Linja-autojen lähtökuittausjärjestelmä

Janica Fagerblom

Haaga-Helia ammattikorkeakoulu
Amk-opinnäytetyö
2021
Tradenomin tutkinto

Tiivistelmä

Tekijä(t)
Janica Fagerblom

Tutkinto
Tradenomi

Raportin/Opinnäytetyön nimi
Linja-autojen lähtökuittausjärjestelmä

Sivu- ja liitesivumäärä
43 + 4

Tämän työn tarkoituksena oli toteuttaa web-pohjainen kuittausjärjestelmä, joka palvelee bussiyrityksen tarpeita, tehostaa työntekoa ja keskittää resurssit. Kuittausjärjestelmä ottaa kuljettajilta ajosarjojen kuittaukset automaattisesti vastaan tekstiviestien avulla ja merkitsee järjestelmään ajosarjat kuitatuiksi. Järjestelmän käyttäjän näkymässä ajosarjan rivi muuttuu vihreäksi kuittauksen tultua, tai punaiseksi, jos ajosarjan lähtöön on alle 30 minuuttia, eikä kuittausta ole tullut.

Sovelluksen palvelinpuoli toteutettiin Spring Boot -sovelluskehystä käyttäen. Selainpuoli on sen sijaan toteutettu React-sovelluskehyksen avulla. Tietojen varastointiin käytettiin PostgreSQL-tietokantaa sekä tekstiviestien lähettämiseen ja vastaanottamiseen Twilion tarjoamaa SMS-sovellusliittymää.

Opinnäytetyön teoriaosuudessa käydään läpi ja pyritään perustelemaan projektissa käytetyt teknologiat sekä käsitellään vaatimusmäärittelyä. Osuuden tarkoituksena on antaa yleinen käsitys käytetyistä teknologioista ja järjestelmän toivotuista toiminnoista.

Toiminnallisessa osuudessa tarkastellaan palvelin- ja selainpuolen toteutusta. Luvuissa pyritään antamaan kokonaiskuva kehitysprosessista käymällä sitä vaihe vaiheelta läpi.

Viimeisessä Pohdinta-luvussa käsitellään työn lopputulosta ja pohditaan tavoitteisiin pääsemistä, jatkokehitystä, vastaan tulleita haasteita sekä opittua prosessin aikana. Valmiin järjestelmän toiminta vastaa vaatimusmäärittelyssä esitettyjä vaatimuksia.

Asiasanat
Ohjelmointi, Spring Boot, React.js, PostgreSQL, SMS-sovellusliittymä, kuittausjärjestelmä

Sisällys

1 Johdanto 1
2 Käytetyt ohjelmistot ja tekniikat 3
2.1 Ohjelmointiympäristo 3
2.2 Palvelinpuolen ohjelmointitekniikka (Spring ja Spring Boot) 4
2.3 JWT-autentikointi 8
2.4 Selainpuolen ohjelmointitekniikka (React) 9
2.5 Tietokanta (PostgreSQL) 11
2.6 SMS-sovellusliittymä (Twilio) 12
3 Vaatimusmäärittely 14
3.1 Yleiskuvaus järjestelmän toiminnasta 14
3.2 Järjestelmän toiminnot 15
4 Palvelinpuolen toteutus 18
4.1 Spring Boot-projektin luominen 18
4.2 API-taso (API Layer) 19
4.3 Palvelutaso (Service Layer) 20
4.4 Tietojensiirtotaso (Data Access Layer) 22
4.5 Spring Security ja JWT-autentikointi 25
5 Selainpuolen toteutus (React.js) 29
5.1 Sovelluksen rakenne ja Sarjalist.js-komponentti 29
5.2 Login.js- ja Changesarjas.js-komponentti 33
5.3 User- ja Admin-roolin näkymät 35
6 Pohdinta 37
6.1 Tulokset 37
6.2 Jatkokehitys 37
6.3 Opinnäytetyöprosessin ja oman oppimisen arviointi 39
Lähteet 41
Liitteet 44
Liite 1. Tunnistetut vaatimukset 44
Liite 2. Pääluokkien kaaviot 45
Liite 3. Esimerkki järjestelmän REST API:stä 46
Liite 4. React-sovelluksen rakenne 47

1 Johdanto

Bussivarikolta työvuoronsa aloittavan linja-autokuljettajan tehtäviin kuuluu ilmoittaa bussin kunnosta sekä valmiudesta lähtöön. Joissakin pienissä linja-autoyrityksissä ei ole minkälaista digitaalista järjestelmää, jonka avulla ilmoittaminen voitaisiin automatisoida. Puolisoni on töissä juuri tällaisessa yrityksessä, ja hän kertoi, että heillä tällä hetkellä ilmoittaminen hoidetaan puhelimitse ja puheluita vastaanottava työntekijä merkitsee tiedot satunnaiselle paperille.

Haasteena siinä on se, että puheluita voi tulla samanaikaisesti paljon, eikä puheluiden vastaanottaja pysty ottamaan vastaan kuin yhden puhelun kerrallaan. Se voi viivästyttää bussien lähtöä. Linja-autokuljettajat joutuvat soittamaan myös siinä tapauksessa, kun kaikki on hyvin. Tämä voi johtaa siihen, että bussin viasta sekä bussin lähdön viivästymisestä ilmoittava kuljettaja joutuu odottamaan vuoroaan, vaikka hänen tilanteeseensa nopea reagointi juuri olisikin hyvin tärkeää yrityksen toiminnallisuuden kannalta.

Kolmantena ongelmana on se, että puheluita vastaan ottavat työntekijät vaihtuvat pitkin päivää. Jokainen merkitsee oman vuoronsa aikana tulleet kuittaukset satunnaiselle paperille. Vuoron vaihtuessa lähtevä työntekijä ilmoittaa vuoroon tulevalle missä mennään, ja tämä aloittaa työt omien muistiinpanojensa kanssa. Tässä tilanteessa voi olla vaarana se, että yhtenäisen kuittausten merkitsemis- ja seuraamissysteemin puuttuminen voi esimerkiksi johtaa joidenkin lähtöjen hukkaamiseen, jos kukaan ei huomaa niiden puuttumista, tai päinvastoin kuittausten tuplatarkistamiseen, joka vie resursseja ja aikaa.

Yllä mainittujen haasteiden takia minulla syntyi idea toteuttaa järjestelmä, joka voisi auttaa hoitamaan kuittaukset suurilta osin automaattisesti, ja joka tarjoaisi reaaliaikaisen kuittausten seurannan. Järjestelmä ottaisi kuljettajilta ajosarjojen kuittaukset automaattisesti vastaan tekstiviestien avulla, merkitsisi ajosarjat järjestelmään kuitatuiksi ja lähettäisi kuljettajille ilmoitukset onnistuneesta tai epäonnistuneesta kuittauksesta. Kuittaus tekstiviestitse olisi helpoin tapa toteuttaa kuittaukset, sillä yrityksen jokaisessa linja-autossa on puhelin. Yrityksen puhelimet eivät kuitenkaan ole yhteydessä Internettiin, joten kuittaus jonkin sovelluksen kautta ei onnistuisi.

Kyseinen järjestelmä olisi mielekäs myös liiketoiminnan kannalta, sillä se tehostaisi työntekoa sekä pienentäisi kustannuksia. Tätä järjestelmää ei toteuteta toimeksiantona. Yrityksen kanssa ei ole tehty sopimusta, joten yrityksen nimeä ei mainita.

Tällainen järjestelmä olisi todennäköisesti myös sovellettavissa muiden yritysten tarpeisiin, joten tästä kehitystyöstä voisi olla hyötyä myös muille yrityksille, joilla voisi olla tarvetta saman tyyppiselle järjestelmälle.

Tavoite ja rajaus

Työn tavoitteena on toteuttaa järjestelmä joka, ottaa kuljettajilta ajosarjojen kuittaukset automaattisesti vastaan tekstiviestien avulla, merkitsee ajosarjat järjestelmään kuitatuiksi ja lähettää kuljettajille ilmoitukset onnistuneesta tai epäonnistuneesta kuittauksesta. Järjestelmä tarjoaa reaaliaikaisen kuittausten seurannan ja mahdollisuuden ajosarjojen muokkaamiseen, lisäämiseen ja poistamiseen sekä kausien muokkaukseen. Järjestelmään toteutetaan myös käyttäjäroolit - Admin ja User, jotka määrittelevät, mihin järjestelmän toimintoihin käyttäjälle annetaan pääsy.

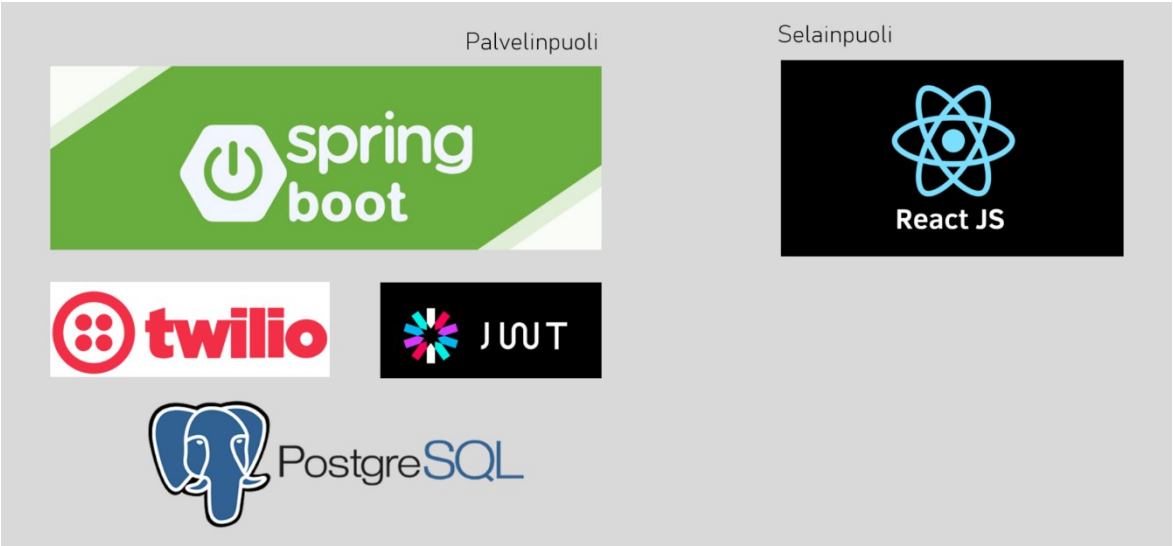
Työ on rajoitettu käsittelemään järjestelmän palvelin- ja selainpuolen toteutusta tunnistettujen käyttövaatimusten pohjalta, joten järjestelmää ei testata eikä oteta käyttöön tämän projektin puitteissa. Järjestelmän testauksen ja käyttöönoton käsittely olisi liian laaja kokonaisuus opinnäytetyön kannalta, minkä takia se on päätetty rajata pois.

Opinnäytetyön ymmärtäminen edellyttää ohjelmoinnin perusteiden tuntemusta sekä jonkin verran teknistä osaamista, sillä järjestelmän toteutusta käsitellään teknisestä näkökulmasta.

Työn peruskäsitteitä ovat:
REST (Representational State Transfer) - arkkitehtoninen tyyli hajautettujen verkkosovellusten suunnittelulle (Varanasi 2015, luku 1);
API (Application Programming Interface) - ohjelmointirajapinta, jonka avulla ohjelmistot voivat kommunikoida keskenään (Lauret 2019, luku 1);
JSON (JavaScript Object Notation) - kevyt tiedonsiirtomuoto, jota ihmisten on helppo lukea ja kirjoittaa, ja koneiden on helppo jäsentää ja luoda (Json n.d.);
JPA (Java Persistence API) - kokoelma luokkia tietojen säilyttämiseen tietokantaan (Paraschiv 2020);
Maven - tehokas projektinhallintatyökalu, jonka koontimääritystiedostona toimii *pom.xml*-tiedosto (Hinkula 2018, 8);
SMS-sovellusliittymä (SMS API) - hyvin määritelty ohjelmointirajapinta, jonka avulla koodi lähettää lyhyitä viestejä SMS-yhdyskäytävän kautta (Kamp n.d.).

2 Käytetyt ohjelmistot ja tekniikat

Tässä luvussa esitellään järjestelmän luomiseen käytettyjä teknologioita. Ohjelmointikielenä käytettiin Javaa ja ohjelmistokehityksenä toimi Eclipse-ohjelmointiympäristö. Järjestelmän palvelinpuolen toteutukseen on käytetty Spring Boot -sovelluskehystä, Twilion SMS-sovellusliittymää sekä PostgreSQL-tietokantaa. Selainpuolen tekniikkana toimi React.js-sovelluskehys. Palvelin- ja selainpuolen keskinäinen tietojen vaihto on suojattu JWT-autentikoinnilla. (Kuva 1.)



Kuva 1. Käytetyt tekniikat

Spring Boot on palvelinpuolen sovelluskehys, jonka avulla palvelinpuolta toteutetaan. Twilio on viestinnän palvelu, joka mahdollistaa tekstiviestien käytön järjestelmässä. PostgreSQL toimii järjestelmän tietokantana, ja JWT on tarkoitettu autentikointiin. React.js on sovelluskehys järjestelmän selainpuolen toteuttamiseen.

2.1 Ohjelmointiympäristö

Tämän työn ohjelmistoympäristönä toimii Eclipse IDE 2019-06. Se on ilmainen ja avoimen lähdekoodin kehitysympäristö, jota on mahdollista laajentaa monilla eri komponenteilla, kuten liitännäisillä. Vuonna 2020 Eclipse oli yksi johtavista ohjelmistoympäristöistä. Alun perin se oli kehitetty Java-sovellusten kehittämiseen, mutta nykyään se tukee myös monia muita ohjelmointikieliä kuten C/C++, PHP, JavaScript, Python, Rust yms. Eclipsen kehityksestä vastaa Eclipse Foundation -säätiö. (CodeJava n.d.; Vogella n.d.)

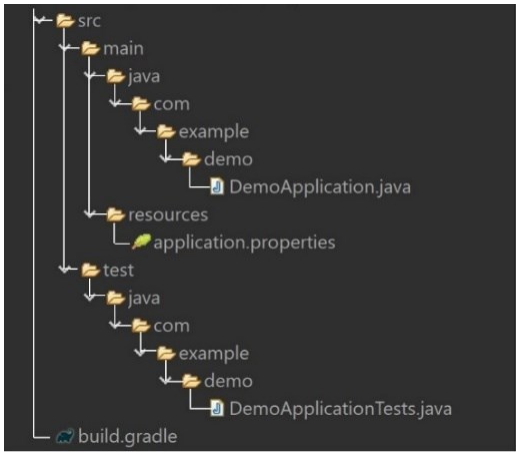
Eclipse IDE on hyvin yhteensopiva Spring Boot -sovelluskehyksen kanssa, eikä mitään ylimääräisiä lisäosia tarvita. Tästä syystä valinta ohjelmistoympäristön suhteen kyseisen projektin osalta oli helppo tehdä. Eclipseen voidaan myös niin halutessaan asentaa Spring Tool -liitännäinen, joka mahdollistaa Spring Boot -projektin luomisen itse ohjelmistoympäristössä sekä tarjoaa muita ominaisuuksia Spring Boot -sovellusten mukavampaan kehittämiseen. Tässä projektissa sitä ei kuitenkaan käytetä.

2.2 Palvelinpuolen ohjelmointitekniikka (Spring ja Spring Boot)

Spring on avoimen lähdekoodin modulaarinen sovelluskehys JVM-alustalle ja kokoelma kirjastoja, joiden ensisijainen tavoite on ratkaista yleiset ohjelmistokehitysongelmat. JVM (Java Virtual Machine) on virtuaalikone, joka ajaa Java-ohjelmia. (Oliveira 2018, luku 1.)

Spring Boot on Spring-sovelluskehyksen laajennus, joka eliminoi Spring-sovelluksen asettamiseen tarvittavat vakiokokoonpanot (boilerplate configurations) (Baeldung 2021). Spring Boot tarjoaa useamman tavan projektin luomiseen. Projektin voi luoda Springin omilta verkkosivulta Spring Initializrin avulla (https://start.spring.io/), tai omalta komentoriviltä Spring Boot CLI:llä, tai Eclipsen (tai Intellij IDEA:n) kehitysympäristössä käyttämällä Spring Tool Suitea. Projektin luominen Spring Initializrin avulla on kuitenkin helpoin tapa, sillä se opastaa valinnoissa ja tarjoaa mahdollisuuden valita kaikki tarvittavat aloituskirjastot jo projektin luomisvaiheessa. Projektin kokoamiseen Spring käyttää joko Mavenia tai Gradlea, joiden määritystietoihin tarvitaan tietyt määritykset Spring Bootin käyttämiseen. (Walls 2016, 12.)

Initializrilla luodun Gradlea käyttävän projektin rakenne näyttää seuraavalta:



Kuva 2. Initializrin avulla luodun Demo-nimisen Gradle-projektin rakenne

Build.gradle on koontimääritystiedosto, jota käytetään Gradlle-pohjaisissa projekteissa. (Demo)*Application.java* on luokka *main()*-metodin kanssa sovelluksen käynnistämistä varten. (Demo)*ApplicationTests.java* on tyhjä JUnit-testiluokka, joka on tarkoitettu Spring-sovelluskontekstin lataamiseen Spring Boot -automaattimäärityksillä. *Application.properties* on taas tyhjä määritystiedosto, johon voidaan lisätä tarvittavia määrityksiä projektia varten, esimerkiksi tietokannan kirjautumistiedot, kuten käyttäjätunnus ja salasana. (Walls 2016, 14.)

Maven-pohjaisessa projektissa koontimääritystiedostona toimii *pom.xml*-tiedosto. Aivan kuten Gradle, Maven on tehokas projektinhallintatyökalu, mutta Maven perustuu POM:iin eli projektiobjektimalliin. Sitä käytetään projektien rakentamiseen, riippuvuuksien määrittelemiseen ja dokumentointiin. Se helpottaa ja yhtenäistää ohjelmistokehitysprosessia. POM on *pom.xml*-tiedosto, joka sisältää perustiedot projektista sekä kaikki riippuvuudet (integraatiokirjastot), jotka Mavenin tulee ladata voidakseen rakentaa projektin. *Pom.xml*-tiedoston tulisi sisältää ainakin projektin juuri, *modelVersion*, *groupId*, *artifactId* ja *version.* (Hinkula 2018, 8.)

Spring Boot perustuu kolmeen perusominaisuuteen: yksinkertaistettu riippuvuuksien (integraatiokirjastojen) hallinta, automaattinen konfigurointi sekä yksinkertaistettu käyttöönotto (Heckler 2021, luku 1). Spring Boot sisältää yli 40 erilaista aloitusmoduulia (*starter modules*), jotka tarjoavat käyttövalmiita integraatiokirjastoja (*dependencies*) monille erilaisille kehyksille, kuten sekä relaatio- että NoSQL-tietokantayhteydet, verkkokehitys, sosiaalisten verkostojen integrointi, kirjautuminen, mallien renderointi ja niin edelleen (Antonov 2018, luku 1.). Näitä integraatiokirjastoja voidaan käyttää kokonaisuudessaan, muokata tai olla kokonaan käyttämättä. Spring Boot määrittää integraatiokirjastojen versiot, eikä ohjelmistokehittäjän tarvitse olla niistä huolissaan, toisin kuin jos rakennettaisiin projektin riippuvuudet alusta asti itse (Walls 2016, 35.).

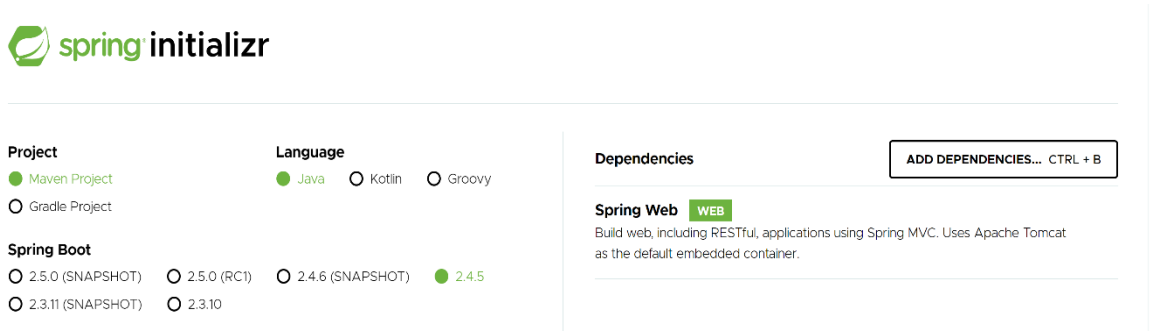
Esimerkiksi ”*spring-boot-starter-web*”-moduuli mahdollistaa Spring MVC:n, REST:n sekä Tomcat sulautetun palvelimen käytön. Pelkkä ”*spring-boot-starter-web*”-moduuli sisältää kaikki verkkokehitykseen liittyvät integraatiokirjastot, mikä vähentää myös koontimäärityksien määrää. ”*Spring-boot-starter-web*” koostuu seuraavista integraatiokirjastoista:

1. org.springframework.boot:spring-boot-starter
2. org.springframework.boot:spring-boot-starter-tomcat
3. org.springframework.boot:spring-boot-starter-validation
4. com.fasterxml.jackson.core:jackson-databind
5. org.springframework:spring-web
6. org.springframework:spring-webmvc. (JavaTpoint n.d.)

Toisin sanojen, jos halutaan integroida esimerkiksi Maven-pohjaiseen projektiin yllä mainitut integraatiokirjastot, ei tarvitse kuin lisätä ”*spring-boot-starter-web*”-moduuli projektin pom.xml-tiedostoon ja ne kaikki ovat heti käytössä. Pom.xml-tiedostossa moduulin riippuvuus näyttää seuraavalta:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Spring Initializrilla kyseisen moduulin lisääminen tapahtuu hyvin helposti projektia luodessa lisäämällä Spring Web -moduuli *Add dependencies* -painikkeesta (kuva 3).



Kuva 3. ”*spring-boot-starter-web*”-moduulin lisääminen projektiin Spring Initializrilla

”*Spring-boot-starter-web*”-moduuli myös määrittää automaattisesti seuraavat web-kehityksessä tarvittavat asiat: Dispatcher Servlet (etuohjain pyyntöjen vastaanottamiseen ja delegointiin sovelluksen muille komponenteille varsinaista käsittelyä varten), Error Page, Web JAR (staattisten riippuvuuksien hallitsemiseen) sekä Embedded servlet container (upotettu palvelinkomponentti) (JavaTpoint n.d.).

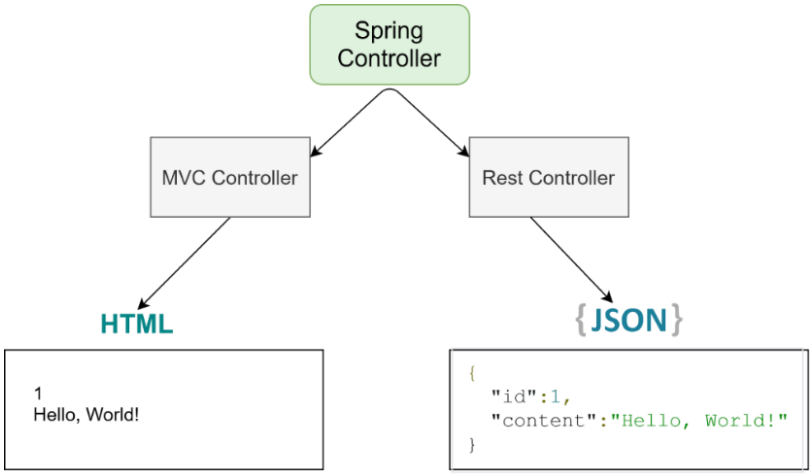
Spring Bootin vielä yksi olennainen osa on automaattiset määritykset, jotka se sisältää. *@SpringBootApplication*- sekä *@EnableAutoConfiguration*-annotaation ansiosta Spring Boot voi päättää itse, mitä sovelluksen ajamiseen tarvitaan. Tämän ansiosta suuren osan vakiokokoonpanojen (boilerplate configurations) määrittelemisen tarve poistuu, mikä on suuri etu verrattuna perinteisiin Spring-sovelluksiin. (Walls 2016, 22.)

Spring Boot tarjoaa myös muutaman tavan ottaa sovellus käyttöön. Spring-sovelluskehyksen perinteinen sovellusten tekemisen tapa on WAR-paketti Mavenilla tai Gradlella, mikä ulottuu myös Spring Bootiin. Tätä tapaa on hyvä käyttää, jos sovelluksen on tarkoitus

pyöriä pilvipalvelussa tai Java-palvelimella. Spring Boot CLI:llä, Mavenilla tai Gradlella Spring Boot -sovelluksesta voi myös tehdä JAR-sovelluksen. Tätä tapaa on hyvä käyttää esimerkiksi ns. konttiympäristöihin, kuten Docker, tai pilvipalveluihin. Kolmas tapa on tehdä Spring Boot -sovelluksesta Groovy-skripti. (Walls 2016, 161–162.)

Spring Boot valikoitui tämän projektin palvelinpuolen tekniikaksi juuri yllä mainittujen ominaisuuksien takia, joiden ansiosta projekti olisi mutkatonta luoda, hallita ja kehittää. Lisäksi Spring Boot tukee RESTful-tapaa luoda sovelluksia. Se on kriittinen vaatimus tässä työssä, sillä järjestelmän palvelin- ja selainpuoli yhdistetään REST API:n avulla. API (Application Programming Interface) on ohjelmointirajapinta, jonka avulla ohjelmistot voivat kommunikoida keskenään (Lauret 2019, luku 1). REST (Representational State Transfer) on taas arkkitehtoninen tyyli hajautettujen verkkosovellusten suunnittelulle (Varanasi 2015, luku 1). Verkkopalvelua, joka noudattaa REST:n arkkitehtonista tyyliä, sanotaan RESTful-palveluksi.

HTTP-pyynnöt sekä Springissä että Spring Bootissa käsitellään Controller-luokan avulla, joka voi olla joko REST Controller tai MVC Controller -luokka. Springin RESTful-verkkopalveluissa HTTP-pyynnöt käsittelee REST Controller -luokka. MVC Controller on taas perinteinen Controller-luokka, jota käytetään kaikissa muissa Springilla luovissa MVC-sovelluksissa. (Spring 2016.) MVC (Model-View-Controller) on arkkitehtoninen malli, joka erottaa sovelluksen kolmeksi loogiseksi komponentiksi: malliksi, näkymäksi ja ohjaimeksi (Tutorialspoint n.d.). MVC Controller eroaa REST Controllerista HTTP-vastausrungon luomistavalla. REST Controller -luokka luo ja palauttaa JSON-olin, kun taas perinteinen MVC Controller -luokka tuottaa palvelimella datan HTML:ksi (kuva 4). (Spring 2016.) JSON (JavaScript Object Notation) on kevyt tiedonsiirtomuoto, jota ihmisten on helppo lukea ja kirjoittaa, ja koneiden on helppo jäsentää ja luoda (Json n.d.).



Kuva 4. Spring-sovelluskehyksen Controllerit

Spring tarjoaa myös tietoturvamoduulin nimeltä Spring Security, joka sisältää suojan yleisiltä hyökkäyksiltä sekä erilaisia ratkaisuja autentikointiin ja autorisointiin. Nämä ratkaisut ovat mukautettavissa ja laajennettavissa käyttäjän tarpeisiin, mikä on yksi Spring Securityn suurimmista eduista. (Alex, Taylor, Winch, Hillert, Grandja & Bryant 2017.)

Spring Security tarjoaa istuntopohjaisen autentikoinnin, joka on hyödyllinen perinteisille MVC-verkkosovelluksille. RESTful-sovellusten autentikointiin on hyvä käyttää JWT-pohjaista autentikointia, jos sovelluksen selainpuoli toteutetaan toisen sovelluskehyksen avulla. (Milosevic n.d.)

2.3 JWT-autentikointi

JWT (JSON Web Token) on kompakti tapa toteuttaa autentikointi nykyaikaisissa verkkosovelluksissa. JWT on todella pienikokoinen, minkä ansiosta se voidaan lähettää URL-osoitteessa, POST-parametrissa tai otsikon (header) sisällä. (Hinkula 2018, 79.)

JSON-verkkotunnus sisältää kolme erilaista pisteillä erotettua osaa (kuva 5). Ensimmäinen osa on otsikko, joka määrittelee tunnuksen tyypin ja hajautusalgoritmin. Toinen osa on hyötykuorma, joka tyypillisesti todentamisen tapauksessa sisältää tietoja käyttäjästä. Kolmas osa on allekirjoitus, jota käytetään varmistamaan, että tunnusta ei ole muutettu matkan varrella. (Hinkula 2018, 79; Spilca 2020, luku 11.)

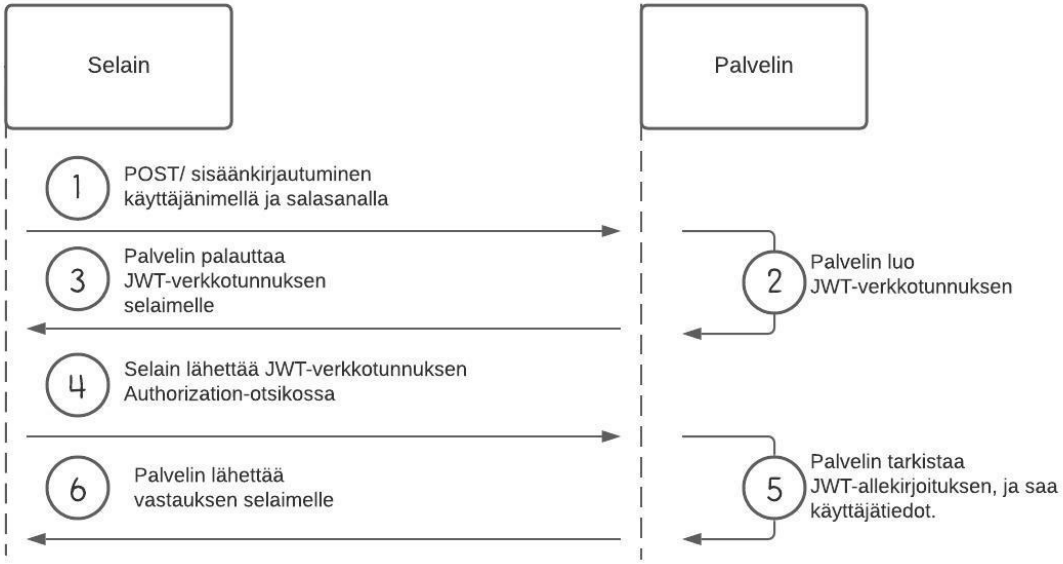

```
eyJhbGciOiJIUzI1NiJ9.
eyJzdWIiOiJKb2UifD.
ipevRNuRP6HflG8cFKnmUPtypruRC4fc1DWtoLL62SY
```


Kuva 5. Esimerkki JWT-verkkotunnuksesta (Hinkula 2018, 79.)

Milosevicin (n.d.) mukaan JWT-autentikointi on suunniteltu toimimaan seuraavasti:

1. Asiakas kirjautuu sisään lähettämällä tunnistetietonsa sovelluksen palvelinpuolelle.
2. Palvelin tarkistaa tunnistetiedot, ja jos kaikki on kunnossa, luo käyttäjätiedot ja käyttöoikeudet palvelujen käyttämiseen sisältävän JWT-verkkotunnuksen sekä asettaa sille vanhentumisajan, joka voi olla myös rajaton.
3. Tunnisteen tarjoaja (palvelin) allekirjoittaa ja tarvittaessa salaa JWT:n sekä palauttaa ja lähettää sen asiakkaalle vastauksena alkuperäiseen pyyntöön kirjautumistiedoilla.
4. Asiakas lähettää tallennetun JWT-verkkotunnuksen Authorization-otsikossa jokaisessa pyynnössä palvelimelle.

5. Palvelin ottaa jokaisen pyynnön JWT-verkkotunnuksen Authorization-otsikosta, purkaa sen sekä vahvistaa allekirjoituksen, ja jos kaikki on kunnossa, poimii käyttäjätiedot ja käyttöoikeudet. Ainoastaan näiden tietojen perusteella, ja uudelleen etsimättä lisätietoja tietokannasta, se voi hyväksyä tai hylätä asiakkaan pyynnön.
6. Palvelin lähettää vastauksen asiakkaalle. (Kuva 6.)



Kuva 6. JWT-autentikointi (mukaillen Milosevic n.d.)

Tärkein ero JWT:n ja muiden tunnusten välillä on tunnuksen sisällön standardointi. On myös suositeltavaa lähettää JWT-tunnus Authorization-otsikossa käyttämällä Bearer-skeemaa. (Milosevic n.d.)

Seuraavassa kuvassa 7 on kuvattu, miltä otsikon sisällön tulisi näyttää:

```
Authorization: Bearer <token>
```

Kuva 7. Authorization-otsikko Bearer-skeeman kanssa

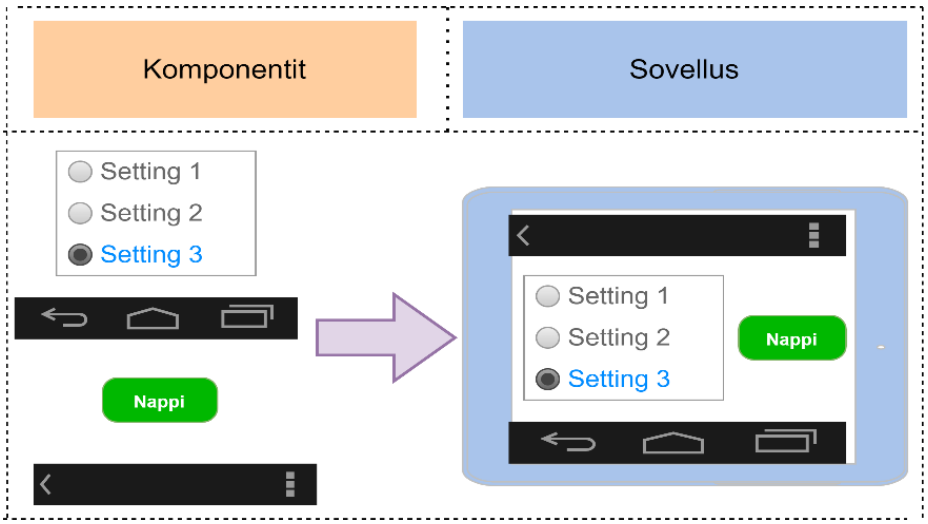
2.4 Selainpuolen ohjelmointitekniikka (React)

React on Facebookin luoma avoimen lähdekoodin JavaScript-kirjasto, joka tarjoaa uudenlaisen lähestymistavan JavaScript-käyttöliittymien rakentamiseen. Julkaisunsa jälkeen kirjasto on nopeasti saanut hyväksynnän ja luonut ympärilleen vilkkaan yhteisön. React ei sovellu palvelinpuolen koodin kehittämiseen. Sitä yleensä katsotaan vain sovelluksen näkymäkerrokseksi. (de Sousa Antonio 2015, luku 1; Boduch 2017, luku 1.)

React on parhaita selainpuolen kirjastoja. Se tarjoaa nopeamman tavan rakentaa ja ylläpitää mobiilisovelluksia. Facebookin lisäksi suuri osa tunnettuja yrityksiä, kuten Twitter, Amazon Prime, Udemy, Expo, Codesandbox, Uber ja Pinterest, käyttää React.js:tä. React mahdollistaa sen, että nämä yritykset voivat rakentaa sovelluksia käyttämällä vähemmän resursseja. (Techmagic 2017.)

React käyttää kevyttä DOM:n muistin sisäistä representaatiota nimeltä Virtual DOM. DOM:n muistin sisäisen representaation käsittely on nopeampaa ja tehokkaampaa kuin itse DOM:n käsittely. Kun sovelluksessa tapahtuu joitakin muutoksia, esimerkiksi käyttäjän vuorovaikutuksen tai tietojen hakemisen seurauksena, React vertaa uutta sisältöä jo näytettyyn sisältöön. Sen eroavuuden perusteella React laskelmoi ja suorittaa muutosten tekemiseen tarvittavat välttämättömät vaiheet. Tämä tekee Reactista erittäin nopean ja tehokkaan. React-sovellukset voivat helposti pyöriä 60 fps:n ruutunopeudella jopa mobiililaitteilla. (Griffiths 2021, luku 1.5; de Sousa Antonio 2015, luku 1.)

React on komponenttipohjainen. Komponentti on itsenäinen, tiettyyn tehtävään tarkoitettu rakennuspalikka. Se voi olla tekstiruutu, painike, lomake, ryhmä muita komponentteja ja niin edelleen. Käyttöliittymä koostuu siis useammasta komponentista (kuva 8). Sovellusten kehittäminen komponenttien avulla mahdollistaa "divide and conquer" -menetelmän, jossa minkään tietyn osan ei tarvitse olla erityisen monimutkainen. Komponentit pidetään pieninä, minkä ansiosta ne on helppo yhdistää ja luoda monimutkaisia ja monipuolisempia komponentteja pienemmistä komponenteista. React-komponenteilla on kyky esittää ajan myötä muuttuvia tietoja, ja muuttuvien tietojen visualisointi on automaattista. (de Sousa Antonio 2015, luku 1; Chiarelli 2018, luku 2.)



Kuva 8. React-sovelluksen rakentuminen komponenteista

React käyttää syntaksilaajennusta JavaScriptille nimeltä JSX, jonka avulla voidaan kirjoittaa koodiin XML:ä (ja siten HTML:ä). JSX on valinnainen, mutta se on laajalti hyväksytty standardi tapa määritellä käyttöliittymät React-komponenteissa sen deklaratiivisen syntaksin, ilmeikkyyden sekä sen takia, että se muunnetaan tavallisiksi JavaScript-funktiokutsuiksi, mikä tarkoittaa sitä, että JSX ei muuta JavaScriptin semantiikkaa. (de Sousa Antonio 2015, luku 1; Thomas 2018, luku 2.4.1.)

Reactin kanssa työskentelyä varten tarvitaan Node.js:ä ja nmp:tä. Node.js mahdollistaa JavaScript-ohjelmien suorittamisen paikallisella tietokoneella ja palvelimella avoimen lähdekoodin komentorivityökalun avulla. Nmp (Node Package Manager) on avoimen lähdekoodin ohjelmistorekisteri ja JavaScript-sovellusten riippuvuudenhallintatyökalu. Yhdessä npm:n kanssa Node.js:stä on tullut korvaamaton raskaiden JavaScript-sovellusten paikalliselle kehitykselle. Niiden avulla kehittäjä voi luoda komentosarjoja suoritettavia tehtäviä varten, esimerkiksi kopioida ja siirtää tiedostoja tai käynnistää paikallisen kehityspalvelimen sekä ladata riippuvuudet automaattisesti. (de Sousa Antonio 2015, luku 1.)

React on mainio sovelluskehys tätä työtä varten, sillä luotava järjestelmä tarvitsee toimiakseen esimerkiksi sivun jatkuvan päivittämisen tietyin aikavälein, mikä olisi mutkatonta toteuttaa Reactin avulla. Yrityksen tarve voi myös muuttua ajan kuluessa, ja järjestelmä joudutaan muuttamaan. Reactin komponenttipohjaisuuden ansiosta järjestelmään olisi helppo lisätä joitakin ominaisuuksia. Reactin tämän projektin selainpuolen ohjelmointitekniikaksi valikoitumiseen vaikutti myös se, että React on tällä hetkellä maailman suosituimpia selainpuolen sovelluskehyksiä (Simform 2021).

2.5 Tietokanta (PostgreSQL)

PostgreSQL on yritysluokan tietokanta, joka tarjoaa edistyneitä ominaisuuksia, kuten moniversioisen samanaikaisuuden hallinnan (MVCC), point-in-time-palautuksen, pöytätilat, asynkronisen replikaation, sisäkkäiset liiketoimet, online- ja pikavarmuuskopioinnin ja niin edelleen (Sharma 2021). Se käyttää ja laajentaa SQL-kieltä yhdistettynä moniin ominaisuuksiin, jotka tallentavat ja skaalaavat turvallisimmin monimutkaisimmatkin tietokuormat (PostgreSQL n.d.).

PostgreSQL on tehokas avoimen lähdekoodin olio-relaatiotietokantajärjestelmä. PostgreSQL toimii suosituimmissa käyttöjärjestelmissä – melkein kaikissa Linux- ja Unix-jakeluissa, Windowsissa ja Mac OS X:ssä. Avoimen lähdekoodinsa ansiosta tietokanta on helppo päivittää tai laajentaa. PostgreSQL:ssä voit määrittää omat tietotyypit, rakentaa

mukautettuja toimintoja ja jopa kirjoittaa koodin toisella ohjelmointikielellä (esim. Python) kääntämättä tietokantaa uudelleen. Lisäksi PostgreSQL-tietokanta on täysin ilmainen. (Sharma 2021; PostgreSQL n.d.).

PostgreSQL valikoitui tämän projektin tietokannaksi, sillä se on ansainnut vahvan maineen todistetusta arkkitehtuurista, luotettavuudesta, tietojen eheydestä, vankista ominaisuuksista, laajennettavuudesta ja avoimen lähdekoodin omistautuneesta yhteisöstä, joka tuottaa jatkuvasti suorituskykyisiä ja innovatiivisia ratkaisuja (PostgreSQL n.d.; Pirozzi, E., Ferrari L. 2020, luku 1; Sharma 2021). Lisäksi PostgreSQL on yhteensopiva Spring Boot -sovelluskehyksen kanssa ja ne on helppo yhdistää keskenään.

2.6 SMS-sovellusliittymä (Twilio)

SMS-sovellusliittymä (SMS API) on hyvin määritelty ohjelmointirajapinta, jonka avulla koodi lähettää lyhyitä viestejä SMS-yhdyskäytävän kautta. Koska tekstiviestinnän ja Internetin infrastruktuurit ovat pääosin jakautuneet, SMS-sovellusliittymiä käytetään usein teknisenä siltana teleoperaattorien tietoliikenneverkkojen ja Internetin välillä. SMS-sovellusliittymien avulla verkkosovellukset voivat lähettää ja vastaanottaa tekstiviestejä helposti tavalliseen sovelluskehykseen kirjoitetun logiikan avulla. (Kamp n.d.)

Twilio Inc. on amerikkalainen yritys, joka on tunnettuja johtajayrityksiä viestintäalustapalvelualalla. Itse Twilio on viestinnän kehitysalusta. Se tarjoaa sovelluskehittäjille tehokkaan sovellusliittymän puhelin- ja tekstiviestipalveluille, jonka avulla Twilio-ratkaisut on helppo integroida ohjelmistoihin ja ohjelmiin. Viestinnän sovellusliittymä mahdollistaa tekstiviestien lähettämisen ja vastaanottamisen, puheluiden soittamisen ja vastaanottamisen sekä keskusteluiden aloittamisen sovelluksissa tai verkkosivustoilla. (Thompson 2020; Twilio 2018.)

Twilio-ratkaisut on suunnattu niille, jotka tarvitsevat viestintäalustoja yritystarkoituksiin, kuten halvat puhelut, numeroiden ostaminen eri maihin, videoneuvottelujen pitäminen yms. Twilio voi toimia esimerkiksi toimiston puhelinkeskuksena ja helpottaa suuren määrän saapuvien puhelujen käsittelyä. Yritys tarjoaa asiakkaille mahdollisuuden keskittyä liiketoiminnan päätavoitteisiin sen sijaan, että he käyttäisivät paljon aikaa viestintäongelmien ratkaisemiseen. (Agilie n.d.) Twilion palveluita ovat käyttäneet esimerkiksi sellaisia yrityksiä kuin Uber, Airbnb, reddit ja ebay (Stackshare n.d.). Yrityksen päätuotteita ovat *ohjelmoitava ääni* (Programmable Voice), *ohjelmoitava video* (Programmable Video) ja *ohjelmoitava viesti* (Programmable SMS) (Kamp n.d.).

Twilion *ohjelmoitava viesti* -tuote (Programmable SMS) tarjoaa vakaan RESTful-sovellusliittymän ja yksinkertaiset, mutta tehokkaat ominaisuudet, jotka tuovat teleoperaattorien tietoliikenteen verkkosovelluksiin. Twilio tarjoaa molemmat API-päätepisteet, joita voidaan ohjelmoida millä tahansa ohjelmointikielellä, sekä valikoiman Helper-kirjastoja, jotka auttavat rakentamaan monimutkaisia viestintäpalveluja tuotteiksi lyhyessä ajassa. (Kamp n.d.)

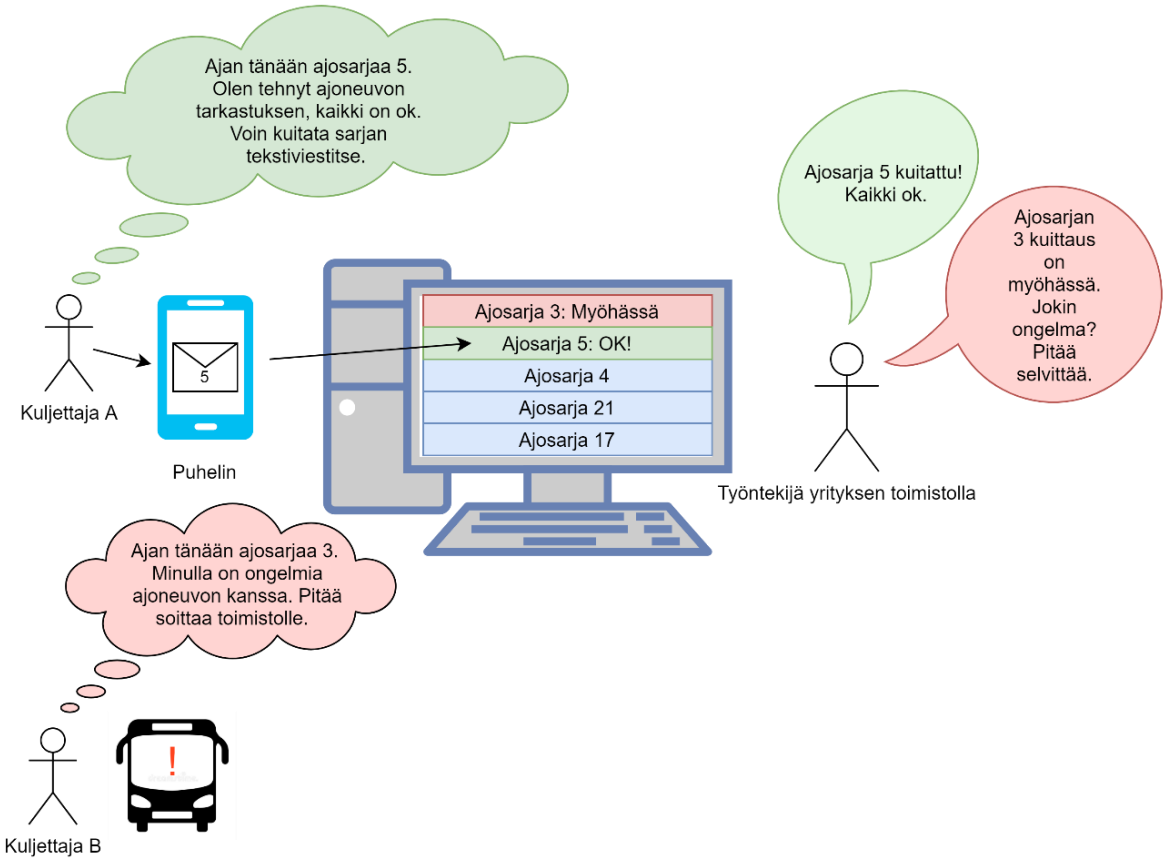
Twiliota on helppo käyttää myös pienimuotoisissa projekteissa ilman suurempia kustannuksia, minkä takia se valikoitui tämänkin projektin työkaluksi. Tässä projektissa käytetään Twilion *ohjelmoitava viesti* -tuotetta (Programmable SMS). Twilion sivuilta löytyivät tarkat ohjeet kyseisen palvelun käytöstä eri ohjelmointikielille, mikä teki palvelun integroinnista helppoa. Käyttääkseen Twilion palveluita käyttäjän tulee luoda tili. Tilin luomisen yhteydessä käyttäjä saa käyttötililleen 15 $ ilmaista rahaa, joiden tarkoituksena on auttaa käyttäjää alkuun ja tutustuttaa palveluiden käyttöön. Kokeilupakettiin kuuluu yksi numero, jonka hinta on 1 $ per kuukausi amerikkalaisesta puhelinnumerosta ja 5 $ suomalaisesta numerosta. Se laskutetaan alkusummasta. Kokeilupaketilla on paljon rajoituksia, mutta kun käyttäjä siirtää tililleen vähintään 20 $, kaikki rajoitukset poistuvat.

3 Vaatimusmäärittely

Tässä luvussa tarkastellaan järjestelmän toivottua toimintaa ja sille esitetyt käyttövaatimukset sekä kuvataan tarkemmin järjestelmän toimintoja rooleittain käyttötapauskaavioiden avulla.

3.1 Yleiskuvaus järjestelmän toiminnasta

Tavoitteena on tarjota yritykselle ja kuljettajille nopea tapa kuitata ajosarjat, kun ajoneuvon tarkastuksen yhteydessä ei ole havaittu mitään ongelmia ja se on valmis lähtöön. Järjestelmän tulee pystyä ottamaan tekstiviestejä vastaan, tunnistamaan viesteistä ajosarjat ja merkitsemään ne vihreäksi järjestelmässä. Jos ajosarjaa ei kuitata vähintään 30 minuuttia ennen sarjan alkua, sarjan tulee muuttua järjestelmässä punaiseksi (kuva 9). Kuljettajan tulee saada ilmoitus onnistuneesta tai epäonnistuneesta kuittauksesta.



Kuva 9. Järjestelmän toimintaperiaate

Järjestelmän tulee pystyä vastaamaan käyttäjien seuraaviin käyttövaatimuksiin:

1) Admin-roolin käyttäjän tulee pystyä:

a. näkemään ajosarjoja
b. lisäämään/muuttamaan/poistamaan ajosarjoja
c. merkitsemään ajosarjojen statuksen vihreäksi (kuitata)
d. päivittämään kausien voimassaoloaikaa
e. nollaamaan kuluvan päivän ajosarjojen statuksen
f. kirjautumaan sisään ja ulos

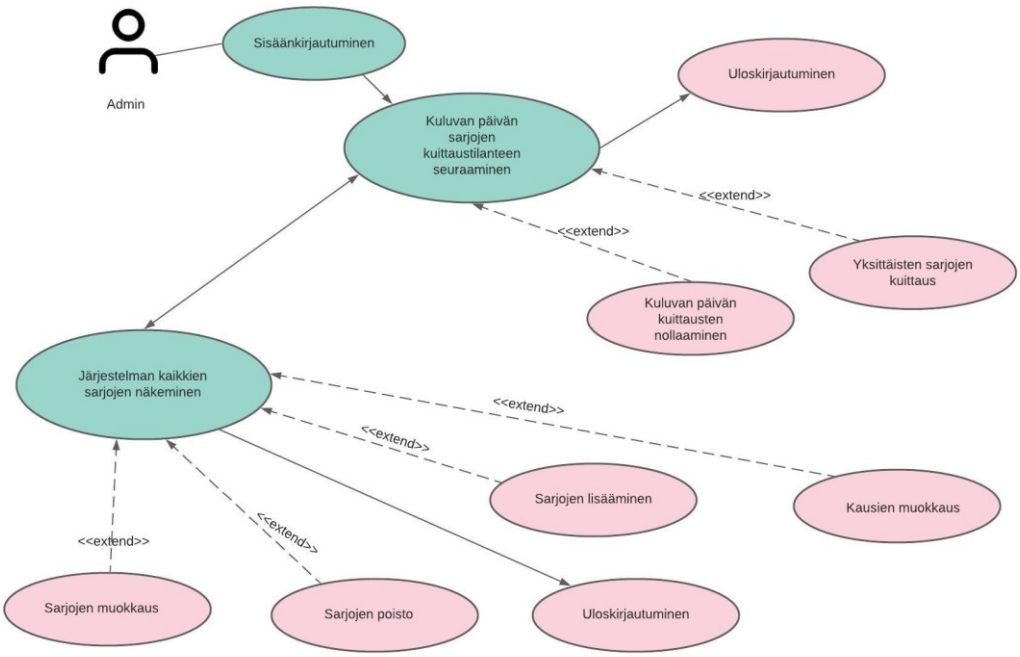
2) User-roolin käyttäjän tulee pystyä:
a. näkemään ajosarjoja
b. merkitsemään ajosarjojen statuksen vihreäksi (kuitata)
c. nollaamaan kuluvan päivän ajosarjojen statuksen
d. kirjautumaan sisään ja ulos

3) Kuljettajan tulee pystyä:
a. kuittaamaan ajosarjan omalta tai yrityksen puhelimelta tekstiviestin välityksellä.

Tarkemmat käyttövaatimukset on kuvattu liitteessä 1.

3.2 Järjestelmän toiminnot

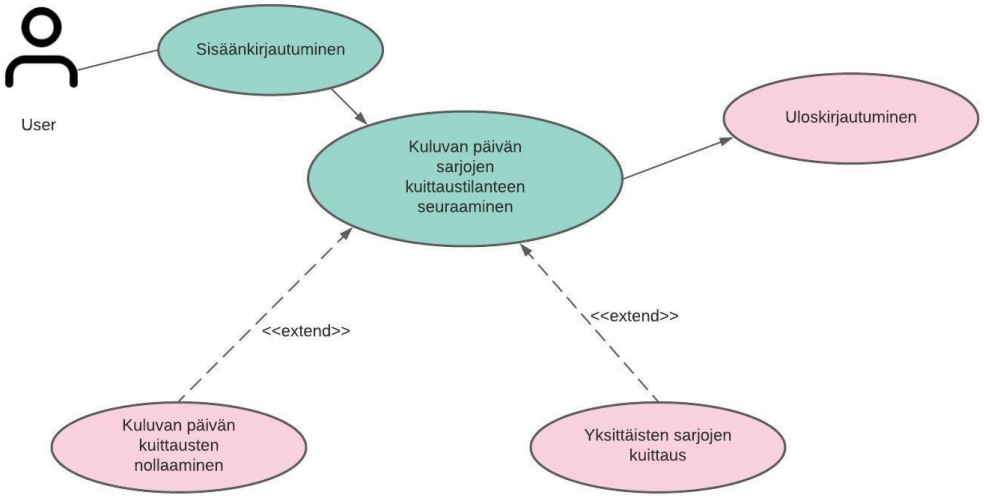
Järjestelmän toiminnot pohjautuvat tunnistettuihin käyttövaatimuksiin (liite 1). Järjestelmän tulee pystyä tarjoamaan toimintoja kahdelle eri roolille: Admin sekä User-roolille. Admin-roolin käyttäjän toiminnot on esitetty seuraavassa käyttötapauskaaviossa (kuva 10).



Kuva 10. Admin-roolin käyttäjän toiminnot

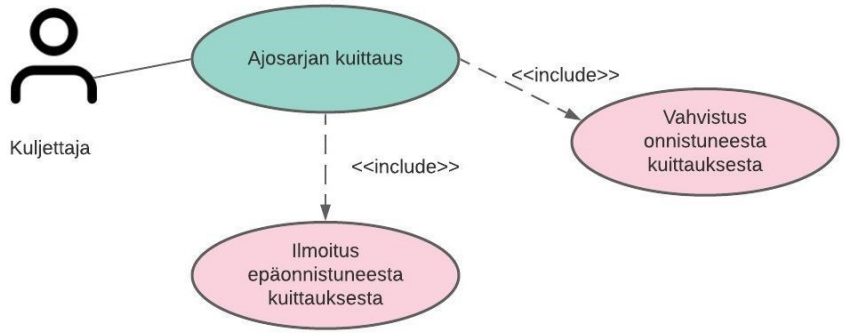
Kuvan 10 mukaisesti Admin-roolin käyttäjä voi kirjautua sisään ja ulos, nähdä kaikkia ajosarjoja ja seurata sarjojen kuittaustilannetta, kuitata yksittäisiä sarjoja, nollata kuluvan päivän kuittaukset, muokata kausia sekä lisätä, poistaa ja muokata ajosarjoja.

User-roolin käyttäjä voi kirjautua sisään ja ulos, seurata sarjojen kuittaustilannetta, kuitata yksittäisiä sarjoja sekä nollata kuluvan päivän kuittaukset (kuva 11).



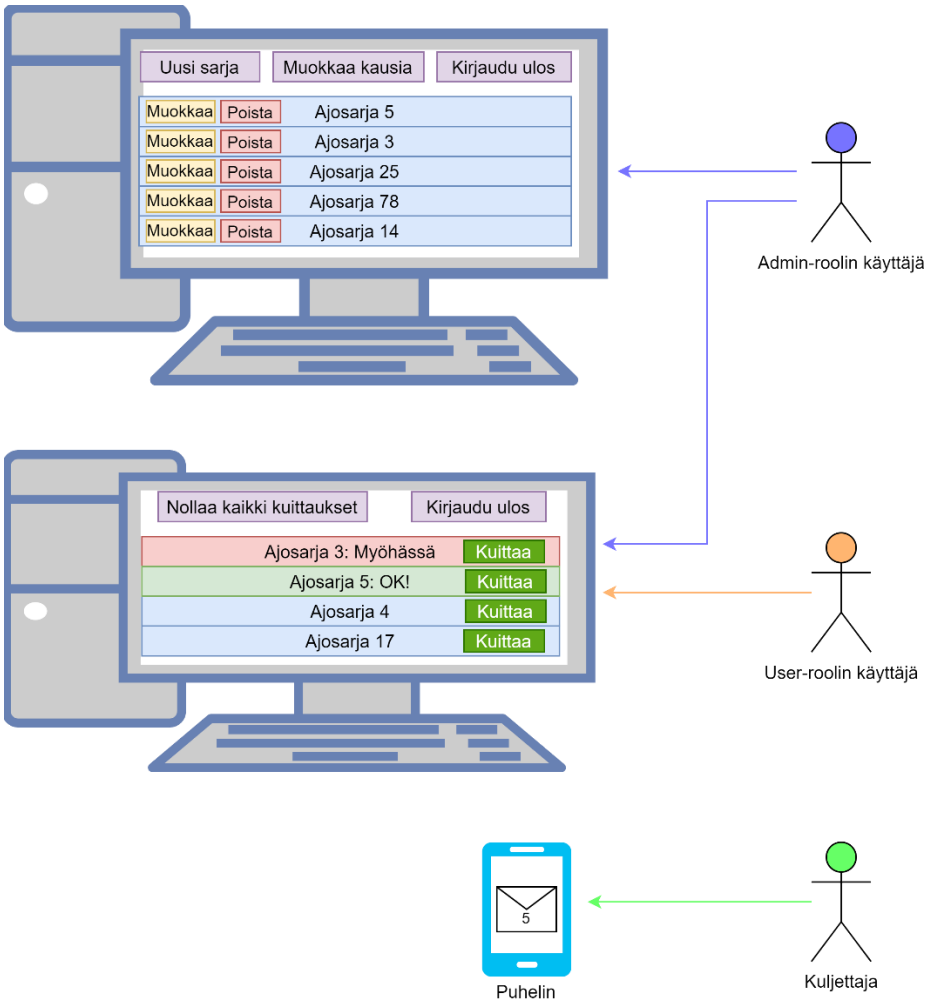
Kuva 11. User-roolin käyttäjän toiminnot

Kuljettaja voi kuitata ajosarjan, minkä jälkeen hänen on saatava ilmoitus onnistuneesta tai epäonnistuneesta ajosarjan kuittauksesta (kuva 12).



Kuva 12. Kuljettajan toiminnot

Kuvassa 13 on hahmoteltu järjestelmän toivotut toiminnot rooleittain:



Kuva 13. Toivotut toiminnot rooleittain

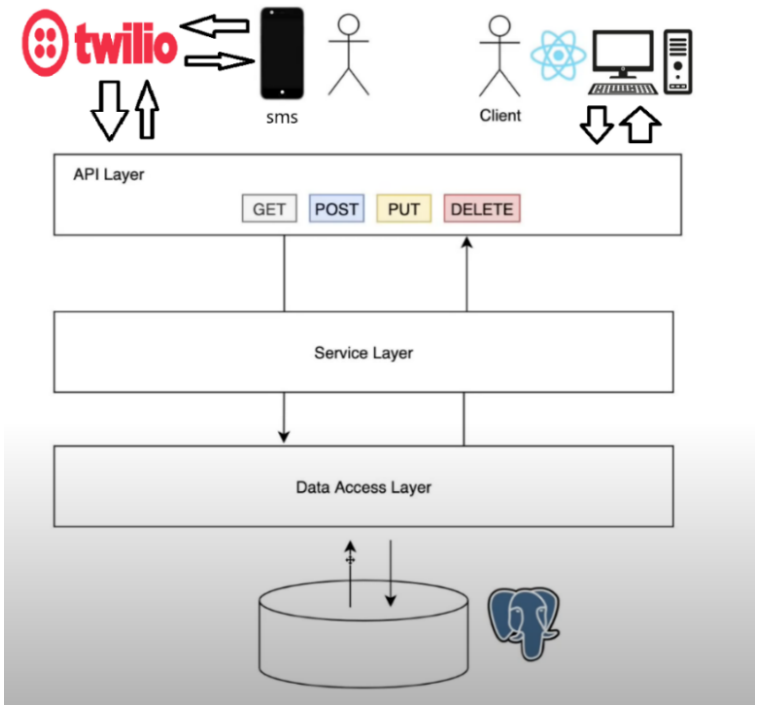
4 Palvelinpuolen toteutus

Tässä luvussa tarkastellaan järjestelmän palvelinpuolen rakennetta ja toteutusta. Järjestelmän palvelinpuoli toteutettiin Spring Boot -sovelluskehystä käyttäen Eclipse IDE -ohjelmistoympäristössä. Eclipse IDE vaatii toimiakseen Java SDK:ta (Software Development Kit, suom. ohjelmistokehityspaketti) vähintään versiota 8. Tässä projektissa käytettiin OpenJDK-ohjelmistokehityspakettia versiota 11.0.7.

4.1 Spring Boot-projektin luominen

Projektin kehitys aloitettiin luomalla Spring Boot Maven -projekti Spring Initializr -verkkosovelluksen avulla (https://start.spring.io/). Luomisen jälkeen projekti tuotiin Eclipse-ohjelmistoympäristöön, jossa jatkettiin projektin kehitystä. Järjestelmän toteutus alkoi toteuttamalla neljää pääluokkaa: Sarja, User, ValidSeason ja ValidWeekDay. Luokat sisältävät perusmuuttujat, konstruktorit sekä *get*- ja *set*-metodit (liite 2). Näiden luokkien avulla luodaan sarjojen, käyttäjien, kausien sekä viikonpäivien olioita eli luokan ilmentymiä.

Projektin palvelinpuoli on toteutettu käyttäen monitasoista arkkitehtuuria (multitier architecture tai *n*-tier architecture), mikä tarkoittaa sitä, että palvelinpuoli koostuu kolmesta tasosta: API-taso (API Layer, myös Presentation Layer), Palvelutaso (Service Layer tai Business Layer) ja Tietojensiirtotaso (Data Access Layer) (kuva 14).



Kuva 14. Järjestelmän rakenne ja toimintaperiaate

Kuten kuvasta 14 havaitaan, kuljettaja lähettää tekstiviestin, jonka vastaanottaa Twilion palvelin. Palvelin lähettää kuljettajan lähettämän tekstiviestin järjestelmälle. Järjestelmän API-taso (API Layer) vastaanottaa viestin ja lähettää sen käsittelyä varten Palvelutasolle (Service Layer). Palvelutaso käsittelee viestin, erottaa viestistä ajosarjan numeron sekä lähettää sen ja kuittaustiedot Tietojensiirtotasolle (Data Access Layer). Tietojensiirtotaso tallentaa tiedot tietokantaan. Tämän jälkeen järjestelmä lähettää vastausviestin kuljettajaa varten Twilion palvelimelle, ja palvelin lähettää vastausviestin kuljettajalle.

4.2 API-taso (API Layer)

API-taso, on taso, jonka avulla yhdistetään järjestelmän palvelinpuoli ja selainpuoli. Tämän tason avulla ne kommunikoivat keskenään ja vaihtavat tietoja. Twilion palvelin ottaa yhteyttä järjestelmään myös tämän tason kautta. API-taso tuottaa dataa JSON-muodossa (liite 3).

API-tasona toimii ohjainluokka (Controller), joka vastaanottaa ja käsittelee REST API-pyyntöjä. Toimiakseen REST-ohjainluokkana Controller-luokka vaatii *@RestConroller*-annotaation (kuva 15). Tämän lisäksi ohjainluokan metodit tarvitsevat *@RequestMapping-annotaation*, joka määrittää metodiin yhdistetyn päätepisteen. Kyseinen annotaatio käsittelee kaikkia HTTP-metodipyyntöjä (GET, POST, PUT jne.) oletusarvoisesti, mutta sallittu HTTP-metodi voidaan myös määrittää erikseen (esim. POST-metodi: *@RequestMapping(value="/endpoint", method=POST)*) (kuva 15). Siinä tapauksessa *@RequestMapping* käsittelee vain yhtä tiettyä HTTP-metodia. *@RequestMapping*-annotaation lisäksi voidaan myös käyttää annotaatioita *@PostMapping*, *@GetMapping*, *@DeleteMapping* tai *@PutMapping* HTTP-metodien määrittämiseen, jolloin erillistä HTTP-metodin määrittelemistä ei tarvita (esim. *@PutMapping("/endpoint")*). Controller-luokassa luotiin metodit tietojen hakemiseen, luomiseen, muokkaamiseen sekä poistamiseen. Controllerissa on myös metodi Twilion palvelimelta viestien vastaanottamiseen nimeltään *replyMessage* (kuva 15).

```
@RestController
public class Controller {


    @RequestMapping(value = "/replyMessage", method = RequestMethod.POST)
    public void replyMessage(HttpServletRequest request, HttpServletResponse response) throws IOException {
        twilioService.service(request, response);

    }
```

Kuva 15. Controllerin ja metodien annotaatiot

API:n palauttamassa datassa on hyvä olla linkkejä, jotka helpottavat API:n käyttöä (liite 3). Niiden avulla sidoksissa olevat datan palaset linkitetään yhteen. Linkkien toteuttamiseen on käytetty HATEOAS (Hypermedia as the Engine of Application State), joka vaatii oman riippuvuuden lisäämisen pom.xml-tiedostoon. HATEOAS tarjoaa *RepresentationModel*-luokan. Sitä laajentamalla (extends) perusluokat perivät *add()*-metodin, joka mahdollistaa linkkien lisäämisen. Controller-luokassa tarvitaan *EntityModel*- tai *CollectionModel*-mallia, jotta *add()*-metodin käyttö olisi mahdollista (kuva 16).

```
@GetMapping(value = "sarjas/{id}")
public EntityModel<Sarja> getsarjaById(@PathVariable Long id){

    List<Sarja> sarjas = new ArrayList<>();
    sarjas.addAll(repository.findAll());
    int index = 0;
    for(int i = 0; i<sarjas.size();i++) {
        if(sarjas.get(i).getId()==id) {
            index = i;
        }
    }
    Sarja sarja = sarjas.get(index);
    Long season_id = sarja.getValidSeason().getId();
    Long weekday_id =sarja.getValidWeekDay().getId();


    sarja.add(linkTo(methodOn(Controller.class).getsarjaById(id)).withSelfRel());
    sarja.add(linkTo(methodOn(Controller.class).getseasonById(season_id)).withRel("valid_season"));
    sarja.add(linkTo(methodOn(Controller.class).getweekDaysById(weekday_id)).withRel("valid_weekdays"));

    return new EntityModel<>(sarja);
}
```

Kuva 16. Esimerkki *EntityModel*-mallin käytöstä ja linkkien lisäämisestä Controllerissa

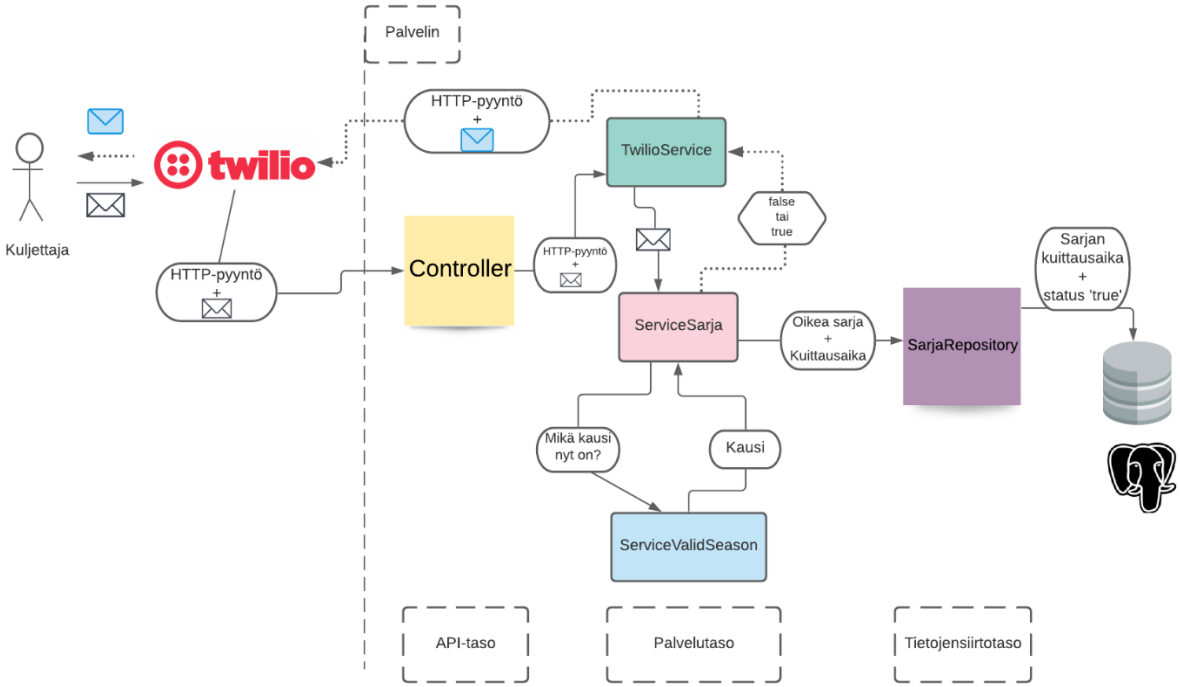
4.3 Palvelutaso (Service Layer)

Palvelutaso (Service Layer tai Business Layer) on vastuussa järjestelmän toimintalogiikasta. Service-luokissa on käytettävä annotaatiota *@Service*. Järjestelmässä on kolme Service-luokkaa (*ServiceSarja*, *ServiceTwilio*, *ServiceValidSeason*), ja ne on riippuvuusinjektoitava *@Autowired*-annotaatiolla *Controller*-luokkaan, jotta niitä voidaan käyttää (kuva 17).

```
@RestController
public class Controller {

    @Autowired
    private TwilioService twilioService;

    @Autowired
    private ServiceSarja serviceSarja;

    @Autowired
    private ServiceValidSeason serviceSeason;
```

Kuva 17. Luokkien riippuvuusinjektio Controller-luokassa

Kyseiset luokat ovat vastuussa vastaanotettujen tekstiviestien käsittelystä sekä oikeiden kausien ja sarjojen tunnistamisesta. Sarjojen status on totuusarvo eli se voi olla joko *tosi* (true) tai *epätosi*(false). Oletusarvoisesti se on *epätosi* (false). Kun sarja kuitataan, sen statuksen arvo muuttuu arvoksi *tosi* (true). Luokkien vastaanotettavien tekstiviestien käsittely on kuvattu seuraavassa kuvassa 18:



Kuva 18. Tekstiviestien käsittely

Tekstiviestien käsittely on toteutettu *TwilioService*-luokan avulla. Kun kuljettaja on lähettänyt tekstiviestin sovittuun numeroon, Twilion palvelin vastaanottaa viestin ja lähettää sen HTTP-pyyntönä järjestelmään. *Controller*-luokka ottaa HTTP-pyynnön vastaan ja siirtää sen käsittelyä varten *TwilioService*-luokalle. Se vuorollaan erottaa HTTP-pyynnöstä viestin sisällön ja lähettää sen eteenpäin *ServiceSarja*-luokalle.

ServiceSarja-luokka käsittelee tekstiviestissä kuitattua ajosarjaa. Ensiksi se lähettää pyynnön *ServiceValidSeason*-luokalle selvittääkseen, mikä kausi nyt on menossa. Ajosarjat muuttuvat riippuen kaudesta (talvi- tai kesäkausi), ja talvi- tai kesäkauden ajosarjoilla voi olla samat sarjanumerot, joilla ajosarjoja kuitataan. Tämän takia kauden tarkistus on välttämätön.

Tämän jälkeen *ServiceValidSeason*-luokka lähettää *ServiceSarja*-luokalle voimassa olevan kauden, jotta oikea sarja voidaan tunnistaa. Seuraavaksi *ServiceSarja*-luokka tarkistaa löytyykö tekstiviestissä olevalla numerolla ja voimassa olevalla kaudella ajosarjaa tietokannasta. Jos ajosarja löytyy, muuttaa luokka Tietojensiirtotason kautta sen statuksen statukseksi *tosi* (true). Tietokantaan tallennetaan myös päivämäärä sekä kellonaika, milloin tekstiviesti on vastaanotettu. Jos kuittaus onnistuu, ServiceSarja palauttaa *TwilioService*-luokalle arvon *tosi* (true). Jos taas kuittaus jostakin syystä ei onnistu, palauttaa se arvon *epätosi* (false).

TwilioService-luokka muodostaa vastausviestin *ServiceSarjan*-luokan palauttaman arvon perusteella ja lähettää sen HTTP-pyyntönä takaisin Twilion palvelimelle. Tämän jälkeen Twilion palvelin lähettää vastaanottamansa vastausviestin kuljettajalle. Jos ajosarjan kuittaus on onnistunut, vastausviestissä kuljettajalle ilmoitetaan onnistuneesta kuittauksesta. Jos ei, kuljettajaa kehotetaan yrittämään uudestaan tai soittamaan toimistolle. Twilion asetuksia mm. puhelinnumeroa ja järjestelmän URL:ä voidaan konfiguroida Twilion sivuilla oman tilin kautta.

Tekstiviestien käsittelyn lisäksi *SeasonService*-luokka osallistuu myös kuluvan päivän sarjojen statuksen nollaamiseen sekä sarjojen suodattamiseen päivän mukaan. Näistä funktioista lisää luvussa 5.

4.4 Tietojensiirtotaso (Data Access Layer)

Tietojensiirtotaso (Data Access Layer) vastaa tietojen siirrosta tietokantaan sekä niiden sieltä hakemisesta. Jotta tämä taso voidaan luoda, tarvitaan tietokanta ja siihen yhteys.

Järjestelmän tietokantana toimii paikallisesti asennettu PosgreSQL-tietokanta versiota 13.0. Pystyäkseen tallentamaan tietoja ja hakemaan niitä relaatiotietokannasta Spring Boot tarvitsee pom.xml-tiedostossa JPA-riippuvuuden ”*spring-boot-starter-data-jpa” sekä ”postgresql”*-riippuvuuden kommunikoidakseen PostgreSQL-tietokannan kanssa. JPA (Java Persistence API) on kokoelma luokkia tietojen säilyttämiseen tietokantaan (Paraschiv 2020). JPA tarjoaa olio-relaatio-mallinnuksen relaatiotietojen hallintaan JAVA-sovelluksissa (ORM), jonka toteutukseen Spring Boot käyttää oletuksena Hibernate-kirjastoa (Paraschiv 2020). Lisäksi *application.properties*-tiedostoon on määritettävä tarvittavat tiedot.

Application.properties-tiedosto sisältää tietokannan tiedot kuten salasanan, käyttäjätunnuksen ja tietokannan URL:n, jotta järjestelmä pystyy kommunikoimaan tietokannan kanssa (kuva 19). Sen lisäksi kyseisessä tiedostossa on muita tärkeitä määrityksiä tietokannan kanssa työskentelyn kannalta, kuten *hibernate.dialect*-asetus, joka kääntää yleiset sql-lauseet tietokannan ymmärtämään muotoon. *Spring.jpa.generate-ddl*-asetus määrittää pitääkö JPA:n alustaa tietokanta (true/false). *Spring.jpa.hibernate.ddl-auto*-asetus määrittää tietokannan alustuksen käyttäytymisen. Mahdollisia arvoja on *none, validate, update, create* ja *create-drop. Update*-arvo luo tietokannan ja päivittää skeeman, jos sitä on muutettu. *Spring.jpa.show-sql*-määritys mahdollistaa SQL-käskyjen kirjaamisen konsoliin. (Hinkula 2018, 31, 48.)

```
*application.properties

spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.properties.hibernate.format_sql=true
spring.datasource.url=jdbc:postgresql://localhost:5432/ack_system
spring.datasource.username=xxxx
spring.datasource.password=xxxx
spring.jpa.generate-ddl-auto=update
spring.jpa.generate-ddl = true
spring.data.rest.basePath=/api
```

Kuva 19. *Application.properties*-tiedosto

Järjestelmässä on neljä perusluokkaa: *Sarja*, *User*, *ValidSeason* ja *ValidWeekDay* (liite 2). Jotta luokista voitaisiin luoda taulut tietokantaan, luokat tarvitsevat @*Entity*-annotaation. Oletuksena on, että tietokannan taulu saa saman nimen kuin luokka, mutta käyttäen @*Table*-annotaatiota @*Entity*-annotaation lisäksi taulun nimi voidaan muuttaa (kuva 20).

```
@Entity
@Table(name="users")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name="id", nullable=false, updatable=false)
    private Long id;

    @Column(name="username", nullable=false, unique=true)
    private String username;

    @Column(name="password", nullable=false)
    private String passwordHash;

    @Column(name="role", nullable=false)
    private String role;


    public User() {

    }

    public User (String username, String passwordHash, String role) {
        this.username = username;
        this.passwordHash = passwordHash;
        this.role = role;

    }
```

Kuva 20. @*Entity*-, @*Table*-, @*Id*-, @*GeneretedValue*- ja @*Column*-annotaation käyttö

Tietokannan taulua varten on myös luotava Id-sarake. Sitä varten käytetään @*Id*-annotaatiota. Jotta perusavaimen (primary key) uniikki id-arvo voitaisiin generoida automaattisesti, tarvitaan toinen annotaatio @*GeneratedValue*. Sen yhteydessä on myös määriteltävä strategia, joka tässä järjestelmässä on AUTO (strategy=Generationtype.AUTO). Se tarkoittaa sitä, että nämä generoidut arvot ovat ainutlaatuisia tietokantatasolla, eikä niitä koskaan kierrätetä. Lisäksi @*Column*-annotaation avulla voidaan määrittää sarakkeen ominaisuuksia kuten *name*, *nullable*, *unique*, *updatable* jne. (kuva 20).

Varsinaisen tietojensiirtotason toteuttamiseksi jokainen luokka, josta luodaan taulu tietokantaan, tarvitsee lisäksi Interface-luokan, jonka nimeksi tulee *Repository*. Interface-luokat laajentavat CrudRepository-luokkaa, joka tarjoaa metodeja vuorovaikutukseen tietokannan kanssa mm. metodeja tietojen tallentamiseen, hakemiseen sekä poistamiseen. Sen parametreina käytetään luokkaa, josta on luotu taulu tietokantaan, sekä Id:tä (Id:n tyyppiä). Lisäksi Interface-luokat tarvitsevat @*Repository*-annotaation sekä kyselymetodin määrittelyn (kuva 21).

```
@Repository
public interface SarjaRepository extends CrudRepository <Sarja, Long>{
    List<Sarja>findAll();


}
```

Kuva 21. Repository-luokan määrittely

Jotta *Contoroller*-luokka pystyisi käyttämään *Repository*-luokkia, ne pitää injektoida *Controller*-luokkaan käyttämällä @*Autowired*-annotaatiota (kuva 22).

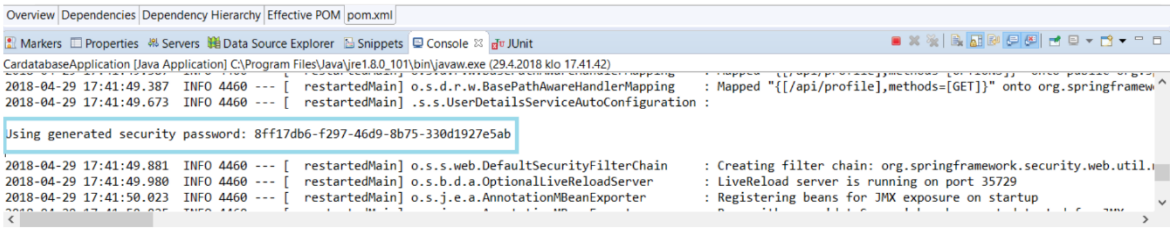
```
@RestController
public class Controller {


    @Autowired
    private SarjaRepository repository;

    @Autowired
    private ValidSeasonRepository vrepository;

    @Autowired
    private ValidWeekDayRepository wrepository;
```

Kuva 22. *Repository*-luokkien injektointi *Controller*-luokkaan

Injektoinnin jälkeen *Controller*-luokka voi alkaa käyttää *Repository*-luokkien (laajennetun *CrudRepository*-luokan) metodeja kommunikoidakseen tietokannan kanssa. Jos esimerkiksi halutaan tallentaa jokin olio tietokantaan, on käytettävä metodia *save()* (esim. *repository.save(newSarja);*). Jos taas halutaan ladata tietokannasta kaikki oliot, on käytettävä metodia *findAll()* (esim. *repository.findAll();*).

4.5 Spring Security ja JWT-autentikointi

Spring Security ja JWT-autentikointi suojaavat sovellusta luvattomalta käytöltä. Spring Securityn käyttämistä varten *pom.xml*-tiedostoon on lisättävä ”*spring-boot-starter-security*”-riippuvuus. Tämän jälkeen Spring Security on heti käytössä, eikä järjestelmää pääse enää käyttämään kirjautumatta sisään. Järjestelmän käynnistyessä konsoliin tulostetaan Spring Securityn generoima salasana (kuva 23). Käyttäjänimenä toimii *user*.



Kuva 23. Salasanan tulostus konsoliin (Hinkula 2018, 69.)

Jotta Spring Securityn käyttäytymistä pystytään konfiguroimaan, järjestelmään on luotava luokka, joka laajentaa luokkaa nimeltä *WebSecurityConfigurerAdapter*. Luotavalle luokalle annetaan nimi *WebSecurityConfig* sekä lisätään *@Configuration*- ja *@EnableWebSecurity*-annotaatiot, jotka laittavat Spring Securityn oletusmäärityksen pois päältä (kuva 24). Tämän jälkeen voidaan konfiguroida Spring Security järjestelmän tarpeisiin.

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
```

Kuva 24. *WebSecurityConfig*-luokan annotaatiot

WebSecurityConfig-luokkaan on lisättävä *configureGlobal*-metodi, jotta käyttäjät olisi mahdollista hakea tietokannasta (kuva 25). Lisäksi kyseinen metodi määrittää salasanojen hajautusalgoritmiksi BCryptin. Salasanoja ei tule tallentaa tietokantaan pelkkänä tekstinä, ja Spring Security tarjoaa useita hajautusalgoritmeja, kuten BCrypt, joiden avulla voidaan

hajauttaa salasanoja. Tämän *configureGlobal*-metodin ansiosta salasana hajautetaan BCrypt-hajautusalgoritmilla ennen sen tallentamista tietokantaan. (Hinkula 2018, 71, 76.)

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(userDetailsService).passwordEncoder(new BCryptPasswordEncoder());

}
```

Kuva 25. *ConfigureGlobal*-metodi

Lisäksi tarvitaan *UserDetailsService*-luokka, jota Spring Boot käyttää käyttäjien autentikointiin ja autorisointiin. Luokkaan pitää injektoida UserRepository-luokka, koska sitä tarvitaan käyttäjän hakemiseen tietokannasta, kun Spring Security käsittelee autentikointia. (Kuva 26.)

```
@Service
public class UserDetailServiceImpl implements UserDetailsService  {

    @Autowired
    private UserRepository repository;


    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException
    {
        User curruser = repository.findByUsername(username);
        UserDetails user = new org.springframework.security.core.userdetails.User(username, curruser.getPasswordHash(),
                true, true, true, true, AuthorityUtils.createAuthorityList(curruser.getRole()));

        return user;
    }

}
```

Kuva 26. *UserDetailsService*-luokka

Järjestelmään on myös toteuttava JWT-autentikointi, jotta järjestelmän palvelinpuolen ja selainpuolen kommunikointi olisi turvallista verkossa. Sitä varten tarvitaan JWT-kirjasto, jonka seuraava riippuvuus on lisättävä *pom.xml*-tiedostoon:

```
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt</artifactId>
  <version>0.9.0</version>
</dependency>
```

JWT-suojaus tarvitsee toimiakseen 4 luokkaa: *AccountCredentials*-, *AuthenticationService*-, *AuthenticationFilter*- sekä *LoginFilter*-luokan. *AccountCredentials*-luokassa säilytetään tunnistetiedot autentikointia varten. Luokassa on perusmuuttujat käyttäjänimelle ja salasanalle, sekä *get*- ja *set*-metodit (kuva 27).

```
public class AccountCredentials {
      private String username;
      private String password;

      public String getUsername() {
        return username;
      }
      public void setUsername(String username) {
        this.username = username;
      }
      public String getPassword() {
        return password;
      }
      public void setPassword(String password) {
        this.password = password;
      }
    }
```

Kuva 27. *AccountCredentials*-luokka

AuthenticationService-luokassa luodaan itse JWT-tunnus Bearer-skeemalla ja asetetaan sille vanhenemisaika. Tämän lisäksi luokka vastaa myös JWT-tunnuksen lisäämisestä HTTP-pyynnön otsikkoon (header) sekä sen sieltä hakemisesta.

AuthenticationFilter- ja *LoginFilter*-luokat ovat suodatusluokkia sisäänkirjautumista sekä autentikointia varten. *LoginFilter*-luokka käsittelee POST-pyynnöt */login*-päätepisteeseen, kun taas *AuthenticationFilter*-luokka on vastuussa kaikkien muiden pääpisteiden autentikoinnista.

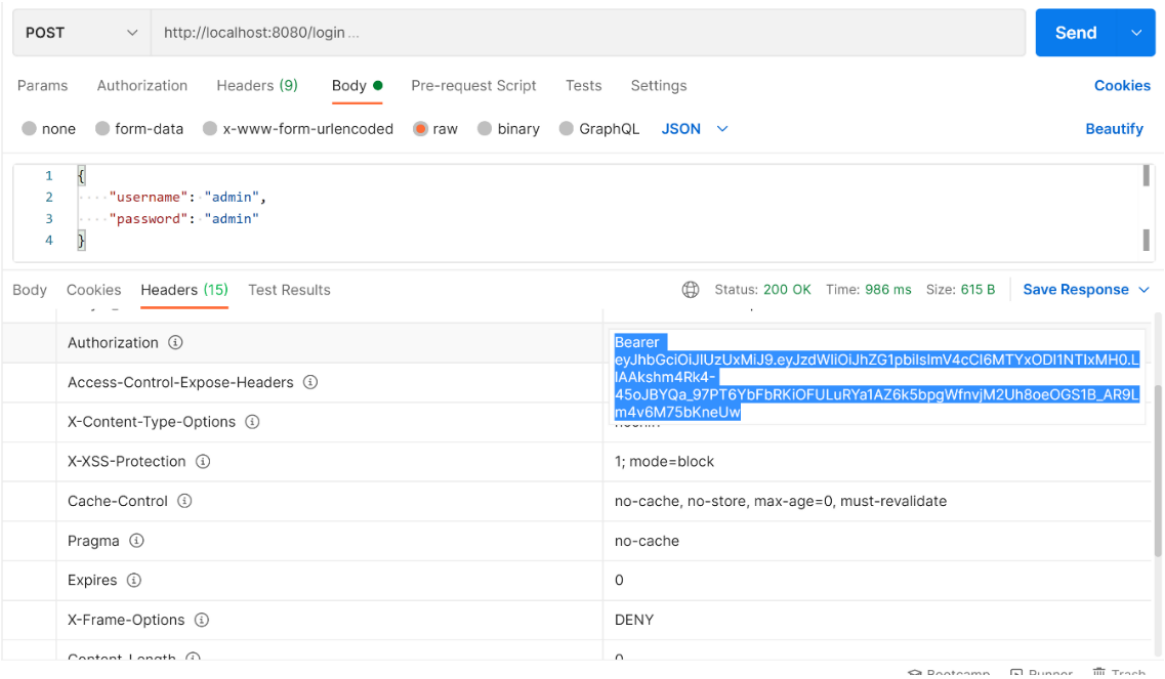
Login-sivulle on päästävä ilman autentikointia, jotta sisäänkirjautuminen olisi mahdollista. Tämän takia *WebSecurityConfig*-luokan *configure()*-metodissa on määriteltävä, että POST-pyyntö */login*-päätepisteelle sallitaan ilman autentikointia (komennolla .antMatchers(HttpMethod.POST, "/login").permitAll()), ja kaikille muille päätepisteille lähetetyt pyynnöt vaativat autentikoinnin(komennolla .anyRequest().authenticated()). (Kuva 28.)

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private UserDetailServiceImpl userDetailsService;

    @Override
    protected void configure(HttpSecurity http) throws Exception {

     http.authorizeRequests().antMatchers("/replyMessage").permitAll();
     http.cors().and().authorizeRequests()
      .antMatchers(HttpMethod.POST, "/login").permitAll()
          .anyRequest().authenticated()
          .and()
          // Filter for the api/login requests
          .addFilterBefore(new LoginFilter("/login",
           authenticationManager()),
                  UsernamePasswordAuthenticationFilter.class)
          // Filter for other requests to check JWT in header
          .addFilterBefore(new AuthenticationFilter(),
                  UsernamePasswordAuthenticationFilter.class);
      http.csrf().disable();
    }
```

Kuva 28. *WebSecurityConfig*-luokan *configure()*-metodi

Järjestelmässä on yksi päätepiste, jota käyttää Twilion palvelin, eikä sitä tarvitse suojata JWT-verkkotunnuksella. Twilion palvelin ei myöskään voi kirjautua järjestelmään käyttämällä käyttäjätietoja. Tämän takia päätepiste, johon se ottaa yhteyttä (”/replyMessage”), määritetään autentikointi- ja auktorisointivapaaksi seuraavasti: *http.authorizeRequests().antMatchers("/replyMessage").permitAll();* (kuva 28).
Spring Security käyttää oletuksena CSRF-suojausta, jolle ei ole tarvetta, jos käytetään JWT-autentikointia. CSRF-suojaus poistetaan lisäämällä *configure()*-metodiin *http.csrf().disable();* (kuva 28).

Tarkistamalla JWT-autentikointia Postman-ohjelman avulla, saadaan JWT-verkkotunnus näkyville pyynnön Authorization-otsikossa (kuva 29).



Kuva 29. JWT-verkkotunnus *Authorization*-otsikossa

5 Selainpuolen toteutus (React.js)

Tässä luvussa käsitellään järjestelmän selainpuolen rakennetta, toteutusta sekä komponenttien toimintaa. Selainpuoli toteutettiin React.js -sovelluskehystä käyttäen Virtual Studio Code -ohjelmistoympäristössä. Projekti luotiin Node.js:n sekä npm-ohjelmistorekisterin avulla. React-sovellus on yhdistetty palvelinpuoleen REST API:n avulla, jonka kautta sovellus lähettää tai hakee tarvittavia tietoja tietokannasta (liite 3).

5.1 Sovelluksen rakenne ja Sarjalist.js-komponentti

Järjestelmän selainpuolen sovellus koostuu yhdeksästä komponentista (liite 4). Niistä neljä ovat pääkomponentteja, joiden varaan järjestelmä rakennetaan (Login.js, Sarjalist.js, SarjalistUser.js ja Changesarjas.js). Kaikki muut ovat pienempiä komponentteja, kuten esimerkiksi nappeja.

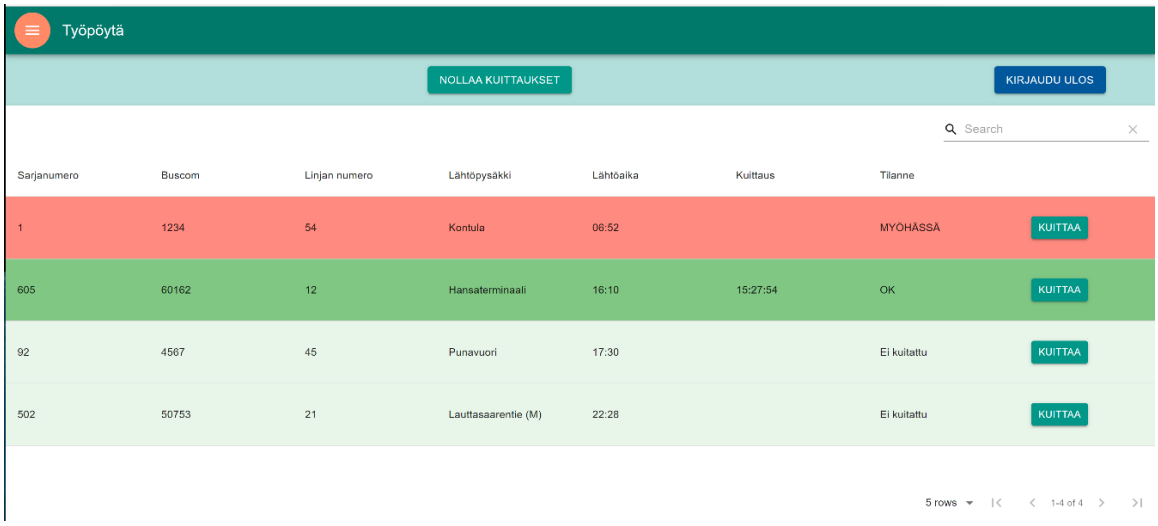
Järjestelmän pääfunktioista vastaa Sarjalist.js-komponentti, tai SarjalistUser.js-komponentti, jos käyttäjän rooli on User. Kyseiset komponentit eroavat toisistaan vain Menu-valikon verran, muuten komponentit ovat samat. Tietokannassa on ajosarjoja, joita ajetaan viikon eri päivinä. Sieltä löytyy myös sarjoja samalla sarjanumerolla, mutta eri kaudella (kesä-tai talvikausi). Sarjalist.js-komponentti (tai SarjalistUser.js-komponentti) hakee tietokannasta juuri oikeita sarjoja REST API:n avulla, järjestää niitä kellonajan mukaan sekä tarkkailee aikaa ja takistaa, onko jokin sarja jo myöhässä, tulossa kuitettavaksi vai jo kuitattu. Kyseinen komponentti myös päivittää järjestelmän joka kahdeksas sekunti, jotta se pysyy ajan tasalla.

Järjestelmän jatkuva päivitys on toteutettu Reactin *useEffect*-funktion sekä *setInterval*- ja *clearInterval*-funktion avulla (kuva 30). Tavallisesti React ajaa useEffect-funktion vain kerran sovelluksen käynnistyessä. *SetInterval*- ja *clearInterval*-funktion avulla React voidaan pakottaa suorittamaan *useEffect*-funktiota tietyin aikavälein. *SetInterval*-funktio päivittää sivun ja asettaa ajastimen. ClearInterval-funktio tyhjentää ajastimen, kun ajastettu aika on kulunut, minkä jälkeen funktion suoritus alkaa alusta. Tässä järjestelmässä optimaalinen päivitysaika on joka kahdeksas sekunti. Liian lyhyt päivitysaika voi estää joidenkin nappien virheettömän toiminnan, kun taas liian pitkä päivytysaika ei pidä järjestelmää ajantasalla.

```
useEffect(() => {
  const intervalId = setInterval(() => {
  fetchData()
  }, 8000)
  return () => clearInterval(intervalId);
});
```

Kuva 30. useEffect- ja setInterval-funktio

Kuljettajan tulee kuitata sarja vähintään 30 minuuttia ennen linja-auton lähtöä, jotta mahdollisiin ongelmiin ehditään reagoida. Jos lähtöön on alle 30 minuuttia, sarja muuttuu järjestelmässä punaiseksi. Jos lähtöön on enemmän kuin 30 min, se näkyy järjestelmässä vaaleana. Kun kuljettaja on kuitannut sarjan tekstiviestitse, sarja muuttuu vihreäksi (kuva 31).



Kuva 31. Sarjalist.js-komponentti (Työpöytä)

Sarjojen status esiintyy tietokannassa totuusarvona. Se voi olla joka *tosi* (true) tai *epätosi (false)*. Oletusarvoisesti sarjojen status on *epätosi*. Kun järjestelmä vastaanottaa viestin kuljettajalta, sarjan status muuttuu statukseksi *tosi*. Järjestelmässä kaikki laskelmoinnit ja suodatukset tapahtuvat joka kahdeksas sekunti päivityksen yhteydessä, joten jos sarjan status muuttuu, järjestelmä tunnistaa sen if-ehtolauseen avulla ja muuttaa sarjan rivin vihreäksi. Tämän jälkeen sarjan rivillä tilanteen kohdalla käyttäjä näkee sanan ”OK”. Lisäksi järjestelmä hakee kuittauksen ajan tietokannasta ja tulostaa sen kohtaan ”Kuittaus” (kuva 31).

Sovellus myös tarkistaa kuinka paljon aikaa on sarjan lähtöön. Jos status on edelleen *epätosi*, vaikka aikaa lähtöön on alle 30 minuuttia, sarjan rivi muuttuu punaiseksi, ja sarjan

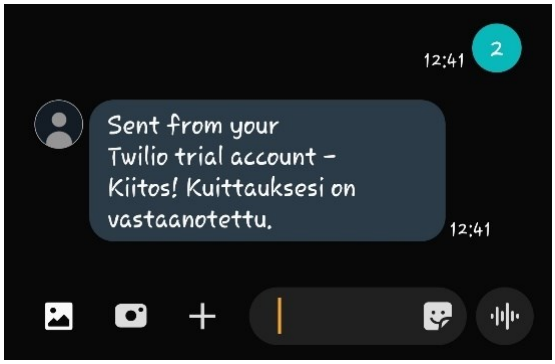
rivillä tilanteen kohdalla käyttäjä näkee sanan ”MYÖHÄSSÄ”. Jos taas aikaa sarjan lähtöön on enemmän kuin 30 minuuttia, sarja näkyy käyttäjälle vaaleana, ja sarjan rivillä tilanteen kohdalla lukee ”Ei kuitattu”. (Kuva 31.)

Seuraavassa kuvassa 32 on esitetty sarjojen suodatus ennen näytölle tulostamista statuksen ja ajan perusteella:

```
var aika = countMinutes(dataTime, time);
let stat = events._embedded.sarjas[k].status;
if(stat === true){
  sarja.push({
    id: events._embedded.sarjas[k].id,
    sarjaNumber: events._embedded.sarjas[k].sarjaNumber,
    buscom: events._embedded.sarjas[k].buscom,
    busNumber: events._embedded.sarjas[k].busNumber,
    departureStop: events._embedded.sarjas[k].departureStop,
    departureTime: events._embedded.sarjas[k].departureTime,
    msgSentTime: events._embedded.sarjas[k].msgSentTime,
    status: 'OK',
  })
} else if(aika <= 30){
    sarja.push({
      id: events._embedded.sarjas[k].id,
      sarjaNumber: events._embedded.sarjas[k].sarjaNumber,
      buscom: events._embedded.sarjas[k].buscom,
      busNumber: events._embedded.sarjas[k].busNumber,
      departureStop: events._embedded.sarjas[k].departureStop,
      departureTime: events._embedded.sarjas[k].departureTime,
      msgSentTime: '',
      status: 'MYÖHÄSSÄ'
    })
  } else {
    sarja.push({
      id: events._embedded.sarjas[k].id,
      sarjaNumber: events._embedded.sarjas[k].sarjaNumber,
      buscom: events._embedded.sarjas[k].buscom,
      busNumber: events._embedded.sarjas[k].busNumber,
      departureStop: events._embedded.sarjas[k].departureStop,
      departureTime: events._embedded.sarjas[k].departureTime,
      msgSentTime: '',
      status: 'Ei kuitattu'
    })
  }
```

Kuva 32. Sarjojen suodatus statuksen ja ajan perusteella

Jos kuljettaja on kuitannut sarjan tekstiviestin avulla, ja järjestelmä on onnistunut löytämään ja kuittaamaan kyseisen sarjan, saa kuljettaja paluutekstiviestinä ilmoituksen kuittauksen vastaanottamisesta (kuva 33). Muussa tapauksessa kuljettaja saa kehotuksen kokeilemaan uudestaan tai soittamaan toimistolle.



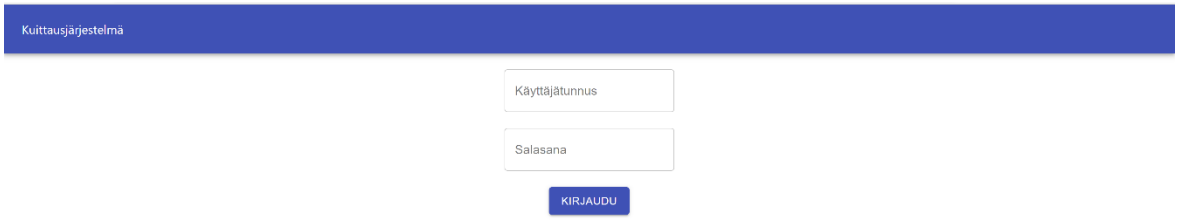
Kuva 33. Esimerkki lähetetystä tekstiviestistä ja ilmoitus onnistuneesta kuittauksesta

Tässä työssä käytetään Twilion maksutonta versiota, minkä takia viestissä lukee myös ”Sent from your Twilio trial account”. Jos omalle Twilio-tilille siirretään vähintää 20 $, tämä teksti viesteistä poistuu.

Sarjalist.js-komponenttiin (ja SarjalistUser.js-komponenttiin) on upotettu kolme muuta komponenttia: Resetstatus.js (Nollaa kuittaukset -nappi), Changestatus.js (Kuittaa-nappi) sekä Logout.js (Kirjaudu ulos -nappi) (kuva 31). Resetstatus.js (Nollaa kuittaukset -nappi) vastaa sarjojen statuksen nollaamisesta. Järjestelmä on ajateltu niin, että se on päällä jatkuvasti. Jos järjestelmä on päällä kello 23:59:50, kaikien kuluneen päivän sarjojen status nollataan, jotta seuraavana päivänä järjestelmä lataa työpöydälle puhtaat sarjat ilman kuittauksia. Voi kuitenkin käydä niin, että järjestelmä ei ole enää päällä sinä aikana, jos esimerkiksi kuitettavia sarjoja ei silloin enää ole, eikä kukaan ole päivystämässä yrityksen toimistolla. Siinä tapauksessa tarvitaan nappi, jota painamalla voidaan päivän alussa puhdistaa ladatut sarjat, jos järjestelmä ei voinut tehdä sitä automaattisesti päivän päätteeksi. Napin avulla asetetaan siis kaikkien kuluvan päivän sarjojen status *epätodeksi* (false).

Changestatus.js-komponentti vastaa Kuittaa-napista, jonka avulla voidaan kuitata sarja toimistolta käsin. Tämä tulee tarpeeseen, jos kuljettajalla ilmenee ongelmia ennen lähtöä, ja hän joutuu soittamaan toimistolle. Asian selvittelyn jälkeen voidaan muuttaa sarjan status vihreäksi kyseisen napin avulla. Toisin sanojen, ajosarjan status muutetaan tietokannassa statukseksi *tosi* (true). Samasta napista statuksen voi muuttaa takaisin alkuperäiseen tilaan (false), jos kuittaus tapahtui esimerkiksi vahingossa.

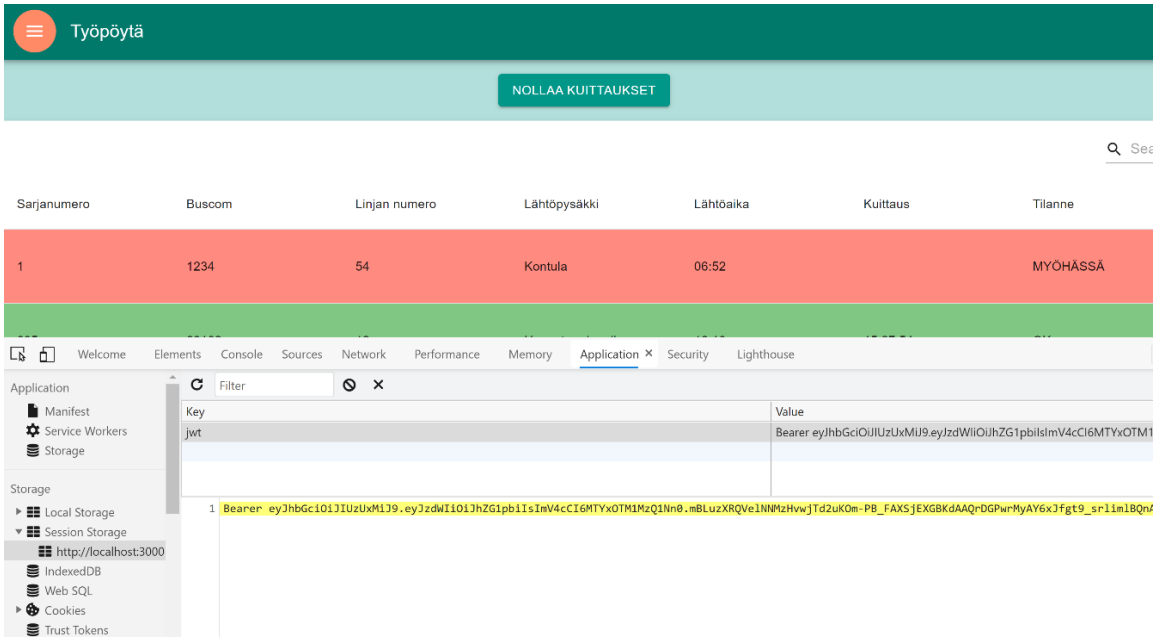
Logout.js (Kirjaudu ulos -nappi) on vastuussa uloskirjautumisesta. Nappia painamalla näkymä siirtyy Login-sivulle (kuva 34).



Kuva 34. Login-sivu

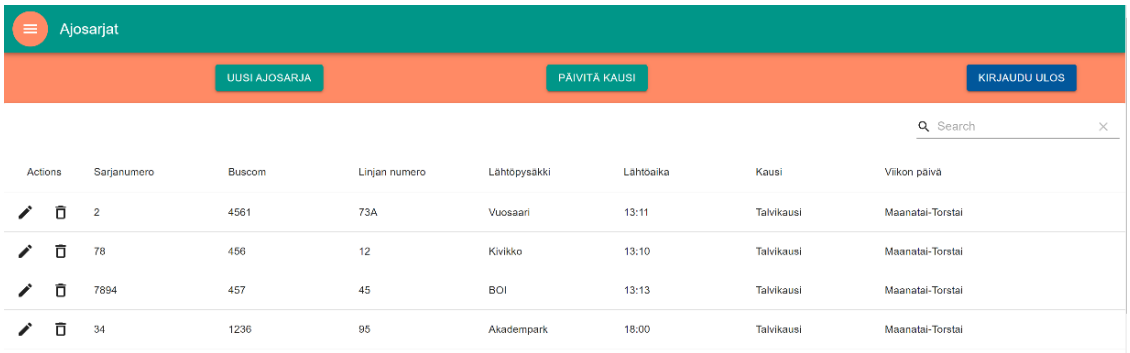
5.2 Login.js- ja Changesarjas.js-komponentti

Login-sivu on Login.js-komponentin tuotos. Kyseinen komponentti lähettää syötetyt käyttäjätiedot palvelinpuolelle. Jos palvelin autentikoi käyttäjän, se muodostaa JWT-verkkotunnuksen sekä palauttaa sen Login.js-komponentille. Login.js-komponentti vuorollaan vastaanottaa vastauksen ja asettaa saadun JWT-verkkotunnuksen selaimen Session Storage -tallennuspaikkaan, josta se haetaan ja asetetaan pyynnön otsikkoon joka kerta ennen tietojen lähettämistä palvelinpuolelle (kuva 35).



Kuva 35. JWT-verkkotunnus selaimen Session Storage -tallennuspaikassa

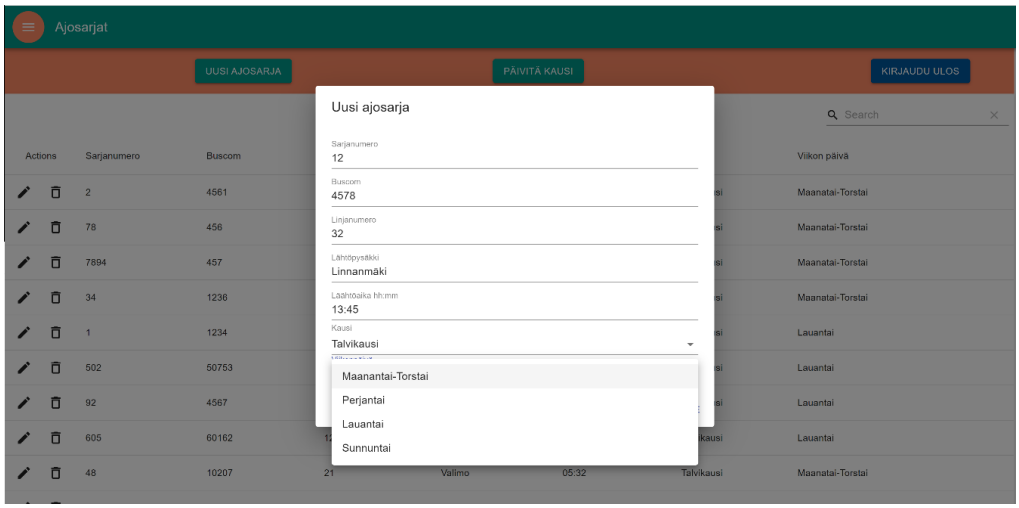
Login.js-komponentti myös hakee käyttäjän roolin palvelimen tietokannasta ja sen perusteella näyttää sisältöä käyttäjälle kirjautumisen jälkeen. Tavallinen käyttäjä (User) voi nähdä vain työpöydän. Admin-roolin käyttäjä voi nähdä työpöydän lisäksi myös Ajosarjat-osion, jossa hän voi lisätä, muokata ja poistaa sarjoja sekä muokata kausia (kuva 36).



Kuva 36. Changesarjas.js-komponentti (Ajosarjat-osio)

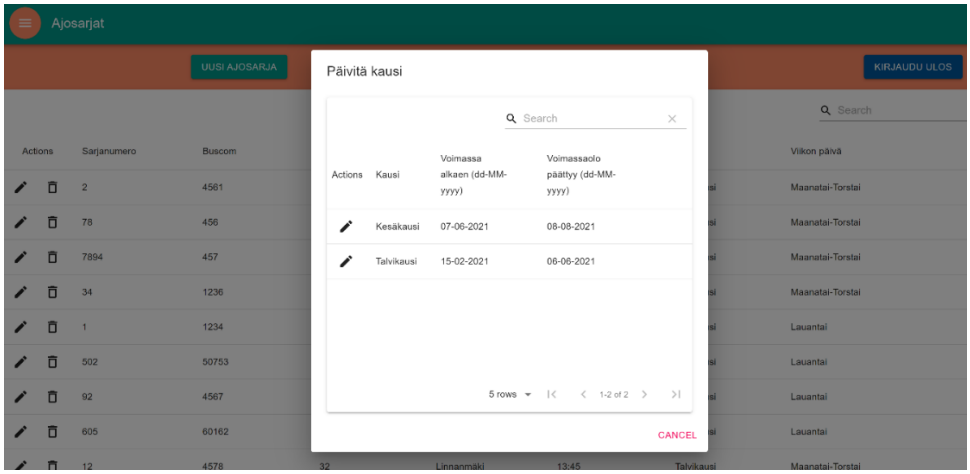
Ajosarjat-osiosta on vastuussa Changesarjas.js-komponentti, johon on upotettu kolme muuta komponenttia: Addsarja.js (Uusi ajosarja -nappi), Addseason.js (Päivitä kausi -nappi) sekä Logout.js (Kirjaudu ulos -nappi) (kuva 36). Changesarjas.js-komponentti lataa kaikki sarjat tietokannasta käyttäjän näkyville, vie tietokantaan sarjoihin tehtyjä muutoksia sekä poistaa tietokannasta sarjoja käyttäjän toimesta.

Uusi sarja -napin avulla voidaan lisätä uusi sarja tietokantaan (kuva 37). Sarjojen muokkaus ja poisto tapahtuu jokaisen sarjan rivillä olevista kuvakkeista (kuva 36).



Kuva 37. Uuden sarjan lisääminen

Päivitä kausi -painikkeesta pääsee muokkaamaan kausia (kuva 38). Ne ovat olennainen osa järjestelmää. Niiden perusteella tietokannasta ladataan sarjoja työpöydälle sekä tarkistetaan kumpaa sarjaa kuljettaja on kuittaamassa, sillä tietokannassa on sarjoja samalla sarjanumerolla sekä kesä- että talvikaudella.



Kuva 38. Kausien muokkaus

5.3 User- ja Admin-roolin näkymät

Sisäänkirjautumisen jälkeen User- ja Admin-roolin näkymät eroavat toisistaan vain Menu-valikon verran. Admin-roolin käyttäjä voi sen nähdä, kun taas User-roolin käyttäjä ei (kuvat 39, 40).

Kuva 39. Admin-roolin käyttäjän näkymä (Sarjalist.js-komponentti)

Kuva 40. User-roolin käyttäjän näkymä (SarjalistUser.js-komponentti)

Kyseinen ominaisuus on toteutettu Login.js-komponentissa if-ehtolauseen avulla (kuva 41). Jos tietokannasta haettu käyttäjän rooli on *admin*, järjestelmä näyttää Sarjalist.js-komponentin (kuva 39). Jos taas käyttäjän rooli on *user*, järjestelmä lataa käyttäjälle näytettäväksi komponentin SarjalistUser.js, jossa ei ole Menu-valikkoa (kuva 40).

```
render() {

    if (this.state.isAuthenticated === true && this.state.role === 'admin') {
      return (
      <div>
        <div style={{float: 'right', marginRight:'50px'}}>
        <Logout logout={this.logout} />
    </div>
          <BrowserRouter >
     <div>
            <Switch>
                <Route exact path="/" component={Sarjalist} />
                <Route path="/changesarjas" component={Changesarjas} />

            </Switch>
      </div>
        </BrowserRouter>
          </div>)
    } else if (this.state.isAuthenticated === true && this.state.role === 'user') {
      return (
        <div>
        <div style={{float: 'right', marginRight:'50px'}}>
        <Logout logout={this.logout} />
      </div>
      <SarjalistUser />

            </div>)
```

Kuva 41. Login.js:n komponenttien suodatus käyttäjän roolin perusteella

Admin-roolin käyttäjä pääsee Menu-valikossa siirtymään Muokkaa sarjoja -napista Ajosarjat-osioon tai Työpöytä-napista Työpöydälle (kuva 42).

Kuva 42. Menu-valikon sisältö

Käyttäjä pääsee kirjautumaan ulos jokaiselta sivulta. Tämä ominaisuus on toteutettu käyttäjämukavuutta ajatellen, ettei Admin-roolin käyttäjän tarvitse tehdä ylimääräisiä siirtoja sivulta toiselle kirjautuakseen ulos. Kyseinen ominaisuus on toteutettu Login.js-komponentissa, joka näyttää Logout.js-komponentin muiden pääkomponenttien kanssa renderöinnin käyttäjäroolisuodattamisen jälkeen (kuva 41).

6 Pohdinta

Tässä luvussa tarkastellaan kehittämisprosessin päätteeksi syntynyttä järjestelmää ja sen jatkokehitystä sekä arvioidaan omaa oppimista tämän projektin aikana.

6.1 Tulokset

Tämän työn tuloksena syntyi web-pohjainen kuittausjärjestelmä, joka vastaa sille asetettuja käyttövaatimuksia. Tavoitteena oli luoda järjestelmä, joka ottaa ajosarjojen kuittauksia vastaan tekstiviestien avulla, merkitsee ajosarjat kuitatuiksi sekä lähettää kuljettajille ilmoitukset onnistuneesta tai epäonnistuneesta kuittauksesta. Järjestelmän piti myös tarjota reaaliaikainen kuittausten seuranta, mahdollisuus ajosarjojen muokkaamiseen, lisäämiseen ja poistamiseen ja kausien muokkaukseen sekä käyttäjäroolit.

Tekstiviestien vastaanotto ja käsittely toimii järjestelmässä toivotulla tavalla. Järjestelmän palvelinpuoli ottaa Twilion viestintäpalvelulta tekstiviestejä vastaan, käsittelee niitä, tarkistaa kuluvan kauden, löytää sen perusteella oikeat ajosarjat ja muuttaa niiden statuksen *todeksi* (true) eli kuittaa niitä, minkä jälkeen kuljettaja saa ilmoituksen onnistuneesta kuittauksesta. Kuljettaja saa ilmoituksen myös epäonnistuneesta kuittauksesta, jos järjestelmässä tapahtuu virhe tai sarjaa ei löydy.

Palvelinpuoli tarjoaa metodeja ajosarjojen muokkaamiseen, lisäämiseen ja poistamiseen sekä kausien muokkaukseen. Lisäksi palvelinpuolen metodit mahdollistavat kuittausten nollaamisen ja yksittäisten ajosarjojen kuittaamisen. Näiden metodien avulla selainpuolella pystyttiin toteuttamaan kaikki toivotut toiminnot. Palvelinpuoli tarjoaa myös REST API-rajapinnan järjestelmän palvelin- ja selainpuolen yhdistämiseksi sekä JWT-autentikoinnin turvalliseen tietojen vaihtoon järjestelmän palvelin- ja selainpuolen välillä.

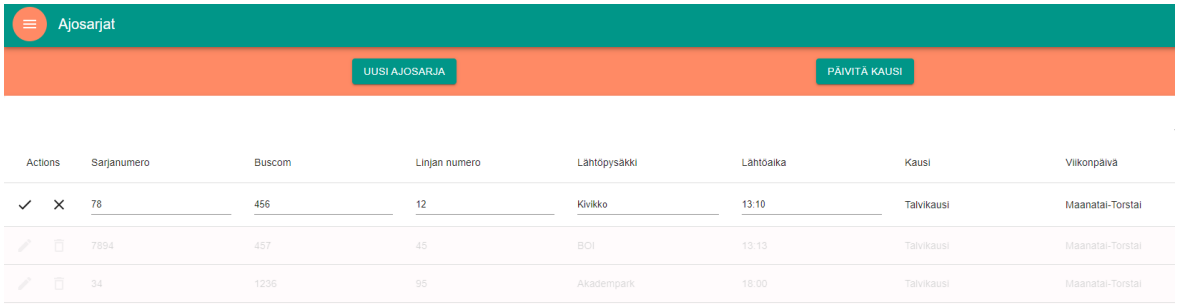
Järjestelmän selainpuoli mahdollistaa ajosarjojen kuittaustilanteen reaaliaikaisen seurannan, ajosarjojen järjestämisen ajan mukaan kuittausten seurannan näkymässä sekä järjestelmän toimintojen käyttöä käyttäjäroolin mukaan. Työssä onnistuttiin toteuttamaan järjestelmään kaikki toivotut toiminnot.

6.2 Jatkokehitys

Järjestelmän palvelin- ja selainpuolen rakenteen ansiosta jatkokehitystä olisi helppo toteuttaa. Järjestelmään voidaan tarvittaessa lisätä uusia ominaisuuksia sekä palvelin- että

selainpuolelle. Esimerkiksi Admin-roolin käyttäjän näkymään olisi hyvä lisätä käyttäjien ja salasanojen hallinta. Myös User-roolin käyttäjän näkymässä salasanan vaihdon mahdollisuus voisi olla hyödyllinen, varsinkin jos User-roolin käyttäjiä olisi paljon. Navigointia sivulta toiselle Admin-roolin käyttäjän näkymässä voitaisiin kehittää sujuvammaksi. Järjestelmän ulkonäköä voitaisiin myös kehittää modernimmaksi sekä saavutettavuutta huomioon ottaen.

Lisäksi järjestelmässä on muutama ratkaisematon ongelma, jotka olisi hyvä ratkaista jatkokehityksessä. Tällä hetkellä ajosarjan muokkauksen voi toteuttaa vain osittain. Ajosarjan jo asetettua kautta tai ajoviikonpäivää on mahdotonta muuttaa (kuva 43). Tämä johtuu siitä, että nämä tiedot tulevat tietokannan eri tauluista. Jotta ajosarjan kyseiset kohdat voitaisiin muuttaa oikein, muokkaustilassa kauden ja viikonpäivän kohdalla pitäisi olla valikot, joissa olisivat kaikki mahdolliset vaihtoehdot. Tätä ei osattu ottaa huomioon järjestelmän suunnitteluvaiheessa ja valittua pöytäkomponenttia (Table) ei voida siihen soveltaa, minkä takia kyseiset kohdat päädyttiin sulkemaan muokkaukselta kokonaan. Tällä ei ole merkitystä järjestelmän pääidean toimivuuden kannalta, mutta käyttömukavuutta ajatellen tämän ominaisuuden olisi kuitenkin hyvä olla, jos järjestelmä otettaisiin yrityksessä käyttöön.



Kuva 43. Ajosarjan muokkaus

Lisäksi ajosarjojen muokkaukseen ja lisäykseen tulisi lisätä validointi. Sitä on jonkin verran palvelinpuolella, mutta järjestelmä ei ilmoita virheellisestä syötöstä käyttäjälle. Kauden validointi on osittain toteutettu selainpuolella kauden muokkauksessa. Järjestelmä ei päivitä kautta, jos se on syötetty väärässä muodossa, mutta se ei myöskään näytä ilmoitusta virheellisestä syötöstä.

Järjestelmän aikalaskentasysteemi vaatii myös laajempaa testausta. Se on toteutettu funktiolla, joka vertaa sarjan lähtöaikaa sekä nykyhetken aikaa ja laskee aikaeron minuuteissa. On huomattu, että joidenkin minuuttien kohdalla järjestelmä saattaa saada

laskennan tulokseksi ”undefined”, mikä voi näkyä järjestelmässä sarjan vääränä värinä minuutin ajan, jos sarjaa ei ole kuitattu.
Lisäksi järjestelmän palvelinpuoli haluttiin toteuttaa englanniksi, jotta sitä voitaisiin niin halutessa soveltaa myös muissa yrityksissä ympäri maailman. Tämä kuitenkin tuotti ylimääräistä päänvaivaa selainpuolta toteuttaessa, sillä kaikki arvot, jotka olivat tietokannassa englanniksi, piti ennen käyttäjälle näyttämistä kääntää suomeksi. On todennäköisesti parempi ratkaisu syöttää tietokantaan tiedot suoraan suomeksi, kuten viikonpäivät ja kaudet, sillä ne on kuitenkin melko helppo vaihtaa, jos jokin muu yritys haluaisi käyttää järjestelmää eri maassa.

6.3 Opinnäytetyöprosessin ja oman oppimisen arviointi

Opinnäytetyön alussa en ollut ollenkaan varma onnistuuko tämä projekti, sillä minulla ei ollut minkälaista käsitystä, miten järjestelmän ominaisuudet voidaan toteuttaa. Epävarmuuteen vaikutti myös se, että työlle oli asetettu melko tiukka aikataulu.

Työ alkoi selvittelyllä, miten tekstiviestien lähetys ja vastaanotto voitaisiin toteuttaa ja minkä palvelun kautta. Suomesta löytyi jonkin verran viestinnän sovellusliittymien tarjoajia, mutta kaikki löytämäni API:t olivat tarkoitettu isojen yritysten käyttöön eivätkä sopineet tämän työn projektiin. Lopulta löysin Twilion, jota pystytään soveltaa myös pienimuotoisiin projekteihin.

Opintojeni kautta olen saanut jonkin verran kokemusta Spring Boot -sovelluskehyksestä, mutta en tiennyt, miten sen yhdistäminen Twilion kanssa tapahtuu. Twilion sivuilta löytyi esimerkkejä Twilion integraatiosta eri ohjelmointikielille, mutta en ollut kuitenkaan varma, miten se voidaan soveltaa järjestelmäni tarpeisiin Spring Bootissa. Kokeilemalla erilaisia tapoja se kuitenkin lopulta onnistui.

Minulla ei myöskään ollut kokemusta kunnon REST API -rajapintojen rakentamisesta, joten esimerkiksi linkkien lisääminen oli tuottanut paljon päänvaivaa. Lisäksi JWT-verkkotunnus oli myös uusi asia, johon en ole ennen tätä työtä törmännyt. Siihen tutustuminen ja sen toteutus veivät aikaa. React.js -sovelluskehykseen olin tutustunut opintojeni alussa enkä sen jälkeen päässyt tekemään sillä muita projekteja, joten sen käyttö ei ollut enää tuoreessa muistissa. Lisäksi suurin osa selainpuolen toteutetuista ominaisuuksista olivat minulle uusia, enkä ollut myöskään varma, onnistuuko niiden toteutus, esimerkiksi ajosarjan värin vaihto statuksesta riippuen tai sivun jatkuva päivittäminen.

Opinnäytetyön prosessin aikana olen oppinut valtavasti asioita. Lisäksi olen kehittänyt loogista ajattelua ja ongelmanratkaisukykyä, sillä järjestelmää toteuttaessa vastaan tuli paljon asioita, joihin ei löytynyt ratkaisua internetistä tai oppikirjoista. Olen hyvin tyytyväinen tämän työn tulokseen. Järjestelmästä tuli juuri sellainen kuin suunnittelin, ja sen toteutus onnistui asetetussa aikataulussa.

Lähteet

Agilie. N.d. Why You Need Twilio to Build a High-Power Communication Platform. Luettavissa: https://agilie.com/en/blog/why-you-need-twilio-to-build-a-high-power-communication-platform. Luettu: 17.5.2021.

Alex, B., Taylor, L., Winch, R., Hillert, G., Grandja, J. & Bryant, J. 2017. Spring Security Reference. Luettavissa: https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle. Luettu: 11.4.2021.

Antonov, A. 2018. Spring Boot 2.0 Cookbook - Second Edition. Packt Publishing. Luettavissa: https://haaga-helia.finna.fi/Record/nelli21.4110000000007635. Luettu: 7.4.2021.

Baeldung, 2021. A Comparison Between Spring and Spring Boot. Luettavissa: https://www.baeldung.com/spring-vs-spring-boot. Luettu: 4.4.2021.

Boduch, A. 2017. React and React Native. Packt Publishing. Luettavissa: https://haaga-helia.finna.fi/Record/nelli21.3710000001092823. Luettu: 8.4.2021.

Chiarelli, A. 2018. Beginning React. Packt Publishing. Luettavissa: https://haaga-helia.finna.fi/Record/nelli21.4100000005599289. Luettu: 8.4.2021.

CodeJava. N.d. What is Eclipse IDE? - for beginner. Luettavissa: https://www.codejava.net/ides/eclipse/what-is-eclipse-ide-for-beginner. Luettu: 20.3.2021.

Griffiths, D. G. 2021. React Cookbook. O'Reilly Media, Inc. Luettavissa: https://haaga-helia.finna.fi/Record/nelli21.4100000011375837. Luettu: 3.5.2021.

Heckler, M. 2021. Spring Boot, Up and Running. Luettavissa: https://learning.oreilly.com/library/view/spring-boot-up/9781492076971/ch01.html. Luettu: 4.4.2021.

Hinkula, J. 2018. Hands-On Full Stack Development with Spring Boot 2.0 and React. Luettavissa: https://haaga-helia.finna.fi/Record/nelli21.4100000004836428. Luettu: 25.3.2021.

JavaTpoint. N.d. Spring Boot Starter Web. Luettavissa: https://www.javatpoint.com/spring-boot-starter-web. Luettu: 7.4.2021.

Json. N.d. Introducing JSON. Luettavissa: https://www.json.org/json-en.html. Luettu: 18.5.2021.

Kamp, P. N.d. SMS API. Luettavissa: https://www.twilio.com/docs/glossary/what-is-sms-api-short-messaging-service. Luettu: 18.5.2021.

Lauret, A. 2019. The Design of Web APIs. Manning Publications. Luettavissa: https://haaga-helia.finna.fi/Record/nelli21.4100000010267657. Luettu: 27.4.2021

Milosevic, D. N.d. REST Security with JWT using Java and Spring Security. Luettavissa: https://www.toptal.com/java/rest-security-with-jwt-spring-security-and-java. Luettu: 11.4.2021.

Oliveira, C. E. d., Antonov, A. & Turnquist, G. L. 2018. Developing Java Applications with Spring and Spring Boot. Packt Publishing. Luettavissa: https://haaga-helia.finna.fi/Record/nelli21.5120000000108133. Luettu: 15.2.2021.

Paraschiv, E. 2020. A Guide to JPA with Spring. Luettavissa: https://www.baeldung.com/the-persistence-layer-with-spring-and-jpa. Luettu: 4.4.2021.

Pirozzi, E., Ferrari L. 2020. Learn PostgreSQL. Luettavissa: https://learning.oreilly.com/library/view/learn-postgresql/9781838985288/eae2bc8b-cbf3-4657-afb9-d348fdfa8460.xhtml. Luettu: 23.4.2021.

PostgreSQL. N.d. About. Luettavissa: https://www.postgresql.org/about. Luettu: 8.4.2021.

Sharma, K. 2021. What is PostgreSQL and why do enterprise developers and start-ups love it? Luettavissa: https://ubuntu.com/blog/what-is-postgresql. Luettu: 17.5.2021.

Simform. 2021. Best Frontend Frameworks of 2021 for Web Development. Luettavissa: https://www.simform.com/best-frontend-frameworks. Luettu: 8.4.2021.

de Sousa Antonio, C. 2015. Pro React. Apress. Luettavissa: https://haaga-helia.finna.fi/Record/nelli21.3710000000541951. Luettu: 8.4.2021.

Spilca, L. 2020. Spring Security in Action. Manning Publications. Luettavissa: https://haaga-helia.finna.fi/Record/nelli21.4100000011523438. Luettu: 28.4.2021.

Spring. 2016. Building a RESTful web-service. Luettavissa: https://spring.io/guides/gs/rest-service. Luettu: 5.4.2021.

Stackshare. N.d. What is Twilio? Luettavissa: https://stackshare.io/twilio. Luettu: 16.2.2021.

Techmagic. 2017. Angular 2 vs React. What to choose in 2017? Luettavissa: https://blog.techmagic.co/angular-2-vs-react-what-to-chose-in-2017. Luettu: 8.4.2021.

Thomas, M. T. 2018. React in Action. Manning Publications. Luettavissa: https://haaga-helia.finna.fi/Record/nelli21.4100000005182746. Luettu: 3.5.2021.

Thompson, V. 2020. What is Twilio and how does it work? An introduction to the leader in customer engagement. Luettavissa: https://www.twilio.com/the-current/what-is-twilio-how-does-it-work. Luettu: 16.2.2021.

Tutorialspoint. N.d. MVC Framework - Introduction. Luettavissa: https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm. Luettu: 2.5.2021.

Twilio. 2018. Video. What is a Twilio? Luettavissa: https://www.yotube.com/watch?v=euJAcVfn0C4. Luettu: 16.2.2021.

Varanasi, B., Belida, S. & Vohra, D. 2015. Spring REST. Apress. Luettavissa: https://haaga-helia.finna.fi/Record/nelli21.3710000000433708. Luettu: 1.5.2021.

Vogella. N.d. Using the Eclipse IDE for Java programming - Tutorial. Luettavissa: https://www.vogella.com/tutorials/Eclipse/article.html. Luettu: 20.3.2021.

Walls, C. 2015. Spring Boot in Action. Manning Publications. Luettavissa: https://haaga-helia.finna.fi/Record/nelli21.3710000000613635. Luettu: 7.4.2021.

Liitteet

Liite 1. Tunnistetut vaatimukset

1. Admin-roolin käyttäjän pitää voida lisätä uusi ajosarja (toiminnallinen, välttämätön)
2. Admin-roolin käyttäjän pitää voida muokata ajosarjaa (toiminnallinen, välttämätön)
3. Admin-roolin käyttäjän pitää voida poistaa ajosarja (toiminnallinen, välttämätön)
4. Admin-roolin käyttäjän pitää voida muokata kausien voimassaoloaikaa (toiminnallinen, välttämätön)
5. Admin-roolin käyttäjän pitää voida nähdä kuluvan päivän sarjojen kuittaustilanne (toiminnallinen, välttämätön)
6. Admin-roolin käyttäjän pitää voida nähdä kaikki järjestelmässä olevat sarjat (toiminnallinen, välttämätön)
7. Admin-roolin käyttäjän pitää voida nollata kuluvan päivän ajosarjojen status (toiminnallinen, tarpeellinen)
8. Admin-roolin käyttäjän pitää voida muuttaa sarjojen statuksen kuitatuksi (toiminnallinen, välttämätön)
9. Admin-roolin käyttäjän pitää voida kirjautua sisään ja ulos (toiminnallinen, välttämätön)
10. User-roolin käyttäjän pitää voida nähdä kuluvan päivän sarjojen kuittaustilanne (toiminnallinen, välttämätön)
11. User-roolin käyttäjän pitää voida nollata kuluvan päivän ajosarjojen status (toiminnallinen, tarpeellinen)
12. User-roolin käyttäjän pitää voida muuttaa sarjojen statuksen kuitatuksi (toiminnallinen, välttämätön)
13. User-roolin käyttäjän pitää voida kirjautua sisään ja ulos (toiminnallinen, välttämätön)
14. Kuljettajan pitää voida kuitata ajosarja (toiminnallinen, välttämätön)
15. Järjestelmässä on oltava roolien hallinta (toiminnallinen, välttämätön)
16. Onnistuneesta kuittauksesta on tultava vahvistus (toiminnallinen, tarpeellinen)
17. Epäonnistuneesta kuittauksesta on tultava ilmoitus (toiminnallinen, tarpeellinen)

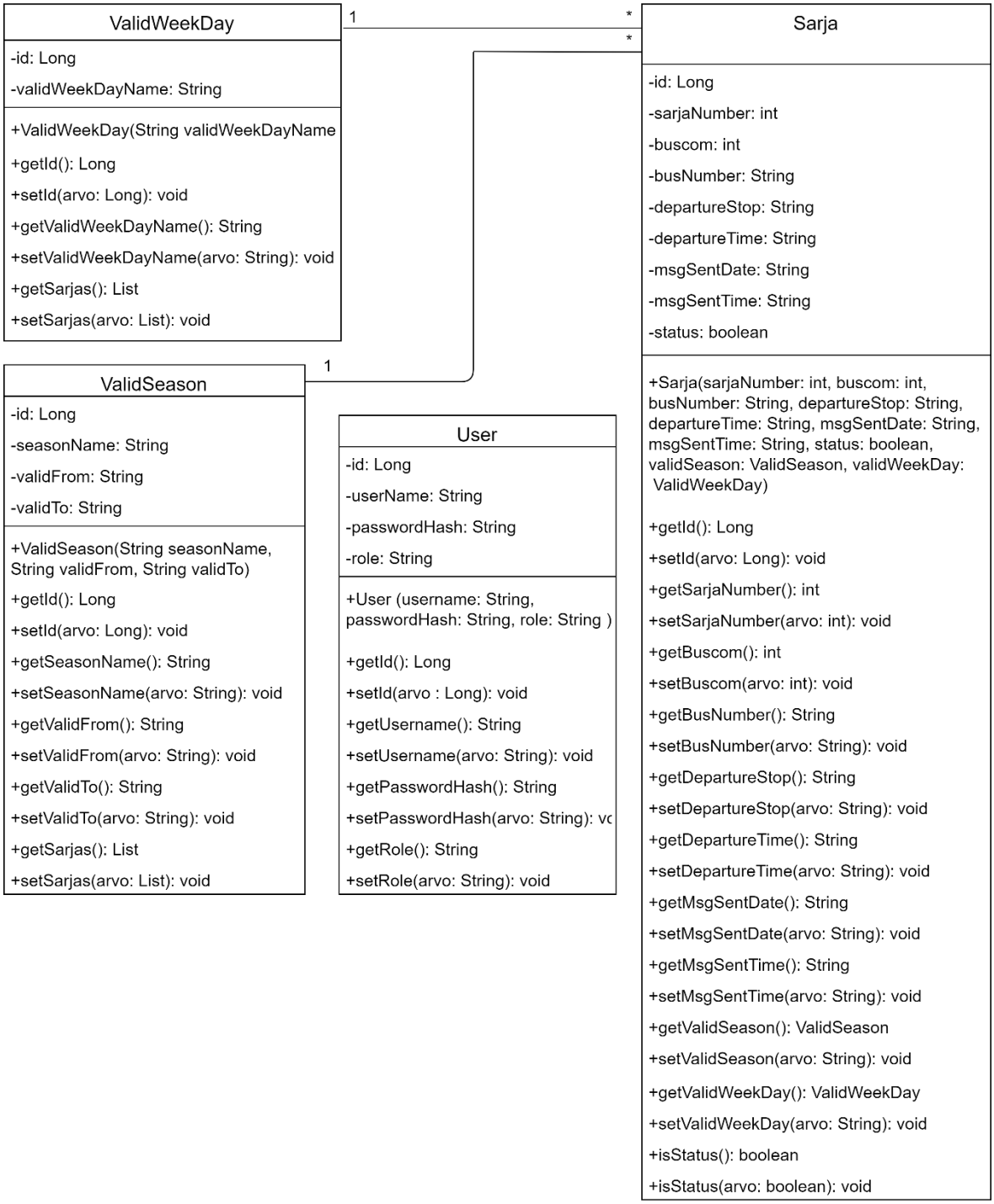
Ei-toiminnalliset vaatimukset

18. Ajosarjojen kuittaustilanteen pitää olla reaaliaikainen (ei-toiminnallinen, välttämätön)
19. Kuljettajan pitää voida kuitata ajosarja nopeasti ja helposti ilman jonotusta (ei-toiminnallinen, välttämätön)
20. Kuljettajan pitää voida kuitata ajosarja mistä tahansa puhelimesta (ei-toiminnallinen, hyödyllinen)

Rajoitukset

21. Kausien nimiä ei voi muokata järjestelmässä (rajoite, välttämätön)

Liite 2. Pääluokkien kaaviot



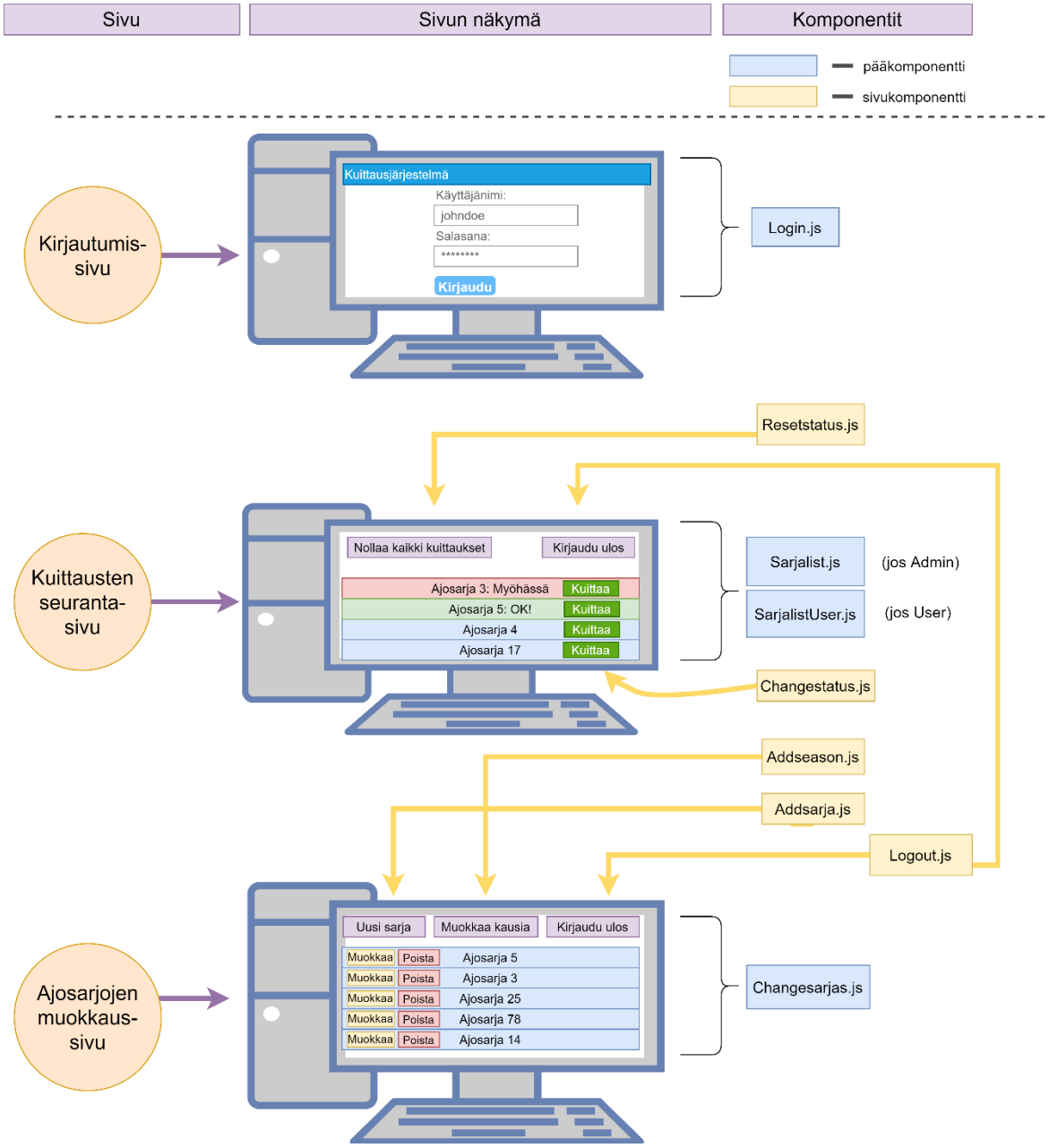
Liite 3. Esimerkki järjestelmän REST API:stä

GET http://localhost:8080/sarjas

Params Authorization Headers (11) Body Pre-request Script Tests Settings

Body Cookies (1) Headers (15) Test Results

Pretty Raw Preview Visualize JSON


```
1  {
2      "_embedded": {
3          "sarjas": [
4              {
5                  "id": 3,
6                  "sarjaNumber": 2,
7                  "buscom": 4561,
8                  "busNumber": "73A",
9                  "departureStop": "Vuosaari",
10                 "departureTime": "13:11",
11                 "msgSentDate": "22-04-2021",
12                 "msgSentTime": "12:41:50",
13                 "status": false,
14                 "validSeason": {
15                     "id": 1,
16                     "seasonName": "winter",
17                     "validFrom": "15-02-2021",
18                     "validTo": "06-06-2021"
19                 },
20                 "validWeekDay": {
21                     "id": 1,
22                     "validWeekDayName": "Monday-Thursday"
23                 },
24                 "_links": {
25                     "self": {
26                         "href": "http://localhost:8080/sarjas/3"
27                     },
28                     "valid_season": {
29                         "href": "http://localhost:8080/seasons/1"
30                     },
31                     "valid_weekday": {
32                         "href": "http://localhost:8080/weekdays/1"
33                     }
34                 }
35             },
36             {
37                 "id": 19,
38                 "sarjaNumber": 78,
39                 "buscom": 456,
40                 "busNumber": "12",
41                 "departureStop": "Kivikko",
```

Liite 4. React-sovelluksen rakenne