Tietokantakirjaston migraatio teknisen velan purkamiseksi: Vanhentuneen Slick-kirjaston korvaaminen ScalikeJDBC-kirjastolla

Jouni Johansson

Haaga-Helia ammattikorkeakoulu
Tradenomi, tietojenkäsittely
Amk-opinnäytetyö
2024

Tiivistelmä

Tekijä
Jouni Johansson

Tutkinto
Tradenomi

Raportin/Opinnäytetyön nimi
Tietokantakirjaston migraatio teknisen velan purkamiseksi: Vanhentuneen Slick-kirjaston korvaaminen ScalikeJDBC-kirjastolla

Sivu- ja liitesivumäärä
45 + 20

Tässä opinnäytetyössä toteutettiin ja dokumentoitiin tietokantakirjaston migraatio Slick-kirjastosta ScalikeJDBC-kirjastoon Viite-nimisessä sovelluksessa. Työn tarve syntyi teknisen velan vähentämisestä, sillä vanhentuneen Slick-kirjaston versio esti Scala-ohjelmointikielen ja muiden teknologioiden päivittämisen. Migraation tavoitteena oli toteuttaa tietokantakerroksen päivitys siten, että sovelluksen matalatasoinen SQL-lähestymistapa ja monimutkaiset transaktioketjut säilyvät toimivina, eikä arkkitehtuuri vaadi suuria muutoksia. Opinnäytetyössä tutkittiin ja dokumentoitiin tarkemmin valikoituja sovelluksen osia, jotka edustivat kattavasti erilaisia tietokantaoperaatioita.

Työn tietoperusta rakentui tietokantaoperaatioiden, tietokantakirjastojen ja transaktioiden sekä näihin liittyvien tietoturvakäytäntöjen ympärille. Työssä tutkittiin erityisesti Scalan soveltuvuutta tietokantaohjelmointiin sekä matalan tason SQL-lähestymistavan ja korkeamman tason ORM-tekniikoiden eroja. Tutkimusmenetelminä käytettiin kirjallisuuskatsausta, nykytilanteen analyysiä, testivetoista kehitystä sekä kokeilevaa lähestymistapaa. Migraatio toteutettiin vaiheittain luomalla ensin tarvittavat apukomponentit istunnonhallintaa ja tietokantaoperaatioita varten, minkä jälkeen tietokantakerros ja palvelukerros päivitettiin yksikkötestejä hyödyntäen.

Migraation tuloksena syntyi toimiva ScalikeJDBC-pohjainen tietokantakerros, joka paransi sovelluksen tietoturvaa sekä selkeytti tietokantakoodia säilyttäen samalla matalan tason SQL-toteutukset. Merkittävimpiä saavutuksia olivat tietokantaistuntojen hallintamekanismin kehittäminen, SQL-injektioiden estäminen parametrisoiduilla kyselyillä sekä virheenkäsittelyn parantaminen. Työn johtopäätöksenä todettiin, että ScalikeJDBC soveltuu erinomaisesti matalan tason SQL-toteutuksia hyödyntävien sovellusten modernisointiin. Migraation myötä Viite-sovelluksen elinkaari piteni merkittävästi, kun se mahdollisti koko sovelluksen teknologiapinon päivittämisen tulevaisuudessa.

Asiasanat
Tietokantakirjastot, migraatio, ScalikeJDBC-kirjasto, Scala, Tekninen velka, Slick-kirjasto

Sisällys

1 Johdanto

Tämä opinnäytetyö käsittelee Scala-pohjaisen sovelluksen tietokantakirjaston migraatiota, jossa Slick-kirjasto korvataan ScalikeJDBC-kirjastolla. Opinnäytetyö toteutetaan toimeksiantona Sito-wise-yritykselle projektiin, jossa toimin sovelluskehittäjänä. Aihe syntyi todellisesta tarpeesta, kun Viite-nimisen sovelluksen kehityksessä havaittiin vanhan tietokantakirjaston rajoittavan sovelluksen kehitystä ja ylläpitoa. Sovelluksen teknistä velkaa purettaessa Slick-kirjaston vanhentuneen version todettiin muodostaneen pullonkaulan sovelluksen ylläpidettävyydelle ja skaalautuvuudelle. Vanhentunut Slick-kirjasto estää Scala -ohjelmointikielen version päivittämisen ja sen myötä myös monien muiden Scala-versiosta riippuvien teknologioiden pitämisen ajan tasalla.

Sen lisäksi, että vanhentuneet kirjastot hankaloittavat muiden teknologioiden pitämistä ajan tasalla ja voivat sisältää bugeja sekä yhteensopivuusongelmia, vanhentuneissa kirjastoissa piilevät haavoittuvuudet luovat merkittäviä tietoturvariskejä sovelluksille. Ohjelmistokirjastoihin julkaistaan jatkuvasti tietoturvapäivityksiä, jotka korjaavat havaittuja haavoittuvuuksia, mutta jos sovellus käyttää vanhentunutta versiota kirjastosta, se jää alttiiksi näille tunnetuille haavoittuvuuksille. Tunnetut haavoittuvuudet ovat erityisen vaarallisia, sillä ne ovat julkisesti raportoituja ja siten myös mahdollisten hyökkääjien tiedossa ja hyödynnettävissä. (Podjarny 2017, 2018.)

Viite-sovelluksen Slick-tietokantakirjaston versiota ei olla päivitetty tuoreempaan, koska sovelluksen nykyinen toteutus perustuu vahvasti matalan tason JDBC-pohjaiseen SQL-koodiin, kun taas uudemmat Slick-versiot ovat siirtyneet kohti korkeamman tason abstraktioita ja funktionaalista ohjelmointityyliä. Uudempiin Slick-kirjaston versioihin päivittäminen vaatisi laajoja muutoksia tietokantaa käsittelevän koodin rakenteeseen ja tapaan kirjoittaa koodia jatkossa. Lisäksi kehitystiimillä on paljon kokemusta matalan tason SQL-koodin käytössä ja tiimi haluaa jatkaa tällä hyväksi havaitulla linjalla. Vaihtoehtoisista tietokantakirjastoista tutkimuksen jälkeen ScalikeJDBC osoittautui sopivimmaksi ratkaisuksi, sillä se tukee paremmin sovelluksen nykyistä suoran SQL-koodin käyttöä ja sopii hyvin Viite-sovelluksen olemassa olevaan arkkitehtuuriin. Toivoisinkin opinnäytetyön herättävän keskustelua matalan tason SQL-koodin hyödyistä ja haitoista verrattuna nykyaikaisempaan oliopohjaiseen korkeamman tason lähestymistapaan.

1.1 Opinnäytetyön rajaus ja tavoitteet

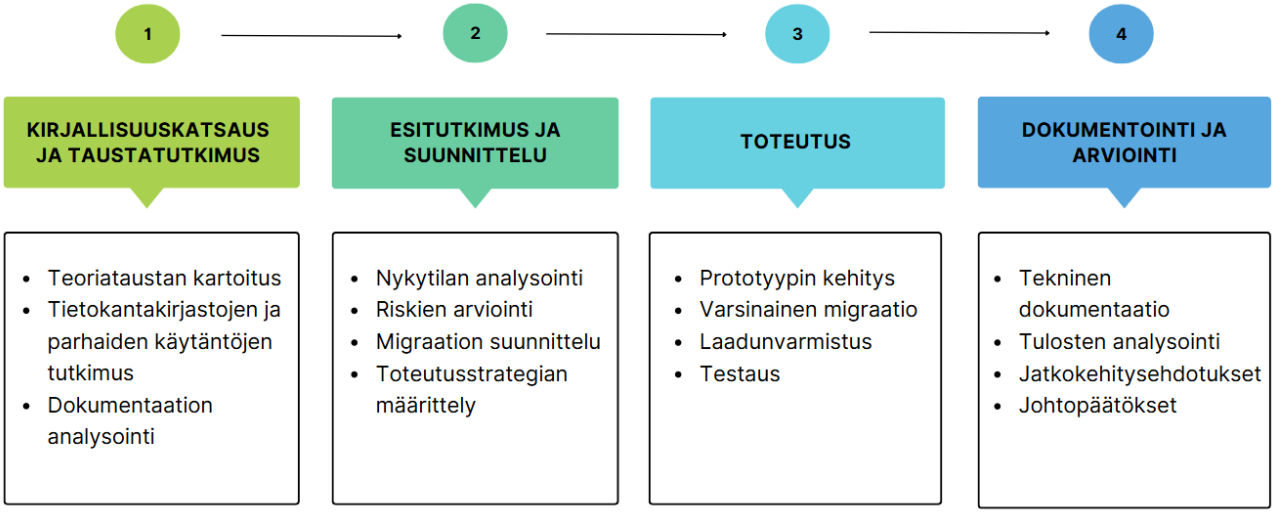
Tämän opinnäytetyön tavoitteena on toteuttaa ja dokumentoida Viite-sovelluksen tietokantakirjaston päivitys Slick-kirjastosta ScalikeJDBC-kirjastoon. Työssä toteutetaan migraatio, joka kattaa kaikki sovelluksen tietokantaoperaatiot, sekä luodaan kattava dokumentaatio migraatioprosessista. Migraation laajuuden vuoksi tarkempi tekninen tarkastelu rajataan huolellisesti valittuihin sovelluksen osiin, jotka edustavat kattavasti erilaisia tietokantaoperaatioita ja monimutkaisia

transaktioketjuja. Muiden sovelluksen osien migraatio kuvataan yleisemmällä tasolla keskittyen keskeisiin teknisiin ratkaisuihin ja havaittuihin haasteisiin. Työ keskittyy erityisesti tietokantaistuntojen hallinnan toteuttamiseen ja monimutkaisten transaktioketjujen toimivuuden varmistamiseen.

Tutkimuksen konkreettisina tuotoksina syntyy päivitetty tietokantakerros, jossa kaikki sovelluksen Slick-toteutukset on korvattu ScalikeJDBC-kirjastolla. Työn tuloksena muodostuu myös tekninen dokumentaatio migraation keskeisistä vaiheista, ratkaisuista ja toteutuksesta tarkempaan tarkasteluun valituissa sovelluksen osissa. Lisäksi tuotoksiin kuuluvat migraatioprosessin aikana kehitetyt yleiset apumetodit ja parhaat käytännöt. Näiden ohella syntyy analyysi ScalikeJDBC-kirjaston soveltuvuudesta matalan tason SQL-toteutuksia sisältävien sovellusten modernisointiin, perustuen migraation aikana kerättyihin kokemuksiin.

1.2 Työn vaiheet

Tutkimus on jaettu neljään päävaiheeseen: kirjallisuuskatsaukseen ja taustatutkimukseen, esitutkimukseen ja suunnitteluun, toteutukseen sekä dokumentointiin ja arviointiin. Vaikka vaiheet on esitetty kronologisessa järjestyksessä, käytännössä ne limittyvät ja niiden välillä liikutaan joustavasti tarpeiden mukaan. Esimerkiksi kirjallisuuskatsaukseen voidaan palata uusien tarpeiden ilmetessä ja dokumentaatiota tuotetaan läpi projektin. Alla oleva kaavio kuvaa työn päävaiheet ja niiden keskeiset sisällöt:



Kuva 1: Opinnäytetyön vaihekaavio

1.3 Keskeiset käsitteet

Tässä kappaleessa määritellään opinnäytetyön keskeiset käsitteet. Käsitteet on valittu tukemaan erityisesti tietokantakirjaston migraation teknistä ymmärtämistä, ja ne kattavat niin ohjelmointikieliin, tietokantoihin kuin ohjelmistokehitykseen liittyvää terminologiaa.

Käsite	Määritelmä
Funktionaalinen ohjelmointi	Ohjelmointitapa, joka perustuu puhtaisiin funktioihin, jotka tuottavat aina saman tuloksen samoilla syötteillä ilman sivuvaikutuksia. Tämä parantaa ohjelman testattavuutta ja ylläpidettävyyttä. (Chiusano, Bjarnason & Pilquist 2023, kappale 1.1.)
DAO (engl. Data Access Obejct)	Suunnittelumalli, joka erottaa tiedonkäsittelyn sovelluksen bisneslogiikaasta tarjoamalla yhtenäisen rajapinnan tietolähteiden käyttöön. Tämä piilottaa taustalla olevat tietokantateknologiat ja mahdollistaa tietokantatoteutuksen vaihtamisen ilman muutoksia bisneslogiikkaan. (Oracle s.a.a.)
JDBC (engl. Java Database Connectivity)	Standardoitu rajapinta, joka mahdollistaa Java-pohjaisten sovellusten yhteyden tietokantoihin. (Oracle 2017.)
Kirjasto (engl. library)	Tarkoittaa sovelluskehityksessä kokoelmaa valmiiksi kirjoitettuja koodikomponentteja, funktioita tai moduuleja, joita ohjelmoijat voivat käyttää uudelleen omissa sovelluksissaan.
Käännösaika (engl. compile time)	Vaihe, jossa ohjelmakoodi käännetään konekielelle suoritusta varten. Käännösaikana tapahtuva tarkistus mahdollistaa virheiden havaitsemisen ja korjaamisen ennen ohjelman suorittamista. (Roy 2024.)
Matalan tason SQL-koodi	Suora tai lähes suora, SQL-kieltä käyttävä lähestymistapa tietokantaoperaatioiden kirjoittamiseen ilman abstraktiokerrosta.
Merkkijonointerpolaatio (engl. string interpolation)	Tapa syöttää muuttujia merkkijonoihin (Scala s.a.).

Perinnejärjestelmä (engl. legacy system)	Järjestelmä, joka on vanhentunut, mutta edelleen käytössä (Tieteen Termipankki 2024).
Relaatiotietokanta	Tietokantatyyppi, joka tallentaa ja tarjoaa pääsyn toisiinsa liittyviin tietoihin taulukkomuodossa, jossa rivit edustavat yksittäisiä tietueita ja sarakkeet näiden tietueiden ominaisuuksia (IBM 2024a).
Scala	Java-ympäristöön perustuva ohjelmointikieli, joka yhdistää olio-ohjelmoinnin ja funktionaalisen ohjelmoinnin ominaisuuksia. Scalassa käytetään tarkkaa tyypitystä, mutta myös tyyppien automaattista päättelyä voidaan hyödyntää. (EPFL s.a.)
ScalikeJDBC	Scala-pohjainen tietokantakirjasto, joka yksinkertaistaa JDBC-rajapinnan käyttöä ja mahdollistaa tietoturvallisten SQL-kyselyiden suorittamisen useissa eri tietokantajärjestelmissä. (ScalikeJDBC 2024a.)
Slick	Scala-pohjainen tietokantakirjasto, joka tarjoaa vahvasti tyypitetyt rajapinnat mahdollistaen tietokantakyselyt Scalan funktionaalisuutta hyödyntäen (Lightbend 2024).
SQL (engl. Structured Query Language)	Standardoitu kyselykieli relaatiotietokannoille tiedon tallentamiseksi, hakemiseksi ja muokkaamiseksi (IBM 2024b.)
Säie (engl. thread)	Ohjelman suorituksen perusyksikkö, joka mahdollistaa rinnakkaisen toimintojen suorittamisen. Säieturvallisuus tarkoittaa, että ohjelmakoodi toimii oikein, vaikka useat säikeet käyttäisivät sitä samanaikaisesti, mikä on erityisen tärkeää esimerkiksi tietokantaistuntojen hallinnassa. (Oracle 2023 s.a.b.)
Tekninen velka	Ohjelmistokehityksessä tehtyjä kompromisseja, jotka nopeuttavat kehitystä lyhyellä aikavälillä mutta vaikeuttavat järjestelmän ylläpitoa ja jatkokehitystä tulevaisuudessa. Tekninen velka vaikuttaa erityisesti järjestelmän ylläpidettävyyteen ja muokattavuuteen. (Nord, Kructen & Ozkaya 2019, luku 1.2.)

2 Tietokantaoperaatiot, -kirjastot ja käytännöt

Nykyaikaiset sovellukset käsittelevät jatkuvasti kasvavia määriä tietoa, ja tämän tiedon tehokas ja turvallinen hallinta on kriittistä sovellusten toiminnalle. Tietokannat muodostavat useimpien sovellusten ytimen, ja niiden käsittely vaatii erityistä huolellisuutta niin suorituskyvyn, tietoturvan kuin tiedon eheydenkin näkökulmasta. Tietokantakirjastot ovat ohjelmistokehittäjien työkaluja, jotka mahdollistavat sovellusten ja tietokantojen välisen kommunikaation. Ne tarjoavat abstraktiokerroksen, joka yksinkertaistaa tietokantaoperaatioiden toteuttamista ja hallintaa sovelluskoodissa. Tässä luvussa käsitellään tietokantaoperaatioihin liittyviä käytäntöjä ja menetelmiä sekä niihin liittyviä riskejä ja suojautumiskeinoja.

2.1 JDBC-ohjelmointirajapinta

Jotta voidaan ymmärtää tietokantakirjastojen toimintaa ja niiden migraation erityispiirteitä, on tärkeää perehtyä niiden teknisiin perustuksiin ja toimintamekanismeihin. Tietokantakirjastot eivät toimi irrallaan muusta teknologiaympäristöstä, vaan ne rakentuvat tiettyjen alustojen ja rajapintojen päälle. Tärkeimpänä niistä on JDBC-rajapinta, joka mahdollistaa tietokantaoperaatioiden suorittamisen Javaan pohjautuvilla sovelluksilla.

Java-pohjaisilla sovelluksilla tarkoitetaan ohjelmistoja, jotka toimivat Java-virtuaalikoneessa (JVM) ja hyödyntävät Java-ekosysteemin teknologioita ja kirjastoja. (IBM 2024c.) Java-pohjaisten sovellusten tietokantakommunikaation perustana toimii JDBC (Java Database Connectivity), joka on ohjelmointirajapinta Javaan pohjautuvien kielten, kuten Scalan, ja relaatiotietokantojen väliseen kommunikaatioon. Sharan kuvailee JDBC-ohjelmointirajapinnan koostuvan joukosta Java-luokkia ja -rajapintoja, jotka toimivat työkaluina tietokantaoperaatioille. Tämä standardoitu lähestymistapa mahdollistaa sovellusten kehittämisen siten, että ne voivat toimia eri tietokantajärjestelmien kanssa yhtenäisellä tavalla. (Sharan 2018, kappale 5.) TypeSafe, Inc ja ScalikeJDBC täydentävät tätä näkemystä osoittamalla, kuinka sekä Slick- että ScalikeJDBC-tietokantakirjastot hyödyntävät JDBC-rajapintaa perustanaan, mutta tarjoavat sen päälle korkeamman tason abstraktioita tehden tietokantakäsittelystä Scala-ystävällisempää. Viite-sovelluksen kontekstissa JDBC-rajapinta muodostaakin perustan sekä nykyiselle Slick-toteutukselle että suunnitellulle ScalikeJDBC-migraatiolle. (ScalikeJDBC 2024a, TypeSafe, Inc 2014a.) Tämän voisi kiteyttää siten, että JDBC on rajapinta helpottamaan yhteyden muodostamista sovelluksen ja tietokannan välille, kun taas tietokantakirjastot rakentavat sen päälle kehittäjäystävällisemmän rajapinnan.

2.2 Scala tietokantaohjelmoinnissa

Scala tarjoaa useita ominaisuuksia, jotka tekevät siitä tehokkaan työkalun tietokantaohjelmoinnissa. Bugnion nostaa esille kielen kyvyn yhdistää vahva staattinen tyypitys älykkääseen tyyppien päättelyyn, minkä ansiosta kääntäjä tunnistaa muuttujien ja funktioiden tyypit automaattisesti. Tämä tekee koodista tiiviimpää ja selkeämpää perinteisiin vahvasti tyypitettyihin kieliin verrattuna. Tietokantaoperaatioissa tämä tarkoittaa, että virheet havaitaan jo käännösvaiheessa esimerkiksi muunnettaessa kyselyiden tuloksia Scala-luokiksi. Lisäksi hänen mukaansa kielen tuki muuttumattomille tietorakenteille ja muuttujille varmistaa, ettei data muutu odottamattomasti sovelluksen suorituksen aikana, mikä on erityisen tärkeää tietokantaoperaatioissa. (Bugnion 2016, luku 1.3.)

Bhattacharjee, Radford & Tome korostavat, että myös implisiittiset parametrit (engl. implicits) sekä hahmontunnistus (engl. pattern matching) ovat Scala-ohjelmointikielen erityispiirteitä, jotka auttavat tietokantaohjelmoinnissa. Implisiittiset parametrit mahdollistavat esimerkiksi tietokantaistunnon ja kyselytulosten muuntosääntöjen automaattisen välittämisen funktioille ilman erillistä määrittelyä. Hahmontunnistuksen avulla puolestaan voidaan käsitellä tietokantakyselyiden tuloksia tyyppiturvallisesti ja varmistaa, että kaikki mahdolliset tulostilanteet, kuten tyhjät arvot, tulevat käsitellyiksi. (Bhattacharjee, Radford & Tome 2021.) Scalan Java-yhteensopivuus täydentää näitä ominaisuuksia mahdollistamalla JDBC-rajapinnan ja muiden Java-pohjaisten tietokantakirjastojen suoran hyödyntämisen, mikä tekee siitä luontevan valinnan erityisesti olemassa olevien Java-teknologioita hyödyntävien järjestelmien kehittämiseen ja ylläpitoon. (Bugnion 2016, luku 1.3.)

ThreadLocal-muuttujat ovat yksi esimerkki tällaisista Java-teknologioista. Oraclen dokumentaatio määrittelee ThreadLocal-muuttujat erityisiksi muuttujiksi, joista jokaisella säikeellä on oma, itsenäinen kopio. Kun ohjelma käyttää ThreadLocal-muuttujaa, se voi lukea tai kirjoittaa arvoja kyseiseen muuttujaan get- ja set-metodien avulla, ja nämä operaatiot vaikuttavat vain nykyiseen säikeeseen. (Oracle 2024.) Klimenko täydentää tätä määritelmää tuomalla esiin, miten ThreadLocal-muuttujia voidaan käyttää käytännössä tietokantaohjelmoinnissa. Hänen mukaansa ThreadLocal-muuttujat helpottavat tietokantayhteyksien hallintaa, sillä jokainen säie voi pitää yllä omaa yhteyttään ilman monimutkaista hallintakoodia. Klimenko korostaa myös ThreadLocal-muuttujien hyödyllisyyttä transaktioiden käsittelyssä, jossa ne auttavat säilyttämään transaktioiden tiedot kuten yhteydet ja kontekstit säiekohtaisesti. Tämä estää eri säikeiden transaktioita häiritsemästä toisiaan, mikä on tärkeää tietokannan eheyden säilyttämiseksi. (Klimenko A. 2024.)

2.3 ORM ja matalan tason SQL-lähestymistavat

Tietokantakirjastojen hyödyntämisessä kehittäjät joutuvat valitsemaan kahden erilaisen lähestymistavan välillä: ORM-pohjaisen (engl. Object-Relational Mapping) ja matalan tason SQL-

lähestymistavan. ORM-pohjaisessa toteutuksessa tietokanta-abstraktion avulla sovelluskehittäjä voi käsitellä tietokannan tietoja ohjelmointikielen olioina, jolloin tietokantakyselyt muodostuvat automaattisesti taustalla. Matalan tason SQL-lähestymistavassa kehittäjä puolestaan kirjoittaa SQL-kyselyt suoraan tai lähes suoraa SQL-kieltä käyttäen, mikä antaa täyden kontrollin tietokantaoperaatioihin. Tämä valinta lähestymistapojen välillä ei ole yksiselitteinen, sillä molemmilla on omat vahvuutensa ja heikkoutensa.

Abba nostaa ORM-työkalujen eduiksi kehitystyön nopeuttamisen ja koodin määrän vähentämisen, kun tietokantaoperaatiot voidaan abstraktoida korkeammalle tasolle. Tämä mahdollistaa tietokannan käsittelyn suoraan ohjelmointikielen olioiden kautta ilman manuaalista SQL-kyselyiden kirjoittamista. Housley ja Reis kuitenkin haastavat tämän näkemyksen tuomalla esiin ORM:n keskeisen ongelman: vaikka kehitystyö nopeutuu, automaattisesti generoidut tietokantaskeemat voivat muodostua ongelmallisiksi. He korostavat, että olio-ohjelmoinnin luonnolliset rakenteet eivät aina käänny tehokkaasti relaatiotietokannan rakenteiksi, mikä voi johtaa sekaviin ja vaikeasti hallittaviin tietokantaratkaisuihin. Kleppmann täydentää tätä näkemystä osoittamalla, että ORM-ratkaisut eivät pysty täysin piilottamaan olio- ja relaatiomallien välisiä eroja, jolloin kehittäjien täytyy joka tapauksessa ymmärtää molempia malleja.

Tietoturvanäkökulmasta keskustelu on moniulotteinen. Siinä missä Abba painottaa ORM-ratkaisujen automaattista suojaa SQL-injektioita vastaan, Tal tuo esiin vähemmän tunnetun näkökulman: matalan tason SQL-toteutus voi olla jopa turvallisempi massiivisten tietokantapäivitysten (engl. mass assignment) yhteydessä. Tal perustelee näkemystään sillä, että eksplisiittinen kenttien määrittely vähentää tahattomien tietoturvaongelmien riskiä. Tämä näkemys täydentää Housleyn ja Reisin havaintoja siitä, että suora kontrolli tietokantaoperaatioihin voi olla eduksi monimutkaisten tietokantarakenteiden hallinnassa. (Abba 2022; Housley & Reis 2022, kappale 7.2; Tal 2024.) Myös Shoham esittää matalan tason SQL-lähestymistavan eduiksi juuri sen, että se tarjoaa kehittäjille täyden kontrollin tietokantaoperaatioihin. Tämä mahdollistaa kyselyiden optimoinnin ja räätälöinnin tarkasti sovelluksen tarpeisiin, mikä voi johtaa parempaan suorituskykyyn. Suorien SQL-kyselyiden käyttö tekee myös koodista läpinäkyvämpää, sillä kehittäjät näkevät tarkalleen, mitä tietokantaoperaatioita suoritetaan. Haittapuolena on, että kehittäjien täytyy kirjoittaa enemmän koodia ja kiinnittää erityistä huomiota tietoturvaan, kuten SQL-injektioiden estämiseen. (Shoham s.a.) Kleppmann nostaa esiin myös SQL-kielen pitkäikäisyyden ja siirrettävyyden edut. Hänen mukaansa standardoitu SQL-kieli on säilyttänyt asemansa tietokantakielenä vuosikymmeniä ja toimii teknologiariippumattomasti, vaikka kilpailevia teknologioita on tullut ja mennyt. (Kleppmann 2025 kappale 3.1.)

2.4 Slick- ja ScalikeJDBC-tietokantakirjastot

Slick on Scala-pohjainen tietokantakirjasto, joka on laajasti tunnettu sen funktionaalisesta lähestymistavasta ja korkean tason abstraktioista (Bugnion 2016, luku 6.1.) Kirjaston vanhemmat versiot mahdollistivat myös suoran SQL-koodin käytön ja transaktioiden räätälöinnin, mutta uudemmissa versioissa on painotettu enemmän tyypitettyjä funktionaalisia rajapintoja. Tässä opinnäytetyössä keskitytään erityisesti Slick-kirjaston vanhempiin versioihin ja niiden korvaamiseen ScalikeJDBC-kirjastolla.

ScalikeJDBC-kirjasto tarjoaa Slick-kirjastoon verrattuna kevyen ja joustavan tavan työskennellä relaatiotietokantojen kanssa. Se on suunniteltu olemaan käytännönläheinen, mahdollistaen kehittäjien keskittymisen SQL-kyselyiden kirjoittamiseen ja tietokantaoperaatioiden suorittamiseen ilman monimutkaisia abstraktiokerroksia. Kirjaston keskeinen filosofia on "Just Write SQL And Get Things Done", mikä heijastuu sen suoraviivaisessa lähestymistavassa tietokantaoperaatioihin. (ScalikeJDBC 2024a.) Tämä sopii erityisen hyvin Viite-sovelluksen kaltaisiin projekteihin, joissa kaikki tietokantakyselyt ovat kirjoitettu suorina SQL-merkkijonoina. ScalikeJDBC tukee lähes kaikkia relaatiotietokantoja, myös PostgreSQL-tietokantaa, jota Viite-sovellus käyttää. Kirjastolla on väin vähän riippuvuuksia, mikä helpottaa sen integrointia olemassa oleviin projekteihin ja vähentävät mahdollisia yhteensopivuusongelmia.

2.5 Transaktiot

Tietojen eheys on kriittistä Viite-sovelluksessa, sillä tieosoitteiden päivitykset vaativat usein monivaiheisia muutoksia useisiin tietokantatauluihin. Esimerkiksi tieosoitteen muutos voi edellyttää samanaikaisia päivityksiä tien perustietoihin, geometriaan ja liittymäpisteisiin. Ilman transaktioita tällaisten monivaiheisten operaatioiden suorittaminen voisi johtaa tietokannan epäjohdonmukaiseen tilaan virhetilanteissa. Connoly määrittelee transaktion tietojenkäsittelyssä kokonaisuudeksi, joka koostuu joukosta loogisesti yhteenkuuluvia tietokantaoperaatioita. Hänen mukaansa keskeistä on järjestelmän palautumiskyky häiriötilanteissa: tietokanta ylläpitää lokitiedostoja, joiden avulla se voi palauttaa tilanteen transaktiota edeltävään tilaan, jos jokin operaatio epäonnistuu tai järjestelmä kaatuu kesken transaktion suorituksen. (Connoly 2015, 668–670.)

Tämä transaktioiden toiminta perustuu ACID-periaatteeseen, jonka Petrov jakaa neljään keskeiseen ominaisuuteen:

1. Atomisuus (engl. Atomicity): Transaktio suoritetaan kokonaan tai ei ollenkaan. Jos mikä tahansa operaatio epäonnistuu, koko transaktio peruutetaan (engl. rollback) ja tietokanta palautetaan lokitiedostojen avulla tilaan, jossa se oli ennen transaktion aloittamista.

2. Eheys (engl. Consistency): Transaktio siirtää tietokannan yhdestä validista tilasta toiseen säilyttäen kaikki määritellyt säännöt, rajoitteet ja suhteet ehjinä.
3. Eristyvyys (engl. Isolation): Samanaikaiset transaktiot eivät vaikuta toisiinsa. Vaikka transaktioita suoritettaisiin rinnakkain, lopputulos on sama kuin jos ne olisi suoritettu peräkkäin.
4. Pysyvyys (engl. Durability): Onnistuneesti toteutetun (engl. commit) transaktion tulokset tallentuvat pysyvästi ja säilyvät myös järjestelmän häiriötilanteissa. (Petrov 2019, kappale 5)

ACID-periaatteen lisäksi tietokantaoperaatioiden suorituksessa on Kleppmannin mukaan tärkeää huomioida transaktioiden oikean suoritusjärjestyksen ja -tavan varmistaminen. Hän korostaa, että yksinkertaisin tapa välttää rinnakkaisuusongelmat, on suorittaa transaktiot sarjallisesti, eli yksi kerrallaan. Tämä periaate heijastuu myös sisäkkäisten transaktioiden käsittelyyn: jos transaktio on jo käynnissä, uuden transaktion aloittaminen sen sisällä voi johtaa monimutkaisiin ongelmiin muutosten järjestyksen ja peruuttamisen kanssa. Vaikka modernit tietokantajärjestelmät pystyvät käsittelemään useita rinnakkaisia transaktioita eri käyttäjiltä, yksittäisen transaktion sisällä operaatioiden sarjallinen suoritus ja sisäkkäisten transaktioiden estäminen ovat edelleen tärkeitä mekanismeja tietojen eheyden varmistamiseksi. (Kleppmann 2016, kappale 7.3.)

ScalikeJDBC tarjoaa tietokantatransaktioiden hallintaan tavan, jossa käytetään implisiittistä tietokantasessiota, minkä avulla useita operaatioita voidaan suorittaa saman transaktion sisällä. Tämä mahdollistaa monimutkaisten transaktioketjujen helpon toteuttamisen ja hallinnan.

```
val count = DB localTx { implicit session =>
  // --- transcation scope start ---
  sql"update emp set name = ${name1} where id = ${id1}".update.apply()
  sql"update emp set name = ${name2} where id = ${id2}".update.apply()
  // --- transaction scope end ---
}
```

Tässä koodiesimerkissä suoritetaan kaksi SQL-komentoa ScalikeJDBC-kirjaston tarjoaman localTx transaktiolohkon sisällä. Molemmat suoritetaan loppuun ainoastaan, jos virhettä ei ilmene kummankaan komennon aikana. (ScalikeJDBC 2024b.) Tämä transaktioiden käsittelytapa on merkittävästi yksinkertaisempi kuin Viite-sovelluksen nykyisessä Slick-toteutuksessa. ScalikeJDBC-kirjaston tarjoama implisiittinen sessio ja localTx-lohko mahdollistavat transaktioiden yksinkertaisen ja turvallisen hallinnan.

2.6 SQL-kyselyjen tietoturva

Viite-sovelluksen tietokantakyselyt on kyettävä luomaan turvallisesti ScalikeJDBC-avulla. Caselli ym. korostavat SQL-injektioiden olevan yleisin tietokantakyselyihin liittyvä haavoittuvuus. Siinä

hyökkääjä pystyy vaikuttamaan sovelluksen tietokantakyselyihin syöttämällä SQL-koodia syöteparametreihin. Yksi tehokkaimmista tavoista estää SQL-injektioita on parametrisoitujen kyselyiden (parameterized queries) käyttö, joissa SQL-lauseita ei muodosteta suoraan merkkijonoina, vaan kyselyissä käytetään paikanvaraajia (engl. placeholders), joihin varsinaiset arvot sijoitetaan turvallisesti sarjoitettuina parametreina. Tämä estää SQL-koodin injektoimisen osaksi kyselyä, sillä parametrien sisältöä ei koskaan tulkita suoritettavana SQL-koodina. (Caselli, Galluccio & Lombari 2020, kappale 5.1)

ScalikeJDBC vastaa tähän turvallisuushaasteeseen hyödyntämällä merkkijonojen Scalan interpolaatiosyntaksia. Esimerkiksi käyttäjän syöttämää tienumeroa käsiteltäessä kysely voidaan muodostaa turvallisesti seuraavasti:

```
sql"SELECT * FROM roads WHERE road_number = ${roadNumber}"
```

Tässä ScalikeJDBC muuntaa roadNumber-parametrin automaattisesti PreparedStatement-muotoon, mikä estää haitallisen SQL-koodin suorittamisen, vaikka muuttuja sisältäisi SQL-injektioyrityksen. Vertailun vuoksi perinteinen, vaarallinen tapa muodostaa kysely merkkijonojen yhdistämisellä olisi:

```
val query = "SELECT * FROM roads WHERE road_number = " + roadNumber
```

Tällainen toteutus altistaa sovelluksen SQL-injektioille, sillä jos roadNumber sisältäisi esimerkiksi arvon "1 OR 1=1", muodostuisi kysely, joka palauttaisi kaikki tietueet taulusta. ScalikeJDBC-kirjaston interpolaatio estää tämän käsittelemällä parametrit turvallisesti.

Lisäksi kirjasto tarjoaa myös SQL-syntaksin tarkistuksen käännösaikana, mikä auttaa havaitsemaan virheet jo kehitysvaiheessa. ScalikeJDBC laajentaa turvallisen kyselyn rakentamisen mahdollisuuksia tarjoamalla SQL-interpolaattorin (sql"...") käytettäväksi myös monimutkaisemmissa tapauksissa, kuten IN-operaattoreissa ja dynaamisissa WHERE-ehdoissa, säilyttäen samalla vahvan suojan SQL-injektioita vastaan. Esimerkiksi kyselyssä, joka sisältää IN-operaattorin ja dynaamisia ehtoja, ScalikeJDBC pystyy edelleen turvallisesti käsittelemään parametrien asettamisen:

```
val roadNumbers = List(101, 102, 103)
val query = sql"SELECT * FROM roads WHERE road_number IN (${roadNumbers}) AND
status = ${status}"
```

Tässä roadNumbers -lista ja status -parametri sijoitetaan turvallisesti kyselyyn ilman riskiä SQL-injektiosta. Kirjasto käsittelee myös tyyppiturvallisuutta huolehtimalla parametrien tyypeistä ennen niiden lähettämistä tietokantaan. Tämä estää vääräntyyppisistä syötteistä johtuvat virheet ja tietoturvaongelmat, jotka voivat syntyä esimerkiksi merkkijonoina välitettävien numeroarvojen

yhteydessä. (ScalikeJDBC 2024c.) Matalan tason SQL-lähestymistavan ja nykyaikaisten turvallisuuskäytäntöjen yhdistäminen on tasapainoilua tarkan kontrollin ja abstraktioiden välillä. ScalikeJDBC näyttää tarjoavan tähän toimivan kompromissin, joka mahdollistaa suoran SQL-koodin kirjoittamisen ilman tietoturvan tai transaktioiden hallinnan heikentymistä.

3 Migraation lähtötilanne ja suunnittelu

3.1 Viite-sovelluksen esittely

Viite-sovellus on Väyläviraston omistama tieosoitejärjestelmän ylläpitosovellus, jonka avulla hallitaan ajoratojen, teiden, solmujen ja liittymien tietoja karttapohjaisessa käyttöliittymässä tielinkkigeometrian perusteella. Tieosoitejärjestelmä määrittää tien ominaisuustietojen sijainnin tietokannassa ja maastossa tunnistetietojen, kuten tien numeron, tieosan numeron, ajoradan numeron ja etäisyyden avulla. Sovellus säilyttää myös tieosoitehistorian, mikä tukee tietojen jäljitettävyyttä ja hyödyntämistä myös muissa järjestelmissä.

Sovelluksen kehitystä hallitaan Git-versionhallinnalla, ja GitHubia käytetään muutosten hallintaan ja tarkistamiseen ennen käyttöönottoa. Sovellus toimii AWS-pilvipalvelussa, jossa sillä on omat kehitys- (dev), testaus- (QA) ja tuotantoympäristönsä (prod). Kun uusi versio julkaistaan päähaaraan GitHubissa, se käynnistää CI/CD-putken, joka suorittaa automaattiset testit ja päivittää kehitysympäristön kontin AWS-pilvipalvelussa, joka lisäksi huolehtii myös sovelluksen toiminnan seurannasta ja resurssien mukauttamisesta tarpeen mukaan. Tekninen arkkitehtuuri koostuu JavaScript-pohjaisesta frontendistä, Scala-pohjaisesta backendistä sekä PostgreSQL-tietokannasta PostGIS-laajennuksella.

3.2 Kehitysprojektin lähtökohdat ja tavoitteet

Opinnäytetyön kannalta olennaisinta on syventyä sovelluksen tietokantaa käsitteleviin toimintoihin. Viite-sovellus käyttää tietokantaoperaatioiden suorittamiseen vanhentunutta Slick-kirjastoa, joka mahdollistaa suorien SQL-kyselyjen hyödyntämisen ja transaktiometodien räätälöinnin. Keskeinen ongelma on, että sovelluksen käyttämät transaktiometodit ovat poistuneet käytöstä Slick-kirjaston uudemmissa versioissa.

Juuri transaktioketjujen suorittaminen uudemmilla Slick-versioilla osoittautui mahdottomaksi nykyisellä arkkitehtuurilla ja kehitystiimin aiempi yritys päivittää Slick uudempaan versioon jäi lopulta toteuttamatta, koska uudemmat Slick-versiot eivät pysty käsittelemään Viite-sovelluksen vaativia transaktioketjuja ilman merkittäviä muutoksia koodiin ja ohjelmointiparadigmaan. Lisäksi uudemmat versiot painottavat funktionaalista ohjelmointia ja korkeampia abstraktiotasoja, mikä ei sovi yhteen sovelluksen nykyisen rakenteen kanssa

Keskeisenä projektin onnistumisen arvioinnissa voidaankin pitää päivityksen toimivuutta ja suoriutumista monimutkaisista tietokantaoperaatioista sekä pitkistä transaktioketjuista. Arvioinnissa voidaan tarkastella myös, kuinka sujuvasti päivitetyt osat integroituvat olemassa olevaan tietokantayhteys- sekä palvelukerrokseen ja miten tehokkaasti testitapaukset saadaan mukautettua uuteen

kirjastoon. Työn tulisi tarjota kattava ja objektiivinen kuva päivityksen tuomista hyödyistä ja mahdollisista haasteista, tai mahdollinen päätelmä siitä, että ScalikeJDBC ei olekaan paras vaihtoehto Viite-sovellukselle.

3.3 Tutkimuksen kohderyhmä

Migraatioprojektin tulokset palvelevat ensisijaisesti Viite-sovelluksen jatkokehitystä. Sovelluksen kehittäjät saavat työstä konkreettisen mallin ja dokumentaation tietokantakirjaston käyttämiseen sekä siihen liittyvien haasteiden ratkaisemiseen. Päivitys helpottaa erityisesti monimutkaisten transaktioketjujen toteuttamista ja ylläpitoa. Lisäksi Väyläviraston tekninen henkilöstö hyötyy työstä päätöksenteon tukena. Tulokset ja dokumentaatio helpottavat sovelluksen jatkokehityksen suunnittelua.

Laajemmassa kuvassa työ tuo merkittävän lisän ScalikeJDBC-kirjaston käytännön dokumentaatioon. Vaikka kirjasto on teknisesti toimiva, sen dokumentaatio ja käytännön esimerkit ovat rajallisia. Tämä työ tarjoaa konkreettisen esimerkin kirjaston käytöstä monimutkaisessa tuotantosovelluksessa, erityisesti tilanteessa, jossa vaaditaan matalan tason SQL-toteutuksen säilyttämistä. Tästä voivat hyötyä muut kehittäjät, jotka harkitsevat ScalikeJDBC-kirjaston käyttöönottoa tai etsivät ratkaisuja vastaaviin migraatiotarpeisiin.

Tietojärjestelmätieteen tutkimuksen näkökulmasta tämä työ tarjoaa merkittävää arvoa tapaustutkimuksena tietokantakirjaston migraation käytännön toteutuksesta. Tutkimus dokumentoi yksityiskohtaisesti matalan tason SQL-toteutuksen etuja ja haasteita modernissa sovelluskehityksessä. Työ esittelee konkreettisen esimerkin siitä, miten perinnejärjestelmän tietokantakerrosta voidaan modernisoida hallitusti. Työ dokumentoi sekä matalan tason SQL-toteutuksen etuja että haasteita ja tarjoaa käytännön havaintoja erilaisten abstraktiokerrosten käytöstä. Samalla se toimii tapaustutkimuksena teknisen velan hallinnasta ja sen vähentämisestä vanhassa järjestelmässä. Nämä kokemukset täydentävät olemassa olevaa tutkimustietoa järjestelmien modernisointiprosesseista.

3.4 Projektin rajoittava tekijät ja resurssit

Teknisinä rajoituksina keskeisimpänä on vaatimus säilyttää matalan tason SQL-toteutukset ja olemassa olevat monimutkaiset transaktioketjut. Uuden tietokantakirjaston on kyettävä tukemaan Viite-sovelluksen arkkitehtuuria ja sen käyttämää PostgreSQL-tietokantaa PostGIS-laajennuksineen, ilman merkittäviä muutoksia sovelluksen nykyiseen rakenteeseen.

Vaikka migraatioprosessilla ei ole tiukkoja aikataulullisia rajoituksia, sen etenemiseen vaikuttaa Viite-projektin muu kehitystyö. Koska tietokantakirjaston migraatio ei ole projektin ykkösprioriteetti, voi sekä migraation toteutus että opinnäytetyön valmistuminen viivästyä, mikäli akuutimpia

kehitystarpeita ilmenee. Erityisesti vuoden lopussa toteutettava tieosoitekierros asettaa rajoituksia migraation julkaisulle - päivitystä ei voida ottaa tuotantokäyttöön ennen kuin kaikki tieosoitekierrokseen liittyvät ongelmat on selvitetty. Tästä syystä työ toteutetaan vaiheittain, keskittyen ensin rajattuun osaan sovelluksen tietokantaoperaatioista. Tämä lähestymistapa mahdollistaa joustavan etenemisen muun kehitystyön rinnalla.

Projekti edellyttää ymmärrystä Scala-ohjelmointikielestä, SQL:stä ja PostgreSQL:stä. Alle vuoden kokemuksella Scala-ohjelmointikielestä, en voi sanoa, että osaamiseni olisi erityisen syvällistä, mutta perusasiat hoituvat jo luontevasti. Minulla on myös jonkin verran kokemusta Viite-sovelluksen tietokantaoperaatioista ja ymmärrän pääpiirteittäin toiminnallisuudet. Git-versionhallinnan, GitHub-julkaisualustan ja ScalaTest-testaustyökalun hallinta ovat välttämättömiä projektin toteuttamiseksi. Resurssien osalta projekti on kustannustehokas, sillä se ei vaadi erillisiä lisäinvestointeja. Pääasialliset resurssitarpeet rajoittuvat työn toteuttamiseen käytettävään aikaan.

3.5 Nykyisen Slick-toteutuksen analysointi

Tässä luvussa tarkastellaan Viite-sovelluksen tietokantakerroksen nykytilaa, mikä on välttämätöntä migraation suunnittelun ja toteutuksen kannalta. Viite-sovelluksessa tietokantaoperaatiot on toteutettu käyttämällä nimellisesti Slick-kirjaston versiota 3.0.3, joka julkaistiin vuonna 2015. Todellisuudessa Viitteen tietokantaoperaatoihin käyttämät tärkeimmät ominaisuudet ovat kuitenkin peruja Slick-kirjaston versiosta 2.1.0. Nämä vanhentuneet tekniikat mahdollistavat suoran SQL-koodin kirjoittamisen Scala-koodin sisällä sekä transaktioiden räätälöidyn hallinnan, mikä on ollut keskeistä sovelluksen monimutkaisten tietokantaoperaatioiden toteuttamisessa.

3.5.1 Tietokantayhteyden alustaminen ja hallinta korkeammalla tasolla

Viite-sovelluksessa tietokantayhteys alustetaan PostGISDatabase-objektissa. Tällä hetkellä yhteydenhallintaan käytetään BoneCP-yhteyspoolia, joka on kuitenkin vanhentunut eikä enää suositella käytettäväksi uusissa projekteissa (Wadge 2014). Kun tietokantayhteys on alustettu, Viite-sovellus hyödyntää Slickin tarjoamaa dynamicSession- ja withDynTransaction (vanhentuneet 2.1.0-version jälkeen) ominaisuuksia tietokantayhteyksien ja transaktioiden hallinnassa. Esimerkiksi withDynTransaction-metodi on toteutettu seuraavasti:

```
def withDynTransaction[T](f: => T): T = {
  if (transactionOpen.get())
    throw new IllegalThreadStateException("Attempted to open nested transac-
tion")
  else {
    try {
      transactionOpen.set(true)
      Database.forDataSource(PostGISDatabase.ds).withDynTransaction {
        setSessionLanguage()
        f
```

```
      }
    } finally {
      transactionOpen.set(false)
    }
  }
}
```

Tätä metodia käytetään kaikissa operaatioissa, joissa muutetaan tietokantaa. Vastaavasti withDynSession-metodia käytetään lukuoperaatioissa. Edellä mainitut metodit toimivat keskeisessä osassa sovelluksen tietokantatoimintojen hallinnassa. Tietokantaoperaatiot kääritään (engl. wrap) näihin metodeihin palvelukerroksessa (engl. service layer) tai REST-rajapinnassa, jotta transaktioita voidaan hallita ja varmistaa tietojen eheys.

3.5.2 Tekniikat, joita käytetään DAO-kerroksen toimintoihin

Nyt kun ymmärtää pääpiirteittäin, miten tietokantaoperaatiot muodostuvat sovelluksessa, voi analysoida tarkemmin, mitä DAO-kerroksessa tapahtuu. DAO-kerros vastaa tietokantayhteyksien hallinnasta ja suorittaa varsinaiset SQL-kyselyt tietokantaan. Se toimii välikerroksena sovelluslogiikan ja tietokannan välillä, eristäen tietokantatoiminnot muusta sovelluksesta. Migraatio ScalikeJDBC-kirjastoon tuleekin vaikuttamaan eniten juuri DAO-luokkien ja objektien metodeihin.

BaseDAO-luokka toimii pohjana kaikille DAO-luokille ja -objekteille sisältäen yleiskäytännöllisiä metodeja tietokantakyselyiden suorittamiseen. Keskeinen metodi on runUpdateToDb, joka suorittaa SQL-päivityskyselyjä tietokantaan:

```
def runUpdateToDb(updateQuery: String): Int = {
  sqlu"""#$updateQuery""".buildColl.toList.head
}
```

Metodi sqlu on Slick-kirjaston tarjoama tapa suorittaa päivityskyselyjä (UPDATE, INSERT, DELETE). Viite-sovellus hyödyntää tästä vanhentunutta Slick 2.1.0-version buildColl-metodia. Kysely välitetään merkkijonona ja interpoloidaan suoraan SQL-lauseeseen (#$updateQuery), mikä mahdollistaa kyselyjen suorittamisen mutta altistaa sovelluksen SQL-injektioille.

```
def expire(id: Long, username: String): Int = {
  val query = s"""Update ROAD_NAME Set valid_to = current_timestamp, created_by
= '$username' where id = $id"""
  runUpdateToDb(query)
}
```

Tässä RoadNameDAO-luokan expire-metodissa SQL-päivityskysely muodostetaan merkkijonona, jossa käyttäjänimi ja id interpoloidaan suoraan kyselyyn. Metodi palauttaa päivityksestä vaikuttuneiden rivien lukumäärän. Hakukyselyissä käytetään vastaavaa lähestymistapaa:

```
private def queryList(query: String) = {
  Q.queryNA[RoadName](query).iterator.toSeq
}
```

```
def getLatestRoadName(roadNumber: Long): Option[RoadName] = {
  val query =
    s"""$roadsNameQueryBase Where road_number = $roadNumber and valid_to is null
and end_date is null"""
  queryList(query).headOption
}
```

RoadNameDAO-luokan esimerkkimetodissa getLatestRoadName kysely rakennetaan merkkijonona yhdistämällä kyselyyn ehtoja. Muuttujien arvot, kuten roadNumber, interpolaoidaan suoraan kyselyyn. Lopuksi rakennettu merkkijono query välitetään queryList-metodille, joka suorittaa kyselyn tietokantaan ja palauttaa tulokset RoadName-olioina. Metodissa käytetään vanhentunutta Slick 2.1.0-version StaticQuery-objektia (Q), jonka queryNA-metodi suorittaa parametrittoman kyselyn ja palauttaa tulokset RoadName-tyyppisinä olioina. Kyselyn tulokset iteroidaan Scala-sekvenssiksi. (TypeSafe, Inc. 2014c.)

Tietokantatulosten muuntaminen Scala-olioiksi tapahtuu implisiittisten GetResult-määrittelyjen avulla:

```
implicit val getRoadNameRow: GetResult[RoadName] = new GetResult[RoadName] {
  def apply(r: PositionedResult) = {
    val roadNameId = r.nextLong()
    val roadNumber = r.nextLong()
    val roadName = r.nextString()
    val startDate = r.nextDateOption().map(d => new DateTime(d.getTime))
    val endDate = r.nextDateOption().map(d => new DateTime(d.getTime))
    val validFrom = r.nextDateOption().map(d => new DateTime(d.getTime))
    val validTo = r.nextDateOption().map(d => new DateTime(d.getTime))
    val createdBy = r.nextString()

    RoadName(roadNameId, roadNumber, roadName, startDate, endDate, validFrom,
validTo, createdBy)
  }
}
```

Tässä GetResult[RoadName] määrittelee, miten yksittäinen tulosrivi (PositionedResult) muunnetaan RoadName-olioksi. Metodissa apply tulosrivin sarakkeiden arvot luetaan oikeassa järjestyksessä ja luodaan uusi RoadName-olio näiden arvojen perusteella (TypeSafe, Inc 2014c.) Kun queryList-metodi suoritetaan, Slick käyttää automaattisesti tätä implisiittistä GetResult-määrittelyä tulosrivien muuntamiseen RoadName-olioiksi. Näin saatu Seq[RoadName] palautetaan takaisin, sitä kutsuvalle metodille.

3.5.3 Testitapaukset

Viite-sovelluksessa myös testit on toteutettu käyttämällä Slick-kirjaston tarjoamia ominaisuuksia tietokantaoperaatioiden suorittamiseen testien aikana. Suurin osa testitapauksista on luotu seuraavan esimerkkitestin mukaista kaavaa hyödyntäen:

```
test("Test esimerkkiService.esimerkkiMetodi("""Esimerkkitesti"""){
  runWithRollback { // Syötetään testidata transaktiolohkon sisällä:
    runUpdateToDb("""Insert into Esimerkki (Esimerkki_id) values ('1')""")
    // Testitapauksen implementointi
     val result = esimerkkiService.esmerkkiMetodi(1)
     result should be expectedValue
    }
  }
}
```

Testissä hyödynnetään PostGISDatabase luokassa luotua runWithRollback-metodia, jonka sisään testi kääritään. Testi suoritetaan runWithRollback-metodin sisällä, joka aloittaa transaktion ja peruuttaa sen testin lopuksi hyödyntäen dynamicSession objektin metodia rollback. Tämä mahdollistaa sen, että testit voivat lisätä, muokata ja poistaa tietoja tietokannasta ilman, että ne vaikuttavat pysyvästi tietokannan tilaan, eivätkä ne siten vaikuta muihin testeihin tai tuotantodataan. (Type-Safe, Inc 2014a.)

3.5.4 Nykytila-analyysin johtopäätökset

Analyysin perusteella Viite-sovelluksen nykyinen Slick-pohjainen toteutus käyttää useita vanhentuneita ja riskialttiitakin tekniikoita, jotka heikentävät sovelluksen turvallisuutta ja ylläpidettävyyttä. Keskeinen ongelmia on merkkijonojen interpoloinnin käyttö SQL-kyselyissä ilman parametrien turvallista sitomista, mikä altistaa sovelluksen SQL-injektiolle, eikä tarjoa virheidenhallintaa kattavasti.

Löydetyt tekniikat, joille on implementoitava vastine ScalikeJDBC-kirjaston avulla:

- Tietokantayhteyden alustaminen
- Merkkijonojen interpolointi SQL-kyselyissä
- Tulosten mapittaminen Scala-luokiksi
- Istuntojen ja transaktioiden hallinta
- Testien toteutus transaktion sisällä rollBack-toimintoa hyödyntäen

Kun nämä keskeiset toiminnot on saatu ratkaistua, voidaan laajamittainen migraatio toteuttaa tehokkaasti ilman suurempia muutoksia sovelluksen arkkitehtuuriin. Tämä lähestymistapa mahdollistaa vanhojen kyselyiden ja palvelukerroksen toimivuuden säilyttämisen samalla, kun taustalla oleva toteutus modernisoidaan turvallisemmaksi ja ylläpidettävämmäksi

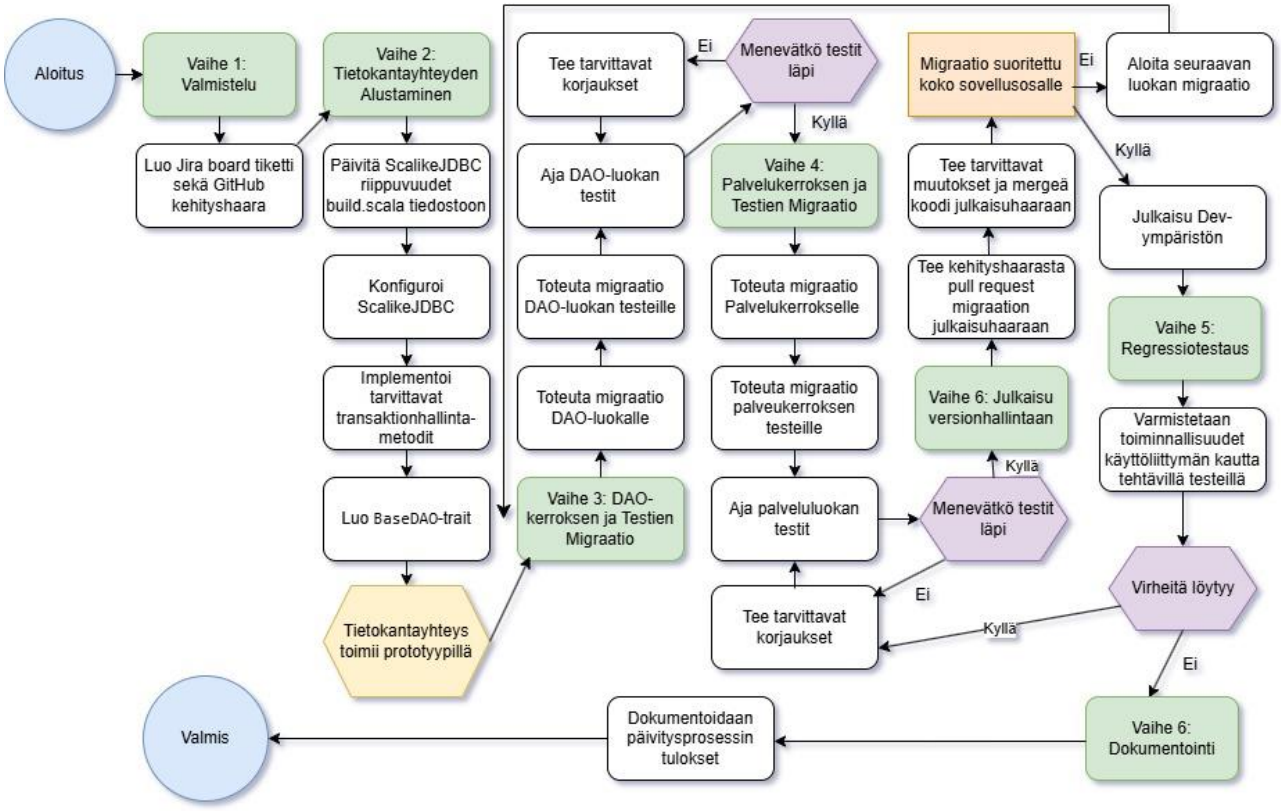
3.6 Migraatiosuunnitelma

Migraatioprosessi Slick-kirjastosta ScalikeJDBC-kirjastoon laaja-alainen hanke. DAO-kerroksessa on noin 30 tiedostoa, jotka koostuvat DAO-luokista ja -objekteista sisältäen yhteensä noin 8000 koodiriviä. Nämä tiedostot sisältävät kaikki sovelluksen tietokantakyselyt ja -operaatiot. Lisäksi testitiedostoja on suuri määrä, ja ne kaikki suorittavat tietokantaoperaatioita. Tämä tarkoittaa, että

myös testit on päivitettävä vastaamaan uutta tietokantakerrosta, jotta ne toimivat oikein migraation jälkeen. Kokonaisuudessaan migraatio tulee vaikuttamaan tietokantakerroksen kautta lähes kaikkiin sovelluksen osa-alueisiin ja se tulee vaatimaan laajamittaista koodin uudelleenkirjoittamista sekä perusteellista testausta toiminnallisuuden varmistamiseksi. Migraatio Slick-kirjastosta ScalikeJDBC-kirjastoon tulee toteuttaa vaiheittain ja suunnitelmallisesti, jotta voidaan minimoida riskit ja varmistaa sovelluksen toimivuus koko prosessin ajan.

3.6.1 Vaiheet ja prosessikuvaus

Migraatio Slick-kirjastosta ScalikeJDBC-kirjastoon toteutetaan iteratiivisesti etenevänä prosessina, jossa jokainen vaihe testataan huolellisesti ennen seuraavaan siirtymistä. Prosessi alkaa valmisteluilla, etenee vaiheittain tekniseen toteutukseen ja päättyy dokumentointiin.



Kuva 1. Migraation vaiheiden prosessikuvaus

Prosessikaavio osoittaa migraation iteratiivisen lähestymistavan. Jokaisen vaiheen jälkeen suoritetaan testit, joiden tulokset määrittävät jatkon. Jos testit läpäistään, voidaan siirtyä seuraavaan vaiheeseen. Jos ongelmia ilmenee, tehdään tarvittavat korjaukset ja testit ajetaan uudelleen. Tämä iteratiivinen testaus-korjaus-sykli toistuu jokaisessa vaiheessa varmistaen, että migraatio etenee vakaasti.

Prosessi käynnistyy valmisteluvaiheella, jossa luodaan erillinen kehityshaara ja Jira-tiketti migraation seurantaa varten. Tämän jälkeen edetään tietokantayhteyden alustamisvaiheeseen, jossa lisätään ScalikeJDBC-kirjasto riippuvuuksineen, konfiguroidaan tietokantayhteys ja luodaan keskeiset komponentit kuten transaktiometodit ja BaseDAO-trait-rajapinta. Kun tämä pohjatyö on testattu toimivaksi, voidaan siirtyä DAO-kerroksen migraatioon.

DAO-kerroksen migraation jälkeen prosessi jatkuu palvelukerroksen päivityksellä. Tämäkin vaihe sisältää oman testaussilmukkansa. DAO-luokan testit varmistava yksikkötason toimivuuden, ja palveluluokan testit varmistavat komponenttien yhteistoiminnan. Kun sekä DAO- että palvelukerros on saatu toimimaan testeissä, muutokset viedään kehityshaaraan pull request -toimintona.

Julkaisuvaiheessa muutokset viedään kehitysympäristöön ja suoritetaan kattava regressiotestaus käyttöliittymän kautta. Jos ongelmia löytyy, ne korjataan ja testataan uudelleen. Lopuksi koko migraatioprosessi ja sen tuotokset dokumentoidaan.

Tämä iteratiivinen lähestymistapa mahdollistaa virheiden varhaisen havaitsemisen ja korjaamisen, mikä minimoi riskejä ja varmistaa sovelluksen toimivuuden koko migraatioprosessin ajan. Vaiheittainen eteneminen keskeisissä komponenteissa (tietokantayhteys → DAO-kerros → palvelukerros → kokonaisuus) mahdollistaa hallitun ja luotettavan migraation monimutkaisessakin järjestelmässä.

3.7 Opinnäytetyössä esiteltävän kohdealueen rajaus

Migraatioprojektin laajuuden vuoksi olen päättänyt keskittyä opinnäytetyössäni kolmeen ensimmäiseen ja tärkeimpään vaiheeseen. Ensimmäiseksi kuvaan, miten uuden tietokantakirjaston perustoiminnot ja yhteydenmuodostuksen varmistetaan. Toisessa vaiheessa esittelen yksinkertaisemman tietokokonaisuuden käsittelyn (teiden nimien hallinta) migraation uuteen järjestelmään. Kolmanneksi käyn läpi monimutkaisemman kokonaisuuden migraation, joka sisältää teiden solmujen ja liittymien käsittelyn useilla tietokantatauluilla. Valitsin nämä vaiheet, koska ne sisältävät projektin tärkeimmät tekniset haasteet ja ratkaisut. Näistä kolmesta vaiheesta saatujen kokemusten pohjalta on mahdollista toteuttaa loput migraatiosta samoja periaatteita noudattaen. Työni lopussa käyn läpi koko projektin aikana kohdatut haasteet ja syntyneet kehitysideat.

4 Migraation toteutus

Tässä luvussa kuvataan Viite-sovelluksen tietokantakirjaston migraation käytännön toteutus vaiheittain. Migraatio toteutettiin suunnitelmallisesti edellisessä kappaleessa määriteltyjen vaiheiden mukaisesti ja ensimmäinen konkreettinen vaihe oli ScalikeJDBC-kirjaston alustaminen ja konfigurointi Viite-sovellukseen. Kunkin vaiheen yksityiskohtaisemmat tekniset toteutukset ja koodiesimerkit on dokumentoitu tämän opinnäytetyön liitteissä.

4.1 ScalikeJDBC-kirjaston lisääminen ja konfigurointi

Ensin oli selvitettävä oikea versionumero-kirjastosta, joka sopisi Viite-sovelluksen nykyisiin riippuvuuksiin. Versionumero 3.4.2 osoittautui uusimmaksi versioksi, joka tukee Scala 2.11 versiosta. Kyseisellä ScalikeJDBC-versiolla on tuki Scalan 2.13 -versioon asti. Tämä tarkoittaa sitä, että tulevaisuudessa, kun Scala päivitetään versioon, 2.12 tai 2.13, voidaan ScalikeJDBC päivittää uusimpaan versioonsa, eikä sen myötä tietokantakirjasto ole enää esteenä Scalan versioiden päivittämiselle. (MvnRepository s.a.)

Aloitin lisäämällä ScalikeJDBC-riippuvuudet projektin build.scala-tiedostoon. ScalikeJDBC-version lisäksi tarvitsi lisätä vain scalikejdbc-config ja scalikejdbc-joda-time, jotka käyttävät samaa ScalikeJDBC-versionumeroa. Päätin hyödyntää ScalikeJDBC-kirjaston oletus yhteyspoolia, joten sitä ei tarvinnut erikseen määritellä riippuvuudeksi. Seuraavaksi loin PostGISDatabaseScalikeJDBC-objektin, joka vastaa tietokantayhteyden hallinnasta:

```
import scalikejdbc._

object PostGISDatabaseScalikeJDBC {
  // Load the PostgreSQL driver
  Class.forName("org.postgresql.Driver")

  // Initialize the connection pool with default settings
  ConnectionPool.singleton(
    url = ViiteProperties.scalikeJdbcUrl,
    user = ViiteProperties.scalikeJdbcUser,
    password = ViiteProperties.scalikeJdbcPassword
  )

  // Logging enabled for all queries for development purposes
  GlobalSettings.loggingSQLAndTime = LoggingSQLAndTimeSettings(
    enabled = true,
    logLevel = 'info
  )
}
```

Toteutuksessa ladataan ensin PostgreSQL-ajuri, jonka jälkeen alustetaan yhteyspooli, joka käyttää sovelluksen konfiguraatiotiedostossa määriteltyjä yhteysasetuksia. Kehitystyön helpottamiseksi ScalikeJDBC-kirjaston tarjoama SQL-kyselyiden lokitus on määritelty päälle infotasolla. Muhin

tietokanta-asetuksiin en tehnyt tässä vaiheessa muutoksia, sillä myös aiempi toteutus pohjautui Slick-kirjaston oletusasetuksiin, jotka vastaavat ScalikeJDBC-kirjaston oletus asetuksia.

4.2 Tietokantayhteyden muodostaminen

Tietokantayhteyden perustoiminnallisuuden testaamista varten toteutin BaseDAO-luokkaan yksinkertaisen apumetodin runWithReadOnlyImplicitSession, joka mahdollisti tietokantaoperaatioiden suorittamisen lukuoikeuksilla. Valitsin ensimmäiseksi testikohteeksi PingApi-rajapinnan, sillä se tarjosi luontevimman tavan testata tietokantayhteyden toimivuutta ilman monimutkaista liiketoimintalogiikkaa. Toteutuksen ydin oli yksinkertainen aikaleiman hakeminen tietokannasta, mikä toimi erinomaisena testinä tietokantayhteyden toimivuudelle.

Huomasin, että PingApi-luokan aiempi testi tarkisti vain HTTP-vastauksen statuskoodin, mutta ei varmistanut tietokantakyselyn onnistumista. Täydensin testausta lisäämällä uuden testin, joka varmistaa myös tietokantakyselyn toimivuuden tarkistamalla palautetun aikaleiman muodon. Lopulta esti tuotti toivotut tulokset. Tämä ensimmäinen toteutus toimi hyvänä perustana laajemman migraation suunnittelulle. Se vahvisti, että ScalikeJDBC-kirjasto toimii odotetusti Viite-sovelluksen ympäristössä ja että tietokantaistuntojen hallinta onnistuu apumetodien avulla. PingAPI, -DAO ja -testitiedostot löytyvät liitteestä 7.

4.3 RoadNameService-luokan ja RoadNameDAO-objektin migraatio

Kun tietokantayhteys ja perustoiminnallisuus oli varmistettu, siirryin teiden nimiä käsittelevän RoadNameService -palvelukerroksen ja siihen osallistuvan RoadNameDAO-objektin pariin. Olin valinnut tämän osa-alueen, koska siihen liittyvät tietokantakyselyt ja palvelukerroksen transaktiot olivat suhteellisen yksinkertaisia. Tämän osa-alueen migraatio toimisi eräänlaisena prototyyppinä lopullisen migraation tekemiselle. Merkittävin muutos koski tietokantatulosten käsittelyä. Slick-toteutuksen GetResult-implisiittien sijaan toteutin tulosten mapituksen ScalikeJDBC-kirjaston apply-metodilla:

```
def apply(rs: WrappedResultSet): RoadName = RoadName(
    id = rs.long("id"),
    roadNumber = rs.long("road_number"),
    roadName = rs.string("road_name"),
    startDate = rs.jodaDateTimeOpt("start_date"),
    // muut kentät
  )
}
```

RoadName-objektissa määritelty apply-metodi ottaa parametrinaan WrappedResultSet-olion, joka sisältää tietokantakyselyn tuloksen, ja luo sen tietojen perusteella uuden RoadName-instanssin. SQLSyntaxSupport-trait-rajapinnan käyttö mahdollistaa taulun nimen määrittelyn ja helpottaa SQL-

kyselyiden rakentamista sekä aliaksien käyttöä. SQL-kyselyiden rakentamisessa hyödynsin aliaksia ja SQL-interpolointia:

```
private val rn = RoadName.syntax("rn")

def getCurrentRoadNamesByRoadNumber(roadNumber: Long)
                                   (implicit session: DBSession): Seq[RoadName]
= {
  val query = sql"""
    SELECT ${rn.id}, ${rn.roadNumber}, ${rn.roadName}, ${rn.startDate},
        ${rn.endDate},${rn.validFrom}, ${rn.validTo}, ${rn.createdBy}
    FROM ${RoadName.as(rn)}
    WHERE ${rn.roadNumber} = $roadNumber
      AND ${rn.endDate} IS NULL
      AND ${rn.validTo} IS NULL
        """
  queryList(query)
}
```

Muuttuja "rn" toimii aliaksena RoadName-objektille, mikä helpottaa kyselyjen kirjoittamista ja tekee niistä luettavampia erityisesti liitoksissa. Tulokset mapitetaan Scala-olioiksi queryList-metodilla, jonka toteutin hyödyntäen ScalikeJDBC-kirjaston list().apply()-kutsua. Tämä apumetodi yksinkertaistaa DAO-luokkien koodia ja vähentää toistoa. Yksityiskohtainen tekninen toteutus on esitelty liitteessä 6.

4.4 Istunnonhallinta

Teiden nimiin aluksi käyttämissäni metodeissa on määriteltynä implisiittinen sessio, avaan sitä hieman seuraavaksi. Kun palvelukerroksessa suoritetaan tietokantaoperaatioita, avataan tietokantaistunto metodilla kuten runWithReadOnlyImplicitSession (liite 7). Tämä metodi luo implisiittisen DBSession-istunnon, joka määritellään lohkossa implicit session. Kaikki tämän lohkon sisällä olevat DAO-metodit, joilla on parametrina implicit session: DBSession, käyttävät automaattisesti tätä istuntoa. Näin istunto välittyy palvelukerroksesta DAO-luokkiin ja edelleen tietokantakyselyihin. Huomasin kuitenkin nopeasti, että implisiittisen istunnon määrittely jokaisen DAO-metodin parametriksi oli työlästä ja teki koodista vähemmän siistiä. Kokeilin ensin scalikejdbc-kirjaston tarjoamaa AutoSession-ominaisuutta, joka tarjoaa automaattisesti saatavilla olevan istunnon. Tämä yksinkertaisti koodia, koska DAO-metodit pystyivät käyttämään istuntoa ilman erillistä implisiittisen istunnon määrittelyä:

```
// Automaattinen istunto kyselyille
implicit val session: DBSession = AutoSession

// Apumetodi kyselyile, jotka palauttavat RoadName-objektin
private def queryList(query: SQL[Nothing, NoExtractor]): Seq[RoadName] = {
  runSelectQuery(query.map(RoadName.apply))
}
```

```
// Metodi, jolla kysely suoritetaan
def getCurrentRoadNamesByRoadNumber(roadNumber: Long): Seq[RoadName] = {
  val query = sql"""
    <SELECT CLAUSE HERE>
      """
  queryList(query)
}
```

AutoSession ei kuitenkaan toiminut luotettavasti testien kanssa - erityisesti rollBack-ominaisuus ei toiminut odotetulla tavalla. Tämän vuoksi päädyin kehittämään oman SessionProvider-komponentin, joka tarjoaa säieturvallisen tavan hallita tietokantaistuntoja ThreadLocal-muuttujan avulla. Tämä haaste ratkaisuineen on kuvattu tarkemmin kappaleessa 6.1.3.

SessionProvider-toteutuksen keskeisin ominaisuus on withSession-metodi, joka hallitsee istuntoja määritellyn koodilohkon sisällä. SessionProviderin session-import mahdollistaa saman automaattisen istunnonhallinnan kuin AutoSession, mutta luotettavammin ja Viite-sovellukseen tarpeisiin räätälöidyllä tavalla. Toteutus estää sisäkkäiset transaktiot, mutta sallii lukuoperaatiot transaktion sisällä. SessionProvider-komponentti on esitelty kokonaisuudessaan liitteessä 2. Tämän pohjalta toteutin PostGISDatabaseScalikeJDBC-objektiin kolme metodia istuntojen hallintaan. Esimerkiksi testeihin käytettävästä automaattisesti tietokantamuutokset peruvasta luomastani runWithRollback -metodista:

```
// Testikäyttöön, muutokset peruutetaan
def runWithRollback[Result](testOperation: => Result): Result = {
  DB.localTx { session: DBSession =>      // 1. Aloitetaan uusi transaktio
    withSession(session) {                // 2. Asetetaan istunto säiekohtaisesti
      val result = testOperation          // 3. Suoritetaan testioperaatiot
      session.connection.rollback()       // 4. Peruutetaan muutokset
      result                             // 5. Palautetaan tulos
    }                                    // 6. Palautetaan aiempi istunto
  }                                      // 7. Suljetaan transaktio
}
```

Kun testi suoritetaan runWithRollback-metodin sisällä:

1. ScalikeJDC-apumetodi DB.localTx aloittaa uuden transaktion
2. Luomani withSession tallentaa mahdollisen aiemman istunnon ja asettaa uuden käyttöön
3. Testissä määritellyt operaatiot suoritetaan
4. Kutsutaan rollback() joka peruuttaa kaikki tehdyt muutokset
5. Palautetaan testin tulos
6. withSession palauttaa aiemman istunnon käyttöön
7. ScalikeJDBC-kirjaston localTx-metodi sulkee transaktion automaattisesti lohkon sulkeutuessa

Näin tietokantaan tehdyt muutokset eivät jää voimaan testin suorituksen jälkeen, mutta testin aikana voidaan silti varmistaa operaatioiden toimivuus. Tämä toteutus mahdollistaa testien suorittamisen Viite-sovelluksen olemassa olevalla arkkitehtuurilla.

Edellä mainitun metodin lisäksi määrittelin PostGISDatabaseScalikeJDBC-objektiin kaksi erilaista metodia tietokantaistuntojen ja -operaatioiden hallintaan. Metodi runWithTransaction avulla voidaan suorittaa tietokantaoperaatiot yhden transaktion sisällä. Transaktio varmistaa, että kaikki muutokset tallentuvat vain, jos jokainen operaatio onnistuu. Jos mikä tahansa operaatio epäonnistuu, kaikki muutokset perutaan automaattisesti. Metodi runWithReadOnlySession puolestaan mahdollistaa tietokannan lukuoperaatiot ilman kirjoitusoikeuksia. Jos metodin sisällä yritetään muokata tietokantaa, nousee SQLException.

Kuten liitteessä 1 tarkemmin esitetystä PostGISScalikeJDBC -konfiguraatio-objektista, käy ilmi, jokainen metodi hyödyntää SessionProviderin withSession-metodia tietokantaistuntojen säieturvalliseen hallintaan, yhdistäen sen ScalikeJDBC-kirja tarjoamiin DB-objektin metodeihin (localTx, readOnly). ScalikeJDBC huolehtii matalan tason JDBC-yhteyksien hallinnasta ja transaktioiden toiminnallisuudesta, kun taas SessionProvider varmistaa istuntojen hallinnan sovellustasolla. Nämä metodit muodostavat pohjan kaikille sovelluksen tietokantaoperaatioille - palvelukerros kutsuu näitä metodeja, jotka puolestaan välittävät istunnon DAO-kerroksen SQL-kyselyille SessionProviderin kautta. Tämä arkkitehtuuri mahdollistaa joustavan ja turvallisen tietokantaoperaatioiden suorittamisen Viite-sovelluksen erilaisissa käyttötapauksissa.

4.5 BaseDAO-trait ja SQL-kyselyjen apumetodit

Kun aloitin laajemman migraation Viite-sovelluksen SQL-kyselyille, kävi nopeasti ilmi tarve yhtenäiselle tavalle suorittaa erilaisia tietokantaoperaatioita. Vaikka ScalikeJDBC tarjoaa kattavat perustoiminnot tietokantakyselyiden suorittamiseen, päätin toteuttaa räätälöidyt apumetodit erilaisille kyselytyypeille, jotta kyselyt voisi suorittaa Slick-toteutusta mukaillen intuitiivisesti. Erityisesti tarvittiin metodeja erilaisten tulosten käsittelyihin: joskus haluttiin kaikki tulokset listana, toisinaan vain ensimmäinen tulos tai tarkistus yksittäisen rivin olemassaolosta.

Slick-toteutuksessa päivityskyselyt suoritettiin yksinkertaisen runUpdateToDb-metodin kautta, joka oli altis SQL-injektioille, sillä kyselymerkkijonot yhdistettiin suoraan ilman parametrisointia. Hakukyselyissä käytettiin suoraan Slick-kirjaston tarjoamia metodeja kuten first ja list. Päätin toteuttaa kattavamman valikoiman apumetodeja BaseDAO-trait-rajapintaan, jotta niitä olisi intuitiivista käyttää ja kaikki tarvittavat metodit löytyisivät keskitetysti ohjeineen yhdestä paikasta.

Trait hyödyntää SessionProvider-objektin tarjoamaa implisiittistä istuntoa, mikä yksinkertaistaa DAO-luokkien toteutusta merkittävästi poistamalla tarpeen määritellä tietokantaistunto erikseen

jokaisessa metodikutsussa. BaseDAO-trait-rajapintaan toteutin metodeja eri tarpeisiin: tulosten hakemiseen listana, ensimmäisen tuloksen hakemiseen tai täsmälleen yhden tuloksen hakemiseen virheidenkäsittelyllä. Tyyppimuunnosten helpottamiseksi toteutin metodeja yksittäisten arvojen hakemiseen määritellyllä tyypillä, kuten alla olevassa esimerkkikoodissa näkyvä runSelectSingleWithType-metodi osoittaa. Metodissa yhdistetään ScalikeJDBC-kirjaston single-metodi ja luomani mukautetut tyyppimuuntimet implisiittisellä mapperilla tietotyypille Long:

```
// Metodi BaseDAO-traitissa:
def runSelectSingleWithType[T](query: SQL[Nothing, NoExtractor])
(implicit mapper: WrappedResultSet => T): T = {
  try {
    query.map(mapper).single().apply().getOrElse(
      throw ViiteException("No value returned")
    )
  } catch {
    case e: IllegalStateException => throw ViiteException(e.getMessage)
  }
}

implicit val longMapper: WrappedResultSet => Long = _.long(1)
```

Esimerkki metodien käytön selkeytymisestä DAO-kerroksessa:

```
// Aiempi Slick-toteutus
def nextRoadwayNumber: Long = {
  Queries.nextRoadwayNumber.as[Long].first
}

// ScalikeJDBC-toteutus räätälöidyllä apumetodilla
def nextRoadwayNumber: Long = {
  runSelectSingleWithType[Long](Queries.nextRoadwayNumber)
}

// ScalikeJDBC-toteutus ilman apumeteodia
def nextRoadwayNumber: Long = {
  Queries.nextRoadwayNumber
    .map(rs => rs.long(1))  // Tyypin muunnos täytyy määritellä manuaalisesti
    .single()               // Haetaan yksittäinen tulos
    .apply()               // Suoritetaan kysely
    .getOrElse(throw new RuntimeException("Failed to get next roadway id"))
}
```

Tämä räätälöity lähestymistapa yksinkertaisti merkittävästi tietokantakyselyiden toteuttamista Viite-sovelluksessa. BaseDAO-trait-rajapinnan apumetodit yhdessä SessionProvider-objektin implisiittisen istunnonhallinnan kanssa tekevät kyselyiden toteuttamisesta sekä suoraviivaisempaa että turvallisempaa. Metodien tarkemmat toteutukset ja dokumentaatio löytyvät liitteestä 3.

4.6 Teiden solmujen ja liittymien käsittelyn migraatio

Kun apumetodit oli toteutettu BaseDAO-trait-rajapintaan, varsinainen DAO-kerroksen migraatio eteni suoraviivaisesti kyselyiden muuntamisella ScalikeJDBC-kirjaston syntaksia hyödyntäväksi. RoadNameDAO-prototyypissä kokeilin aiemmin ScalikeJDBC-kirjaston tarjoamia abstraktiokerroksia, kuten taulualiaksia (eng. table alias) ja SQL-SyntaxSupport-ominaisuutta (liite 6). Tämä lähestymistapa osoittautui kuitenkin virhealttiiksi ja teki migraatiosta tarpeettoman monimutkaista ja aikaa vievää, sillä jo toimivaksi todetut suorat SQL-kyselyt olisivat vaatineet paljon muokkaamista. Sarakkeiden käsittelyn haasteet ovat kuvattu tarkemmin kappaleessa 6.1.1. Haasteiden johdosta päätin pitää SQL-kyselyt mahdollisimman lähellä alkuperäistä muotoaan ja keskittyä vain välttämättömiin muutoksiin. Muutostyö keskittyi lopulta yksinkertaisesti merkkijonointerpolaation korvaamiseen ScalikeJDBC-kirjaston sql-interpolaattorilla ja parametrisoiduilla kyselyillä. Esimerkiksi JunctionDAO-luokassa kysely voitiin muuttaa ScalikeJDBC-muotoon yksinkertaisesti:

```
// Slick versio
s"SELECT * FROM JUNCTION WHERE NODE_NUMBER in (${nodeNumbers.mkString(", ")})

// ScalikeJDBC-versio
sql"SELECT * FROM junction WHERE node_number IN ($nodeNumbers)"
```

Yksinkertaistettu lähestymistapa osoittautui toimivaksi ratkaisuksi. Kyselyt selkeytyivät, kun ScalikeJDBC-kirjaston SQL-interpolaattori hoitaa automaattisesti parametrien käsittelyn ja SQL-injektioilta suojautumisen. Samalla kyselyiden rakenne säilyi tuttuna ja helposti ymmärrettävänä. Monimutkaisemmissa kyselyissä, joissa tarvittiin dynaamista SQL-kyselyiden rakentamista, hyödynsin ScalikeJDBC-kirjaston sqls-interpolaattoria, tarkempi kuvaus ja sen käytöstä on kuvailtuna liitteessä 4.

Aiemmin käytetyt Slickin Q.queryNA ja sqlu-metodit korvautuivat BaseDAO-trait-rajapinnan apumetodeilla, jotka tarjosivat yhtenäisen ja turvallisen tavan käsitellä tietokantaoperaatioita. Tämä lähestymistapa nopeutti migraation toteutusta merkittävästi. Kun SQL-kyselyiden rakenne pysyi lähes muuttumattomana, pystyin keskittymään olennaiseen: Slickin metodien korvaamiseen BaseDAO-trait-rajapintaan vastaavilla metodeilla. Esimerkiksi:

```
// Slick versio
Q.queryNA[JunctionWithLinearLocation](query).iterator.toSeq

// ScalikeJDBC-versio
runSelectQuery(query.map(JunctionWithLinearLocation.apply))
```

Osana DAO-kerroksen migraatiota minun tuli toteuttaa myös ratkaisu useiden tietueiden samanaikaiseen päivittämiseen tietokannassa (engl. batch update). Slick-toteutuksessa oli aiemmin

käytetty JDBC-rajapinnan prepareStatement-metodia ja executeBatch-kutsua, jotka mahdollistivat useiden rivien yhtäaikaisen lisäämisen tai päivittämisen. Toteutin tämän toiminnallisuuden ScalikeJDBC-kirjastolla lisäämällä BaseDAO trait-rajapintaan runBatchUpdateToDb-metodin, joka tarjoaa selkeän tavan eräajojen suorittamiseen hyödyntäen ScalikeJDBC-kirjaston batch-metodia. Uusi toteutus osoittautui aiempaa tiiviimmäksi ja selkeämmäksi, sillä metodi mahdollistaa parametrien välittämisen suoraan kokoelmina, kun taas Slick-versiossa jokainen parametri täytyi asettaa erikseen. Myös null-arvojen käsittely yksinkertaistui, kun kirjasto hoitaa automaattisesti tietotyyppien muunnokset. Tietokannan päivitykset tapahtuivat nyt siistimmällä syntaksilla, jossa kysely ja parametrit määriteltiin selkeästi erillisinä komponentteina. Tietokantapäivityksiä koskeva tarkempi tekninen dokumentaatio löytyy liitteestä 4.

DAO-luokkien testitiedostoihin tarvittavat muutokset olivat pieniä. Usein riitti, että Slick-toteutuksen RunWIthRollback korvattiin ScalikeJDBC-toteutuksellani. Ajaessani DAO-luokkien testejä, ilmeni useita virheitä kirjoittamassani koodissa, mutta ne olivat suurilta osin syntaksivirheitä, jotka oli helppo paikantaa ja korjata.

Kun kaikki DAO-kerroksen metodit oli muutettu ScalikeJDBC-muotoon siten että testit menivät läpi, seuraavana vuorossa oli NodesAndJunctionsService -palvelukerroksen migraatio. Palvelukerros vastaa tietokantaistuntojen aloittamisesta, mutta muuten tietokantakyselyt hoidetaan DAO-kerroksessa. Tehtävänä oli siis korvata Slick -toteutuksen istuntolohkojen alustukset uusilla ScalikeJDBC-versioilla. Ensin kävin läpi jokaisen palvelukerroksessa käytetyn transaktion ja istuntolohkon. Ne operaatiot, joissa vain luettiin dataa, alustin luomallani runWithReadOnlySession-metodilla. Kun taas istunnot, joissa dataa muutettiin, alustin runWithTransaction-metodilla. Metodit kuvattu liitteessä 1.

Aiemmassa Slick-toteutuksessa oli lisäksi metodi withDynTransactionNewOrExisting, jolla mahdollistettiin sisäkkäiset transaktiot poikkeustapauksissa. Tutkin kyseisen metodin tarpeellisuutta ja tulin siihen tulokseen, ettei sitä ScalikeJDBC -toteutuksessa tarvita. Metodi osoittautui tarpeettomaksi, sillä kohdat, joissa se oli käytössä, olivat joko lukuoperaatioita tai jo olemassa olevan transaktion sisällä. Koska olin määritellyt SessionProvider-komponenttiin sisäkkäiset lukuistunnot sallituiksi, riitti withDynTransactionNewOrExisting-istuntojen alustamiseen pelkkä runWithReadOnlySession-metodi. Vastaavasti transaktioiden alustaminen oli turhaa niissä kohdissa, joissa koodi jo suoritettiin toisen transaktion sisällä, joten ylimääräiset transaktioalustukset voitiin poistaa kokonaan. Aiemmassa Slick toteutuksessa oli käytetty transaktioita kaikissa istunnoissa ja korvaamalla lukuistunnot sille tarkoitetulla metodilla ja karsimalla ylimääräiset transaktioalustukset pois, tuli toteutuksesta turvallisempi, virheen käsittely parani ja koodista tuli helpommin luettavaa.

Myös liittymien ja solmujen palvelukerroksen testitiedoston migraatio käyttämään ScalikeJDBC-kirjastoa sujui nopeasti, sillä tarvittavat muutokset olivat pieniä. Testitapauksia ajettaessa kuitenkin havaittiin enemmän virheitä kuin DAO-kerroksen testeissä. Myös palvelukerroksen osalta virheet liittyivät pääosin SQL-kyselyiden syntaksiin tai tyyppimunnoksiin, mutta paljastivat myös merkittävämpiä ongelmia tietokantaoperaatioissa. Haasteista tarkemmin kappaleessa 6.1.

4.7 Julkaisu Dev ympäristöön ja regressiotestaus

Kun kaikki migraation vaatimat muutokset oli saatu valmiiksi, käsittäen yhteensä lähes 100 tiedostoa ja tuhansia koodirivejä, mukaan lukien opinnäytetyön rajauksessa mainittujen keskeisten komponenttien lisäksi myös kaikki muut DAO- ja palvelukerroksen luokat testeineen, oli aika varmistaa kokonaisuuden toimivuus. Päätin suorittaa kattavan regressiotestauksen ensin lokaalissa kehitysympäristössäni ennen muutosten viemistä yhteiseen dev-kehitysympäristöön. Tämä lähestymistapa mahdollisti tehokkaamman virheenetsinnän ja korjauksen, sillä virheiden paikallistaminen lokaalissa ympäristössä on tyypillisesti nopeampaa kuin jaetulla kehityspalvelimella ja arvelin virheitä löytyvän runsaasti.

Regressiotestauksen toteutin käymällä läpi monisivuisen testauskaavakkeen, joka kattoi sovelluksen keskeiset toiminnallisuudet selainrajapinnan kautta. Testauksessa ilmeni lopulta kuitenkin vain muutamia virheitä. Ainoa transaktioihin liittynyt ongelma oli tapauksessa, jossa dataa muuttava transaktio oli virheellisesti alustettu lukuoperaatioille tarkoitetulla runWithReadOnlySession-metodilla. Muut löydetyt virheet olivat pääosin syntaksivirheitä, kuten merkkijonojen käyttämistä sqls-interpolaattorin sijaan tai väärin kirjoitettuja taulusarakkeiden nimiä. Löytyneiden virheiden perusteella huomasin, että ComplimentaryLink-luokalta puuttui yksikkötestit lähes kokonaan, minkä vuoksi virheitä oli päässyt läpi testauksessa. Ratkaisuna loin Jiraan tiketin, jossa ehdotin testikattavuuden parantamista tälle luokalle.

Regressiotestauksen jälkeen oli aika viedä muutokset dev-ympäristöön. Muutosten yhdistäminen päähaaraan käynnisti CI/CD-putken, joka sisälsi automaattisten testien ajon ja muutosten julkaisun. Ensimmäinen julkaisuyritys päättyi virheeseen, koska olin migraation yhteydessä uudelleennimennyt tiedoston, joka oli määritelty AWS-palvelun ECS-tehtävämäärittelyssä (task definition). Tilanteen ratkaisemiseksi päätimme palauttaa tiedoston alkuperäisen nimen ja siirtää uudelleennimeämisen myöhempään ajankohtaan.

Tämän korjauksen jälkeen sovelluksen julkaisu dev-ympäristöön onnistui ongelmitta. Kattavat testitapaukset sekä huolellinen regressiotestaus varmistivat, että migraatio ei aiheuttanut odottamattomia ongelmia kehitysympäristössä. Vaikka tietokantakirjaston migraation tekninen toteutus oli saatu valmiiksi, on todennäköistä, että pieniä virheitä löytyy vielä jatkossa laajemman käytön

myötä. Tämä on tavallista näin laajan migraation yhteydessä. Oleellista kuitenkin on, että migraation toimivuus oli nyt todennettu käytännössä, ja havaitut virheet olivat luonteeltaan korjattavissa ilman merkittäviä rakenteellisia muutoksia. Vaikka jatkuva seuranta ja ylläpito ovat edelleen tarpeen, oli migraation kriittisin ja teknisesti haastavin osuus nyt takanapäin.

5 Migraation tuotokset

Tässä luvussa tarkastelen migraatioprojektin tuloksia suhteessa asetettuihin tavoitteisiin ja esittelen keskeiset tekniset saavutukset. Migraation päätavoitteena oli korvata vanhentunut Slick-tietokantakirjasto ScalikeJDBC-kirjastolla siten, että sovelluksen matalan tason SQL-toteutukset ja monimutkaiset transaktioketjut säilyvät toimivina. Samalla pyrin parantamaan koodin ylläpidettävyyttä ja luomaan pohjan sovelluksen jatkokehitykselle.

5.1 Tekninen toteutus

Migraation tekninen toteutus koostui useista toisiinsa liittyvistä komponenteista, jotka yhdessä mahdollistivat Slick-kirjaston korvaamisen ScalikeJDBC-kirjastolla:

1. **PostGISDatabaseScalikeJDBC-objekti**: Määrittelin tietokantayhteyden asetukset ja yhteydenhallinnan menetelmät sovelluksen tarpeisiin mukautettuna. Objekti korvasi PostGISDatabase-objektin ja konfiguraation lisäksi se tarjoaa metodit erilaisten tietokantaoperaatioiden suorittamiseen.
2. **SessionProvider-objekti**: Toteutin tietokantaistuntojen hallintaan säieturvallisen ratkaisun, joka käyttää ThreadLocal-muuttujaa istuntojen säilyttämiseen. Tämä mahdollisti implisiittisten istuntojen käytön DAO-metodeissa ja paransi koodin luettavuutta. SessionProvider varmistaa myös, ettei sisäkkäisiä transaktioita synny vahingossa.
3. **BaseDAO-trait**: Räätälöin yhtenäisen rajapinnan tietokantaoperaatioille, joka sisältää monipuoliset apumetodit erilaisten kyselyiden suorittamiseen. Trait yhdistää ScalikeJDBC-kirjaston perusmetodit ja SessionProviderin tarjoaman istunnonhallinnan helposti käytettäväksi kokonaisuudeksi.
4. **DAO-luokkien migraatio**: Päivitin kaikki DAO-luokat käyttämään ScalikeJDBC-kirjaston SQL-interpolaattoria sekä BaseDAO-traitin apumetodeja. Tietokantakyselyjen tulosten tyyppimuunnokset toteutin apply-metodeilla, jotka muuntavat tulosrivit sovelluslogiikan käyttämiksi olioiksi. Lisäksi toteutin selkeän tavan suorittaa massapäivityksiä batch update -operaatioina.
5. **Palvelukerroksen päivitys**: Muutin kaikki palvelukerroksissa olevat tietokantaistuntojen alustukset käyttämään uusia PostGISDatabaseScalikeJDBC-objektin tarjoamia istuntometodeja.
6. **Testitapauksien päivitys**: Päivitin kaikki testit käyttämään luomaani ScalikeJDBC-versiota runWithRollback-metodista, sekä tein tarvittavat muutokset testeissä tehtäviin SQL-kyselyihin SQL-interpolaattorin sekä BaseDAO-trait-rajapinnan apumetodien avulla.

5.2 Keskeiset parannukset

Slick-kirjaston korvaaminen ScalikeJDBC-kirjastolla toi sovellukseen useita merkittäviä parannuksia. Tietoturva parani huomattavasti, kun alkuperäisen toteutuksen haavoittuva merkkijonojen yhdistäminen SQL-kyselyissä korvattiin ScalikeJDBC-kirjaston SQL-interpolaattorilla, joka huolehtii automaattisesti parametrien turvallisesta sitomisesta ja suojaa sovellusta SQL-injektioilta. Tietokantakoodista tuli selkeämpää siirryttäessä käyttämään sarakenimiä järjestysnumeroiden sijaan, mikä paransi koodin luettavuutta ja vähensi virhealttiutta. Samalla kaikki SQL-kyselyt muotoiltiin yhtenäiseen, selkeään muotoon migraation yhteydessä.

BaseDAO-trait-rajapinta mahdollisti keskitetyn ja yhtenäisen rajapinnan tietokantaoperaatioille. Metodit kuten runSelectSingle, runSelectSingleOption ja runSelectQuery tarjoavat johdonmukaisen tavan käsitellä tietokantakyselyiden tuloksia. Erityisesti virheidenkäsittely parani, kun esimerkiksi runSelectSingle-metodi heittää informatiivisen poikkeuksen kyselyn palauttaessa useita rivejä tai ei yhtään riviä, mikä mahdollistaa virheiden varhaisen havaitsemisen ja käsittelyn.

Uusi SessionProvider jalosti transaktioidenhallintaa yhdistämällä sisäkkäisten transaktioiden eston säiekohtaiseen istuntojenhallintaan. Tämä lähestymistapa mahdollistaa lukuoperaatioiden suorittamisen transaktioiden sisällä ilman, että lukuistunnosta yritetään avata uutta transaktiota. Toteutus käsittelee nyt eksplisiittisesti eri istuntotyypit (transaktio, lukuistunto, autocommit) ja tarkistaa niiden yhteensopivuuden ennen istunnon vaihtamista, estäen esimerkiksi tilanteen, jossa lukuistunto yritetään muuttaa transaktioksi kesken operaation.

ScalikeJDBC-kirjaston tuoreempi versio loi modernin teknologiapohjan poistamalla riippuvuuden vanhentuneista Slick-tekniikoista ja mahdollistaen sovelluksen päivittämisen tulevaisuudessa uudempiin Scala-versioihin. Samalla syntyi hyvä pohja jatkokehitykselle, sillä sovelluksen arkkitehtuuri on nyt joustavampi ja helpommin laajennettavissa. BaseDAO-trait-rajapinnan ansiosta uusien tietokantaoperaatioiden toteuttaminen on suoraviivaista ja yhdenmukaista, ja selkeästi jäsennelty koodi sekä päivitetty tietokantakerros mahdollistavat uusien ominaisuuksien lisäämisen nopeammin ja luotettavammin.

Migraation aikana tuotetut liitteet ja koodiin lisätty dokumentaatio tarjoavat arvokasta tietoa sekä Viite-sovelluksen jatkokehittäjille että muille vastaavanlaisia migraatioita tekeville kehittäjille. Dokumentaatio sisältää yksityiskohtaiset kuvaukset toteutetuista ratkaisuista, kohdatuista haasteista ja niiden ratkaisuista sekä parhaat käytännöt ScalikeJDBC-kirjaston hyödyntämiseen matalan tason SQL-kyselyjen toteuttamisessa.

5.3 Koodin laadun parannukset

Migraation myötä koodin laatu parani merkittävästi useilla osa-alueilla. ScalikeJDBC-kirjaston modernimmat ominaisuudet ja räätälöidyt apumetodit mahdollistivat selkeämmän, tiiviimmän ja turvallisemman koodin kirjoittamisen. Tässä kappaleessa on esiteltynä koodin laadun parannuksia sekä konkreettisia esimerkkejä toteutuksien välillä.

Istuntojen hallinta yksinkertaistui merkittävästi ScalikeJDBC-kirjaston tarjoamien metodien ja SessionProvider-objektin yhdistelmän ansiosta. Toimintalogiikka selkeytyi ja istuntometodien toteutukset (liite 1) ovat nyt tiiviimpiä, helpommin luettavia ja selkeämmin rajattuja käyttötarkoituksiinsa. Esimerkiksi transaktion aloittamiseen tarkoitetun metodin koodi yksinkertaistui merkittävästi:

```
// Aiempi Slick versio

def withDynTransaction[T](f: => T): T = {
  if (transactionOpen.get())
    throw new IllegalThreadStateException("Attempted to open nested transac-
tion")
  else {
    try {
      transactionOpen.set(true)
      Database.forDataSource(PostGISDatabase.ds).withDynTransaction {
        setSessionLanguage()
        f
      }
    } finally {
      transactionOpen.set(false)
    }
  }
}

// ScalikeJDBC versio

def runWithTransaction[Result](databaseOperation: => Result): Result = {
  DB.localTx { session =>
    withSession(session) {
      databaseOperation
    }
  }
}
```

Tyyppimuunnosten selkeys parani, sillä ScalikeJDBC-toteutus käyttää sarakkeiden nimiä indeksien sijaan. Tämä tekee koodista itsedokumentoivaa ja helpommin ylläpidettävää, koska kehittäjän ei tarvitse muistaa sarakkeiden järjestystä kyselyissä. Sarakkeiden nimien käyttö vähentää myös virheiden riskiä tietokantaskeeman muuttuessa. Lisäksi erikoistyyppien, kuten päivämäärien, käsittely yksinkertaistui, kun ScalikeJDBC tarjoaa valmiit metodit, kuten esimerkiksi jodaDateTimeOpt, yleisimpien tietotyyppien lukemiseen ilman monimutkaisia muunnoksia. Tämä vähentää boilerplate-koodia ja tekee tietokantatulosten käsittelystä intuitiivisempaa:

```
// Aiempi Slick-toteutus (pohjautuu sarakkeiden järjestykseen)

def apply(r: PositionedResult) = {
  val roadNameId = r.nextLong()
  val roadNumber = r.nextLong()
  val roadName = r.nextString()
  val startDate = r.nextDateOption().map(d => new DateTime(d.getTime))
        // Muut kentät järjestyksessä…
}

// Uusi ScalikeJDBC-toteutus (pohjautuu sarakkeiden nimiin)

def apply(rs: WrappedResultSet): RoadName = RoadName(
  id          = rs.long("id"),
  roadNumber  = rs.long("road_number"),
  roadName    = rs.string("road_name"),
  startDate   = rs.jodaDateTimeOpt("start_date"),
        // Muut kentät nimettynä…
)
```

Tietokantaoperaatioiden yhtenäistäminen BaseDAO-trait-rajapinnnan avulla toi merkittäviä parannuksia koodin yhdenmukaisuuteen. Aiemmassa Slick-toteutuksessa käytettiin useita erilaisia syntakseja ja metodikutsuja samankaltaisten operaatioiden suorittamiseen, mikä teki koodista paikoin vaikeasti ymmärrettävää. Uudessa toteutuksessa kaikki kyselyt suoritetaan selkeästi nimettyjen apumetodien kautta, mikä yhtenäistää koodia läpi sovelluksen ja tekee siitä johdonmukaisempaa. BaseDAO-trait tarjoaa intuitiiviset metodit eri tietokantaoperaatioille, mikä helpottaa uusien ominaisuuksien kehittämistä ja koodin ylläpitoa:

```
// Aiempi Slick-toteutus (erilaisia tapoja riippuen kontekstista)

Q.queryNA[RoadName](query).iterator.toSeq sql"""#$query""".as[User].firstOption
sqlu"""#$updateQuery""".buildColl.toList.head

// Uusi ScalikeJDBC-toteutus (yhtenäiset apumetodit)

runSelectQuery(query.map(RoadName.apply))
runSelectSingleOption(query.map(User.apply))
runUpdateToDb(updateQuery)
```

SQL-kyselyiden selkeys ja luettavuus parani ScalikeJDBC-kirjaston interpolaattorien ansiosta. Erityisesti kokoelmien käsittelystä tuli yksinkertaisempaa ja koodia tarvittiin vähemmän. ScalikeJDBC osaa automaattisesti käsitellä kokoelmaparametrit oikein, mikä tekee koodista tiiviimpää ja helpommin luettavaa. Erityisesti kokoelmien käsittelystä tuli merkittävästi yksinkertaisempaa, kun ScalikeJDBC osaa automaattisesti käsitellä kokoelmaparametrit oikein SQL-kyselyissä:

```
// Aiempi Slick-toteutus

WHERE rw.ROADWAY_NUMBER IN (${roadwayNumbers.mkString(", ")})
```

```
// Uusi ScalikeJDBC-toteutus

WHERE rw.roadway_number IN ($roadwayNumbers)
```

Migraation yhteydessä yhtenäistin myös SQL-kyselyjen formaatin ja nimeämiskäytännöt. Sarakkeiden nimet kirjoitetaan nyt pienillä kirjaimilla, SQL-avainsanat isoilla kirjaimilla. Kyselyt on myös jäsennelty selkeästi rivittämällä ne loogisiin kokonaisuuksiin. Sarakkeiden nimet kirjoitetaan nyt johdonmukaisesti pienillä kirjaimilla, SQL-avainsanat isoilla kirjaimilla, ja kyselyt on jäsennelty selkeästi rivittämällä ne loogisiin kokonaisuuksiin. Tämä yhtenäistäminen ei pelkästään paranna koodin esteettistä laatua, vaan tekee myös kyselyistä helpommin luettavia ja ylläpidettäviä:

```
// Aiempi Slick-toteutus

SELECT ID, ROADWAY_NUMBER, ADDR_M, CREATED_BY, CREATED_TIME, MODIFIED_BY, MODI-
FIED_TIME
from ROADWAY_POINT $whereClause

// Uusi ScalikeJDBC-toteutus

SELECT id, roadway_number, addr_m, created_by, created_time, modified_by,
       modified_time
FROM roadway_point
$whereClause
```

Paransin myös koodin dokumentaatiota migraation yhteydessä lisäämällä ScalaDoc-kommentit keskeisiin metodeihin ja luokkiin. Esimerkiksi BaseDAO-trait-rajapinnan ja SessionProvider-objektin metodit on nyt dokumentoitu selkeästi kuvaamaan niiden toimintaa, parametreja ja paluuarvoja:

```
// Aiempi Slick-toteutus

/* OLD Slick 3.0.0 way to run direct SQL update queries. */
def runUpdateToDb(updateQuery: String): Int = {
  sqlu"""#$updateQuery""".buildColl.toList.head //sqlu"""#$updateQuery""".exe-
cute
}

// Nykyinen ScalikeJDBC-toteutus

/**
 * Executes a SELECT query and returns the first result.
 * Example use: runSelectFirst(query.map(rs => rs.A("column_name")))
 *
 * @param query SQL query with result type A
 * @tparam A Type to map the result to
 * @return The first result as an A Type Option
 */
def runSelectFirst[A](query: SQL[A, HasExtractor]): Option[A] = {
    query.first().apply()
}
```

5.4 Työkalut ja menetelmät

Migraatioprojektissa hyödynsin monipuolisesti erilaisia tiedonhankinta- ja kehitysmenetelmiä käyttäen iteratiivista lähestymistapaa. Teknisen tietopohjan rakensin Slick- ja ScalikeJDBC-kirjastojen virallisista dokumentaatioista sekä Scala- sekä tietokantaohjelmointia käsittelevästä kirjallisuudesta sekä ajankohtaisista artikkeleista. Lisäsin tietoperustaan osioita migraation edetessä, esimerkiksi tutuessani thread local -muuttujiin istunnonhallinan vaihtoehtona, päätin lisätä niistä oman osion teoriaosioon.

Vaikka ScalikeJDBC-kirjaston dokumentaatio oli melko niukkaa, se riitti kehitystyön pohjaksi, mutta edellytti kokeilevaa lähestymistapaa käytännön toteutuksessa. Olemassa olevat yksikkö- ja integraatiotestit toimivat sekä tutkimus- että validointimenetelminä migraation aikana. Testien avulla pystyin nopeasti todentamaan, tuottiko valitsemani ratkaisu oikeita tuloksia, sekä tunnistamaan ja korjaamaan migraatioprosessissa ilmenneitä ongelmia. Tämä metodinen lähestymistapa säästi merkittävästi aikaa verrattuna manuaaliseen testaukseen.

Kävin säännöllisesti keskusteluja kokeneempien kehittäjien kanssa, kun tarvitsin apua sovelluksen arkkitehtuurin tai alkuperäisten toteutusten tausta-ajatuksien ymmärtämisessä. Nämä keskustelut auttoivat tunnistamaan kriittisiä kohtia ja välttämään mahdollisia ongelmakohtia. Projektinhallinnassa hyödynsin Jira-tikettejä dokumentoidakseni havaintoja, kehitysideoita ja ratkaisumalleja migraation eri vaiheissa. Tämä mahdollisti tiedon systemaattisen keräämisen ja jakamisen tiimin kesken.

Koodikatselmoinnit muodostivat tärkeän osan kehitysprosessia. Kaikki muutokset vietiin erissä kehityshaaraan GitHubin pull request -toiminnallisuutta hyödyntäen, joka mahdollisti koodimuutosten katselmoinnin vaiheittain. Tiimiläisten palautteen avulla kykenin jalostamaan jokaisen merkittävän migraatiovaiheen ratkaisuja ja varmistamaan niiden laadukkuuden.

Dokumentoin migraatioprosessia tarkasti, kirjaamalla muistiinpanoja havaituista ongelmista ja niiden ratkaisuista sekä tallentamalla koodiesimerkkejä onnistuneista toteutuksista. Ennen laajamittaista toteutusta rakensin prototyyppejä testatakseni ScalikeJDBC-kirjaston soveltuvuutta erilaisiin käyttötapauksiin, mikä auttoi tunnistamaan potentiaalisia ongelmia etukäteen. Koko projektin läpi sovelsin iteratiivista kehitysprosessia: toteutin, testasin ja arvioin migraation rajattuja osia vaiheittain. Tämä lähestymistapa mahdollisti jatkuvan oppimisen ja nopean reagoinnin havaittuihin haasteisiin. Menetelmien yhdistelmällä varmistin migraation teknisen onnistumisen ja samalla kartutin arvokasta kokemusta tietokantakirjastojen vaihtamisesta tuotantojärjestelmässä.

6 Pohdinta

Migraatioprojektin päätyttyä on aika arvioida sen onnistumista, kohdattuja haasteita ja saavutettuja hyötyjä. Tässä luvussa käsittelen kriittisesti migraation eri vaiheita, teknisiä ratkaisuja ja projektin aikana syntyneitä oivalluksia. Tarkastelen myös, miten projekti on vaikuttanut omaan ammatilliseen kehitykseeni, ja mitkä ovat keskeiset jatkokehityssuunnat Viite-sovellukselle migraation jälkeen.

6.1 Migraatiossa ilmenneitä haasteita

Vaikka lopullinen migraatio sujuikin suurilta osin luontevasti, kohtasin prosessin aikana useita haasteita, joista suurimmat liittyivät kirjastojen erilaisiin tapoihin käsitellä tietokantaoperaatioita.

6.1.1 Sarakkeiden käsittelyn haasteet

Suuri osa teknisistä haasteista migraatiossa liittyi tietokantataulujen sarakkeiden käsittelyyn, sillä apply-metodeissa oli käytettävä kyselyjen SELECT-sarakkeiden nimiä. Aiemmin Slickin apply-metodi perustui sarakkeiden järjestykseen, mutta ScalikeJDBC-kirjastolla suositellaan sarakenimien käyttöä ja ne oli määriteltävä manuaalisesti. Tämä teki koodista huomattavasti paljon luettavampaa ja selkeämpää, mutta tekniikka osoittautui riskialttiiksi syntaksivirheille laajamittaisen migraation aikana, koska automaattista täydennystoimintoa ei ollut käytettävissä. Vaikka nämä virheet olivat suhteellisen helppoja havaita ja korjata, monimutkaisemmissa tapauksissa ilmeni haasteita, kuten tilanteissa, joissa useilla yhdisteltävillä tauluilla oli samannimisiä sarakkeita. ScalikeJDBC-kirjaston WrappedResultSet-lähestymistapa hakee sarakkeita nimien perusteella, mutta se ei osaa automaattisesti erottaa, mistä taulusta kukin samanniminen sarake tulee. Tämä aiheutti ongelmia erityisesti monimutkaisissa JOIN-kyselyissä. Yhtenä ratkaisuvaihtoehtona kokeilin sarakkeiden hakemista järjestysnumeroiden perusteella, mutta tämä ei osoittautunut luotettavaksi. Testit paljastivat, että joissain tilanteissa tyyppimuunnokset eivät toimineet odotetusti, mikä saattoi johtua siitä, että sarakkeiden järjestys ei aina ollut täysin ennustettavissa. Mielestäni sarakkeiden nimipohjainen haku onkin parempi tapa toteuttaa hakutulosten mapittaminen, sillä se on intuitiivisempi ja selkeämmin luettava tapa, kuin järjestysnumeroiden hallinta. Liite 8 esittelee monimutkaisen ProjectLink-objektin muodostamistavat Slick ja Scalike menetelmillä.

6.1.2 Null-arvojen käsittelyn haasteet

Kenties eniten päänvaivaa tuottaneista haasteista liittyi NULL-arvojen käsittelyyn tietokannasta. Ongelma ilmeni, kun sama tietokantakysely tuotti erilaisia tuloksia Slick- ja ScalikeJDBC-kirjastoilla. Ongelman ydin oli siinä, että ScalikeJDBC tuotti virheen kohdatessaan tyhjän (NULL) arvon tietokannassa, kun taas vanha Slick-toteutus käsitteli nämä tilanteet hiljaisesti ja vaikutti muuntavan tyhjät Long-tyypin arvot muotoon 0L. Ongelman todellisen syyn selvittäminen oli hankalaa, sillä

en löytänyt Slick-dokumentaatioista mitään mainintaa tyhjien Long-arvojen automaattisesta muuntamisesta. Lopulta päädyin kokeilemaan JDBC-rajapinnan wasNull-metodia Slick-toteutuksessa tulostamaan, miten tyhjiä arvoja käsiteltiin. Paljastui, että Slick muunsi tyhjät arvot automaattisesti nolliksi, ja merkitsi samalla erilliseen muuttujaan tiedon siitä, että alkuperäinen arvo oli tyhjä. Alla tapa, jolla sain selvitettyä ongelman:

```
// Debug lokitukset ennen linear_location_id:n lukemista
  logger.warn(s"""About to read linear_location_id:
  Previous value (roadwayId) = $roadwayId
  wasNull = ${r.wasNull} // Tämän ei pitäisi olla ollut tyhjä arvo
""")

  val linearLocationId = r.nextLong() // Tämä todellisena selvityksen kohteena,
                                      onko arvo tyhjä ennen kuin siitä muuttu 0L

  logger.warn(s"""After reading linear_location_id:
  linearLocationId = $linearLocationId
  wasNull = ${r.wasNull}
""")
```

```
TEST 2024-11-06 10:22:04,651 WARN  [ScalaTest-run-running-ProjectServiceSpec] f.l.v.d.ProjectLinkDAO
[ProjectLinkDAO.scala:263] About to read linear_location_id:
    Previous value (roadwayId) = 1021800
    wasNull = false
TEST 2024-11-06 10:22:04,655 WARN  [ScalaTest-run-running-ProjectServiceSpec] f.l.v.d.ProjectLinkDAO
[ProjectLinkDAO.scala:270] After reading linear_location_id:
    linearLocationId = 0
    wasNull = true
```

Kuva 2: Tulostus, joka todisti arvon olleen tyhjä ennen automaattista muunnosta arvoon 0L

Tämä taustalla tapahtuva NULL-arvon automaattinen käsittely selitti, miksi aiempi toteutus toimi ongelmitta, vaikka tietokannassa oli tyhjiä arvoja. Kun lopulta sain todistettua, miten Slick käsitteli tyhjiä arvoja, pystyin toteuttamaan vastaavan toiminnallisuuden ScalikeJDBC-kirjastolla. Ratkaisu vaati yksinkertaisia muutoksia muutamiin tietokantakyselyihin, jotta järjestelmä toimii samalla tavalla kuin aiemminkin, vaikka tyhjien arvojen käsittely tapahtuukin hieman eri tavalla. Koko apply-metodi, johon lisäsin lokitulosteen selvittääkseni ongelman, löytyy liitteestä 8.

6.1.3 Autosession ja rollback-mekanismeihin liittyneet haasteet

Migraation alkuvaiheessa kokeilin ScalikeJDBC-kirjaston tarjoamaa AutoSession-ominaisuutta, joka mahdollistaa tietokantaistuntojen implisiittisen käytön ilman eksplisiittistä istunnon määrittelyä. Tämä lähestymistapa vaikutti aluksi houkuttelevalta, koska se yksinkertaisti koodia merkittävästi. Käytännön toteutuksessa ja testauksessa kuitenkin ilmeni merkittäviä haasteita AutoSession-

mekanismin käytössä. Testitapaukset epäonnistuivat, kun testin suorittamat tietojen lisäykset eivät olleet näkyvissä saman testitransaktion sisällä tehdyille kyselyille. Lisäksi havaitsin, että testeissä käytetty rollback-toiminnallisuus ei peruuttanut kaikkia testin aikana tehtyjä muutoksia, mikä johti testidatan jäämiseen tietokantaan.

Ongelmien taustalla oli ScalikeJDBC-kirjaston AutoSession-toteutuksen tekninen rakenne. Kirjaston dokumentaatiossa kuvataan, että AutoSession on tarkoitettu tilanteisiin, joissa metodeja halutaan käyttää joustavasti sekä transaktiokontekstissa että sen ulkopuolella. Dokumentaatio selventää, että AutoSession lainaa lukuoikeudellisen istunnon ja palauttaa sen, kun metodia kutsutaan ilman eksplisiittistä istuntoa, mutta käyttää implisiittistä istuntoa, kun sellainen on saatavilla. (ScalikeJDBC 2024d.) Tarkempi tarkastelu ScalikeJDBC-kirjaston lähdekoodista paljasti, että AutoSession-luokassa tietokantayhteys (Connection) on kuitenkin määritelty null-arvoksi ja transaktiotieto (tx) None-arvoksi:

```
override private[scalikejdbc] val conn: Connection = null
override val tx: Option[Tx] = None
val isReadOnly: Boolean = false
```

Käytännössä tämä tarkoitti, että monimutkaisissa transaktioketjuissa ja testeissä AutoSession-pohjainen ratkaisu rikkoi transaktioiden eheyttä, kun operaatiot, joiden olisi pitänyt kuulua samaan atomiseen transaktioon, hajautuivat erillisiin tietokantayhteyksiin.

ScalikeJDBC-dokumentaatiossa suositeltiin testitapuksien rollback-toiminnoksi AutoRollback-rajapintaa, mutta koska halusin kehittää rollback-toiminnon siten, ettei olemassa oleviin testeihin tarvinnut tehdä juurikaan muutoksia säilyttämällä samantyylisen rollback-istunnon kuin Slick-toteutuksessa, tuotti rollback-mekanismin luominen haasteita dokumentaation puutteellisuuden vuoksi.

Havaittujen ongelmien ratkaisemiseksi päädyin kehittämään oman SessionProvider-objektin, joka tarjoaa säieturvallisen tavan hallita tietokantaistuntoja ThreadLocal-muuttujan avulla. Tämä ratkaisu mahdollisti sekä istuntojen implisiittisen käytön että luotettavan transaktionhallinnan varmistamalla, että samaan loogiseen transaktioon liittyvät operaatiot käyttävät samaa tietokantayhteyttä.

6.1.4 Projektinhallintaan liittyneet haasteet

Migraatiota aloitettaessa käytin lähestymistapaa, jossa ScalikeJDBC-pohjaiset luokat luotiin erillisinä uusina tiedostoina, siten että Slick-toteutus säilytettiin rinnalla. Tämä mahdollisti uuden kirjaston käytön opettelun ja erillisten yksikkötestien ajamisen ilman, että vanha toteutus häiriintyi. Kuitenkin projektin edetessä tästä lähestymistavasta muodostui haaste. Uusien DAO-luokkien lisääminen vanhan koodin rinnalle vaati ylimääräistä työtä, kuten erillisten testien ylläpitoa ja kahden eri tietokantayhteyksiä hallitsevan mekanismin yhteensovittamista. Lisäksi palvelukerroksen metodit

eivät voineet kutsua sekä Slick- että ScalikeJDBC-pohjaisia DAO-luokkia samanaikaisesti ilman riskialttiita väliaikaisratkaisuja, mikä vaikeutti vaiheittaista siirtymistä. Keskusteltuani aiheesta projektin teknisen arkkitehdin kanssa, päätin muuttaa lähestymistapaa ja tehdä migraation suoraan korvaamalla olemassa olevat tiedostot. Tämä ratkaisi ongelman koodin hajautumisesta kahteen eri toteutukseen, mutta toi mukanaan uuden haasteen, sillä palvelukerroksen testaus ei ollut mahdollista ennen kuin kaikki siihen liittyvät DAO-kerrokset oli päivitetty käyttämään ScalikeJDBC-kirjastoa. Koska Slick ja ScalikeJDBC käsittelevät tietokantayhteyksiä ja transaktioita eri tavoin, niiden sekoittaminen samassa palveluluokassa olisi voinut johtaa odottamattomiin virheisiin ja epäyhteensopiviin istuntoihin. Tämä tarkoitti, että palvelukerroksien toimivuus voitiin varmistaa vasta, kun kaikki riippuvuudet oli siirretty ScalikeJDBC-kirjastoon, mikä kasvatti kerralla tehtävän muutoksen laajuutta ja vaikeutti iteratiivista testausta. Toisaalta tällä lähestymistavalla, kykenin etenemään migraatiossa nopeammalla tahdilla, vaikkakin uuden koodin testaaminen ja virheiden korjaaminen jäikin käytännössä kokonaan migraation loppupuoliskolle. Koska olin kuitenkin varmistanut toteutuksen toimivuuden aiemmilla prototyypeillä ja perusteellisesti testatuilla apukomponenteilla, pystyin hallitsemaan riskit ja saattamaan migraation onnistuneesti loppuun. Tulevissa migraatioprojekteissa pyrin kuitenkin harkitsemaan tarkemmin lähestymistavan valintaa projektin alkuvaiheessa ja suunnittelemaan testausstrategian huolellisemmin riippuvuuksien hallinnan näkökulmasta.

Migraation aikana kohtasin myös projektin rajaukseen liittyviä haasteita. Alkuperäinen suunnitelmani oli käsitellä opinnäytetyössä vain migraation alkuvaiheet valmistuakseni tutkinnosta aikataulussa. Työn venyessä seuraavalle lukukaudelle, ja sen myötä aikataulun salliessa, päätin kuitenkin laajentaa työn kattamaan koko migraatioprosessin, sillä itse migraatio eteni hyvää tahtia. Vaikka ratkaisu tarjosi mahdollisuuden kattavampaan analyysiin, se kasvatti työn laajuutta merkittävästi ja vaikeutti vapaa-ajallani tapahtuvan dokumentoinnin ja varsinaisten työtehtävien yhteensovittamista, sillä työn kirjoittamiseen kului arvioitua huomattavasti paljon enemmän aikaa. Projektinhallinnan näkökulmasta tämä osoitti, kuinka tärkeää on huomioida laajuuden muutosten vaikutukset kaikkiin työvaiheisiin.

6.2 Migraation onnistumisen arviointi

Mielestäni migraatioprojekti onnistui kokonaisuutena erinomaisesti, sillä kaikki sille asetetut tekniset tavoitteet (kappale 3.2) saavutettiin ja ScalikeJDBC-kirjaston valinta osoittautui onnistuneeksi. Alun perin eniten huolta aiheuttaneet monimutkaiset transaktiot eivät lopulta tuottaneet juurikaan ongelmia sen jälkeen, kun implisiittisten sessioiden käyttö oli saatu ratkaistua SessionProviderin avulla. SessionProvider-komponentin kehittäminen oli mielestäni projektin keskeisimpiä oivalluksia. Se ratkaisee ongelman, joka on yleinen monissa tietokantakirjastoissa, eli istuntojen hallinnan.

Uskon, että tämänkaltaiset räätälöidyt ratkaisut ovat usein välttämättömiä, kun modernisoidaan vanhempia järjestelmiä.

Koin päätöksen säilyttää alkuperäisten SQL-kyselyiden rakenne yhdeksi migraation keskeisistä onnistumisista. Matalan tason SQL-toteutusten säilyttämisellä saavutettiin mielestäni kaksi merkittävää etua: se pienensi merkittävästi migraation riskejä, kun sovelluksen perustoimintalogiikkaa ei tarvinnut uudelleenkirjoittaa, ja se mahdollisti kehitystiimin jo olemassa olevan SQL-osaamisen hyödyntämisen jatkossakin. Vaikka ScalikeJDBC-kirjasto olisi mahdollistanut myös niin kutsuttujen taulualiaksien (engl. table alias) käytön (liite 6), mikä edustaisi eräänlaista välimuotoa ORM-pohjaisen ja puhtaan matalan tason SQL-lähestymistavan välillä, tämä olisi käytännössä tarkoittanut jo toimivien kyselyjen osittaista uudelleenkirjoittamista. Uskonkin, että jos järjestelmää olisi lähdetty kehittämään puhtaalta pöydältä, olisi taulualiaksien käyttö todennäköisesti ollut optimaalinen valinta ScalikeJDBC-kirjastolla, mutta Viite-sovelluksen tapauksessa tämä olisi lisännyt merkittävästi sekä työmäärää että riskejä. Näiden huomioiden pohjalta voikin todeta, että teknologiamigraatiossa ei aina kannata tavoitella kaikkein edistyneintä toteutustapaa, vaan joskus tutun ja hyväksi havaitun lähestymistavan säilyttäminen voi tuottaa parhaan lopputuloksen kokonaisuuden kannalta. Vertaillessa korkeamman abstraktion ja matalan tason lähestymistapoja on selvää, että molemmat lähestymistavat säilyttävät paikkansa modernissa ohjelmistokehityksessä. ORM sopii erityisesti yksinkertaisiin tietokantaoperaatioihin ja nopeaan kehitykseen, kun taas matalan tason SQL-toteutus tarjoaa parempaa kontrollia monimutkaisemmissa tietokantaoperaatioissa ja suorituskykykriittisissä sovelluksissa. Tärkeintä on tunnistaa, milloin tekninen uudistus tuo aitoa arvoa ja milloin se lähinnä lisää monimutkaisuutta ja riskejä ilman merkittäviä hyötyjä.

Vaikka suorituskykyvertailu ei ollut migraation keskeinen tavoite, tein kevyen vertailun DAO-kerroksen testien suoritusnopeudesta ennen ja jälkeen migraation. Tulokset osoittivat, että suorituskyky pysyi käytännössä muuttumattomana, mikä oli odotettavissa, sillä molemmissa toteutuksissa hyödynnettiin suoraa SQL-koodia. Laajempi suorituskykyvertailu olisi hyvä kohde jatkotutkimukselle, mutta nykyiset tulokset antavat alustavia viitteitä siitä, että ScalikeJDBC-kirjasto vastaa tehokkuudeltaan vähintään Slick-kirjaston suoraa SQL-toteutusta.

Myös kattavan testauksen merkitys paljastui projektin aikana entistä selvemmin. Testit eivät vain varmistaneet toiminnallisuuden säilymistä, vaan ne auttoivat myös ymmärtämään kirjastojen toimintalogiikkaa syvällisemmin. Esimerkiksi NULL-arvojen käsittelyssä havaitut erot Slick- ja ScalikeJDBC-kirjastojen välillä olisivat helposti jääneet huomaamatta ilman perusteellista testausta. Viite-sovelluksen testien kattavuus oli itsessään positiivinen yllätys, sillä minun ei tarvinnut juurikaan lisätä uusia testejä migraation aikana, vaan olemassa olevat testit riittivät päivityksen testaamiseen. Se tuli todistettua viimeistään testiautomaation jälkeisellä regressiotestauksella, sillä muutamaa

syntaksivirhettä lukuun ottamatta, kaikki toimi niin kuin pitääkin. Tämä vahvistaa käsitystäni siitä, että testiautomaatio ei ole tärkeää vain laadunvalvonnan kannalta, vaan se on myös arvokas työkalu järjestelmän toiminnan ymmärtämisessä.

Migraation merkitys Viite-sovelluksen tulevaisuudelle on selvä. Kirjaston päivitys poistaa esteen Scala-version päivittämiselle, mikä mahdollistaa koko teknologiapinon modernisoimisen. Samalla koodista tuli selkeämpää ja helpommin ylläpidettävää. Sarakkeiden kutsuminen nimellä indeksien sijaan tekee koodista intuitiivisempaa lukea, kun koodista näkee heti mitä tietoa haetaan. Hyvin dokumentoidut metodit kertovat selvästi niiden tarkoituksen, ja yhtenäiset virheenkäsittelyt näyttävät suoraan missä ja miksi ongelma ilmeni. Tämä kaikki säästää aikaa sekä kehittäjiltä että testaajilta. Uskon vahvasti, että tämä työ kantaa hedelmää pitkälle tulevaisuuteen, kun sovelluksen elinkaari piteni ja jatkokehitys helpottui. Vaikka aikaa migraation dokumentointiin kuluikin paljon, olen tyytyväinen siihen, että dokumentoin prosessin ja löydökset tarkasti, sillä ne voivat auttaa muitakin samankaltaisten haasteiden kanssa painivia kehittäjiä.

6.3 Oma oppiminen

Tämä migraatioprosessi oli ylivoimaisesti suurin kokonaisuus, jota olen lyhyellä urallani hallinnoinut. Mielestäni projekti kehitti osaamistani merkittävästi useilla alueilla. Scala-ohjelmointikielen syvällisempi ymmärrys vahvistui käytännön työssä, kun jouduin perehtymään sen edistyneempiin ominaisuuksiin kuten implisiittisiin parametreihin ja tyyppimuunnoksiin. Tietokantakerroksen eri tasojen hahmottaminen ja erilaisten abstraktioiden merkitys selkiytyi, kun vertailin eri toteutustapojen hyötyjä ja haittoja käytännössä.

Oli huikeaa nähdä, kuinka SessionProvider-komponentin kaltaiset ratkaisut syntyivät iteratiivisesti ongelmia ratkoessa. Projektin aikana opin arvostamaan versionhallinnan tuomaa turvaa, kun kokeilin erilaisia teknisiä ratkaisuja rinnakkaisissa kehityshaaroissa. Käytännössä perehdyin Gitin edistyneempiin toimintoihin kuten haarojen monipuoliseen käyttöön, koodin varastoinnin (engl. stashing) ja paikallisten muutosten hallintaan. Tämä oli todella opettavainen kokemus projektin laajuuden hallinnassa, ja uskonkin, että tulevaisuudessa osaan jakaa monimutkaisen kokonaisuuden hallittaviin osiin huomattavasti helpommin.

Ammatillisen kehitykseni kannalta erityisen merkittävää oli käytännön kokemus teknisten ideaalien ja projektin käytännön tarpeiden yhteensovittamisesta. Opin, että konkreettinen, toimiva ratkaisu voi olla parempi, kuin hienostunut, mutta monimutkainen tekninen toteutus, joka toisi mukanaan ylimääräisiä riskejä. Ymmärrän nyt myös syvällisemmin, miksi teknisen velan käsite on niin keskeinen ohjelmistokehityksessä, ja miten se vaikuttaa projektien pitkäaikaiseen ylläpidettävyyteen.

Opinnäytetyön kirjoittaminen migraation rinnalla pakotti minut jäsentämään ajatuksiani ja dokumentoimaan ratkaisujani perusteellisemmin kuin olisin muuten tehnyt. Mielestäni tämä syvensi omaa ymmärrystäni projektin eri vaiheista ja pakotti minut reflektoimaan tekemiäni ratkaisuja analyyttisemmin. Projektin rajauksen muuttuminen kesken prosessin opetti minulle arvokkaita läksyjä työn laajuuden hallinnasta ja resurssien mitoittamisesta. Opin, että varsinkin dokumentointivaihe vaatii huomattavasti enemmän aikaa kuin alun perin olin arvioinut. Aikataululliset haasteet opinnäytetyön ja migraation yhteensovittamisessa opettivat minulle myös priorisointitaitoja ja realistista tavoitteiden asettelua - taitoja, jotka ovat arvokkaita missä tahansa ohjelmistoprojektissa.

6.4 Jatkokehitys

Migraation onnistuneen toteutuksen myötä Viite-sovellukselle avautuu useita jatkokehitysmahdollisuuksia. Käyttöönotettu ScalikeJDBC-versio 3.4.2 tukee Scala-versioita 2.11–2.13, mikä mahdollistaa Scala-version päivittämisen ensin versioon 2.12 ja myöhemmin 2.13-versioon, jonka jälkeen ScalikeJDBC voidaan päivittää aina uusimpaan asti vain versionumeroa päivittämällä, eikä tietokantakirjasto ole siten enää esteenä Scala-versioille.

Nyt, kun ScalikeJDBC-migraatio on avannut oven Viite-sovelluksen teknologiapinon kokonaisvaltaiselle modernisoinnille, tulisi jatkokehityksen suunnittelussa huomioida pitkän aikavälin tavoitteet ja edetä hallitusti. Migraation laajuuden vuoksi on tärkeää seurata tuotantokäytössä mahdollisesti ilmeneviä virhetilanteita ja reagoida niihin nopeasti. Kehitystiimille olisi hyödyllistä koota ohjeistus tyypillisimmistä virhetilanteista ja niiden ratkaisumalleista, jotta kaikilla olisi valmiudet tunnistaa ja korjata mahdollisia ongelmia tehokkaasti.

Jatkokehityksen kannalta on tärkeää omaksua jatkuvan parantamisen malli. Parannus- ja kehitysideoita tulisi kerätä ylös esimerkiksi Jira-tiketeiksi sitä mukaan, kun niitä ilmenee. Tämä mahdollistaa kehitysideoiden jäljittämisen ja priorisoinnin tulevissa kehitysjaksoissa. Nykyistä teknistä toteutusta voisi lähteä laajentamaan muun muassa BaseDAO-trait-rajapinnan toiminnallisuutta lisäämällä luomalla uusia implisiittisiä mappereita erilaisten tietotyyppien käsittelyyn. Esimerkiksi runSelectQuery-metodiin voitaisiin lisätä tuki implisiittisille mappereille, mikä yksinkertaistaisi kyselyjä entisestään. Myös SQL-SyntaxSupport-ominaisuuden sekä taulualiaksien käyttöönottoa voitaisiin harkita iteratiivisesti erityisesti monimutkaisissa SQL-kyselyissä, joissa on paljon JOIN-liitoksia.

Migraation aikana saadut opit ja kehitetyt komponentit aion dokumentoida kattavasti ja esitellä muun muassa Digiroad-tiimille, joka käyttää samaa vanhentunutta Slick-versiota järjestelmässään. Tarkoituksena on jakaa parhaita käytäntöjä ja tukea organisaation muita tiimejä vastaavissa migraatioprojekteissa.

Lähteet

Abba, I. 2022. What is an ORM – The Meaning of Object Relational Mapping Database Tools. Luettavissa: https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mapping-database-tools/. Luettu 21.9.2024.

Bugnion, P. 2016a. Scala for data science. Packt Publishing. Birmingham. E-kirja. Luettu: 29.9.2024.

Bhattacharjee, R., Radford, D. & Tome, E. 2024. Data Engineering with Scala and Spark. Packt Publishing. Birmingham. Luettu: 1.2.2025.

Caselli, E., Galluccio, E. & Lombari, G. 2020. SQL-Injection Strategies. E-kirja. Luettu 19.10.2024.

Chiusano, P., Bjarnason, R. & Pilquist, M. 2023. Functional Programming in Scala. Toinen painos. Manning Publications. New York. E-kirja. Luettu 12.10.2024.

Connoly, T. 2025. Database Systems. Kuudes painos. Pearson. Harlow.

EPFL. s.a. Tour of scala. Luettavissa: https://docs.scala-lang.org/tour/tour-of-scala.html. Luettu: 21.9.2024.

Housley, M & Reiss, J. 2022. Fundamentals of data engineering. O'Reilly Media, Inc. Sebastopol. E-kirja. Luettu 19.10.2024.

IBM. 2024a. Relational database. Luettavissa: https://www.ibm.com/docs/en/i/7.5?topic=concepts-relational-database. Luettu: 26.9.2024.

IBM. 2024b. Structured Query Language. Luettavissa: https://www.ibm.com/docs/en/i/7.5?topic=concepts-structured-query-language. Luettu: 21.9.2024.

IBM 2024c. Java virtual machine. Luettavissa: https://www.ibm.com/docs/en/i/7.5?topic=platform-java-virtual-machine. Luettu: 20.1.2025.

Kleppmann, M. 2016. O'Reily Media, Inc. Sebastopol. E-kirja. Luettu: 20.1.2025.

Klimenko, A. 2024. Effective Use of ThreadLocal in Java Applications. Luettavissa: https://medium.com/@alxkm/effective-use-of-threadlocal-in-java-applications-f4eb6a648d4a. Luettu 12.1.2025.

Lightbend. 2024. Slick. Luettavissa: https://github.com/slick/slick. Luettu 5.10.2024.

Nord, K., Kructen, P. & Ozkaya, I. 2019. Managing Technical Debt: Reducing Friction in Software Development. Addison-Wesley Professional. Boston. E-kirja. Luettu 28.9.2024.

MvnRepository. s.a. https://mvnrepository.com/artifact/org.scalikejdbc/scalikejdbc. Luettu. 28.10.2024.

Oracle. s.a.a. Data Access Object. Luettavissa: https://www.oracle.com/java/technologies/data-access-object.html. Luettu: 16.2.2025.

Oracle .s.a.b. Class Thread. Luettavissa: https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html. Luettu 16.2.2025.

Oracle. 2024. Thread-Local Variables. Luettavissa: https://docs.oracle.com/en/java/javase/21/core/thread-local-variables.html#GUID-2CEB9041-3DF7-43DA-868F-E0596F4B63FD. Luettu 12.1.2025.

Oracle. 2017. Java JDBC API. Luettavissa: https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/. Luettu: 24.9.2024.

Petrov, A. 2019. Database Internals. O'Reily Media, Inc. Sebastopol. E-kirja. Luettu: 14.10.2024.

Podjarny, G. 2018. Mitigating known security risks in open source libraries. Luettavissa: https://www.oreilly.com/content/mitigating-known-security-risks-in-open-source-libraries/. Luettu 29.9.2024.

Podjarny, G. 2017. Securing Open Source Libraries. O'Reilly Media, Inc. Sebastobol. E-kirja. Luettu 5.10.2024.

Roy, S. 2024. Runtime vs. Compile Time. Luettavissa: https://www.baeldung.com/cs/runtime-vs-compile-time. Luettu 19.12.2024.

Scala. s.a.a Contextual Parameters, aka Implicit Parameters. Luettavissa: https://docs.scala-lang.org/tour/implicit-parameters.html. Luettu 19.12.2024.

Scala. s.a. String Interpolation. Luettavissa: https://docs.scala-lang.org/scala3/book/string-interpolation.html. Luettu 9.11.2024.

ScalikeJDBC. 2024a. ScalikeJDBC. Luettavissa: https://scalikejdbc.org/. Luettu 21.9.2024.

ScalikeJDBC. 2024b. Transaction. Luettavissa: https://scalikejdbc.org/documentation/transaction.html. Luettu 13.10.2024.

ScalikeJDBC. 2024c .SQLInterpolation. Luettavissa: https://scalikejdbc.org/documentation/sql-interpolation.html. Luettu 13.10.2024.

ScalikeJDBC. 2024d. Auto Session. Luettavissa: https://scalikejdbc.org/documentation/auto-session.html. Luettu 25.1.2025.

Sharan, K. 2018. Java APIs, Extensions and Libraries: With JavaFX, JDBC, jmod, jlink, Networking, and the Process API. Apress. E-kirja. Luettu 13.10.2024.

Tal, L. 2024. Raw SQL Queries are Actually Better for Security Than ORMs? Luettavasissa: https://www.nodejs-security.com/blog/raw-sql-queries-better-for-security-than-orms. Luettu 26.10.2024.

Tieteen termipankki. 2024. Perinnejärjestelmä. Luettavissa: https://tieteentermipankki.fi/wiki/Tietojenkäsittelytiede:perinnejärjestelmä. Luettu. 2.10.2024.

TypeSafe, Inc. 2014a. Connections / Transactions. Luettavissa: https://scala-slick.org/doc/2.1.0/connection.html. Luettu 5.10.2024.

TypeSafe, Inc. 2014b. Plain SQL queries. Luettavissa: https://scala-slick.org/doc/2.1.0/sql.html. Luettu 7.10.2024.

TypeSafe, Inc. 2014. Coming from SQL to Slick. Luettavissa: https://scala-slick.org/doc/2.1.0/sql-to-slick.html. Luettu 12.10.2024.

Wadge, W. 2014. BoneCP is deprecated - move to HikariCP. Luettavissa: https://jira.atlassian.com/browse/BSERV-5402. Luettu. 12.10.2024.

Liitteet

Näissä liitteissä esitellään migraation keskeisiä teknisiä toteutuksia ja komponentteja. Liitteet tarjoavat yksityiskohtaista tietoa ScalikeJDBC-kirjaston käyttöönotosta, istunnonhallinnasta ja tietokantaoperaatioiden toteuttamisesta. Ne täydentävät varsinaista opinnäytetyötä tarjoamalla konkreettisia koodiesimerkkejä ja ratkaisumalleja, joita voidaan hyödyntää vastaavanlaisissa migraatioprojekteissa.

Liite 1. PostGISDatabaseScalikeJDBC-objekti

PostGISDatabaseScalikeJDBC-objekti toimii tietokantayhteyden alustajana ja hallinnoijana ScalikeJDBC-kirjastolla. Se määrittelee tietokantayhteysasetukset, alustaa yhteydet ja tarjoaa metodit erilaisten tietokantaoperaatioiden suorittamiseen. Objekti hyödyntää SessionProvider-komponentin toiminnallisuutta istuntojen hallintaan ja ScalikeJDBC-kirjaston tarjoamia metodeja transaktioiden käsittelyyn. Objekti sisältää kolme pääasiallista metodia eri käyttötarkoituksiin:

- *runWithTransaction* operaatioihin, jotka muokkaavat tietokantaa
- *runWithReadOnlySession* tietokannan lukuoperaatioihin
- *runWithRollback* testikäyttöön, jossa muutokset perutaan automaattisesti

```
import scalikejdbc._
import SessionProvider._


object PostGISDatabaseScalikeJDBC {
  // Load the PostgreSQL driver
  Class.forName("org.postgresql.Driver")

  // Initialize the connection pool with default settings
  ConnectionPool.singleton(
    url = ViiteProperties.dbJdbcUrl,
    user = ViiteProperties.dbUsername,
    password = ViiteProperties.dbPassword
  )

  // Logging enabled for all queries for development purposes
  GlobalSettings.loggingSQLAndTime = LoggingSQLAndTimeSettings(
    enabled = true,
    logLevel = 'info
  )


  /**
   * Executes `databaseOperation` block within a single transaction, ensuring
          that all data modifications
   * are committed only if the entire operation is successful.
```

```
   * Uses withSession to handle the session and to prevent nested transactions.
   * If any action fails, the transaction is rolled back, preventing partial
          updates and maintaining data integrity.
   *
   * @param databaseOperation The operation to run
   * @tparam Result The return type of the operation
   * @return The result of the operation
   */
  def runWithTransaction[Result](databaseOperation: => Result): Result = {
    DB.localTx { session =>
      withSession(session) {
        databaseOperation
      }
    }
  }

   /**
   * Executes `readOnlyOperation` within a read-only session.
   *
   * Use this method for database operations that only read data.
   * If a write operation is attempted within this session,
      a `java.sql.SQLException` will be thrown.
   *
   * @param readOnlyOperation The operation to execute.
   * @tparam Result The return type of the operation.
   * @return The result of `readOnlyOperation`.
   * @throws java.sql.SQLException If a write operation is attempted in
      read-only mode.
   */
  def runWithReadOnlySession[Result](readOnlyOperation: => Result): Result = {
    DB.readOnly { session =>
      withSession(session) {
        readOnlyOperation
      }
    }
  }

  /**
   * Executes `testOperation` within a transaction that is rolled back after
      execution.
   * This method is useful for testing purposes, allowing you to perform data
      modifications without persisting changes.
   * After `testOperation` completes, the transaction is rolled back, discarding
      any data modifications.
   *
   * @param testOperation The operation to execute.
   * @tparam Result The return type of the operation.
   * @return The result of `testOperation`.
   */
  def runWithRollback[Result](testOperation: => Result): Result = {
    DB.localTx { session: DBSession =>
      withSession(session) {
        val result = testOperation
        session.connection.rollback()
        result
      }
    }
  }
}
```

Liite 2. SessionProvider-objekti

```
/**
 * Provides a thread-local session for ScalikeJDBC to handle transactions.
 * This object maintains thread-local state for database sessions and
       transaction status, ensuring that database operations are properly
scoped and preventing nested  transactions.
 */
object SessionProvider {
  // Tracks the current database session per thread
  private val threadLocalSession = new ThreadLocal[DBSession]()

  /**
   * Provides implicit access to the current thread's database session.
   * This allows ScalikeJDBC queries to automatically use the correct session.
   *
   * @throws IllegalStateException if no session is currently set
   * @return The current DBSession for this thread
   */
  implicit def session: DBSession = threadLocalSession.get() match {
    case null =>
      val errorMsg =   "No DBSession is set. Ensure you are within a transaction
                       or session."
      logger.error(errorMsg)
      throw new IllegalStateException(errorMsg)
    case s => s // Use the current session
  }

  /**
   * Executes code block in a database session, preventing nested transactions.
   * Sets up the session, runs the operation, and ensures cleanup afterwards by
restoring previous session.
   *
   * @param newDbSession The `DBSession` to use during `f`.
   * @param f          The block of code to execute.
   * @tparam T         The return type of `f`.
   * @throws IllegalStateException if called within an existing transaction
   */
  def withSession[T](newDbSession: DBSession)(f: => T): T = {
    val currentSession = threadLocalSession.get() // Save the current session
    if (newDbSession.tx.isDefined && currentSession != null) { // New transac-
tion sessions are not allowed inside existing sessions
      val currentType = currentSession match {
        case s if s.tx.isDefined => "transaction session"
        case s if s.isReadOnly => "read-only session"
        case _ => "autocommit session"
      }
      throw new IllegalStateException(s"Can't start transaction inside $cur-
rentType")
    }

    try {
      threadLocalSession.set(newDbSession) // Set the new session
      f
    } finally {
      threadLocalSession.set(currentSession) // Restore the previous session
    }
  }
}
```

SessionProvider objektissa on määritelty withSession-metodi, joka mahdollistaa tietokantaistuntojen hallinnan määritellyn koodilohkon sisällä. Metodi tallentaa mahdollisen aiemman istunnon, asettaa uuden istunnon käyttöön koodilohkon ajaksi ja palauttaa alkuperäisen istunnon koodilohkon suorituksen jälkeen. SessionProvider-metodin ehto estää sisäkkäiset transaktiot, mutta sallii lukuistuntojen avautumisen (read-only) myös transaktion sisällä. Käytännössä tämä tarkoittaa:

1. Jos koodissa yritetään luoda uutta transaktioistuntoa silloin, kun vanha istunto on jo transaktiossa, SessionProvider heittää poikkeuksen. Näin estetään hallitsemattomat sisäkkäiset transaktiot.
2. Vastaavasti, jos uusi istunto olisi transaktio, mutta olemassa oleva istunto on read-only, SessionProvider myös heittää virheen. Tämän ansiosta koodi ei vahingossa muuta lukuistuntoa transaktioksi kesken prosessin.
3. Lukuistunto (read-only) ei määritä tx.isDefined-flagia, joten ehtorakenne ei estä read-only-tilan avaamista jo käynnissä olevan transaktion aikana. Näin voi olla tilanteissa, joissa isompi transaktiokokonaisuus käsittelee dataa, mutta yksittäinen osa-alue halutaan lukea read-only-tilassa.

Erityisen hyödyllinen ominaisuus SessionProvider-objektissa on implisiittinen session-muuttuja, jonka avulla tietokantaistunto voidaan tuoda helposti käytettäväksi millä tahansa koodialueella. Kehittäjä voi hyödyntää tätä ominaisuutta lisäämällä import-lauseen koodinsa alkuun:

```
import fi.vaylavirasto.viite.postgis.SessionProvider.session
```

Tämän jälkeen implisiittinen istunto on automaattisesti käytettävissä kaikissa ScalikeJDBC-kirjaston metodeissa, jotka tarvitsevat istuntoparametria. Tämä yksinkertaistaa koodia merkittävästi, koska istuntoa ei tarvitse välittää eksplisiittisesti metodikutsujen välillä tai määritellä toistuvasti implicit-parametriksi jokaiseen metodiin.

Liite 3. BaseDAO-trait

Tässä liitteessä on kuvattuna koko BaseDAO-tiedosto ja trait-rajapinnan tietokantaoperaatioita varten luodut apumetodit. BaseDAO-trait tarjoaa yhtenäisen ja turvallisen rajapinnan tietokantaoperaatioiden suorittamiseen DAO-luokissa. Se sisältää monipuolisen valikoiman apumetodeja, jotka helpottavat erilaisten SQL-kyselyiden suorittamista ja tulosten käsittelyä.

Koodissa näkyvät tyyppimäärittelyt kuten SQL[Nothing, NoExtractor] ja SQL[A, HasExtractor] liittyvät ScalikeJDBC-kirjaston tapaan käsitellä erilaisia SQL-kyselyitä. Ensimmäinen tyyppiparametri (Nothing tai A) määrittelee kyselyn tuloksen tietotyypin: Nothing tarkoittaa, että kyselyn tulosta ei ole vielä mapitettu Scala-olioksi, kun taas A viittaa jo mapitettuun tulostyyppiin. Toinen tyyppiparametri (NoExtractor tai HasExtractor) kertoo, onko kyselylle määritelty tulosrivien käsittelijää: NoExtractor tarkoittaa, että kyselylle ei ole vielä määritelty tulosten käsittelijää, kun taas HasExtractor ilmaisee, että kysely sisältää toiminnallisuuden tulosrivien muuntamiseen halutuiksi olioiksi. Näiden tyyppimäärittelyjen avulla varmistetaan tyyppiturvallinen kyselyn tulosten käsittely ja mahdollistetaan virhetilojen havaitseminen jo käännösaikana.

```
package fi.vaylavirasto.viite.dao

import scalikejdbc._
import org.slf4j.{Logger, LoggerFactory}
import fi.vaylavirasto.viite.postgis.SessionProvider.session
import fi.vaylavirasto.viite.util.ViiteException
import java.sql.Date

// Methods to run queries using ScalikeJDBC and session provider
trait BaseDAO {
  protected def logger: Logger = LoggerFactory.getLogger(getClass)

  /**
   * Executes an UPDATE, INSERT, or DELETE query.
   *
   * @param updateQuery SQL query to execute
   * @return The number of rows affected
   */
  def runUpdateToDb(updateQuery: SQL[Nothing, NoExtractor]): Int = {
    updateQuery.update.apply()
  }

  /**
   * Executes a batch update query with multiple parameter sets.
   *
   * @param query query SQL query to execute
   * @param batchParams Sequence of parameter sets, each set corresponding to
one batch operation
   * @return List of numbers indicating rows affected by each batch operation
   */
  def runBatchUpdateToDb(query: SQL[Nothing, NoExtractor], batchParams:
Seq[Seq[Any]]): List[Int] = {
    query.batch(batchParams: _*).apply()
  }
```

```
  /**
   * Executes a SELECT query and returns all results.
   *
   * @param query SQL query with result type A
   * @tparam A Type to map the results to
   * @return List of results mapped to type A
   */
  def runSelectQuery[A](query: SQL[A, HasExtractor]): List[A] = {
    query.list().apply()
  }

  /**
   * Executes a SELECT query and returns the first result.
   * Example use: runSelectFirst(query.map(rs => rs.A("column_name")))
   *
   * @param query SQL query with result type A
   * @tparam A Type to map the result to
   * @return The first result as an A Type Option
   */
  def runSelectFirst[A](query: SQL[A, HasExtractor]): Option[A] = {
      query.first().apply()
  }

  /**
   * Executes a SELECT query and returns a single matched row as an Option.
   * If multiple rows are returned unexpectedly, a runtime exception is thrown.
   *
   * @param query SQL query with result type A
   * @tparam A Type to map the result to
   * @return None if no results found, Some(result) if exactly one exists
   * @throws ViiteException if multiple rows are returned
   */
  def runSelectSingleOption[A](query: SQL[A, HasExtractor]): Option[A] = {
    try {
      query.single().apply()
    } catch {
      case e: IllegalStateException => throw ViiteException(e.getMessage)
    }
  }

  /**
   * Executes a SELECT query and returns a single matched row.
   * Throws ViiteException if no results or more than 1 result is found.
   *
   * @param query SQL query with result type A
   * @tparam A Type to map the result to
   * @return The first result
   * @throws ViiteException if no results found or if multiple rows are returned
   */
  def runSelectSingle[A](query: SQL[A, HasExtractor]): A = {
    try {
      query.single().apply().getOrElse(
        throw ViiteException("No result found")
      )
    } catch {
      case e: IllegalStateException => throw ViiteException(e.getMessage)
    }
  }

  /**
```

```
    * Executes a SELECT query and maps the single found result Option using an
implicit mapper function.
    * Example use: runSelectSingleFirstOptionWithType[Long](query)
    *
    * @param query SQL query to execute
    * @param mapper Implicit function to convert WrappedResultSet to type T
    * @tparam T Type to map the result to
    * @return None if no results found, Some(T) if exactly one result exists
    * @throws ViiteException if multiple rows are returned
    */
  def runSelectSingleOptionWithType[T](query: SQL[Nothing, NoExtractor])(im-
plicit mapper: WrappedResultSet => T): Option[T] = {
    try {
      query.map(mapper).single().apply()
    } catch {
      case e: IllegalStateException => throw ViiteException(e.getMessage)
    }
  }

  /**
    * Executes a SELECT query and maps the single found result using an implicit
mapper function.
    * If no results are found, a ViiteException is thrown.
    * Example use: runSelectSingleFirstWithType[Long](query)
    *
    * @param query SQL query to execute
    * @param mapper Implicit function to convert WrappedResultSet to type T
    * @tparam T Type to map the result to
    * @return The mapped result
    * @throws ViiteException if no results found or if multiple rows are returned
    */
  def runSelectSingleWithType[T](query: SQL[Nothing, NoExtractor])(implicit map-
per: WrappedResultSet => T): T = {
    try {
      query.map(mapper).single().apply().getOrElse(
        throw ViiteException("No value returned")
      )
    } catch {
      case e: IllegalStateException => throw ViiteException(e.getMessage)
    }
  }

  // Implicit conversions for common types
  // (1) maps the first column of a result set to the desired type
  implicit val longMapper: WrappedResultSet => Long = _.long(1)
  implicit val stringMapper: WrappedResultSet => String = _.string(1)
  implicit val intMapper: WrappedResultSet => Int = _.int(1)
  implicit val dateTimeMapper: WrappedResultSet => Date = _.date(1)
  // Add more implicit mappers as needed

}
```

Päivitysmetodit:

- *runUpdateToDb:* Suorittaa yksittäisen päivityskyselyn (UPDATE, INSERT, DELETE). Metodi huolehtii parametrien turvallisesta käsittelystä ja palauttaa päivitettyjen rivien määrän.

- *runBatchUpdateToDb*: Mahdollistaa useiden samankaltaisten päivitysten suorittamisen tehokkaasti yhtenä eränä. Hyödyllinen esimerkiksi useiden rivien lisäämisessä tai päivittämisessä.

Hakumetodit:

- *runSelectQuery*: Hakee kaikki kyselyn tulokset listana. Käytetään kun halutaan hakea useita rivejä.
- *runSelectFirst*: Palauttaa kyselyn ensimmäisen tuloksen optiona. Hyödyllinen kun tiedetään, että tuloksia on korkeintaan yksi.
- *runSelectSingleOption*: Varmistaa, että kysely palauttaa korkeintaan yhden tuloksen. Heittää poikkeuksen, jos löytyy useampi tulos.
- *runSelectSingle*: Kuten edellinen, mutta palauttaa tuloksen ilman optio-käärettä. Heittää poikkeuksen, jos tulosta ei löydy.

Tyyppimuunnosmetodit:

- *runSelectSingleOptionWithType*: Muuntaa yksittäisen tuloksen halutuksi tietotyypiksi käyttäen implisiittisiä mappereita.
- *runSelectSingleWithType*: Sama kuin edellinen, mutta ilman optio-käärettä.

Impliittiset mapperit:

- Trait sisältää valmiit mapperit yleisimmille tietotyypeille (Long, String, Int, Date). Nämä helpottavat tyyppimuunnoksia yksinkertaisissa kyselyissä, joissa halutaan hakea vain yksi sarake.

Nämä apumetodit yksinkertaistavat tietokantaoperaatioiden toteuttamista Viite-sovelluksessa ja tekevät koodista selkeämpää, turvallisempaa ja yhtenäisempää, kun kaikki tarvittavat metodit löytyvät keskitetysti yhdestä rajapinnasta.

Liite 4. Update- ja BatchUpdate-toteutukset ScalikeJDBC-kirjastolla

Tässä liitteessä esitellään, miten kirjastoa hyödynnettiin tietokantapäivityksien toteuttamiseen Viite-sovelluksessa.

BatchUpdate-toteutus

BatchUpdate-operaatiolla voidaan lisätä useita rivejä tehokkaasti yhdellä tietokantakutsulla:

```
// Method to insert batch of Node Points to database
def create(nodePoints: Iterable[NodePoint]): Seq[Long] = {

// Populate parameters for batch update
  val batchParams: Iterable[Seq[Any]] = createNodePoints.map {
    nodePoint =>
      Seq(
        nodePoint.id,
        nodePoint.beforeAfter.value,
        nodePoint.roadwayPointId,
        nodePoint.nodeNumber,
        nodePoint.nodePointType.value,
        nodePoint.createdBy
      )
  }

// SQL-insert query for the update
  val query =
    sql"""
          INSERT INTO NODE_POINT (ID, before_after, roadway_point_id,
               node_number, type, created_by)
          VALUES (?, ?, ?, ?, ?, ?)
    """

// Run the batch update using runBatchUpdateToDb helper method
  runBatchUpdateToDb(query, batchParams.toSeq)

}
```

Tässä yksinkertaistetussa esimerkissä luodaan useita NodePoint-objekteja tietokantaan. Jokaisen objektin kenttäarvot kootaan Seq-kokoelmaan, ja nämä kokoelmat välitetään batch-parametreina SQL-kyselylle. ScalikeJDBC korvaa kyselyssä olevat kysymysmerkit parametrien arvoilla järjestyksessä.

Yksittäisen päivityksen toteutus

Tavalliseen päivitysoperaatioon käytetään myös BaseDAO-trait-rajapinnan apumetodia:

```
// Expire Node Points using runUpdateToDb helper method
def expireById(ids: Iterable[Long]): Int = {
  val query =
    sql"""
        UPDATE NODE_POINT
        SET valid_to = CURRENT_TIMESTAMP
        WHERE valid_to IS NULL
        AND id IN ($ids)
    """
    runUpdateToDb(query)
  }
}
```

Tämä metodi merkitsee valitut NodePoint-oliot vanhentuneiksi asettamalla niiden valid_to-kentän arvoksi nykyisen aikaleiman. ScalikeJDBC käsittelee automaattisesti IN-lauseen parametrit, joten kokoelman välittäminen on suoraviivaista.

Molemmissa esimerkeissä ScalikeJDBC huolehtii parametrien turvallisesta käsittelystä, mikä suojaa sovellusta SQL-injektioilta.

Liite 5. Sqls-interpolaattori, parametrien sitominen ja kyselyjen dynaaminen rakentaminen

ScalikeJDBC-kirjaston interpolaattori käsittelee automaattisesti kokoelmatyyppisiä parametreja, kuten listoja ja sekvenssejä. Se muuntaa ne SQL-turvallisiksi parametreiksi, jolloin kehittäjän ei tarvitse huolehtia merkkijonojen yhdistämisestä tai pilkkujen lisäämisestä arvojen väliin.

```
// Slick esimerkki
val query =
s"SELECT * FROM JUNCTION WHERE NODE_NUMBER in (${nodeNumbers.mkString(", ")})"

// ScalikeJDBC-toteutus
val query =
sql"SELECT * FROM junction WHERE node_number IN ($nodeNumbers)"
```

Monimutkaisemmissa kyselyissä, joissa tarvittiin dynaamista SQL-kyselyiden rakentamista, hyödynsin ScalikeJDBC-kirjaston sqls-interpolaattoria. Tämä osoittautui hyödylliseksi erityisesti kyselyissä, joissa oli useita valinnaisia parametreja. Esimerkiksi NodeDAO-luokan fetchNodesForRoadAddressBrowser-metodissa valinnaisten hakuehtojen lisääminen onnistuu selkeästi sqls-interpolaattorin avulla:

```
// Rakennetaan SQL-lauseen osat sqls-interpolaattorilla
val dateCondition = situationDate.map { date => sqls"AND rw.start_date <=
        ${date}::date".getOrElse(sqls"")
val elyCondition = ely.map(ely => sqls" AND rw.ely = $ely").getOrElse(sqls"")
val roadNumberCondition = roadNumber.map(roadNumber => sqls" AND rw.road_number
= $roadNumber").getOrElse(sqls"")

// Yhdistetään osat suoritettavaksi kyselyksi sql-interpolaattorilla
sql"""
  $query
  WHERE node.valid_to IS NULL AND node.end_date IS NULL
  $dateCondition
  $elyCondition
  $roadNumberCondition
  $roadPartCondition
  ORDER BY rw.road_number, rw.road_part_number, rp.addr_m
  """
```

Sqls-interpolaattori toimii yhdessä sql-interpolaattorin kanssa siten, että sqls:llä luodaan SQL-kyselyn osia, jotka voidaan yhdistää lopulliseen kyselyyn. Sqls tuottaa SQLSyntax-tyyppisiä olioita, joita voidaan turvallisesti yhdistää sql-interpolaattorin sisällä. Kyselyjen dynaamisessa rakentamisessa sqls on erityisen hyödyllinen, koska:

1. Jokainen sqls-osa käsitellään erikseen parametrisoituna, mikä säilyttää SQL-injektion eston
2. Tyhjä sqls"" ei vaikuta kyselyn syntaksiin, joten ehtolauseita voidaan lisätä dynaamisesti
3. SQLSyntax-olioita voidaan yhdistää turvallisesti osaksi suurempaa kyselyä

Esimerkiksi valinnaisen päivämääräehdon lisääminen kyselyyn:

```
def createDateFilter(dates: Option[Seq[DateTime]]) = dates match {
  case Some(dateList) => sqls"AND start_date IN (${dateList})"
  case None => sqls"" // Tyhjä SQLSyntax, ei vaikuta kyselyyn
}

val finalQuery = sql"""  // Lopullinen SQL-kysely
  SELECT * FROM road_data
  WHERE status = 'active'
  ${createDateFilter(selectedDates)}
  """
```

Tämä lähestymistapa mahdollistaa monimutkaisten kyselyiden rakentamisen dynaamisesti ja turvallisesti, säilyttäen samalla koodin selkeyden ja ylläpidettävyyden.

Liite 6. RoadNameDAO, SQL-SyntaxSupport ja table alias

```
case class RoadName     (
                        id: Long,
                        roadNumber: Long,
                        roadName: String,
                        startDate: Option[DateTime],
                        endDate: Option[DateTime] = None,
                        validFrom: Option[DateTime] = None,
                        validTo: Option[DateTime] = None,
                        createdBy: String
                        )

object RoadName extends SQLSyntaxSupport[RoadName] {
  override val tableName = "road_name"

  def apply(rs: WrappedResultSet): RoadName = RoadName(
    id          = rs.long("id"),
    roadNumber  = rs.long("road_number"),
    roadName    = rs.string("road_name"),
    startDate   = rs.jodaDateTimeOpt("start_date"),
    endDate     = rs.jodaDateTimeOpt("end_date"),
    validFrom   = rs.jodaDateTimeOpt("valid_from"),
    validTo     = rs.jodaDateTimeOpt("valid_to"),
    createdBy   = rs.string("created_by")
  )
}
```

RoadName-objektissa määritelty apply-metodin avulla tietokantatulos mapitetaan Scala-olioiksi. Metodi ottaa parametrinaan WrappedResultSet-olion, joka sisältää tietokantakyselyn tuloksen, ja luo tuloksen tietojen perusteella uuden RoadName-instanssin. Se lukee road_name-taulun sarakkeet, kuten id, road_number, road_name, sekä optiotyyppiset päivämäärät startDate, endDate, validFrom ja validTo, hyödyntäen ScalikeJDBC-kenttien hakumetodeja, kuten rs.long ja rs.jodaDateTimeOpt. SQLSyntaxSupport-trait-rajapinnan käyttö mahdollistaa taulun nimen määrittelyn (tableName = "road_name") ja helpottaa SQL-kyselyiden rakentamista sekä aliaksien käyttöä.

```
// rn is an alias for RoadNameScalike object to be used in queries
private val rn = RoadName.syntax("rn")

/**
 * Get current roadNames by roadNumbers (endDate is null)
 */
def getCurrentRoadNamesByRoadNumbers(roadNumbers: Seq[Long]): Seq[RoadName] = {
  val query = sql"""
    $selectAllFromRoadNameQuery
    WHERE ${rn.roadNumber} IN ($roadNumbers)
      AND ${rn.endDate} IS NULL
      AND ${rn.validTo} IS NULL
  """

  queryList(query)
}
```

Tässä getCurrentRoadNamesByRoadNumbers-metodissa on esimerkki SQL-kyselyn rakentamisesta käyttämällä ScalikeJDBC-kirjaston SQL-interpolointia. Muuttuja ”rn" toimii aliaksena RoadName-objektille, mikä helpottaa kyselyjen kirjoittamista ja tekee niistä luettavampia, erityisesti monimutkaisemmissa tilanteissa, kuten liitoksissa (engl. join). DAO-Tulokset mapitetaan Scala-olioiksi kutsumalla queryList-metodia:

```
// Method to run select queries for RoadName objects
private def queryList(query: SQL[Nothing, NoExtractor]): Seq[RoadName] = {
  runSelectQuery(query.map(RoadName.apply))
}
```

Loin jokaisella DAO-luokalle oman queryList metodin, joka palauttaa tulokset kyseisen DAO-luokan tyyppimuodossa. Tässä esimerkissä queryList-metodi luotu helpottamaan kyselyjä, jotka palattavat RoadName-objekteja.

Liite 7. PingDAO ja yhteyden testaus

Migraatioprosessin alussa ensimmäinen luomani BaseDAO-luokan apumetodi, oli runWithReadOnlyImplicitSession, joka yksinkertaistaa tietokantaoperaatioiden suorittamista lukuoikeuksilla ScalikeJDBC-kirjaston avulla käyttäen kirjaston perinteistä tapaa implisiittisen instunnon (implicit session) välittämiseen:

```
def runWithReadOnlyImplicitSession[Result](readOnlyOperation: DBSession => Result): Result = {
  DB.readOnly { implicit session =>
    readOnlyOperation(session)
  }
}
```

Huomioitavaa: runWithReadOnlyImplicitSession -metodi ei jäänyt lopulliseen toteutukseen, sillä sessionProider-objektin myötä implisiittisistä istuntojen määrittelyistä voitiin luopua.

Ensimmäinen kokeilu tämän metodin hyödyntämisestä tehtiin PingApi-rajapinnan avulla. PingDAO-luokassa oli aiemmin Slick-kirjastoon perustuva toteutus, jonka muunsin ScalikeJDBC-versioksi käyttämällä uutta apumetodia:

```
// Get the current time from the database
def getDbTime(implicit session: DBSession): String = {
  sql"""SELECT TO_CHAR(NOW(), 'YYYY-MM-DD HH24:MI:SS')"""
    .map(rs => rs.string(1)) // Get the first column as a string
    .single() // Get the 1 row as an Option or None
    .apply() // Execute the query
    .getOrElse( // Throw an exception if the result is None
      throw new RuntimeException("Failed to get the current time from the database")
    )
}
```

Tässä getDbTime-metodissa suoritetaan SQL-kysely, joka hakee nykyisen ajan tietokannasta. Apumetodin ansiosta istuntoa ei tarvitse määritellä erikseen, vaan se hoidetaan taustalla PingApi-luokassa seuraavasti:

```
get("/") {
  PostGISDatabaseScalikeJDBC.runWithReadOnlyImplicitSession { implicit session =>
    try {
      val dbTime = PingDAO.getDbTime
      Ok(s"OK (DB Time: $dbTime)\n")
    } catch {
      case e: Exception =>
        logger.error("Ping failed. DB connection error.", e)
        InternalServerError("Ping failed. DB connection error.")
    }
  }
}
```

Lisäsin uuden testin, joka varmistaa tietokantakyselyn onnistumisen ja palautetun ajan oikean muodon:

```
test("Ping API should return OK and database time in correct format") {
    get("/ping") {
      body should include("OK")
      body should include("DB Time:")
      body should fullyMatch regex """OK \(DB Time: \d{4}-\d{2}-\d{2}
\d{2}:\d{2}:\d{2}\)\n"""
    }
  }
}
```

Liite 8. ProjectLink-objektin SQL-kysely sekä Slick- ja ScalikeJDBC-versiot apply-metodista

Tässä liitteessä esitellään monimutkaisempi SQL-kysely ja sen mapittaminen ScalikeJDBC-muotoon. Esimerkkinä toimii Viite-sovelluksen ProjectLink-luokka, joka on yksi sovelluksen monimutkaisimmista tietomallinnusobjekteista sisältäen lukuisia kenttiä ja riippuvuuksia useisiin eri tauluihin.

Alla oleva kysely hakee ProjectLink-objektin tiedot useiden taulujen liitoksilla. Kysely sisältää muun muassa CASE-lausekkeita, aliaksia ja laskettuja kenttiä, mikä tekee siitä hyvän esimerkin monimutkaisen tietomallin käsittelystä.

SQL-kyselyn osa, jolla sarakkeet haetaan:

```
SELECT
  plh.id, plh.project_id,
  plh.track, plh.discontinuity_type,
  plh.road_number,
  plh.road_part_number,
  plh.start_addr_m,
  plh.end_addr_m,
  plh.original_start_addr_m,
  plh.original_end_addr_m,
  plh.start_measure,
  plh.end_measure,
  plh.side,
  plh.created_by,
  plh.modified_by,
  plh.link_id,
  plh.geometry,
  (plh.end_measure - plh.start_measure) as length,
  plh.start_calibration_point,
  plh.end_calibration_point,
  plh.orig_start_calibration_point,
  plh.orig_end_calibration_point,
  plh.status,
  plh.administrative_class,
  plh.link_source,
  plh.roadway_id,
  plh.linear_location_id,
  plh.ely,
  plh.reversed,
  plh.connected_link_id,
  CASE
    WHEN status = ${RoadAddressChangeType.NotHandled.value} THEN NULL
    WHEN status IN (${RoadAddressChangeType.Termination.value}, ${RoadAddress-
ChangeType.Unchanged.value}) THEN roadway.START_DATE
    ELSE prj.start_date END as start_date,
  CASE WHEN status = ${RoadAddressChangeType.Termination.value} THEN
prj.start_date - 1 ELSE NULL END as end_date,
  plh.adjusted_timestamp,
  CASE
    WHEN rn.road_name IS NOT NULL AND rn.end_date IS NULL AND rn.valid_to IS
NULL THEN rn.road_name
    WHEN rn.road_name IS NULL AND pln.road_name IS NOT NULL THEN pln.road_name
    END AS road_name_pl,
  roadway.start_addr_m AS ra_start_addr_m,// HUOM. Saman nimiset sarakkeet, kuin
  roadway.end_addr_m AS ra_end_addr_m,  // plh-taulussa, on nimettävä
  roadway.track AS ra_track,           // Scalike apply-metodia varten
```

```
  roadway.road_number AS ra_road_number,
  roadway.road_part_number AS ra_road_part_number,
  roadway.roadway_number AS ra_roadway_number,
  plh.roadway_number AS roadway_number
  FROM project prj JOIN project_link_history plh ON (prj.id = plh.project_id)
    LEFT JOIN roadway ON (roadway.id = plh.roadway_id)
    LEFT JOIN linear_location ON (linear_location.id = plh.linear_location_id)
    LEFT JOIN road_name rn ON (rn.road_number = plh.road_number AND rn.end_date
IS NULL AND rn.valid_to IS NULL)
    LEFT JOIN project_link_name pln ON (pln.road_number = plh.road_number AND
pln.project_id = plh.project_id)
```

Slick-versio perustuu sarakkeiden järjestykseen PositionedResult-luokan nextX-metodien avulla.

Tämä lähestymistapa on altis virheille, jos kyselyn sarakkeiden järjestys muuttuu.:

```
implicit val getProjectLinkRow: GetResult[ProjectLink] = new GetResult[Project-
Link] {
  def apply(r: PositionedResult): ProjectLink = {
    val projectLinkId = r.nextLong()
    val projectId = r.nextLong()
    val trackCode = Track.apply(r.nextInt())
    val discontinuityType = Discontinuity.apply(r.nextInt())
    val roadNumber = r.nextLong()
    val roadPartNumber = r.nextLong()
    val startAddrM = r.nextLong()
    val endAddrM = r.nextLong()
    val originalStartAddrMValue = r.nextLong()
    val originalEndAddrMValue = r.nextLong()
    val startMValue = r.nextDouble()
    val endMValue = r.nextDouble()
    val sideCode = SideCode.apply(r.nextInt)
    val createdBy = r.nextStringOption()
    val modifiedBy = r.nextStringOption()
    val linkId = r.nextString()
    val geom = r.nextObjectOption()
    val length = r.nextDouble()
    val calibrationPoints = (CalibrationPointType.apply(r.nextInt), Calibration
        PointType.apply(r.nextInt))
    val originalCalibrationPointTypes = (CalibrationPointType.apply(r.nextInt),
        CalibrationPointType.apply(r.nextInt))
    val status = RoadAddressChangeType.apply(r.nextInt())
    val administrativeClass = AdministrativeClass.apply(r.nextInt())
    val source = LinkGeomSource.apply(r.nextInt())
    val roadwayId = r.nextLong()

        // Lisäämäni Debug lokitukset ennen linear_location_id:n lukemista
    logger.warn(s"""About to read linear_location_id:
    Previous value (roadwayId) = $roadwayId
    wasNull = ${r.wasNull} // Tämän ei pitäisi olla ollut tyhjä arvo
    """)

    val linearLocationId = r.nextLong() // Tämä todellisena selvityksen
        kohteena, onko arvo tyhjä ennen kuin siitä muuttuu 0L

    logger.warn(s"""After reading linear_location_id:
    linearLocationId = $linearLocationId
    wasNull = ${r.wasNull}
    """)

    val ely = r.nextLong()
```

```
    val reversed = r.nextBoolean()
    val connectedLinkId = r.nextStringOption()
    val startDate = r.nextDateOption().map(d => new DateTime(d.getTime))
    val endDate = r.nextDateOption().map(d => new DateTime(d.getTime))
    val geometryTimeStamp = r.nextLong()
    val roadName = r.nextString()
    val roadAddressStartAddrM = r.nextLongOption()
    val roadAddressEndAddrM = r.nextLongOption()
    val roadAddressTrack = r.nextIntOption().map(Track.apply)
    val roadAddressRoadNumber = r.nextLongOption()
    val roadAddressRoadPartNumber = r.nextLongOption()
    val roadwayNumber = r.nextLong()
    val projectRoadwayNumber = r.nextLong()

    val roadAddressRoadPart = if(roadAddressRoadNumber.nonEmpty && roadAddress
        RoadPartNumber.nonEmpty) { Some(RoadPart(roadAddressRoadNumber.get,
        roadAddressRoadPartNumber.get)) } else None
    ProjectLink(projectLinkId, … and the rest)
  }
}
```

Esitellyssä Slick-koodissa on myös mukana, miten selvitin NULL-arvojen käsittelyn eron Slick- ja ScalikeJDBC-kirjastojen välillä. Slick-toteutuksessa NULL-arvot muunnettiin automaattisesti primitiivityypeille oletusarvoiksi (esim. Long-tyypin NULL → 0L), kun taas ScalikeJDBC heittää virheen kohdatessaan NULL-arvon primitiivityypin kohdalla. Tämän selvittämiseksi lisäsin debug-lokitukset ennen ja jälkeen linearLocationId-kentän lukemista, jolloin voitiin todeta, että Slick käytti wasNull()-metodia merkitsemään NULL-arvojen tunnistamisen, mutta palautti silti arvon 0L.

ScalikeJDBC-toteutuksessa tämä ongelma ratkaistiin käyttämällä rs.longOpt-metodia, joka palauttaa Option[Long]-tyypin ja käsittelee NULL-arvot automaattisesti Noneksi, joka voidaan sitten muuntaa oletusarvoksi getOrElse-metodilla.

ScalikeJDBC-versio käyttää sarakkeiden nimiä WrappedResultSet-luokan metodien avulla, mikä tekee koodista luettavampaa ja vähemmän virhealtista.:

```
object ProjectLink extends SQLSyntaxSupport[ProjectLink] {
  override val tableName = "project_link"

  def apply(rs: WrappedResultSet): ProjectLink = {
    // Calculate length of road address to avoid multiple calls to the database
    val roadAddressStartAddrM = rs.longOpt("ra_start_addr_m")
    val roadAddressEndAddrM   = rs.longOpt("ra_end_addr_m")
    val roadAddressLength = roadAddressEndAddrM.map { endAddr =>
      endAddr - roadAddressStartAddrM.getOrElse(0L)
    }

    new ProjectLink(
      id            = rs.long("id"),
      roadPart      = RoadPart(
        roadNumber  = rs.long("road_number"),
        partNumber  = rs.long("road_part_number")),
      track         = Track(rs.int("track")),
      discontinuity = Discontinuity(rs.int("discontinuity_type")),
      addrMRange    = AddrMRange(
```

```
        start       = rs.long("start_addr_m"),
        end         = rs.long("end_addr_m")
      ),
      originalAddrMRange = AddrMRange(
        start       = rs.long("original_start_addr_m"),
        end         = rs.long("original_end_addr_m")
      ),
      startDate     = rs.jodaDateTimeOpt("start_date"),
      endDate       = rs.jodaDateTimeOpt("end_date"),
      createdBy     = rs.stringOpt("created_by"),
      linkId        = rs.string("link_id"),
      startMValue   = rs.double("start_measure"),
      endMValue     = rs.double("end_measure"),
      sideCode      = SideCode(rs.intOpt("side").getOrElse(0)),
      calibrationPointTypes = (
        CalibrationPointType(rs.int("start_calibration_point")),
        CalibrationPointType(rs.int("end_calibration_point"))
      ),
      originalCalibrationPointTypes = (
        CalibrationPointType(rs.int("orig_start_calibration_point")),
        CalibrationPointType(rs.int("orig_end_calibration_point"))
      ),
      geometry              = GeometryDbUtils.loadJGeometryToGeome
        try(rs.anyOpt("geometry")),
      projectId             = rs.long("project_id"),
      status                = RoadAddressChangeType(rs.int("status")),
      administrativeClass   = AdministrativeClass(rs.int("administerative_
        class")),
      linkGeomSource        = LinkGeomSource(rs.int("link_source")),
      geometryLength        = rs.double("length"),
      roadwayId             = rs.longOpt("roadway_id").getOrElse(0L),
      linearLocationId      = rs.longOpt("linear_location_id").getOrElse(0L),
      ely                   = rs.long("ely"),
      reversed              = rs.boolean("reversed"),
      connectedLinkId       = rs.stringOpt("connected_link_id"),
      linkGeometryTimeStamp = rs.long("adjusted_timestamp"),
      roadwayNumber = {
        val projectRoadwayNumber = rs.longOpt("roadway_number").getOrElse(0L)
        val roadwayNumber        = rs.longOpt("ra_roadway_number").getOrElse(0L)
        if (projectRoadwayNumber == 0 || projectRoadwayNumber == NewIdValue)
          roadwayNumber
        else
          projectRoadwayNumber
      },
      roadName              = rs.stringOpt("road_name_pl"),
      roadAddressLength     = roadAddressLength,
      roadAddressStartAddrM = roadAddressStartAddrM,
      roadAddressEndAddrM   = roadAddressEndAddrM,
      roadAddressTrack      = rs.intOpt("track").map(Track.apply),
      roadAddressRoadPart   = (rs.longOpt("ra_road_number"), rs.lon
        gOpt("ra_road_part_number")) match {
        case (Some(roadNumber), Some(roadPartNumber)) => Some(RoadPart(
              roadNumber, roadPartNumber))
        case _ => None
      }
    )
  }
}
```