Tampereen ammattikorkeakoulu

Oscar Cloud -järjestelmän tietokantojen hallinnan optimointi

Teemu Salminen

OPINNÄYTETYÖ
Joulukuu 2022

Tietojenkäsittely
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Ohjelmistotuotanto

SALMINEN, TEEMU:
Oscar Cloud -järjestelmän tietokantojen hallinnan optimointi

Opinnäytetyö 54 sivua, joista liitteitä 6 sivua
Joulukuu 2022

Opinnäytetyö tehtiin yhdessä toimeksiantaja Oscar Software Oy:n kanssa. Opinnäytetyön tavoitteena oli tutkia ja kehittää toimeksiantajan Oscar Cloud -järjestelmän tietokantojen hallintaa. Oscar Cloud on Java-pohjainen Spring-ohjelmistokehyksellä toteutettu ohjelmisto.

Opinnäytetyön toimeksiantaja tarjoaa asiakkailleen laajan kirjon toiminnanohjaukseen liittyviä järjestelmiä. Näistä järjestelmistä suuri määrä käyttää Oscar Cloudia taustalla. Nykyisessä Oscar Cloudin tietokantojen hallinnan toteutuksessa tietokannat määritellään manuaalisesti ja niitä voi olla rajallinen määrä kutakin instanssia kohti. Ongelma aiheuttaa toimeksiantajalle suurehkon määrän Oscar Cloudin ylläpitotöitä ja altistaa palvelun katkoksille.

Opinnäytetyössä käsitellään yksityiskohtaisesti Oscar Cloudin nykyinen toiminta tietokantojen hallinnassa. Opinnäytetyön tuloksena syntyi kaksi eri ratkaisua työn ongelmaan. Ensimmäisessä ratkaisussa muokataan Oscar Cloudin nykyistä tietokantojen hallintaa siten, että tietokantoja pidetään ohjelmistossa toiminnassa hallitusti niiden käytön mukaan. Jälkimmäinen ratkaisu refaktoroi Oscar Cloudin tietokantojen hallintaa poistamalla nykyisen toteutuksen kokonaan ja konfiguroimalla Hibernate-ohjelmistokehyksen ominaisuuksia Oscar Cloudin käyttöön. Opinnäytetyön yhteydessä kehitettiin oma testiohjelma, jolla opinnäytetyössä syntyneitä ratkaisuja testattiin nykyiseen tietokantojen hallintaan verraten.

Opinnäytetyön tuloksena syntyneitä ratkaisuja aletaan toimeksiantajan puolesta kehittämään tuotantokäyttöön. Oscar Cloudissa on monia ominaisuuksia, jotka täytyy kehittää yhteensopiviksi opinnäytetyön ratkaisujen kanssa ennen kuin tuotantokäyttöä voidaan harkita. Opinnäytetyön tulokset loivat vahvan pohjan toimeksiantajalle, kun työn tuloksista aletaan kehittämään tuotantoon vietävää ratkaisua.

Asiasanat: java, spring, hibernate

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Software Development

SALMINEN, TEEMU:
Optimization of Oscar Cloud Software's Database Management

Bachelor's thesis 54 pages, appendices 6 pages
December 2022

The client of this thesis, Oscar Software, offers a wide variety of programs related to enterprise resource planning for its clients. Many of these programs use Oscar Cloud in the background. Oscar Cloud is Java-based software built on top of Spring Framework. In Oscar Cloud's current database management, the databases are configured manually and there can be a limited number of databases configured for a single instance. This problem exposes the client for a huge amount of maintenance work related to Oscar Cloud, and it also exposes the service for downtimes.

The thesis covers a detailed explanation how the Oscar Cloud's current database management works. The result of the thesis was two different solutions to solve the problem. The first solution focuses on modifying the current Oscar Cloud's database management so that the databases are kept in the program by their demand. The second solution refactors the Oscar Cloud's database management thoroughly and focuses on using Hibernate Framework's features to solve the problem.

The results of the thesis are going to be further developed by the client. There are many issues that need to be resolved in the Oscar Cloud that relied on the current database management that need to be developed compatible with the results.

Key words: java, spring, hibernate

SISÄLLYS

LYHENTEET JA TERMIT

Annotaatio	Java-ohjelmointikielessä käytettyjä tunnisteita, joiden avulla on mahdollista lisätä metatietoja luokkiin, rajapintoihin, metodeihin tai kenttiin
API	Application Programming Interface eli ohjelmointirajapinta, jolla tarkoitetaan rajapintaa ohjelmien välillä, joka mahdollistaa niiden välisen kommunikaation
Bean	Spring-ohjelmistokehyksen kontekstin hallinnassa olevia ohjelman osia, jotka ovat myös Java-olioita
IoC	IoC eli Inversion of Control on ohjelmistojen suunnitteluperiaate, jossa ohjelman olioiden riippuvuudet tulevat olion ulkopuolelta
Java	Ohjelmointikieli
Multitenancy	Ohjelmiston arkkitehtuurimalli, jossa yksi ohjelmiston instanssi palvelee useampaa asiakasryhmää eli tenanttia
Pääkonteksti	IoC-konttiin viittaava opinnäytetyössä käytetty termi
REST	Representational state transfer (REST) on rajapinnan arkkitehtuurimalli
Spring	Ohjelmistojen perhe, joka pitää sisällään myös Spring-ohjelmistokehyksen
Tenant	Ohjelmistoa käyttävä käyttäjien ryhmä, esimerkiksi asiakasyritys, jolla on sama tietolähde

1 JOHDANTO

Tämä opinnäytetyö on toteutettu toimeksiantona Oscar Software Oy:lle. Oscar Software Oy on vuodesta 2005 lähtien toiminut Suomalainen yritys, joka tarjoaa pääosin pk-yrityksille toiminnanohjausjärjestelmiä.

Oscar Software Oy tarjoaa asiakkailleen laajan kirjon erilaisia järjestelmiä asiakkaiden tarpeiden mukaan. Näistä järjestelmistä useampi hyödyntää jollakin tapaa taustalla olevaa Oscar Software Oy:n Oscar Cloud -järjestelmää. Oscar Cloud on Java pohjainen Spring-ohjelmistokehyksellä toteutettu ohjelmisto. Opinnäytetyön tekohetkellä Oscar Cloud -ohjelmistoa käytti noin 150 Oscar Software Oy:n asiakasyrityksistä, mikä tarkoittaa sitä, että ohjelmisto on erittäin kriittinen sekä Oscar Software Oy:n että sen asiakkaiden liiketoiminnan kannalta.

Nykyisellä toteutuksellaan, jokaiseen Oscar Cloud -instanssiin määritellään manuaalisesti kyseistä instanssia käyttävien asiakasyrityksien tietokannat. Yhdellä Oscar Cloud -instanssilla on keskimäärin noin 30 eri tietokantaa käytössään. Tämä tarkoittaa sitä, että Oscar Cloudista on useampi instanssi samanaikaisesti päällä, ja sitä käyttävä järjestelmä on ohjattava oikeaan instanssiin asiakasyrityksen perusteella. Kun tietokantojen määrä kasvaa, niin tietokantojen määrän kasvu korreloi instanssin käyttämien resurssien kanssa. Kun instansseja on useampi ja tietokannat määritellään manuaalisesti instansseihin, syntyy tästä ylimääräisiä prosesseja toimeksiantajalle esimerkiksi Oscar Cloudin päivitysten ja hallinnan suhteen. Manuaalisesti määritellyt instanssin tietokannat altistavat Oscar Cloudin käyttäjät katkoksille, koska jos tietylle asiakkaalle tarvitsee esimerkiksi korjata tai päivittää ominaisuuksia, jotka vaativat Oscar Cloudin uudelleenkäynnistämisen, vaikuttaa uudelleenkäynnistäminen suoraan saman instanssin muihin asiakasyrityksiin.

Opinnäytetyön tavoitteena oli tutkia ja kehittää ratkaisu edellisessä kappaleessa kuvattuun ongelmaan. Toimeksiantajan tavoitteena oli päästä tilanteeseen, jossa yksi Oscar Cloud -instanssi tukisi samanaikaisesti kaikkien eri asiakasyritysten tietokantoja eikä instanssi veisi niin paljon resursseja per tietokanta kuin nykyisessä toteutuksessa.

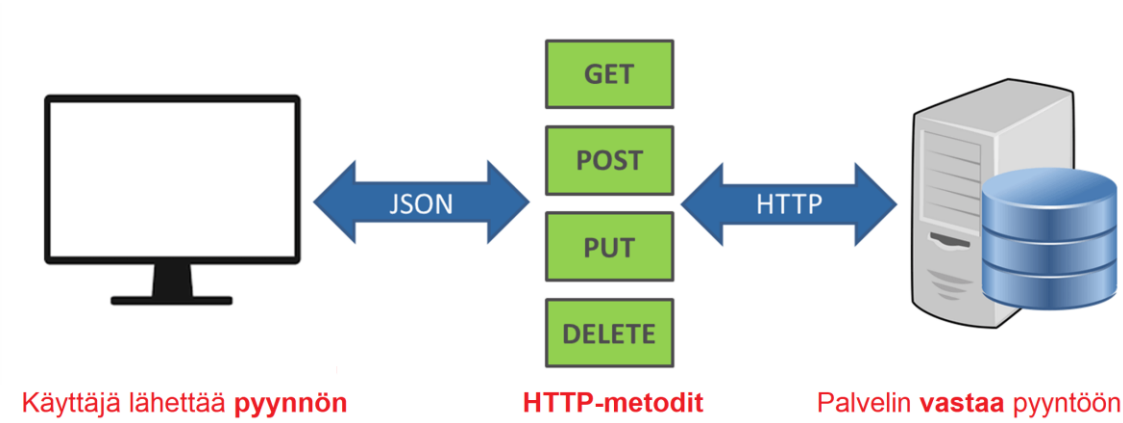
Opinnäytetyössä tutkittiin syventävästi, miten Oscar Cloud hallitsee nykyisellä toteutuksellaan yhdessä Spring-ohjelmistokehyksen kanssa tietokantojen käytön. Työn tuloksena toteutui kaksi mahdollista ratkaisua edellä kuvattuun ongelmaan. Ensimmäinen ratkaisu pohjautuu siihen, että se hallitsee nykyisen Oscar Cloudin tietokantojen hallintaa rajoittamalla tietokantojen olemassaoloa tietokantojen kysynnän mukaan. Toinen ratkaisu pohjautuu nykyisen toteutuksen refaktorointiin, eli Oscar Cloudin tietokantojen hallinnan uudelleen kirjoittamiseen. Jälkimmäisessä ratkaisussa hyödynnetään Oscar Cloudin käyttämän Hibernate-ohjelmistokehyksen ominaisuuksia.

Opinnäytetyön testiohjelmalla oli tarkoitus testata Oscar Cloudia REST-rajapinnan kautta. Testiohjelmalla tarkastellaan miten Oscar Cloudin nykyinen käytössä oleva tietokantojen hallinta eroaa opinnäytetyössä kehitettyihin ratkaisuihin nähden.

2 OSCAR CLOUDIN TEKNOLOGIAT JA TYÖKALUT

2.1 REST-rajapinta

Representational state transfer (REST) on rajapinnan arkkitehtuurimalli, joka toimii *Hypertext Transfer Protocol* (HTTP) protokollan ylitse. HTTP on verkon yli tiedon siirtoa varten kehitetty protokolla. Dataa haetaan ja muokataan rajapinnasta käyttämällä GET-, PUT-, POST- ja DELETE-pyyntöjä (Gillis 2020).



KUVIO 1. REST-rajapinnan toiminnan kuvaus asiakkaan ja palvelimen välillä (Benharosh 2018).

Kuviosta 1 näkee REST-rajapinnan toiminnan asiakkaan ja palvelimen välillä. Asiakas lähettää GET-, POST-, PUT-, tai DELETE-pyynnön palvelimelle ja palvelin vastaa asiakkaan pyyntöön.

REST-rajapinnan oleellisia ominaisuuksia on se, että se on tilaton. REST-arkkitehtuurimallin toteuttava ohjelmisto ei säilytä mitään sessiodataa. Rajapintaan tulevat pyynnöt eivät tiedä mitään toisistaan, vaan ne käsitellään itsenäisesti.

2.2 Spring ja Spring-ohjelmistokehys

Termillä Spring voidaan viitata nykyään moniin eri asioihin kontekstista riippuen. Sillä voidaan viitata Spring-ohjelmistokehykseen tai muihin Spring-projekteihin,

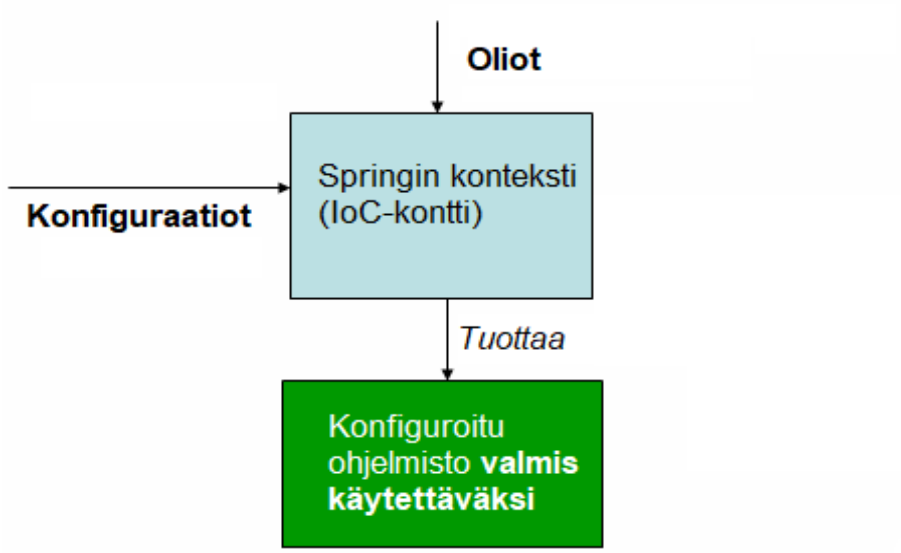
joita on ajan saatossa kehitetty Spring-ohjelmistokehyksen päälle. Yleensä termillä Spring viitataan Spring-projektien perheeseen. (Spring 2022b.)

Spring-ekosysteemiin kuuluu paljon eri ohjelmistoja, jotka on rakennettu Spring-ohjelmistokehyksen päälle. Näitä ovat esimerkiksi Spring Boot, Spring Web ja Spring Data. Kaikki edellä mainitut ohjelmistot ovat Oscar Cloudissa käytössä. Opinnäytetyössä termillä Spring viitatataan kaikkiin Spring-ekosysteemin ohjelmistoihin, mikäli ohjelmistoa ei ole erikseen mainittu termin yhteydessä.

Spring-ohjelmistokehys on Java-sovelluksien kehitykseen tarkoitettu *Inversion of Control* (IoC) -periaatteen toteuttava ohjelmistokehys. Periaate tunnetaan myös nimellä riippuvuusinjektio (dependency injection). Riippuvuusinjektio tarkoittaa prosessia, jonka mukaan olion riippuvuudet tulevat olion ulkopuolelta. (Spring 2022a.) Oscar Cloudin tapauksessa ohjelmiston olioiden riippuvuudet injektoidaan Spring-ohjelmistokehyksen IoC-kontista.

2.2.1 IoC-kontti ja beanit

IoC-kontin vastuulla on luoda, konfiguroida ja rakentaa ohjelmisto sen olioista eli beaneista. Bean on yleisesti käytetty termi Spring-maailmassa, ja sillä tarkoitetaan IoC-kontin hallinnoimaa oliota. Beanit voivat sisältää viittauksia toisiin beaneihin, ja yhdessä IoC-kontin kanssa beanit ja niiden konfiguraatiot luovat Spring-ohjelman. (Spring 2022a.)



KUVIO 2. Kuvaus Spring-ohjelmistokehyksen IoC-kontin, beanien ja konfiguraation toiminnasta (Spring 2022a).

Kuviosta 2 näkee miten Spring-ohjelmistokehyksen IoC-kontti, ohjelmiston oliot eli beanit ja konfiguraatio toimivat yhdessä. IoC-kontille ohjeistetaan ohjelmiston konfiguraatiossa, miten ohjelmiston beanit tulee luoda ja miten beaneja kuuluu hallita. Konfiguraatio on esiteltynä joko XML-tiedostoissa, Java-annotaatioina tai Java-koodina. (Spring 2022a.) IoC-konttia kutsutaan yleensä myös kontekstiksi tai pääkontekstiksi, ja luettavuuden helpottamiseksi raportissa kutsutaan sitä tästä lähtien pääkontekstiksi

2.2.2 Spring Boot

Spring Boot on Spring-ohjelmistokehyksen lisäosa. Se esimerkiksi luo automaattisesti tarvittavia konfiguraatioita Spring-ohjelmiston pystyttämisessä (Baeldung 2022d).

Oscar Cloud on rakennettu Spring Bootin päälle, ja se hyödyntää useita ominaisuuksia, jotka tulevat Spring Bootin kautta automaattisesti. Yksi ominaisuuksista on esimerkiksi profiilit, jotka käsitellään luvussa 2.2.3.

2.2.3 Profiilit

Profiilit ovat Springin ominaisuus, joka mahdollistaa Spring-ohjelmiston määrittelyn ympäristökohtaisesti. Profiileilla voidaan vaikuttaa ohjelmiston asetuksiin ja siihen, mitkä beanit luodaan ohjelmiston pääkontekstille. (Hombergs 2022.)

Oscar Cloud hyödyntää tätä ominaisuutta, kun se luo alikontekstin (ks. luku 3.6.2) Oscar Cloudia käyttävälle tenantille. Tenantti tarkoittaa ohjelmistoa käyttävää käyttäjien ryhmää, joilla on sama tietolähde. Oscar Cloudin kohdalla tenantit ovat asiakasyrityksiä. Alikontekstille injektoidaan profiilinsa mukaan esimerkiksi eri palveluluokkia, jotka voivat erota jonkin toisen profiilin palveluluokista. Tämä prosessi sekä alikonteksti on käsitelty tarkemmin luvuissa 3.5 ja 3.6.

2.3 Java Persistence API ja Hibernate

Java Persistence API (JPA) on spesifikaatio, joka määrittelee, miten persistoidaan dataa Java-sovelluksissa. JPA:n pääfokus on *Object-Relation Mapping* (ORM) -kerros. ORM tarkoittaa prosessia, jossa Java-objektit kartoitetaan tietokannan tauluiksi. Hibernate on JPA-standardin toteuttava ohjelmistokehys, ja se on yksi suosituimmista Javan kanssa käytettävistä ORM-ohjelmistokehyksistä. (Baeldung 2022c.) *Application Programming Interface* (API) tarkoittaa ohjelmointirajapintaa, jolla tarkoitetaan rajapintaa ohjelmien välillä, joka mahdollistaa ohjelmien välisen kommunikaation.

Oscar Cloud hyödyntää Hibernatea Oscar Cloudin ja tietokannan välisissä toiminnoissa. Hibernate hoitaa taustalla entiteettien persistoinnin tietokantaan.

2.4 Java-annotaatiot

Annotaatiot ovat Java-ohjelmointikielessä käytettyjä tunnisteita, joiden avulla on mahdollista lisätä metatietoja luokkiin, rajapintoihin, metodeihin tai kenttiin (Javatpoint 2022). Annotaatiot merkitään ohjelmakoodin @-symbolilla (kuva 1).

```
@Bean
public OCResourceListener ocResourceListener() {
    return new OCResourceListener();
}
```

KUVA 1. Kuvakaappaus Oscar Cloudissa käytetystä *@Bean*-annotaatiosta.

Kuvasta 1 näkee esimerkin, jossa käytetään Spring-ohjelmistokehyksen *@Bean*-annotaatiota. *@Bean*-annotaatio luo Springissä oliosta beanin, ja esimerkissä luodaan *OCResourceListener*-luokasta bean Spring-ohjelmistoon.

2.5 Rinnakkaisuus ja säikeet

Ohjelmakoodia suoritettaessa voidaan miettiä, että se ajetaan yhtenä jonona komento toisensa perään. Rinnakkaisuudella voidaan tarkoittaa ohjelmakoodin suorittamista niin sanoen rinnakkain tai samanaikaisesti. Rinnakkainen suoritus tarkoittaa, että komentojonot suoritetaan oikeasti samaan aikaan. Samanaikaisuus tarkoittaa, että komentojonot ajetaan näennäisesti samaan aikaan, eli komentojonoja on suoritettavana samanaikaisesti, mutta niitä ajetaan toisensa jälkeen.

Javassa on mahdollista suorittaa ohjelmakoodia joko rinnakkain tai samanaikaisesti. Javassa rinnakkaisuus tapahtuu säikeillä, joita on mahdollista luoda useammalla tavalla. Näitä ovat esimerkiksi uuden Thread-luokan olion luonti, Runnable- tai Callable-rajapinnan toteuttaminen. Opinnäytetyössä ei oteta kantaa, kun puhutaan rinnakkaisuudesta tai säikeistä, että tapahtuuko suorittaminen rinnakkain vain samanaikaisesti.

2.6 Postman

Postman on ohjelmointirajapintojen rakentamiseen ja käyttöön tarkoitettu työkalu. Postman yksinkertaistaa jokaista vaihetta ohjelmointirajapinnan elinkaaressa ja virtaviivaistaa yhteistyötä, jotta kehittäjät voivat rakentaa tehokkaammin parempia ohjelmointirajapintoja. (Postman n.d.)

Opinnäytetyössä Postmania käytetään testaamiseen. Postmanilla voi rakentaa helposti REST-rajapintoja varten testiympäristön, mistä voidaan tehdä pyyntöjä rajapintaan.

2.7 Opinnäytetyön tuloksien testaaminen

Opinnäytetyön tuloksia testattiin manuaalisesti Postman-työkalun avulla ja kuormitustestausta varten kehitettiin oma testiohjelma. Kuormitustestausta varten tutkittiin paljon eri työkaluja REST-rajapintojen testaamiseen. Suurin ongelma työkalun valinnassa oli, että monet työkalut olivat joko maksullisia tai niissä ei ollut tukea pyyntöjen rinnakkain suorittamiseen. Rinnakkain suorittamisella tarkoitetaan pyyntöjen yhteydessä sitä, että pyyntöjä on mahdollista suorittaa Oscar Cloudiin samanaikaisesti. Testausohjelman kehityksessä käytettiin NodeJS-ajoympäristöä, TypeScript-ohjelmointikieltä ja Axios-kirjastoa.

NodeJS on avoimen lähdekoodin ajoympäristö Javascript-ohjelmointikielelle (NodeJs n.d.). TypeScript on JavaScript-ohjelmointikielen päälle rakennettu tyypitetty versio JavaScriptistä. Axios-kirjasto on Promise-pohjainen HTTP-asiakasohjelma NodeJS- sekä selainympäristöön (Axios n.d.).

Testausohjelmalla tehdään kevyttä kuormitustestausta Oscar Cloudiin. Sillä oli tarkoitus testata, miten Oscar Cloud toimii opinnäytetyössä kehitetyillä ratkaisuilla simuloiden Oscar Cloudin käyttöä samanaikaisesti useamman eri tenantin käytössä. Testausohjelman toiminta on selitetty tarkemmin luvussa 4.5.1, ja sen lähdekoodia voi tarkastella liitteestä 2.

3 OSCAR CLOUD

Opinnäytetyön pääkohde Oscar Cloud on Springillä toteutettu Java-pohjainen ohjelmisto ja se toteuttaa luvussa 2.1 käydyn REST-rajapinta arkkitehtuurimallin. Oscar Cloud ei ole avoin rajapinta eli Oscar Cloudin loppukäyttäjät käyttävät sitä jonkin toimeksiantajan tuotteen kautta. Eli tuote käyttää toimiakseen Oscar Cloudia taustalla.

Oscar Cloudia käyttää useampi toimeksiantajan järjestelmä, ja se on toimeksiantajan yksi tärkeimpiä ohjelmistoja, koska moni järjestelmä on riippuvainen Oscar Cloudin toiminnasta. Tässä luvussa käydään yksityiskohtaisesti läpi Oscar Cloudin arkkitehtuuri ja toiminnallisuus opinnäytetyön osalta.

3.1 Oscar Cloud ja REST

Oscar Cloud toteuttaa REST-rajapinta-arkkitehtuurimallin, ja sitä käytetään mallin mukaisesti, kun ohjelmistoihin viedään tai niistä haetaan dataa. Kun Oscar Cloudiin tehdään HTTP-pyyntöjä, niihin sisällytetään pyynnön polussa tietokannan nimi, josta data halutaan hakea. Esimerkki Oscar Cloudille tehdystä HTTP-pyynnön polusta: *https://cloud_url/ocresource/rest/OCResource/tietokanta/accounts/count*.

Spring käsittelee jokaisen Oscar Cloudille tulleen pyynnön omassa säikeessään. Ilman rinnakkaisuutta Oscar Cloudia ei pystyisi käyttämään samanaikaisesti, vaan pyynnöt joutuisivat odottamaan, että edelliset pyynnöt olisi käsitelty, ennen kuin uudet pyynnöt voidaan käsitellä.

3.2 Oscar Cloudin arkkitehtuuri

Oscar Cloud rakentuu kolmesta kerroksesta, jotka ovat kontrolleri-, palvelu-, ja data access object -kerros (DAO-kerros). Kun Oscar Cloudiin tehdään pyyntö, nämä kolme kerrosta toimivat yhdessä ja muodostavat vastauksen pyyntöön.

Oscar Cloudin arkkitehtuuri ei kiteydy kuitenkaan täysin edellisen kappaleen kolmeen kerrokseen. Nämä kerrokset, jotka käsitellään luvuissa 3.2.1, 3.2.2 ja 3.2.3 käsittelevät miten Oscar Cloud toimii pääosin REST-pyyntöjen yhteydessä.

3.2.1 Kontrollerikerros

Oscar Cloudiin tulevaan HTTP-pyyntöön reagoidaan ensimmäiseksi kontrollerikerroksella. Tämä kerros kuuntelee ohjelmistoon määriteltyjä polkuja ja välittää pyynnöstä tarvittavat tiedot palvelukerrokselle. Tämän jälkeen, kun muut kerrokset ovat tehneet niiden määritellyt toiminnot, kontrollerikerros vastaa pyyntöön.

Kontrollerikerroksella olevia beaneja käyttävät kaikki ohjelmiston profiilit jaetusti, eli niitä ei konfiguroida profiileille erikseen. Kontrollerikerros hakee pyynnön polun mukana tulevasta tenantin tunnisteesta sitä vastaavan Springin alikontekstin ohjelmistosta. Haettua alikontekstia käytetään koko pyynnön loppuun suorittamisessa. Alikontekstit käsitellään luvussa 3.6.

3.2.2 Palvelukerros

Palvelukerros sisältää ohjelmiston business-logiikan. Business-logiikalla tarkoitetaan ohjelmiston varsinaista käsittelylogiikkaa. Tässä kerroksessa kontrolleri- tai DAO-kerrokselta tulevaa tai sinne menevää dataa mahdollisesti käsitellään ja manipuloidaan.

Palvelukerroksen oliot manipuloivat ja tarkistavat ohjelmiston dataa palveluluokassa määriteltyjen toimintojen mukaisesti. Olioille yleensä injektoidaan myös muita palvelukerroksien olioita, koska ohjelmistossa olevat entiteetit tarvitsevat usein muiden entiteettien tietoja datan manipulointia varten. Esimerkiksi laskuja käsittelevä palveluluokan olio tarvitsee tietoja laskuille merkityistä tileistä. Tällöin laskuja käsittelevälle palveluluokan oliolle on injektoitu tilejä käsittelevä palvelu, jonka kautta tilejä voidaan hakea tarpeen mukaan.

3.2.3 DAO-kerros

DAO-kerroksen vastuulla on datan hakeminen, lisääminen, muokkaaminen tai poistaminen tietokannasta. DAO-kerros rakentaa palvelutasolta tulevan pyynnön ja parametrien mukaan kyselyn, joka suoritetaan tietokantaan.

DAO-kerros hyödyntää luvussa 2.3 käsiteltyjä JPA:n ja Hibernaten ominaisuuksia, kun kerrokselle tulevista parametreista rakennetaan kysely tietokantaan. Kyselyä ei tarvitse kirjoittaa manuaalisesti vaan sen rakentamisessa voidaan käyttää esimerkiksi JPA:n Criteria-APIa. Criteria-API mahdollistaa kyselyn rakentamisen ohjelmallisesti, eli käytännössä Java-koodia kirjoittamalla.

3.3 Multitenancy

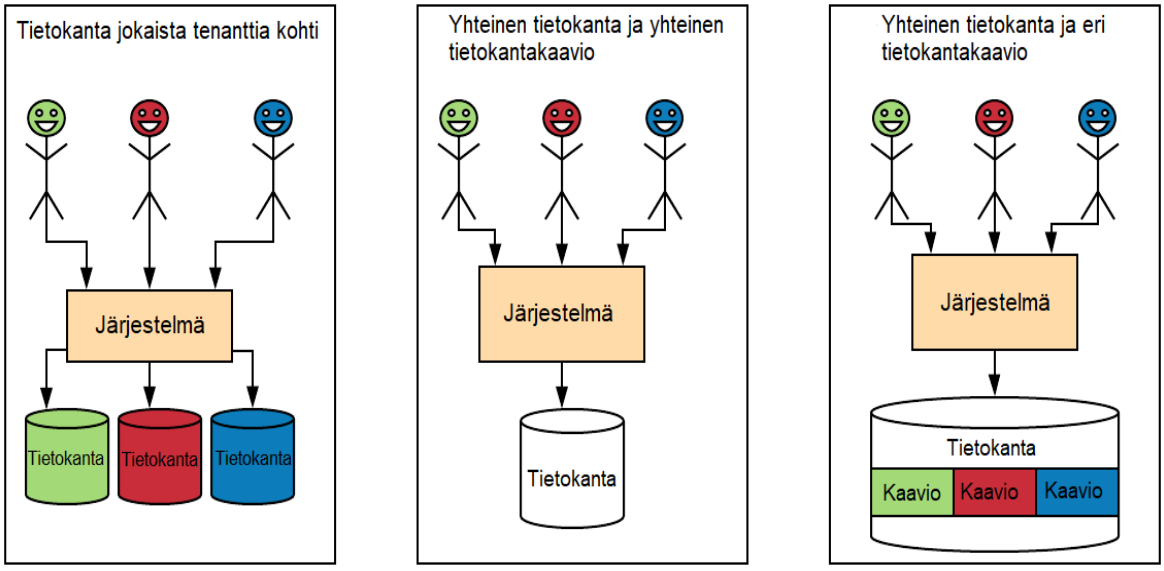
Multitenancy on ohjelmiston arkkitehtuurimalli, jossa yksi ohjelmiston instanssi palvelee useampaa asiakasryhmää eli tenanttia. Ohjelmistossa kaikki muu on jaettua erilaisin säännöin paitsi asiakkaiden data. Tenantit eivät tiedä toisistaan, ja niiden data pidetään toisistaan erillään. (Beskow 2020.)

Nykyisessä toteutuksessa Oscar Cloud käyttää yllä kuvattua arkkitehtuurimallia, mutta siinä on puutteita erityisesti resurssienhallinnan suhteen. Tällä toteutuksella Oscar Cloudia on raskasta käyttää ja ylläpitää, kun asiakkaiden määrä kasvaa. Tästä johtuen instansseja on käytössä useampi ja kuhunkin instanssiin on manuaalisesti määritelty instanssiin kuuluvat tenantit.

3.4 Multitenancyn toteutusmallit

Multitenancy-arkkitehtuurimalli voidaan toteuttaa järjestelmään ainakin kolmella eri tavalla (kuvio 3). Nämä ovat

- tietokanta jokaista tenanttia kohti,
- yhteinen tietokanta ja yhteinen tietokantakaavio sekä
- yhteinen tietokanta ja eri tietokantakaavio.



KUVIO 3. Multitenancyn toteutusmallit (Das 2019).

Oscar Cloudissa käytetään tietokanta jokaista tenanttia kohti -mallia. Käytettyä mallia ei lähdetä vaihtamaan opinnäytetyössä, vaan tietokannat pysyvät koskemattomina.

3.5 Oscar Cloudin profiilit

Oscar Cloud suunniteltiin alun perin siten, että samaa instanssia on tarkoitus käyttää useamman eri tuotteen kanssa. Oscar Cloudiin on luotu eri konfiguraatiot ja mahdollisesti eriävää ohjelmalogiikkaa Oscar Software Oy:n eri tuotteita varten. Näitä konfiguraatioita ja toimintoja käytetään Spring-profiilien kautta, ja jokaiselle Oscar Cloud -instanssia käyttävälle tietokannalle määritellään sen profiili.

Yhtä Oscar Cloud -instanssia voidaan konfiguroida siten, että samalla instanssilla voi olla ajon aikana käytössä eri profiilien beaneja. Toisin sanoen yksi Oscar Cloud -instanssi tukee useampaa profiilia ajon aikana.

3.6 Oscar Cloudin alustusprosessi

Oscar Cloudin alustusprosessissa eli käynnistyksessä käy ilmi, miten se käsittelee ja hallitsee kyseisen instanssin tietokannat. Alustuksessa tapahtuu paljon muitakin ohjelmistoon sekä Springiin liittyviä asioita, mutta ne eivät ole relevantteja opinnäytetyön kannalta.

Oscar Cloudin nykyinen tietokantojen hallinta on opinnäytetyössä tutkittava ja kehitettävä asia. Tämä luku käy vaiheittain läpi, miten Oscar Cloudin alustusprosessissa käsitellään tietokannat, sekä miten tietokantoja hallitaan alustuksen jälkeen.

3.6.1 Tietokantojen konfiguraatio

Oscar Cloudin käynnistyessä ensimmäinen oleellinen asia on konfiguraatio. Jokainen Oscar Cloud -instanssi hakee sille määritellyn konfiguraatiotiedoston, joka sijaitsee keskitetyssä paikassa, erillisellä palvelimella.

Konfiguraatiotiedosto sisältää Springiin, palvelimeen, erillisiin kirjastoihin sekä Oscar Cloudiin liittyviä alustus- ja käyttöparametreja. Oscar Cloud sisältää *BackendConfiguration*-luokan, joka on annotoitu Springin *@ConfigurationProperties*-annotaatiolla (kuva 2). Tämä annotaatio sitoo sen arvon mukaan konfiguraatiosta tulevan parametrin arvon tähän luokkaan.

```
@Configuration
@ConfigurationProperties("backend-list")
public class BackendConfiguration {

    private List<OscarBackend> oscarBackends = new ArrayList<>();

    public List<OscarBackend> getOscarBackends() { return oscarBackends; }

    public void setOscarBackends(List<OscarBackend> oscarBackends) { this.oscarBackends = oscarBackends; }

}
```

KUVA 2. Kuvakaappaus BackendConfiguration-luokasta.

Kuvassa 2 näkyvä BackendConfiguration-luokan oscarBackends-kenttä sisältää alustuksen jälkeen tietokantojen konfiguraatiot listana. Yksi tietue listassa vastaa OscarBackend-luokkaa, joka pitää sisällään tietokannan tiedot, kuten esimerkiksi nimen, yhteystiedot sekä autentikointiin liittyvät tiedot. Liitteessä 1 on esiteltynä OscarBackend-luokka, josta voi tarkastella, mitä kenttiä eli asetuksia yksi tietue sisältää.

3.6.2 Pääkonteksti, alikontekstit ja kontekstihierarkia

Oscar Cloudin käynnistyessä, kun konfiguraatiotiedosto on haettu, aletaan tiedostosta luomaan pääkontekstille eli IoC-kontille alikonteksteja. Yhtä alikontekstia kuvaa *OscarAnnotationConfigApplicationContext*-luokka, joka periytyy Springin *AnnotationConfigApplicationContext*-luokasta.

Springin *AnnotationConfigApplicationContext*-luokka on itsenäinen kontekstinsa (Spring n.d.-a). Itsenäinen konteksti tarkoittaa sitä, että kontekstille määritellään omat beanit eivätkä nämä beanit ole muilla konteksteilla käytössä. Oscar Cloudin käyttötapauksessa nämä ovat alikonteksteja, jotka käyttävät *@Component*- ja *@Configuration*-annotaatioilla varustettuja luokkia konfiguraationaan.

Tilannetta, jossa Spring-ohjelmistot sisältävät pääkontekstin sekä alikonteksteja, kutsutaan kontekstihierarkiaksi. Kontekstihierarkia tarkoittaa Spring-ohjelmistokehyksessä sitä, että pääkontekstin alapuolella olevat itsenäiset alikontekstit voivat käyttää pääkontekstin beaneja. Alikontekstit voivat myös ylikirjoittaa pääkontekstin konfiguraation. (Baeldung 2022a.)

3.6.3 Alikontekstien alustaminen

Oscar Cloudissa alikontekstien luomisesta vastaa *ContextLoaderRunnable*-luokka. *ContextLoaderRunnable*-luokka toteuttaa *Runnable*-rajapinnan, eli *ContextLoaderRunnable*-luokasta luotuja olioita voidaan suorittaa omassa säikeessään. *Runnable*-rajapinnan toteutetun run-metodin ohjelmakoodi voidaan siis suorittaa rinnakkain muun ohjelmakoodin kanssa.

Oscar Cloudin käynnistyessä *OCResource*-luokka kutsuu *ContextLoaderRunnable*-luokan run-metodia. Metodi käy läpi konfiguraatiotiedoston *oscarBackends*-kentän, joka sisälsi listan kyseisen Oscar Cloud-instanssin tietokantojen tiedoista. Kaikille tietueille luodaan ja konfiguroidaan oma luvussa 3.5.2 käsitelty *OscarAnnotationConfigApplicationContext*-luokka eli alikonteksti. Kun kaikki alikontekstit on luotu, Oscar Cloud on valmis käytettäväksi.

3.6.4 Alikontekstit Oscar Cloudin käytössä

Kun Oscar Cloudille tulee pyyntö, luvun 3.2.1 kontrollerikerroksen olio ottaa pyynnön polusta talteen tietokannan nimen. Tietokannan nimellä haetaan pääkontekstin beanin kentältä tietokannan nimeä vastaava alikonteksti. Alikonteksti hakee sen omista beaneistaan pyynnössä käytettävän palvelukerroksen luokan olion, jonka jälkeen pyynnön käsitteleminen jatkuu DAO-kerrokselle asti ja niin edelleen.

Oscar Cloud -instanssille luodut alikontekstit ovat olemassa niin kauan kuin instanssikin. Alikonteksteja ei siis missään Oscar Cloudin ajon vaiheessa poisteta instanssista. Alikontekstit ovat paljon resursseja, kuten muistia, tarvitsevia olioita.

3.7 Oscar Cloudista poistetut teknologiat

Opinnäytetyötä varten Oscar Cloudista otettiin pois käytöstä kaksi eri työkalua. Ensimmäiseksi pois käytöstä otettiin Flyway. Flyway on tietokantojen migraatioon tarkoitettu avoimen lähdekoodin työkalu (Redgate n.d.). Flyway poistettiin käytöstä, koska sen käyttäminen tuo merkittävän viiveen tietokantojen alustuksessa koska Oscar Cloud suorittaa tietokantojen migraation alikontekstin alustuksen yhteydessä. Flywayn käyttöönotto tarvitsee muutenkin toteuttaa uudestaan Oscar Cloudiin, kun opinnäytetyön tuloksia aletaan soveltamaan tuotantokäyttöön.

Toinen käytöstä poistettu työkalu on EHCache. EHCache on avoimen lähdekoodin standardipohjainen hajautettu välimuisti (EHCache 2022). EHCache

poistettiin käytöstä, koska opinnäytetyön tavoitteena on, että yksi Oscar Cloud -instanssi voisi käyttää kaikkia tietokantoja. Mikäli instansseja on useampi, ne käyttäisivät joko instanssin omaa tai instanssien yhtenäistä välimuistia, joihin tietoja varastoidaan tietokantakohtaisesti. Se miten tämä saataisiin toimimaan luotettavasti, on ongelma, joka on opinnäytetyön tutkimusalueen ulkopuolella.

4 TIETOKANTOJEN HALLINNAN OPTIMOINTI ALIKONTEKSTIN ELINKAAREN HALLINNALLA

Tässä luvussa käydään läpi opinnäytetyön ongelmaan toteutetuista ratkaisuista ensimmäinen. Tämä on toinen mahdollisista tuotantoon vietävistä kehitystöistä.

Ratkaisu pohjautuu uuteen *ContextManager*-luokkaan, jonka tarkoitus on luoda ja tuhota alikonteksteja Oscar Cloudille tulevien pyyntöjen mukaan. Alikonteksteille kehitettiin konfiguraatiotiedoston kautta hallittava aikaikkuna, joka määrittelee, kuinka kauan alikonteksti pysyy ohjelmiston muistissa.

4.1 Korkean tason kuvaus ratkaisun eri vaiheista

Ratkaisussa muutettiin Oscar Cloudin tietokantojen hallintaa siten, että kun Oscar Cloud käynnistyy, siihen ei alusteta käynnistyksen aikana alikonteksteja, jotka pitävät sisällään tietokannan tiedot yhtenä osana. Alikonteksteille lisättiin uusi kenttä, joka toimii alikontekstin elinkaaren ajastimena (ks. luku 4.3.3). Alikontekstin elinkaarella tarkoitetaan konfiguraatiosta sille määriteltyä aikaikkunaa, minkä aikana se on pääkontekstin hallinnassa. Jos aikaikkuna umpeutuu, alikonteksti ja kaikki siihen liittyvät beanit tuhotaan muistista eikä sitä ole enää olemassa.

Kun Oscar Cloudiin tehdään pyyntö, pyyntöä varten luodaan sen polusta napatusta tietokannan nimestä sitä vastaava alikonteksti. Tilanteessa, missä alikonteksti on jo luotu ja Oscar Cloudiin tehdään pyyntöjä, jotka käyttävät kyseistä alikontekstia, alikonteksti käsittelee pyynnöt ja alikontekstin elinkaaren ajastin resetoidaan eli alikonteksti pidetään elossa. Oscar Cloud siis luo, säilyttää ja tuhoaa alikonteksteja niiden kysynnän ja käytön mukaan.

4.2 Tietokantojen konfiguraation yhtenäistäminen

Luvussa 3.6.1 läpi käydystä Oscar Cloudin konfiguraatiosta täytyi yhdistää tietokantojen konfiguraatiot. Kun yhden Oscar Cloud -instanssin tulee tukea kaikkia

mahdollisia tietokantoja, tehtiin instansseja varten yhteinen konfiguraatiotiedosto, josta kaikki instanssit hakevat kaikkien tietokantojen tiedot.

Tämä onnistui kopioimalla konfiguraatiotiedostoista tietokantojen tiedot omaan konfiguraatiotiedostoon ja lisäämällä uuden tiedoston nimi Springin *Bootstrap.yml*-nimiseen konfiguraatiotiedostoon. Itse ohjelmakoodiin ei tarvinnut tehdä tämän osalta muutoksia. Spring hoitaa taustalla konfiguraation hakemisen ja kartoittamisen.

4.3 Alikontekstien hallinta

Oscar Cloudista poistettiin kaikki ohjelmalogiikka liittyen alikontekstien alustamiseen käynnistyksen yhteydessä. Alikontekstien alustuksesta vastasi alun perin *ContextLoaderRunnable*-luokka.

ContextLoaderRunnable-luokan ohjelmalogiikka muuttui siten, että kun luokan oliolta kysytään alikontekstia, olio joko luo, resetoi alikontekstin elinkaaren ja palauttaa tai palauttaa kontekstin suoraan riippuen alikontekstin tilasta. Muutoksien yhteydessä luokka nimettiin *ContextManager*-luokaksi.

4.3.1 ContextManager-luokka

Kun Oscar Cloudissa halutaan käyttää alikontekstia, kutsutaan silloin *ContextManager*-luokan *getContext*-metodia (kuva 3). Kyseinen metodi palauttaa alikontekstin olion suoraan, mikäli se on jo luotuna ohjelmistossa. Jos alikontekstia ei ole luotuna, niin *getContext*-metodi laittaa alikontekstin alustumaan ja palauttaa sen, kun alustus on valmis.

```
public OscarAnnotationConfigApplicationContext getContext(String backendName)
    throws BackendNotFoundException, ExecutionException, InterruptedException {

    if (contexts.containsKey(backendName) && contexts.get(backendName) != null) {
        OscarAnnotationConfigApplicationContext foundContext = contexts.get(backendName);

        logger.info("Resetting context timer for " + foundContext.getOscarBackend().getBackendName());
        resetContextDestroyFuture(foundContext);

        return foundContext;
    }

    List<OscarBackend> oscarBackends = backendConfiguration.getOscarBackends();
    Optional<OscarBackend> oscarBackendOptional = oscarBackends
        .stream()
        .filter(backend -> backend.getBackendName().contentEquals(backendName))
        .findAny();

    if (oscarBackendOptional.isPresent()) {
        OscarBackend oscarBackend = oscarBackendOptional.get();

        if (loadingContexts.containsKey(oscarBackend.getBackendName())) {
            logger.info("Waiting for context for Oscar backend {} to be loaded.");
            return loadingContexts.get(oscarBackend.getBackendName()).get();
        } else {
            startContextInitializationSync(oscarBackend);
            return loadingContexts.get(oscarBackend.getBackendName()).get();
        }
    } else {
        logger.error("Oscar backend {} not found / configured.", backendName);
        throw new BackendNotFoundException();
    }
}
```

KUVA 3. Kuvakaappaus *ContextManager*-luokan *getContext*-metodista.

Kuvassa 3 näkyy *getContext*-metodin toiminta, josta varsinkin *logger*-olion kutsuja seuraamalla saa käsityksen mitä metodi tekee missäkin kohtaa. *ContextManager*-luokan tueksi kehitettiin uusi *SingleContextLoaderCallable*-luokka, joka vastaa yksittäisen alikontekstin alustamisesta ja tuhoamisesta.

4.3.2 SingleContextLoaderCallable-luokka

SingleContextLoaderCallable-luokka on *Callable*-rajapinnan toteuttava luokka. *Callable*-rajapinnan toteuttamisella voidaan suorittaa luokan *call*-metodi omassa säikeessään ja odottaa metodin palautusarvoa. *SingleContextLoaderCallable*-luokka luo ja palauttaa *call*-metodissaan uuden *OscarAnnotationConfigApplicationContext*-luokan olion, joka vastaa yhtä alikontekstia (kuva 4).

```java
@Override
public OscarAnnotationConfigApplicationContext call() {
    logger.info("Starting to load context for Oscar backend " + oscarBackend.getBackendName() + ".");

    OscarAnnotationConfigApplicationContext context = new OscarAnnotationConfigApplicationContext();
    context.setOscarBackend(oscarBackend);
    context.setMethodAccessRolesMap(convert(oscarBackend.getMethodAccessRoles()));
    context.setParent(this.rootContext);
    context.register(JpaConfiguration.class);
    context.getEnvironment().setActiveProfiles(oscarBackend.getBackendType());
    context.refresh();

    ContextManager contextManager = rootContext.getBean(ContextManager.class);
    final int upkeepTime = oscarBackend.getUpkeepTime() != null ?
        oscarBackend.getUpkeepTime() : contextManager.DEFAULT_CONTEXT_UPKEEP_TIME;

    context.setContextDestroyFuture(
        this.contextManager.getContextDestroyExecutorService().schedule(
            rootContext.getBean(ContextDestroyCallable.class), upkeepTime, TimeUnit.MINUTES
        )
    );

    this.contextManager.getContexts().put(oscarBackend.getBackendName(), context);
    this.contextManager.getLoadingContexts().remove(oscarBackend.getBackendName());

    logger.info("Finished loading context for Oscar backend " + oscarBackend.getBackendName() + ".");
    return context;
}
```

KUVA 4. Kuvakaappaus *SingleContextLoaderCallable*-luokan *call*-metodista.

Kuvasta 4 näkee uuden alikontekstin luomisprosessin. *ContextManager*-luokka käyttää kuvan metodia, kun se laittaa uuden alikontekstin alustumaan. Kun *ContextManager*-luokka hakee *SingleContextLoaderCallable*-luokan beanin, antaa *ContextManager*-luokka beanin haun yhteydessä *OscarBackend*-luokan, joka alustetaan *SingleContextLoaderCallable*-luokan oscarBackend-kenttään. Kuvassa näkyy, kuinka *oscarBackend*-kentän oliosta käytetään tietoja alikontekstin alustuksessa.

SingleContextLoaderCallable-luokka on prototyyppinen beani. Prototyyppinen beani tarkoittaa sitä, että aina kun *SingleContextLoaderCallable*-luokan beani haetaan pääkontekstilta, luodaan siitä aina uusi olio.

Kuvan 4 metodista näkee myös, että alikontekstin luonnin jälkeen metodi muokkaa *ContextManager*-luokan *contexts*-kenttää lisäämällä juuri luodun

alikontekstin sinne. *Contexts*-kenttä pitää sisällään kaikki kyseisen Oscar Cloud -instanssin elossa olevat alikontekstit.

4.3.3 Alikontekstin elinkaari

OscarBackend-luokalle lisättiin uusi kenttä *upkeepTimeInMinutes*, jolle alustetaan arvo konfiguraatiotiedoston kautta. Kenttä kertoo, kuinka kauan *OscarBackend*-luokkaa vastaava alikonteksti pysyy elossa minuuteissa sen jälkeen, kun sille ei ole tullut yhtäkään pyyntöä. Jokaisen pyynnön yhteydessä *ContextManager*-luokka resetoi kentän arvon konfiguraatiotiedostossa määriteltyyn alkuarvoon.

Alikontekstin tuhoaminen tapahtuu *ContextDestroyCallable*-luokan avulla. Tämä luokka toteuttaa *Callable*-rajapinnan ja tuhoaa luokan *call*-metodissa alikontekstin poistaen sen myös *ContextManager*-luokan *contexts*-kentästä. *OscarAnnotationConfigApplicationContext*-luokalle eli alikontekstille lisättiin uusi kenttä *contextDestroyFuture* johon alustetaan *ContextDestroyCallable*-luokan olio suoritettavaksi, jos *upKeepTimeInMinutes*-kentän arvon mukainen aikaikkuna menee umpeen.

4.4 Testikontrolleri

Oscar Cloudin kontrollerikerrokselle luotiin uusi testikontrollerina toimiva *DatabaseTestController*-luokka. Testikontrolleri sisältää kolme metodia, joilla on tarkoitus testata opinnäytetyön tuloksia (kuva 5).

```
@RequestMapping(value = "/databasetest/memory",
    method = RequestMethod.GET, produces = MediaType.TEXT_PLAIN_VALUE)
public String getCurrentMemoryUsage() {
    Runtime runtime = Runtime.getRuntime();
    final long mb = 1024 * 1024;
    long currentMemoryUsageMB = (runtime.totalMemory() - runtime.freeMemory()) / mb;
    return String.valueOf(currentMemoryUsageMB);
}

@RequestMapping(value = "/{oscarBackendName}/databasetest/context",
    method = RequestMethod.GET, produces = MediaType.TEXT_PLAIN_VALUE)
public String contextTest(@PathVariable String oscarBackendName) throws Exception {
    OscarAnnotationConfigApplicationContext context = ocresource.getContext(oscarBackendName);
    return String.format("Context for Oscar backend %s succesfully initialized.", context.getOscarBackend().getBackendName());
}

@RequestMapping(value = "/{oscarBackendName}/databasetest/currency/{id}",
    method = RequestMethod.GET, produces = MediaType.APPLICATION_JSON_VALUE)
public CurrencyDTO currencyTest(@PathVariable String oscarBackendName, @PathVariable long id) throws Exception {
    OscarAnnotationConfigApplicationContext context = ocresource.getContext(oscarBackendName);
    CurrencyService currencyService = context.getBean(CurrencyService.class);

    Currency currency = currencyService.read(id);
    if (currency == null) throw new EntityNotFoundException(Currency.class, "id", Long.toString(id));

    CurrencyDTO currencyDTO = new CurrencyDTO();
    Assembler currencyAsm = DTOAssembler.newAssembler(CurrencyDTO.class, Currency.class, registry);
    try {
        currencyAsm.assembleDto(currencyDTO, currency, converters: null, dtoBeanFactory: null);
    } catch (InspectionInvalidEntityInstanceException e) {
    }
    return currencyDTO;
}
```

KUVA 5. Kuvakaappaus *DataBaseTestController*-luokan REST-metodeista.

Kuvassa 5 näkyvissä metodeissa on kaikissa *@RequestMapping*-annotaatio. Tällä annotaatiolla määritellään esimerkiksi se, mitä polkua metodit kuuntelevat, mitä HTTP-pyyntöä ne kuuntelevat ja minkä tyyppistä tietoa ne palauttavat.

GetCurrentMemoryUsage-metodilla saadaan tieto siitä, kuinka paljon Oscar Cloud -instanssi käyttää pyynnön hetkellä muistia. Muistin määrään ei voida kuitenkaan ihan sokeasti luottaa, koska Java virtual machine (JVM) eli ympäristö, missä Java-ohjelmaa suoritetaan, kerää automaattisesti turhat oliot pois muistista. Metodin palautusarvoon saattaa siis vaikuttaa huomattavasti se, onko JVM juuri ennen pyyntöä suorittanut roskien keruuta vaiko ei. Metodilla kuitenkin saadaan suuntaa antavia tuloksia, koska sitä on tarkoitus testien yhteydessä kutsua useasti.

ContextTest-metodilla testataan alikontekstin luominen, ja luonnin jälkeen se palauttaa merkkijonon, joka kertoo, että alikonteksti on luotu. *CurrencyTest*-metodilla haetaan tietokannasta *Currency*-entiteetti metodille polussa annetun id-parametrin mukaan. *Currency*-entiteetti vastaa tietoa jostakin valuutasta. *CurrencyTest*-metodilla testataan tietokannan toimivuutta.

4.5 Alikontekstien elinkaarien hallinnan testaaminen

Alikontekstien elinkaarien toimintaa testattiin tekemällä Postman-työkalun avulla pyyntöjä *DatabaseTestController*-luokan *contextTest*-metodille. Kuvista 3 ja 4 näkee, että Oscar Cloud kirjoittaa lokitietoja muun muassa alikontekstin luonnin, tuhoamisen ja elinkaaren resetoinnin yhteydessä.

Elinkaarien toimintaa testattiin Postmanilla simuloiden seuraavia skenaarioita:

- Pyyntö tehtiin tilanteessa, jossa alikontekstia ei ole luotuna ja varmistettiin, että *ContextManager*-luokka luo alikontekstin onnistuneesti.
- Pyyntö tehtiin tilanteessa, jossa alikonteksti on alustumassa ja varmistettiin, että *ContextManager*-luokka palauttaa jo valmiiksi alustumassa olevan alikontekstin eikä ala alustamaan uutta alikontekstia rinnakkain.
- Pyyntö tehtiin tilanteessa, jossa alikonteksti on jo luotuna ohjelmistossa. Tässä varmistuttiin siitä, että jo valmiiksi olemassa olevaa alikontekstia käytetään ja sen elinkaaren aikaikkuna resetoidaan pyynnön yhteydessä.
- Pyyntö tehtiin tilanteessa, jossa aikaisemmin elossa ollut alikonteksti on tuhottu, ja varmistettiin, että *ContextManager*-luokka luo alikontekstin uudestaan.

Kaikkien skenaarioiden toiminta todettiin Oscar Cloudin lokitietoja seuraamalla sekä tarkastelemalla ohjelmakoodin muuttujien arvoja pyyntöjen yhteydessä.

Postman-työkalulla tehtyjen testien yhteydessä huomattiin myös, että kun tehdään pyyntö tilanteessa, jossa alikontekstia ei ole luotuna, niin pyynnön kesto kasvaa haitallisesti, koska alikontekstin luominen on aikaa ja resursseja vievä prosessi. Kun alikonteksti on valmiiksi luotuna, niin pyyntöjen kestot ovat todella nopeita, mitä niiden kuuluukin normaalissa käytössä olla.

4.5.1 Testiohjelman toiminta

Luvussa 2.7 pohjustetun testiohjelman toiminta on seuraavanlainen. Testiohjelmalle määritellään testiin liittyvät parametrit ennen kuin testiohjelma suoritetaan (kuva 6).

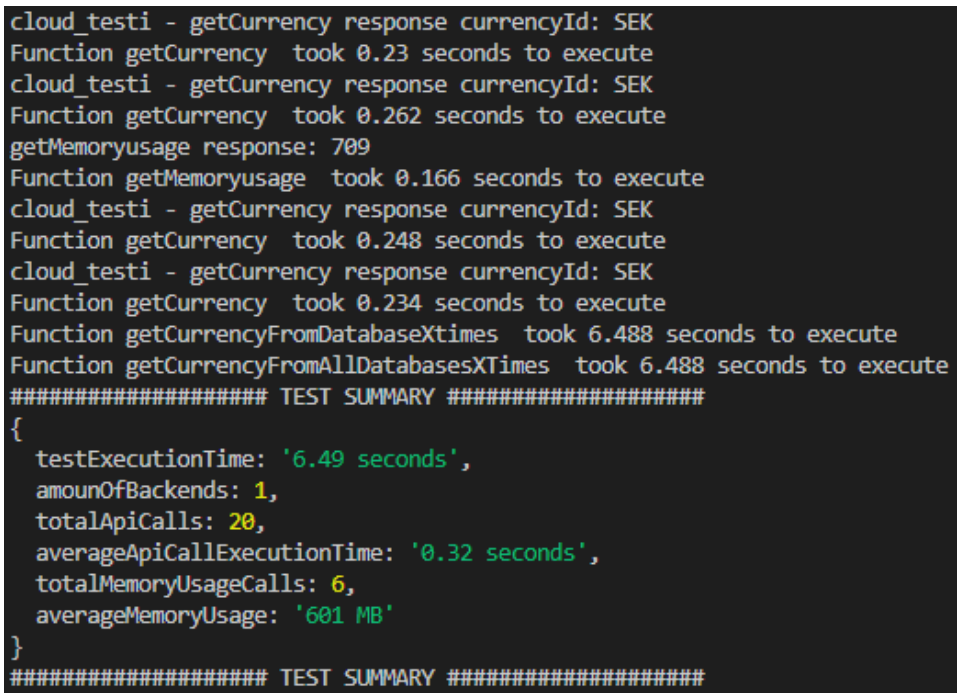
```
/* ##### TEST EXEC PARAMETERS ##### */
const DB_SET_1_ENABLED: boolean = false;
const DB_SET_2_ENABLED: boolean = false;
const DB_SET_3_ENABLED: boolean = false;
const INIT_CONTEXTS_BEFORE_TESTS: boolean = false;
const RUN_GET_CURRECY_FROM_DB_TIMES: number = 20;
/* ##### TEST EXEC PARAMETERS ##### */
```

KUVA 6. Kuvakaappaus testiohjelman parametreista.

Testiohjelman suoritus muuttuu kuvassa 6 näkyvien parametrien mukaan. Parametreilla voidaan määritellä mitkä tietokantasetit ovat käytössä ja mitkä ei, alustetaanko alikontekstit (mikäli mahdollista) ennen testiä ja kuinka monta kertaa luvun 4.4 testikontrollerin *currencyTest*-metodia kutsutaan jokaiselle testissä käytetylle tietokannalle. Mikäli alikontekstit alustetaan ennen testiajoa, ei alustavien pyyntöjen tietoja huomioida testiohjelman tuloksissa.

Kun varsinainen testi käynnistyy alustuksien jälkeen, se tekee koko testin ajon ajan sekunnin välein pyyntöjä testikontrollerin *getCurrentMemoryUsage*-metodille. Testiohjelma suorittaa pyyntöjä *currencyTest*-metodille jokaiselle testissä käytössä olevalle tietokannalle samanaikaisesti. Pyyntöjä suoritetaan testiparametreissa määritellyn arvon verran.

Testiohjelma kerää pyyntöjen kestoista, pyyntöjen määrästä ja muistin käytöstä dataa. Kun kaikki pyynnöt on suoritettu, testiohjelma tulostaa yhteenvedon kerätystä datasta (kuva 7).



KUVA 7. Kuvakaappaus testiohjelman tulostamista lokitiedoista.

Kuvasta 7 näkee testiohjelman suorittamisen aikana tulleita lokitietoja ja testin lopuksi tulostetun yhteenvedon testistä. Ohjelma tulostaa jokaisen pyynnön jälkeen lokitietoja, joista näkee, mihin tietokantaan pyyntö on tehty, kauanko pyyntö kesti ja mitä Oscar Cloud vastasi pyyntöön. Tulosteessa näkyy myös tietoja Oscar Cloudin muistin käytöstä testiohjelman suorituksen aikana.

4.5.2 Testiohjelman tulokset Oscar Cloudin nykyisellä tietokantojen hallinnalla

Jotta opinnäytetyötä varten saadaan viitearvoja, joihin voidaan verrata selvittäessä, miten Oscar Cloudin suorituskyky muuttuu opinnäytetyössä kehitettyjen ratkaisujen myötä, ajetaan testiohjelma aluksi Oscar Cloudiin, jossa ei ole opinnäytetyössä tehtyjä muutoksia.

Kuvan 6 parametrit asetetaan tätä testiajoa varten siten, että alikontekstien alustus poistetaan käytöstä, koska se ei ole mahdollista tässä Oscar Cloudin versiossa. Tässä Oscar Cloudin versiossa alikontekstit alustetaan käynnistyksen yhteydessä automaattisesti. *CurrencyTest*-metodin kutsujen määrä laitetaan

arvoon 20. Testiohjelma suoritetaan kolmeen kertaan. Jokaisella kerralla testiajossa on käytössä yksi tietokantajoukko enemmän kuin edellisessä testiajossa.

TAULUKKO 1. Testiohjelman tulokset Oscar Cloudin nykyiseen versioon

Tietokantojen lkm	Tehtyjen pyyntöjen määrä	Pyynnön kesto keskiarvo (sekuntia)	Muistin käyttö keskiarvo (megatavua)	Testin kokonaiskesto (sekuntia)
5	100	0,76	930	15,29
10	200	1,07	1 955	21,65
15	300	1,62	2 694	33,48

Taulukosta 1 nähdään testiohjelman kolmen eri ajon tulokset Oscar Cloudin nykyisen tietokantojen hallinnan kanssa. Taulukosta näkee, että varsinkin muistin käyttö kasvaa huomattavasti tietokantojen lukumäärän kasvaessa. Tämä kertoo siitä, että tietokantojen käsitteleminen alikonteksteina on paljon resursseja vievä tapa.

4.5.3 Testiohjelman tulokset alikontekstien elinkaarien hallinnan ratkaisuun

Ratkaisun tuloksien testaamista varten testiohjelma ajettiin kahdella eri setillä. Testiparametrit ovat muuten samat kuin luvun 4.5.2, mutta ensimmäinen testisetti suoritettiin siten, että alikonteksteja ei alusteta ennen varsinaista testiä eli ne alustuvat *getCurrencyTest*-pyyntöjen yhteydessä. Jälkimmäisessä testisetissä testiohjelma ajettiin siten, että alikontekstit alustettiin ennen varsinaista testiä.

TAULUKKO 2. Testiohjelman tulokset alikontekstien elinkaarien hallinnan ratkaisua käyttäessä, **kun alikonteksteja ei ole alustettu ennen testiajoa**

Tietokantojen lkm	Tehtyjen pyyntöjen määrä	Pyynnön kesto keskiarvo (sekuntia)	Muistin käyttö keskiarvo (megatavua)	Testin kokonaiskesto (sekuntia)
5	105	2,60	796	55,58
10	210	4,31	1 285	91,56
15	315	5,99	1 761	154,08

Taulukosta 2 näkee ensimmäisten testiajojen tulokset, joissa alikontekstia ei alustettu etukäteen. Luvun 4.5 Postman-testien tuloksien perusteella oli odotettavissa, että pyyntöjen kestojen keskiarvot sekä testin kokonaiskesto kasvavat, koska alikonteksteja ei alustettu ennen testiajoa, ja näin tapahtuukin. Vaikka muistin käytön tulokset ovatkin suuntaa antavia, vaikuttaisi niihin silti tulleen kevyt parannus jokaisen testiajon kohdalla.

Jotta voidaan vertailla tilannetta, missä alikontekstit ovat jo olemassa, ajetaan seuraavaksi testi täysin samoilla parametreilla, mutta alikontekstit alustetaan ennen *getCurrencyTest*-metodin pyyntöjä. (taulukko 3)

TAULUKKO 3. Testiohjelman tulokset alikontekstien elinkaarien hallinnan ratkaisuun, **kun alikontekstit alustetaan ennen testiajoa**

Tietokantojen lkm	Tehtyjen pyyntöjen määrä	Pyynnön kesto keskiarvo (sekuntia)	Muistin käyttö keskiarvo (megatavua)	Testin kokonaiskesto (sekuntia)
5	100	0,80	1 093	16,23
10	200	1,12	1 802	23,04
15	300	1,64	2 438	33,87

Taulukon 3 tuloksista huomataan, että kaikki testiajot ovat hyvin lähellä taulukon 1, eli Oscar Cloudin nykyisen toteutuksen testiajojen tuloksia. Tämä oli

odotettavissa, koska testien aikana uudella toteutuksella on sama määrä alikontekseja alustettuna kuin taulukon 1 toteutuksella.

Konkreettiset tulokset huomataan vasta testiajojen jälkeen, kun odotetaan, että alikontekstit tuhotaan instanssilta. Taulukon 3 viimeisessä testiajossa, jossa tietokantojen lukumäärä oli 15, jokaisen testissä käytetyn alikontekstin *upkeepTimeInMinutes*-parametri (luku 4.3.3) oli asetettu arvoon yksi eli yhteen minuuttiin. Kun kyseisen testiajon aikana luotujen alikontekstien automaattisen tuhoamisen jälkeen Postman-työkalulla tehtiin pyyntö testikontrollerin *getMemoryUsage*-metodille, oli muistin käyttö vähentynyt 273 megatavuun.

4.6 Ensimmäisen ratkaisun tulokset

Ratkaisu toimii Oscar Cloudin profiilien ja kontekstihierarkian kanssa tyydyttävästi. Kuten testeistä huomasi, merkittävänä ongelmana on pyyntöjen kesto tilanteissa, joissa alikontekstia ei ole luotuna. Pyyntöjen kesto tulisi näkymään haitallisesti tuotantokäytössä, koska loppukäyttäjälle se näyttäytyy Oscar Cloudia käyttävän järjestelmän hitautena.

Toinen merkittävä ongelma oli muistin käytön huomattava kasvu, kun tietokantoja otettiin enemmän käyttöön. Vaikka alikontekstien elinkaarien hallinnalla muistin käyttö tippuu huomattavasti siinä kohtaa, kun alikonteksteja tuhotaan, olisi tuotantokäytössä silti huomattavasti suurempi määrä alikonteksteja käytössä samanaikaisesti kuin testiohjelman ajon aikana. Tämä altistaa Oscar Cloudin erilaisille virhetilanteille, kuten esimerkiksi muistin loppumiseen palvelimelta.

Mikäli tätä ratkaisua alettaisiin soveltamaan tuotantokäyttöön, tarvitsisi tutkia, onko mahdollista nopeuttaa alikontekstien alustusprosessia. Muistin käyttöä on vaikea hallita tuotantokäytössä, kun yhtä Oscar Cloud -instanssia käytetään samanaikaisesti eri profiileilla kontekstihierarkian avulla. Kontekstihierarkian toteuttaminen on jo luonnostaan paljon muistia vievä prosessi Oscar Cloudin tapauksessa. Tätä koskien toimeksiantajan kanssa keskusteltiin ja todettiin, että on järkevämpää panostaa enemmän opinnäytetyön toiseen ratkaisuun, sillä siinä ei olisi ongelmia ainakaan pyyntöjen keston pitenemisen suhteen.

5 TIETOKANTOJEN HALLINNAN OPTIMOINTI HIBERNATEN MULTITENANCY-OMINAISUUKSIEN KÄYTTÖÖNOTOLLA

Toimeksiantajan kanssa ensimmäisen ratkaisun tuloksiin pohjautuneiden keskusteluiden aikana mietittiin uusia ratkaisuja opinnäytetyön ongelmaan. Keskusteluissa kävi ilmi, että Oscar Cloudilla ei ole tarvetta tukea samanaikaisesti sitä eri tavoin konfiguroivia järjestelmiä. Tämä tarkoittaa sitä, että yksi Oscar Cloud -instanssi ei palvele sekä Pro- että Ventus-asiakkaita, vaan esimerkiksi Pro-asiakkaille olisi oma instanssinsa ja Ventus-asiakkaille omansa. Oscar Pro sekä Ventus ovat Oscar Software Oy:n eri tuotteiden nimiä.

Kontekstihierarkiasta voitiin siis luopua, koska jatkossa ei ollut tarpeellista luoda jokaiselle tietokannalle erikseen alikontekstia, eikä alikontekstien beaneja tarvinnut alustaa alikontekstin profiilin mukaan. Oscar Cloudin toiminta muuttui siten, että yhdelle Oscar Cloud -instanssille annettiin profiili käynnistyksen yhteydessä. Instanssille alustetaan ja konfiguroidaan beanit instanssin profiilin mukaan. *OscarBackend*-luokkia ei käsitellä enää alikonteksteina vaan tietolähteinä. Kun kontekstihierarkiesta luovuttiin, Oscar Cloudille tarvitsi määritellä taustalle alikonekstien sijaan useampi tietolähde *OscarBackend*-luokkien mukaan. Kun tietolähteet oli määritelty, Oscar Cloud konfiguroitiin käyttämään oikeaa tietolähdettä pyynnön mukaan.

5.1 Tietolähde ja HikariCP

Tietolähteellä tarkoitetaan mitä tahansa laitosta, missä ohjelmiston data on. Oscar Cloudin tietolähteet ovat siis tietokantoja. Javassa tietolähdettä kuvataan DataSource-rajapinnalla.

HikariCP on korkean suorituskyvyn JDBC (Java Database Connectivity) -yhteysallaskirjasto. Yhteysaltaalla tarkoitetaan tietokantayhteyksien hallintaa siten, että yhteyksiä ei suljeta transaktion päätteeksi, vaan yhteyksiä voidaan käyttää uudestaan tulevissa pyynnöissä. (ZetCode 2020.)

Oscar Cloud käyttää HikariCP:tä tietokantayhteyksien hallintaan. Kun Oscar Cloudissa luodaan tietolähde, siihen käytetään HikariCP:n HikariDataSource-luokkaa, joka toteuttaa Javan *DataSource*-rajapinnan. Kuvassa 12 näkyy esimerkki, jossa Oscar Cloud luo *HikariDataSource*-tyyppisen olion.

5.2 Ratkaisun alustavat toimenpiteet

Oscar Cloudista poistettiin sen toteuttama kontekstihierarkia ja kaikki siihen liittyvät, kuten esimerkiksi *ContextLoaderRunnable*- ja *OscarAnnotationConfigApplicationContext*-luokat. Oscar Cloudin käynnistyessä ei tämän muutoksen jälkeen tapahdu mitään alikonteksteihin liittyviä alustuksia.

Oscar Cloud -instanssi ei jatkossa alusta beaneja, kuten esimerkiksi palvelukerroksen oliota, useamman profiilin mukaan. Oscar Cloud -instanssille annetaan käynnistyksen yhteydessä instanssin profiili, jolloin instanssi alustaa käynnistyksen yhteydessä tarvittavat beanit profiilin mukaan. Aikaisemmin beanit alustettiin alikontekstin profiilin mukaan alikontekstille.

5.3 TenantContext-luokka

Kontekstihierarkiasta luovuttaessa ei Oscar Cloudissa ole käytössä enää mitään tietoa, mistä voidaan hakea kontrolleritasolla alikonteksti, josta löytyy tietolähde. Tähän ongelmaan kehitettiin ratkaisu uudella *TenantContext*-luokalla (kuva 8). *TenantContext*-luokassa säilytetään tieto Oscar Cloudia käyttävästä tenantista. Tiedon avulla Oscar Cloud tietää, mitä tietolähdettä sen kuuluu kunkin pyynnön kohdalla käyttää.

```
public final class TenantContext {

    private static InheritableThreadLocal<OscarBackend> CURRENT_TENANT = new InheritableThreadLocal<>();

    public static void setCurrentTenant(OscarBackend oscarBackend) { CURRENT_TENANT.set(oscarBackend); }

    public static OscarBackend getCurrentTenant() { return CURRENT_TENANT.get(); }

    public static void clear() { CURRENT_TENANT.remove(); }
}
```

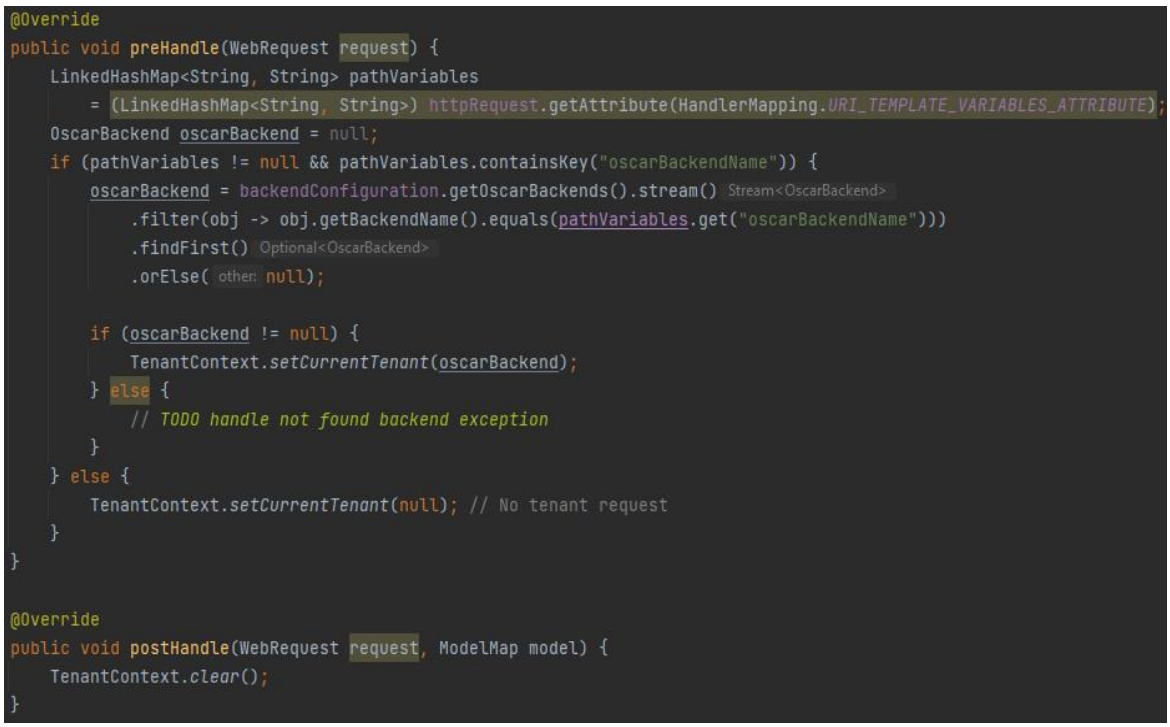
KUVA 8. Kuvakaappaus *TenantContext*-luokasta.

Kuvasta 8 näkee *TenantContext*-luokan ja sen sisältämän yhden kentän, *CURRENT_TENANT*, sekä yksinkertaisia metodeja kentän mutatoimiseen. *CURRENT_TENANT*-kentän on tarkoitus pitää sisällään pyynnön aikaisen tenantin *OscarBackend*-luokan olio.

5.4 TenantContext-luokan CURRENT_TENANT-kentän alustaminen

TenantContext-luokan *CURRENT_TENANT*-kentälle pitää alustaa Oscar Cloudille tulevan pyynnön alkuvaiheessa *OscarBackend*-luokka, joka vastaa pyynnön polussa tulevaa tietoa. Tieto tenantista täytyy säilyttää *TenantContext*-luokassa, jotta myöhemmin DAO-kerroksella osataan käyttää oikeaa tietolähdettä.

Spring sisältää *WebRequestInterceptor*-rajapinnan. Tämä rajapinta on tarkoitettu Spring-ohjelmistolle tulevien WEB-pyyntöjen sieppaamiseen (Spring n.d.-b). Kun tämä rajapinta toteutetaan ja konfiguroidaan pääkontekstiin, voidaan toteutuksen yhteyteen kehittää *CURRENT_TENANT*-kentälle *OscarBackend*-olion alustuminen Oscar Cloudille tulleen pyynnön polun mukaan.



```
@Override
public void preHandle(WebRequest request) {
    LinkedHashMap<String, String> pathVariables
        = (LinkedHashMap<String, String>) httpRequest.getAttribute(HandlerMapping.URI_TEMPLATE_VARIABLES_ATTRIBUTE);
    OscarBackend oscarBackend = null;
    if (pathVariables != null && pathVariables.containsKey("oscarBackendName")) {
        oscarBackend = backendConfiguration.getOscarBackends().stream() Stream<OscarBackend>
            .filter(obj -> obj.getBackendName().equals(pathVariables.get("oscarBackendName")))
            .findFirst() Optional<OscarBackend>
            .orElse( other: null);

        if (oscarBackend != null) {
            TenantContext.setCurrentTenant(oscarBackend);
        } else {
            // TODO handle not found backend exception
        }
    } else {
        TenantContext.setCurrentTenant(null); // No tenant request
    }
}

@Override
public void postHandle(WebRequest request, ModelMap model) {
    TenantContext.clear();
}
```

KUVA 9. Kuvakaappaus *WebRequestInterceptor*-rajapinnan toteuttavan luokan *preHandle*- ja *postHandle*-metodeista.

Kuvassa 9 näkyvä *preHandle*-metodi ajetaan ennen kuin Oscar Cloudille tullut pyyntö käsitellään ja *postHandle*-metodi ajetaan pyynnön jälkeen. *PreHandle*-metodista näkee, että se yrittää hakea Oscar Cloudille tulleen pyynnön polusta parametrin *oscarBackendName* arvoa. Jos arvo löytyy, niin silloin kuvan x luokan *backendConfiguration*-kentästä haetaan *oscarBackendName*-parametrin arvoa vastaava *OscarBackend*-luokka. Kyseinen *OscarBackend*-luokka asetetaan pyynnön ajaksi kuvan 8 *TenantContext*-luokan *CURRENT_TENANT*-kenttään. *PostHandle*-metodi tyhjentää *CURRENT_TENANT*-kentän.

5.5 Tietolähteiden luonti ja Hibernaten konfigurointi

Luvussa 5.3 mainitut *TenantContext*-luokka ja *CURRENT_TENANT*-kenttä eivät sellaisenaan tee vielä mitään. Kyseiset luokka ja kenttä ovat pohjatyötä Oscar Cloudin käyttämän Hibernate-ohjelmistokehyksen konfigurointia varten. Seuraavaksi luodaan Hibernatelle tietolähteet ja konfiguroidaan Hibernate tunnistamaan asiakas *CURRENT_TENANT*-kentän mukaan.

Hibernate tarvitsee konfiguroida seuraavin vaihein. Hibernaten täytyy osata identifioida asiakas *CURRENT_TENANT*-kentän mukaan. Hibernatea varten tarvitsee luoda kuvassa 2 näkyvän *BackendConfiguration*-luokan tietojen mukaan tietolähteet. Lopuksi Hibernate täytyy konfiguroida käyttämään sille luotuja tietolähteitä *CURRENT_TENANT*-kentän mukaan.

5.6 Tenantin identifiointi CURRENT_TENANT-kentän mukaan

Hibernate sisältää rajapinnan *CurrentTenantIdentifierResolver* joka tarvitsee toteuttaa tenantin identifiointia varten. Rajapinta sisältää kaksi metodia, joista toinen on nimeltään *resolveCurrentTenantIdentifier,* jonka palautusarvona on merkkijono. Metodin dokumentaatio kertoo, että metodilla on tarkoitus ratkaista nykyisen tenantin tunniste (Red Hat n.d.-a). Tämä metodi täytyy toteuttaa siten, että se palauttaa *CURRENT_TENANT*-kentän *OscarBackend*-luokan oliosta löytyvän tietokannan nimen (kuva 10).

```
@Override
public String resolveCurrentTenantIdentifier() {
    OscarBackend oscarBackendTenant = TenantContext.getCurrentTenant();

    if (oscarBackendTenant != null) {
        return oscarBackendTenant.getBackendName();
    }

    return "BOOTSTRAP";
}
```

KUVA 10. Kuvakaappaus Hibernaten *CurrentTenantIdentifierResolver*-rajapinnan *resolveCurrentTenantIdentifier*-metodin toteutuksesta.

Kuten kuvasta 10 näkee, tenantin tunnisteeksi eli tietokannan nimeksi haetaan *TenantContext*-luokan *CURRENT_TENANT*-kentän arvo *getCurrentTenant*-metodilla. Kentän arvosta, joka on *OscarBackend*-olio, palautetaan olion tietokannan nimi. Kuvan metodiin on myös toteutettu olemattoman arvon tarkastelu, koska Oscar Cloudissa on toimintoja, jotka eivät tarvitse pyynnön aikana tietoa, tenantista eikä sitä tällaista toimintoa käsiteltäessä ole *TenantContext*-luokassa.

5.7 Tietolähteiden luonti

Luvussa 3.6.1 käsitellyn *BackendConfiguration*-luokan *oscarBackends*-kenttä pitää sisällään konfiguraatiotiedostosta haetut tiedot tietokannoista. Näistä tiedoista tarvitsee luoda tietolähde jokaiselle tietokannalle erikseen.

```
/**
 * Holds all the data sources of this Cloud instance.
 */
private LoadingCache<String, DataSource> tenantDataSources;

@PostConstruct
private void initializeDataSourceCache() {
    this.tenantDataSources = CacheBuilder.newBuilder() CacheBuilder<Object, Object>
        .maximumSize(1000)
        .expireAfterAccess( duration: 30, TimeUnit.SECONDS)
        .removalListener((RemovalListener<String, DataSource>) removal -> {
            HikariDataSource ds = (HikariDataSource) removal.getValue();
            logger.info("Starting to close data source: {}.", ds.getPoolName());
            ds.close(); // tear down properly
            logger.info("Closed data source: {}", ds.getPoolName());
        }) CacheBuilder<String, DataSource>
        .build((CacheLoader) (oscarBackendName) -> {
                OscarBackend oscarBackend = backendConfiguration.getOscarBackends()
                    .stream() Stream<OscarBackend>
                    .filter(obj -> obj.getBackendName().equals(oscarBackendName))
                    .findFirst() Optional<OscarBackend>
                    .get();

                return createDataSourceFromOscarBackend(oscarBackend);
        });
}
```

KUVA 11. Kuvakaappaus *initializeDataSourceCache*-metodista ja tietolähteiden varaston sisältävästä *tenantDataSourceCache*-kentästä.

Kuvassa 11 näkyvä *initializeDataSourceCache*-metodi on annotoitu Javan *@PostConstruct*-annotaatiolla. Spring tukee myös tätä annotaatiota, ja tällä annotaatiolla varustetut metodit suoritetaan, kun Spring on luonut ja injektoinut ohjelmiston beanit.

InitializeDataSourceCache-metodi (koodissa *initializeDataSourceCache*) alustaa *tenantDataSourceCache*-kenttään varaston, josta voidaan hakea Oscar Cloud -instanssin tietolähteet. Kenttä hyödyntää välimuistin hallintaa kentän

LoadingCache-tietotyypin avulla minkä takia kentän alustus näyttää hieman abstraktilta. *LoadingCachen* toiminta toimii hieman samaan tapaan, miten opinnäytetyön ensimmäiseen ratkaisuun toteutettu alikontekstien elinkaaren hallinta.

Oleellisinta *initializeDataSourceCache*-metodissa on *LoadingCache*-olion alustuksessa näkyvä *CacheLoader*-luokan *load*-metodi. Metodia kutsutaan, kun *tenantDataSourceCache*-kentästä haetaan tietolähde. Metodi luo uuden *HikariDataSource*-olion tenantin *OscarBackend*-luokan mukaan kutsumalla *createDataSourceFromOscarBackend*-metodia. Lopuksi *load*-metodi palauttaa luodun tietolähteen samalla lisäten sen varastoon myöhemmin käytettäväksi.

5.8 MultiTenantConnectionProvider-rajapinta

Hibernaten *MultiTenantConnectionProvider*-rajapinta on tarkoitettu tietokantayhteyksien valintaan tenantin tunnisteen mukaan. Hibernatessa on abstrakti luokka *AbstractDataSourceBasedMultiTenantConnectionProviderImpl* joka toteuttaa tämän rajapinnan. Kyseinen luokka on tarkoitettu käytettäväksi, kun käytössä on tietolähdepohjainen toteutus.

AbstractDataSourceBasedMultiTenantConnectionProviderImpl-luokan periyttämällä tarvitsee toteuttaa luokan *selectAnyDataSource*- ja *selectDataSource*-metodit. Luokka käyttää näitä metodeja, kun se toteuttaa *MultiTenantConnectionProvider*-rajapinnan *getAnyConnection*- ja *getConnection*-metodit.

SelectDataSource-metodin toteutus on hyvin yksinkertainen. Metodi saa parametrina merkkijonon, johon tulee arvo kuvan 10 metodilta, josta näkyy, että se palauttaa tietokannan nimen, mikäli tenantti on olemassa. Tietokannan nimellä haetaan kuvan 11 *tenantDataSourceCache*-kentästä nimeä vastaava tietolähde ja palautetaan se.

5.9 Päätietolähde

Edellisen luvun *MultiTenantConnectionProvider*-rajapinnan *getAnyConnection*-metodista on dokumentoitu, että metodi mahdollistaa pääsyn taustalla olevan tietokannan metatietoihin tilanteissa, joissa tenantin tunnistetta ei ole käytettävissä ei ole käytettävissä (kuten esimerkiksi käynnistysprosessi) (Red Hat n.d.-b). Oscar Cloudiin tarvitsee siis tehdä yksi tietolähde, joka on aina saatavilla. Tämä onnistuu luomalla *DataSource*-tietotyyppinen beani (kuva 12).

```
/**
 * Oscar backend name for master data source.
 * TODO Implement better way of getting & configuring a master data source.
 */
public static final String MASTER_DATASOURCE_DB_NAME = "cloud_testi";

@Bean
public DataSource masterDataSource() {
    OscarBackend oscarBackend = backendConfiguration.getOscarBackends() List<OscarBackend>
        .stream() Stream<OscarBackend>
        .filter(obj -> obj.getBackendName().equals(MASTER_DATASOURCE_DB_NAME))
        .findFirst() Optional<OscarBackend>
        .orElseThrow(() -> new OscarBackendNotFoundException(MASTER_DATASOURCE_DB_NAME));

    HikariDataSource dataSource = new HikariDataSource();
    dataSource.setDriverClassName(oscarBackend.getJdbcDriverClassName());
    dataSource.setJdbcUrl(oscarBackend.getJdbcUrl());
    dataSource.setUsername(oscarBackend.getJdbcUsername());
    dataSource.setPassword(oscarBackend.getJdbcPassword());
    dataSource.setPoolName("masterDataSource");

    return dataSource;
}
```

KUVA 12. Kuvakaappaus *DataSource*-tietotyyppisen beanin luonnista

Kuvassa 12 näkyvä *DataSource*-tyyppinen beani, eli päätietolähde, täytyy seuraavaksi injektoida luvun 5.8 *AbstractDataSourceBasedMultiTenantConnectionProviderImpl*-luokasta periävän luokan saataville. Tämän jälkeen se voidaan laittaa saman luokan *selectAnyDataSource*-metodin palautusarvoksi.

5.10 Tietolähteiden käyttö Hibernaten kanssa

Hibernate sisältää *PersistenceContext*-rajapinnan, jolla kuvataan persistenssikontekstia. Persistenssikontekstilla tarkoitetaan ohjelmiston ja tietokannan välissä olevaa yksikköä. Se on paikka, jossa entiteetit joko tallennetaan tietokantaan tai haetaan tietokannasta. (Baeldung 2022b.)

Persistenssikontekstia käytetään EntityManager-API:n kautta. Oscar Cloudiin tarvitsee luoda beani tehdasluokasta, joka luo *EntityManager*-olioita, ja konfiguroida luokka siten, että se käyttää lukujen 5.6 ja 5.8 *CurrentTenantIdentifierResolver*- ja *MultiTenantConnectionProvider*-rajapintojen toteutuksia (kuva 13).

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory(
    DataSource dataSource,
    MultiTenantConnectionProvider multiTenantConnectionProviderImpl,
    CurrentTenantIdentifierResolver currentTenantIdentifierResolverImpl) {

    LocalContainerEntityManagerFactoryBean em = new LocalContainerEntityManagerFactoryBean();
    em.setDataSource(dataSource);
    em.setPackagesToScan(getPackagesToScan(DEFAULT_PROFILE));
    em.setMappingResources(getMappingResources(DEFAULT_PROFILE));
    em.setJpaVendorAdapter(this.jpaVendorAdapter());

    HashMap<String, Object> jpaProperties = this.jpaProperties(DEFAULT_PROFILE);
    jpaProperties.put(Environment.MULTI_TENANT, MultiTenancyStrategy.DATABASE);
    jpaProperties.put(Environment.MULTI_TENANT_CONNECTION_PROVIDER, multiTenantConnectionProviderImpl);
    jpaProperties.put(Environment.MULTI_TENANT_IDENTIFIER_RESOLVER, currentTenantIdentifierResolverImpl);

    em.setJpaPropertyMap(jpaProperties);

    return em;
}
```

KUVA 13. Kuvakaappaus *LocalContainerEntityManagerFactoryBean*-luokan eli *EntityManager*-olioiden tehdasluokan beanin luonnista.

Kuvassa 13 näkyy Oscar Cloudin EntityManager-olioiden tehdasluokan beanin luonti. Metodi saa parametreinaan edellisessä kappaleessa mainitut rajapintojen toteutukset.

Tässä kohtaa kaikki Hibernatea varten tehdyt kehitystyöt kytketään yhteen. Kun *EntityManager*-olioiden tehdasluokka on konfiguroitu, on Oscar Cloudissa käytössä Hibernaten Multitenant-ominaisuudet.

5.11 Toteutettujen Multitenancy-ominaisuuksien testaaminen

Tämän ratkaisun osalta, ei lopullisissa testauksissa käytetty ollenkaan Postman-työkalua. Ratkaisua testattiin pelkästään luvuissa 2.7 ja 4.5.1 käsitellyllä testiohjelmalla. Koska tässä ratkaisussa Oscar Cloudissa ei ole kontekstihierarkiaa eikä täten alikonteksteja, jouduttiin tekemään kevyitä muutoksia luvussa 4.4 käsiteltyyn testikontrolleriin.

Testikontrollerista poistettiin *contextTest*-metodi, koska alikonteksteja ei ole enää olemassa. Testikontrollerin *getCurrency*-metodi ei hae enää alikontekstia, jonka kautta se alun perin haki *CurrencyService*-olion. *CurrencyService*-oliota varten *DatabaseTestController*-luokkaan lisättiin luokkatasolle *currencyService*-kenttä. Koska Spring luo ja injektoi tässä ratkaisussa käynnistyksen yhteydessä profiiliin liittyvät beanit, voitiin currencyService-kenttään injektoida oikea olio Springin *@Autowired*-annotaatiolla. Testiohjelman parametreilla otettiin testiohjelman alikontekstien alustaminen pois käytöstä, koska niitä ei tässä ratkaisussa ole olemassa.

5.12 Testiohjelman tulokset

Kuten luvun 4.5.2 Oscar Cloudin nykyisen tietokannan hallinnan testeissä, testiohjelma ajettiin kolmesti samoilla tietokannoilla ja tietokantojen määrillä. Testiohjelman tulokset olivat kontekstihierarkiasta luopumisen jälkeen lupaavia. (taulukko 4)

TAULUKKO 4. Testiohjelman tulokset Hibernaten MultiTenancy-ominaisuuksien käyttöönoton jälkeen

Tietokantojen lkm	Tehtyjen pyyntöjen määrä	Pyynnön kesto keskiarvo (sekuntia)	Muistin käyttö keskiarvo (megatavua)	Testin kokonaiskesto (sekuntia)
5	100	0,80	506	16,07
10	200	1,23	536	25
15	300	1,91	522	39,04

Kun tuloksia verrataan Oscar Cloudin nykyisen tietokannan hallinnan (ks. taulukko 1) sekä alikontekstien elinkaarien hallinnan tuloksiin (ks. taulukot 2 ja 3), niin merkittävimpänä parannuksena huomataan, että vaikka tietokantojen lukumäärä lisääntyy, sillä ei näytä testeissä olevan vaikutusta muistin käytön suhteen. Vertailuissa huomataan myös, että pyynnön keskimääräinen kesto kasvaa mutta hyvin vähän tässä ratkaisussa. Tämä johtuu siitä, että ensimmäisen pyynnön yhteydessä luodaan tietolähde pyynnön tietokantaa varten. Tietolähteen luonti tuo hyvin pienen viiveen ensimmäiseen pyyntöön, mutta se ei ole merkittävä ongelma tuotantokäytön suhteen.

5.13 Ratkaisun tulokset ja pohdinta

Kuten edellisen luvun testeistä nähtiin, kun Oscar Cloudin tietokantojen hallintaan käytettiin Hibernaten Multitenancy-ominaisuuksia, ovat tulokset erittäin lupaavia. Tulokset paranivat kumpaankin, eli Oscar Cloudin nykyiseen tietokantojen hallinnan toteutukseen sekä alikontekstien elinkaarien hallinnan toteutukseen verrattuna paremmin.

Kun ratkaisua käytiin toimeksiantajan kanssa läpi, olimme yksimielisiä siitä, että tämä on järkevämpi ratkaisu lähteä kehittämään tuotantokäyttöä varten. Tämänlaisenaan ratkaisua ei voi kuitenkaan vielä ottaa tuotantokäyttöön. Oscar Cloudissa on vielä monia ratkaistavia asioita, jotka täytyy muokata ja kehittää uuden mallin kanssa toimiviksi.

6 POHDINTA

Opinnäytetyön tavoitteeseen eli Oscar Cloudin tietokantojen hallinnan optimointiin syntyi opinnäytetyön tuloksena kaksi eri ratkaisua. Ratkaisut eroavat toisistaan suuresti. Ensimmäinen ratkaisu keskittyy ratkaisemaan ongelman kehittämällä Oscar Cloudin nykyistä tietokantojen hallintaa. Toisen ratkaisun kohdalla voisi sanoa, että hallinta tehtiin kokonaan uusiksi. Oscar Cloudin alustuksen prosessi, profiilien hallinta sekä kontekstihierarkia muuttui täysin tai poistui kokonaan.

Opinnäytetyötä varten kehitetyn testiohjelman tuottamista testituloksista käy ilmi, että Hibernate-ohjelmistokehyksen ominaisuuksien käyttöönotto on tutkituista vaihtoehdoista selvästi parempi Oscar Cloudin tietokantojen hallinnan kehittämiseksi. Ominaisuuksien käyttöönotto mahdollisti kaikkien tietokantojen käytön yhdellä Oscar Cloud -instanssilla, eikä sillä ollut huomattavaa vaikutusta instanssin nopeuteen ja se vähensi instanssin resurssien, kuten muistin, käyttöä.

Käytäessä opinnäytetyön tuloksia toimeksiantajan kanssa päätettiin, että niitä aletaan jatkokehittämään ja tutkimaan tuotantokäyttöä varten. Työn tulokset loivat vahvan perustan toimeksiantajalle jatkokehitystä varten. Työn tuloksista kuitenkaan ei kumpaakaan voi sellaisenaan viedä tuotantoon. Oscar Cloudissa on monia ominaisuuksia, jotka toimivat tällä hetkellä vain nykyisen Oscar Cloudin tietokantojen hallinnan toteutuksen kanssa. Kaikki ominaisuudet tarvitsevat jatkokehityksen yhteydessä kehittää yhteensopiviksi opinnäytetyön ratkaisujen kanssa.

LÄHTEET

Baeldung. 2022a. Context Hierarchy with the Spring Boot Fluent Builder API. Verkkosivu. Viitattu 26.9.2022. https://www.baeldung.com/spring-boot-context-hierarchy

Baeldung. 2022b. JPA/Hibernate Persistence Context. Verkkosivu. Viitattu 30.11.2022. https://www.baeldung.com/jpa-hibernate-persistence-context

Baeldung. 2022c. Learn JPA & Hibernate. Verkkosivu. Viitattu 23.11.2022. https://www.baeldung.com/learn-jpa-hibernate

Baeldung. 2022d. Learn Spring Boot. Verkkosivu. Viitattu 20.11.2022. https://www.baeldung.com/spring-boot

Benharosh, J. 2018. What is REST API? In plain English. Phpenthusiast. Verkkosivu. Viitattu 25.5.2022. https://phpenthusiast.com/blog/what-is-rest-api

Beskow, B. 2020. Dynamic Multi Tenancy with Spring Boot, Hibernate and Liquidbase. Callista. Verkkosivu. Viitattu 1.11.2022. https://callistaenterprise.se/blogg/teknik/2020/09/19/multi-tenancy-with-spring-boot-part1/

EHCache. n.d. EHCache: Java's Most Widely Used Cache. Verkkosivu. Viitattu 7.5.2022. https://www.ehcache.org/about/

Gillis, A. 2020. REST API (RESTful API). TechTarget. Verkkosivu. Viitattu 23.4.2022. https://www.techtarget.com/searchapparchitecture/definition/RESTful-API

Hombergs, T. 2020. One-Stop Guide to Profiles with Spring Boot. Reflectoring. Verkkosivu. Viitattu 20.9.2022. https://reflectoring.io/spring-boot-profiles/

Javatpoint. n.d. Java Annotations. Verkkosivu. Viitattu 27.11.2022. https://www.javatpoint.com/java-annotation

NodeJS. n.d. Home. Verkkosivu. Viitattu 27.11.2022. https://nodejs.org/en/

Postman. n.d. What is Postman?. Verkkosivu. Viitattu 21.9.2022. https://www.postman.com/product/what-is-postman/

Red Hat. n.d.-a. Interface CurrentTenantIdentifierResolver. Verkkosivu. Viitattu 30.11.2022. https://docs.jboss.org/hibernate/orm/current/javadocs/org/hibernate/context/spi/CurrentTenantIdentifierResolver.html

Red Hat. n.d.-b. Interface MultiTenantConnectionProvider. Verkkosivu. Viitattu 30.11.2022. https://docs.jboss.org/hibernate/orm/current/javadocs/org/hibernate/engine/jdbc/connections/spi/MultiTenantConnectionProvider.html

Redgate. n.d. Documentation. Verkkosivu. Viitattu 6.5.2022. https://flywaydb.org/documentation/

Spring. 2022a. Core Technologies. Verkkosivu. Viitattu 20.9.2022. https://docs.spring.io/spring-framework/docs/current/reference/html/core.html

Spring. 2022b. Spring Framework Overview. Verkkosivu. Viitattu 20.9.2022. https://docs.spring.io/spring-framework/docs/current/reference/html/overview.html

Spring. n.d.-a. Class AnnotationConfigApplicationContext. Verkkosivu. Viitattu 18.11.2022. https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/AnnotationConfigApplicationContext.html

Spring. n.d.-b. Interface WebRequestInterceptor. Verkkosivu. Viitattu 18.11.2022. https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/context/request/WebRequestInterceptor.html

Suman, Das. 2019. Multi-Tenancy Implementation using Spring Boot + Hibernate. Medium. Verkkosivu. Viitattu 1.11.2022. https://medium.com/swlh/multi-tenancy-implementation-using-spring-boot-hibernate-6a8e3ecb251a

LIITTEET

Liite 1. OscarBackend-luokka (get- ja set-metodit poistettu)

```
package fi.oscar.cloud.settings;

import java.util.ArrayList;
import java.util.List;

public class OscarBackend {

   private String backendName;
   private String inMemoryUsername;
   private String inMemoryPassword;
   private Boolean isEndpointAdmin;
   private String backendType;
   private String jdbcDriverClassName;
   private String jdbcUrl;
   private String jdbcUsername;
   private String jdbcPassword;
   private Integer minPoolSize;
   private Integer maxPoolSize;
   private Integer maxIdleTime;
   private Long maxLifetime;
   private Integer leakDetectionThreshold;
   private Boolean registerMbeans;
   private String databaseMetadataRefreshRate;
   private String cacheRefreshRate;
   private String repairRetryRate;
   private String cacheDiskStorePath;
   private List<Cache> caches = new ArrayList<>();
   private String initializationVector;
   private String keyString;
   private String mappingFile;
   private Integer hibernateJdbcBatchSize;
   private Integer hibernateJdbcFetchSize;
   private Integer boostFetchSize;
   private Boolean hibernateShowSql;
   private Boolean hibernateFormatSql;
   private Boolean hibernateUseSqlComments;
   private Boolean hibernateGenerateStatistics;
   private List<MethodAccessRole> methodAccessRoles = new ArrayList<>();
   private String acuCobol;
   private String path;
   private String ldLibraryPath;
   private String aTermcap;
   private String genesisHome;
   private String vortexHome;

}
```

Liite 2. Opinnäytetyön tuloksien testaamiseen kehitetty testiohjelma

```
/* File app.ts */

import { getContext, getCurrency, getMemoryusage } from './api';
import TestStatistics from './testStatistics';
import { getTimeInSecondsBetweenDates, printFunctionExecutionTime,
printTestSummary } from './util';

const DB_SET_1 = [/* ADD 5 DATABASES */];
const DB_SET_2 = [/* ADD 5 DATABASES */];
const DB_SET_3 = [/* ADD 5 DATABASES */];
export let DB_LIST: string[] = [];

/* ##### TEST EXEC PARAMETERS ##### */
const DB_SET_1_ENABLED: boolean = true;
const DB_SET_2_ENABLED: boolean = false;
const DB_SET_3_ENABLED: boolean = false;
const INIT_CONTEXTS_BEFORE_TESTS: boolean = false;
const RUN_GET_CURRECY_FROM_DB_TIMES: number = 20;
/* ##### TEST EXEC PARAMETERS ##### */

async function main() {
  if (DB_SET_1_ENABLED)
    DB_LIST.push(...DB_SET_1);
  if (DB_SET_2_ENABLED)
    DB_LIST.push(...DB_SET_2);
  if (DB_SET_3_ENABLED)
    DB_LIST.push(...DB_SET_3);

  if (INIT_CONTEXTS_BEFORE_TESTS)
    await initializeAllContexts();

  const start = new Date();
  try {
    const memoryUsageCallIntervalId: number = createMemoryUsageCallInter-
val();
    await getCurrencyFromAllDatabasesXTimes(RUN_GET_CUR-
RECY_FROM_DB_TIMES);
    clearInterval(memoryUsageCallIntervalId)
  } catch(error) {
    console.log('Error while executing tests!');
    console.error(error);
  }

  const end: Date = new Date();
  TestStatistics.testExecutionTime = getTimeInSecondsBetweenDates(start,
end);
  printTestSummary();
```

```
}

function createMemoryUsageCallInterval() {
  return setInterval(() => {
    getMemoryusage();
  }, 1000);
}

async function initializeAllContexts() {
  const start: Date = new Date();
  const promiseAllResponse = await Promise.all(DB_LIST.map(database =>
getContext(database)));
  const end: Date = new Date();
  printFunctionExecutionTime('initializeAllContexts', start, end);
}

async function getCurrencyFromDatabaseXtimes(database: string, times:
number) {
  const start: Date = new Date();

  for (let i = 0; i < times; i++) {
    const result = await getCurrency(database, 1);
  }

  const end: Date = new Date();
  printFunctionExecutionTime('getCurrencyFromDatabaseXtimes', start,
end);
}

async function getCurrencyFromAllDatabasesXTimes(times: number) {
  const start: Date = new Date();

  const promises: Promise<any>[] = [];
  DB_LIST.forEach(database => {
    promises.push(getCurrencyFromDatabaseXtimes(database, times));
  });
  const promisesResult = await Promise.all(promises);

  const end: Date = new Date();
  printFunctionExecutionTime('getCurrencyFromAllDatabasesXTimes', start,
end);
}

main();

/* End of file app.ts */
```

```
/* File api.ts */

import axios from 'axios';
import TestStatistics from './testStatistics';
import { printFunctionExecutionTime, getTimeInSecondsBetweenDates } from
'./util';

axios.defaults.baseURL = "http://localhost:8180";
const restUrl: string = '/ocresource/rest/OCResource';

const auth = {
  username: '',
  password: ''
};

export async function getContext(backend: String) {
  const start: Date = new Date();

  try {
    const { data } = await axios.get(`${restUrl}/${backend}/databaset-
est/context`, { auth: auth });
    console.log(`${backend} - getContext response: ${data}`);
  } catch (error) {
      console.log(error)
  }

  const end: Date = new Date();
  printFunctionExecutionTime('getContext', start, end);
}

export async function getCurrency(backend: string, currencyId: number) {
  const start: Date = new Date();

  let response = '';
  try {
    TestStatistics.totalApiCalls += 1;
    const { data } = await axios.get(`${restUrl}/${backend}/databaset-
est/currency/${currencyId}`, { auth: auth });
    if (data.currencyId) {
      response = `${backend} - getCurrency response currencyId:
${data.currencyId}`;
    }
  } catch (error) {
    if (error.response?.status === 404) {
      response = `${backend} - getCurrency response : ${error.re-
sponse.data.message}`;
    } else {
      console.log(error)
```

```
    }
  } finally {
    console.log(response);
  }

  const end: Date = new Date();
  TestStatistics.sumOfApiCallsExecutionTimes += getTimeInSecondsBe-
tweenDates(start, end);
  printFunctionExecutionTime('getCurrency', start, end);
}

export async function getMemoryusage() {
  const start: Date = new Date();

  try {
    TestStatistics.totalMemoryUsageCalls += 1;
    const { data } = await axios.get(`${restUrl}/databasetest/memory`, {
auth: auth });
    const memoryUsage = Number(data);
    TestStatistics.sumOfMemoryUsages += memoryUsage;
    console.log(`getMemoryusage response: ${memoryUsage}`);
  } catch (error) {
    console.log(error);
  }

  const end: Date = new Date();
  printFunctionExecutionTime('getMemoryusage', start, end);
}

/* End of file api.ts */
```

```
/* File util.ts */

import { DB_LIST } from "./app";
import TestStatistics from "./testStatistics";

export function getTimeInSecondsBetweenDates(start: Date, end: Date):
number {
  return (end.getTime() - start.getTime()) / 1000;
}

export function printFunctionExecutionTime(functionName: string, start:
Date, end: Date): void {
  const duration: number = getTimeInSecondsBetweenDates(start, end);
  console.log(`Function ${functionName}  took ${duration} seconds to exe-
cute`);
}

export function printTestSummary() {
```

```
  const averageApiCallExecutionTime = TestStatistics.sumOfApiCallsExecu-
tionTimes / TestStatistics.totalApiCalls;
  const averageMemoryUsage = TestStatistics.sumOfMemoryUsages / TestSta-
tistics.totalMemoryUsageCalls;

  const summaryObject = {
    testExecutionTime: `${TestStatistics.testExecutionTime.toFixed(2)}
seconds`,
    amounOfBackends: DB_LIST.length,
    totalApiCalls: TestStatistics.totalApiCalls,
    averageApiCallExecutionTime: `${averageApiCallExecution-
Time.toFixed(2)} seconds`,
    totalMemoryUsageCalls: TestStatistics.totalMemoryUsageCalls,
    averageMemoryUsage: `${averageMemoryUsage.toFixed(0)} MB`
  }

  console.log('#################### TEST SUMMARY ####################');
  console.log(summaryObject);
  console.log('#################### TEST SUMMARY ####################');
}

/* End of util.ts */
```

```
/* File testStatistics.ts */

interface TestStatisticType {
  testExecutionTime: number;
  totalApiCalls: number;
  sumOfApiCallsExecutionTimes: number;
  totalMemoryUsageCalls: number;
  sumOfMemoryUsages: number;
}

const TestStatistics: TestStatisticType = {
  testExecutionTime: 0,
  totalApiCalls: 0,
  sumOfApiCallsExecutionTimes: 0,
  totalMemoryUsageCalls: 0,
  sumOfMemoryUsages: 0,
};

export default TestStatistics;

/* End of file testStatistics.ts */
```