Joonas Kolppanen

Sähköisen taloushallinnon taustajärjestelmän suunnittelu

Käyttötarkoituksiin sopivien teknologioiden valitseminen

Opinnäytetyö
CENTRIA-AMMATTIKORKEAKOULU
Tieto- ja viestintätekniikan koulutusohjelma
Joulukuu 2020

TIIVISTELMÄ OPINNÄYTETYÖSTÄ

Centria-ammattikorkeakoulu	**Aika** Joulukuu 2020	**Tekijä/tekijät** Joonas Kolppanen
Koulutusohjelma Tieto- ja Viestintätekniikka		
Työn nimi SÄHKÖISEN TALOUSHALLINNON TAUSTAJÄRJESTELMÄN SUUNNITTELU. Käyttötarkoituksiin sopivien teknologioiden valitseminen.		
Työn ohjaaja Jari Isohanni		**Sivumäärä** 29 + 0
Työelämäohjaaja Olli Mäkipelto		

Tässä työssä tutkittiin erilaisia tapoja rakentaa tietokanta ja sitä käyttävä rajapinta. Tekniikoita ja teknologioita verrattiin toisiinsa ja selvitettiin kunkin tekniikan hyvät ja huonot puolet sekä mahdolliset käyttökohteet. Opinnäytetyön alkupuolisko koostuu teoreettisestä viitekehyksestä, jossa tutkittiin muutamien eri tietokantojen ja rajapintojen historiaa, rakennetta, vahvuuksia ja heikkouksia. Teoreettisen viitekehyksen jälkeen alkaa opinnäytetyön toinen vaihe, jossa valittiin hyväksi koetut teknologiat ja toteutetaan taloushallinnon laskujenhyväksyntäjärjestelmä.

Opinnäytetyön loppuosassa testattiin kehitetyn tietokannan ja rajapinnan toimivuutta. Testaamista seuraavassa yhteenvetokappaleessa todettiin, että teknologioiden valitseminen onnistui, mutta rajapinnan toteutus oli hieman vääränlainen. RESTfull-arkkitehtuuri harhaantui liian kauaksi RESTin periaatteista, eikä arkkitehtuurin vahvuusalueista saatu täyttä hyötyä.

Asiasanat

NoSQL, Rajapinta, SQL, Tietokanta, Web API

Centria
UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Centria University of Applied Sciences	**Date** December 2020	**Author** Joonas Kolppanen
Degree programme Information and communication technology		
Name of thesis DESIGNING BACKEND FOR E-ACCOUNTING. Choosing the right technologies for each use case.		
Instructor Jari Isohanni		**Pages** 29 + 0
Supervisor Olli Mäkipelto		

In this thesis we explored different ways to develop a database and an interface which will be using those databases. A handful of methods and technologies were compared together in order to find the best solution for each use case.

The first section of this thesis consists of a theoretical framework which includes the history, theory and structure of these chosen databases and interfaces.

After these different technologies are well established begins the second section of this thesis in which we choose between these technologies the best ones to use in the implementation of financial management system.

In the last section we review the program we developed in a critical way and ponder if we chose the right databases and interfaces or if there still could be room for improvement.

Key words

API, Database, NoSQL, SQL, Web API

KÄSITTEIDEN MÄÄRITTELY

CPU	Tietokoneen osa, joka suorittaa konekielisiä käskyjä.
DATA	Yleinen käsite kaikenlaiselle sähköisessä muodossa olevalle tiedolle ja informaatiolle.
RAM	Tietokoneen työmuisti.
SSD	Tietokoneen massamuisti.

1 JOHDANTO

Tämän opinnäytetyön tavoitteena on tutkia erilaisia tapoja toteuttaa sähköisessä taloushallinnossa käytettävä laskujenhyväksyntäjärjestelmä. Opinnäytetyö kattaa ainoastaan taustajärjestelmät eli backendin, joka koostuu tietokannasta ja rajapinnasta. Tietokantaa käytetään laskujen ja muun tiedon tallentamiseen, ja rajapinnan avulla muut sovellukset pääsevät käsiksi tietokannassa säilytettyyn dataan.

Opinnäytetyön viitekehyksessä tutkitaan relatiivisten ja epärelatiivisten tietokantojen ominaisuuksia ja toteutustapoja. Tämän vertailun perusteella selvitetään, millaiset käyttökohteet ovat otollisia kummallekin tietokantatyypille. Samanlainen selvitys tehdään myös rajapintojen kanssa, joista on valittu kolme yleisintä toteutustapaa: REST, RPC ja GraphQL.

Näiden alustavien tutkimusten pohjalta opinnäytetyön käytännön osuudessa tehtävään ohjelmointiprojektiin on valittu parhaiten soveltuvat tietokannat ja rajapinnat. Tietokannan ja rajapinnan rakentamista selvitetään vaihe vaiheelta, jotta päästään paremmin syventymään siihen, kuinka valitut teknologiat toimivat käytännössä.

2 TIETOKANNAT

Tietokannat ovat erittäin tehokas tapa säilöä ja käsitellä suurta määrää dataa. Fyysisen kokoelman kerääminen ja ylläpitäminen on erittäin työlästä ja aikaa vievää, mutta yksi suurimmista eroista näkyy, kun arkistoja halutaan tutkia. Papereiden selaaminen aakkosittain ja vieläpä yleensä pelkkien otsikkojen mukaan ei ole kovin tehokasta, ja sisällön perusteella etsiminen on lähes mahdotonta. Tietokantaan voidaan suorittaa erittäin tarkkoja hakuja ja löytää juuri ne tiedot, joita kulloinkin tarvitaan. (Stephens 2008, 6–10.)

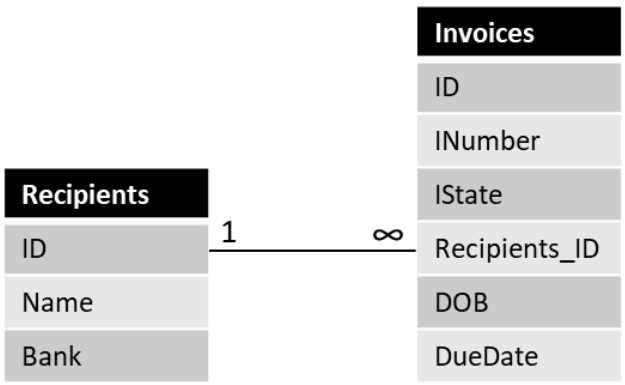
Ohjelmistojen kehittämistä voidaan verrata rakennuksen rakentamiseen. Ennen kuin ryhdytään itse työhön, on kaikki suunniteltava etukäteen hyväksi todettuja menetelmiä noudattaen. Jos kivijalka ei ole kunnossa, talo ei pysy pystyssä. Ohjelmiston kivijalkana toimii tietokanta. Valtaosa sovelluksista käyttää tietokantaa ja siihen tallennettua informaatiota erilaisiin toimintoihin, kuten käsittelee sitä tai esittää sen sovelluksen käyttäjälle. Jos tallennetut tiedot korruptoituvat, katoavat tai niiden käsittelyssä menee valtavasti aikaa, tietokantaa käyttävä ohjelma ei toimi odotetulla tavalla. Tietokantaa käyttävä sovellus ei voi korjata tai paikata tietokannassa tapahtuvia virheitä. Täten koko kokonaisuuden toimivuutta ajatellen tietokanta kannattaa suunnitella todella hyvin. (Stephens 2008, 4–6.)

2.1 Tietokannanhallintajärjestelmä

DBMS (Database Management System) on ohjelma, jota käytetään tietokannan hallitsemiseen. DBMS:n avulla käyttäjä voi luoda, muokata, tai poistaa tietokantoja ja niissä säilytettyjä tietoja. Relatiivisessa tietokannassa käytettävää hallintajärjestelmää kutsutaan nimellä RDBMS (Relative Database Management System), kun taas muun tyyppisissä tietokannoissa ohjelmaa kutsutaan nimellä DBMS. (DBMS Definition.)

2.2 Relatiivinen malli

Relatiivisen tietokantamallin keksi E. F. Codd vuonna 1970. Kirjoittamassaan tutkimuksessa Codd määrittelee, mitä termi relatiivinen tarkoittaa, sekä ne tarkat ominaisuudet, jotka tietokannassa pitää olla, että sitä voidaan kutsua relatiiviseksi tietokannaksi. (Codd 1970.)



KUVIO 1. Esimerkki kahden taulun sarakkeista ja niiden välisestä liitoksesta.

Relatiivisessa tietokannassa informaatio on jäsennelty taulukoihin, jotka koostuvat riveistä ja sarakkeista (TAULUKKO 1). Rivit vastaavat yksittäisiä jäseniä ja sarakkeilla on jäsenten eri ominaisuudet. Tässä esimerksissä Invoices-taulun jäsenet ovat laskuja, joissa yhden laskun tiedot ovat yhdellä rivillä (TAULUKKO 2). Tietokannassa on myös taulujen välisiä yhteyksiä, joita kutsutaan relaatioiksi (KUVIO 1). Näistä on hyötyä tilanteissa, jossa yhdellä jäsenellä on monta arvoa, kuten esimerkiksi yhdellä yrityksellä saattaa olla monta eri laskua. (Stephens 2008, 27–28.)

ID	Name	Bank
1	Mäkelä Oy	Nordea
2	Vuorela Oy	S-Pankki
3	Kumpula Oy	Danske Bank

TAULUKKO 1. Recipients-taulu, joka sisältää asiakkaan tiedot.

ID	INumber	IState	Recipients_ID	DOB	DueDate
1	125005	Confirmed	2	2020-01-10	2020-06-30
2	125006	New	3	2020-03-21	2020-06-30
3	125007	New	1	2020-05-03	2020-06-30
4	125008	Paid	1	2020-05-03	2020-06-30

TAULUKKO 2. Invoices-taulu, joka sisältää laskun tiedot.

2.2.1 SQL:n historia

SQL-kieli sai syntynsä 1970-luvun alkupuolella IBM:n San Josen tutkimuslaboratoriossa. Kyselykielen kehittivät Donald D. Chamberlin ja Raymond F. Boyce kuultuaan Edgar F. Coddin kehittämästä relaatiomallista. (Chamberlin & Boyle 1974, 250.)

SQL-kielen luomisella oli pääosin kaksi tavoitetta. Ensimmäisenä oli vähentää työn määrää ohjelmointiprojekteissa ja täten pienentää ohjelmistokehityksessä syntyviä kuluja. Toisena tavoitteena oli helpottaa tietokantojen käyttöä, mikä mahdollistaa muidenkin kuin ainoastaan koulutettujen asiantuntijoiden kommunikoimisen tietokantojen kanssa. (Chamberlin & Boyle 1974, 250.)

Sen jälkeen, kun IBM oli vakuuttunut SQL:n toimivuudesta ja tehokkuudesta, aloittivat he sen tuotteistamisen. Ensimmäiset IBM:n System R:ään pohjautuvat relatiiviset tietokannat ja SQL-kieliset hallintajärjestelmät tulivat kaupallisesti saataville 1970-luvun lopulla. (IBM 2020.)

2.2.2 SQL-kieli

SQL (Structured Query Language) on kieli, jolla voidaan suorittaa kyselyitä relatiiviseen tietokantaan, minkä jälkeen tietokanta palauttaa kyselyn kriteereitä vastaavien tietueiden tiedot. Kyselyä määritettäessä voidaan myös erotella, mitkä osat tietueiden tiedoista palautetaan. Esimerkkinä voidaan tutkia kuvitteellisen taloushallintajärjestelmän tietokantaa, josta löytyy laskuja, yrityksiä, pankkitilejä ja niin edelleen. Tässä esimerkissä tahdomme hakea tietokannasta kaikkien uusien laskujen numerot. Kuvassa 1 on esitettynä tällaisia hakukriteereitä vastaava SQL-lause. (Wilton & Colby 2010, 11–13.)

```
SELECT INumber FROM Invoices WHERE IState = 'New';
```

KUVA 1. SQL-kysely uusien laskujen numeroiden saamiseksi.

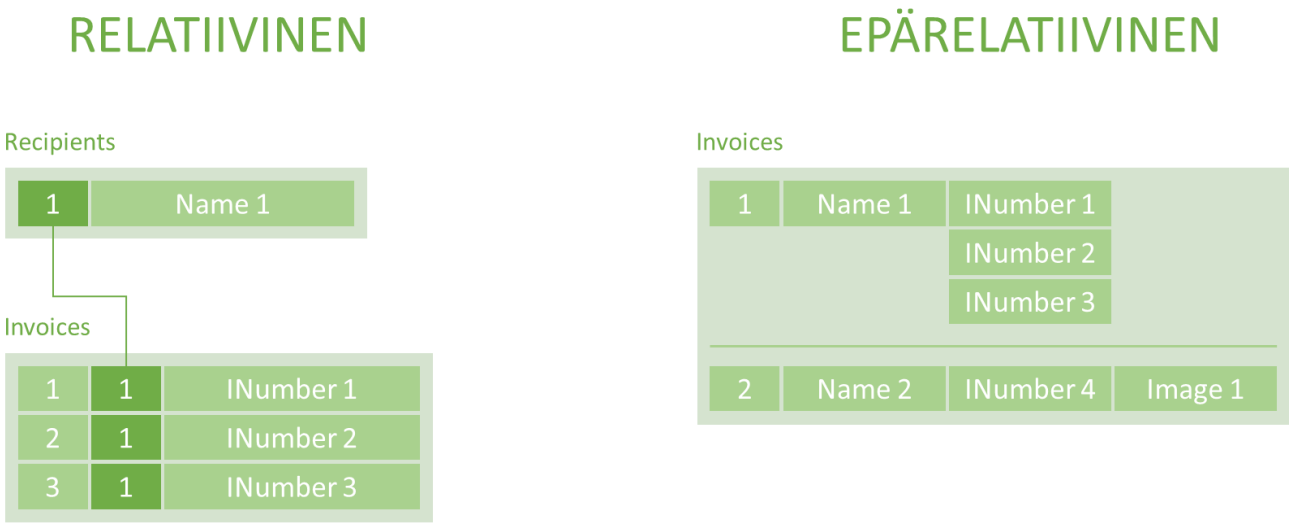
Kuvan 1 SQL-kyselyssä SELECT-lause kertoo tietokannanhallintajärjestelmälle, mitä tietoja halutaan haulla noudettavan. Koska tiedot ovat relatiivisessa tietokannassa taulukoissa, INumber viittaa taulukosta löytyvään samannimiseen sarakkeeseen. Seuraavassa FROM-lauseessa määritellään, mistä tiedot löytyvät. Tässä tapauksessa tietojen pitäisi löytyä taulukosta, jonka nimi on Invoices. Viimeisenä tulee ehtolause eli WHERE-lause. Aiemmissa lauseissa on määritelty, mitä haetaan ja mistä haetaan, mutta vielä viimeisenä voidaan lisätä ehtoja, jotka rajaavat hakua. Emme halua kaikkien laskujen laskunumeroita, vaan ainoastaan niiden, jotka ovat uusia. (Wilton & Colby 2010, 11–13.)

2.3 NoSQL ja epärelatiiviset tietokannat

Termiä NoSQL käytettiin ensimmäisen kerran, kun Carlo Strozzi vuonna 1998 kuvaili omaa tietokannanhallintajärjestelmäänsä, jossa ei käytetty SQL-kyselyitä. Itse tietokantarakenne oli kuitenkin samanlainen kuin muissa SQL-kieltä käyttävissä järjestelmissä eli relatiivinen. (Strozzi 2010.)

Ajan saatossa termi on kuitenkin kehittynyt, ja nykyään sillä tarkoitetaan mitä tahansa yleistä epärelatiivista tietokantaa (KUVIO 2), joka ei myöskään käytä SQL-kieltä tietokantaan tehtävissä kyselyissä (Vaish 2013, 26).

Tarve NoSQL-tietokantojen kehittämiseen tuli Web-ohjelmoinnista ja sen tarpeista. Kävi ilmi, että perinteisillä relatiivisilla tietokannanhallintajärjestelmillä on rajoituksia ja heikkouksia. Näitä ovat pääosin skaalattavuus, kustannustehottomuus ja parallelisaatio, joista jälkimmäinen tarkoittaa järjestelmän heikkoa kykyä suorittaa lukuisia kyselyitä samanaikaisesti. (Vaish 2013, 24.)



KUVIO 2. Relatiivisen ja epärelatiivisen mallin eroavaisuus esimerkki (mukaillen Białecki 2018).

2.4 Epärelatiivisten kantojen esittely

Tämän luvun alaluvuissa käydään läpi eri toteutustapoja, joilla voidaan toteuttaa tietokanta, joka käyttää epärelatiivista mallia tietojen järjestämiseen. Näitä toteutustapoja ovat key-value-, document-, graph- ja in-memory-tietokannat.

2.4.1 Key-value-tietokanta

Data tallennetaan kokoelmana key-value-pareja, joissa key toimii uniikkina tunnisteena. Sekä key että value voivat olla lähestulkoon mitä tahansa: yksittäisiä merkkijonoja, valtavia objekteja ja kaikkea siltä väliltä. Kaikessa yksinkertaisuudessaan key-value-tietokanta muistuttaa muista ohjelmointikielistä tuttuja objekteja, kuten taulukkoa, karttaa ja sanastoa. Key-value poikkeaa näistä objekteista tietenkin siten, että tieto tallennetaan pysyvästi ja sitä hallitaan tietokannan hallintajärjestelmällä. (What is NoSQL?; MongoDB 2020.)

2.4.2 Document-tietokanta

Document-tietokannan toimintaperiaate muistuttaa key-value-tietokantaa, sillä sekin perustuu avaimiin ja arvoihin. Erona kuitenkin on, että document-tietokannassa olevat objektit ovat huomattavasti rikkaampia ominaisuuksiltaan ja tallennettuina erillisiin JSON-tiedostoihin. Tietokanta mukailee samaa dokumenttimallia kuin useimmissa sovelluksessa käytetty koodi, joka helpottaa hakujen muodostamista huomattavasti. (What is NoSQL?; MongoDB 2020.)

2.4.3 Graph-tietokanta

Graph-tietokannassa eri tietojen ja objektien väliset yhteydet ovat erittäin merkittäviä. Tällä pyritään siihen, että uusien tietojen tallentamista ei rajoiteta ennalta määrätyillä tiukoilla rakenteilla. Tietokanta koostuu solmuista ja niiden välisistä suhteista, joista jälkimmäinen on graafitietokannan suurin etu. Hakujen suorittaminen on erittäin nopeaa, koska suhteet ovat pysyviä sen sijaan, että ne laskettaisiin haun aikana. (What is NoSQL?; Neo4j 2020.)

2.4.4 In-memory-tietokanta

Päinvastoin kuin muissa tietokannoissa, joissa data on säilyvää ja tietokokoelma on tallennettu jonkinlaiselle levylle, in-memory-tietokannassa data on järjestelmän työmuistissa. Tämä tarkoittaa sitä, että datan käsittely on erittäin nopeaa, koska sitä ei tarvitse lukea levyltä, vaan se on koko ajan ladattuna ja käytettävissä. Riskinä tässä on, että data menetetään, jos järjestelmässä tapahtuu vikoja tai katkoksia. Datasta saadaan tarvittaessa pysyvää kirjoittamalla lokitiedostoja tai tallentamalla snapshotteja järjestelmän tilasta palvelimen massamuistiin. (What is NoSQL?; In-Memory Database Definition.)

2.5 Tietokantojen eroavaisuudet

Kuten aikaisemmista luvuista on voinut ymmärtää, relatiivinen ja epärelatiivinen malli ja kanta eroavat toisistaan melko huomattavasti. Seuraavissa alaluvuissa näitä eroja tutkitaan hieman tarkemmin ja selitetään, mitä vaikutuksia niillä on tietokannan toimivuuteen ja kustannustehokkuuteen.

2.5.1 Transaktiot

Perinteisessä RDBMS:ssä keskitytään ACID-transaktioihin, joissa datan eheydellä on suuri merkitys, kun taas NoSQL-tietokannanhallintajärjestelmissä keskitytään BASE-transaktioihin, jotka mahdollistavat datan helpomman saatavuuden. (Vaish 2013, 26.)

ACID-lyhenne muodostuu sanoista atomicity (atomisuus), consistency (eheys), isolation (eristyneisyys) ja durability (pysyvyys). Kyseiset käsitteet määritellään seuraavasti:

- Atomisuus: Kaikki transaktiot suoritetaan loppuun asti onnistuneesti, ellei niitä peruuteta.
- Eheys: Tietokantaa kirjoitettavan datan täytyy vastata kaikkia sille määriteltyjä vaatimuksia.
- Eristyneisyys: Yhtäaikaisesti aloitetut transaktiot eivät vaikuta toisiinsa, eikä niitä ajeta rinnakkain, vaan pikemminkin peräkkäin.
- Pysyvyys: Tietokantaan lähetetyt transaktiot suoritetaan loppuun asti, eivätkä ne katoa, vaikka järjestelmä uudelleen käynnistyisi tai muita ongelmia tulisi.

(Vaish 2013, 26; Chan 2019.)

BASE- lyhenne muodostuu sanoista basic availability, soft state, eventual consistency. Nämä käsitteet määritellään seuraavasti:

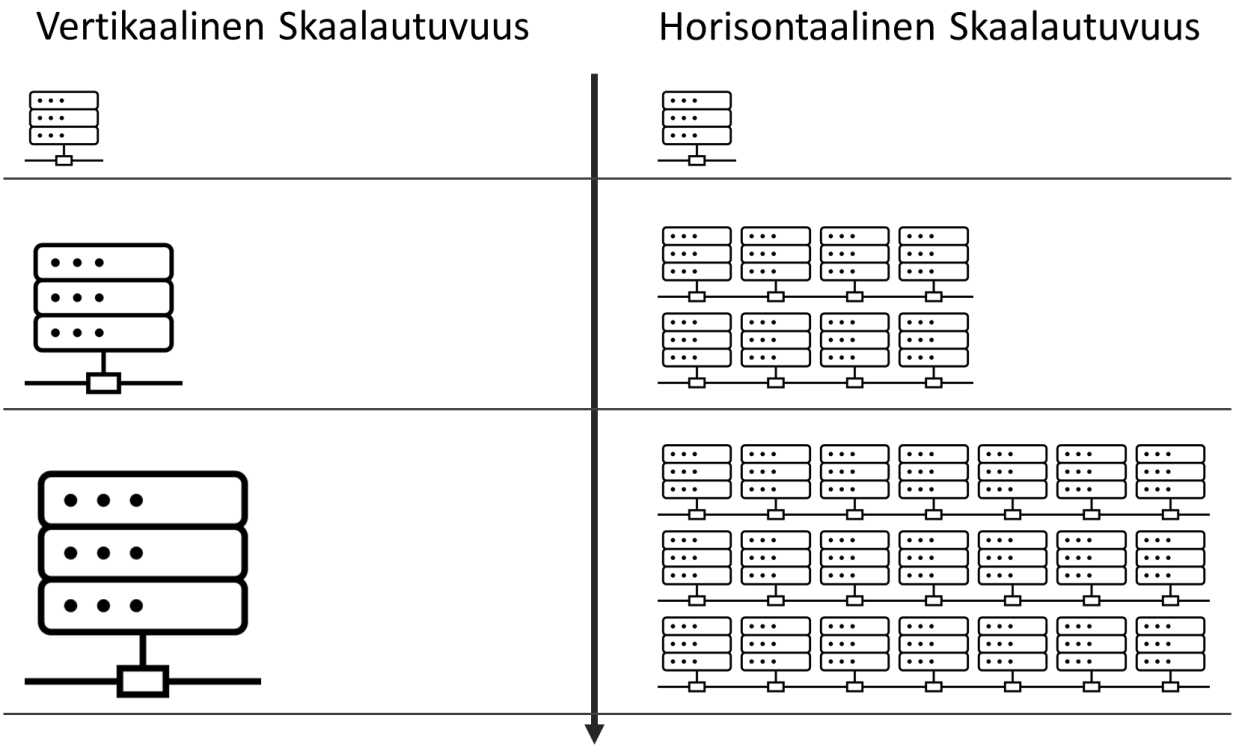
- Basic Availability: Kaikkiin hakuihin vastataan onnistuneesti tai epäonnistuneesti riippumatta tietokannassa olevan datan eheydestä.
- Soft State: Tietokannan tila saattaa muuttua itsestään ilman käyttäjien toimia. Loppujen lopuksi tämä kuitenkin johtaa eheyteen.
- Eventual Consistency: Tietokanta ei välttämättä ole koko ajan eheä, mutta ajan myötä koostuu eheäksi.

(Vaish 2013, 26; Chan 2019.)

2.5.2 Skaalattavuus ja kustannustehokkuus

SQL- ja NoSQL-tietokannat eroavat toisistaan myös skaalattavuudeltaan. SQL on vertikaalisesti skaalautuva (KUVIO 3) lukuun ottamatta poikkeuksia. SQL-tietokantaa ei jaeta useammalle koneelle, koska fokus on aina datan eheyden säilyttämisessä. SQL-tietokanta sijaitsee siis vain yhdellä koneella, eli ainut keino parantaa palvelimen tehokkuutta on päivittää sen osia: CPU, RAM ja SSD. Tämä muodostaa vertikaaliselle skaalaamiselle katon. On olemassa rajat, kuinka tehokkaan yhdestä palvelimesta voi saada perustuen täysin saatavilla olevaan teknologiaan. (Chan 2019.)

Toisin kuin SQL, NoSQL on horisontaalisesti skaalautuva (KUVIO 3). Koska NoSQL-tietokannat eivät ole yhtä tarkasti järjestetyt kuin SQL, ovat ne paljon itsenäisempiä, ja niiden pilkkominen osiin on mahdollista. Tämä tarkoittaa, että NoSQL-tietokanta voi olla jaettuna useammalle palvelimelle, mikä taas tarkoittaa, että kun tarvitaan lisää tehoa, riittää että lisätään palvelimien määrää. Tämä on huomattavasti edullisempaa kuin SQL:n vertikaalinen skaalaus, eikä sillä ole periaatteessa ylärajaa maksimaalista suoritustehoa ajatellen. (Chan 2019.)



KUVIO 3. Vertikaalinen ja horisontaalinen skaalautuvuus (mukaillen Mikelatos 2014).

3 WEB Rajapinta

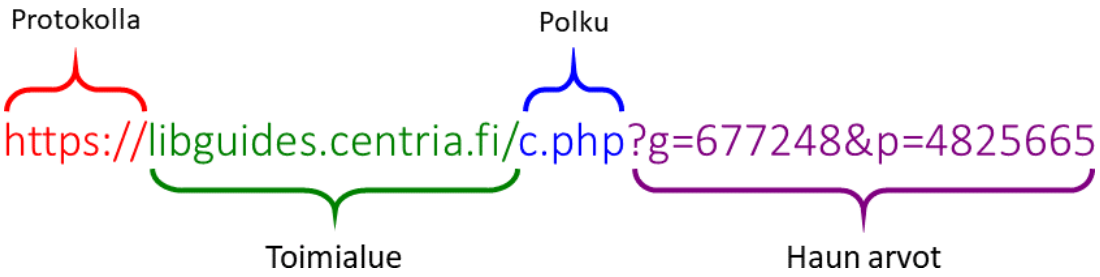
API (Application Programming Interface) on rajapinta, joka määrittelee ja mahdollistaa erilaisten ohjelmien ja systeemien välisen kommunikoinnin. Tämä pätee myös Web API:n kohdalla, mutta tarkemmin kuvailtuna se on kokoelma määritelmiä, jotka ovat useimmiten HTTP (Hypertext Transfer Protocol) -pyyntö- ja vastausviestejä. Näiden viestien rakenne on ennalta määrätty ja yleensä XML-formaatissa (Extensive Markup Language), tai JSON-formaatissa (JavaScript Object Notation). (Rudrakshi, Varshney, Yadla, Kanneganti & Somalwar 2014.)

3.1 HTTP

HTTP (Hyper Text Transfer Protocol) sai syntynsä vuonna 1990 osana WordWideWeb-projektia. HTTP:n ensimmäisen version kehittäjän Sir Tim Berners-Leen tavoitteena oli kehittää HTML:ään perustuva protokolla, jota CERN:n tiedemiehet ja tutkijat voivat käyttää tiedon jakamisessa. Ennen tätä keksintöä informaatio oli tallennettu moniin erilaisiin formaatteihin ja vaati aina vastaavia ohjelmia datan noutamiseen ja muokkaamiseen. Tällainen epästandardisoitu ympäristö teki datan kanssa työskentelystä erittäin työlästä. Berners-Leen alkuperäisiä innovaatiota oli kolme kappaletta: HTML, HTTP ja URL. HTML on merkintäkieli, jolla tekstin viesti ja logiikka erotettiin toisistaan (KUVA 2). HTTP on protokolla, joka määrittelee tavan, jolla data siirretään eri laitteiden välillä. URL on tapa, jonka avulla eri sijainneissa oleviin resurssit voidaan viitata (KUVIO 4). (Cox, Jones & Szumski 2012, 27-34.)

```
<!DOCTYPE html>
<html>
        <head>
                <title>Hello World Example</title>
        </head>
        <body>
                <h1>Hello World!</h1>
        </body>
</html>
```

KUVA 2. Esimerkki HTML:n rakenteesta.



KUVIO 4. Osat, jotka muodostavat URL-osoitteen.

3.1.1 Metodit

HTTP:ssä on määriteltynä kokoelma pyyntömetodeja, jotka osoittavat, millainen toiminto kutsutulle resurssille suoritetaan. Vaikkakin osa näistä metodien nimistä on substantiiveja, niitä kaikkia kutsutaan kuitenkin termillä HTTP-verbi. Jokainen näistä verbeistä suorittaa hieman erilaisen toiminnon, mutta ~~osalla~~ niiden välillä on kuitenkin jonkin verran samanlaisuuksia. (HTTP request methods.)

HTTP-verbejä on yhdeksän ja ne ovat

- GET: Metodi pyytää resurssin datasta esityksen. Tätä käytetään ainoastaan datan noutamiseen.
- HEAD: Lähes samanlainen kuin GET-metodi, mutta vastausviestin runko jätetään pois.
- POST: Tällä metodilla lähetetään dataa tai muutetaan kohderesurssia.
- PUT: Kohderesurssi korvataan viestissä lähetetyllä datalla.
- DELETE: Metodi poistaa kohderesurssin.
- CONNECT: Muodostaa tunnelin palvelimelle.
- OPTIONS: Pyytää kuvailun kohderesurssin kommunikaatioasetuksista.
- TRACE: Metodi suorittaa yhteystestin kohderesurssin ja lähettäjän välillä.
- PATCH: Metodi suorittaa osittaisia muutoksia kohderesurssiin.

(HTTP request methods.)

3.1.2 JSON

JSON (JavaScript Object Notation) on kevyt ja helppolukuinen tiedonvälitysformaatti. Se perustuu osaan JavaScript-ohjelmointikielen standardia EMCA-262 kolmas laitos – Joulukuu 1999. JSON on täysin riippumaton ohjelmointikielestä, joten se on ideaali rajapintojen ja muiden toisistaan poikkeavien järjestelmien väliseen tiedon siirtoon. (Crockford.)

JSON muodostuu kahdesta eri rakenteesta:

- Kokoelma nimi-arvopareja. Monissa kielissä näitä kutsutaan objekteiksi.
- Arvoista muodostettu jäsennelty lista. Useimmissa kielissä tästä käytetään termiä taulukko.

(Crockford.)

Nämä kaksi rakennetta ovat universaaleja ja lähes jokainen moderni ohjelmointikieli tukee niitä jollain tavalla. Tämä rakenteellinen samanlaisuus, johon JSON perustuu, helpottaa eri kielten välisen tiedon siirtämistä rakenteen perusteella. Kuvassa 3 on esimerkki JSON-formaatissa olevista uuden käyttäjätunnuksen luomiseen tarvittavista tiedoista, joka on otettu opinnäytetyön käytännön osuudesta. (Crockford.)

```
{
    "FirstName": "Joonas",
    "LastName": "Kolppanen",
    "Email": "joonas.kolppanen@cou.fi",
    "Password": "345loremipsum#¤%"
}
```

KUVA 3. Uuden käyttäjän tiedot JSON-formaatissa.

3.2 REST- ja RESTful-arkkitehtuurit

REST (Representional State Transfer) ei ole konkreettinen systeemi tai kieli, vaan HTTP:n arkkitehtuurillinen tyyli, jota voidaan käyttää web-rajapintoja suunniteltaessa. Harvat web-rajapinnat tosin käyttävät ainoastaan HTTP-konsepteja, vaan mukaan on yleensä lisätty tyylejä muista teknologioista. Tämän seurauksena on syntynyt uusi termi RESTful, joka kuvaa tällaista yhdistelmäarkkitehtuurimallia. (Mulloy 2016, 8–9.)

3.3 RPC

RPC:n (Remote Procedure Call) avulla toimintoja voidaan suorittaa eri osoitealueessa, josta niitä kutsutaan. Useimmissa tapauksissa kutsuja on päätelaitteen asiakasohjelma ja suorittaja on palvelin.
On olemassa tilanteita, joissa yleisen järjestelmän kehittämisestä on hyötyä. Tässä kokoonpanossa sovellukset ja suoritettava koodi on jaettu usealle eri tarkoitusperiä omaaville palvelimelle tai muulle laitteelle. Tilanne, jossa koodi kutsuu koodia oman sovelluksensa ja järjestelmän ulkopuolelta kutsutaan lyhenteellä RPC. (Fawcett, Quin & Ayers 2012, 539–541.)

Näitä etätoimintokutsuja suorittaessa nousee kuitenkin monia kysymyksiä~~, joista osa on~~:

- Missä kutsuttava koodi sijaitsee?
- Tarvitseeko kutsuttu toiminto jotain arvoja, ja jos tarvitsee niin missä muodossa?
- Palauttaako toiminto arvoja ja jos palauttaa niin missä formaatissa?

Näihin kysymyksiin vastauksena on kehitetty useita RPC-protokollia, jotka määrittelevät kaikki viestin rakennetta, käsittelyä ja muuta koskevat tiedot. Tunnetuimpia RPC-protokollia ovat DCOM, IIOP, Java RMI ja SOAP. (Fawcett, Quin & Ayers 2012, 540.)

3.4 GraphQL

GraphQL on Facebookin kehittämä kyselykieli rajapintojen rakentamista varten. GraphQL-skeema on vahvasti tyypitetty ja tämän ansoista voidaan suorittaa erittäin tarkkoja ja monipuolisia kyselyjä. Toisin kuin RPC:ssä, jossa kuvaillaan toimintoja, tai RESTissä, jossa kuvaillaan resursseja, GraphQL-hauissa kuvaillaan itse hakua ja sen tiedon rakennetta. Käyttäjä määrittelee, mitä muuttujia (henkilö) hän haullaan hakee ja mitä arvoja (etunimi) hän mistäkin muuttujasta haluaa. Haku palauttaa ainoastaan ne muuttujat ja niiden arvot, jotka käyttäjä on kyselyssään määritellyt. Tieto siirtyy laitteelta toiselle JSON-formaatissa. Kuvassa 4 on esimerkki käyttäjän haluamista muuttujista ja niiden arvoista. Kuvassa 5 on kyselyn palauttaman arvot. (GraphQL 2020; Barbettini 2018.)

```
{
    employee {
        FirstName
        LastName
        Email
    }
}
```

KUVA 4. Esimerkki GrapQL-kyselystä ja sen arvoista.

```
{
    "FirstName": "Joonas",
    "LastName": "Kolppanen",
    "Email": "joonas.kolppanen@cou.fi"
}
```

KUVA 5. Kuvan 4 pyynnön vastaus JSON-formaatissa.

3.5 Rajapintojen vertailu

REST-rajapinnan ja muun järjestelmän välinen kytkös on erittäin löyhä, mikä mahdollistaa molempien puolien itsenäisen kehittämisen ilman, että tästä aiheutuu mitään yhteentoimivuusongelmia. Tämä löyhä kytkös myös auttaa tietoturva-asioissa siten, että käyttämällä rajapintaa muusta taustajärjestelmästä on vaikea saada mitään selville. Toinen REST-arkkitehtuurin etu on HTTP-työkalujen uudelleenkäyttö. Osa rajapinnan toimintoihin käytettävistä toiminnallisuuksista ja muista asioista löytyy jo HTTP:stä itsestään. (Barbettini 2018.)

REST:n ongelmiin kuuluu standardisoinnin puuttuminen ja viestien suuret koot. Koska REST on ainoastaan arkkitehtuurillinen tyyli, sen käyttämisestä ja toteuttamisesta ollaan useaa eri mieltä, eikä täten ole olemassa mitään yhtä tiettyä hyväksi koettua ja pitkälle kehitettyä toimintatapaa. REST-viestien isot koot johtuvat viestissä siirtyvästä metadatan suuresta määrästä, joka on järjestelmän toiminnan kannalta välttämätöntä. (Barbettini 2018.)

RPC:n hyödyt ovat yksinkertaisuus, kyselyiden keveys ja tehokkuus. RPC:n käyttö ja toiminta ovat erittäin helppoja sisäistää, sillä se muistuttaa erittäin vahvasti muuta ohjelmointia. Kyselyiden keveys johtaa siihen, että taustajärjestelmä on erittäin tehokas. (Barbettini 2018.)

RPC:n heikkoutena on rajapinnan ja muun taustajärjestelmän välinen tiukka yhteys. Tämä tarkoittaa sitä, että toiminnot ja kutsut saattavat paljastaa järjestelmästä ja sen rakenteesta liikaa ominaisuuksia, mikä aiheuttaa tietoturvariskin. Muita kompastuskiviä ovat erittäin huono löydettävyys ja toimintojen kasaantuminen. Kutsuista ja niiden palauttamista arvoista on erittäin vaikea päätellä, mitä kaikkia toimintoja rajapinnasta löytyy. On siis turvauduttava dokumentaatioon, joka ei ole optimaalista. Toimintojen kasaantumista tapahtuu, kun ajan myötä erilaisten käyttötarkoitusten määrä kasvaa. Toiminnot itsessään eivät ole hirveän joustavia, joten uusia joudutaan tekemään aina aika ajoin, mikä johtaa siihen, että niitä tulee ajan myötä todella monta. (Barbettini 2018.)

GraphQL:n vahvuuksiin kuuluu viestien tarkkuus, joka vähentää vaadittavien kyselyjen määrää sekä yksittäisten viestien kokoa. Tietokannan skeema on myös erittäin pitkälle suunniteltu, mikä tarkoittaa, että rajapinnan käyttö on erittäin helppo opetella. GraphQL soveltuu myös erittäin hyvin graafisien tietokantojen rajapinnaksi. (Barbettini 2018.)

GraphQL:n kompastuskivet ovat monimutkaisuus, väliaikaistietojen tallentaminen ja sen uutuus. GraphQL:n käyttäminen ja asentaminen on huomattavasti monimutkaisempaa RPC- ja REST-rajapinnoissa. Väliaikaistietojen tallentaminen on myös vaikeaa, koska käytössä ei ole HTTP-opittuja metodeja. Tämä johtuu siitä, että suurin osa GraphQL-kyselyistä käyttää POST-toimintoa, jonka väliaikatietojen keräämiselle ei ole valmiita toiminnallisuuksia, vaan tälle on tehtävä jokin oma järjestelmä. GraphQL on myös erittäin uusi teknologia, joka johtaa siihen, että yleisiä käytäntötapoja ja muuta ei ole vielä kehitetty. Esimerkkinä versioinnista ei sen hallitsemisesta olla vielä yhtä mieltä. (Barbettini 2018.)

4 TOTEUTUS

Toteutuksen tavoitteena on suunnitella ja kehittää järjestelmä, joka koostuu tietokannasta ja rajapinnasta, sekä testata tämän järjestelmän toimivuutta. Tietokanta tulee olemaan relatiivinen ja rajapinta RESTful-arkkitehtuuria noudattava. Nämä päätökset on tehty opinnäytetyön teoreettisessa viitekehyksessä ilmenneiden havaintojen perusteella. Toteutuksessa käytettävät ohjelmistot ja teknologiat ovat Microsoft SQL Server, Visual Studio 2018 Community Edition, ASP.NET Web API ja Internet Information Services Express (IIS Express).

Koska ohjelmassa tullaan tarkastamaan ja hyväksymään laskuja sekä muita rahaan liittyviä asioita, on ensisijaisen tärkeää, ettei data korruptoidu tai tallennu väärin missään vaiheessa prosessia. Suuri käyttäjämäärä ei ole myöskään ongelma, sillä ohjelma on suunniteltu ainoastaan yrityskäyttöön. Järjestelmän ei siis tarvitse olla skaalattavissa eikä tehokas, vaan erittäin helposti hallittavissa ja tarvittaessa räätälöitävissä. Tietokannan tyypiksi opinnäytetyössä on valittu relatiivinen tietokantamalli. Relatiivisia tietokannanhallintajärjestelmiä (RDBMS) on monia, mutta tässä opinnäytetyössä tietokannan hallitsemiseen käytetty ohjelma on Microsoft SQL Server Management Studio, joka käyttää Microsoftin omaa versiota SQL-kyselykielestä nimeltä Transact-SQL.

Rajapinnan tavoitteina on olla erittäin turvallinen, helppo ja universaali. Joustavuutta ja helppokäyttöisyyttä tarvitaan siksi, että lopputyöhön ei kuulu tätä taustaohjelmaa käyttävän sovelluksen kehittäminen, joten kaikki ovet kannattaa pitää avoinna soveltuvien ohjelmointikielien ja päätelaitteiden osalta. Rajapinnan turvallisuus on myös erittäin tärkeää, koska kyse on yritystoiminnassa käytettävästä pankkiohjelmasta. Näiden vaatimuksien pohjalta rakennetaan Web-rajapinta, jossa noudatetaan RESTful-arkkitehtuurimallia. Kehitysympäristöksi valitaan ASP.NET Web API, koska sekin on Microsoftin kehittämä, joten se on jo ennestään erittäin yhteensopiva käytettävän tietokannan kanssa.

4.1 Suunnittelemisen aloittaminen

Tämänkaltaista ohjelmistoa, joka koostuu useammasta eri osasta, voidaan lähteä suunnittelemaan monella eri tavalla. Projektin alussa tehtiin valinta code-first- ja data-first-lähestymistapojen väliltä. Code-fist-lähestymistavassa tietokanta luodaan vasta rajapinnan toiminnallisuuksien luomisen jälkeen, kun taas data-first-lähestymistavassa aloitetaan tietokannan suunnittelemisella ja toteuttamisella. (Hedge 2017.) Tämän opinnäytetyön käytännönosuudessa käytettiin data-first-tapaa, sillä relatiivisen tietokannan eheys ja normalisointi oli erittäin tärkeää. Työn alussa saatiin myös neuvoja taloushallintoalalla työskentelevältä henkilöltä, mikä helpotti tietokannan suunnittelua.

4.2 Tietokanta

Tulevissa alaluvuissa kerrotaan tietokannan luomisprosessista ja siitä, kuinka eri taulut ja niiden väliset liitokset on luotu käyttötarpeiden perusteella. Tietokantojen luominen on myös koko ohjelmointityön ensimmäinen vaihe, koska käytössä on data-first-lähestymistapa.

4.2.1 Jäsentaulut

Suunnittelu aloitettiin kartoittamalla, millaisia toimintoja loppukäyttäjä haluaa mahdollisesti suorittaa. Tämä tietokanta on suunniteltu näiden vaatimuksien pohjalta. Tietokannassa tulee siis löytyä ainakin tarkastajia ja laskuja, joita nämä tarkastajat hyväksyvät. Laskut maksetaan joltain tililtä eli täytyy löytyä myös pankkitilejä. Tämän perusteella tarvitaan siis seuraavat taulut: tarkastaja (person), maksutili (account) ja lasku (invoice) (KUVIO 6).

4.2.2 Junktiotaulut

Sen lisäksi, että tietokannassa on tarkastajia, tilejä ja laskuja, täytyy ne kytkeä toisiinsa jotenkin. Loppukäyttäjän täytyy pystyä määrittelemään, mitkä laskut ovat kunkin tarkastettavissa. Koska järjestelmää käyttää useampi henkilö yhtäaikaisesti, olisi myös todella hyvä, että käyttäjät voisivat kommunikoida keskenään ja kirjoittaa muistiinpanoja. Toisin sanottuna kommenttien kirjoittamista ja hyväksyntävastuun jakamista varten on tehtävä taulut tarkastajan ja laskun välille. Tämän lisäksi

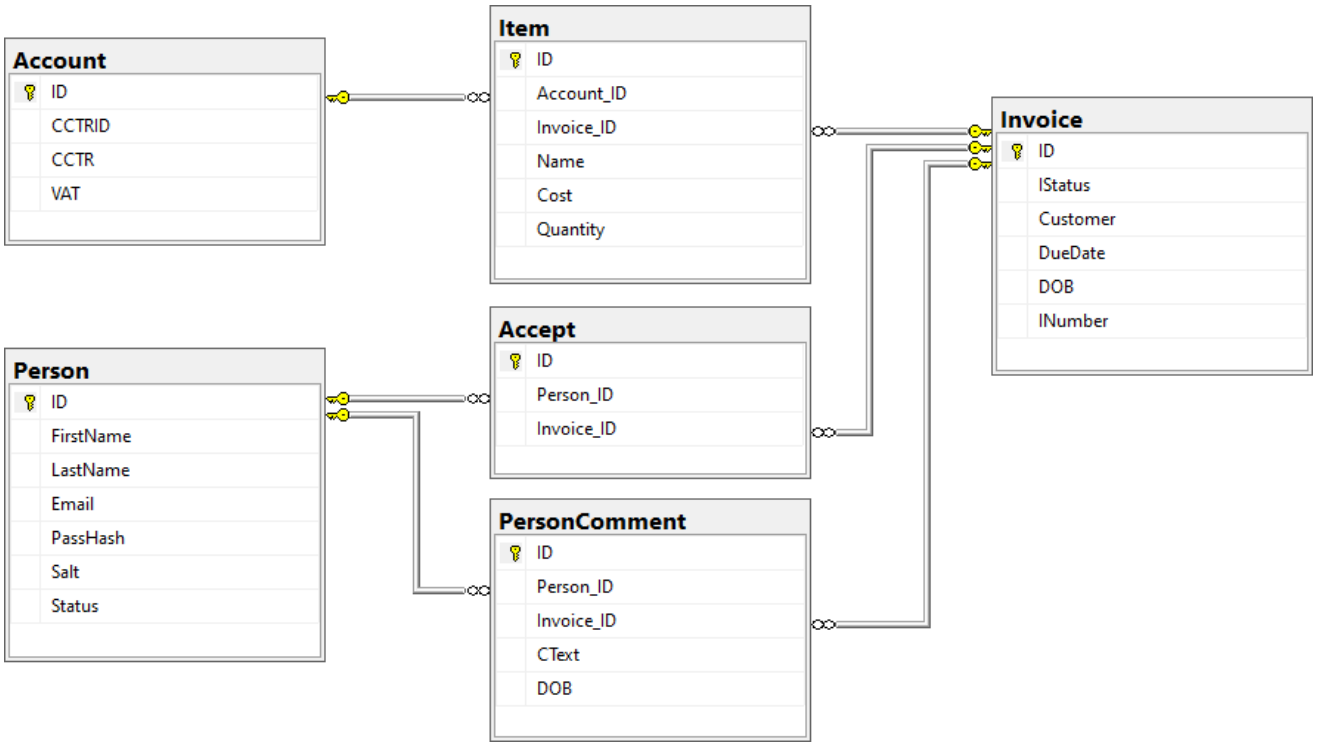
tarkastajien täytyy nähdä, miltä tililtä niitä ollaan maksamassa. Maksutili- ja laskutaulut voisi muuten yhdistää, mutta on olemassa tilanteita, joissa laskun lukuisat eri jäsenet pitää pystyä erottaa toisistaan, jos ne tahdotaan maksaa eri tileiltä. Tämä tarkoittaa sitä, että laskutaulua ei voida yhdistää suoraan maksutilitauluun. Näiden kolmen jäsentaulun välille täytyy luoda junktiotaulut: kommentti (comment), hyväksyntä (accept), maksurivi (item) (KUVIO 6).

4.2.3 Pää- ja vierasavaimet

Nyt kun tietokantaan on luotu taulut Person, Account, Invoice, Item, Accept ja Comment, täytyy ne vielä yhdistää jotenkin. Tämä onnistuu pää- ja vierasavaimilla. Pääavain on taulussa oleva sarake, joka on uniikki eli millään rivillä ei ole samaa arvoa. Tämä pääavain kytketään toisesta taulusta löytyvään sarakkeeseen, joka on vierasavain. Näillä kahdella avaimella on identtiset arvot ja tämän perusteella molemmista tauluista voidaan hakea dataa. Tässä tietokannassa kaikista yksinkertaisin tapa on yhdistää tietokannat ID-sarakkeiden perusteella. Taulun oma pääavain on aina nimeltään ID, ja vierasavaimen tunnistaa sen nimessä olevasta etuliitteestä, joka on taulun nimi, johon se on kytketty. Esimerkkinä kommenttitaulussa viitataan kommentin jättäneen tarkastajan ID:seen kentässä Person_ID (KUVIO 6).

4.2.4 Tietokannan normalisointi

Kun tietokannan ensimmäinen versio on valmis, ja se sisältää kaikki tarvittavat taulut ja niiden väliset yhteydet, jotka on luotu pää- ja vierasavaimilla, täytyy tietokanta vielä normalisoida. Normalisoinnin tavoitteena on etsiä tietokannasta epäkohtia, jotka voivat aiheuttaa datan toistumista sekä eheyden rikkoutumista. Kun viat on löydetty, ne korjataan muokkaamalla tietokannan rakennetta. Näiden epäkohtien löytämisessä käytetään säännöstöä, joka koostuu neljästä normaalimuodosta. (Mooc.fi 2019.)



KUVIO 6. Kuva harjoitustyön tietokannan tauluista ja niiden välisistä liitoksista.

4.3 Rajapinta

Ohjelmointiprojektin toinen osuus on rajapinnan suunnitteleminen. Edellisessä luvussa luotujen tietokantojen käyttämistä varten kehitetään rajapinta, jonka avulla loppukäyttäjä pääsee käsiksi tietokantaan tallennettuun informaatioon.

4.3.1 ASP.NET WEB API

Rajapinnan koodaaminen suoritettiin Visual Studiolla ja Microsoftin kehittämällä ASP.NET frameworkilla. Koska REST on ainoastaan arkkitehtuurillinen malli, ei käytettävällä ohjelmalla ja kirjastoilla ole suurta merkitystä lopputuloksen kannalta. Tästä valinnasta saattaa olla hyötyä myös joskus myöhemmin yhteensopivuutta ja muuta sellaista ajatellessa, koska tietokanta on myös Microsoftin kehittämällä järjestelmällä pystytetty.

4.3.2 Tietokantayhteys

Jotta rajapinta saadaan toimimaan, on se yhdistettävä tietokantaan. Tätä varten rajapintaan on tehty DataAccess.cs-luokka, josta löytyvät SQL-kyselyiden kutsumiset. Rajapinnassa itsessään ei luoda SQL-kyselyitä, vaan tietokantaan on luotu valmiiksi nämä kyselyt. DataAccess.cs-luokassa näitä tietokannasta löytyviä kyselyitä kutsutaan ja tässä kutsussa matkaan annetaan argumentit, eli arvot, joilla päätetään lopputulos. Kuvassa 6 on esimerkki DataAccess.cs-luokasta ja toiminnosta, joka noutaa tietokannasta yhden käyttäjän tiedot argumenttina syötetyn ID:n perusteella.

```
6 references
public class DataAccess
{
    3 references | 0 exceptions
    public static string ConString(string name)
    {
        return ConfigurationManager.ConnectionStrings[name].ConnectionString;
    }


    0 references | 0 exceptions
    public Person GetPersonByID(int ID)
    {
        using (IDbConnection connection = new System.Data.SqlClient.SqlConnection(ConString("MobileConnector"))) {
            var list = connection.Query<Person>("dbo.person_getbyid @ID", new { ID }).ToList();
            if (list != null && list.Any())
                return list[0];
            else {
                return null;
            }
        }
    }
```

KUVA 6. Rajapinnan DataAccess.cs jolla luodaan yhteys tietokantaan.

4.3.3 Datamallit

Jotta tietokannasta haettavia tietoja olisi helpompi tallentaa välimuistiin ja kaiken kaikkiaan käsitellä, luotiin ohjelmaan datamalleja, jotka ovat C#-luokkia. Kuvassa 7 on rajapinnan Person-datamallista ja sen luomisesta JSON-taulun sisältämien arvojen avulla.

```
16 references
public class Person
{
    2 references | 0 exceptions
    public int ID { get; set; }
    4 references | 0 exceptions
    public string FirstName { get; set; }
    4 references | 0 exceptions
    public string LastName { get; set; }
    5 references | 0 exceptions
    public string Email { get; set; }
    4 references | 0 exceptions
    public string PassHash { get; set; }
    5 references | 0 exceptions
    public string Salt { get; set; }
    1 reference | 0 exceptions
    public int Active { get; set; }

    0 references | 0 exceptions
    public Person() { }

    1 reference | 0 exceptions
    public Person(JObject data)
    {
        FirstName = data.GetValue("FirstName").ToString();
        LastName = data.GetValue("LastName").ToString();
        Email = data.GetValue("Email").ToString();
        Salt = BCryptHelper.GenerateSalt();
        PassHash = BCryptHelper.HashPassword(data.GetValue("Password").ToString(), Salt);
    }
```

KUVA 7. Rajapinnan Person.cs, joka on henkilön datamalli.

4.3.4 Ohjaimet

Rajapinnasta löytyy niin sanottuja ohjaimia, joiden sisälle on kirjattu erilaisia rajapinnan toimintoja. Ohjaimet määrittelevät, mitä toimintoja on suoritettava, jos loppukäyttäjä kutsuu tiettyä URL-osoitetta tietynlaisella metodilla. Kuvassa 8 on AuthController.cs-ohjaimesta, josta löytyvät kaikki kirjautumiseen ja tunnistautumiseen liittyvät toiminnot sekä osoitteet, joista näitä toimintoja voi kutsua.

```
[RoutePrefix("api/auth")]
0 references
public class AuthController : ApiController
{


    [Route("register")]
    0 references | 0 requests | 0 exceptions
    public IHttpActionResult Insert(JObject args)
    {
        DataAccess db = new DataAccess();
        Person person = new Person(args);
        JObject result = new JObject();

        if (db.InsertPerson(person)) {
            result.Add("Message", "Registration Succeeded");
            return Content(HttpStatusCode.Created, result);
        }
        else
            result.Add("Message", "Registration Failed");
            return Content(HttpStatusCode.BadRequest, result);
    }
```

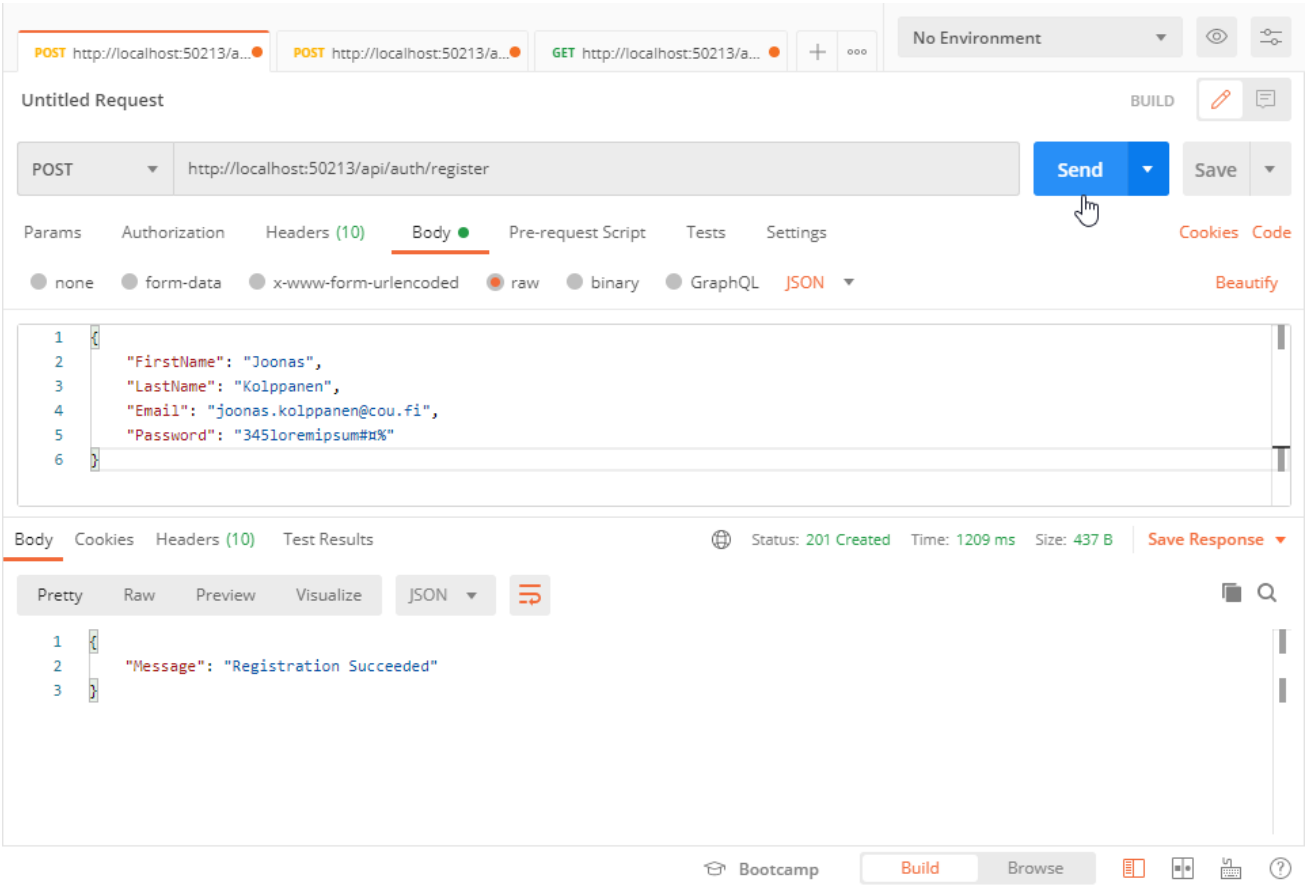
KUVA 8. Rajapinnan AuthController.cs, jossa määritellään kirjautuminen ja tunnistautuminen.

5 TESTAAMINEN

Kun tietokanta on saatu luotua ja rajapinta rakennettua, aloitetaan testaaminen. Testaamistarkoituksiin käytetään tällä kertaa Postman-nimistä ohjelmaa, jolla voidaan tehdä suoraan HTTP-protokollaa käyttäviä kutsuja rajapinnan URL-resursseihin. Rajapinnalle ei ole kehitetty omaa käyttöliittymään testaamista varten, koska se ei ole tarpeen, sillä rajapinnan testaamisessa voidaan käyttää hyödyksi tällaisia kolmannen osapuolen ohjelmia. (Postman 2020.)

5.1 Rekisteröityminen

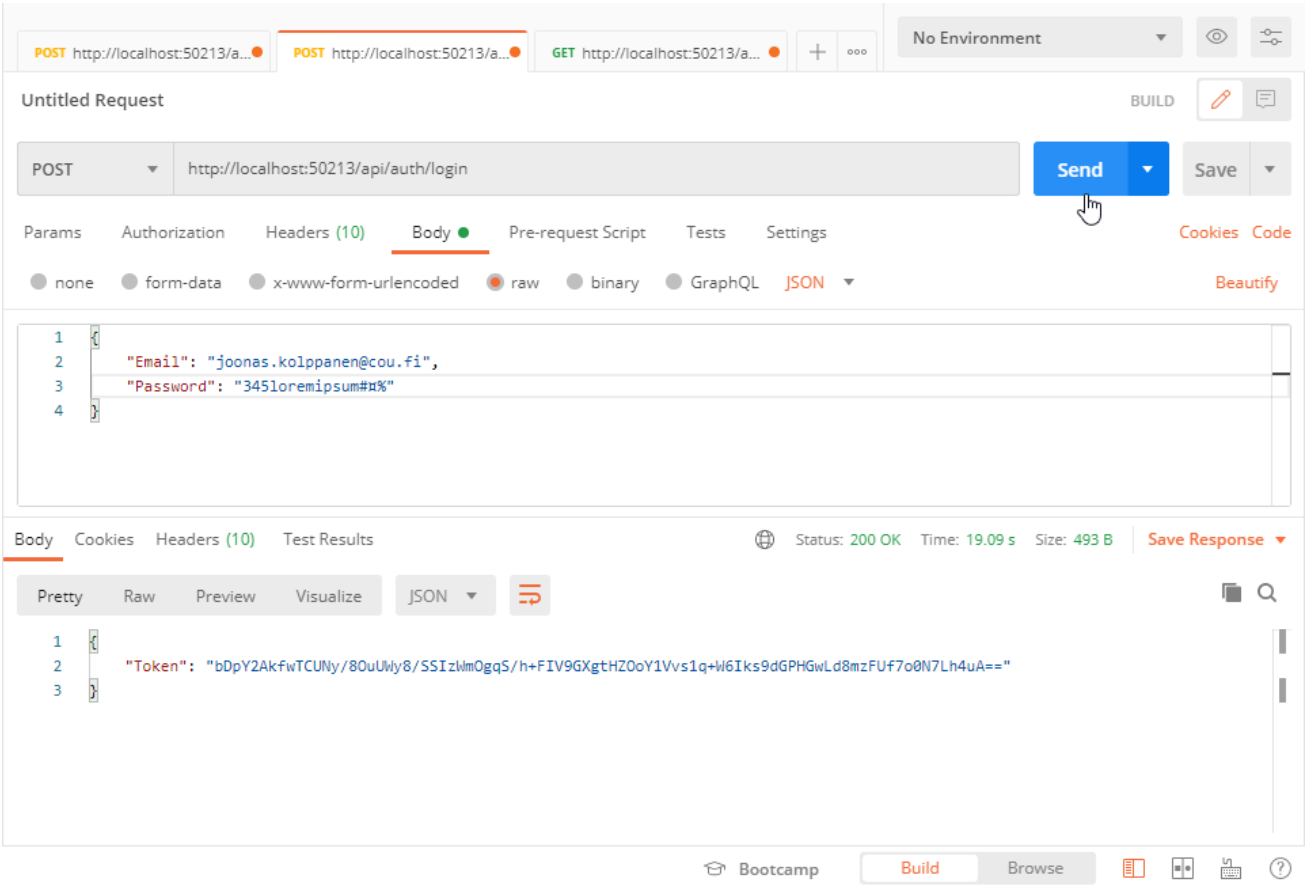
Ensimmäisessä testissä kokeillaan, kuinka uusi käyttäjä lisätään järjestelmään. Kuvan 9 ylemmässä osassa näkyy viestin metodi, joka on POST ja BODY, jossa on uuden käyttäjän tiedot JSON-formaatissa. Viesti lähetetään URL-osoitteeseen *http://localhost:50213/api/auth/register*. Kohta: *"/api/auth"* ohjaa viestin ja sen tiedot tunnistautumiskontrolleriin (KUVA 8) ja *"/register"*-toimintoon nimeltä Insert (KUVA 8). Tietokantaan yritetään luoda uusi käyttäjä viestissä saaduilla tiedoilla ja jos se onnistuu, rajapinta palauttaa vastausviestin, jonka tilakoodi on 201 eli Created sekä JSON-formaatissa olevan viestin, joka varmistaa, että rekisteröityminen on onnistunut. Tämä vastausviesti ja sen sisältö näkyvät kuvan 9 alaosassa.



KUVA 9. Uuden käyttäjän luominen tietokantaan käyttäen rajapinnan toimintoja.

5.2 Kirjautuminen

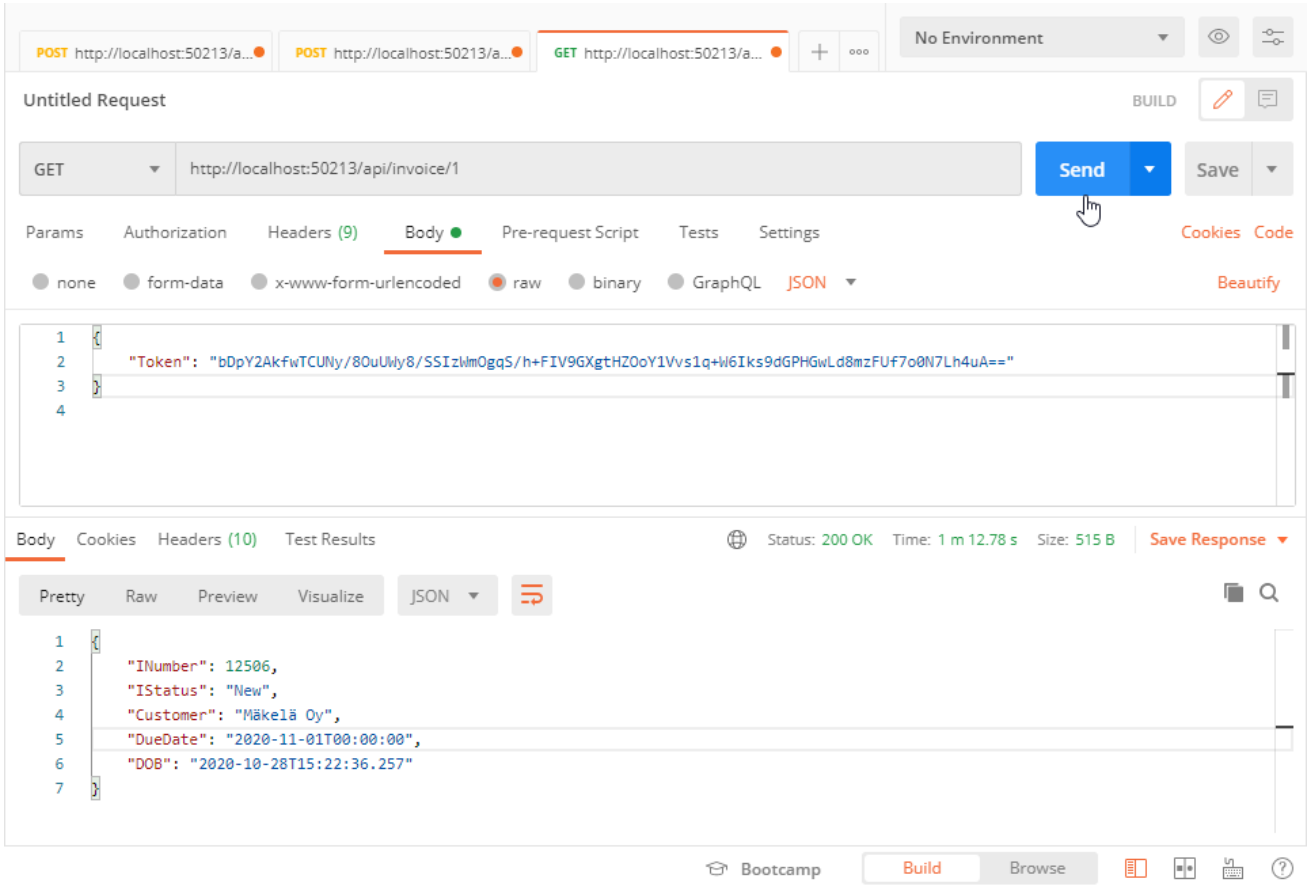
Toisessa testissä rajapintaan kirjaudutaan edellisessä vaiheessa luoduilla uusilla käyttäjätunnuksilla. Pyyntöviestin tyyppi on samanlainen, mutta tällä kertaa pyyntöviestissä ei tarvitse lähettää muita tietoja kuin käyttäjän sähköposti ja salasana. Salasana kryptataan ja sitä verrataan tietokannasta löytyvään, aikaisemmin kryptattuun salasanaan. Jos nämä ovat identtiset, kirjautuminen onnistuu ja rajapinta muodostaa ja palauttaa käyttäjälle tokenin. Token on pitkä kryptattu merkkijono, joka koostuu kirjautumiseen käytettävistä ja muista erilaisista saatavilla olevista tiedoista, esimerkiksi käyttäjän ID:stä, tokenin eliniästä, tokenin luomishetkestä tai päätelaitteen selaimen tiedoista (KUVA 10). Nämä tiedot ympätään sellaisinaan yhdeksi pitkäksi merkkijonoksi, joka suojataan kryptaamalla. Tokenin kelpaavuus tarkistetaan dekryptaamalla se ja tutkimalla merkkijonon sisällön eheyttä ja paikkansapitävyyttä.



KUVA 10. Käyttäjän kirjautuminen ja tokenin saaminen.

5.3 Laskun hakeminen

Kun uusi käyttäjä on luotu ja kirjautuminen on onnistunut, voidaan tietokannasta hakea laskuja. Rajapinta vaatii toimiakseen voimassa olevan tokenin. Rajapinta yrittää dekryptata tokenin ja jos se onnistuu, tarkistetaan, ovatko tokeniin tallennetut tiedot kelpaavia. Mikäli näin on, rajapinta hakee tietokannasta laskun, jonka ID on 1. Rajapinta katsoo tämän ID-numeron HTTP-viestin URL-osoitteesta, joka on tässä esimerkissä numero 1. Jos rajapinta löytää laskun tietokannasta oikean laskun, se palauttaa sen tiedot JSON-formaatissa (KUVA 11).



KUVA 11. Tunnistautuminen tokenin avulla ja laskun noutaminen tietokannasta ID:n perusteella.

6 YHTEENVETO

Yksi opinnäytetyön tavoitteista oli tietokantojen ja rajapintojen tutkiminen ja vertailu. Toinen tavoite opinnäytetyössä oli toteuttaa tietokanta ja rajapinta taloushallinnossa käytettävälle laskujenhyväksyntäjärjestelmälle. Teknologiat ja toteutustavat valittiin opinnäytetyön alussa olevassa tutkimisessa tehtyjen havaintojen perusteella.

Taloushallinnon tarpeita ajatellen tietokannan rakenne jäi hieman vajaaksi puuttuvien speksauksien ja muun tiedon takia. Tämä ei kuitenkaan haitannut tavoitteiden saavuttamisessa, sillä tietokantaa ei kehitetty käyttöä varten, vaan ainoastaan mockup-mielessä relatiivisen mallin toiminnan testaamiseksi. Tietokantavalinta oli erittäin toimiva.

Rajapinta kehitettiin ASP.NET-frameworkilla, joka oli ihan hyvä valinta, sillä sen yleisyys tarkoittaa, että sen käyttöön ja muuhun vastaavaan löytyi paljon apua netistä. Rajapinnan oli tarkoitus mukailla RESTful-arkkitehtuuria, mutta tämä ei täysin toteutunut. RESTin ja RESTfulin yksi suurimmista vahvuuksista jäi hyödyntämättä, kun rajapinnan kutsut olivatkin toimintoja eivätkä pelkkiä URL-resursseja. Suurin ongelma tästä tulee, kun rajapintaa yritetään hyödyntää muissa ohjelmissa. Se ei noudata mitään helposti opittavia standardeja, vaan rajapinnan dokumentoinnista on opeteltava kaikki toiminnallisuus, jota halutaan käyttää ohjelmassa. Idea rajapinnasta oli hyvä, mutta itse toteutus oli väärä.

LÄHTEET

AWS. 2020. What is NoSQL? Saatavissa: https://aws.amazon.com/nosql/. Viitattu 27.10.2020.

Barbettini, N. 2018. Nate Barbettini – API Throwdown: RPC vs REST vs GraphQL, Iterate 2018. Saatavissa: https://www.youtube.com/watch?v=IvsANO0qZEg. Viitattu 14.12.2018.

Białecki, M. 2018. Relational vs non-relational databases. Saatavissa: https://www.michalbialecki.com/2018/03/16/relational-vs-non-relational-databases/. Viitattu 24.11.2020.

Chamberlin, D. & Boyle, R. 1974. SEQUEL: A Structured English Query Language.

Chan, M. 2019. SQL vs. NoSQL – what's the best option for your database needs? Saatavissa: https://www.thorntech.com/2019/03/sql-vs-nosql/. Viitattu 27.10.2020.

Codd, E. 1970. A Relational Model of Data for Large Shared Data Banks.

Cox, J., Jones N. & Szumski, J. 2012. Professional IOS Network Programming. John Wiley & Sons, Incorporated.

Fawcett, J., Quin, L. & Ayers, D. 2012. Beginning XML. John Wiley & Sons, Incorporated.

Foote, K. 2018. A Brief History of Non-Relational Databases. Saatavissa: https://www.dataversity.net/a-brief-history-of-non-relational-databases/. Viitattu 27.10.2020.

GraphQL. 2020. Introduction to GraphQL. Saatavissa: https://graphql.org/learn/. Viitattu 27.10.2020.

Hedge, G. 2017. Untity Framework Code First v/s Database First Approach. Saatavissa: https://www.c-sharpcorner.com/blogs/entity-framework-code-first-vs-database-first-approach#:~:text=In%20code%20first%20approach%20we,run%20database%20will%20get%20created.. Viitattu 25.11.2020.

IBM. 2020. IBM Archives: 1970s. Saatavissa: https://www.ibm.com/ibm/history/history/year_1978.html. Viitattu 27.10.2020.

Introducing JSON. Saatavissa: https://www.json.org/json-en.html. Viitattu 27.10.2020.

Mikelatos, S. 2014. Designing your application for growth. Saatavissa: https://blog.leaseweb.com/2014/05/22/designing-application-growth/. Viitattu 24.11.2020.

MongoDB. 2020. Key-Value Databases. Saatavissa: https://www.mongodb.com/key-value-database. Viitattu 27.10.2020.

Mooc.fi. 2019. Tietokannan normalisointi. Saatavissa: https://tietokantojen-perusteet-19.mooc.fi/osa-4/1-tietokannan-normalisointi. Viitattu 27.10.2020.

Mozilla. 2020. HTTP request methods. Saatavissa: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods. Viitattu 27.10.2020.

Mulloy, B. 2016. Web API Design – Crafting Interface that Developers Love. Saatavissa: https://pages.apigee.com/rs/351-WXY-166/images/Web-design-the-missing-link-ebook-2016-11.pdf. Viitattu 27.10.2020.

Neo4j. 2020. What is a Graph Database? Saatavissa: https://neo4j.com/developer/graph-database/. Viitattu 27.10.2020.

OmniSci. DBMS Definition. Saatavissa: https://www.omnisci.com/technical-glossary/dbms#:~:text=A%20Database%20Management%20System%20(DBMS,manage%20data%20in%20a%20database.. Viitattu 27.10.2020.

OmniSci. 2020. In-Memory Database Definition. Saatavissa: https://www.omnisci.com/technical-glossary/in-memory-database. Viitattu 27.10.2020.

Postman. 2020. Exploratory Testing. Saatavissa: https://www.postman.com/use-cases/exploratory-testing/. Viitattu 25.11.2020.

Rudrakshi, C., Varshney, A., Yadla, B., Kanneganti, R. & Somalwar, K. 2014. Api-fication. Saatavissa: http://www.hcltech.com/sites/default/files/apis_for_dsi.pdf. Viitattu 27.10.2020.

Stephens, R. 2008. Beginning Database Design Solutions. John Wiley & Sons, Incorporated.

Strozzi, C. 2010. NoSQL: a non-SQL RDBMS. Saatavissa: http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page. Viitattu 27.10.2020.

Vaish, G. 2013. NoSQL Starter. Packt Publishing, Limited.

Wilton, P. & Colby, J. 2010. Beginning SQL. John Wiley & Sons, Incorporated.