OAMK OULUN AMMATTIKORKEAKOULU

Riku Ylikangas

RAJAPINNAN UUDELLEEN SUUNNITTELU JA KEHITTÄMINEN LARAVELIN AVULLA

RAJAPINNAN UUDELLEEN SUUNNITTELU JA KEHITTÄMINEN LARAVELIN AVULLA

Riku Ylikangas
Opinnäytetyö
Kevät 2024
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, ohjelmistokehityksen suuntautumisvaihtoehto

---

Tekijä(t): Riku Ylikangas
Opinnäytetyön nimi: Rajapinnan uudelleen suunnittelu ja kehittäminen Laravelin avulla
Työn ohjaaja(t): Teemu Korpela
Työn valmistumislukukausi ja -vuosi: Kevät 2024 Sivumäärä: 41

Työn tilaajana toimii Anicare Oy, joka valmistaa porojen paikantamista varten laitteita. Opinnäytetyön tavoitteena oli jatkaa aloitetun rajapinnan kehittämistä siihen pisteeseen, että sen voisi ottaa käyttöön yrityksen käyttämän rajapinnan tilalle. Rajapintaa käyttävät niin sovellus kuin paikannuslaite, mutta opinnäytetyössä keskityttiin sovelluksen käyttämiin rajapintakutsuihin. Uuden rajapinnan tekemiseen käytettiin Laravel-ohjelmistokehystä ja rajapinta tehtiin REST-tyyliseksi.

Opinnäytetyössä käydään yleisesti läpi rajapintojen periaatteita. UML:ää käytetään mallinnuskielenä, kun laaditaan kaavioita rajapinnan ja siihen liittyvien asioiden kuvaamiseen. Työssä käydään läpi dokumentointityökaluja sekä kerrotaan testauksesta ja rajapinnan kehittämisen vaiheet suunnittelusta toteutukseen ja testaukseen.

Lopputuloksena on toimiva rajapinta, joka on laajasti testattu ja dokumentoitu. Tavoitetta ei saavutettu, koska rajapinta vaatii vielä jonkin verran kehittämistä, että sen voisi ottaa käyttöön käytössä olevan rajapinnan tilalle. Työn aikana tulleita ideoita sekä rajapinnan vaatimia toimenpiteitä käydään läpi opinnäytetyön Jatkokehitys-osiossa.

Asiasanat: ohjelmistorajapinta, ohjelmistotestaus, ohjelmistokehitys

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology, Option of Software Development

---

Author(s): Riku Ylikangas
Title of thesis: API redesign and development using Laravel
Supervisor(s): Teemu Korpela
Term and year when the thesis was submitted: Spring 2024
Number of pages: 41

The goal of this thesis was to continue developing the application programming interface for Anicare to the point it would have been able to replace the application programming interface that was in use. The application programming interface was developed using the Laravel framework. The created application programming interface is REST application programming interface.

Thesis contains some theory about application programming interfaces and REST. UML was used as modelling language when creating diagrams to describe application programming interface. There is also some knowledge about documentation tools and comparison of different tools. The process of development is also written from planning to documentation and testing.

The result of thesis is functional application programming interface that is documented and tested. The goal was not reached, because it would still need some work before it could be used to replace the application programming interface in use. These steps are mentioned in the chapter that goes through the next steps of development.

Keywords: application programming interface, testing, software development

SISÄLLYS

1 JOHDANTO 6
2 RAJAPINNAT JA TIETOMALLIT 7
2.1 Verkon välityksellä kommunikointi 7
2.2 Tietokannan tietomallit 7
2.3 Ohjelmistorajapinnat 10
2.4 REST-rajapinnat 11
3 YRITYKSEN KÄYTTÄMÄT RAJAPINNAT 14
3.1 Rajapinnan käyttötarkoitukset 14
3.2 Rajapinnan havaitut ongelmat ja puutteet 15
4 KÄYTETYT TEKNOLOGIAT JA METODIT 17
4.1 Ohjelmistokehykset 17
4.2 Dokumentointiohjelmien vertailu ja valinta 18
4.3 Testaustyökalut ja käytännöt 21
5 RAJAPINNAN KEHITTÄMISEN VAIHEET 23
5.1 Uuden rajapinnan suunnittelu 23
5.2 Rajapinnan toteutus 24
5.3 Rajapinnan dokumentointi 28
5.4 Rajapinnan testaus 31
6 JATKOKEHITYS 35
7 JOHTOPÄÄTÖKSET 37
LÄHTEET 39

1 JOHDANTO

Opinnäytetyön tarkoituksena oli kehittää ja dokumentoida uutta ohjelmointirajapintaa. Työn tilaajana toimi Anicare Oy. Anicare Oy on vuonna 2017 perustettu oululainen yritys, jonka pääasiallinen tulonlähde on poroille asennettava paikannuslaite. Laitteen avulla saadaan ilmoitus kuolleista poroista sekä poroja pystytään seuramaan, missä ne menevät. (1.)

Yritys sai alkunsa, kun toimitusjohtaja Aki Marttila, joka on myös poronomistaja, sai idean porojen paikannuslaitteesta. Markkinoilla ei ollut mitään vastaavaa laitetta, joka toimisi samalla tavalla, ja muutenkin porotalous oli jäämässä unohduksiin teknologian ja valmistajien suhteen. Ideaa mietittiin jonkin aikaa, kunnes tuntui, että saatavilla olevat teknologiat mahdollistaisivat laitteen toteutuksen, ja näin Anicare Oy sai alkunsa. Tuotekehityksessä on käytetty toimitusjohtajan omaa kokemusta sekä otettu huomioon asiakkaiden palautteet, jotta laite ja sovellus olisivat käyttäjäystävällisiä ja täyttäisivät poronhoidon tarpeet. (2.)

Työn tavoitteena oli tehdä valmiiksi ohjelmointirajapinta, jonka tekemisen olin aloittanut yrityksessä yritysprojektien ja harjoitteluiden aikana. Tavoitteena on, että uusi ohjelmointirajapinta toimii samalla tavalla kuin yrityksen sen hetkinen rajapinta ja sen pitää myös olla dokumentoitu ja täysin testattu, että sen voi ottaa käyttöön. Yrityksen sen hetkisen rajapinnan sisältämät ongelmat ja puutteet korjataan uuteen rajapintaan. Dokumentointiin panostetaan, koska se on jäänyt osittain tekemättä tai sitten se on hyvin vähäinen. Rajapinnan testaamiseen ei ole ollut minkäänlaisia testejä olemassa, joten sen toimivuutta ei pystynyt testaamaan. Sen takia uutta rajapintaa varten tehdään kattavat ja hyvät testit, joiden avulla voidaan testata, että rajapinta toimii oletetulla tavalla.

Tarve uuteen rajapintaan johtuu yrityksen tämänhetkisestä rajapinnasta. Rajapinnan koodin rakenne ei ole hyvä ja koodausta on tehty kopioimalla alkuperäisestä koodista, joten kaikki virheet ja ongelmat ovat periytyneet sieltä. Koodia ei ole kovin paljoa kommentoitu sekä dokumentaatio on osittain puutteellista tai olematonta. Joissakin toiminnoissa on havaittu jonkinlaisia odottamattomia ongelmia. Tämä rajapinta olisi vaatinut täydellisen läpi käymisen, korjaamisen, testaamisen ja dokumentoinnin. Tämä kuitenkin olisi vienyt hyvin paljon aikaa, joten uuden rajapinnan tekeminen oli järkevämpi ratkaisu. Uusi rajapinta myös mahdollistaa tulevaisuudessa monien eri asioiden tekemisen, verrattuna vanhaan toteutukseen.

2 RAJAPINNAT JA TIETOMALLIT

2.1 Verkon välityksellä kommunikointi

Rajapintojen keskusteluun käytetään asiakas-palvelinsuhdetta. Siinä palvelin toimii resurssien tai palveluiden tarjoajana ja asiakas palvelun pyyntöjen tekijänä. Kommunikaatio toimii pyyntö-vastaus-kaavalla. Palvelin kuuntelee kokoajana pyyntöjä asiakkailta. Pyyntö voi olla tietojen hakua tai jokin toiminto, kuten esimerkiksi käyttäjän kirjautuminen. Palvelin lähettää vastauksen asiakkaan lähettämään pyyntöön. Asiakas-palvelinsuhteen etuna on se, että kaikki tiedot ovat yhdessä paikassa. Se helpottaa ongelmien ratkaisemisessa sekä tiedostojen hallinnassa. Ongelmana voi olla, että jos palvelimelle tulee liikaa pyyntöjä, yhteys voi hidastua tai palvelin voi kaatua. Jos palvelimella tapahtuu vika, koko verkko katkeaa. (3.)

Toinen yleinen kommunikaatioon käytetty malli on P2P, peer-to-peer eli vertaisverkko. Vertaisverkko poikkeaa asiakas-palvelinmallista siten, että jokainen verkon jäsen toimii sekä asiakkaana ja palvelimena. Jäsenet ovat suoraan vuorovaikutuksessa keskenään ja voivat jakaa resursseja, kuten esimerkiksi prosessointikykyä tai levytilaa. Mitä enemmän jäseniä liittyy, sitä enemmän verkon kapasiteetti kasvaa. Vaikka joku jäsen lähtee tai kohtaa jonkin virheen, verkko kuitenkin jatkaa toimintaa. Vertaisverkon ongelmina on turvallisuus. Jäsenet voivat levittää haitallisia tiedostoja tai hyökätä toisen jäsenen kimppuun, jotka johtavat mahdollisiin turvallisuus riskeihin. (4.)

2.2 Tietokannan tietomallit

Tietomalli kuvaa, miten tietoa esitetään ja miten siihen pääsee käsiksi tietojärjestelmässä. Tietomallit ovat abstraktioita jo olemassa olevista tai esitetyistä tietokannoista. Tietomalleja on olemassa kolme päätyyppiä: käsitteellinen, looginen ja fyysinen. Käsitteellinen tietomalli on korkean tason malli, joka esittää tiedon ja suhteiden käsitteet teknologiasta riippumattomalla tavalla. Looginen tietomalli on luotu tietyn tyyppisestä tietokannan hallintajärjestelmästä, kuten esimerkiksi relaatiotietokannasta. Fyysinen tietomalli on luotu noudattaen tietokannan hallintajärjestelmän ominaisuuksia ja rajoituksia, kuten esimerkiksi MySQL:n ominaisuuksia ja rajoituksia. (5.)

Tietomallit ovat eläviä dokumentteja, jotka kehittyvät tarpeiden mukaan. Niillä on tärkeä rooli suunniteltaessa tietotekniikan arkkitehtuuria ja strategiaa. Tietomallinnus helpottaa tietojen suhteiden näkemistä ja ymmärtämistä tietokannassa. Se myös vähentää virheiden määrää kehityksessä sekä parantaa suorituskykyä. (6.)

Tietomallinnusta voidaan tehdä esimerkiksi UML:n avulla, esimerkiksi käyttäen luokkakaaviota, vaikka se ei sisällä määrittelyä tietomallinukseen. UML on lyhenne, joka tulee englannin kielen sanoista unified modeling language. Se kehitettiin jo 1990-luvulla ja on vielä nykyäänkin standardi notaatio kehittäjille. Se yksinkertaistaa monimutkaisuutta sekä parantaa työn laatua. On olemassa kaksi päätyyppiä: rakennekaaviot sekä käyttäytymiskaaviot. Näiden päätyyppien alle lukeutuu vielä useita muita kaavioita. Rakennekaaviot esittävät ohjelman tai järjestelmän staattisen rakenteen sekä eri abstraktio- ja toteutustasot. Esimerkkinä rakennekaavioista on luokkakaavio. Käyttäytymiskaaviot keskittyvät ohjelmistojärjestelmän tai prosessin dynaamisiin näkökohtiin. Nämä kaaviot esittävät järjestelmän toiminnallisuudet ja korostavat, mitä järjestelmässä pitää tapahtua. Esimerkkinä käyttäytymiskaavioista on käyttötapauskaavio. (7.)

Normalisoinnin avulla voidaan vähentää tiedon redundanssia ja poistaa lisäys-, päivitys- ja poistohäiriöt. Normalisoinnissa on sääntöjä, joiden avulla suuret taulut jaetaan pienempiin tauluihin ja ne linkitetään relaatioiden avulla. Tarkoitus on poistaa toistuvaa tietoa ja varmistua, että tieto on tallennettu loogisesti. (8.)

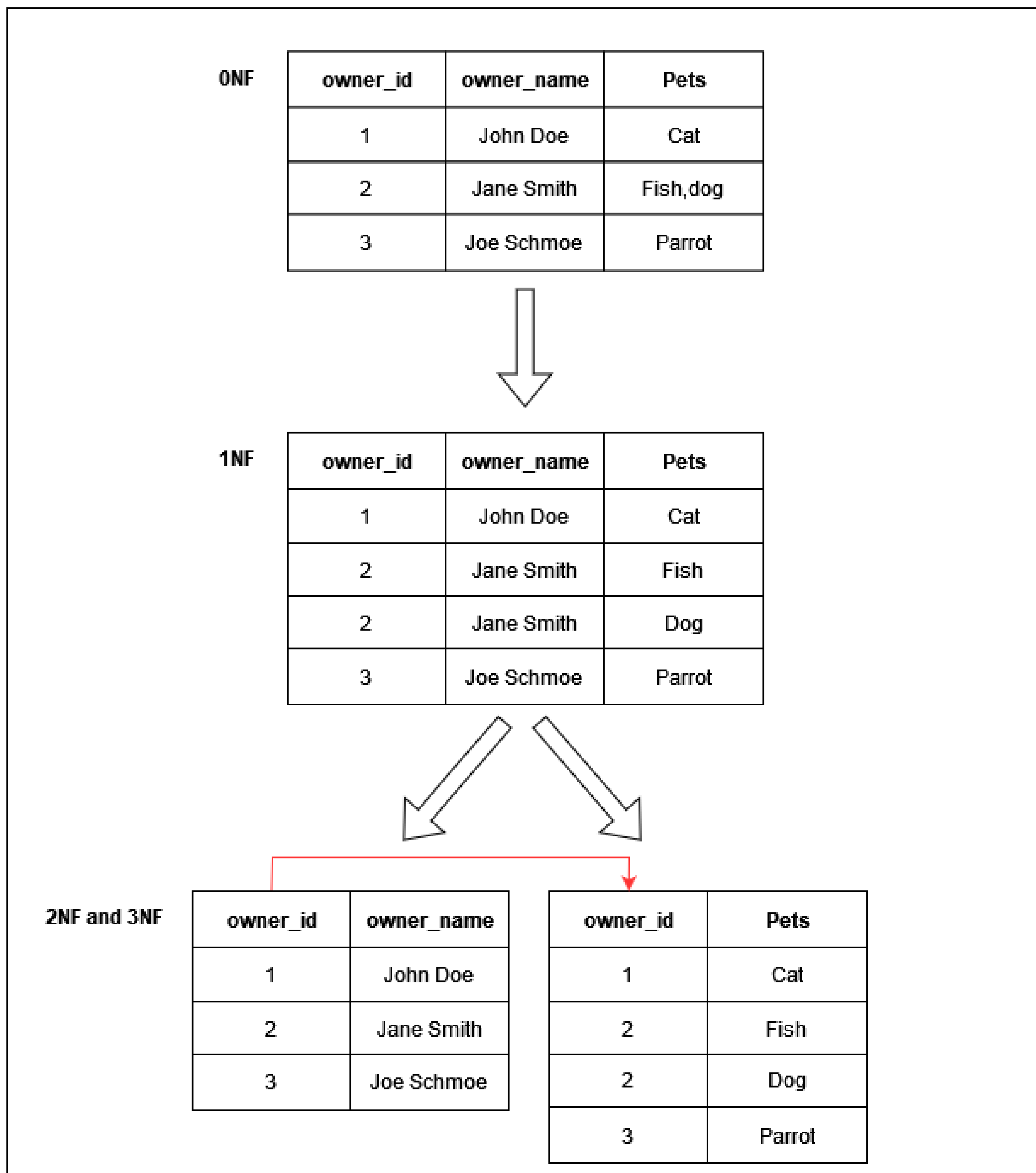
Lisäyshäiriö tapahtuu silloin, kun yritetään lisätä tietoja tietokantaan. Tällainen tilanne tapahtuu, jos tiedon lisääminen vaatii toisen tiedon lisäämistä, vaikka ne eivät suoranaisesti liity toisiinsa. Päivityshäiriö tapahtuu, kun jonkin tiedon päivittämisessä huomaa, että pitää päivittää useita eri kohtia. Poistohäiriö tapahtuu, kun yhden tietuen poisto vaatii useamman tietueen poistamisen. Nämä ongelmat ratkeavat suorittamalla normalisoinnin. (9.)

Normalisoinnille on olemassa kuusi muotoa, mutta parhaan tuloksen yleensä saavuttaa jo kolmannella muodolla. Ensimmäisen muodon sääntö vaatii, että jokaisessa taulun solussa on vain yksi arvo eli solussa ei voi olla listoja ja jokainen tietue on uniikki. Toisen muodon sääntö vaatii, että taulu on ensimmäisessä muodossa ja on yksi primääriavain, joka ei ole riippuvainen ehdokasavaimen relaatiosta. Primääriavain on sarake, jonka avulla tietue tunnistetaan uniikiksi. Sen ominaisuuksia ovat, että se ei voi olla arvoltaan NULL, arvon pitää olla uniikki, sarakkeen arvoja tarvitsee harvoin vaihtaa ja sille pitää antaa arvo, kun uusi tietue lisätään. Kolmannen muodon sääntö vaati,

että taulu on toisessa muodossa ja taulussa ei ole transitiivisia riippuvuuksia. Tämä tarkoittaa sitä, että jos muokataan jotakin ei-avainsaraketta, niin sen seurauksena toisen ei-avainsarakkeen arvo muuttuu. (8.)

Kolmannen muodon jälkeen tulee Boyce-Codd-muoto, jonka sääntö vaatii, että taulu on kolmannessa muodossa ja siinä on vain yksi avainehdokas. Sitten tulee neljäs muoto, jonka sääntö vaatii, että taulu on Boyce-Codd-muodossa ja siinä ei ole kuin yksi moniarvoinen riippuvuus. Viides muoto on viimeinen muoto ja sen sääntö vaatii, että taulu on neljännessä muodossa ja taulua ei pysty hajottamaan pienempiin tauluihin ilman tiedon katoamista. Kolmanteen muotoon normalisointi on hyvin yleistä, mutta on vielä mahdollisia tapauksia, että on olemassa mahdollisia häiriöitä, joiden takia pitää sitten normalisoida enemmän. (8.)

KUVA 1 on esimerkki tietokantataulusta eri normalisoinnin tiloissa. Alkutilanne on 0NF, joka tarkoittaa, että taulua ei ole normalisoitu. Jotta taulu saadaan ensimmäiseen muotoon, sen pets-sarakkeen arvojen listoista luodaan uusia tietueita. Toiseen muotoon normalisointiin taulusta luodaan kaksi taulua, joista toinen kuvaa omistajia ja toinen kuvaa lemmikkejä ja viiteavaimena toimii owner_id-sarake, jota käytetään omistajan tunnisteena. Samalla saavutetaan kolmas normalisointi, koska ei ole olemassa transitiivisia riippuvuuksia. Näin saavutetaan normalisoinnin lopputulos, jossa tietokannassa olevat tiedot eivät ole toistuvia ja tietokantarakenne on selkeämpi, mikä helpottaa tietokannan tietojen sekä relaatioiden ymmärtämistä.



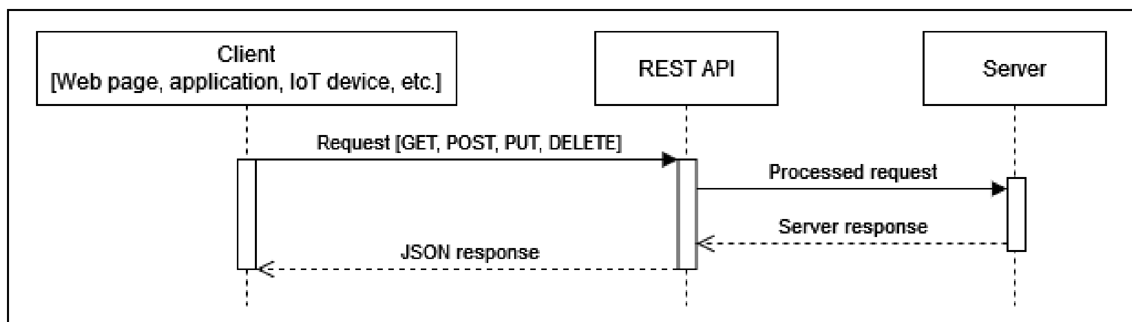
KUVA 1. Taulu eri normalisointimuodoissa. Lähtökohta on normalisoimaton eli 0NF ja siitä seuraava on ensimmäinen normalisointi eli 1NF. Kun ensimmäisestä normalisoinnista siirrytään toiseen normalisointiin, luodaan kaksi eri taulua, joilla on relaatio owner_id-sarake, jota punainen nuoli kuvaa. (8.)

2.3 Ohjelmistorajapinnat

Ohjelmistorajapintojen avulla sovellukset ja järjestelmät pystyvät kommunikoimaan keskenään. Esimerkkinä rajapinta voi olla välikäsi sovelluksen ja tietokannan välillä, ja kun sovelluksesta tehdään pyyntö hakea tietoa, kutsu menee rajapinnalle ja rajapinta hakee tiedon tietokannasta ja palauttaa tiedot sovellukselle. (10.)

Rajapinnan käytössä on useita eri hyötyjä. Samaa rajapintaa voidaan käyttää useiden eri sovelluksien ja ohjelmien kanssa. Rajapintaa voidaan käyttää sen sijaan, että tehtäisiin tietokantakyselyitä suoraan tietokantaan. Sen avulla voidaan tarkistaa annetut arvot ja tehdyt kyselyt, jotta käyttäjät eivät voi tehdä kyselyitä, jotka esimerkiksi voisivat vahingoittaa tietokantaa. Rajapinnan avulla voidaan varmistua, että käyttäjällä on oikeudet päästä tietokantaan. (11.)

Rajapinnan toimintaa havainnollistamisessa käytetään monesti esimerkkitapausta ravintolasta, jossa asiakas tekee tilauksen tarjoilijalle, joka sitten toimittaa asiakkaan pyynnön keittiöön, jossa tilaus valmistetaan, ja sitten tarjoilija toimittaa sen asiakkaalle. Tässä esimerkissä tarjoilija on rajapinta asiakkaan ja keittiön välillä ja ottaa vastaan asiakkaiden tilaukset, eli pyynnöt ja toimittaa ne keittiölle. Keittiö kuvastaa palvelinta, joka suorittaa asiakkaan pyynnön ja lähettää pyynnön vastauksen rajapinnalle, jonka kautta se kulkeutuu asiakkaalle (KUVA 2). (12.)



KUVA 2. UML-sekvenssikaavio REST-rajapinnan toiminnasta

On olemassa erityyppisiä rajapintoja, jotka määräytyvät rajapinnan julkisuuden mukaan. On olemassa yksityisiä rajapintoja, joita esimerkiksi yritykset käyttävät omissa sovelluksissa ja joihin ei ole pääsyä ulkopuolisille. Sitten on olemassa julkisia rajapintoja, jotka ovat saavavilla julkisesti. Näiden avulla on mahdollista käyttää esimerkiksi yrityksen tietoja tai palveluita omissa ympäristöissä. Julkisia rajapintoja voidaan rajoittaa esimerkiksi rajoittamalla tehtyjen pyyntöjen määrää tai laittamalla joitakin toimintoja maksumuurin taakse. (12.)

2.4 REST-rajapinnat

Yleensä puhutaan REST-rajapinnasta, jolla tarkoitetaan rajapintaa, joka on tehty noudattaen REST-määritelmää. REST on lyhenne, joka tulee englannin kielen sanoista representational state

transfer. Roy Fielding loi määritelmän vuonna 2000 väitöskirjassaan. Määritelmässä on kerrottu periaatteet, joita rajapinnan pitää noudattaa, että se on REST-rajapinta. Näitä periaatteita on viisi pakollista:

1. Yhtenäinen rajapinta
2. Asiakas-palvelinmalli
3. Tilattomuus
4. Välimuistin käytettävyys
5. Kerroksittainen rakenne.

Ensimmäinen periaatteen mukaan samaa kohdetta koskevien pyyntöjen tulee olla samanlaisia riippumatta siitä, mistä pyyntö tehdään ja vastausten ei pidä olla liian suuria, mutta niiden pitää kuitenkin sisältää kaiken, mitä asiakas voisi tarvita. Toisen periaatteen mukaan asiakas ja palvelin ovat täysin riippumattomia toisistaan. Kolmannen periaatteen mukaan rajapinta on tilaton, tarkoittaen, että mitään pyyntöjen tietoja ei tallenneta ja kaikissa pyynnöissä on kaikki vaadittu tieto pyynnön suorittamiseen. Neljännen periaatteen mukaan resursseja pitää olla mahdollista tallettaa välimuistiin, kun se on sallittu. Ja viidennen periaatteen mukaan sillä on kerroksittainen rakenne, tarkoittaen, että pyynnöt voivat mennä usean eri kerroksen kautta. (13.)

REST-rajapinnoista puhuttaessa käytetään usein termiä CRUD. CRUD on lyhenne, joka tulee englannin kielen sanoista create, read, update ja delete. REST-rajapinnat tekevät HTTP-pyyntöjen avulla tavallisia tietokantafunktioita, kuten luovat, lukevat, päivittävät ja poistavat tietoja. Pyyntöjen tekemiseen voidaan käyttää eri metodeja. GET-metodin avulla voidaan hakea tietoa, POST-metodin avulla luodaan uusi olio, PUT-metodin avulla voidaan muokata oliota ja DELETE-metodin avulla voidaan poistaa olio. Myös muita HTTP-metodeja voidaan käyttää, mutta GET, POST, PUT ja DELETE ovat yleisimmät. (13.)

REST-rajapinnan käytössä on monenlaisia hyötyjä. Kun asiakas ja palvelin pidetään erillään, rajapintaa on helppo skaalata. Samalla rajapinnan kehittäminen itsenäisesti on mahdollista. Rajapinta on joustava ja siirrettävissä muille palvelimille migraatioiden avulla. Rajapinnat ovat lisäksi nopeita ja kevyitä käyttää, koska käytössä on HTTP-standardi, joka tukee monia eri muotoja, kuten esimerkiksi JSON. Tämä mahdollistaa sen, että rajapintaa voidaan käyttää niin sovellusten kanssa kuin myös IoT-laitteiden sekä moniin muihin tarkoituksiin. (14.)

Kuitenkaan REST-rajapintojen käyttö ei ihan onnistu haasteitta. Rajapinnan päätepisteiden pitää olla yhtenäisiä ja johdonmukaisia, ja toimintoja lisätessä päätepisteiden määrät ja yhdistelmät kasvavat. Suurten koodikantojen kanssa sekä useiden kehittäjien välillä voi olla hankalaa säilyttää johdonmukaisuutta. Suuria päivityksiä tehdessä julkaistaan monesti uusi versio rajapinnasta ja pidetään vanha versio vielä käyttäjille saatavilla. Tämä lisää työmäärä, kun pitää ylläpitää useampaa eri rajapintaa. Voi olla hankalaa päättää, milloin vanhan rajapinnan ylläpito lopetetaan ilman, että se häiritsisi asiakkaita. (14.)

Vaikka puhutaan, että rajapinta käsittelee annettuja arvoja, kuitenkin tietokannan päälogiikka on palvelimessa. Se huolehtii tietokannan relaatioista sekä varmistaa, että arvot sopivat sarakkeen määrittelyyn. Virheiden tai häiriöiden sattuessa se ilmoittaa rajapinnalle. Rajapinta ottaa vastaan palvelimen palauttamat tiedot ja käsittelee ne siten, että asiakkaan on helpompi ymmärtää esimerkiksi, miksi jokin pyyntö ei onnistu.

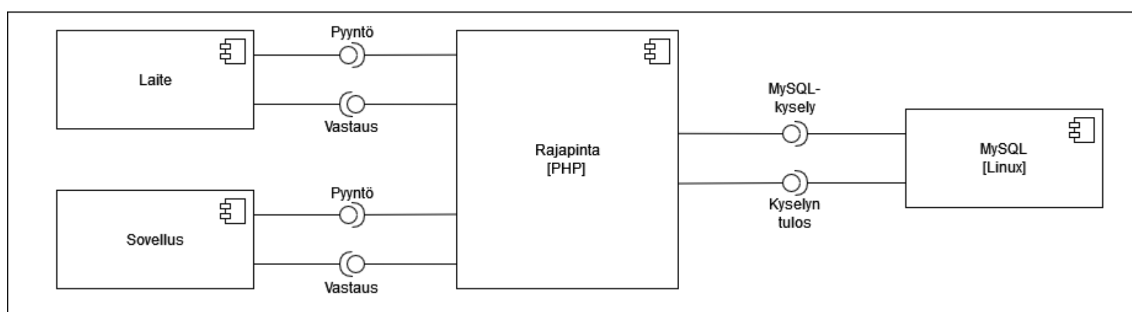
3 YRITYKSEN KÄYTTÄMÄT RAJAPINNAT

Yrityksen käytössä oleva rajapinta toimii palvelimella, johon on asennettu Apache-verkkopalvelin ja kaikki kutsut on tehty PHP:n avulla. Yrityksen käyttämä tietokanta on MySQL, joka on myös asennettu palvelimelle. Käytössä on kokoelma ohjelmia, josta käytetään lyhennettä LAMP eli Linux, Apache, MySQL ja PHP (15).

Suunnitteilla on ollut vertaisverkon käyttämistä laitteiden kesken, mutta on tiedostettu, että sen käyttäminen vaatisi muutoksia niin laitteisiin kuin rajapintaan.

3.1 Rajapinnan käyttötarkoitukset

Yrityksellä on käytössä oma rajapinta, jota laitteet ja sovellukset käyttävät kommunikoimiseen tietokannan kanssa (kuva 3). Yritys käyttää nykyistä rajapintaansa tiedon hakemiseen ja muuttamiseen tietokannasta. Opinnäytetyö keskittyy enemmän sovelluksen käyttämiin rajapintakutsuihin, joiden avulla poroja pystytään paikantamaan. Rajapinnan avulla tietokannasta haetut arvot pystytään palauttamaan sovellukselle paremmassa muodossa. Monella kutsulla on tapana muuttaa tietoja useammasta taulusta, joten tämä nopeuttaa ja selkeyttää sitä.



KUVA 3. UML-komponenttikaavio sovelluksen ja laitteen rajapinnan käytöstä

Rajapinnan käyttö mahdollistaa sen, ettei sovelluksella tarvitse olla suoraa yhteyttä tietokantaan. Se lisää aina tietoturvaa ja rajapintaan voidaan lisätä omia autentikointeja, mikä lisää suojausta. Rajapinnan käyttö mahdollistaa pyyntöjen tekemisen ja käyttämisen muuallakin kuin vain sovelluksessa. Ongelma tilanteissa rajapinta auttaa virheiden löytämisessä, koska yleensä on loki, mihin rajapinta kirjoittaa virhetilanteissa.

3.2 Rajapinnan havaitut ongelmat ja puutteet

Nykyisessä rajapinnassa on muutamia ongelmia, joita uutta rajapintaa kehittäessä pyritään korjaamaan. Virhetilanteiden sattuessa rajapinta kirjoittaa virheet lokiin, minne järjestelmä kirjoittaa kaikki tapahtumat. Onneksi kuitenkin lokiviestiin on lisätty tiedoston nimi ja rivi, missä virhe on tapahtunut.

Koodissa on paikoitellen turhia tietokantakyselyitä, joita voisi osan jättää pois ja osan voisi yhdistää. Kutsuja vähentämällä saadaan vähennettyä liikennettä rajapinnan ja tietokannan välillä. Uudessa rajapinnassa osan kyselyistä voi jättää pois, kun käytetään istuntoa, johon tietoja voi tallentaa ja sitten noutaa tarvittaessa.

Koodissa ei ole kunnolla määritetty, mitkä arvot ovat pakollisia. Sen takia on mahdollista tehdä kutsuja, joista puuttuu tärkeitä arvoja, mutta rajapinta ei kuitenkaan palauta minkäänlaista virhettä.

Joissakin kohdissa koodia rajapinta tekee kutsuja rajapinnassa oleviin kutsuihin. Tämä tietysti vähentää saman koodin uudelleen kirjoittamista, mutta koodia lukiessa vaikeuttaa ymmärtämistä, kun ohjelmoijan pitää mennä katsomaan koodia toisesta tiedostosta, josta rajapinta sitä kutsuu.

Rajapinnan lähettämät vastaukset eivät juuri kerro kovin paljon kutsun toiminnasta. Yleensä rajapinta palauttaa merkkijonon ”true” tai ”false” (kuva 4). Virhetilanteiden sattuessa koodissa kutsutaan die-funktiota ja palautetaan merkkijono ”error”. Yleensä kun rajapinta palauttaa merkkijonon ”true”, se tarkoittaa, että kutsu on toiminut kuten pitääkin ja päässyt loppuun. Palautettu merkkijono ”false” taas yleensä tarkoittaa, että esimerkiksi jokin tarkistus ei ole mennyt läpi, mutta kuitenkaan virhettä ei ole tapahtunut. Näissä tapauksissa pitääkin sitten käydä koodista tarkistamassa, missä tilanteissa rajapinta palauttaa kyseisen merkkijonon.

KUVA 4. Esimerkki Postmanin avulla lähetetyn kutsun palauttamasta vastauksesta

Rajapinnan vastauksiin ei ole määritelty HTTP-tilakoodia, joten olettavasti kaikki kutsut palauttavat tilakoodin 200, joka tarkoittaa, että kutsu on onnistunut. Palvelin palauttaa virheitä esimerkiksi, jos annetulla osoitteella ei löydy mitään, palautetaan koodi 404. Uuteen rajapintaan on tarkoitus lisätä palautettavat HTTP-tilakoodit siten, että niiden avulla voidaan helposti päätellä tehtyjen kutsujen tila.

Rajapinnan koodeissa on joitakin virheitä, joiden takia osa kutsuista ei toimi oletetulla tavalla. Koodi on myös hiukan vaikeasti luettavaa ja koodissa on käytetty vähän kommentteja. Lisäksi joidenkin rajapinnan kutsujen dokumentointi on todella huono tai olematon. Nämä on onneksi huomioitu ja tullaan korjaamaan uuteen rajapintaan.

Rajapinnan koodit on luotu kopioimalla aiemmin tehtyä koodia ja muokkaamalla se käyttöön sopivaksi. Tämä aiheuttaa sen, että kun alkuperäisessä koodissa on ollut virheitä, ne ovat periytyneet uusiin tehtyihin kutsuihin. Koodin rakenne on myös huono, mikä vaikeuttaa koodin hallintaa.

Rajapinta ei noudata HTTP 1.1-protokollaa (16). Siinä määritellään, että GET-metodin avulla haetaan vain tietoa ja tulos on aina sama, vaikka pyyntö tehtäisiin kerran tai sata kertaa. Joissakin kutsuissa rajapinta GET-metodin avulla päivittää tietoja tietokantaan. Tätä ei tulla kuitenkaan korjaamaan uuteen rajapintaan, koska sen on oltava toiminnaltaan sama kuin nykyinen rajapinta.

4 KÄYTETYT TEKNOLOGIAT JA METODIT

Uuden rajapinnan luominen aloitettiin tyhjältä pöydältä, joten rajapintaa varten hankittiin uusi palvelin ja sen tekemiseen käytetään ohjelmistokehystä. Tämä helpottaa työskentelyä, koska ohjelmistokehykset sisältävät komponentteja ja kirjastoja ohjelmistokehitykseen (17).

Dokumentointia varten piti löytää uusi parempi työkalu, jonka avulla se onnistuu helposti. Tähän asti yritys oli käyttänyt Google Docsia dokumenttien kirjoittamiseen.

Rajapintaa pitää testata, että vältytään mahdollisilta virheiltä, jotka pahimmassa tapauksessa voisivat vaarantaa tietoturvaa. Kun rajapinta on testattu, sovelluskehittäjien on helpompi ottaa rajapinta käyttöön, kun tiedetään, että rajapinta toimii kuten pitääkin. (18.)

4.1 Ohjelmistokehykset

Ohjelmistokehykset ovat alustoja ohjelmistokehitykselle. Ohjelmistokehykset sisältävät valmiita kirjastoja, joiden avulla voidaan kehittää erilaisia ohjelmistoja. Ohjelmistokehyksiä on helppo muokata omaan käyttöön sopivaksi lisäämällä tai poistamalla toimintoja. Verkkokehykset ovat käytetyimpiä ohjelmistokehyksiä. Niiden avulla sovellusten ja sivustojen kehittäminen on nopeampaa. Suosittuja verkkokehyksiä ovat esimerkiksi Django ja Laravel. (19.)

Django on vuonna 2005 julkaistu Python-pohjainen verkkokehys. Sillä on avoin lähdekoodi ja se on ilmainen käyttää. Siihen on saatavilla yli 10 000 pakettia, joista esimerkiksi löytyy käyttäjien autentikointia ja lomakkeiden validointia. (20.)

Laravel on vuonna 2011 julkaistu PHP-pohjainen verkkokehys, jonka kehitti Taylor Otwell (21). Laravelin kehitys on jatkunut ja uusin saatavilla oleva versio on versio 10, joka julkaistiin helmikuussa 2023 (22). Laravel sisältää valmiiksi kaiken tarvittavan, kun halutaan kehittää uutta rajapintaa. Kaikki Laravelin versiot ovat hyvin laajasti dokumentoituja sekä Laravel on hyvin suosittu, joten tietoa löytyy netistä paljon.

Päädyimme käyttämään Laravelia rajapinnan luomiseen. Laravelissa on MVC-arkkitehtuuri, joka on lyhenne englannin kielen sanoista model-view-controller, joka tarkoittaa malli-näkymä-kontrolleri. Laravelin käyttö on ilmaista ja se on avoimen lähdekoodin ohjelmistokehys. Laravelin avulla voidaan tehdä kutsuja, joita käytetään tiedon siirtämiseen, tai sitten sillä voi tehdä näkymiä, jotka toimivat kuten verkkosivut. Laravelissa on käytettävissä objekti-relaatiokartoitus, jonka avulla voidaan hallita tietokantaa yksinkertaisemmin ja turvallisemmin. (23.)

Kun rajapinnan tekeminen aloitettiin, Laravelin uusin versio oli 9. Laravelin seuraavasta versiosta, versio 10:stä oli jo jonkinlaista tietoa ja siihen siirryttiin, kun se oli julkaistu.

4.2 Dokumentointiohjelmien vertailu ja valinta

Rajapinnan dokumentointiin löytyy hyvin paljon työkaluja ja tapoja. Dokumentointi on tärkeä osa ohjelmointia, koska sen avulla muiden on helpompi ymmärtää rajapinnan toimintoja ilman, että he vain lukisivat koodia. Kun dokumentointi on tehty kunnolla, uusien ihmisten on helppo jatkaa kehittämistä siitä, mihin on jääty. (24.)

Tehtäväni oli etsiä ja tutkia dokumentointiin tarkoitettuja työkaluja ja niistä listata parhaat ja huonot puolet. Minulla oli neljä työkalua, joita lähdin tutkimaan: Swagger, Stoplight, Visual Paradigm ja Postman. Työkaluja vertaillessani otin huomioon hinnan ja sen, mitä työkalut tarjosivat. Mietin lisäksi työkalun käytännöllisyyttä ja tarvetta. Yhtenä huomioon otettavana asiana tuli tilaajalta, että jos työkalu tullaan vaihtamaan johonkin toiseen, tehty dokumentaatio pitää saada tallennettua itselle ja työkaluun ei pidä syntyä minkäänlaista riippuvuutta. Käydessäni työkaluja läpi kirjoitin kaikista asioita, jotka koin positiivisena tai negatiivisena (TAULUKKO 1). Nämä asiat vaikuttivat työkalun valintaan.

TAULUKKO 1. Työkaluista listatut positiiviset ja negatiiviset asiat

Työkalu	Posiitivistä	Negatiivistä
Google Docs	Tiedostojen helppo jakaminen ja muokkaus. Tiedostojen pilvitallennus Google Driveen. Maksuton käyttö.	Dokumentin päivittäminen työlästä. Ei niin selkeä.

	Dokumenttien linkittäminen.	
Swagger	Visuaalinen työkalu sekä tekstieditori. Dokumentin vienti JSON- tai YAML-tiedostona. Esimerkkikoodin luominen.	Kallis käyttää. Paljon ominaisuuksia, joille ei tarvetta.
Visual Paradigm	Hyvä työkalu kaavioiden ja diagrammien tekoon. Rest API-kaavio, jonka pohjalta osaa luoda dokumentaation. Dokumentin vienti HTML-tiedostona.	Dokumenttien jako työlästä. Projektitiedostoja ei voi avata tai muokata muilla työkaluilla.
Postman	Nopea kutsujen ja parametrien lisäys suoraan tehdyistä kutsuista. Kutsuihin tehdyt muutokset päivittyvät myös dokumenttiin.	Dokumentaatiota ei saa vietyä. Dokumentoitujen kutsujen oltava Postmanissa.
Stoplight	Visuaalinen työkalu sekä tekstieditori. OpenAPI-määritelmä JSON- tai YAML-tiedostona. Aiempi positiivinen kokemus.	Ei osaa luoda esimerkkikoodia.

Kävin läpi aluksi yrityksen käyttämän Google Docsin. Se on tekstinkäsittelyohjelma, joka toimii selaimessa. Sen hyviä puolia ovat se, että tiedostot ovat Google Drivessa, joten niitä on helppo jakaa ja muidenkin tarvittaessa muokata. Sen käyttäminen on myös ilmaista. (25.) Googlen työkalujen avulla luotujen tiedostojen linkittäminen on helppoa ja se nopeuttaa sekä parantaa luettavuutta. Huonona puolena ainakin rajapinnan dokumentointiin on se, että kun tietoa on paljon, dokumentista tulee sekava ja tietoa pitää etsiä.

Ensimmäisenä lähdin tutkimaan työkalu nimeltään Swagger. Swagger alun perin loi avoimen lähdekoodin määritelmän REST-tyylisille rajapinnoille. Myöhemmin tästä tuli nykyäänkin tunnettu OpenAPI-määritelmä. (26.) Swagger sisältää työkaluja esimerkiksi koodin luomiseen OpenAPI-määritelmän pohjalta ja OpenAPI-määritelmän visuaaliseen esittämiseen (27). Swaggerin avulla voi kirjoittaa tai visuaalisesti luoda OpenAPI-määritelmän rajapinnasta. OpenAPI-määritelmän saa helposti vietyä JSON- tai YAML-muodossa.

Visual Paradigm on työkalu, jonka avulla pystyy mallintamaan järjestelmiä, ohjelmia ja suunnitelmia (28). Se sisältää myös REST-rajapinta kaavion, jonka pohjalta pystyy luomaan dokumentaation. Dokumentaation saa vietyä HTML-muodossa, eli se on helppo saada omalle palvelimelle. Tämä oli kaikista muista poiketen työpöytäsovellus, vaikkakin Postman toimii myös selaimessa, mutta on saatavilla työpöytäsovelluksena. Tämä tarkoittaa, että kaikki tiedostot tallennetaan paikallisesti tietokoneelle. Se aiheuttaa lisätöitä esimerkiksi tiedostojen jakoon sekä varmuuskopiointiin. Luodut projektit sisältävät tiedostopäätteen ”vpp”, ja näitä tiedostoja ei voi avata kuin Visual Paradigmin avulla. Tämä luo eräänlaisen riippuvuuden Visual Paradigmin käyttöön.

Postman on alusta rajapintojen tekemiseen ja käyttämiseen (29). Sen avulla on helppo tehdä kutsuja eri rajapintoihin. Se osaa luoda dokumentaatiota tehdyistä kutsuista. Kutsujen palauttamat arvot joko tallennetaan, kun kutsu on tehty tai lisätään esimerkkinä kutsuun. (30.) Postmanin avulla dokumentointi olisi nopeinta, jos kaikki kutsut tullaan lisäämään Postmaniin. Dokumenttia ei kuitenkaan saa mitenkään vietyä Postmanista, eli dokumentointi jää Postmaniin, jos halutaan vaihtaa toiseen työkaluun.

Sitten lähdin tutkimaan Stoplightia. Stoplight on rajapinnan suunnittelu työkalu (31). Stoplight on hyvin saman tyylinen kuin Swagger, mutta verrattaessa halvempi. Dokumentoinnin saa tehtyä visuaalisesti työkalulla tai sitten kirjoittamalla OpenAPI-määritelmää ja sen saa helposti viety JSON- tai YAML-muodossa. Stoplightin käytöstä minulla löytyi jo aiempaa kokemusta, joten osasin sanoa, että se on hyvä työkalu dokumentointiin.

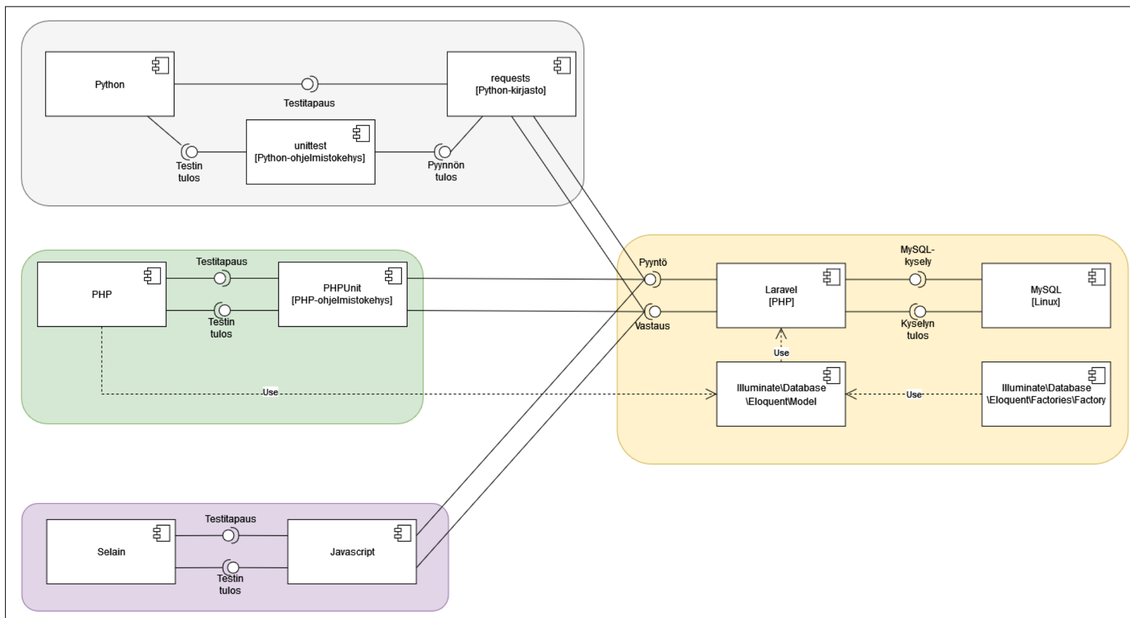
Näistä työkaluista valittiin Stoplight. Se oli hinnaltaan ihan sopiva ja sisälsi kuitenkin vaadittavat ominaisuudet dokumentoinnin laatimiseen. Kustannuksia laskiessa käytettiin esimerkki tapauksena, että olisi viisi käyttäjää, joiden pitäisi voida käyttää työkalua ja jokaiselle käyttäjälle olisi oma tilaus, josta veloitettaisiin kuukausittain. Näiden laskelmien mukaan Swagger olisi tullut kalleimmaksi käyttää. Lisäksi aiempi kokemus työkalusta helpotti valintaa, kun ei tarvinnut alkaa opettelemaan täysin uuden työkalun käyttöä.

Uusi dokumentointityökalu mahdollistaa paremman versiohallinnan dokumentaatioiden välillä. Stoplightissa pystyy luomaan uuden version vanhan version pohjalta ja sitten siihen tekemään muutokset. Muutoksien tekeminen on nopeaa ja helpompaa visuaalisen editorin avulla, kun ei tarvitse niin paljoa kirjoittaa. Dokumentin vienti YAML-muodossa mahdollistaa sen, että halutessaan dokumentoinnin voi julkaista omalla palvelimella.

4.3 Testaustyökalut ja käytännöt

Rajapinnan testaus on tärkeää. Sen avulla varmistutaan, että rajapinta toimii oletetulla tavalla ja koodiin ei ole jäänyt minkäänlaisia virheitä. Testatessa on tärkeää testata kaikki mahdolliset tilanteet. Rajapintaa testatessa kannattaa kokeilla tilanteita, joissa jokin arvo puuttuu, niin saa selville, miten rajapinta siihen reagoi. Tämmöiset tilanteet voivat olla harvinaisia, jos esimerkiksi ajatellaan, että sovellus tekisi pyynnön rajapintaan, niin silloin kaikkien arvojen pitäisi olla pyynnössä, kun sovelluskehittäjä on sen niin ohjelmoinut. Kuitenkin on hyvä aina varmistua, että rajapinta osaa toimia niin sanotuissa poikkeustilanteissa, eikä esimerkiksi jää jumiin tai tee jotakin todella odottamatonta. (32.)

Rajapinnan testaamiseen löytyy hyvin paljon eri kirjastoja eri ohjelmointikielille. Näistä saa valita itselleen mieluisen ohjelmointikielen mukaan sopivimman vaihtoehdon. Laravelin kanssa testaamiseen olisi voitu käyttää Laravelin mukana tulevaa PHPUnit-ohjelmistokehystä, mutta testaus päätettiin tehdä Python-kielellä käyttäen unittest-ohjelmistokehystä. Näin saadaan testattua rajapinnan toimivuus kokonaisuudessaan, koska Laravelissa tehtyjen testien ei tarvitse muodostaa yhteyttä palvelimelle. KUVA 5 on UML-komponenttikaavio, jossa on esitetty eri menetelmien avulla tehtyjen pyyntöjen kulkua. Python tekee pyynnöt täysin erillään Laravelista, mutta kun käytetään PHPUnitia, se käyttää Laravelin kirjastoja ja koodia testien suorittamiseen. Yhdessä vaiheessa ideana oli tehdä yksinkertainen testisivu, jossa olisi erilaisia toimintoja, joilla voisi esimerkiksi testata rajapinnan toimivuutta tekemällä pyyntöjä Javascriptin avulla. Tämä ei kuitenkaan ollut kovin tehokas keino, joten se jäi tekemättä ja keskittyminen jäi testien tekemiseen muilla menetelmillä.



KUVA 5. Eri tavoilla tehtyjen testien toiminta

Tehdyissä testeissä tarkistetaan, että saatava vastaus ja HTTP-tilakoodi ovat mitä niiden oletetaan olevan (KUVA 6). Testeissä tehdään pyyntö requests-kirjaston avulla. Pyyntöön pystyy määrittelemään monia eri asetuksia, kuten metodin, osoitteen ja parametrit. Sitten pyynnön palauttamia arvoja verrataan arvoihin, jotka ovat oletetut arvot kyseiselle pyynnölle. Jos annetut arvot eivät vastaa pyynnön palauttamia arvoja, testi palauttaa silloin virheen, jossa yleensä on rivi, josta virhe on tullut. Jos testi on onnistunut, ohjelma jatkaa suorittamista ja lopuksi antaa yhteenvedon ohitetuista, onnistuneista ja epäonnistuneista testeistä. Testit, joissa tarkistetaan pyynnön vastaus, odottavat vastauksen JSON-muodossa, mutta jos pyynnössä on tapahtunut virhe ja rajapinta ei palauta vastausta JSON-muodossa, niin silloin testissä tapahtuu virhe ja ohjelma lopettaa testaamisen siihen.

```
import unittest
import requests

class TestStringMethods(unittest.TestCase):
    def test_basic(self):
        """
        Basic unittest test
        """

        response = requests.get("https://localhost:8000/users/1")

        self.assertEqual(response.status_code, 200)

if __name__ == '__main__':
    unittest.main()
```

KUVA 6. Pythonilla tehty esimerkkitestitapaus

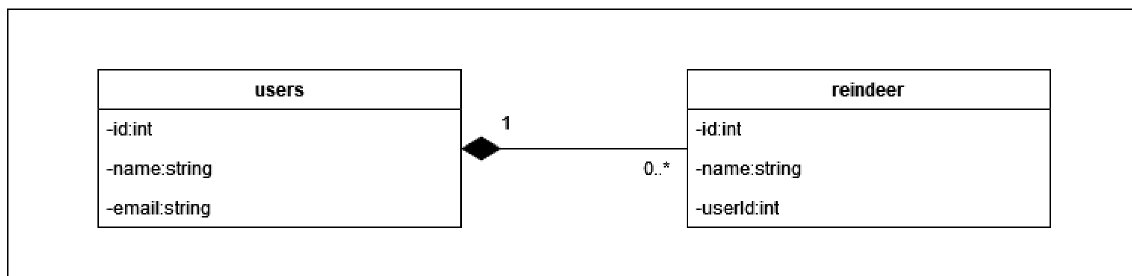
5 RAJAPINNAN KEHITTÄMISEN VAIHEET

5.1 Uuden rajapinnan suunnittelu

Aluksi täytyi opetella tietokanta-asioita ja PHP:tä. Minulla oli jonkinlainen käsitys molemmista, mutta täytyi opetella paremmin asiat, että ymmärsin, miten tietokanta ja nykyinen rajapinta toimivat. Lisäksi piti kerrata ja opetella UML-mallien tekemistä, koska niitä tultaisiin tekemään ja aiempaa kokemusta oli vähän.

Samalla piti opetella REST-rajapinnan määritelmää, koska uusi rajapinta tultiin tekemään REST-tyyliseksi. Rajapinnoista tai REST-rajapinnoista tarvitsin paljon enemmän tietoa, kun olin lähdössä tekemään täysin uutta rajapintaa yritykselle ja sitä tullaan joku päivä käyttämään.

Aluksi loin luokkakaavion yrityksen tietokannasta (KUVA 7). Sen avulla sain havainnollistettua tietokannan rakenteen ja taulujen ja sarakkeiden relaatiot. Kaaviosta oli myös hyötyä yritykselle, koska semmoista ei aiemmin ollut tehty.



KUVA 7. Esimerkki UML-luokkakaavio tietokantarakenteesta

Kun PHP ja tietokanta oli hyvin hallussa, aloin käymään läpi rajapinnan koodeja. Koodeja lukiessani huomasin, että suurin osa tehdyistä kutsuista ei palauta kunnollista vastausta, josta saisi selvää, onko pyyntö onnistunut. Muitakin havaittuja ongelmia ja puutteita oli, jotka sitten kirjasin ylös, jotta muistan ne.

Sen jälkeen oli aika alkaa tutustumaan Laravelin, joka oli valittu ohjelmistokehys. Laravel on onneksi hyvin dokumentoitu ja paljon käytetty, joten kysymyksiin ja ongelmiin oli helppo löytää vas-

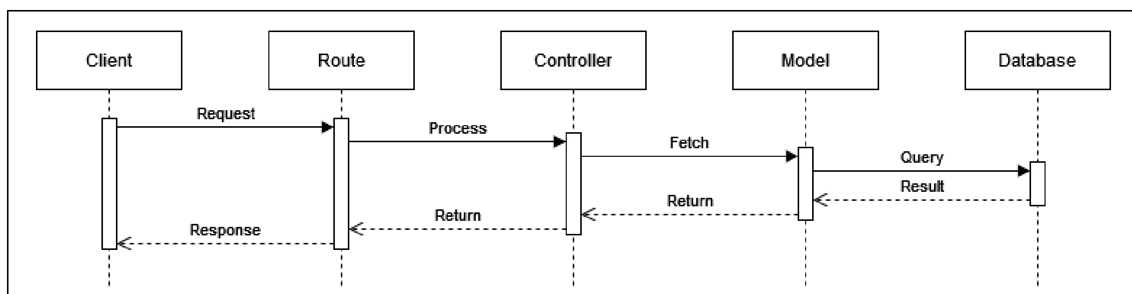
tauksia. Laravelin tiedostorakenne tuntui aluksi oudolta ja hiukan hankalalta, mutta kun tuli koodattu, sen kyllä oppi. Olioiden, kontrollereiden ja väliohjelmistojen käyttö oli ollut vähäinen, joten niitäkin piti opetella ja kertailla. Laravelin testaamista varten minulla oli omalla koneella se asennettuna. Rajapintaa varten Laravel asennettiin palvelimelle.

5.2 Rajapinnan toteutus

Kun aloitin Laravelin käytön, ensimmäisenä opettelin ja kävin läpi perusasiat. Niiden avulla pystyin tekemään ensimmäiset kutsut, joita sitten testasin selaimella. Sen jälkeen aloin käymään läpi Laravelin tietokantafunktioita ja -toimintoja, koska tiesin, että niitä tullaan käyttämään paljon.

Laravelilla tehdessä aluksi minulla oli omalla koneella paikallinen tietokanta, johon olin luonut joitakin tauluja samalla tavalla kuin ne olivat yrityksen tietokannassa. Sitten laitoin sinne joitakin esimerkkiarvoja, että pystyin testaamaan Laravelin ja tietokannan toimintaa. Tämä toimi ihan hyvin, mutta sitten myöhemmin vein kopion yrityksen tietokannasta koneelle. Tämä mahdollisti sen, että minulla oli enemmän tietoa tietokannassa ja ne olivat varmasti siinä muodossa kuin niiden piti olla. Tämän avulla pystyin testaamaan, miten rajapinta toimisi oikean tietokannan kanssa, ilman että tarvitsisi pelätä tekevänsä jotain peruuttamatonta.

Kun aloin tekemään uuttaa rajapintaa Laravelin avulla, tein jokaiselle tietokannassa olevalle taululle oman mallin, joka mahdollistaa olion avulla tehdä tietokantapyyntöjä. Sitten loin jokaiselle mallille oman kontrollerin, johon kokosin kaikki malleihin liittyvät funktiot, jotka sisältävät toiminnot eri pyynnöille. Kun rajapintaan tulee kutsu, se menee kontrolleriin, josta se sitten hakee olion ja lopulta hakee tietokannasta tiedot (KUVA 8). Luotuihin malleihin piti tehdä hiukan muutoksia, koska Laravel olettaa, että mallia vastaava taulu on mallin nimi monikossa, eli s-pääte. Tämä tarkoittaisi sitä, että jos olisi käyttäjä malli nimeltään user, Laravel etsii taulua, joka on nimeltään users.



KUVA 8. UML-sekvenssikaavio tehdyn pyynnön kulusta Laravelin rakenteessa

Laravel luo lokin virheitä varten, mutta sinne kirjoitetaan kaikki Laravelissä tapahtuvat virheet. Sen takia, mahdollisia virhetilanteita varten loin oman lokin, johon tulee kaikki pyynnöistä johtuvat virheet ja selitykset. Tämä nopeuttaa virheiden etsimistä, kun ne on pelkästään koottu yhteen tiedostoon. Lokiin voi myös kirjoittaa muutakin kuin virheitä, esimerkiksi kun luodaan uusi käyttäjä, siitä voidaan kirjoittaa lokiin tieto.

Kun lähdin tekemään pyyntöjä uutta rajapintaa varten, katsoin yrityksen sen hetkisestä rajapinnasta kutsujen osoitteet ja mitä ne tekevät. Sitten lähdin tekemään kutsut uuteen rajapintaan. Tarkoituksena oli tehdä toiminnoiltaan samanlaiset kutsut kuin sen hetkisessä rajapinnassa, mutta uuden rajapinnan mukaan. Se mahdollisti muun muassa olioiden käytön tietokantakyselyiden tekemiseen sen sijaan, että olisi kirjoitettu suoraan tietokantakysely.

Uuteen rajapintaan on lisätty tarkastuksia tietokantakyselyiden jälkeen, joka joissakin tilanteissa palauttaa virheen, jos jotakin vaadittua tietuetta ei löydy. Nykyisen rajapinnan ongelmia on korjattu uuteen rajapintaan. Jokaiseen rajapinnan palauttamaan arvoon on lisätty HTTP-tilakoodi, joka vastaa rajapinnan tilannetta. Jos rajapinnassa tapahtuu jonkinlainen virhe, kaikki virheeseen liittyvä tieto kirjoitetaan lokiin.

Rajapintaan on lisätty validointi kaikkiin POST-metodien parametreihin. Se varmistaa, että vaadittavat arvot on annettu ja ne ovat oikeassa muodossa. Myös kyselyparametrit tarkistetaan, että ne on annettu, kun niitä vaaditaan.

Uusi rajapinta hyödyntää istuntoa. Kun käyttäjä kirjautuu sisään, talletetaan käyttäjään liittyviä muuttujia istuntoon. Sen avulla saadaan vähennettyä tehtyjä tietokantapyyntöjä. Jos tehty pyyntö muuttaa istuntoon tallennettua arvoa tietokannassa, sen arvo päivitetään myös istuntoon, että arvot ovat ajan tasalla.

Yritys käyttää joitakin julkisia rajapintoja, kuten esimerkiksi OneSignalia ilmoitusten lähettämiseen käyttäjille. Yrityksen sen hetkinen rajapinta käytti cURL-kirjastoa pyyntöjen tekemiseen. Laravel sisältää valmiiksi Guzzle-kirjaston, jonka avulla pystyy tekemään pyyntöjä. Toiminnaltaan kirjastot ovat hyvin samanlaisia, mutta Guzzlella tehty koodia on helpompi lukea. Ilmoitusten sisällön ja kaiken muun rajapintoihin liittyvän pidin samanlaisena kuin aiemmin, niin ei tule minkäänlaisia ongelmia.

Rajapinnan versiohallintaa varten käytettiin GitLabia, jota yritys on jo käyttänyt pitkän aikaa. GitLab käyttää Gitiä, joka on versionhallintajärjestelmä, jota esimerkiksi GitHub käyttää. Gitin käyttöön on olemassa useita erilaisia työkaluja niin visuaalisesta työkalusta aina komentorivityökaluihin. Päädyin käyttämään Git Bash -sovellusta, jonka avulla pystyy käyttämään Gitiä komentoriviltä. Käyttö vaati aluksi harjoittelua, koska Gitin käyttö aiemmin oli ollut hyvin vähäistä, mutta kyllä sitten perus komennot tuli tutuksi käytön myötä.

Rajapintaan luotiin tietokantamigraatiot. Näiden avulla pystytään luomaan kaikki vaadittavat tietokantataulut. Tietokantamigraatioissa voi määrittää tauluihin sarakkeet, niiden tyypit ja sarakkeiden ja taulujen väliset relaatiot (KUVA 9). Tietokantamigraatio käyttää up-funktioita taulun luomiseen ja down-funktio on sitä varten, jos tehty tietokantamigraatio halutaan peruuttaa. Tässä esimerkissä peruuttaminen poistaa taulun tietokannasta. Tietokantamigraatioiden avulla voi muokata tai poistaa jo olemassa olevia tauluja tai tauluissa olevia sarakkeita. Tietokantamigraatioita varten piti katsoa taulujen rakennetta ja tehdä taulu samanlaisilla ominaisuuksilla funktioon. Hieman poiketen nykyisestä rakenteesta, tauluihin lisättiin viittauksia sekä sarakkeet, milloin rivi on luotu ja milloin sitä on viimeksi muokattu.

Tietokantamigraatioiden suoritus järjestyksellä on merkitystä. Kun tietokantamigraatio luodaan, sille voidaan antaa kuvaava nimi, mutta kuitenkin sen tiedostonimen alkuun lisätään automaattisesti päivänmäärä ja aika, milloin migraatio on luotu. Tiedostonimeä voi joutua muuttamaan, jos on olemassa relaatioita taulujen välillä tai on migraatio, joka lisää tai poistaa olemassa olevasta taulusta sarakkeita. Migraatioita suorittaessa ne käydään läpi tiedostonimen mukaan järjestyksessä ja kun tiedostot ovat oikeassa järjestyksessä, saadaan toimiva ja oikeanlainen tietokanta.

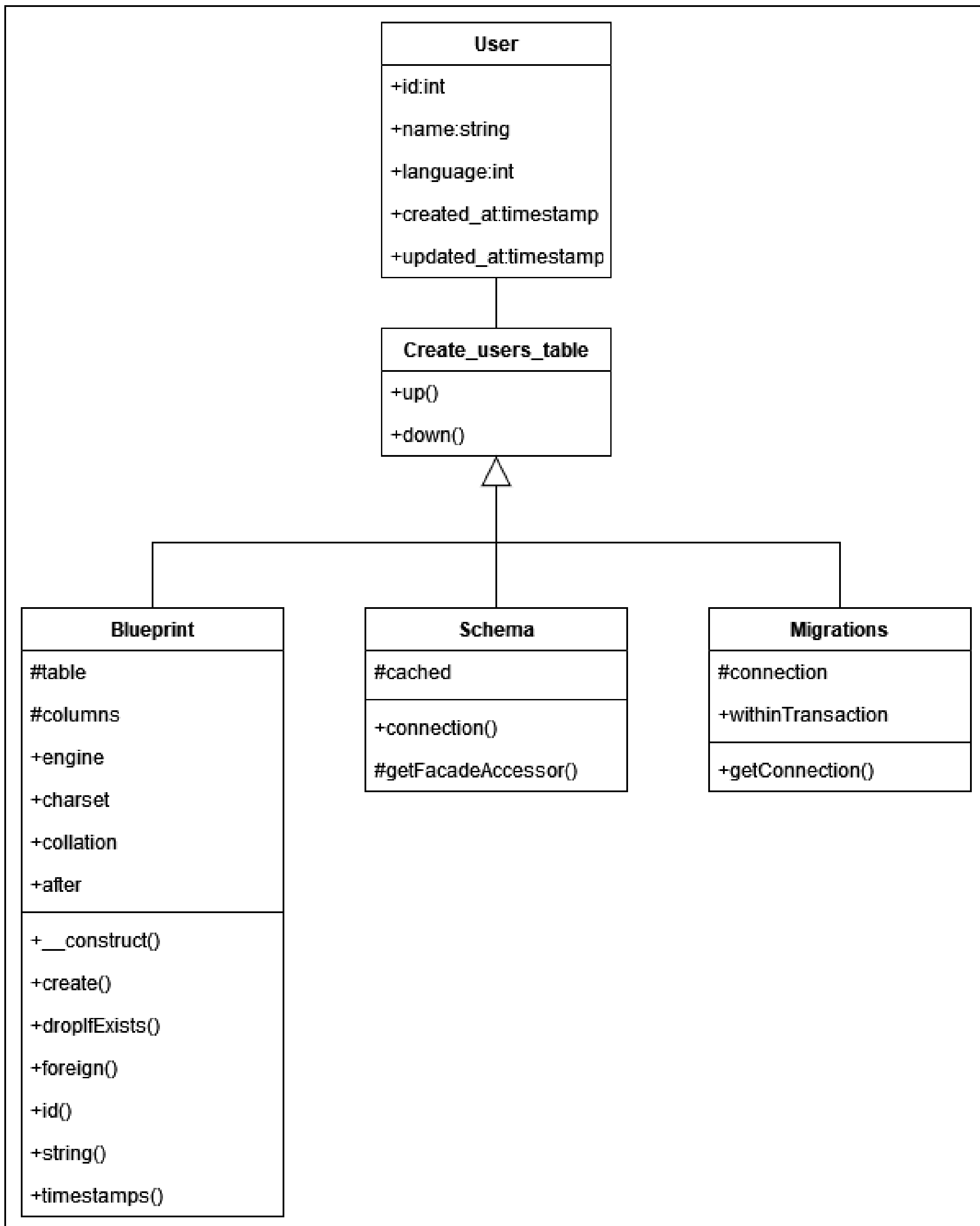
```php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->id();
            $table->string('name', 50);
            $table->integer('language');
            $table->timestamps();
            $table->foreign('language')->references('id')->on('languages');
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('users');
    }
};
```

KUVA 9. Esimerkki migraatiosta, jolla luodaan users-taulu

Migraatioiden tekemiseen käytetään useita eri kirjastoja. KUVA 10 on kuvattu luokkakaaviolla migraation käyttämiä kirjastoja sekä User-olio, joka löytyy luodusta users-taulusta. Alla kuvatut blueprint, schema ja migrations ovat tiedostoja, jotka tämä migraatio perii ja se käyttää niiden funktioita migraation tekemiseen.

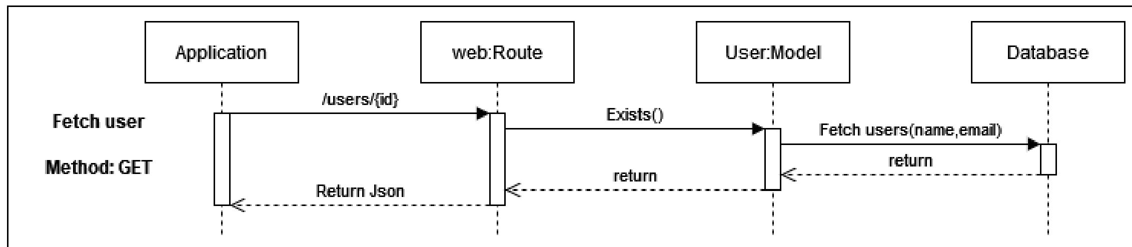


KUVA 10. UML-luokkakaavio migraatiossa käytetyistä kirjastoista ja olio, joka luodaan migraation avulla

5.3 Rajapinnan dokumentointi

Rajapinnan kehittämisen aikana on tullut laadittua UML-kaavioita. Kuten aiemmin mainittiin, tein luokkakaavion tietokantarakenteesta. Sen lisäksi olen tehnyt sekvenssikaavion rajapinnan kutsuista. Siinä on esimerkiksi käyty läpi, mitä olioita ja kontrollereita kutsut käyttävät, mitä ne hakevat

tai tallentavat tietokantaan ja mitä kutsut palauttavat. Kaaviota tehdessä ei ole suoraan noudatettu sekvenssikaavion tyyliä, vaan se on tehty sillä tavalla, että muidenkin rajapintaa käyttävien olisi helppo ymmärtää sitä (KUVA 11).



KUVA 11. Esimerkki laaditusta UML-sekvenssikaaviosta

Rajapinnan dokumentointiin käytettiin jo aiemmin mainittua Stoplightia. Dokumentoin yrityksen nykyisen rajapinnan, jotta sekin löytyy samasta paikasta. Sitten laadin dokumentaation tekemäni rajapinnan pohjalta. Tämä dokumentaatio tullaan vielä päivittämään ja tarkistamaan, että se vastaa viimeisintä versiota rajapinnasta, kun se on valmis. Dokumentoinnin tarkoitus on auttaa muita kehittäjiä, jotka tulevat käyttämään rajapintaan.

Dokumentaation laadittiin kaikki reitit ja niiden tukemat HTTP-metodit. Dokumentaatiossa kerrotaan lyhyesti kutsujen toimintaa ja sitten mitä ne vaativat. Dokumentoinnista käy ilmi vaadittavat parametrit, niiden tyypit, esimerkkiarvot ja ovatko ne vaadittuja. Jos lähetetään parametri, jota ei ole listattu, rajapinta ei huomioi sitä. Jos vaadittu parametri puuttuu tai jos annettu parametri ei vastaa annettua tyyppiä, kutsu ei toimi ja rajapinta lähettää virheen. Joillekin parametreille on määritelty pituuden raja-arvot ja mitä ne voivat sisältää. Parametreille on lisätty selitykset siitä, mitä ne ovat tai mitä ne tarkoittavat. Jokaiseen kutsuun on lisätty esimerkkivastaukset, jotka sisältävät HTTP-tilakoodit ja JSON-muotoiset vastaukset. KUVA 12 näkee dokumentointityökalun näkymän sekä sen vieressä dokumentaation visuaalisen ilmeen.

KUVA 12. Stoplightin näkymä, jossa vasemmalla työkalu ja oikealla luotu dokumentti

Opinnäytetyön aikana laadin lisäksi vaatimusmäärittelyn tulevaa pääkäyttäjän liittymää varten. Liittymän on tarkoitus toimia visuaalisena työkaluna eri asioiden hallintaan ja hakemiseen sen sijaan, että ne esimerkiksi suoritettaisiin manuaalisesti tietokantakyselyiden avulla. Vaatimusmäärittelyitä ei kovin paljoa ole tullut laadittua, joten tämä oli hyvää harjoitusta sen oppimiseen. Osalle toiminnoista on jo olemassa jonkinlainen toiminto rajapinnassa, mutta suurin osa toiminnoista on semmoisia, joita varten pitää sitten tehdä rajapintaan toiminto. Koska kyseessä on pääkäyttäjän liittymä, niin yksi toiminto esimerkiksi oli, että voidaan hakea rajapinnan lokitiedostot.

Vaatimusmäärittely tehtiin Google Sheetsin avulla ja sarakkeisiin päätettiin sopivat otsikot kuten TAULUKKO 2, joiden alle toiminnon tiedot tultiin kirjoittamaan. Ensimmäinen sarake on id, jota käytetään tunnisteena vaatimuksille. Päätoiminolla tarkoitetaan kokonaisuutta, jonka alle kootaan sitten siihen liittyvät toiminnot. Kuvauksen avulla pyritään kertomaan mitä toiminnon avulla pitää pystyä tekemään. Kuvauksessa ei kerrota laajemmin toiminnasta, vain pääpirteittäin ja siten, että toiminnot eivät mene sekaisin ja ovat tunnistettavissa. Toiminnon tärkeys arvioidaan asteikolla 1–5. Asteikolla arvo 1 tarkoittaa, että toiminto on pakollinen ja näin myös prioriteetti on korkeampi. Asteikolla arvo 5 tarkoittaa, että toiminto olisi hyvä olla olemassa, mutta se ei ole pakollinen ja näin sen prioriteetti on matala. Riippuvuudet muihin vaatimuksiin alle listataan toimintojen id, joihin toiminnolla on riippuvuus. Riippuvuudella tarkoitetaan, että toiminto jakaa joitakin toimintoja toisen toiminnon kanssa, joten se on tehtävä ensin tai toiminto vaatii jonkin toisen toiminnon olemassa olemisen, että se voi toimia. Tila määrittelee vaatimuksen sen hetkisen tilan. Tilalla on mahdollista olla arvot ei aloitettu, odottaa, kesken ja valmis. Kuvattu-sarakkeelle laitetaan linkki toiminnosta

tehtyyn dokumentaatioon. Dokumentissa kerrotaan tarkemmin toiminnosta ja esimerkiksi toiminnosta voidaan laatia käyttötapauskaavio. Lisätty-sarake sisältää päivänmäärän, jolloin toiminto on lisätty. Sen avulla pystytään seuramaan, jos jokin toiminto on ollut kauan odottamassa tai unohtunut. Sitten se voidaan nostaa korkeammalle prioriteetille.

TAULUKKO 2. Vaatimusmäärittelyssä laaditut sarakkeet ja lyhyt kuvaus, mitä niillä tarkoitettaan

Sarake	Lyhyt kuvaus
ID	Juokseva numerointi, jota käytetään vaatimuksen tunnisteena. Voidaan käyttää viitatessa vaatimuksiin.
Päätoiminto	Laajempi käsitys toiminnosta, jonka alle kootaan siihen liittyvät vaatimukset.
Kuvaus	Vaatimuksen toiminnot kerrottuna siten, että se on ymmärrettävissä.
Tärkeys	Vaatimuksen tärkeys asteikolla 1–5. Asteikon alkupää sisältää vaatimukset, jotka ovat vaadittuja tai pakko olla. Loppupäässä on vaatimukset, joita on toivottu, mutta niiden prioriteetti on vähäinen.
Riippuvuudet muihin vaatimuksiin	Vaatimusten tunnisteet, joita vaatimus tarvitsee toimiakseen tai jos se jakaa toiminnallisuuksia vaatimuksen kanssa.
Tila	Käytetään vaatimuksen edistymisen kuvaamiseen. Valittavissa neljä vaihtoehtoa: ei aloitettu, odottaa, kesken ja valmis.
Kuvattu	Linkki dokumentaatioon, jossa vaatimusta kuvattu tarkemmin tai siitä on laadittu esimerkiksi käyttötapauskaavio.
Lisätty	Päivämäärä, jolloin vaatimus on lisätty. Pystytään seuramaan, ettei mikään toiminto pääse unohtumaan ja voidaan sitten esimerkiksi nostaa prioriteetissa korkeammalle, jos näin on käynyt.

5.4 Rajapinnan testaus

Rajapinnan testaamista varten testit piti kirjoittaa. Testejä on paljon, koska monelle kutsulle on tehty enemmän kuin vain yksi testi. Testeissä esimerkiksi käydään läpi, että mikä on tulos, jos jokin parametri puuttuu tai jos annetaan arvo, jota ei löydy tietokannasta. Sitten pitää tietysti testata, mikä tulos on, kun parametrit ovat annettu ja on annettu arvo, joka löytyy tietokannasta. Testit kirjoitettiin koodin pohjalta, eli ei tehty testivetoista kehitystä, jossa testit luodaan ensin ja sitten

koodataan. Tähän osasyynä oli se, että rajapinnan kutsujen toiminta oli jo tiedossa nykyisen rajapinnan pohjalta.

Vaikka testit toimivat ja menevät läpi, se ei tarkoita, että ne olisivat hyviä. Kun testit oli tehty, käytettiin Xdebug PHP-laajennusta, joka tarkisti, kuinka kattavasti testit käyvät koodeja läpi. Sen avulla on mahdollista palauttaa testejä suorittaessa raportti, joka vain näytetään yksinkertaisesti komentorivillä tai halutessaan sen voi tallentaa HTML-muodossa (KUVA 13). Näin tulokset ovat paljon kattavammat sekä on mahdollista nähdä, mitkä rivit on käyty läpi ja kuinka monta prosentti mistäkin tiedostosta on käyty.

	Code Coverage					
	Lines		Functions and Methods		Classes and Traits	
Total	58.06%	18 / 31	53.85%	7 / 13	40.00%	4 / 10
Console	66.67%	2 / 3	50.00%	1 / 2	0.00%	0 / 1
Exceptions	100.00%	2 / 2	100.00%	1 / 1	100.00%	1 / 1
Http	0.00%	0 / 9	0.00%	0 / 3	0.00%	0 / 3
Models	n/a	0 / 0	n/a	0 / 0	n/a	0 / 0
Providers	82.35%	14 / 17	71.43%	5 / 7	60.00%	3 / 5

Legend

Low: 0% to 50% **Medium**: 50% to 90% **High**: 90% to 100%

KUVA 13. Esimerkki testien koodin kattavuuden tuloksista

Raportti paljasti, että yli puolet tehdyistä koodeista oli kokonaan testaamatta. Raportista etsittiin testaamattomat rivit ja lisättiin olemassa oleviin testeihin sisältöä ja lisättiin täysin uusia testejä. Testien määrä kaksinkertaistui tämän takia. Laajasti testattu koodi kuitenkin vähentää virheitä ja todistaa, että koodi toimii oletetusti. Työkalun ja raportin suhteen pitää kuitenkin ajatella, että se kertoo vain koodin kattavuuden, mutta ei sitä onko se oikeaa. Se tehtävä jää ohjelmoijalle.

Koodin kattavuuden selvittämiseksi testit täytyi tehdä PHP:n avulla Laravelin kansioihin. Testausta helpottamiseksi Laravel sisältää toiminnon, jonka avulla voidaan luoda olioita tietokantaan. Näitä olioita voidaan käyttää sitten testaamisessa, kun esimerkiksi testitapaus olettaa, että kyseinen olio on olemassa. Oliolle voidaan antaa oletusarvot, joita sitten käytetään olion luomiseen tietokantaan. Nämä annetut oletusarvot voidaan kuitenkin halutessaan muuttaa testeissä. Laravel sisältää myös funktioita, joiden avulla voidaan luoda esimerkiksi nimiä tai sähköpostiosoitteita. Nämä helpottavat olioiden luonnissa, kun pystytään luomaan erilaisia olioita.

Sen lisäksi, että rajapinnan testaamiseen käytettiin testejä, lisäksi rajapinnan koodia testattiin. Tähän käytettiin työkalua nimeltään PHPStan. PHPStan käy koodin läpi ja antaa sitten palautetta löytyneistä virheistä. KUVA 14 on esimerkki PHPStanin palauttamasta tuloksesta suorittamisen

jälkeen. Sen palauttamat virheet kertovat, että koodissa on tilantarkistus, joka on aina tosi sekä määrittelemättömiä muuttujia. PHPStan pystyy tulkitsemaan PHP koodia ja siihen on vielä saatavilla laajennuksia, joiden avulla se saadaan paremmin ymmärtämään eri alustojen koodia, kuten Laravelin. PHPStan ajetaan komentorivillä ja sille voi määrittää tason, millä tarkistus suoritetaan ja kansion, joka tarkistetaan. Nämä voidaan myös määrittää erillisessä tiedostossa, joka mahdollistaa sen, että PHPStan voidaan suorittaa ilman ylimääräisiä komentoja. Tasoja on yhteensä 10 ja ne alkavat tasosta 0 ja viimeinen taso on 9. Taso 0 on perustarkistus, joka suoritetaan esimerkiksi, jos tasoa ei ole erikseen määritelty. Se palauttaa yksinkertaiset virheet, kuten esimerkiksi jos muuttujaa ei ole määritelty. Jokainen taso lisää asioita, joita PHPStan tarkistaa ja se on paljon tarkempi ylemmillä tasoilla.

```
1/1 [============================] 100%

------ -----------------------------
 Line   ExampleController.php
------ -----------------------------
 13     If condition is always true.
 15     Undefined variable: $text
 19     Undefined variable: $text
------ -----------------------------

[ERROR] Found 3 errors
```

KUVA 14. Esimerkki PHPStan työkalun palauttamista virheistä. Ensimmäinen virhe kertoo, että tilantarkistuksessa tulos on aina tosi, eli toista mahdollista tilaa ei tulla ikinä toteuttamaan. Toinen ja kolmas virhe kertovat määrittelemättömästä muuttujasta $text.

TAULUKKO 3 on listattu eri PHPStan tasot ja mitä tasolla tarkistetaan. Tasot myös sisältävät edeltävien tasojen tarkistukset. Jos vielä tason 9 jälkeen haluaa tiukempaa tarkistusta, on saatavilla laajennuksia ja asetuksia, joita lisäämällä ja muuttamalla saadaan vielä tiukempi tarkistus. (33.)

TAULUKKO 3. PHPStan eri tasojen suorittamat tarkistukset (33)

Taso	Tarkastukset
0	Perustarkistukset, tuntemattomat luokat ja funktiot, tuntemattomat metodi kutsut, argumenttien määrä metodeissa ja funktioissa, määrittelemättömät muuttujat
1	Määrittelemättömät muuttujat, tuntemattomat taikametodit ja ominaisuudet luokissa, joissa __call ja __get
2	Kaikkien tuntemattomien metodien tarkistaminen, PHPDocin validointi

3	Palautettavat tyypit, ominaisuuksille asetetut tyypit
4	Käyttämättömät koodit
5	Metodeille ja funktioille annettujen argumenttien tyypit
6	Puuttuvat tyyppivihjeet
7	PHP:n liittotyypit
8	Metodien ja ominaisuuksien kutsuminen tyhjillä arvoilla
9	Määritetyt mixed-tyypit

PHPStan paljasti koodeista paljon virheitä, joita piti sitten korjata. Yleisimpiä virheitä oli, että muuttujan nimi oli väärä tai siinä oli jokin kirjoitusvirhe ja joitakin muuttujia ei ollut määritelty ennen niiden käyttöä. Lähdin suorittamaan tasosta 0 ja aina askel kerrallaan siirryin seuraavalle tasolle, kun virheet oli korjattu. Näin sain suoritettua tarkastuksen aina tasolle 5 asti kunnes PHPStan alkoi valittamaan määrittelemättömistä palautustyypeistä funktioille. Kuitenkin nämä oli määritelty suurimmassa osassa, koska kaikissa ei ollut määritelty palauttamaan mitään. Tämä johtui siitä, että rajapinta oli tehty samalla tavalla kuin alkuperäinen rajapinta.

6 JATKOKEHITYS

Uuden rajapinnan käyttöönotto vaatii vielä hiukan jatkokehitystä. Se vaatii jonkinlaisen autentikoinnin tai suojauksen, jonka avulla pystytään rajoittamaan rajapinnan käyttöä. Jonkinlainen pyyntöjen rajoittaja olisi hyvä olla, että esimerkiksi käyttäjä ei pystyisi tekemään useita kymmeniä pyyntöjä minuutissa. Sen avulla voitaisiin ehkäistä palvelimen hidastumista tai kaatumista, jos esimerkiksi tapahtuisi virhe, joka tekisi useita pyyntöjä hallitsemattomasti. Koodiin on myös lisättävä käytettyjen rajapintojen osoitteet sekä avaimet ja muut tunnisteet, koska niitä ei ole ollut käytössä, kun on testattu toimivuutta. Sitten myös luultavasti käyttöönotto vaatii muutoksien tekemistä sovellukseen, koska rajapinnan palauttamat arvot ovat muuttuneet sekä nyt on käytössä HTTP-tilakoodit, joiden avulla pystyy tulkitsemaan rajapinnan vastauksia.

Rajapintaa tehdessä on tullut vastaan monia asioita, joita tulevaisuudessa kannattaa lähteä tekemään. Yksi tällainen on pääkäyttäjän liittymä. Pääkäyttäjän liittymän tarkoituksena on mahdollisuus järjestelmän toiminnan monitorointiin sekä ylläpitoon yhdessä käyttöliittymässä. Näihin esimerkiksi kuuluu rajapinnan lokien hakeminen sekä käyttäjien ja laitteiden hallinta. Siinä on paljon hommaa, kun suunnittelee ja toteuttaa sen modulaarisena ja sitten sen jälkeen alkaa tekemään vaatimusmäärittelyssä listattuja toimintoja. Ne vaativat lisäksi suunnittelua ennen kuin niitä lähtee tekemään.

Tehty rajapinta on REST-rajapinta, mutta se vaatii vielä kehittämistä. REST-rajapinnan täytyy noudattaa viittä periaatetta, joita on jo aiemmin käyty työssä läpi. Kuitenkin viidettä kohtaa eli kerroksittaista rakennetta ei ole kovin paljoa otettua huomioon, joten se vaatisi vielä jonkinlaista kehittämistä.

Tässä toteutuksessa tehtyjä testejä ajetaan manuaalisesti, että saadaan selville rajapinnan toiminta. Jatkokehitysideana on automaattitestaus rajapintaan, joka ajaa testejä esimerkiksi kerran viikossa tai pari kertaa kuussa, että varmistetaan rajapinnan toimivuus. Jokaisen testiajon jälkeen kirjataan raportti, josta käy ilmi ajon aikana rajapinnassa tapahtuneet mahdolliset virheet. Mahdollisuus olisi lisätä jonkinlainen ilmoitus esimerkiksi sähköpostiin, jos virheitä on tapahtunut. Ilman ilmoitusta pitäisi aina tarkistaa tehdyistä raporteista mahdolliset virheet.

Jatkokehityksenä lisäksi on mainittava uuden valitun dokumentointityökalun käyttö dokumentaatioita tehdessä. Työkalu on yritykselle uusi, joten se varmasti vaatii perehdytystä käyttöön. Se kuitenkin nopeuttaa sekä selkeyttää dokumentointia.

Vaikka opinnäytetyössä on keskitytty sovelluksen käyttämiin rajapintakutsuihin, on ollut jonkinlaista keskustelua myös laitteiden käyttämistä rajapintakutsuista. Mahdollisuutena olisi tehdä rajapintakutsut Laravelin avulla. Kuitenkin on keskusteltu, että käytettäisiin CoAP-protokollaa.

7 JOHTOPÄÄTÖKSET

Opinnäytetyön tavoitteena oli saada käyttövalmiiksi rajapinta, jonka tekemistä olin jo aloittanut yritysprojektien ja harjoitteluiden aikana. Käyttövalmiilla tarkoitetaan sitä, että rajapinta sisältää samat toiminnallisuudet kuin yrityksen oma rajapinta sekä se on testattu ja hyvin dokumentoitu. Sen jälkeen rajapinta olisi valmis ja sen käyttöä voitaisiin aloittaa suunnitella sovelluksessa.

Tavoitetta ei täysin saavutettu. Rajapinta on dokumentoitu sekä kattavasti testattu ja testien kattavuus on myös varmistettu. Yrityksen nykyinen rajapinta oli hyvin niukasti kommentoitu, joten nyt koodia tehdessä olen varmistanut, että koodi on kommentoitu. Toiminnallisuuksiltaan rajapinnat ovat samanlaisia. Kuitenkin se, että rajapinta olisi käyttövalmis, vaatii vielä hiukan tekemistä. Sen käyttämisen turvallisuuden varmistaminen vaatii vielä tekemistä ja näistä on mainittu jatkokehityksessä.

Kun mietin koko prosessia rajapintaa tehdessä, en oikein osaa sanoa olisiko jotain kannattanut tehdä toisin. Koen, että rajapintaa tehdessä on edetty ihan loogisesti järjestyksessä siinä mitä on tehty ja opeteltu. Paljon olisi varmasti vielä ollut käytävänä ja opittavana, jos olisi päästy täysin testaamaan ja tekemään rajapinnan käyttöönoton. Kuitenkin rajallisen ajanpuutteen vuoksi ne jäivät tekemättä. Resurssien ja ajan puutteen vuoksi rajapinnan testit piti itse suunnitella ja toteuttaa. Testejä laatiessa yleensä on testaajat erikseen, jotka laativat testit, jotta saataisiin testattua monenlaisia eri tapauksia, jotka voisivat jäädä koodaajalta tekemättä.

Tämä on ollut todella opettavainen kokemus. Opin todella paljon tietokannoista, joista minulla oli aiemmin todella vähäinen tieto. Rajapinnoista ei myöskään ollut kovin paljoa tietoa, joten tämä auttoi ymmärtämään niiden tarkoituksen ja tärkeyden. UML-malleja oli vähän tullut tehtyä ennestään, mutta nyt sain opeteltua ne todella hyvin käytännössä ja opin missä tilanteissa eri malleja käytetään. PHP:n aiempi käyttö oli todella vähäinen, joten tässä on tullut opittua todella paljon PHP:n käytöstä ja eri funktioista. Laravel oli ihan täysin uusi asia, jonka koen nyt ymmärtäväni hyvin. Testauksesta tietoni oli hyvin vähäinen eikä ollut tullut tehtyä kovin paljoa testejä. Nyt kuitenkin olen oppinut paljon testeistä ja sen, että ne eivät ole turhia ja vaikka niiden tekeminen vie aikaa, se kuitenkin säästää aikaa jatkossa. Dokumentointia ei aiemmin ollut tullut tehtyä kovin paljoa. Nyt kuitenkin opin, että dokumentointi on tärkeää, koska olin itse siinä tilanteessa, että piti alkaa tekemään uudelleen rajapintaa, jota oli dokumentoitu hyvin vähäisen. Sama tilanne oli myös

koodin kommentoimisen kanssa. Opin myös, että on mahdollista tarkistaa tehtyjen testien kattavuus sekä on olemassa työkaluja, joiden avulla voidaan tarkistaa tehtyä koodia virheiden varalta. Uskon, että kaikella oppimallani on käyttöä tulevaisuudessa.

LÄHTEET

1. Finder. Anicare Oy. Hakupäivä 25.9.2023. https://www.finder.fi/Viestint%C3%A4lait-teet+kuvansiirtolaitteet+ja+palvelut/Anicare+Oy/Oulu/yhteystiedot/3200849
2. Anicare. Tahdosta auttaa porotaloutta. Hakupäivä 20.10.2023. https://anicare.fi/#tarina
3. Singh, Ela 2021, Client-Server Architecture. Medium. Hakupäivä 24.11.2023. https://medium.com/codex/client-server-architecture-5e103aa0106d
4. Jain, Nikhil 2023. Peer-to-Peer Architecture: A Deep Dive into the Future of Networking. Medium. Hakupäivä 24.11.2023. https://medium.com/@nikhiljain1203/peer-to-peer-architecture-a-deep-dive-into-the-future-of-networking-d0f07945dca5
5. Oppel, Andy 2009, luku 1. Data Modeling, A Beginner's Guide. First edition. The McGraw-Hill Companies. Hakupäivä 23.11.2023. O'Reilly. Vaatii käyttöoikeuden. https://learning.oreilly.com/library/view/data-modeling-a/9780071623988
6. IBM. What is data modelling? Hakupäivä 24.11.2023. https://www.ibm.com/topics/data-modeling
7. Mau, Derek 2019. Guide to UML diagramming and database modelling. Microsoft. Hakupäivä 24.11.2023. https://www.microsoft.com/en-ww/microsoft-365/business-insights-ideas/resources/guide-to-uml-diagramming-and-database-modeling%20ha-kup%C3%A4iv%C3%A4%2024.11.2023
8. Peterson, Richard 2023. What is Normalization in DBMS (SQL)? 1NF, 2NF, 3NF Example. Guru99. Hakupäivä 24.11.2023. https://www.guru99.com/database-normalization.html
9. Momoh, Emmanuel 2023. Database Normalization in Simple Terms (1st, 2nd and 3rd Normal Forms). DEV Community. Hakupäivä 27.11.2023. https://dev.to/nuel000/database-normalization-in-simple-terms-1st-2nd-and-3rd-normal-forms-47ce
10. IBM. What is an API? Hakupäivä 13.10.2023. https://www.ibm.com/topics/api
11. C, DaNeil 2020. API =/= Database. DEV Community. Hakupäivä 9.11.2023. https://dev.to/caffiendkitten/api-database-4le6
12. Postman. What is an API? Hakupäivä 9.11.2023. https://www.postman.com/what-is-an-api/
13. IBM. What is a REST API? Hakupäivä 13.10.2023. https://www.ibm.com/topics/rest-apis
14. The Postman Team 2020. What Is a REST API? Examples, Uses and Challenges. Postman. Hakupäivä 16.11.2023. https://blog.postman.com/rest-api-examples/

15. IBM. What is LAMP stack? Hakupäivä 11.10.2023. https://www.ibm.com/topics/lamp-stack
16. Nielsen, Henrik, Mogul, Jeffrey, Masinter, Larry, Fielding, Roy, Gettys, Jim, Leach, Paul & Berners-Lee, Tim 1999. Hypertext Transfer Protocol – HTTP/1.1. Datatracker. Hakupäivä 9.10.2023. https://datatracker.ietf.org/doc/html/rfc2616
17. Juste, Guillaume 2023. Exploring the Advantages and Disadvantages of Incorporating Frameworks into Web Development. LinkedIn. Hakupäivä 12.10.2023. https://www.linkedin.com/pulse/exploring-advantages-disadvantages-incorporating-frameworks-juste
18. Priya, Yamini 2023. API Functional Testing – Why Is It Important And How to Test. Testsigma. Hakupäivä 12.10.2023. https://testsigma.com/blog/api-functional-testing/
19. O'Grady, Brian 2015. What is a Framework? Why We Use Software Frameworks. Code Institute. Hakupäivä 27.9.2023. https://codeinstitute.net/global/blog/what-is-a-framework/
20. IBM. What is Django? Hakupäivä 27.9.2023. https://www.ibm.com/topics/django
21. Surguy, Maks 2013. History of Laravel PHP framework, Eloquence emerging. Maxoffsky. Hakupäivä 27.9.2023. https://maxoffsky.com/code-blog/history-of-laravel-php-framework-eloquence-emerging/
22. Redmond, Paul 2023. Laravel 10 is now released. Laravel News. Hakupäivä 27.9.2023. https://laravel-news.com/laravel-10
23. SPDEV 2022. PHP Laravel – pros and cons. Linkedin. Hakupäivä 5.12.2023. https://www.linkedin.com/pulse/php-laravel-pros-cons-spdeveloper
24. Vasudevan, Keshav 2023. Why Does API Documentation Matter? Swagger. Hakupäivä 12.10.2023. https://swagger.io/blog/api-documentation/what-is-api-documentation-and-why-it-matters/
25. Martindale, Jon 2023. Microsoft Word vs. Google Docs. Digitaltrends. Hakupäivä 12.10.2023. https://www.digitaltrends.com/computing/microsoft-word-versus-google-docs/
26. Swagger. About Swagger. Hakupäivä 29.9.2023. https://swagger.io/about/
27. Kayte, Sangramsing 2023. Swagger Tools for API Developers. Medium. Hakupäivä 12.10.2023. https://sangramsing.medium.com/swagger-tools-for-api-developers-605e45b4fc71
28. Visual Paradigm. FAQ. Hakupäivä 29.9.2023. https://www.visual-paradigm.com/support/faq.jsp
29. Postman. What is Postman? Hakupäivä 29.9.2023. https://www.postman.com/product/what-is-postman/

30. Postman. Create examples of request responses to illustrate API use cases. Hakupäivä 12.10.2023. https://learning.postman.com/docs/sending-requests/examples/
31. Stoplight. About Stoplight. Hakupäivä 29.9.2023. https://stoplight.io/about
32. Feloney, Stephan 2021. API Function Testing: How to Do it Right. BlazeMeter. Hakupäivä 12.10.2023. https://www.blazemeter.com/blog/functional-api-testing
33. PHPStan. Rule Levels. Hakupäivä 1.12.2023. https://phpstan.org/user-guide/rule-levels