkakin se on myös heikkous, sillä avoimuuden vuoksi se on altis haitallisille käyttäjille. Lisäksi rajapinnan ylläpito, dokumentointi ja testaus vievät resursseja, vaativat ohjelmointiosaamista ja aiheuttavat siten kustannuksia.

Rajapinta luodaan tapauskohtaisesti sovitulla ohjelmointikielellä. Verkkopalveluja toteutetaan nykyisin yhä useammin www-palveluna, joka käyttää tiedonsiirrossa HTTP-protokollaa. XML- ja JSON-muotoja suositaan vastauksen muotona, koska ne on helppo muokata muille sovelluksille ja ohjelmille sopiviksi. Rajapinnan kutsun tulee olla yhteensopiva rajapinnan tietorakenteen kanssa. CRUD-perustoiminnot vastaavat HTTP-menetelmiä POST, GET, PUT ja DELETE.

Ohjelmointirajapinnoissa, niiden testauksessa ja automatisoinnissa voidaan hyödyntää useita eri teknologioita ja työkaluja. Verkkorajapintojen tiedonsiirtoa standardoidaan erilaisilla protokollilla, joista yleisimpiä ovat SOAP ja REST. Viime vuosina ovat yleistyneet myös GraphQL ja gRPC. Manuaalisen testauksen työkaluja ovat esimerkiksi Robot Framework, Postman ja Wireshark. Automatisoinnin työkalut auttavat testien säännöllisessä ajossa ilman testaajan vuorovaikutusta. Automaatiotestauksen työkaluja ovat esimerkiksi Jenkins ja GitLab CI/CD.

Rajapinnan testauksen tavoitteena on toiminnallisuuden varmistaminen ja virheiden etsiminen. Ohjelmiston testaus on prosessi tai prosessien sarja, joka on suunniteltu varmistamaan, että ohjelmiston koodi tekee sen, mitä se on suunniteltu tekemään. Testaus on keskeinen vaihe ohjelmiston kehittämistä ja koko sen elinkaaren hallintaa. Täysin kattavaa testausta on kuitenkin mahdotonta tehdä, sillä se edellyttäisi kaikkien ohjelmistoon ja testaukseen vaikuttavien asioiden huomioimista, mikä on mahdotonta.

Ohjelmistotestausta voi toteuttaa ja tarkastella erilaisilla tasoilla ja menetelmillä. Perinteinen tapa luokitella testausmenetelmiä on jakaa ne mustalaatikko- ja lasilaatikkotestaukseen. Mustalaatikkotestaus on käyttäjän käyttäytymiseen perustuva, dataohjattu testaus, kun taas lasilaatikkotestaus on ohjelmiston sisäistä rakennetta testaava, logiikkaohjattu testaus. Suositeltavaa on käyttää testauksessa molempia menetelmiä, mitä kutsutaan hybriditestausstrategiaksi.

Testaussuunnitelman tekeminen on keskeinen osa testausprosessin hallintaa. Suunnitelmassa määritellään testauksen toteutus, resurssit ja aikataulu. Onnistunut ohjelmiston testaus perustuu myös testitapausten tunnistamiseen ja määrittelyyn. Testaussuunnitelman ja testitapausten määrittely auttavat varmistamaan, että testaus priorisoituu oikeisiin asioihin.

Manuaalisessa testauksessa testaaja itse testaa ohjelmiston toimivuuden suorittamalla testitapauksia ilman testausjärjestelmiä. Manuaalisessa testauksessa testaaja pystyy helpommin havaitsemaan visuaalisia ohjelmistovirheitä ja tarkistamaan pieniä ohjelmistossa tehtyjä muutoksia. Manuaalinen testaus on myös edellytys testauksen automatisoinnille.

Testauksen automatisointi voi olla järkevää pidemmällä tähtäimellä. Automatisoidussa testauksessa testejä voidaan helposti toistaa, minkä vuoksi testauksesta saadaan kattavampaa, nopeampaa ja kustannustehokkaampaa. Automaatiotestaus tulee saada osaksi rajapinnan jatkuvaa integraatiota ja kehittämistä, jolloin virheitä pystytään ehkäisemään ja havaitsemaan nopeammin ja siten varmistamaan ohjelmiston laatutaso.

Automatisoidun testauksen elinkaari on syklinen prosessi. Se alkaa testauksen automatisoinnin päätöksellä, testityökalun hankinnalla ja automatisoituun testaukseen perehtymisellä siirtyen sen jälkeen testauksen suunnitteluun ja kehitykseen, testien suorittamiseen ja hallintaan sekä testausprosessin tarkasteluun ja arviointiin.

Oikein toteutettuna testauksen automatisointi tuo monia hyötyjä, esimerkiksi se parantaa testattavan ohjelman luotettavuutta ja testauksen laatua sekä vähentää testaukseen käytettävää manuaalista työtä. Testauksen automatisoinnissa on myös heikkouksia. Testaukseen voi kohdistua odotuksia, jotka ovat epärealistisia. Automatisointi ei vastaa asetettuja odotuksia esimerkiksi silloin, jos testaajat eivät ole kokemukseensa perustuen tyytyväisiä testaukseen tai testaustyökaluun, testaukset on organisoitu huonosti, dokumentointi on puutteellista tai testit eivät löydä virheitä odotetusti.

Insinöörityön tapaustutkimuksessa suunniteltiin ja toteutettiin verkko-ohjelmointirajapinnan testaus ja testauksen automatisointi. Testattava ohjelmointirajapinta oli yksinkertainen ja pieni GraphQL-rajapinta verkkosovelluksesta, jossa käyttäjät voivat luoda reseptejä, kommentoida niitä ja antaa niistä arvosteluja. Rajapinta oli ohjelmoitu JavaScript-ohjelmointikielellä, ja se käytti MongoDB-tietokantaa. Manuaalinen testaus tehtiin Postmanilla, testit luotiin Robot Frameworkillä ja testien automatisointi Jenkinsilla. Testaus on osa hyväksymistestausta, ja se toteutettiin mustalaatikkotestauksella, joka on käyttäjän käyttäytymiseen perustuva testaus. Testit olivat osa CI/CD:tä eli jatkuvaa integrointia ja julkaisua.

Tapaustutkimus toteutettiin vaiheittain. Ensin selvitettiin testattavan sovelluksen GraphQL-rajapinta, sitten laadittiin testaussuunnitelma ja määriteltiin testitapaukset, tehtiin manuaalinen testaus Postmanilla ja toteutettiin suunnitelmaan ja testitapauksiin perustuva testaus Robot Frameworkillä sekä lopuksi toteutettiin testauksen automatisointi Jenkinsillä.

Testaussuunnitelman osalta päähavainto oli suunnitelman laatimisen hyödyllisyys, sillä sen avulla pystyi selventämään testauksen kohteen, menetelmät ja

rajoitteet. Haasteellista suunnitelman laatimisessa oli saada hahmotettua testauksen kokonaisuus jo ennen varsinaisen testauksen käynnistymistä. Testitapausten suunnittelu oli keskeinen osa testausprosessia ja siksi tärkeä vaihe. Testitapausten määrittely etukäteen helpotti testauksen toteutusta. Haastavaa oli pitää tapausten määrä rajattuna, mutta silti kattavana, ja tunnistaa ohjelman käyttöä estävät merkittävimmät virhetilanteet.

Tapaustutkimuksessa varsinaisen testauksen osalta yksi päähavainto oli, että testauksen on tärkeää olla mukana jo heti ohjelmakehityksen yhteydessä, jolloin virheitä pystytään huomaamaan ja korjaamaan nopeammin, mikä säästäisi aikaa ja resursseja. Toinen päähavainto oli, että sekä manuaalisessa testauksessa, testien luomisessa että testauksen automatisoinnissa oli tärkeää, että löydetyt virheet korjataan ja että ohjelmakoodin korjauksen ja muokkauksen jälkeen testataan testitapauksen toimivuus uudelleen. Testauksen automatisoinnin osalta on myös muistettava, että vaikka testaus on automatisoitu, on manuaalisia testejä välttämätöntä tehdä jossain määrin edelleen.

Lähteet

1 Ohjelmointirajapinta. 2021. Verkkoaineisto. Termipankki. <https://termipankki.fi/tepa/fi/haku/ohjelmointirajapinta>. Luettu 10.12.2021.

2 Avoimen rajapinnan määritelmä. 2014. Verkkoaineisto. Avoin rajapinta. <http://avoinrajapinta.fi>. Päivitetty 11.10.2014. Luettu 13.12.2021.

3 Nolle, Tom. Application program interface (API). Verkkoaineisto. <https://searchapparchitecture.techtarget.com/definition/application-program-interface-API>. Luettu 30.12.2021.

4 API-vallankumous on täällä! Ohjelmistorajapinnat muuttavat yritysten liiketoimintamalleja. 2019. Verkkoaineisto. Lowell. <https://profit.lowell.fi/api-vallankumous-on-taalla-ohjelmistorajapinnat-muuttavat-yritysten-liiketoimintamalleja>. Luettu 11.5.2022.

5 Dustin, Elfriede; Rashka, Jeff & Paul, John. 1999. Automated Software Testing. Introduction, Management and Performance. Addison-Wesley.

6 Levin, Guy. 2019. Top 7 REST API Security Threats. Verkkoaineisto. DZone. < https://dzone.com/articles/top-7-rest-api-security-threats>. Luettu 10.5.2022.

7 Ohjelmointikielet vertailussa. Verkkoaineisto. Vincit Oy. <https://www.vincit.fi/fi/ohjelmointikielet-vertailussa/>. Luettu 8.4.2022.

8 Top 6 Main Uses of Java in Real World. Verkkoaineisto. CodeAvail. <https://www.codeavail.com/blog/main-uses-of-java/#top-6-main-uses-of-java-in-real-world>. Luettu 8.4.2022.

9 JavaScript. 2022. Verkkoaineisto. Mozilla. <https://developer.mozilla.org/en-US/docs/Web/JavaScript>. Luettu 10.5.2022.

10 Tietoliikenteen perusteet 2. Verkko-ohjelmointi ja sovelluksen rajapinta. 2020. Verkkoaineisto. Helsingin yliopisto. <https://tietoliikenteen-perusteet-2-20.mooc.fi/osa-2/5-verkko-ohjelmointi-rajapinta>. Luettu 13.12.2021.

11 Fielding R & Reschke J. 2014. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230. Verkkoaineisto. Internet Engineering Task Force (IETF). <http://www.ietf.org/rfc/rfc7230.txt>. Luettu 10.5.2022.

12 What is an API? 2017. Verkkoaineisto. Red Hat, Inc. <https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces>. Luettu 10.5.2022.

13 HTTP-tilakoodit - mitä ne tarkoittavat? 2020. Verkkoaineisto. Webhosting. <https://webhosting.de/fi/http-status-codes-what-do-they-mean/>. Luettu 16.1.2022.

14 What is CRUD? Verkkoaineisto. Codecademy. <https://www.codecademy.com/article/what-is-crudz>. Luettu 15.1.2022.

15 Pajukangas, Jukka. 2020. REST-rajapinnan testauksen kattavuuden mittaaminen. Diplomityö. Oulun yliopisto. Jultika-arkisto.

16 SOAP vs. REST: A Look at Two Different API Styles. 2021. Verkkoaineisto. Upwork Staff. <https://www.upwork.com/resources/soap-vs-rest-a-look-at-two-different-api-styles>. Luettu 29.3.2022.

17 What is SOAP? Verkkoaineisto. Tutorialspoint. >https://www.tutorialspoint.com/soap/what_is_soap.htm>. Luettu 29.3.2022.

18 Johnson, Tom. What is a REST API? Verkkoaineisto. <https://idratherbewriting.com/2015/08/28/all-about-rest-apis/>. Luettu 11.5.2022.

19 Mikä on GraphQL ja miten se eroaa REST-rajapinnoista (API)? Verkkoaineisto. Symfony Finland. <https://symfony.fi/artikkeli/mika-on-graphql-ja-miten-se-eroaa-rest-rajapinnoista-api>. Luettu 15.1.2022.

20 The OpenAPI Specification (OAS). Verkkoaineisto. OpenAPI Initiative. <https://github.com/OAI/OpenAPI-Specification>. Luettu 10.5.2022.

21 Robot Framework. Verkkoaineisto. Robot Framework. <https://robotframework.org>. Luettu 29.3.2022.

22 What is Postman API Test. Verkkoaineisto. Encora. <https://www.encora.com/insights/what-is-postman-api-test>. Luettu 7.4.2022.

23 What is Postman? Verkkoaineisto. Postman, Inc. <https://www.postman.com>. Luettu 29.3.2022.

24 Wireshark Frequently Asked Questions. Verkkoaineisto. Wireshark. <https://www.wireshark.org/faq.html#_what_is_wireshark>. Luettu 7.4.2022.

25 Test Automation vs Manual Testing. Picking the Right Balance. Verkkoaineisto. Testim, a Tricentis company. <https://www.testim.io/blog/test-automation-vs-manual-testing/>. Luettu 10.5.2022.

26 Jenkins User Documentation. Verkkoaineisto. Jenkins. <https://www.jenkins.io/doc/>. Luettu 30.3.2022.

27 Jenkins Press Information. Verkkoaineisto. Jenkins.io. <https://www.jenkins.io/press/>. Luettu 30.3.2022.

28 GitLab - CI/CD. Verkkoaineisto. Tutorialspoint. <https://www.tutorialspoint.com/gitlab/gitlab_ci_cd.htm>. Luettu 30.3.2022.

29 GitLab CI/CD. Verkkoaineisto. GitLab. <https://docs.gitlab.com/ee/ci/>. Luettu 30.3.2022.

30 Myers, Glenford J.; Badgett, Tom & Sandler, Corey. 2012. The Art of Software Testing. Third Edition. John Wiley & Sons.

31 Spillner, Andreas; Linz, Tilo & Schaefer, Hans. 2014. Software Testing Foundations: A Study Guide for the Certified Tester Exam. Rocky Nook.

32 What is End-to-End (E2E) Testing? All You Need to Know. Verkkoaineisto. <https://katalon.com/resources-center/blog/end-to-end-e2e-testing>. Luettu 7.4.2022.

33 Testaussanasto - ohjelmistotestauksen tärkeimmät termit selitettynä. Verkkoaineisto. ProjectTOP. <https://projecttop.com/testaussanasto/>. Luettu 30.3.2022.

34 Jäntti, Marko. 2003. Testitapausten suunnittelu UML-mallinnuksen avulla. Pro gradu -tutkielma. Kuopion yliopisto. UEF eRepository-arkisto.

35 What is Unified Modeling Language (UML)? Verkkoaineisto. Visual Paradigm. <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/>. Luettu 30.3.2022.

36 Fowler, Martin. 2006. Continuous integration. Verkkoaineisto. Thoughtworks. <https://martinfowler.com/articles/continuousIntegration.html>. Luettu 10.5.2022.

37 Postman, 2022. Sovellus. Postman, Inc.

38 Visual Studio Code. 2022. Sovellus. Microsoft.

39 Shahin, Mojtaba; Zahedi, Mansooreh; Babar, Muhammad Ali & Zhu, Liming. 2018. An empirical study of architecting for continuous delivery and deployment. Verkkoaineisto. arXiv. <(https://arxiv.org/pdf/1808.08796.pdf>. Luettu: 10.5.2022.

40 Koriseva, Lauri. 2018. Jatkuvan toimituksen käyttöönotto ohjelmistoprojektissa. Diplomityö. Tampereen teknillinen yliopisto. Trepo-arkisto.