Grigorij Semykin

Ajoneuvokaluston digitaalinen kaksonen

Metropolia Ammattikorkeakoulu
Insinööri (AMK)
Tieto- ja viestintätekniikka
Insinöörityö
24.5.2021

Tiivistelmä

Tekijä:	Grigorij Semykin
Otsikko:	Ajoneuvokaluston digitaalinen kaksonen
Sivumäärä:	27 sivua + 1 liite
Aika:	24.5.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Lehtori Juha Pekka Kämäri

Insinöörityön tavoitteena oli kehittää toimeksiantajan määritettyjen vaatimusten mukainen tietokanta sekä siihen REST-rajapinta. Työn toisena tavoitteena oli kehittää tietokantarajapinnalle kevyt ja yksinkertainen käyttöliittymä CRUD-operaatioita varten. Työn tavoitteena oli kehittää sovellus, jolla olisi mahdollista tallentaa, muokata, poistaa sekä analysoida erilaisia ajoneuvoja sekä niiden metatietoja. Työn toimeksiantajana oli Vediafi Oy, joka tarjoaa ratkaisuja tehokkaampaan ja ympäristöystävällisempään logistiikkaan.

Projektin aikarajoitteiden takia toimeksiantaja oli hyväksynyt minun ehdotetut kehitystyökalut sekä -ympäristön. Projekti oli kehitetty Javalla sekä Spring-ohjelmistokehyksellä, joka tarjoaa kattavat työkalut REST-rajapintojen, SQL-tietokantojen sekä käyttöliittymien kehitykseen.

Insinöörityön lopputuloksena saatiin kehitettyä toimiva sekä hyvin helposti päivitettävä ja laajennettava sovellus, jonka tarkoituksena on korvata toimeksiantajan nykyinen tietokantaratkaisu. Kaikki toimeksiantajan asettamat tavoitteet on saavutettu ja sovellus on lähetetty asiakkaalle käyttöönottoa, testausta ja jatkokehitystä varten.

Avainsanat:	REST, rajapinta, CRUD, SQL

Abstract

Author:	Grigorij Semykin
Title:	Vehicles DigitalTwin
Number of Pages:	27 pages + 1 appendix
Date:	24 May 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Software Engineering
Supervisors:	Juha Pekka Kämäri, Principal Lecturer

The goal of the thesis was to develop a database and a REST API that would meet the requirements set by the client. As a secondary objective, a simple and easy to use user interface had to be created alongside the application, to perform CRUD- operations.

Due to a limited time given for the development, the client had accepted the tools and development environment that I had suggested. The project was developed using Java and Spring framework, which has comprehensive tools for developing REST APIs, SQL databases and user interfaces.

The resulting application met all the requirements that the client had set. The latest version of application is stable, easy to update and expand if needed. The goal of the application is to replace client's current database solution with mine. The application was sent to the client for studying, testing, and deployment.

Keywords: REST, API, CRUD, SQL

Sisällys

Lyhenteet

1 Johdanto 1

2 Vediafi Oy ja CVW 1

2.1 Vediafi Oy 1
2.2 CVW - Clean Vehicle Wizard -työkalu 2
2.3 EU:n asettama puhtaiden ajoneuvojen direktiivi 3
2.4 Ajoneuvokaluston digitaalinen kaksonen 4

3 Projektin lähtökohdat ja toimintaympäristö 4

4 Projektin kehitys 5

4.1 Projektin kokonaiskuva 5
4.2 Spring Boot -projektin aloitus 7
4.3 Tietokannan alustaminen 9
4.4 JPA ja tietokantamallit 9
4.5 @Service- ja @Repository-luokat 11
4.6 REST-API-, @Controller- ja @RestController-luokat 14
4.7 UML-luokkakaavio 16
4.8 JWT 18
4.9 Käyttöliittymä 20
4.10 Testit 23

5 Yhteenveto 26

Lähteet 27

Asiakkaan määritetyt vaatimukset 1

Liitteet

Liite 1: Asiakkaan määritetyt vaatimukset

Lyhenteet

API	*Application Programming interface*. Ohjelmointirajapinta on määritelmä, jonka mukaan ohjelmat voivat kommunikoida.
JPA	Java Persistance API on rajapinta, joka määrittelee relaatiotietokannan datan käytön Java-ohjelmissa.
Hibernate	Hibernate on kirjasto, joka toteuttaa JPA:n rajapinnan määritelmät.
PostgreSQL	Avoimeen lähdekoodin perustuva olio-relaatiotietokantapalvelin, joka on luotettava ja sisältää paljon ominaisuuksia.
Spring	Javalle suunnattu ohjelmistokehys.
HTTP	Hypertext Transfer Protocol. Verkkoselaimien ja WWW-palvelimien käyttämä tiedonsiirtoprotokolla.
REST	REpresentational State Transfer. Yleinen arkkitehtuurimalli rajapintojen toteuttamiseen web-palveluille.
JWT	JSON Web Token. JSON-pohjainen avoimen standardin menetelmä käyttöoikeustietueiden hallinnoimiseen eri ohjelmistojen välillä.
HTML Tag	HTML Tag tai tunnisteet ovat avainsanoja, jotka määrittelevät, kuinka verkkoselain muotoilee ja näyttää HTML-sivun sisällön.

1 Johdanto

Tämän insinöörityön aiheena on kehittää asiakasyrityksen Vediafi Oy:n antamien vaatimusten mukainen tietokannan hallintajärjestelmä API-rajapinnalla. Järjestelmän tulisi olla sovellettavissa EU:n asettamaan puhtaiden ajoneuvojen direktiiviin ja vastata sen vaatimuksiin.

Työn tavoitteena on kehittää toimiva, jatkokehitettävä ja tarvittaessa helposti päivitettävä järjestelmä, joka täyttää kaikki asiakkaan asettamat vaatimukset.

Työssä tullaan käsittelemään, miten kyseinen järjestelmä voidaan kehittää Spring Boot -ympäristössä ja mitä työkaluja sekä kirjastoja Spring tarjoaa ja mitä tullaan käyttämään.

2 Vediafi Oy ja CVW

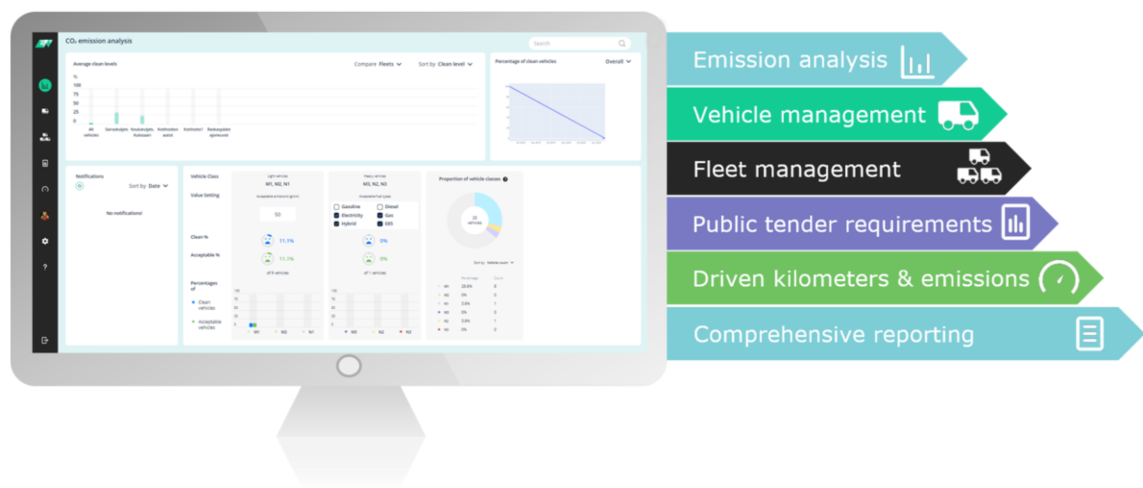
Tässä luvussa kerrotaan yrityksestä, heidän käytössään olevasta ohjelmistosta sekä projektin alustavasta tarpeesta.

2.1 Vediafi Oy

Tässä projektissa asiakasyrityksenä toimi Vediafi tai lyhyemmin Vedia, joka on vuonna 2013 perustettu suomalainen yritys. Yritys kehittää ja myy ratkaisuja, joilla voidaan tehostaa logistiikkaa ja edistää ympäristöystävällisyyttä. Yritys tarjoaa ratkaisuja, joissa yhdistyy mobiili-, IoT-, paikannuspalvelut sekä datapohjaiset palvelut esimerkiksi logistiikkapalvelutarjoajille, kunnille tai rahdinantajille Vedia CaaS -brändin alla. Alussa yritys keskittyi enimmäkseen henkilöliikenteen palveluiden parantamiseen sekä ympäristöasioiden edesauttamiseen, mutta 2016 aikana palveluiden painopiste siirtyi kokonaan logistiikan palveluiden kehittämiseen. [1.]

2.2 CVW - Clean Vehicle Wizard -työkalu

CVW eli Clean Vehicle Wizard on yrityksellä tällä hetkellä käytössä. Se on jatkuvassa kehityksessä oleva verkkopohjainen työkalu, joka mahdollistaa kestävän kehityksen huomioimisen erilaisissa kuljetuspalveluissa sekä omassa ajoneuvokalustossa ja kuljetustoiminnassa. Työkalu mahdollistaa yksittäisten ajoneuvojen tietueiden seuraamista sekä raportointia. Tärkeimpänä parametrina viime aikoina on ollut CO2-päästöt ja varsinkin niiden vähentäminen.

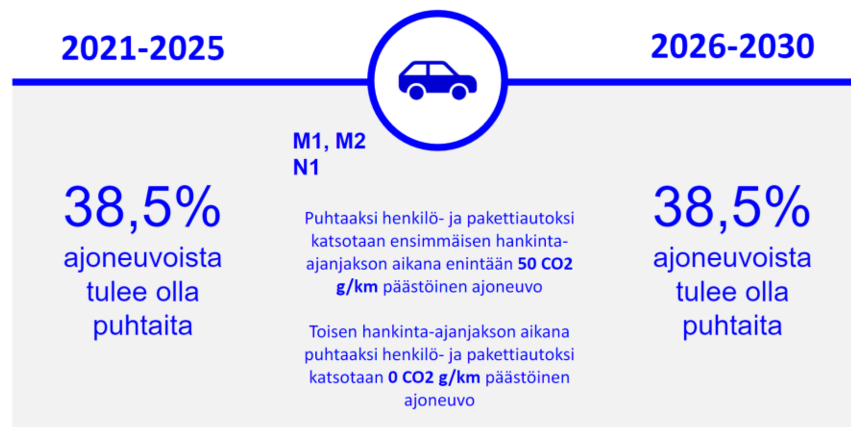


Kuva 1. Clean Vehicle Wizard -työkalun pääominaisuudet

Kuvassa 1 näkyy verkkopohjaisen CVW-työkalun pääominaisuudet. Työkalun avulla saadaan hyvin koostettu tilannekuva koko ajoneuvokaluston tilanteesta, sisällöstä sekä puhtaudesta ajoneuvoluokittain sekä luotujen ryhmien osalta. [2.]

2.3 EU:n asettama puhtaiden ajoneuvojen direktiivi

Vuonna 2017 Euroopan unioni antoi ehdotuksen (COM (2017) 653 final) muuttaa vuonna 2009 luotua direktiiviä (2009/33/EY) uudeksi direktiiviksi (Clean Vehicle directive, CVD). Uusi direktiivi koskee useita soveltamisaloja, joilla toteutetaan julkisia hankintoja ajoneuvo- sekä liikennepalveluina. Direktiivi koskee EU:n jäsenmaita ja edellyttää julkiselta sektorilta ajoneuvokaluston hankinnassa, vuokrauksessa sekä osamaksukaupalla toteutettavassa hankinnassa puhtaiden ajoneuvojen prosentuaalisia vähimmäisosuuksia. Vaatimukset koskevat ainoastaan uusia hankintoja, jotka ylittävät EU:n hankintalainsäädännön hintakynnykset. [3; 4.]



Kuva 2. Henkilö- ja pakettiautoille asetetut vaatimukset

Kuvassa näkyy, että ensimmäisessä jaksossa 2021–2025 38,5 % ajoneuvoista tulee olla puhtaita eli ajoneuvo voi tuottaa enintään 50 CO2 g/km. Toisessa jaksossa 2026–2030 puhtaaksi ajoneuvoksi katsotaan se, joka ei tuota päästöjä ollenkaan. Tämä on vain lyhyt esimerkki henkilö- ja pakettiautoista. Todellisuudessa ajoneuvotyyppejä, -luokkia sekä erikoistapauksia on huomattavasti enemmän, joita kyseinen direktiivi koskee.

2.4 Ajoneuvokaluston digitaalinen kaksonen

Terminä digitaalinen kaksonen tarkoittaa tietokonemaailmassa objektia, jolla on reaalimaailman objektin identtiset parametrit sekä ominaisuudet. Meidän tapauksessa sekä tässä projektissa digitaalisia kaksosia on rajattu ajoneuvoihin sekä ajoneuvokalustokokonaisuuksiin. EU:n uudesta direktiivistä johtuen Vedia oli päättänyt laajentaa sekä päivittää heidän nykyistä tietokantansa, jotta heidän Clean Vehicle Wizard -työkalua olisi entistä helpompaa käyttää ja se olisi ajan tasalla direktiivin kanssa. Tästä alkoi minun varsinainen rooli kehittäjänä: täysin uuden tietokannan kehitys rajapinnalla, asiakkaan antamien vaatimusten mukaan.

3 Projektin lähtökohdat ja toimintaympäristö

Tässä luvussa kerrotaan yrityksen asettamista määrittelyistä, vaatimuksista sekä minun käyttämistäni teknologioista.

Nykypäivän rajoitusten takia koko projekti kehitettiin etätyömuodossa kotoa. Kuitenkin pidettiin videopalavereja muutaman viikon välein. Näin saatiin tehokasta projektin kehitystä omalta osalta sekä välitöntä palautetta asiakkaalta. Asiakas olisi alustavasti toivonut ohjelmointikieleksi Pythonin, mutta arvelin, että sen oppimiseen, käyttöönottoon sekä hyvänlaatuiseen ohjelmiston kehitykseen olisi mennyt paljon enemmän aikaa, mitä oli alustavasti varattu. Koska kyseisellä projektilla olisi rajapinta, mikä mahdollistaisi tietokonnan kanssa keskustelun laitteesta sekä ohjelmistosta riippumatta asiakas hyväksyi ehdotukseni rakentaa projektin Java-ohjelmointikielellä. Toimeksiantajan yksi tärkeimmistä vaatimuksista oli se, että tietokantana olisi PostgreSQL ja sovelluksella olisi REST API -ohjelmointirajapinta. Kaikki vaatimukset ovat nähtävissä liitteessä 1. Näiden vaatimusten perusteella päätin rakentaa sovelluksen Spring Boot -ohjelmointikehyksen avulla, joka sopii täydellisesti kyseiseen tehtävään.

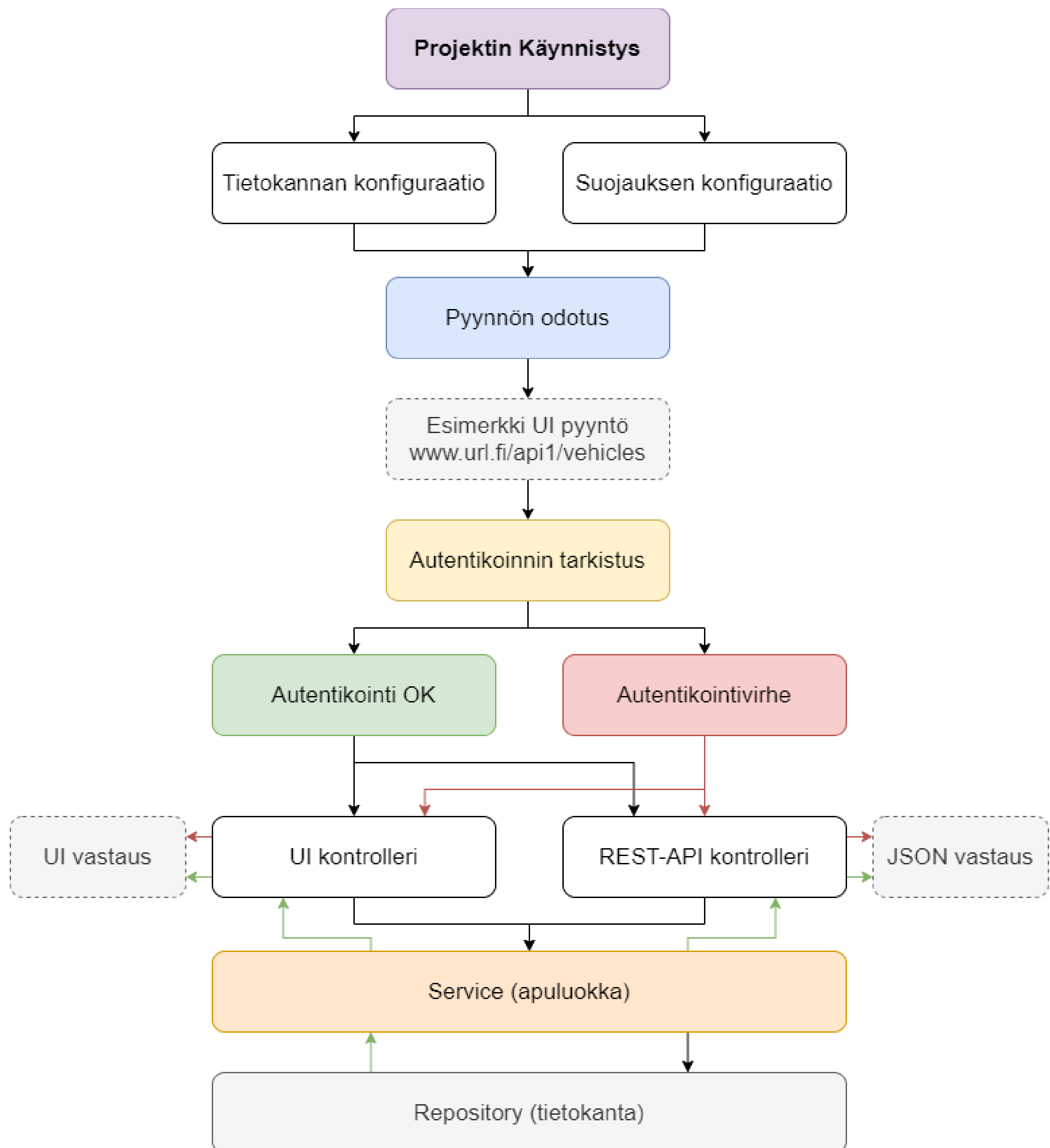
Projektin lähtökoodi sijaitsee minun omassa yksityisessä GitHubissa [5], joka on Microsoftin omistama web-pohjainen versionhallintasovellus. Koska olin projektin ainoa kehittäjä, minulle riitti kaksi kehityshaaraa: ”develop”, joka edustaa sovelluksen kehitystä, sekä ”master”, joka edustaa sovelluksen vakainta versiota. Projektin rakentamiseen sekä kirjastojen lataamiseen ja ylläpitämiseen käytetään Gradlea [6]. Gradle on erityisesti Java-ekosysteemin kielille tarkoitettu moderni avoimen lähdekoodin rakennusautomaatio-työkalu, joka keskittyy joustavuuteen ja suorituskykyyn. Lisäksi se on hyvin muokattavissa ja mahdollistaa tehtävien suorittamisen nopeasti uudelleen käyttämällä aiempien koontien tuloksia.

4 Projektin kehitys

Tässä luvussa käydään läpi projektin yleiskuva sekä tekninen rakenne.

4.1 Projektin kokonaiskuva

Teknisestä näkökulmasta tämä projekti voidaan jakaa kolmeen isoon aihealueeseen, jotka ovat REST-API, käyttöliittymä-API sekä tietokanta ja siihen liittyvät komponentit. Kaikille näille oasalueille Spring tarjoaa valmiit, helppokäyttöiset ja hyvin kattavat työkalut, joita on otettu käyttöön. Kuvassa 3 on esitetty projektin jako erilaisiin komponentteihin ja miten projekti käyttäytyy käynnistyksen jälkeen.



Kuva 3. Projektin tekninen kokonaiskuva

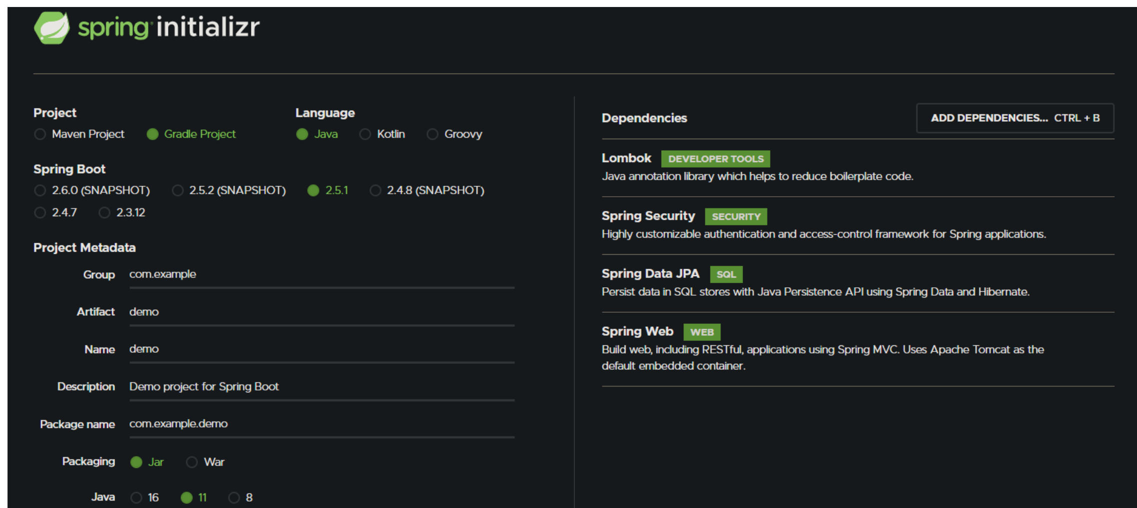
UI-kontrolleri ja REST-API-kontrolleri ovat projektin päälogiikan komponentit ja toimivat päärooleissa. Kontrollerit eivät ole suoraan yhteydessä tietokantaan, vaan Service-komponenttien kautta. Service-komponentti ei ole kuitenkaan pakollinen eikä Spring-kehys pakota ottamaan sitä käyttöön kyseisellä projektin kokoonpanolla, vaan se toimii niin sanottuna apuluokkana, jolla saadaan vähennettyä koodin toistuvuutta. Tämä onnistuu siten, että määritetään Service-luok-

kia esimerkiksi validoimaan kaikki data, mitä tallennetaan tietokantaan tai tarkistamaan erilaisia ehtoja, ennen kuin ruvetaan poistamaan tai muokkaamaan tietokannan tietueita. Projektin suojaus, kuten API:t, on jaettu kahteen osaan REST-API:n sekä käyttöliittymä-API:n suojauksiin. Käyttöliittymän suoja on ehkä kaikista yleisin käyttäjätunnuksen ja salasanakombinaation tarkistus tietokantaa vasten. Koska kyseinen suojamekanismi on niin yksinkertainen, sitä ei tulla käymään sen kummemmin läpi. REST-API-suoja on toisaalta muutaman asteen vaikeampi sekä kiinnostavampi. Tämä suoja oli yksi asiakkaan vaatimuksista, se on JSON Web Token tai lyhyemmin JWT. Lyhykäisyydessä jos salasana ja käyttäjätunnus ovat oikein ja käyttäjä on olemassa tietokannassa, generoidaan ainutlaatuinen pitkä merkkijono. Kaikissa seuraavissa kutsuissa ja pyynnöissä kyseinen merkkijono täytyy olla Header-elementissa mukana päästäkseen haluttuihin resursseihin ja REST-API:n toimintoihin. JWT käydään tarkemmin läpi luvussa 4.7.

Seuraavissa luvuissa käydään kaikki tässä luvussa käydyt osat läpi tarkemmin.

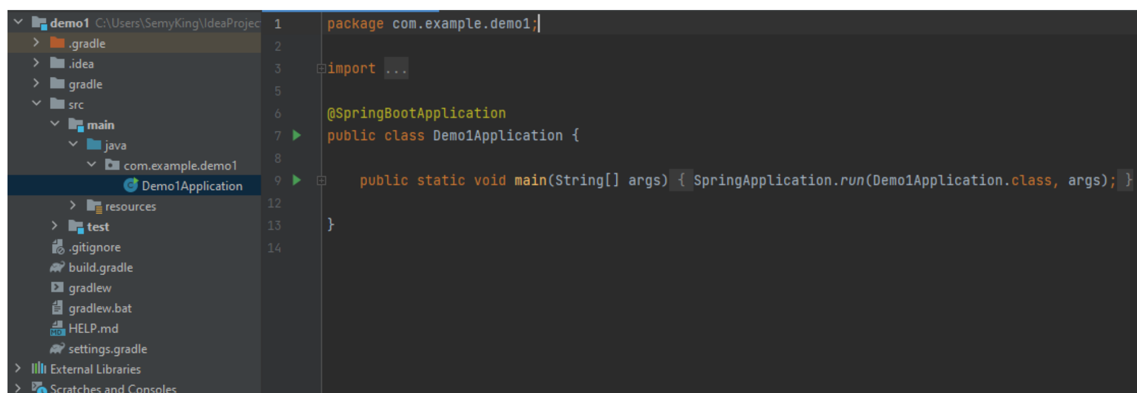
4.2 Spring Boot -projektin aloitus

Spring-projektin aloittamiseen on kehitetty hyvin selkeä ja yksinkertainen sivusto: spring initializr [7]. Kuvassa 4 näkyy esimerkki tämän projektin konfiguraatiosta. Kyseisen sivuston avulla käyttäjä voi valita itselleen sopivat Spring-komponentit, kuten esimerkiksi tuleeko projekti olemaan Maven- vain Gradle-pohjainen, Java-, Kotlin- vai Groovy-kielinen, Spring Boot version, projektin nimen, projektin pakkaustyypin sekä Java-ohjelmointikielen version. Lisäksi on mahdollista etukäteen valita riippuvuudet, joita projekti tulee tarvitsemaan myöhemmin kehitysvaiheessa.



Kuva 4. Spring Initializr -sivuston käyttöliittymä [7]

Kaikki valinnat tehtyään voidaan generoida ja ladata tyhjä Spring-projekti valituilla parametreilla. Tämän jälkeen projekti tuodaan tai avataan itselleen mieluisassa koodieditorissa, joka tässä tapauksessa on ollut IntelliJ IDEA. Koodieditori heti lataa kaikki projektin tarvittavat komponentit ja konfiguraation päätettyä saadaan aikaan kuvan 5 mukainen tyhjä aloitusprojekti yhdellä luokalla sekä main-metodilla, jotta voidaan heti käynnistää ja varmistaa, että kaikki toimii.



Kuva 5. Tyhjä Spring-projekti

Käynnistämällä projektin tässä vaiheessa kaiken pitäisi toimia ongelmitta ja kehitystyökalun konsolissa meille ilmoitetaan url-osoite, joka tuleekin olla meidän projektin sisääntulopiste.

4.3 Tietokannan alustaminen

Tietokantana tässä projektissa toimi PostgreSQL. Sen alustamiseen varsinainen PostgreSQL-ohjelmisto tulee ladata, asentaa ja käynnistää. Asennuksessa on tärkeä kohta, portin valinta, jossa tietokantaa tulee pyörimään. Toisena kohtana on pgAdmin-ohjelmiston asennus, joka on tietokannan helppokäyttöinen visuaalinen editori. Kyseistä editoria käyttäen luodaan uusi tietokanta ja annetaan sille nimeksi – ’vedia_vehicles_database’.

Kaikki edelliset askeleet tehtyään voidaan nyt määrittää projektin käyttävän äsken luotua uutta tietokantaa. Kyseinen Spring-konfigurointi tapahtuu application.properties-tiedostossa, joka sijaitsee src/main/resources-kansiossa. Tässä vaiheessa tärkeimmistä avain-arvopareista ovat tietokannan url-osoite, joka tässä tapauksessa on seuraava - jdbc:postgresql://localhost:5432/vedia_vehicles_database. Localhost ilmaisee, että tietokanta pyörii samalla koneella, kuten projektikin, portti oli asennuksen yhteydessä annettu ja tietokannan nimi itse luotu. Muut tarvittavat avain-arvoparit ovat tietokannan käyttäjänimi sekä salasana, tietokantaan pääsyä varten.

4.4 JPA ja tietokantamallit

Tietokantamallina tarkoitetaan Javan dataluokkaa, jossa ei ole funktionaalista toimintaa getterien ja setterien lisäksi. Näiden luokkien yläpuolelle kirjoitetaan @Entity- sekä @Table-annotaatiot. Nämä JPA-standardin mukaiset annotaatiot merkitsevät ne Java-luokat, joita projektin käynnistäessä käännetään SQL-tauluiksi ja tallennetaan meidän tietokantaan. Tätä toimintaa voidaan hallita spring.jpa.hibernate.ddl-auto-avaimen arvoina. Näitä arvoja on neljä ja niiden toiminnallisuus on seuraava:

- validate: tarkista skeema, mutta älä tee mitään muutoksia
- update: päivitä skeema, jos Java-luokkiin on tullut muutoksia

- create: luo aina uusi skeema ja poista edellinen
- create-drop: poista skeema, kun sessio päättyy.

SQL-skeema on tietokannan rakenteen tekstimuotoinen esitys, jossa annetaan tietokannan luomiseen tarvittavat SQL-komennot. Tämän esitystavan etuna on, että se on varmasti täsmällinen ja voimme halutessamme luoda suoraan tietokannan sen perusteella. Esimerkkikoodissa 1 nähdään esimerkkirivi SQL-skeemasta, jonka avulla olemassa olevaan tietokantaan luodaan uusi User-tietokantataulu. Todellisuudessa skeemassa olisi enemmän rivejä, joissa on määritetty koko tietokannan rakenne.

```
CREATE TABLE User (id INTEGER PRIMARY KEY, name TEXT, username TEXT);
```

Esimerkkikoodi 1. Esimerkki skeemasta, jonka avulla luodaan User-taulu.

Projektin kehitysvaiheessa on hyvä käyttää update-arvoa. Huomautuksena on kuitenkin, että kyseinen arvo ei poistaa tauluja eikä tietokantarivejä varsinaisesta tietokannasta, vaikka ne olisi poistettu Java-luokasta. Parametrit eivät haittaa kehitystä, sillä ne eivät ole käytössä, mutta jossain vaiheessa huomaa, että tietokannassa on ylimääräisiä tauluja ja rivejä. Tietokannan sarakkeiden (column) nimet sekä niiden oletusarvot määritetään @Entity-annotoiduissa Java-luokissa muuttujilla, niiden tyypillä sekä niiden arvoilla. Muuttujille määritetään @Column-annotaatio, joka on kaikista yleisin. Erikoistapauksissa @Column-annotaation sijaan määritetään muut annotaatiot. Esimerkkikoodissa 2 näkyy esimerkiksi id-muuttuja ja sen JPA-annotaatiot. Tärkeimpinä annotaatioina ovat myös viiteavainten määrittäminen. Näissä joutuu välillä pohdiskelemaan, sillä niissä täytyy myös määrittää yhteydet ja niin sanotut osallistumisrajoitteet tietokantataulujen välillä.

Osallistumisrajoitteet voivat olla - yksi moneen (one to many), moni yhteen (many to one) tai moni moneen (many to many). Näitä voi helposti määrittää väärin, eikä sovellus välttämättä ilmoita virheestä ennen kuin tehdään jonkinlai-

nen kutsu tai operaatio kyseisille muuttujille. Samassa esimerkkikoodissa nähdään fleets-muuttujan konfiguraatio. Fleet on niin sanottu ajoneuvokanta tai kokoelma ajoneuvoja asiakkaan toiveiden mukaan. Ajoneuvo ja ajoneuvokanta pitäisi olla konfiguroitu seuraavalla tavalla – ajoneuvo voi olla useammassa ajoneuvokannassa ja yhdessä ajoneuvokannassa voi olla useampi ajoneuvo. Tämä tarkoittaa sitä, että Vehicle- ja Fleet-dataluokkien välillä pitäisi olla moni moneen -yhteys. Tästä konfiguraatiosta vastaavat kahdet annotaatiot: @ManyToMany sekä @JoinTable(…). Muut annotaatiot liittyvät Lombok – Javan annotaatiokirjastoon.

4.5 @Service- ja @Repository-luokat

@Service ja @Repository ovat viimeiset luokat, jotka liittyvät tietokantaan kyseisessä projektissa. Esimerkkikoodissa 2 on Vehicle entity -dataluokan alkuosa. Tälle luokalle voimme luoda tietokannan käsittelyyn käytettävän rajapinnan. Spring-sovelluskehystä ja JPA-standardia käyttäessämme tietokannan käsittelyyn tarkoitettu rajapinta perii valmiin JpaRepository-rajapinnan, joka määrittelee normaalin CRUD-toiminnallisuuden (create, read, update, delete) sekä joukon muita metodeja. Tämä voidaan nähdä kuvassa 6. Perittävälle JpaRepository-rajapinnalle annetaan kaksi tyyppiparametria. Ensimmäisellä tyyppiparametrilla kerrotaan tietokantataulua kuvaava luokka ja toisella tyyppiparametrilla tietokantataulun pääavaimen tyyppi. Näistä rajapinnoista ei tarvitse tehdä konkreettista toteutusta, sillä Spring luo automaattisesti rajapinnan toteuttavan olion sovelluksen käynnistyksen yhteydessä. Esimerkkikoodissa 3 nähdään ajoneuvojen @Repository-rajapintatoteutus. CRUD-toiminnallisuuden lisäksi, jota peritään JpaRepository-rajapinnasta, voidaan myös luoda omia tietokantakutsuja @Query-annotaation avulla, johon kirjoitetaan tarvittava SQL-kutsu. Samassa esimerkkikoodissa nähdään itseluotu SQL-kutsu, jonka avulla haetaan kaikki ajoneuvot, jotka ovat tietyssä organisaatiossa, kyseisen organisaation id-parametrin avulla.

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table
public class Vehicle {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   @Column
   private String name;


   @ToString.Exclude
   @EqualsAndHashCode.Exclude
   @ManyToMany(fetch = FetchType.EAGER)
   @JsonIgnoreProperties("vehicles")
   @JoinTable(
         name = "vehicle_fleets",
         joinColumns = @JoinColumn(name = "vehicle_id"),
         inverseJoinColumns = @JoinColumn(name = "fleet_id"))
   private Set<Fleet> fleets = new HashSet<>();
...
}
```

Esimerkkikoodi 2. Vehicle-dataluokan alkuosa.

```
@Repository
@Qualifier("vehicles")
public interface VehicleRepository extends JpaRepository<Vehicle,
Long> {

    @Query("SELECT v FROM Vehicle v WHERE v.organisation IS NOT NULL
AND v.organisation.id = :id")
    List<Vehicle> findAllByOrganisationId(@Param("id") Long id);
}
```

Esimerkkikoodi 3. VehicleRepositoryn rajapinta.

Koska @Repository-luokat ovat rajapintoja, niissä ei voi toteuttaa mitään loogista toiminnallisuutta, esimerkiksi datan validointia ennen tietokantaan tallentamista. Kyseistä toiminnallisuutta ei myöskään haluta toteuttaa kaikissa Controller-luokissa, jolloin avuksemme tulee @Service-luokka. Kyseinen luokka toimii niin sanottuna väliluokkana Controller- ja @Repository-luokkien välillä. Tämä luokka toimii samaan tapaan, kuten @Repository-luokassa oleva @Query-metodi. Molempia ei tarvitse käyttää, jos perus-CRUD-toiminnallisuus on riittävä. Meidän tapauksessamme, datan validointi on hyvin tärkeä, jolloin sen toteutus onkin tehty @Service-luokissa. Tästä seuraa se, että kaikki meidän Controller-

luokat eivät ole suoraan yhteydessä tietokantaan eli omiin @Repository-rajapintoihin, vaan omiin @Service-luokkiin. Esimerkiksi ennen tietokantaan tallentamista kaikki data tarkistetaan validate-metodissa, minkä jälkeen dataa tallennetaan tai ilmoitetaan virheestä. Esimerkkikoodissa 4 on esitetty VehicleService-luokan kokonaisrakenne metodeineen kuitenkin ilman metodien toiminnallisuutta. Luokassa nähdään, että varsinainen tietokanta eli VehicleRepository on esitetty VehicleService-luokan muuttajana. Kun Controller-luokka esimerkiksi pyytää välittää sille kaikki tietokannassa olevat ajoneuvot, se kutsuu VehicleService-luokan getAll()-metodia, joka omalta osaltaan kutsuu vehicleRepository-muuttujan findAll()-metodia, joka lopulta palauttaa ajoneuvolistan. Koska kyseessä on luokka eikä rajapinta, voimme helposti ennen ”return vehicleRepository.findAll();” -riviä laittaa esimerkiksi käyttöoikeuksien tarkistuksen tai oikeastaan mitä vaan loogista toimintaa. Tämä pitää meidän Controller-luokat siisteinä ja helpottaa projektin ylläpitoa. Kyseistä projektin rakennetta pidetään yleisesti hyvänä ja oikeana tapana rakentaa sovellusta. Controller-luokat tekevät vain sen, mitä niiden täytyykin, eli reagoivat kyselyihin ja pyyntöihin pyytämällä tai antamalla tiedot @Service-luokille, jotka sitten tekevät kaiken muun loogisen työn.

```
@Service
@RequiredArgsConstructor
public class VehicleService {

   private final VehicleRepository vehicleRepository;


   @Transactional
   public List<Vehicle> getAll() {
       return vehicleRepository.findAll();
   }

   public ValidationResponse validate(Vehicle vehicle, Mapping
mapping) {}


   @Transactional
   public Vehicle getById(Long id) {}

   @Transactional
   public Vehicle save(Vehicle vehicle) {}

   @Transactional
   public void delete(Vehicle vehicle) {}

   @Transactional
```

```
   public void deleteAll() {}
}
```

Esimerkkikoodi 4. VehicleService-luokka

4.6 REST-API-, @Controller- ja @RestController-luokat

Yksi projektin tärkeimmistä vaatimuksista oli REST-API:n toteutus. Lisätavoitteena, ajan salliessa, oli luoda API:lle yksinkertaiset näkymät, joilla olisi mahdollista toteuttaa kaikki CRUD-funktionaalit. Näihin tehtäviin Spring-ohjelmistokehyksellä on olemassa valmiit ratkaisut. API:n näkymistä vastaavat ne luokat, jotka on merkitty "@Controller"-annotaatiolla, kuten esimerkkikoodi 5 ja luokat, jotka vastaavat REST-API:sta, on merkitty "@RestController"-annotaatiolla, kuten esimerkkikoodi 6. Nopeasti katsottuna luokat muistuttavat rakenteeltaan toisiaan, mutta niillä on kuitenkin pieni, mutta hyvin merkittävä ero.

```
@Controller
@RequiredArgsConstructor
@RequestMapping(Constants.UI_API + "/vehicles")
public class VehicleController {

@Autowired
private final VehicleService vehicleService;

@GetMapping({"", "/"})
public String getAll(Model model) {
   List<Vehicle> vehicles = vehicleService.getAll();
   model.addAttribute("vehicles", vehicles);

   return "vehicle/vehicles_list_page";
}

@GetMapping("/{id}")
public String getById(@PathVariable Long id, Model model) {}

@PostMapping({"", "/"})
public String post(@ModelAttribute Vehicle vehicle, Model model) {}
...
}
```

Esimerkkikoodi 5. VehicleController-luokka

```
@RestController
@RequiredArgsConstructor
@RequestMapping(Constants.JSON_API + "/vehicles")
public class VehicleRestController {

   @Autowired
   private final VehicleService vehicleService;


   @GetMapping(value = {"", "/"}, produces =
MediaType.APPLICATION_JSON_VALUE)
   public List<Vehicle> getAll() {
     return vehicleService.getAll();
}


   @GetMapping(value = "/{id}", produces =
MediaType.APPLICATION_JSON_VALUE)
   public Vehicle getByID(@PathVariable Long id) {}


   @PostMapping(value = {"", "/"}, consumes =
MediaType.APPLICATION_JSON_VALUE, produces =
MediaType.APPLICATION_JSON_VALUE)
   public ResponseEntity<RestResponse<Vehicle>> post(@RequestBody
Vehicle vehicle) {}


   private Vehicle handlePatchChanges(Long id, Map<String, Object>
changes) throws JsonParseException {}
...
}
```

Esimerkkikoodi 6. VehicleRestController-luokka.

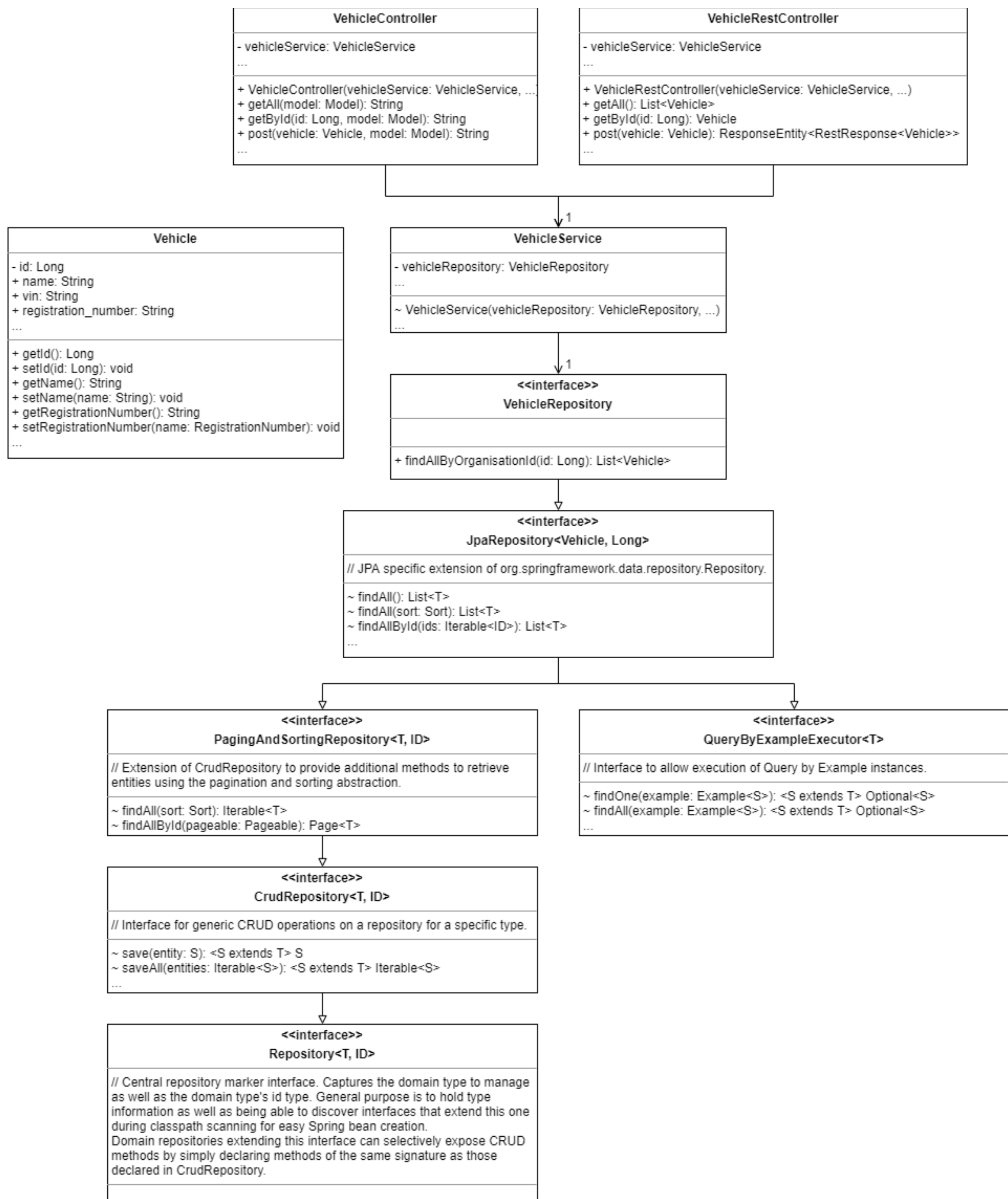
Molemmille VehicleController- ja VehicleRestController-luokille on määritetty oma @RequestMapping-annotaatio luokan yläpuolella, joka toimii pää-URL-osoitteen jatko-osana, minkä kautta päästään kyseiseen luokkaan. Jokaisella metodilla on oma annotaatio, kuten esimerkiksi @GetMapping, @PostMapping, @PutMapping, @PatchMapping tai @DeleteMapping. Jokaisella kyseessä olevalla annotaatiolla on oma URL-osoite, joka toimii jälleen jatkeena luokan RequestMapping-annotaation osoitteesta. Näiden annotaatioiden ensimmäinen osa – Get, Post… ovat HTTP- protokollan sanomatyypit, jotka määrittelevät, miten tiedot lähetetään palvelimelle ja vastaanotetaan palvelimelta. Tästä seuraa se, että tiedetään hyvin tarkasti, mikä operaatio täytyy toteuttaa, haetaanko dataa tietokannasta, vai muokataanko tai tallennetaanko dataa tietokantaan.

Pieni mutta merkittävä ero näissä luokissa on metodien paluuarvot. @Controller-luokissa kaikki paluuarvot ovat String-tyyppiä ja tästä seuraa myös se, että paluuarvona on polku haluttuun HTML-näkymätiedostoon, joka sijaitsee resources/templates-kansion juuressa. Esimerkkikoodissa 5 nähdään getAll-metodin esimerkki, jossa aluksi haetaan kaikki ajoneuvot tietokannasta, kootaan ne ”vehicles”-listaan, minkä jälkeen asetetaan meidän näkymälle ”vehicles”-attribuutille arvoksi kyseinen lista ja palautetaan ”vehicles/vehicles_list_page”-html tiedosto. @RestController-luokassa sama metodi palauttaa tietokannasta haetun ”vehicles”-listan, jota Spring kääntää JSON-muotoon.

4.7 UML-luokkakaavio

Kuvassa 6 nähdään Vehicle-, VehicleController-, VehicleRestController-, VehicleService-luokkien ja VehicleRepository-rajapinnan sekä sen periydyttyjen rajapintojen luokkakaavio. Vehicle-dataluokkaa ei ole missään luokissa konfiguroitu sen luokan muuttujaksi, vaan se toimii apuluokkana vehicle-entity-objektien kapselointiin ja sitä käytetään vain metodeissa, eli toimii niin sanottuna paikallisena muuttujana. Haettaessa tai tallentaessa tietoa tietokantaan VehicleController- ja VehicleRestController-luokat pyytävät kyseessä olevan tiedot VehicleService-luokalta, joka omalta osaltaan on yhteydessä tietokantaan eli tässä tapauksessa VehicleRepository-luokkaan. Tilan säästämiseksi luokkiin on jätetty vain niitä yhdistävät metodit ja muuttujat.

Tässä on kuvattu vain yhden Vehicle-Entity-dataluokan toteutus. Näitä Entity-dataluokkia on kyseisessä projektissa yhteensä 14 ja jokaiselle on tehty samanlainen toteutus kuten kuvassa 6. Tämä ehkä antaa paremman käsityksen ainakin yhdestä laajasta projektin osasta.



Kuva 6. Luokkakaavio, jossa on kuvattu ajoneuvoluokat sekä VehicleRepository-rajapinta ja sen riippuvuudet.

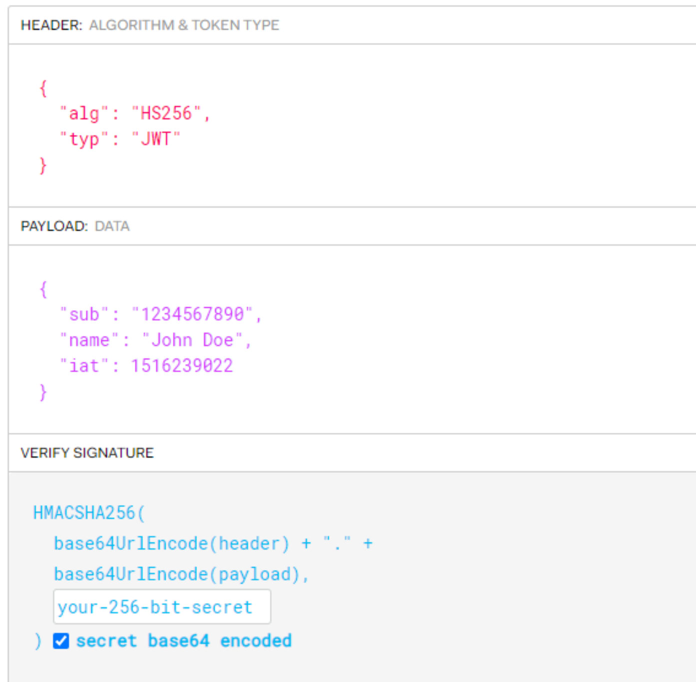
4.8 JWT

JSON Web Token on avoimen standardin menetelmä, joka tarjoaa tavan välittää tietoja eri osapuolten välillä JSON-objektin muodossa. Tätä menetelmää käytetään yleisemmin käyttäjän autentikoinnin varmistamiseen ja käyttäjän oikeuksien tarkistamiseen. Yksi suurimmista JWT:n eduista on sen keveys. Osa tarvittavasta datasta löytyy itse tietueesta, eikä käyttäjää tarvitse hakea tietokannasta uudestaan jokaisen pyynnön kanssa. Toinen on se, että se on sovellettavissa monelle ohjelmistoalustalle.

Rakenteeltaan JWT koostuu kolmesta osasta: otsikosta (header), sisällöstä (payload) ja allekirjoituksesta (signature). Otsikko sisältää tiedon Tokenin tyypistä ja salaukseen käytettävästä algoritmista, kun taas sisältö koostuu väitteistä (claims), jotka koskevat usein käyttäjän käyttöoikeuksia. Viimeisenä osana on allekirjoitus, joka koostuu salaisesta avaimesta, otsikosta ja sisällöstä. Nämä tiedot erotetaan toisistaan pisteellä ja muutetaan base64-muotoon [9]. Esimerkkikoodissa 7 nähdään, miltä JWT-merkkijono näyttää, jos rakentaisimme JWT:n kuvan 6 mukaisilla parametreilla. Kuvassa 7 nähdään koko prosessi, jossa käyttäjälle, jolla on oikeat ja voimassa olevat käyttäjänimi ja salasana, luodaan JWT-merkkijono ja palautetaan se käyttäjälle. Käyttäjän halutessa päästä palvelimen suojatuille sivuille, on annettava kyseinen JWT-merkkijono pyynnön Header-elementissä, jota tarkistetaan. Kaiken ollessa kunnossa päästetään käyttäjä eteenpäin. Juuri tällainen suojaprosessi onkin toteutettu onnistuneesti kyseisessä projektissa.


```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ3aWtpcGVkaWEiOiIxMjM0In0.cThII
oDvwdueQB468K5xDc5633seEFoqwxjF_xSJyQQ
```


Esimerkkikoodi 7. Salattu ja rakennettu JWT-merkkijono.


HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

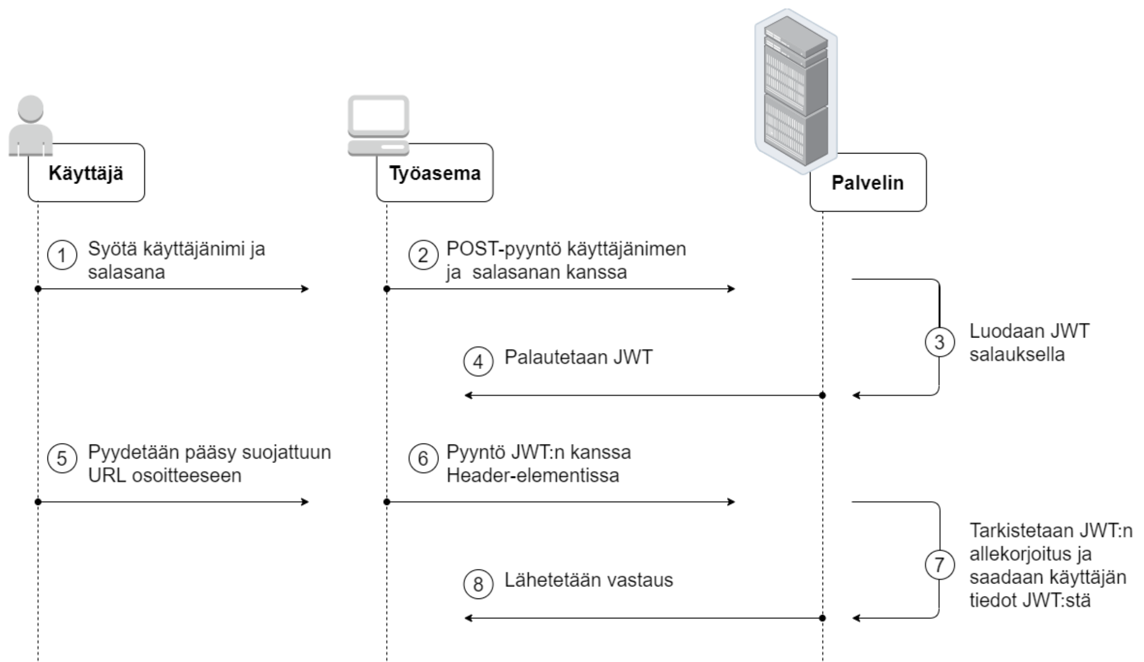
PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) secret base64 encoded
```

Kuva 6. JWT:n parametrit



Kuva 7. Käyttäjän, työaseman ja palvelimen kommunikointiprosessi.

Tässä projektissa JWT oli otettu käyttöön REST-API:n suojana. Rajapintaan voi tulla hyvin paljon erilaisia kyselyjä ja pyyntöjä, jotka voivat rasittaa palvelinta, jolla

projekti pyörii. JWT toimii tässä kevyenä, mutta silti tehokkaana autentikointiprosessina. Käyttäjän ensimmäistä kertaa kirjautuessa sisään ja kaiken ollessa kunnossa hän saa käyttöönsä yksityisen JWT-merkkijonon. Kaikissa seuraavissa rajapintakutsuissa kyseinen merkkijono täytyy olla mukana kutsun Header-elementissa avain-arvoparina, jonka voi konfiguroida halutulla tavalla. Jos autentikoinnissa oli virhe tai käyttäjä yrittää päästää suojattuun URL-osoitteeseen tai metodiin, hänelle yksinkertaisesti tulee viesti: ”Pääsy vaatii autentikoinnin” eikä hänen pyyntöään käsitellä.

4.9 Käyttöliittymä

Tässä luvussa käydään läpi projektissa kehitetty käyttöliittymä sekä vertaillaan sitä muihin olemassa oleviin käyttöliittymäratkaisuihin.

Tämän projektin lisätavoitteena, asiakkaan pyynnöstä, oli kehittää sovellukselle hyvin yksinkertainen käyttöliittymä. Lisätavoitteena tämä oli siksi, että sovellus toimii myös ilman sitä REST-rajapinnan avulla. Käyttöliittymä kuitenkin tuo aika paljon mukanaan koko projektiin, kuten esimerkiksi paremman käyttökokemuksen ja datan hahmotuksen sekä visualisoinnin. Kyseisen käyttöliittymän ei kutenkaan olisi tarkoitus olla mitenkään visuaalisesti merkittävä ja ominaisuuksiltaan laaja, sillä tämä olisi niin sanottu admin-käyttöliittymä, eikä olisi asiakkaiden käytettävissä.

Valitsemani käyttöliittymäratkaisu on nimeltään Thymeleaf. Thymeleaf on moderni palvelinpuolen Java-mallimoottori sekä verkkoympäristöihin että erillisiin ympäristöihin. Thymeleafin päätavoitteena on tuoda kehitystyönkulkuun tyylikkäät luonnolliset näkymät, jotka voidaan näyttää oikein selaimissa ja jotka toimivat myös staattisina prototyyppeinä. Thymeleaf tarjoaa Spring Framework -moduuleja, monia integraatioita erilaisiin työkaluihin ja mahdollisuuden liittää omat toiminnot, joten se on ihanteellinen nykypäivän HTML5 JVM -verkkokehitykseen [10]. Olen päätynyt kyseiseen käyttöliittymäratkaisuun suurimmaksi osaksi kahdesta syystä. Kyseinen ratkaisu ei ollut itselleni tuttu ja halusin oppia uusia menetelmiä. Toisena syynä on sen keveys ja Springin yhteensopivuus.

Alustavana ideana oli käyttää Vaadin-ohjelmistokehystä, joka mahdollistaa helpon ja nopean käyttöliittymäkehityksen, sillä se kääntää Java-koodin javascriptiksi ja html:ksi. Vaadin on edellisistä projekteista tuttu ja oikein hyvä työkalu Java-sovellusten käyttöliittämien kehittämiseen. Kuitenkin se vaatii enemmän resursseja ja on Thymeleafiä raskaampi palvelimen näkökulmasta.

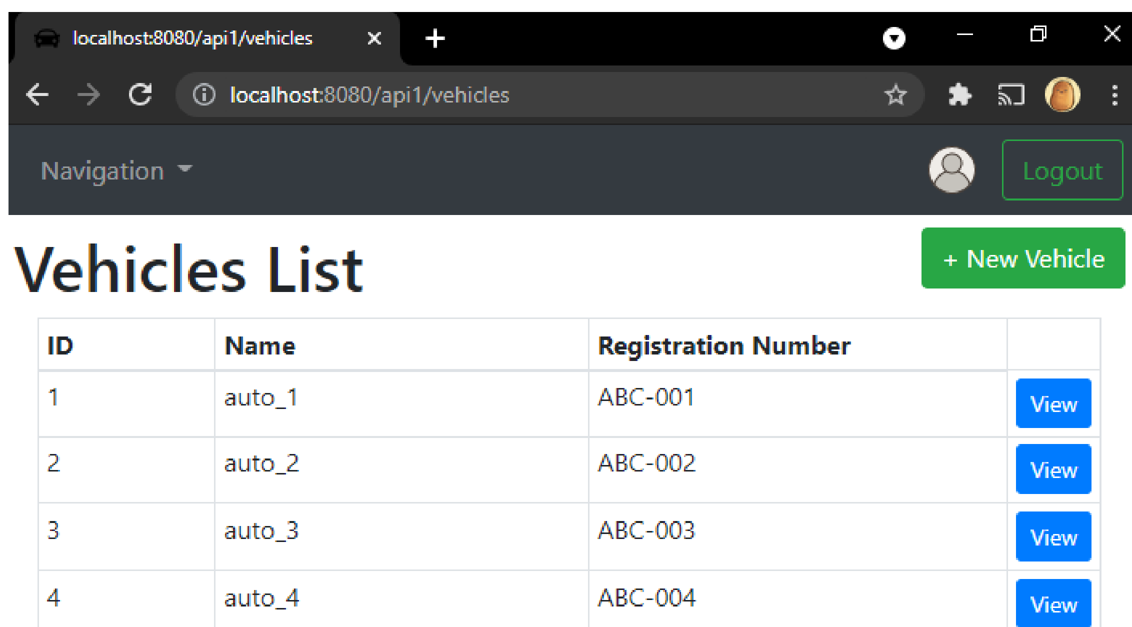
Thymeleafin idea perustuu yksinkertaiseen HTML-sivuun, jossa HTML-tunnisteissa käytetään Thymeleafin määrittelyjen mukaisia attribuutteja. HTML-tiedostoja kutsutaan sapluunoiksi (template) ja ne sijaitsevat resources/templates -kansiossa, josta Spring osaa etsiä niitä kuvan 9 mukaan getAll-metodin paluuarvosta.

Esimerkkikoodissa 8 nähdään HTML-taulu elementin tr-elementti, jossa käytetään Thymeleafin th:each-attribuuttia. Attribuutille annetaan vehicles-muuttuja, joka voidaan nähdä esimerkkikoodissa 5 getAll-metodin sisällä. Kyseinen vehicles-muuttuja pitää sisällään listan kaikista tietokannasta saaduista ajoneuvoista. Kyseinen lista välitetään siis attribuutin muuttujana, Thymeleaf rakentaa jokaisesta listan ajoneuvosta oman tr-elementin lopulliseen käyttöliittymäsivuun, jossa nähdään halutut ajoneuvon parametrit. Tässä tapauksessa ne ovat id, name ja registration_number. Koska tämä on ainoa käyttöliittymälogiikka, tässä ei käytetty erillistä javascriptiä, kyseinen käyttöliittymäratkaisu onkin hyvin kevyt ja nopea. Kuvassa 8 on esitetty esimerkkikoodin 8 rakentama html-sivu kaikilla elementeillä. Valitettavasti kyseinen ratkaisu tuo mukanaan myös esteitä erilaisille toiminnoille.

```
<tr th:each="vehicle : ${vehicles}">
   <td>
      <span th:text="${vehicle.id}"></span>
   </td>
   <td>
      <span th:text="${vehicle.name}"></span>
   </td>
   <td>
      <span th:text="${vehicle.registration_number}"></span>
   </td>

   <td style="width: 1px;">
      <form action="#" th:action="${'/api1/vehicles/' + vehicle.id}"
class="m-0 p-0" method="GET">
         <button type="submit" class="btn btn-primary btn-
sm">View</button>
      </form>
   </td>
</tr>
```

Esimerkkikoodi 8. Esimerkkikoodi Thymeleaf tr -elementistä.



Kuva 8. Käyttöliittymäesimerkki ajoneuvojen kokoelmasta.

Kyseinen ratkaisu on hyvä silloin, kun käyttöliittymän tarkoitus on olla hyvin yksinkertainen. Esimerkkiesteenä voi olla dynaamisesti latautuva sisältö. Jos meillä on kaksi alasvetovalikkoa, joista toisen halutaan näytettävän arvoja riippuen ensimmäisen alasvetovalikon arvosta, kyseinen toiminto ei ole helposti

tehtävissä ilman erillistä javascriptiä. Onneksi kyseisessä projektissa ei tarvinnut käyttää monimutkaista käyttöliittymälogiikka, vaan kaikki toiminnallisuus oli rajoittunut Thymeleafin mahdollisuuksiin. Tähän verrattuna esimerkiksi Vaadin [11] tarjoaa paljon enemmän toiminnallisuutta, yllä mainittu este olisi helposti ratkaistavissa.

Alustavana ideana olikin ottaa Vaadin käyttöön tässä projektissa ja sillä rakentaa kaikki käyttöliittymät. Vaadin on siinä hyvä, että se mahdollistaa hyvin helposti, jos ohjelmoija osaa Java-kielen, rakentaa kaikki näkymät Javalla, minkä jälkeen se kääntää Java-koodin html:ksi ja javascriptiksi itsenäisesti ilman erillistä konfiguraatiota. Tätä voi miettiä toisellakin tavalla – meillä on kaksi Java-luokkaa, joista yksi on @Controller, mitä on käyty ennen läpi ja toinen on näkymäluokka. Koska molemmat on kirjoitettu Javalla, kehittäjällä on valtaa tehdä mitä vaan loogisia temppuja, mitä hän ikinä keksii. Tästä syystä Vaadin onkin yski parhaimmista käyttöliittymäkehitystyökaluista Java-kehittäjille. Se on hyvin helposti otettavissa käyttöön, eikä vie paljon aikaa oppimiseen. Yleensä riittää, että oppii, miten yksi komponentti toimii, minkä jälkeen huomaakin, että koko sivusto on jo valmis ja näyttää kauniilta. Itse olen käyttänyt Vaadinia edellisessä projektissa ja valitsin tälle projektille Thymeleaf-käyttöliittymäratkaisun, koska se oli tuntematon ja halusin saada selville sen mahdollisuuksia ja mahdollisia rajoituksia.

4.10 Testit

Ohjelman testaus on hyvin tärkeä kehitysprosessin osa. Testien määrittely auttaa sekä nopeuttaa ohjelman virheiden löytämisen ja korjaamisen, ainakin silloin, kun testit toimivat oikein. Tässä projektissa päätin testata ohjelman tärkeimpiä osa-alueita, jotka olivat RestController-luokat ja kaikki REST-rajapintakutsut, niihin liittyvä logiikka sekä tietokantaoperaatiot. Kyseiseen tehtävään sopivat oikein hyvin integraatiotestit [12].

Testien ideana on testata kaikki kontrolleriluokassa olevat metodit, mahdollisuuksien mukaan, kaikilla datakombinaatioilla. Näin saadaan simuloitua käyttäjän käyttäytymistä ja vuorovaikutusta sovelluksen kanssa ja käydään läpi kaikki mahdolliset skenaariot kaikilla mahdollisilla paluuarvoilla. Tähän Springillä on olemassa valmiit työkalut, joita otin käyttöön.

Esimerkkikoodissa 9 nähdään testiluokka muutamalla Post-metodilla sekä testausta varten käyttöönotetut tärkeimmät työkalut. MockMVC-luokka on osa Spring MVC -testikehystä, joka auttaa testaamaan ohjaimia nimenomaisesti käynnistämällä Servlet-säilön. Testit voidaan suorittaa käynnistämättä koko sovellusta sekä http-palvelinta, mikä huomattavasti nopeuttaa koko prosessia. Samalla otetaan käyttöön meidän ajoneuvoja sisältävä tietokantaluokka: VehicleRepository. Sitä kuitenkaan ei tuoda samalla tavalla, kuten MockMvc:tä @Autowired-annotaatiolla, vaan @MockBean-annotaatiolla.

```
@SpringBootTest
@AutoConfigureMockMvc
@TestInstance(PER_CLASS)
class VehicleRestControllerTest {

   @Autowired
   private MockMvc mockMvc;

   @MockBean
   private VehicleRepository vehicleRepository;

   private final String ENTITY = "vehicle";
   private final String URL = Constants.JSON_API + "/vehicles";
   private String JWT_TOKEN = "";

   @Test
   public void testPostList_NullElementAndNonNullElement() throws
Exception {}

   @Test
   public void testPostList_InternalServerError() throws Exception {}

   @Test
   public void testPost_Null() throws Exception {}

   @Test
   public void testPost_NonNullElementWithEmptyString() throws
Exception {}

   @Test
   public void testPost_NonNullElement() throws Exception {}

   @Test
```

```
   public void testPost_InternalServerError() throws Exception {}

   @Test
   public void testPost_ById() throws Exception {}
...
}
```

Esimerkkikoodi 9. VehicleRestControllerTest-luokka muutamalla testimetodilla.

Jos käyttäisimme @Autowired-annotaatiota, ottaisimme käyttöön oikean tietokannan, jossa on oikeat ajoneuvot. Testejä varten tämä on huono asia, koska voimme tehdä muutoksia varsinaiseen tietokantaan. Tähän tuleekin avuksi MockBean-annotaatio, joka luo testin ajon ajaksi tyhjän ajoneuvojen tekotietokannan, jota voidaan käyttää ongelmitta.

Esimerkkikoodissa 10 nähdään, miten testimetodi rakentuu ja mitä vastauksia rajapinnalta odotetaan. Jos kaikki .andExpect-metodit ovat toteutuneet, testi katsotaan läpäistyksi. Jos yksikään ei toteudu, aletaan miettiä, missä on virhe. Onko se testissä vai varsinaisessa sovelluksessa?

```
@Test
public void testPost_NonNullElement() throws Exception {
   Vehicle toBeSaved = getVehicle(null, "vehicle_name");

Mockito.when(vehicleRepository.save(toBeSaved)).thenReturn(toBeSaved);

   this.mockMvc.perform(MockMvcRequestBuilders
         .post(URL).header("Authorization", "Bearer " + JWT_TOKEN)
         .content(objectMapper.writeValueAsString(toBeSaved))
         .contentType(MediaType.APPLICATION_JSON)
         .accept(MediaType.APPLICATION_JSON))
         .andExpect(status().is(HttpStatus.OK.value()))
         .andExpect(jsonPath("$.http_status").value("OK"))
         .andExpect(jsonPath("$.message").value(ENTITY + " saved
successfully"))
         .andDo(print());
}
```

Esimerkkikoodi 10. Testimetodi, jossa testataan POST-metodia oikeilla tiedoilla.

5 Yhteenveto

Insinöörityössä oli tarkoitus kehittää PostgreSQL-tietokanta, tietokannan hallintajärjestelmä API-rajapinnalla sekä käyttöliittymällä. Projektin alussa, asiakkaan kanssa pidettyjen palaverien jälkeen, toimeksiantaja asetti vaatimukset kehitettävälle ohjelmistolle, joita on otettu työn kehityksessä huomioon ja joita lopputulos tyydyttää. Kaikki asetetut vaatimukset ovat nähtävissä liitteessä 1.

Projektin alussa oli arviointiprosessi työn kokonaisrakenteesta sekä käytettävistä työkaluista. Päädyttyään Javaan sekä Spring-kehykseen oli lyhyt työkalujen oppimisvaihe, minkä jälkeen työ oli nopeasti aloitettu. Spring osoittautui hyvin sopivaksi työkaluksi kyseiseen projektiin ja tarjosi paljon hyödyllisiä sekä helposti käyttöön otettavia työkaluja erilaisia prosesseja varten. Työkalujen avulla saatiin helposti kehitettyä kaikki projektin keskeisimmät komponentit yhdistettyä niitä toimivaan yhtenäisyyteen.

Työn lopputuloksena saatiin kehitettyä kaikki halutut toiminnallisuudet ja tavoitettu kaikki asettamat vaatimukset. Ohjelmisto rakentaa itsenäisesti PostgreSQL-tietokannan määritettyjen @Entity-luokkien avulla ja on käytettävissä sekä API-rajapinnan kautta että käyttöliittymän kautta.

Kyseinen projekti on tuottanut paljon uutta kokemusta ennalta tutuista sekä täysin uusista teknologioista ja työkaluista.

Lähteet

1 Vediafi Oy. Vediafi. Verkkoaineisto. <https://www.vedia.fi/fi/meista/>. Luettu 09.06.2021.

2 Vediafi Oy. Vediafi. Verkkoaineisto. <https://www.vedia.fi/fi/cvw-clean-vehicles-wizard-2/>. Luettu 09.06.2021.

3 Liikenne- ja viestintäministeriö. Verkkoaineisto. <https://www.lvm.fi/-/direktiivi-puhtaustavoitteet-julkisten-hankintojen-ajoneuvoille-1012283>. Luettu 09.06.2021.

4 Valtioneuvoston kanslia. Verkkoaineisto. <https://vnk.fi/-/10184/julkisia-ajoneuvohankintoja-koskevan-lainsaadannon-valmistelu-etenee-vahapaastoisia-ajoneuvoja-ja-palveluita-suomeen>. Luettu 09.06.2021.

5 GitHub Inc. GitHub. Verkkoaineisto. <https://github.com/>. Luettu 09.06.2021.

6 Gradle Inc. Gradle. Verkkoaineisto. <https://gradle.org/>. Luettu 09.06.2021.

7 Pivotal Software, Inc. Verkkoaineiso. <https://start.spring.io/> Luettu 15.06.2021.

8 JetBrains. IntelliJ IDEA Community Edition 2020.3.2 x64. Verkkoaineisto. <https://www.jetbrains.com/idea/>. Luettu 15.06.2021.

9 Introduction to JSON Web Tokens. Verkkoaineisto. <www.jwt.io/introduction>. Luettu 05.07.2021.

10 The Thymeleaf Team. Thymeleaf. Verkkoaineisto. <https://www.thymeleaf.org/>. Luettu 06.07.2021.

11 Vaadin. Vaadin. Verkkoaineisto. <https://vaadin.com>. Luettu 20.08.2021

12 Baeldung. Testing in Spring Boot. Verkkoaineisto. <https://www.baeldung.com/spring-boot-testing>. Luettu 07.07.2021.

Asiakkaan määritetyt vaatimukset

- API
  - Support 5 http methods
    - https://www.restapitutorial.com/lessons/httpmethods.html
  - Support error codes
    - 200 OK
    - 400 Bad Request
    - 405 Method Not Allowed
    - 403 Forbidden
    - 404 Not Found
    - 500 Internal Server Error
  - Endpoints
    - Content type is application/json
    - keys are snake case
    - When request is sent response must contain at least same data that was sent (might more)
    - Endpoints should accept and return single and multiple objects (can be separate endpoints)
    - single object in multiple list should be created, updated, deleted, retrieved atomic
  - Validation
    - There must be validation of all incoming data
    - Give sane and descriptive validation error responses

            - Each error should contain message and error key
            - Structure response JSON same way as structure of request JSON
            - If there's a generic object error place it in that object in response
            - If there's a generic error place it in response
        - It's API responsibility sanitize and to accept only sane and valid data
    - Documentation
        - Use OpenAPI schema https://swagger.io/specification/
        - Present nicely (preferable swagger way)
    - Authorization
        - Bearer Authentication
        - Preferable JWT token
    - Tests
        - Automated tests
        - Test only your own endpoints and/or unit test pieces of code
        - Test all endpoints (some more than the others)
        - Mock all external services if you need them during tests
        - Fixtures for tests and for other developers, with fake data
        - Some default users with same simple password
        - Some data to play with, that is also useful for tests
    - Database

- Use database migration library
- Keep history of database changes in repo (migrations scripts for library)
    - So database can be destroyed, schema migrated from nothing to most recent version and populated with fixtures
- PostgreSQL
- Foreign keys, use them
- Cascade on delete, use them
- Don't keep same data in two different tables just for sake of less complex join
- If you need same data in two different places, you probably doing something wrong
- You can allow NULL, but you can't allow empty strings in varchar columns
- Time zone UTC
- Don't use PostgreSQL Enum, Array types, if you really must, use JSON, but avoid it, if possible (we're not doing NOSQL database and it's a nightmare to keep it sane)