Joonatan Peltonen

WEB-RAPORTOINTITYÖKALUN KEHITTÄMINEN

Tekniikka
2022

VAASAN AMMATTIKORKEAKOULU
Tietotekniikka

TIIVISTELMÄ

Tekijä	Joonatan Peltonen
Opinnäytetyön nimi	Web-raportointityökalun kehittäminen
Vuosi	2022
Kieli	suomi
Sivumäärä	56
Ohjaaja	Anna-Kaisa Saari

Opinnäytetyön tarkoituksena oli kehittää verkkopohjainen raportointityökalu Wapice Oy:n asiakkaalle. Työkalun avulla käyttäjä voi hakea dataa kahdesta eri tietokannasta erilaisten suodattimien avulla. Käyttäjä voi luoda, muokata tai poistaa raporttipohjia, joissa voidaan valita haluttu tietokantataulu sekä sen sarakkeet. Työkalu palauttaa taulukkodataa, jota voidaan käsitellä esimerkiksi Excelissä.

Projektin käyttöliittymä toteutettiin Angular-sovelluskehyksellä. Palvelimen kehitykseen käytettiin Javaa sekä Spring Boot-sovelluskehystä. Spring Bootissa hyödynnettiin erilaisia kirjastoja esimerkiksi koodin määrän vähentämiseen ja sovelluksen testaamiseen. Käyttöliittymän ja palvelimen kommunikaatio toteutettiin REST-rajapinnan avulla. Projektissa käytettiin Oraclen tietokantoja sekä hyödynnettiin tietokanta näkymiä, joiden avulla saatiin yksinkertaistettua monimutkaisia SQL-kyselyjä.

Työn tuloksena saatiin toimintavalmis raportointityökalu, jonka avulla käyttäjä voi luoda raporttipohjia, joiden avulla voidaan hakea dataa kahdesta eri tietokannasta. Datan määrää voidaan rajoittaa erilaisten suodattimien avulla. Raportin perusteella löydettyä dataa voidaan tarkastella käyttöliittymässä olevan taulukon avulla tai se voidaan ladata laitteelle CSV-tiedostona.

Avainsanat raportointityökalu, kehitys, Spring Boot, Angular

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Tietotekniikka

ABSTRACT

Author	Joonatan Peltonen
Title	Developing a web-based reporting tool
Year	2022
Language	Finnish
Pages	56
Name of Supervisor	Anna-Kaisa Saari

The purpose of this thesis was to develop a web-based reporting tool for a customer of Wapice Ltd. With the reporting tool the user can search data from two databases using different filters. The user can create, modify or delete report templates where the database table and its columns can be selected. The tool returns table data which can be processed in Excel, for example.

The projects user interface was implemented using Angular framework. Java and Spring Boot framework were used to develop the server side. Spring Boot uses various libraries to reduce the amount of code and test the application, for example. The communication between the user interface and the server was implemented using the REST interface. The project used Oracle databases and database views to simplify complex SQL queries.

The result of the project is a finished reporting tool that allows the user to create report templates that can be used to search data from two different databases. The amount of data can be limited by using various filters. The data found with the report can be previewed in the user interface or downloaded to the device as a CSV file.

Keywords reporting tool, development, Spring Boot, Angular

SISÄLLYS

KUVA- JA TAULUKKOLUETTELO

LYHENNELUETTELO

JS	JavaScript
TS	TypeScript
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
HTTP	Hypertext Transfer Protocol
CLI	Command-line interface
JVM	Java virtual machine
DB	Database
SQL	Structured Query Language
REST	Representational state transfer
API	Application Program Interface
URI	Uniform Resource Identifier
JSON	JavaScript Object Notation
CSV	Comma-separated values
JPA	Java Persistence API
ORM	Object Relational Mapping

1 JOHDANTO

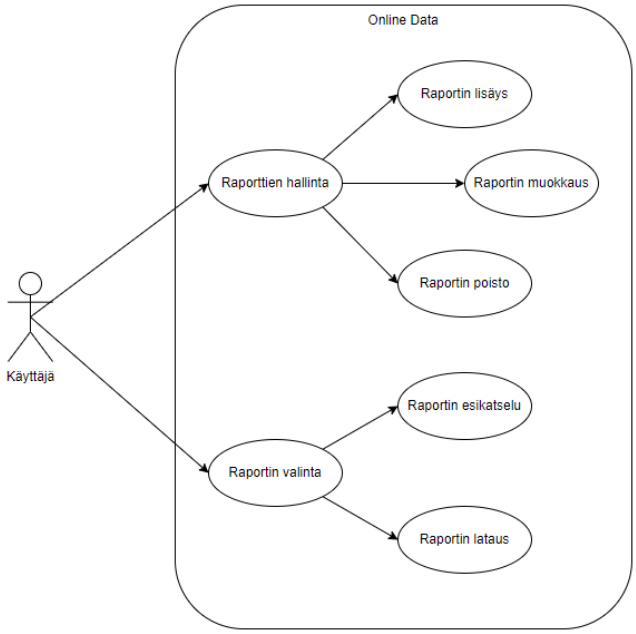
Tässä opinnäytetyössä kehitetään web-pohjainen raportointityökalu Wapice Oy:n asiakkaalle. Asiakas haluaa web-sovelluksen, joka yhdistää kahden eri sovelluksen tietokannat yhteen paikkaan datan raportoinnin helpottamiseksi. Tietokannan data halutaan ladata sovelluksesta CSV-tiedostona. Muita tiedostomuotoja lisätään mahdollisesti tulevaisuudessa.

Hyödyntämällä erilaisia sovelluskehyksiä (engl. application framework) saadaan tehokkaasti kehitettyä tuotantotason sovelluksia. Useiden sovelluskehyksien avulla voidaan luoda laajoja ja skaalautuvia sovelluksia, joita on myös helppo laajentaa tarvittaessa.

Opinnäytetyössä tutustutaan teknologioihin kuten Angular, Java, Spring Boot, Oracle tietokannat ja Docker. Projektin arkkitehtuuriosiossa perehdytään esimerkiksi REST-rajapintaan sekä tietokantarakenteeseen. Projektin toteutusvaiheessa keskitytään pääosin Angular- ja Spring Boot-sovelluskehysten toimintoihin sekä toteutuksiin. Myös käyttöliittymän rakenne käydään läpi toteutusvaiheessa.

2 PROJEKTIN TARKOITUS JA TAVOITTEET

Online Data-projektin tarkoitus on kehittää web-pohjainen raportointityökalu. Sovellus vaatii sisäänkirjautumisen ennen sen käyttämistä. Käyttäjillä on erilaisia oikeuksia, joiden avulla voidaan rajoittaa mille sivuille käyttäjä pääsee tai mitä toimintoja käyttäjä voi tehdä. Käyttäjäoikeuksia ja -ryhmiä hallitsee asiakkaan erillinen sovellus. Työkalu rakentuu kahdesta eri päänäkymästä. Ensimmäinen näkymä on raportin luontia ja muokkaamista varten. Toisen näkymän tarkoitus on valita raporttipohja ja hakea dataa sen perusteella, jonka jälkeen data voidaan ladata paikallisesti omalle laitteelle CSV-tiedostona. Sovelluksen perustoimintoja on esitetty kuvassa 1.



Kuva 1. Kaavio sovelluksen toiminnoista

Projektin tavoitteina on saada tehokas ja helppokäyttöinen raportointityökalu, jonka avulla käyttäjä pystyy hakemaan dataa kahdesta eri tietokannasta. Raportointityökalulla voidaan tarkasti määritellä mistä tietokannan tauluista dataa haetaan sekä mitä kenttiä raporttiin sisällytetään. Dataa voidaan suodattaa erilaisten suodatustoimintojen kuten listojen tai raja-arvojen avulla. Esimerkiksi numeeri-

sille kentille voidaan asettaa raja-arvoja, joiden avulla dataa voidaan rajoittaa. Raportointityökalu sisältää huomattavasti enemmän valittavia kenttiä ja suodatusvaihtoehtoja kuin tietokantojen omat sovellukset. Nämä sovellukset pystyvät vain hakemaan dataa omista tietokannoista, mutta raportointityökalulla molemmat tietokannat tuodaan yhteen paikkaan, joten tietoa voidaan hakea molemmista.

3 KÄYTETYT TEKNOLOGIAT

Tässä luvussa käydään läpi sovelluksen kehittämisessä käytettyjä teknologioita, jotka luovat sovelluksen arkkitehtuurin. Käyttöliittymän kehittämiseen käytetään Angular-sovelluskehystä, joka käyttää TypeScript-, HTML- ja CSS-ohjelmointikieliä. Palvelimen (engl. server) kehittämiseen käytetään Javaa. Tietokantoina käytetään Oraclen relaatiotietokantoja. Docker ympäristöä käytetään sovelluksen rakentamiseen sekä ajamiseen esimerkiksi tuotantoympäristössä.

3.1 Angular

Angular on Googlen kehittämä komponenttipohjainen sovelluskehys. Angularin avulla voidaan rakentaa web-pohjaisia sovelluksia. Angularin mukana tulee paljon erilaisia työkaluja sovelluksen kehittämiseen, testaukseen ja päivittämiseen. Angular projektit voivat skaalautua yhden kehittäjän projektista yritystason sovelluksiin.[1]

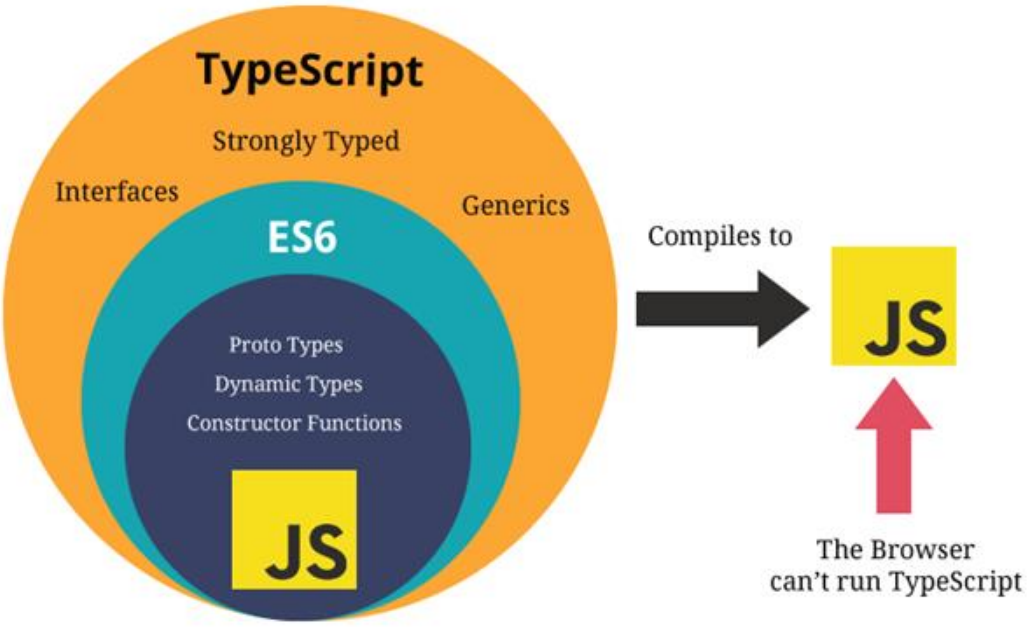
3.1.1 Yleisesti

Alkuperäinen Angularin versio, joka tunnetaan nykyään nimellä AngularJS, julkaistiin vuonna 2010. AngularJS oli alun perin myös Googlen kehittämä JavaScript-pohjainen sovelluskehys, jonka tarkoituksena oli helpottaa dynaamisten verkkosivujen luontia. Myöhemmin Google päätti tehdä seuraavan Angular version kokonaan uudelleen, koska muut kilpailevat JavaScript-sovelluskehykset menivät AngularJS:n ohi.[2]

[1] Angular. What is Angular?
[2] Altexsoft. 2020. The Good and the Bad of Angular Development.

3.1.2 TypeScript

Vuonna 2016 Google julkaisi Angular 2 version.[3] Tämä uusi versio oli uudelleen kirjoitettu käyttäen TypeScript ohjelmointikieltä JavaScriptin sijasta. TypeScript on tyypitetty versio JavaScriptistä, jonka tarkoituksena on parantaa koodin laatua sekä löytää mahdollisia virheitä. Kuvassa 2 on havainnollistava kaavio TypeScriptin ja JavaScriptin eroista.



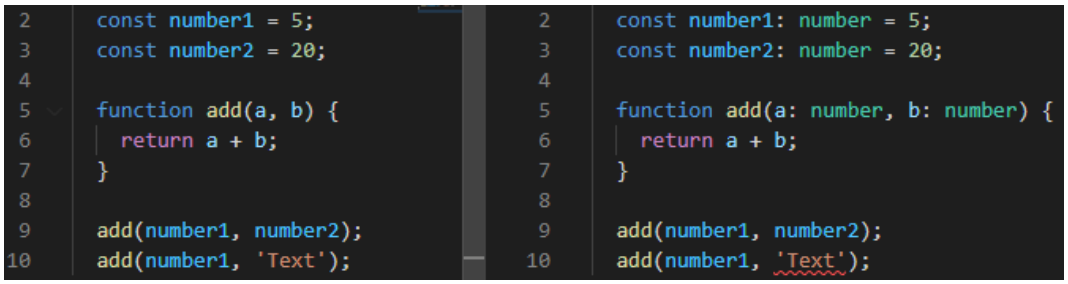
Kuva 2. TypeScriptin ja JavaScription erot.[4]

Kuvassa 3 on sama koodi kirjoitettu JavaScriptillä sekä TypeScriptillä. Koodi näyttää melkein samalta, mutta TypeScript-versiossa muuttujiin sekä funktion para-

[3] Altexsoft. 2020. The Good and the Bad of Angular Development.
[4] Vishnupriya. 2018. What is Typescript?

metreihin on lisätty tyypit. Kuvassa rivillä 10 parametri 'Text' on alleviivattu punaisella, koska Typescript huomaa virheen, kun tekstiä yritetään syöttää funktioon, joka hyväksyy vain numeroja.

TypeScript on hyödyllinen ohjelmointikieli sekä työkalu ohjelmoijalle, koska se voi estää mahdollisten virheiden pääsemisen ohjelmakoodiin sekä parantaa koodin luettavuutta. Esimerkiksi selaimet eivät ymmärrä TypeScript koodia suoraan, vaan se pitää kääntää takaisin JavaScript koodiksi ennen kuin sitä voi ajaa.[5]

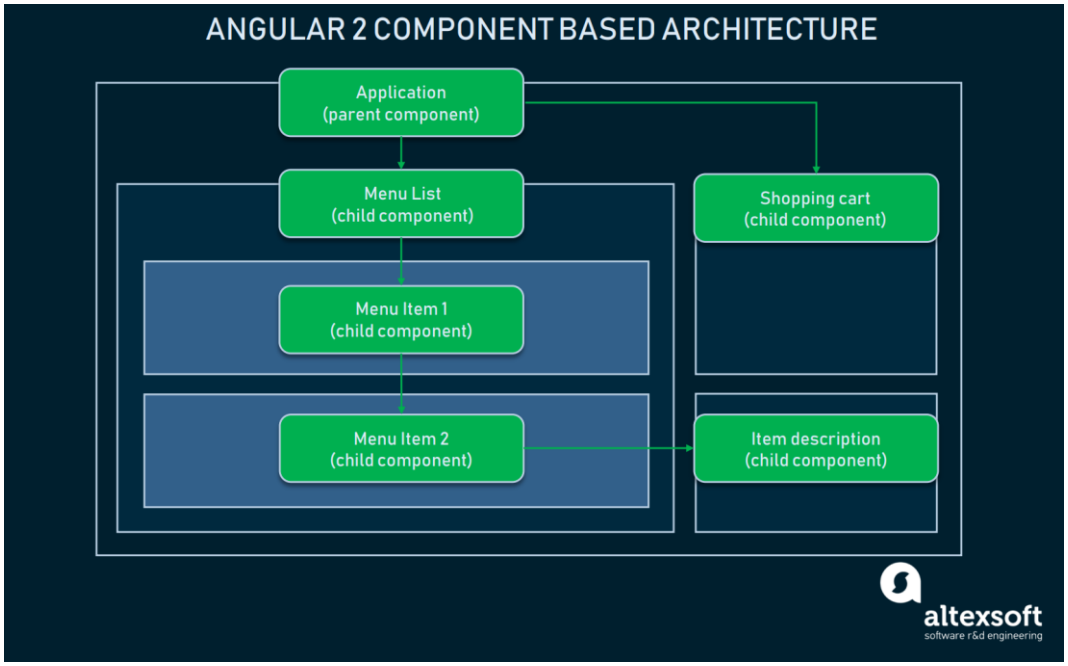


Kuva 3. JavaScript koodi oikealla ja TypeScript koodi vasemmalla.

3.1.3 Komponentit

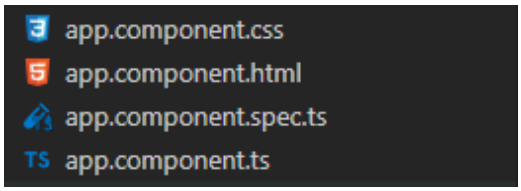
Angular 2 versiossa myös arkkitehtuuri muuttui paljon. Angular siirtyi komponenttipohjaiseksi, mikä tarkoittaa, että käyttöliittymä rakentuu useista eri komponenteista. Esimerkiksi yksi sivu voi olla yksi komponentti tai se voidaan jakaa useampaan pienempään komponenttiin. Yksittäistä komponenttia voidaan käyttää uudestaan eri sivulla tai näkymällä. Komponenttipohjaisuus lisää koodin uudelleenkäyttämistä sekä vähentää duplikaattikoodin määrää. Esimerkiksi kuvassa 4 on jaettu sivu useaan komponenttiin, jossa yksi laatikko kuvaa yhtä komponenttia. Yhden komponentin sisällä olevia komponentteja kutsutaan lapsikomponenteiksi.[6]

[5] Altexsoft. 2020. The Good and the Bad of Angular Development.
[6] Altexsoft. 2020. The Good and the Bad of Angular Development.



Kuva 4. Komponenttien hierarkiarakenne.[7]

Angularissa yksi komponentti rakentuu yleensä neljästä eri tiedostosta; Typescript-, HTML-, CSS- sekä specs.ts-tiedostoista. TypeScript-tiedosto sisältää komponentin logiikan. HTML-tiedosto, jossa on määritelty komponentin rakenne html-kielellä. CSS-tiedosto, jossa on komponenttiin liittyvät tyylit. Komponentilla voi olla myös testausta varten oleva spec.ts-tiedosto, jossa testataan komponentin toimivuutta. Kuvassa 5 on esitetty app-komponentin kaikki tiedostot.



Kuva 5. App-komponentin kaikki tiedostot.

[7] Altexsoft. 2020. The Good and the Bad of Angular Development.

Kuvassa 6 on esitetty app-komponentti, jolla on komponentin määrittelyyn tarvittava @Component-annotaatio. Annotaatiossa on selector-kenttä, jolla määritetään komponentin HTML-tagin nimi. TemplateUrl-kentässä määritetään komponentin HTML-tiedosto ja styleUrls-kentässä määritetään komponentin tyylejä sisältävän tiedoston. HTML-koodille ja CSS-tyyleille ei ole pakko luoda erillisiä tiedostoja. Tällöin HTML-koodi määritellään template-kentässä ja tyylit styles-kentässä. Tämä on esitetty kuvassa 7.[8]

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  title = 'app';

  constructor() {}
}
```

Kuva 6. App-komponentin TypeScript-tiedosto.

```
@Component({
  selector: 'app-component-overview',
  template: '<h1>Hello World!</h1>',
  styles: ['h1 { font-weight: normal; }']
})
```

Kuva 7. HTML-koodin ja tyylien määrittely komponentin sisällä.[9]

[8] Angular. Angular Components Overview.
[9] Angular. Angular Components Overview.

3.1.4 Palvelut

Angularissa palvelu (engl. service) on luokka, joka sisältää TypeScript logiikkaa, jota halutaan uudelleen käyttää useassa eri komponentissa. Palvelua voidaan käyttää esimerkiksi HTTP-kutsujen tekemiseen tai sinne voidaan varastoida muuttujien arvoja, joita voidaan lukea tai muokata eri komponenteissa. Palvelu määritellään @Injectable-annotaatiolla kuten kuvassa 8 on esitetty. Palvelulla on log()-funktio, joka ottaa parametrina viestin, joka tallennetaan logs-muuttujaan sekä viesti näytetään selaimen konsolissa. Palvelu voidaan injektoida komponenttiin kuvan 9 mukaisesti. Injektoinnin jälkeen komponentti voi käyttää palvelun toimintoja.[10]

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class Logger {
  logs: string[] = [];

  log(message: string) {
    this.logs.push(message);
    console.log(message);
  }
}
```

Kuva 8. Esimerkki Logger palvelusta.[11]

```
constructor(private logger: Logger) {  }
```

Kuva 9. Palvelun injektointi konstruktorilla.[12]

[10] Angular. Intro to services and DI.
[11] Angular. Dependency injection.
[12] Angular. Dependency injection.

3.1.5 Moduulit

Angular-sovellus voidaan jakaa useampiin moduuleihin. Esimerkiksi yksi näkymä sekä siihen liittyvät komponentit että palvelut voidaan pakata yhteen moduuliin. Sovelluksella pitää olla vähintään yksi juurimoduuli, joka on yleensä valmiiksi luotu AppModule. Moduuli luodaan @NgModule-annotaatiolla kuten kuvassa 10 on esitetty.[13]

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Kuva 10. Esimerkki AppModule-moduulista.[14]

3.1.6 Direktiivit

Direktiivien (engl. Directive) avulla voidaan muokata HTML-elementtien toimintaa. Esimerkiksi ngClass-direktiivillä voidaan asettaa HTML-elementille jokin CSS-luokka, jos tietty ehto toteutuu. Kuvassa 11 lisätään special-tyyliluokka div-elementille, jos isSpecial-muuttuja palauttaa true-arvon. *ngIf-direktiivin avulla voidaan näyttää tietty HTML-elementti vain silloin, jos sille annettu ehto toteutuu. Kuvassa 12 näytetään div-elementti, jos currentCustomer-muuttuja on määritelty.

[13] Angular. Intro to modules.
[14] Angular. Intro to modules.

Jos halutaan näyttää lista elementtejä, se voidaan tehdä helposti käyttäen *ngFor-direktiiviä. Kuvassa 13 luodaan items-listan jokaiselle esineellä oma div-elementti, jossa näytetään esineen nimi.[15]

```
<div [ngClass]="isSpecial ? 'special' : ''">This div is special</div>
```

Kuva 11. NgClass-direktiivin käyttö.[16]

```
<div *ngIf="currentCustomer">Hello, {{currentCustomer.name}}</div>
```

Kuva 12. NgIf-direktiivin käyttö.[17]

```
<div *ngFor="let item of items">{{item.name}}</div>
```

Kuva 13. NgFor-direktiivin käyttö.[18]

3.1.7 Datan sidonta

Datan sidonta (engl. data binding) TypeScript- ja HTML-tiedoston välillä on tärkeä osa Angularin toimintaa. Erilaisten muuttujien arvot halutaan usein näyttää selaimessa käyttäjälle ja käyttäjä voi vaikuttaa näiden muuttujien arvoihin esimerkiksi tekstikenttien ja nappien avulla. Angularissa datan sidonta voi olla yksi- tai kaksisuuntaista.[19]

Esimerkiksi kuvassa 12 viitataan kaksoisaaltosulkujen sisällä currentCustomer-objektin name-kenttään, joka on määritetty komponentin TypeScript-tiedostossa. Jos arvoa muutetaan TypeScript-tiedostossa, silloin HTML-elementti päivitetään,

[15] Angular. Built-in directives.
[16] Angular. Built-in directives.
[17] Angular. Built-in directives.
[18] Angular. Built-in directives.
[19] Garg, B. 2019. Angular Architecture Overview.

jotta muuttunut arvo tulee selaimeen näkyville. Tämä on esimerkki yksisuuntaisesta sidoksesta (engl. one-way binding).

Kaksisuuntainen sidos (engl. two-way binding) voidaan luoda [(ngModel)] -direktiivillä. Tämä direktiivi voidaan kytkeä esimerkiksi tekstikenttään, jonka avulla käyttäjä voi muuttaa tekstikentän arvoa ja muutos näkyy myös ohjelmakoodin puolella. Jos arvoa muutetaan ohjelmakoodin puolella, muutos tulee myös selaimelle näkyviin. Kuvassa 14 luodaan kaksisuuntainen sidos currentItem-objektin name-kenttään.[20]

```
<input [(ngModel)]="currentItem.name" id="example-ngModel">
```

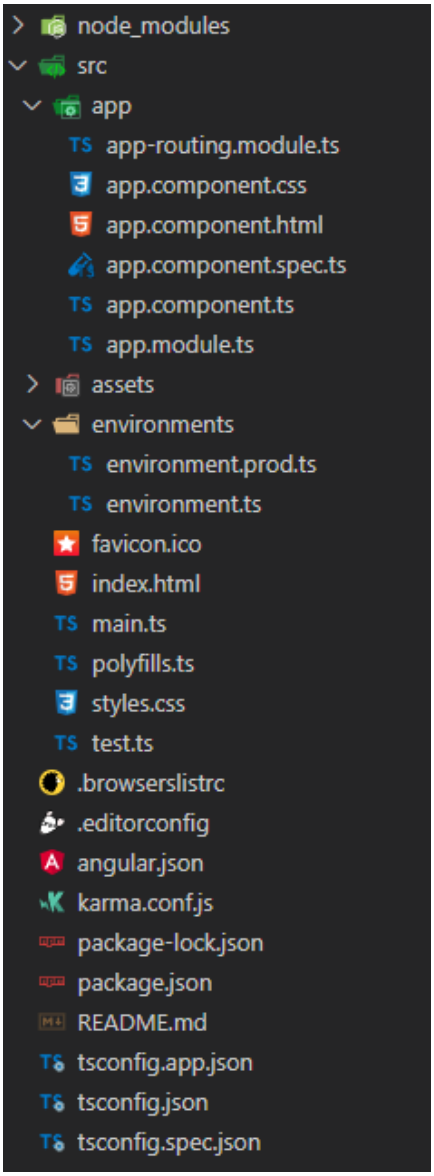
Kuva 14. NgModel-direktiivin käyttö.[21]

3.1.8 Angular CLI

Angular CLI on komentokehotetyökalu, jonka avulla voidaan luoda ja kehittää Angular-sovelluksia. Ennen Angular CLI asentamista Node.js tulee olla asennettuna koneelle. Komentokehote voidaan asentaa komennolla *npm install -g @angular/cli*. Komennolla *ng version* nähdään asennetun työkalun versio ja komento *ng help* listaa kaikki käytettävissä olevat komennot.[22]

Uusi Angular projekti voidaan luoda komennolla *ng new <project>,* jossa <project> on halutun projektin nimi. Tämä komento luo kuvan 15 mukaisen pohjan projektille. Sovellus voidaan käynnistää komennolla *ng serve*, joka rakentaa ja käynnistää sovelluksen paikallisesti. Käynnistyksen jälkeen selaimella voidaan siirtyä osoitteeseen http://localhost:4200/, jossa sovellus on käynnissä.

[20] Angular. Built-in directives.
[21] Angular. Built-in directives.
[22] Angular. CLI Overview and Command Reference.



Kuva 15. Angular projektin generoitu pohja.

Angular CLI:n avulla voidaan myös generoida esimerkiksi moduuleja, komponentteja tai palveluja. Komennolla *ng generate component home*, CLI generoi home komponentin kuvan 16 mukaisesti. Komento lisää myös luodun komponentin app-moduuliin automaattisesti.[23]

[23] Angular. Ng generate.

```
                                        ng generate component home
CREATE src/app/home/home.component.html (19 bytes)
CREATE src/app/home/home.component.spec.ts (612 bytes)
CREATE src/app/home/home.component.ts (267 bytes)
CREATE src/app/home/home.component.css (0 bytes)
UPDATE src/app/app.module.ts (467 bytes)
```

Kuva 16. Komponentin generointi Angular CLI:llä.

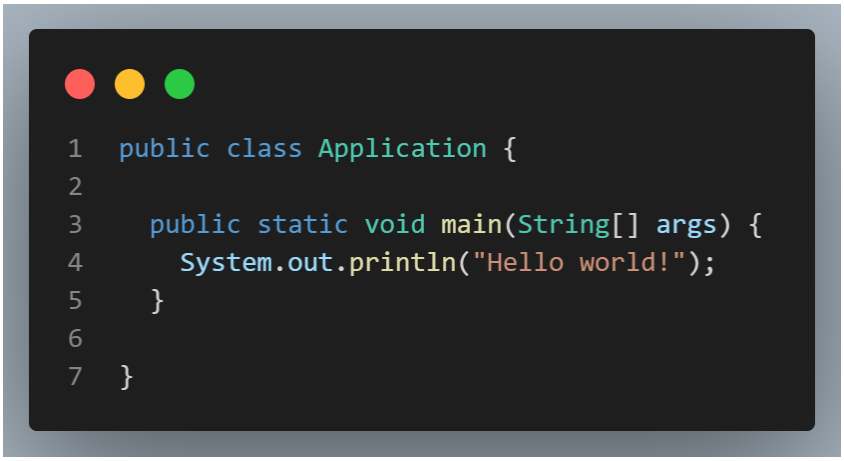
3.2 Java

Java on Sun Microsystems:n kehittämä oliopohjainen ohjelmointikieli, joka julkaistiin vuonna 1995 ja on yhä yksi suosituimmista ohjelmointikielistä. Java on alustariippumaton, mikä tarkoittaa, että sitä voidaan ajaa usealla eri järjestelmällä, esimerkiksi Windowsilla, Linuxilla tai Macilla. Javalla voidaan luoda esimerkiksi verkkosovelluksia, mobiilisovelluksia tai pelejä.[24]

3.2.1 Yleisesti

Java on luokkapohjainen sekä olio-painotteinen ohjelmointikieli, jonka syntaksi on yksinkertainen ja muistuttaa paljon toisia ohjelmointikieliä kuten C ja C++. Ohjelmakoodi käännetään tavukoodiksi, joka ajetaan Java virtuaalikoneessa (JVM). Tämän etuna on se, että koodi voidaan kirjoittaa kerran ja se voidaan sitten ajaa millä tahansa Javaa tukevalla laitteella. Java virtuaalikoneella on myös oma ”roskien kerääjä” (engl. garbage collector), jonka tarkoituksena on hallita muistia poistamalla käyttämättömiä olioita. Kuvassa 17 on esimerkki yksinkertaisesta Java-koodista, jossa tulostetaan lause ”Hello world!”. Javassa main-metodi on sovelluksen aloituspiste.[25]

[24] w3schools. Java Introduction.
[25] GeeksforGeeks. Introduction to Java.



```
1  public class Application {
2
3    public static void main(String[] args) {
4      System.out.println("Hello world!");
5    }
6
7  }
```

Kuva 17. Esimerkki Java-ohjelmasta.

3.2.2 Spring Boot

Spring Boot on Java-pohjainen sovelluskehys, jonka avulla voidaan luoda helposti tuotantovalmiita sovelluksia. Sovelluskehyksellä voidaan luoda esimerkiksi REST-rajapintaa käyttäviä web-sovelluksia. Spring Boot-sovellukseen voidaan valita Maven tai Gradle projektinhallintatyökalu, jonka avulla voidaan asentaa erilaisia kirjastoja tai riippuvuuksia (engl. dependency) projektiin. Projektin toteutusvaiheessa käydään läpi syvemmin Spring Bootin eri toimintoja ja ominaisuuksia.

Kuvassa 18 on esitetty Spring Initializr verkkosivu, jonka avulla voidaan luoda helposti Spring Boot-projektin pohja. Initializr:ssa voidaan valita esimerkiksi projektinhallintatyökalu, ohjelmointikieli ja Spring Boot versio. Projektiin liittyviä tietoja kuten projektin nimeä tai kuvausta voidaan muokata. Tarvittavia riippuvuuksia voidaan lisätä projektiin oikeasta reunasta. Riippuvuuksia voidaan lisätä myös projektin luonnin jälkeen. Spring Boot-projekti voidaan ladata zip-tiedostona generate-napilla.

Kuva 18. Spring Boot-projektin luonti Initializr:lla.[26]

Spring Boot-sovelluksessa perusmäärittelyt ovat valmiiksi tehty, mikä vähentää erilaisten konfiguraatioiden luontia. Spring Bootissa on erinomainen tuki yksikkötesteille, joiden avulla testataan sovelluksen toimivuutta. Sovellus sisältää HTTP-palvelimen kuten Tomcat tai Jetty, joiden avulla sovellusta ei tarvitse asentaa erilliselle web-palvelimelle. Spring Bootissa on myös helppo yhdistää eri tietokantoja sovellukseen. Sovelluksen toimintaa voidaan hallita myös erilaisten annotaatioiden (engl. annotation) avulla.[27]

[26] Spring. Spring Initializr.
[27] Mulders, M. 2019. What is Spring Boot?

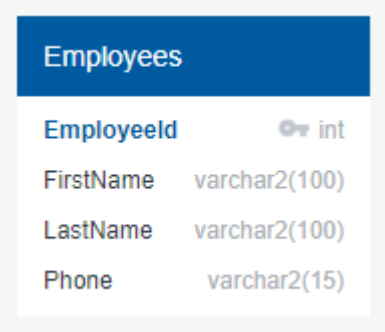
3.3 Oracle Database

Oracle DB on Oraclen kehittämä relaatiotietokanta. Kommunikointi tietokannan kanssa tapahtuu SQL-kyselyjen avulla. SQL-kyselyjen avulla voidaan esimerkiksi hakea, lisätä, päivittää tai poistaa tietokannassa olevaa dataa. Kuvassa 19 on esimerkki SQL-kyselystä, jolla haetaan Employees-taulusta FirstName- ja LastName-kentät. Löydetyt rivit järjestetään FirstName-kentän mukaan.[28]

```
SELECT FirstName, LastName FROM Employees ORDER BY FirstName;
```

Kuva 19. Esimerkki SQL-kyselystä.

Relaatiotietokannassa data tallennetaan tauluihin. Taulussa on sarakkeita, jotka kuvaavat taulun eri kenttiä. Esimerkiksi kuvassa 20 on Employees-taulu, jolla on sarakkeet EmployeeId, FirstName, LastName ja Phone. Sarakkeilla on eri tietotyyppejä kuten int, joka on numeroarvo tai varchar, joka on tekstiarvo.



Kuva 20. Employees-tietokantataulu sekä sen sarakkeet.[29]

Taululle yleensä määritetään pääavain, joka on uniikki avain jokaiselle taulun rivile. Pääavainten avulla voidaan luoda relaatioita eri taulujen välillä. Esimerkiksi kuvassa 21 Contacts-taulussa viitataan Employees-taulun EmployeeId-kenttään.

[28] Buttice, C. 2021. Oracle Database.
[29] Oracle Tutorial. What Is Oracle Database.

Näin taulujen välille syntyy ”one-to-many”-relaatio, mikä tarkoittaa sitä, että yhdellä asiakkaalla voi olla useampi kontakti.

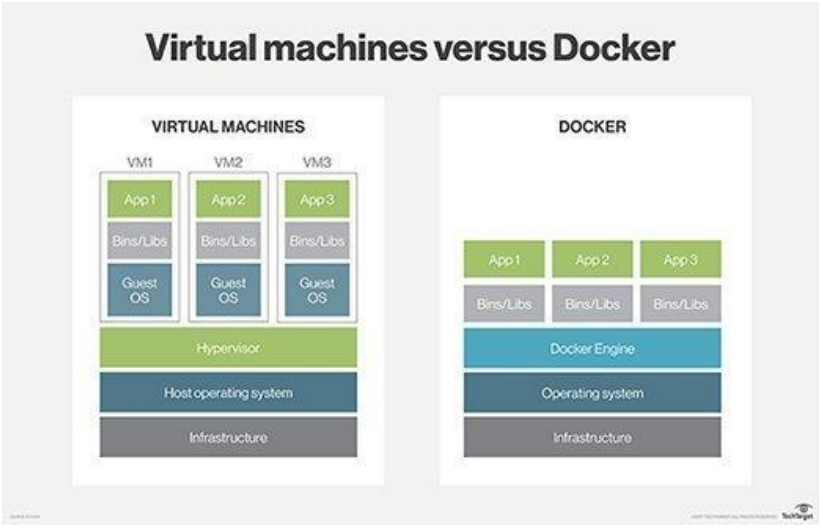
Kuva 21. Kahden tietokantataulun relaatio.[30]

3.4 Docker

Docker on avoimen lähdekoodin ohjelmisto, jonka avulla sovelluksia voidaan pakata virtuaalisiin Docker-kontteihin (engl. container). Docker-kontti on eristetty isäntälaitteesta, mikä tekee siitä ympäristöriippumattoman. Dockerin etuja on pienempi resurssien käyttö sekä nopeampi pystytys verrattuna virtuaalikoneisiin. Dockeria käytetään esimerkiksi tietokantojen pystyttämiseen kehitysympäristössä tai sovelluksen asentamisessa tuotantoympäristöön.[31]

Kuvassa 22 on esitetty virtuaalikoneiden ja Docker-konttien ero. Jokaisella virtuaalikoneella on oma käyttöjärjestelmä, mutta docker-kontit pystyvät jakamaan isäntälaitteen käyttöjärjestelmän. Tämän takia docker-kontit käyttävät vähemmän resursseja.

[30] Oracle Tutorial. What Is Oracle Database.
[31] Bigelow, S. 2020. Docker.



Kuva 22. Dockerin ja virtuaalikoneiden ero.[32]

[32] Bigelow, S. 2020. Docker.

4 PROJEKTIN ARKKITEHTUURI

Tässä luvussa käydään läpi projektin rakennetta sekä REST-arkkitehtuuria ja HTTP-protokollaa.

4.1 Sovelluksen rakenne

Projektin käyttöliittymä toteutettiin käyttäen Angular 13 sovelluskehystä, joka kutsuu palvelinta REST-rajapinnan avulla. Palvelin toteutettiin käyttäen Spring Boot sovelluskehystä sekä Java 17-versiota. Sovelluksen asennusvaiheessa esimerkiksi tuotantoympäristöön sekä käyttöliittymä että palvelin rakennetaan Docker-kontin sisälle, jossa sovellus käynnistetään. Sovellus käyttää Oraclen tietokantaa, johon otetaan yhteyttä palvelimelta käyttäen JPA-rajapintaa. Sovelluksen rakenne on esitetty kuvassa 23.

Kuva 23. Sovelluksen arkkitehtuuri.

4.2 REST-arkkitehtuuri

Palvelin käyttää REST-rajapintaa käyttöliittymän ja palvelimen väliseen kommunikaatioon. REST on arkkitehtuuri, jonka perusteella voidaan luoda verkkopalveluja. Palvelun avulla voidaan lukea tai muokata palvelimella olevia resursseja. Kommunikaatio REST palvelujen kanssa tapahtuu HTTP-protokollan avulla.

Kuvassa 24 on esimerkki HTTP-pyynnöstä, joka lähetetään palvelimelle. HTTP-metodin avulla kerrotaan mitä halutaan tehdä. Taulukossa 1 on kuvattu yleisimmät HTTP-metodit, jota esimerkiksi REST-rajapinnassa käytetään. Pyynnössä URI on

polku palvelimella, johon viitataan. Otsikko (engl. header) sisältää lisätietoja pyynnöstä. Runko (engl. body) on vapaaehtoinen osa, joka sisältää dataa, jota halutaan lähettää palvelimelle. Rungon data on usein JSON formaatissa. Kuvan 24 esimerkki lähettää palvelimelle POST-kutsun /create-user osoitteeseen ja rungossa välitetään name- ja age-kentät, joilla on ”John” ja 35 arvot.[33]

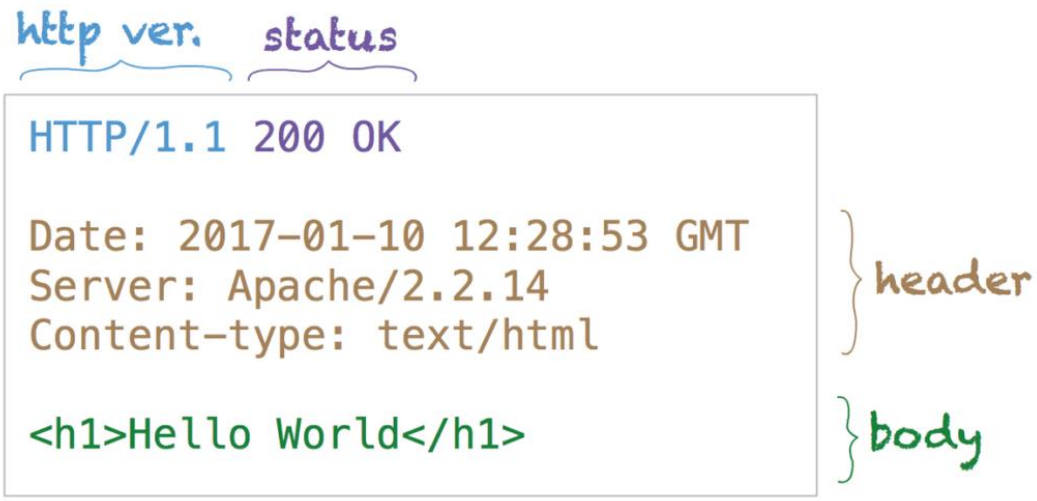
Kuva 24. HTTP-pyynnön rakenne.[34]

Taulukko 1. Neljä yleisintä HTTP-metodia.

HTTP-metodi	Toiminto
GET	Hakee resurssin
POST	Lisää uuden resurssin
PUT	Päivittää resurssin
DELETE	Poistaa resurssin

[33] Matoso, D. 2017. HTTP primer for frontend developers.
[34] Matoso, D. 2017. HTTP primer for frontend developers.

Kuvassa 25 on HTTP-vastaus, joka on rakenteeltaan hieman erilainen kuin pyyntö. Vastauksessa on status, joka kertoo pyynnön statuksen. Nämä statukset (engl. status code) ovat kolminumeroisia ja ne kategorioidaan ensimmäisen numeron perusteella. Taulukossa 2 on eri statustentarkoituksia.



Kuva 25. HTTP-vastauksen rakenne.[35]

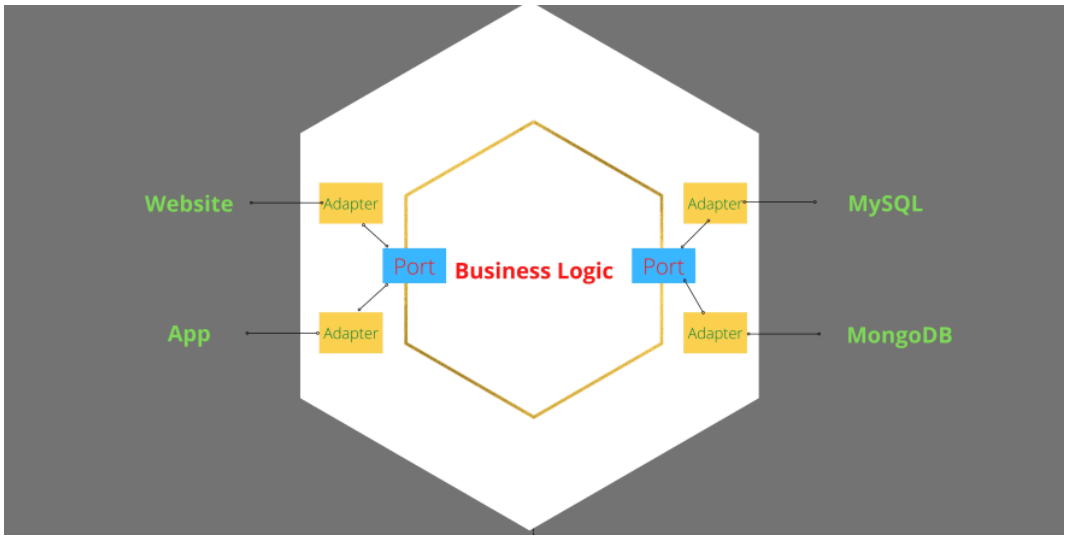
Taulukko 2. HTTP-tilakoodien tarkoitukset.

Statuksen numero	Tarkoitus	Englanniksi
1xx	Informaatiota	Informational
2xx	Onnistunut	Successful
3xx	Uudelleenohjaus	Redirection
4xx	Asiakaspuolen virhe	Client error
5xx	Palvelinpuolen virhe	Server error

[35] Matoso, D. 2017. HTTP primer for frontend developers.

4.3 Heksagonimainen arkkitehtuuri

Sovelluksen palvelimen kehityksessä käytettiin heksagonimaista arkkitehtuuria (engl. hexagonal architecture). Arkkitehtuurin ideana on erotella sovelluksen rakenne eri osiin siten, että sovelluksen ydin on erillään sen ulkopuolisilta osilta. Ydin sisältää sovelluksen logiikan, joka kommunikoi eri kerrosten kanssa porttien avulla kuten kuvassa 26 on esitetty. Tämän erottelun avulla logiikka pysyy erillään, jolloin esimerkiksi tietokanta voidaan vaihtaa ilman suuria muutoksia.[36]



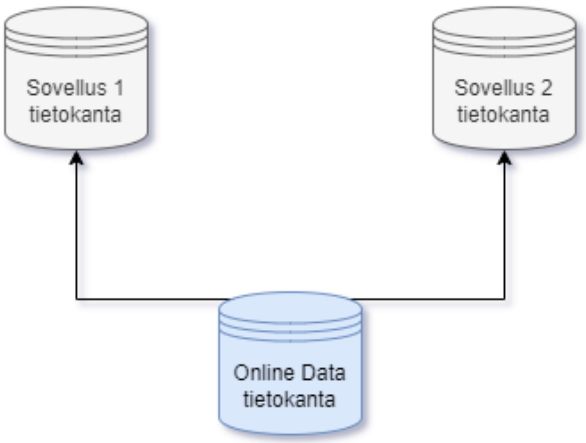
Kuva 26. Esimerkkirakenne heksagonimaisesta arkkitehtuurista.[37]

4.4 Tietokantarakenne

Online Datalle luodaan tietokantaan uusi käyttäjä, jolle annetaan oikeudet lukea kahden muun sovelluksen tietokantatauluja. Näin uusi käyttäjä voi hakea kahdesta eri tietokannasta dataa yhdellä haulla. Kuvassa 27 on esitetty sovelluksen tietokantarakenne. Oraclen tietokannoissa SELECT-oikeus voidaan antaa kyselyllä

[36] Pandey, A. 2021. Hexagonal Architecture.
[37] Pandey, A. 2021. Hexagonal Architecture.

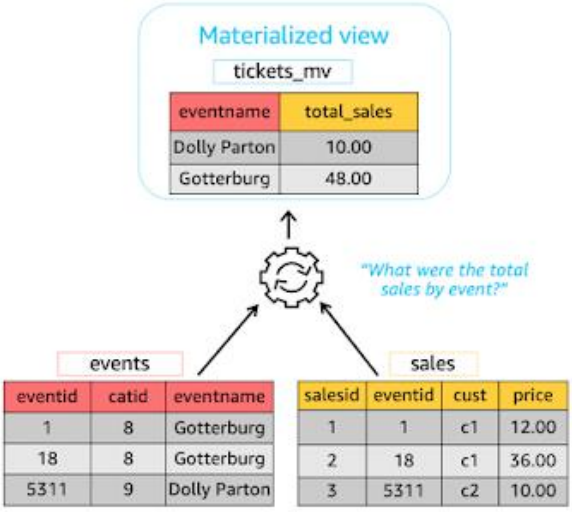
GRANT SELECT ON taulu TO käyttäjä. Online Datan tietokantaan varastoidaan luodut raportit sekä tietoja tietokantatauluista, joita raportit voivat käyttää.



Kuva 27. Sovelluksen tietokantakaavio.

Monimutkaisesta SQL-kyselystä voidaan luoda näkymä (engl. view), joka varastoi SQL-kyselyn ja luo virtuaalisen taulun kyselyn perusteella. Näkymää voidaan kutsua myöhemmin uudestaan esimerkiksi kyselyllä *SELECT * FROM näkymä*. Näkymä on virtuaalinen taulu, jossa data ei ole varastoitu minnekään. Näkymästä voidaan hakea ja suodattaa dataa käyttämällä näkymän nimeä samalla tavalla kuin tavallisesta tietokantataulusta. Näkymä on hyödyllinen tilanteissa, jossa dataa halutaan useasta eri taulusta, koska näkymän SQL-kyselyyn ei tarvitse yhdistää toisia tauluja. Jos kyselyn tulokset halutaan varastoida johonkin, voidaan luoda materialisoitu näkymä (engl. materialized view), joka varastoi SQL-kyselyn tulokset tauluun. Kuvassa 28 on esitetty materialisoitu näkymä, jossa otetaan events- ja sales-tauluista dataa.[38]

[38] Java67. Difference between View and Materialized View in Database or SQL?



Kuva 28. Esimerkki materialisoidusta näkymästä.[39]

Materialisoitu näkymä on suorituskyvyltään nopeampi kuin tavallinen näkymä, koska data on varastoitu tauluun. Materialisoitu näkymä pitää päivittää tietyin väliajoin, jotta data pysyy ajan tasalla. Tätä ongelmaa ei ole tavallisella näkymällä.[40]

Projektissa luodaan näkymiä monimutkaisista kyselyistä, joissa haetaan dataa useasta eri taulusta. Raporttiin voidaan valita vain yksi tietokantataulu, siksi monimutkaiset kyselyt muutetaan näkymiksi. Näin käyttäjän ei tarvitse yhdistellä useita tauluja käyttöliittymällä.

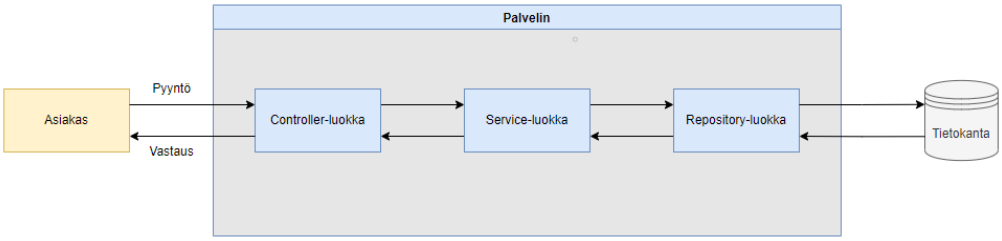
[39] Java67. Difference between View and Materialized View in Database or SQL?
[40] Java67. Difference between View and Materialized View in Database or SQL?

5 SOVELLUKSEN TOTEUTUS

Tässä luvussa käydään läpi sovelluksen toteutusvaihe. Ensin käydään läpi palvelimen kehitystä. Sen jälkeen käyttöliittymän kehitystä sekä ulkoasun rakennetta.

5.1 Palvelimen kehitys

Palvelimen kehityksessä käytetään Spring Boot 2.6.4 versiota. Projektissa käytetään riippuvuuksia kuten Lombok ja Spring Data JPA, joiden avulla helpotetaan ja nopeutetaan sovelluksen kehitystä. Kuvassa 29 on esitetty kaavio palvelimen eri kerrosten välisistä yhteyksistä.



Kuva 29. Asiakkaan, palvelimen sekä tietokannan välinen kommunikaatio.

5.1.1 Konfiguraatio

Palvelin yhdistettiin tietokantaan application.properties-tiedostossa, joka on Spring Boot-sovellusten käyttämä konfiguraatiotiedosto. Konfiguraation avulla voidaan muokata esimerkiksi palvelimen tai eri riippuvuuksien asetuksia. Kuvassa 30 on määritetty tietokanta-asetukset sekä palvelimen porttinumero.

```
server.port=8101

spring.datasource.hikari.driver-class-name=oracle.jdbc.OracleDriver
spring.datasource.hikari.jdbc-url=jdbc\:oracle\:thin\:@localhost\:1522\
spring.datasource.hikari.username=
spring.datasource.hikari.password=
```

Kuva 30. Palvelimen tietokanta- ja porttinumeromääritykset.

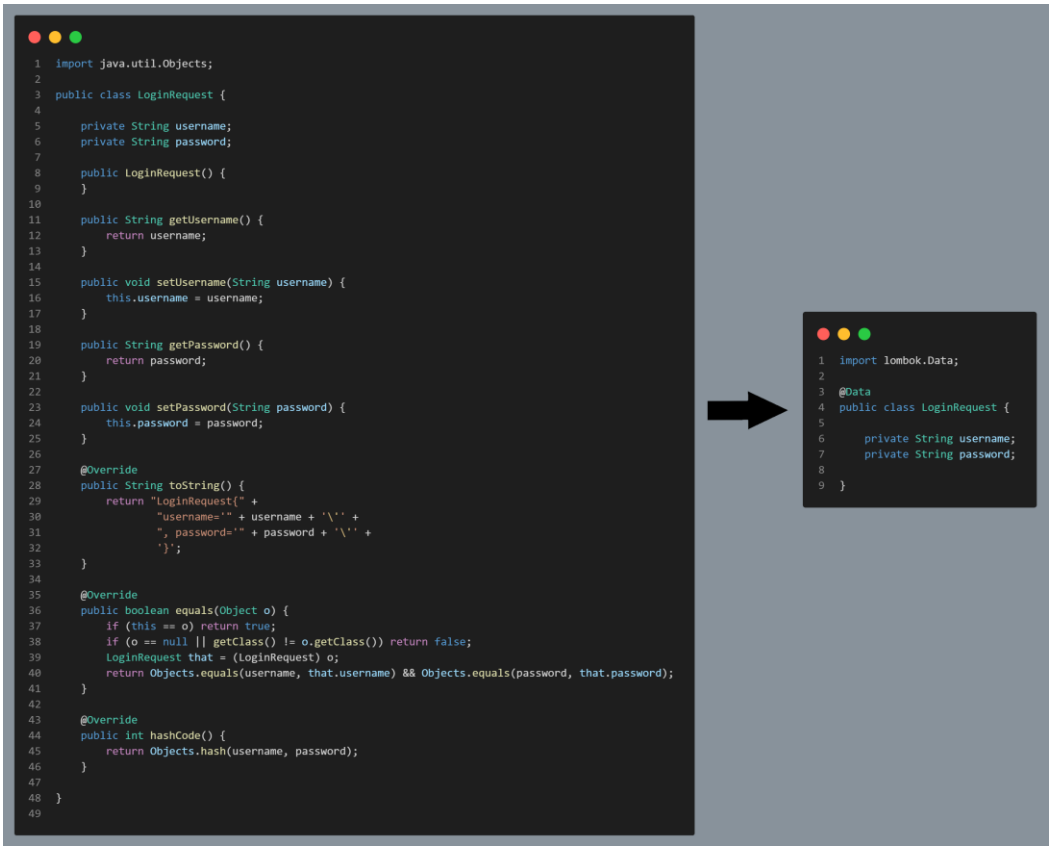
Eri ympäristöille voidaan tehdä oma konfiguraatio tiedosto, esimerkiksi tuotanto-ympäristöön voidaan luoda application-prod.properties -tiedosto. Jos prod-profiili on käytössä, prod-konfiguraatiossa määritetyt asetukset päällekirjoittavat perus-määritykset, jotka ovat application.properties-tiedostossa. Konfiguraation määri-tyksiä voidaan myös injektoida ja käyttää Java-koodissa @Value-annotaation avulla kuvan 31 mukaisesti.

```
@Value("${jwt.secret}")
private String secretKey;
```

Kuva 31. Määrittelyn injektio Java-luokkaan.

5.1.2 Lombok-kirjasto

Projektissa käytetään Lombok-nimistä Java-kirjastoa, jonka avulla voidaan vähen-tää koodin määrää sekä parantaa koodin luettavuutta erilaisten määrittelyjen avulla. Lombokin avulla voidaan generoida automaattisesti esimerkiksi Java-luo-kan konstruktoreja, setter- ja getter-metodeja sekä muita metodeja. Kuvassa 32 on vasemmalla puolella tavallinen Java-luokka ja oikealla Java-luokka, jossa on käytetty Lombokin @Data-annotaatiota. @Data-annotaatio generoi luokan jokai-selle kentälle setter- ja getter-metodit sekä toString()-, equals()- ja hashCode()-metodit. Jos luokalle halutaan esimerkiksi vain getter-metodit, voidaan käyttää @Getter-annotaatiota.



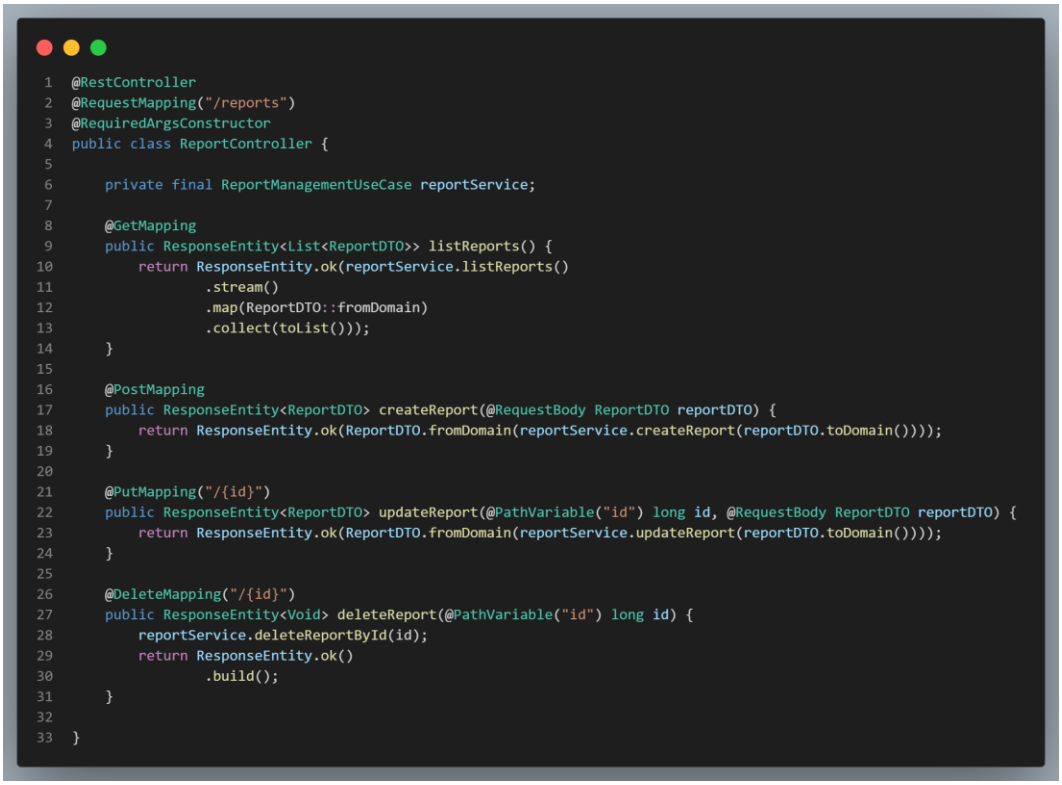
Kuva 32. Tavallisen Java-luokan yksinkertaistaminen Lombokin avulla.

5.1.3 Controller-luokat

Kontrolleri (engl. controller) on luokka, jonka tarkoitus on vastaanottaa HTTP-pyyntöjä. Spring Bootissa Java-luokasta tehdään REST kontrolleri @RestController-annotaatiolla. @RequestMapping-annotaatiolla voidaan asettaa kontrollerin polku, jonka avulla oikea pyyntö ohjataan oikealle kontrollerille. Lombokin @RequiredArgsConstructor-annotaatio generoi luokalle konstruktorin, johon syötetään kaikki pakolliset kentät. Spring käyttää tätä konstruktoria injektioon, jotta esimerkiksi kuvan 33 reportService-luokan metodeja voidaan käyttää kontrollerissa.

Kontrolleri-luokan metodit voidaan määritellä eri HTTP-pyyntöjä varten. Kuvassa 33 on esimerkki kontrolleri luokasta, jossa on määritelty neljä eri metodia raporttien hallintaa varten. Määritelty metodi vastaa yhtä REST-päätepistettä (engl. endpoint), jota esimerkiksi käyttöliittymä voi kutsua. Taulukossa 3 on kuvattu kontrollerin päätepisteet sekä niiden toiminnot. Pyynnön runko voidaan lukea

@RequestBody-annotaatiolla. Dynaamisesti määritetyt polun osat merkataan aaltosulkujen sisälle ja voidaan lukea @PathVariable-annotaation avulla.



```java
@RestController
@RequestMapping("/reports")
@RequiredArgsConstructor
public class ReportController {

    private final ReportManagementUseCase reportService;

    @GetMapping
    public ResponseEntity<List<ReportDTO>> listReports() {
        return ResponseEntity.ok(reportService.listReports()
                .stream()
                .map(ReportDTO::fromDomain)
                .collect(toList()));
    }

    @PostMapping
    public ResponseEntity<ReportDTO> createReport(@RequestBody ReportDTO reportDTO) {
        return ResponseEntity.ok(ReportDTO.fromDomain(reportService.createReport(reportDTO.toDomain())));
    }

    @PutMapping("/{id}")
    public ResponseEntity<ReportDTO> updateReport(@PathVariable("id") long id, @RequestBody ReportDTO reportDTO) {
        return ResponseEntity.ok(ReportDTO.fromDomain(reportService.updateReport(reportDTO.toDomain())));
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteReport(@PathVariable("id") long id) {
        reportService.deleteReportById(id);
        return ResponseEntity.ok()
                .build();
    }

}
```

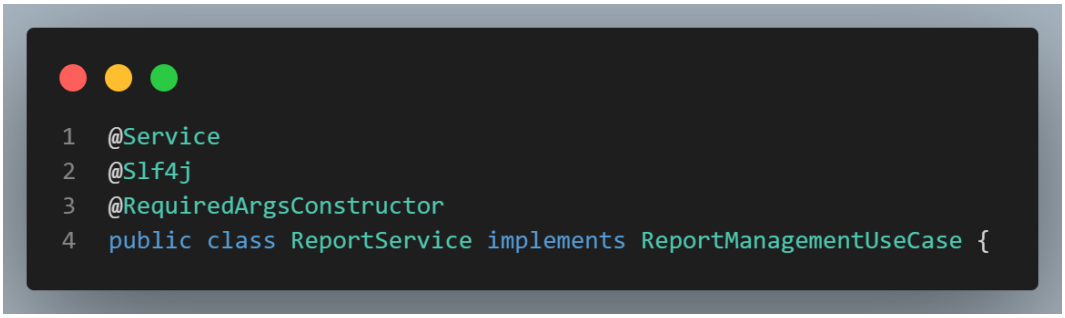
Kuva 33. Kontrolleri-luokka raporttien hallintaan.

Taulukko 3. ReportController-luokan päätepisteet.

REST-päätepiste	Toiminto
GET /reports	Palauttaa kaikki raportit
POST /reports	Tallentaa uuden raportin
PUT /reports/{id}	Päivittää olemassa olevan raportin id arvon perusteella
DELETE /reports/{id}	Poistaa raportin id arvon perusteella

5.1.4 Service-luokat

Service-luokat sisältävät sovelluksen logiikan. Service-kerros on usein controller- ja repository-luokkien välissä. Spring Boot-sovelluksissa service-luokat määritellään usein @Service-annotaatiolla kuten kuvassa 34. Tämän määrittelyn avulla Spring Boot luo sovelluksen käynnistysvaiheessa instanssin luokasta sovelluksen kontekstiin, jota muut luokat voivat käyttää. Lombokin @Slf4j-annotaatiolla voidaan lisätä luokalle logittaja, jonka avulla voidaan esimerkiksi tulostaa virheilmoituksia palvelimen konsoliin.



Kuva 34. Service-luokan määrittely.

5.1.5 Entity-luokat

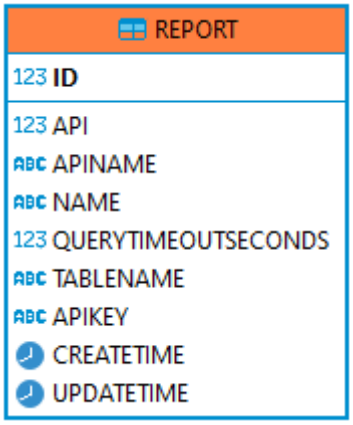
Palvelimen ja tietokannan kommunikointiin käytetään Spring Data JPA- sekä Hibernate-riippuvuuksia. JPA on vain rajapinta, jolla ei ole toteutusta, siksi tarvitaan myös Hibernate, joka on ORM-työkalu, jonka avulla voidaan luoda tietokantatauluja suoraan Java-luokista.

Java-luokasta tehdään entity-luokka @Entity-annotaatiolla kuten kuvassa 35 on esitetty. Tietokantataulun nimi voidaan määritellä @Table-annotaatiolla. Jos nimeä ei ole määritelty, asetetaan taulun nimeksi Java-luokan nimi. @Id-annotaatiolla asetetaan pääavain, joka on usein numeerinen arvo. Tietokanta generoi uudelle entitylle id:n automaattisesti. Jos entity-luokkaan lisätään uusi kenttä tai poistetaan olemassa oleva kenttä, JPA päivittää tietokantataulua automaattisesti muutosten mukaisesti. Kuvassa 36 on esitetty ReportJPA-luokasta generoitu tietokantataulu.

Raportin entity-luokassa käytetään JPA:n @EntityListeners-annotaatiota, jonka avulla voidaan suorittaa erilaisia toimintoja ennen tietokantaan tallentamista tai tallentamisen jälkeen. @CreateDate-annotaatiolla lisätään päivämäärä luokalle vain ensimmäisellä tallennuskerralla. @LastModifiedDate-annotaatiolla update-Time-kenttä päivitetään uudella päivämäärällä joka kerta, kun entity tallennetaan tietokantaan. Sovelluksen pää- tai konfiguraatioluokkaan pitää lisätä @EnableJpaAuditing-annotaatio, joka laittaa auditoinnin päälle.

```java
@Entity
@EntityListeners(AuditingEntityListener.class)
@Table(name = "Report")
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class ReportJpa {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "REPORT_SEQ_GEN")
    @SequenceGenerator(name = "REPORT_SEQ_GEN", sequenceName = "REPORT_SEQ_GEN", allocationSize = 1)
    private long id;

    @NonNull
    @Column(unique = true)
    private String name;

    @NonNull
    private String tableName;

    @NonNull
    private boolean api;

    @Column(unique = true)
    private String apiKey;

    @Column(unique = true)
    private String apiName;

    @NonNull
    private int queryTimeoutSeconds;

    @CreatedDate
    @Column(updatable = false)
    private LocalDateTime createTime;

    @LastModifiedDate
    private LocalDateTime updateTime;

    @OneToMany(mappedBy = "report", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<ReportColumnJpa> columns;

}
```

Kuva 35. Raportin Entity-luokka.



Kuva 36. Entity-luokasta generoitu tietokantataulu.

5.1.6 Repository-luokat

Spring Data JPA:n avulla pystytään helposti luomaan repository-rajapinta (engl. interface), jonka avulla voidaan hakea, tallentaa, päivittää tai poistaa entity-luokkia. Repository-rajapinta voidaan tehdä kuvan 37 mukaisesti. JpaRepository-rajapinnalle annetaan tyyppiparametreina entity-luokka sekä entity-luokan pääavaimen tyyppi. Repository-rajapinnalle voidaan tehdä omia metodeja. Esimerkiksi kuvassa 38 on findByName-metodi, joka hakee raportin name-kentän perusteella. Taulukossa 4 on listattu osa JpaRepository:n metodeista.

```
1  public interface ReportJpaRepo extends JpaRepository<ReportJpa, Long> {
2
3      ReportJpa findByName(String reportName);
4
5      ReportJpa findByApiName(String apiName);
6
7  }
```

Kuva 37. Repository-rajapinta raporteille.

Taulukko 4. JpaRepositoryn metodeja.

Metodi	Toiminto
save(entity)	Tallentaa parametrina annetun entity-luokan
findAll()	Hakee kaikki entityt tietokannasta
findById(id)	Hakee entityn id:n perusteella
deleteAll()	Poistaa kaikki entityt tietokannasta
deleteById(id)	Poistaa entityn id:n perusteella

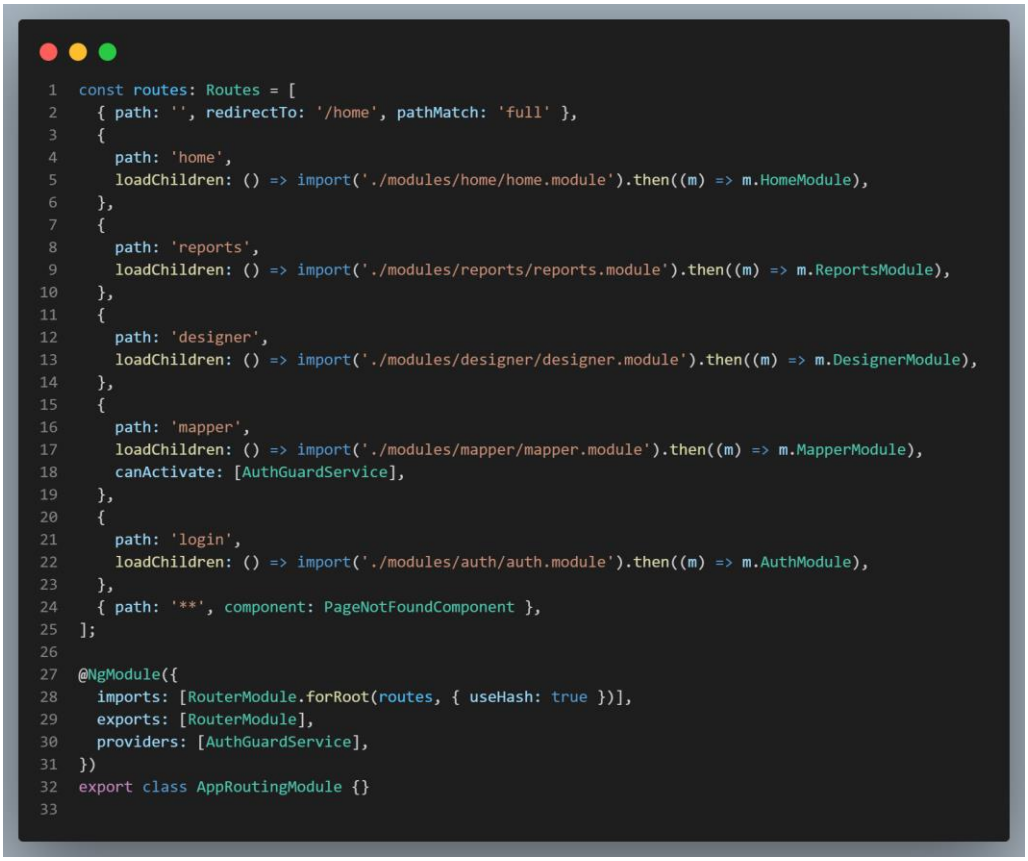
5.2 Käyttöliittymän kehitys

Käyttöliittymän kehitykseen käytettiin Angular versiota 13.3.0. Projektissa käytettiin myös Angular Material-komponenttikirjastoa sekä RxJS-kirjastoa. Käyttöliittymän tyylitykseen käytetään Angular Material-komponenttikirjastoa, jossa on valmiiksi tyylitettyjä nappeja, tekstikenttiä, taulukkoja sekä muita komponentteja. Kirjaston mukana tulee myös valmiiksi luotuja väriteemoja, joita voidaan helposti vaihtaa tai muokata.

5.2.1 Reititys

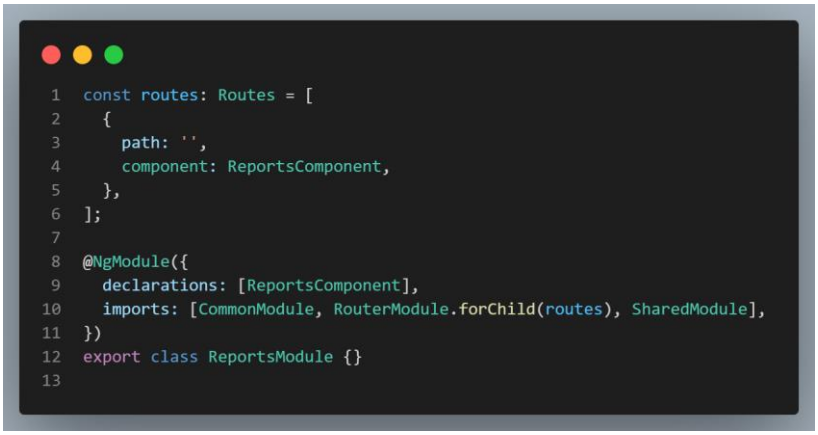
Kuvassa 38 on reititysmoduuli, jossa on määritelty käyttöliittymän eri polut. Reitityksessä käytetään ”Lazy loading”-toimintoa, jonka avulla moduulin JavaScript-tiedostot ladataan vasta silloin, kun käyttäjä navigoi siihen liittyvään näkymän. Tämä nopeuttaa sovelluksen latausaikoja, koska vain tarvittavat JavaScript-tiedostot ladataan. Jos käyttäjä navigoi näkymälle, jota ei ole olemassa, näytetään PageNotFound-näkymä.

Esimerkiksi käyttäjän navigoituessa reports-näkymälle ladataan ReportsModule-moduuliin liittyvät koodit. Kuvassa 39 on ReportsModule, jossa on määritelty ReportsComponent-komponentti, joka näytetään, kun siirrytään reports-näkymälle.



```
const routes: Routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  {
    path: 'home',
    loadChildren: () => import('./modules/home/home.module').then((m) => m.HomeModule),
  },
  {
    path: 'reports',
    loadChildren: () => import('./modules/reports/reports.module').then((m) => m.ReportsModule),
  },
  {
    path: 'designer',
    loadChildren: () => import('./modules/designer/designer.module').then((m) => m.DesignerModule),
  },
  {
    path: 'mapper',
    loadChildren: () => import('./modules/mapper/mapper.module').then((m) => m.MapperModule),
    canActivate: [AuthGuardService],
  },
  {
    path: 'login',
    loadChildren: () => import('./modules/auth/auth.module').then((m) => m.AuthModule),
  },
  { path: '**', component: PageNotFoundComponent },
];

@NgModule({
  imports: [RouterModule.forRoot(routes, { useHash: true })],
  exports: [RouterModule],
  providers: [AuthGuardService],
})
export class AppRoutingModule {}
```

Kuva 38. AppRoutingModule-reititysmoduuli.



```
const routes: Routes = [
  {
    path: '',
    component: ReportsComponent,
  },
];

@NgModule({
  declarations: [ReportsComponent],
  imports: [CommonModule, RouterModule.forChild(routes), SharedModule],
})
export class ReportsModule {}
```

Kuva 39. ReportsModule-moduuli.

5.2.2 Yhteys palvelimeen

Angularissa luodaan usein service-luokkia, joissa otetaan yhteyttä palvelimeen. Angularin HttpClient-luokkaa käytetään HTTP-pyyntöjen tekemiseen. Kuvassa 40 on ReportService-luokka, jolla otetaan yhteyttä palvelinpuolen ReportsController-luokkaan, joka on esitetty aiemmin kuvassa 33. ReportService-luokka voidaan injektoida komponenttiin, jotta se voi käyttää service-luokan funktioita sekä näyttää palvelimelta palautetun datan käyttöliittymässä.

Kuvassa 40 on loadReports()-funktio, joka tekee GET-kutsun /reports-osoitteeseen. Kutsun tehtävänä on palauttaa palvelimelta kaikki raportit. Jos pyyntö epäonnistuu, käyttäjälle näytetään virheilmoitus. Funktiossa käytetään RxJS-kirjaston pipe- ja tap-funktioita. Pipe-funktiolle syötetään lista funktioita, jotka suoritetaan syöttämisjärjestyksessä. Tap-funktion avulla voidaan suorittaa erilaisia toimintoja esimerkiksi, kun pyyntö on valmis tai se epäonnistuu.

```
@Injectable({
  providedIn: 'root',
})
export class ReportService {
  constructor(
    private http: HttpClient,
    private notificationService: NotificationService,
    @Inject('API_URL') private apiUrl: string
  ) {}

  loadReports(): Observable<Report[]> {
    return this.http.get<Report[]>(`${this.apiUrl}/reports`).pipe(
      tap({
        error: () => {
          this.notificationService.showError('Failed to load the reports');
        },
      })
    );
  }

}
```

Kuva 40. Angular service-luokka raporttien hakua varten.

5.2.3 Komponentin toiminta

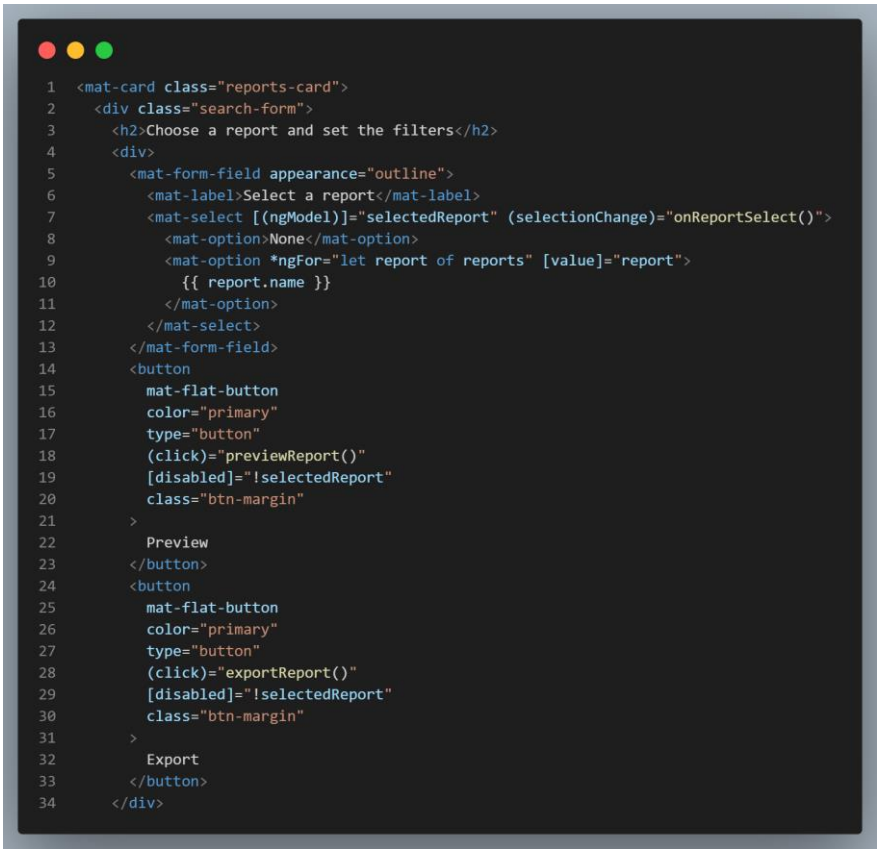
Kuvassa 41 on esitetty osa ReportsComponent-komponenttiluokasta. Komponentissa käytetään ngOnInit()-funktiota, joka on yksi Angular-komponenttien ”lifecycle”-funktioista. Kun komponentti näytetään käyttöliittymällä, ngOnInit()-funktio suoritetaan kerran. Komponentin pitää toteuttaa (engl. implements) OnInit-rajapintaa, jotta ngOnInit()-funktioita voidaan käyttää. Komponentin ngOnInit()-funktiossa haetaan kaikki raportit käyttäen ReportService-luokkaa, joka on injektoitu konstruktorilla. Palvelimelta palautetut raportit asetetaan reports-muuttujaan. Kun raportti valitaan käyttöliittymällä, suoritetaan onReportSelect()-funktio, joka asettaa valitun raportin suodattimet käyttöliittymän taulukkoon.

```
@Component({
  selector: 'app-reports',
  templateUrl: './reports.component.html',
  styleUrls: ['./reports.component.scss'],
})
export class ReportsComponent implements OnInit {
  reports: Report[];
  selectedReport: Report;

  reportFilters = new MatTableDataSource<ReportColumn>();
  reportFilterColumns: string[] = [
    'tableName',
    'columnName',
    'displayName',
    'dataType',
    'filter',
    'filterOnly',
  ];

  constructor(private reportService: ReportService) { }

  ngOnInit(): void {
    this.reportService.loadReports().subscribe((data: Report[]) => {
      this.reports = data;
    });
  }

  onReportSelect() {
    this.reportFilters.data = this.selectedReport.columns;
  }
```

Kuva 41. Reports-komponentin TypeScript-tiedosto.

Kuvassa 42 on osa reports-komponentin HTML-tiedostosta. Tiedostossa määritellään Angular Material-komponenttikirjaston <mat-select>-valintalaatikko, johon asetetaan kaikki raportit. *ngFor-direktiivillä luodaan <mat-option>-elementti jokaiselle raportille. Select-laatikkoon on määritetty [(ngModel)]-direktiivi, jolla voidaan asettaa käyttöliittymällä valittu raportti ohjelmakoodin selectReport-muuttujaan. Kun valintaa muutetaan, onReportSelect()-funktio suoritetaan.

Raportin esikatselu ja lataus toiminnoille luodaan omat napit käyttöliittymällä. Kun esikatselu-nappia painetaan, suoritetaan previewReport()-funktio, joka palauttaa raportin pohjalta löydettyä dataa ja asettaa ne käyttöliittymän taulukkoon. Kun raportin latausta varten olevaa nappia painetaan, suoritetaan exportReport()-funktio, joka palauttaa raportin datan CSV-tiedostona. Molemmat napit ovat pois päältä, jos raporttia ei ole valittu.



```
<mat-card class="reports-card">
  <div class="search-form">
    <h2>Choose a report and set the filters</h2>
    <div>
      <mat-form-field appearance="outline">
        <mat-label>Select a report</mat-label>
        <mat-select [(ngModel)]="selectedReport" (selectionChange)="onReportSelect()">
          <mat-option>None</mat-option>
          <mat-option *ngFor="let report of reports" [value]="report">
            {{ report.name }}
          </mat-option>
        </mat-select>
      </mat-form-field>
      <button
        mat-flat-button
        color="primary"
        type="button"
        (click)="previewReport()"
        [disabled]="!selectedReport"
        class="btn-margin"
      >
        Preview
      </button>
      <button
        mat-flat-button
        color="primary"
        type="button"
        (click)="exportReport()"
        [disabled]="!selectedReport"
        class="btn-margin"
      >
        Export
      </button>
    </div>
```

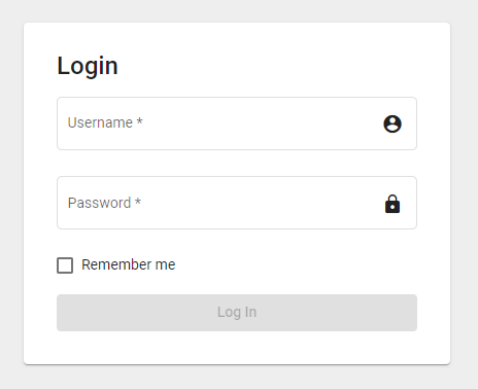
Kuva 42. Reports-komponentin HTML-tiedosto.

5.3 Käyttöliittymän näkymät

Sovelluksen käyttöliittymä rakentuu kuudesta eri näkymästä. Käyttäjä ohjataan kirjautumisnäkymälle, jos käyttäjä ei ole vielä kirjautunut sisään. Home-näkymällä on tietoa sovelluksesta sekä sen käyttämisestä.

5.3.1 Kirjautuminen

Sovellus käyttää asiakkaan omaa käyttäjänhallintasovellusta sisäänkirjautumiseen. Käyttäjillä on tiettyjä oikeuksia, joiden avulla voidaan rajoittaa esimerkiksi mille näkymille käyttäjä päästetään tai mitä toimintoja käyttäjä voi tehdä. Sovelluksen sisäänkirjautumisnäkymä on esitetty kuvassa 43. Kirjautumisen jälkeen käyttäjän oikeuksien mukaiset navigaatiovaihtoehdot tulevat näkyviin.



Kuva 43. Kirjautumisnäkymä.

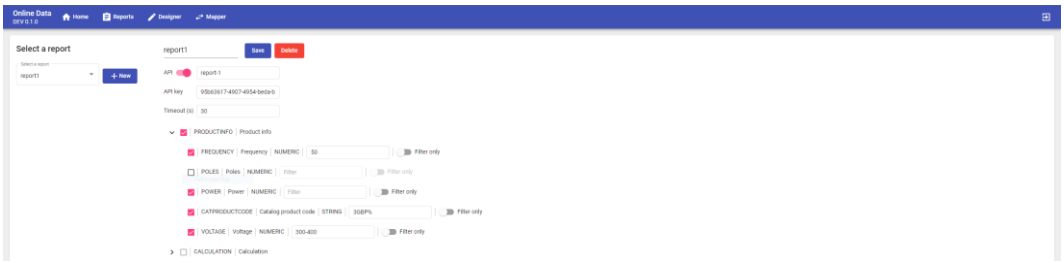
5.3.2 Mapper

Mapper-näkymässä käyttäjä voi helposti lisätä raporteille käytettäväksi uuden tietokantataulun sekä taulun sarakkeet. Näkymällä voidaan valita käytetyt sarakkeet sekä raporteissa käytettyä displayName-kenttää voidaan muuttaa. Näkymä on tarkoitettu vain admin- tai kehittäjäkäyttäjille. Kuvassa 45 on haettu PRODUCTINFO-taulun sarakkeet, joista osa on valittu raporttien käyttöä varten.

Kuva 44. Mapper-näkymä.

5.3.3 Designer

Designer-näkymässä käyttäjä voi luoda, muokata tai poistaa raportteja. Kun raportti on valittu, näkymän oikealla puolella näytetään raportin tiedot. Raporttia voidaan käyttää julkisen APIn kautta, jos API-slider on päällä ja raportille on annettu API nimi. API-avain generoidaan automaattisesti, kun raportti tallennetaan ensimmäistä kertaa. API-avainta käytetään julkisen APIn pyynnöissä. Käyttäjä ei voi muokata avainta jälkikäteen. Timeout-kentässä voidaan asettaa tietokantahaun aikarajoitus. Käyttäjä voi valita raportille yhden tietokantataulun sekä mitkä taulun sarakkeet raporttiin sisällytetään. Jokaiselle sarakkeelle on oma tekstikenttä suodatusta varten.



Kuva 45. Designer-näkymä.

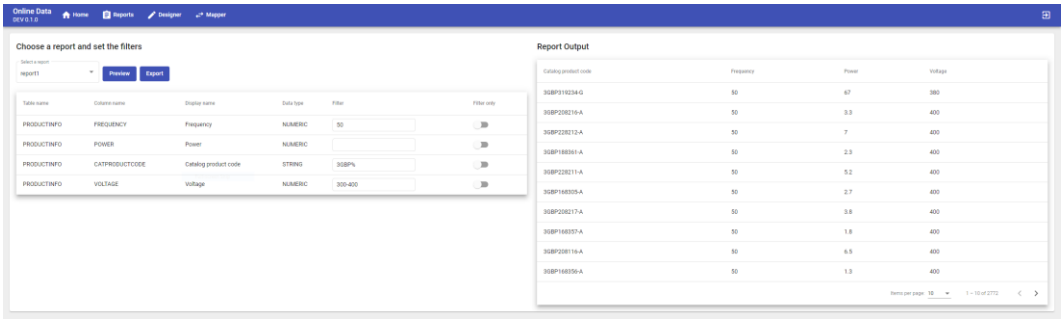
Raportille on kuvassa 46 valittu PRODUCTINFO-taulu sekä FREQUENCY, POWER, CATPRODUCTCODE ja VOLTAGE sarakkeet. Taulun sarakkeita voidaan suodattaa tekstikentän avulla. Taulukossa 5 on listattu käytettävät suodatustoiminnot. Wildcard-suodatuksessa prosenttimerkki voi olla myös tekstin alussa tai välissä ja sitä voidaan käyttää useassa eri kohdassa. Suodatustoimintoja voidaan yhdistää, esimerkiksi voidaan suodattaa dataa listalla, jossa käytetään wildcard-toimintoa.

Taulukko 5. Raportin sarakkeiden suodatustoimintoja.

Suodatin	Esimerkki	Selitys
Lista suodatus	10, 20, 30	Hakee kaikki rivit, jossa sarakkeen arvo on 10, 20 tai 30
Raja-arvo suodatus	100-200	Hakee kaikki rivit, jossa sarakkeen arvo on 100 ja 200 välillä
Wildcard-suodatus	3GBP%	Hakee kaikki rivit, jossa sarakkeen arvo alkaa 3GBP-merkeillä

5.3.4 Reports

Reports-näkymässä, joka on esitetty kuvassa 47, näytetään kaikki saatavilla olevat raportit. Näistä raporteista käyttäjä voi valita haluamansa ja tulostaa raportin datan käyttöliittymään käyttäen preview-nappia tai ladata CSV-tiedoston export-nappia käyttäen. Raportin eri suodattimia voidaan myös muokata tai poistaa käytöstä tässä näkymässä. ”Filter only”-sliderilla voidaan suodattaa kenttää, mutta sen arvoja ei oteta mukaan raporttiin. Kuvassa 47 oikealla puolella oleva taulukko tulee näkyviin preview-napista painamalla. Taulukossa näytetään tietokannasta löydettyä dataa määritettyjen suodattimien perusteella. Export-nappi lataa kuvan 48 kaltaisen CSV-tiedoston, joka sisältää raportin datan.



Kuva 46. Reports-näkymä.

	A	B	C	D
1	Catalog product code	Frequency	Power	Voltage
2	3GBP319234-G	50	67	380
3	3GBP208216-A	50	3.3	400
4	3GBP228212-A	50	7	400
5	3GBP188361-A	50	2.3	400
6	3GBP228211-A	50	5.2	400
7	3GBP168305-A	50	2.7	400
8	3GBP208217-A	50	3.8	400
9	3GBP168357-A	50	1.8	400
10	3GBP208116-A	50	6.5	400
11	3GBP168356-A	50	1.3	400
12	3GBP168360-A	50	7.5	400

Kuva 47. Raportti avattuna Excelissä.

5.4 Testaus

Sovellusten testauksella varmistetaan, että ohjelma toimii oikealla tavalla. Spring Boot sovelluksia voidaan testata helposti lisäämällä spring-boot-starter-test riippuvuus, jonka mukana tulee testausta varten erilaisia kirjastoja kuten JUnit ja Mockito. Spring Boot sovelluksen eri kerroksia voidaan testata käyttäen näiden kirjastojen annotaatioita.

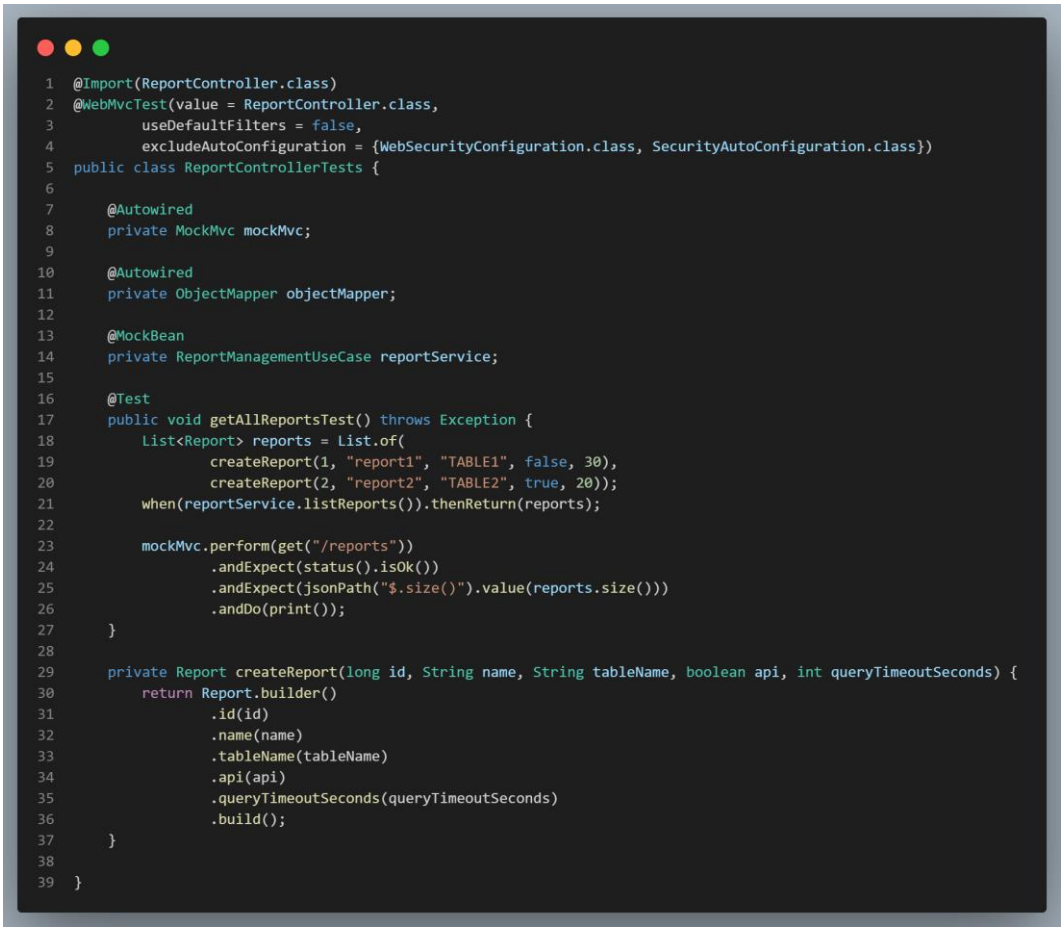
Testaus voidaan automatisoida erilaisten työkalujen avulla. Esimerkiksi lisättäessä uutta koodia GitHub tai Bitbucket versionhallintaan, voidaan testit suorittaa automaattisesti. Näin varmistetaan, että tehdyt muutokset eivät riko sovelluksen toimintaa.

5.4.1 Controller-kerroksen testaus

Controller-luokkia voidaan testata käyttäen @WebMvcTest-annotaatiota. Annotaatiolle annetaan testattava luokka ja voidaan poistaa käytöstä autentikointiin liittyvät suodattimet ja konfiguraatiot. Kuvassa 49 on testiluokka ReportController-luokalle, jossa testataan esimerkiksi kaikkien raporttien hakua. Testattaville metodeille lisätään @Test-annotaatio ja palautustyypiksi void, koska testimetodit eivät yleensä palauta mitään.

Kuvan 49 getAllReportsTest()-metodissa luodaan lista, jossa on kaksi raporttia testausta varten. Mockito-kirjaston when- ja thenReturn-metodeilla voidaan palauttaa haluttuja arvoja, kun tiettyä metodia kutsutaan. Esimerkiksi, kun controller-luokka kutsuu reportService.listReports()-metodia, palautetaan aikaisemmin luodut raportit.

MockMvc-luokalla voidaan suorittaa erilaisia HTTP-pyyntöjä ja voidaan testata palauttaako HTTP-vastaus oikean arvon. Kuvassa 49 suoritetaan GET-pyyntö /reports-polkuun ja tarkistetaan andExpect()-metodilla, että vastauksen status on ok ja rungon objektien lukumäärä on sama kuin luotujen raporttien määrä. Lopuksi pyyntö sekä vastaus tulostetaan konsoliin print()-metodilla. Jos yksikin ehdoista ei toteudu, testi menee epäonnistunut-tilaan.



```java
@Import(ReportController.class)
@WebMvcTest(value = ReportController.class,
        useDefaultFilters = false,
        excludeAutoConfiguration = {WebSecurityConfiguration.class, SecurityAutoConfiguration.class})
public class ReportControllerTests {

    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private ObjectMapper objectMapper;

    @MockBean
    private ReportManagementUseCase reportService;

    @Test
    public void getAllReportsTest() throws Exception {
        List<Report> reports = List.of(
                createReport(1, "report1", "TABLE1", false, 30),
                createReport(2, "report2", "TABLE2", true, 20));
        when(reportService.listReports()).thenReturn(reports);

        mockMvc.perform(get("/reports"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.size()").value(reports.size()))
                .andDo(print());
    }

    private Report createReport(long id, String name, String tableName, boolean api, int queryTimeoutSeconds) {
        return Report.builder()
                .id(id)
                .name(name)
                .tableName(tableName)
                .api(api)
                .queryTimeoutSeconds(queryTimeoutSeconds)
                .build();
    }

}
```

Kuva 48. Controller-luokan testaus.

5.4.2 Service-kerroksen testaus

Service-kerroksen testauksessa varmistetaan, että sovelluksen ydinlogiikka toimii oikein. Service-luokkien testauksessa käytetään usein mock-versiota repository-luokista, jotta voidaan keskittyä vain service-kerroksen toimintaan. Mockito-kirjaston eri annotaatioilla ja metodeilla voidaan luoda mock-instansseja luokista sekä palauttaa haluttuja arvoja esimerkiksi luomatta yhteyttä tietokantaan.

Kuvassa 50 on esitetty ReportService-luokan testiluokka. Testiluokassa käytetään @ExtendWith(MockitoExtension.class)-annotaatiota, jonka avulla voidaan luoda ja injektoida mock-instansseja. ReportPort-luokasta luodaan mock-instanssi @Mock-annotaatiolla. Tämä mock-luokka injektoidaan ReportService-luokkaan

@InjectMocks-annotaatiolla, koska ReportService käyttää ReportPort-luokan metodeja.

Testimetodissa getAllReportsTest() luodaan kaksi raporttia testausta varten. Mockito-kirjaston when- ja thenReturn-metodeilla palautetaan luodut raportit, kun findAllReports()-metodia kutsutaan reportPort-luokassa. ReportService-luokan listReports()-metodia kutsutaan ja tarkistetaan assertThat()-metodilla, että service-luokka palautti kaksi raporttia. Mockito-kirjaston verify()-metodilla tarkistetaan, että reportPort-luokan findAllReports()-metodia kutsuttiin yhden kerran.

```java
@ExtendWith(MockitoExtension.class)
public class ReportServiceTests {

    @Mock
    private ReportPort reportPort;

    @InjectMocks
    private ReportService reportService;

    @Test
    public void getAllReportsTest() {
        List<Report> reports = List.of(
                createReport(1, "report1", "TABLE1", false, 30),
                createReport(2, "report2", "TABLE2", true, 20));
        when(reportPort.findAllReports()).thenReturn(reports);

        List<Report> foundReports = reportService.listReports();

        assertThat(foundReports).hasSize(2);
        verify(reportPort, times(1)).findAllReports();
    }

    private Report createReport(long id, String name, String tableName, boolean api, int queryTimeoutSeconds) {
        return Report.builder()
                .id(id)
                .name(name)
                .tableName(tableName)
                .api(api)
                .queryTimeoutSeconds(queryTimeoutSeconds)
                .build();
    }

}
```

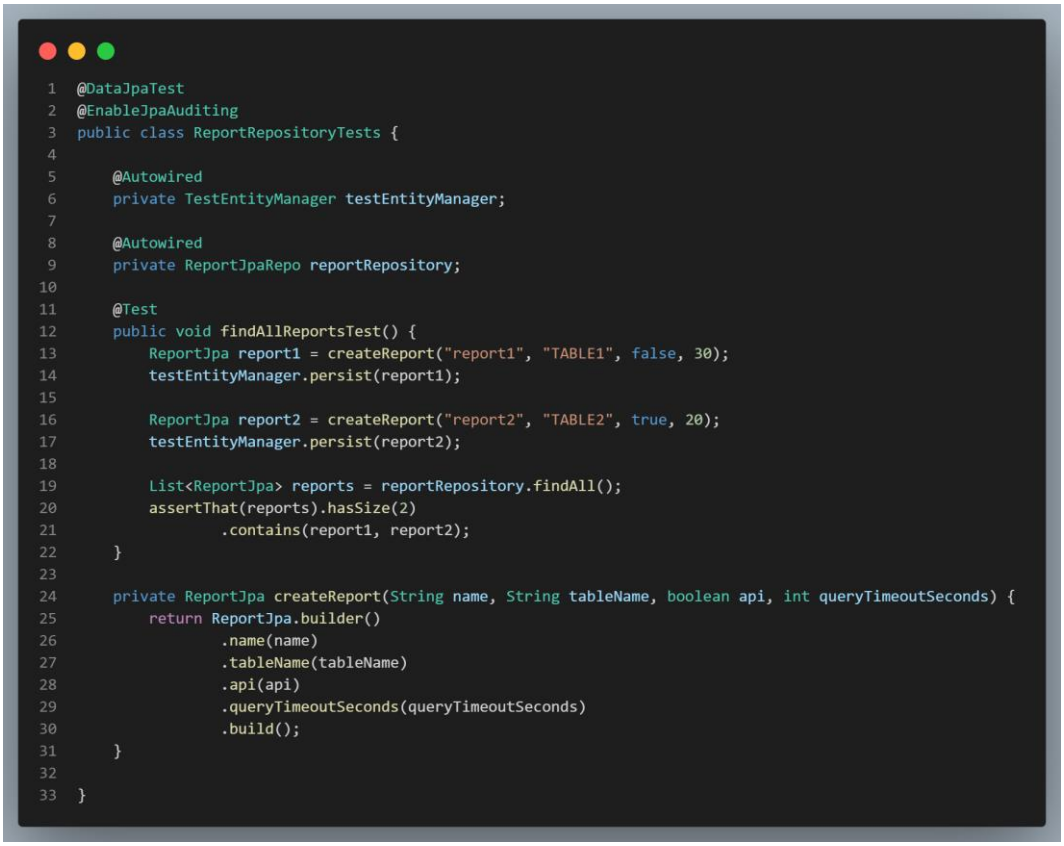
Kuva 49. Service-luokan testaus.

5.4.3 Repository-kerroksen testaus

JPA Repository-luokkia voidaan testata käyttäen @DataJpaTest-annotaatiota kuten kuvassa 51 on esitetty. @EnableJpaAuditing-annotaatiolla aktivoidaan entity-luokkien auditointi testiluokalle. Raporttien repository-luokka sekä TestEntity-

manager injektoidaan testiluokalle @Autowired-annotaatiolla. TestEntity-Manager-luokalla voidaan tallentaa entity-luokkia tietokantaan testausta varten, koska testauksessa käytetty tietokanta on testien alussa tyhjä.

Kuvassa 51 on findAllReportsTest()-metodi, jossa luodaan kaksi raporttia, jotka tallennetaan tietokantaan. Tämän jälkeen suoritetaan reportRepository.findAll()-metodi, joka hakee tietokannasta kaikki raportit. Lopuksi tarkistetaan assertThat()-metodilla, että tietokannasta löytyy kaksi raporttia, jotka ovat samat kuin testin alussa luodut raportit.



```
@DataJpaTest
@EnableJpaAuditing
public class ReportRepositoryTests {

    @Autowired
    private TestEntityManager testEntityManager;

    @Autowired
    private ReportJpaRepo reportRepository;

    @Test
    public void findAllReportsTest() {
        ReportJpa report1 = createReport("report1", "TABLE1", false, 30);
        testEntityManager.persist(report1);

        ReportJpa report2 = createReport("report2", "TABLE2", true, 20);
        testEntityManager.persist(report2);

        List<ReportJpa> reports = reportRepository.findAll();
        assertThat(reports).hasSize(2)
                .contains(report1, report2);
    }

    private ReportJpa createReport(String name, String tableName, boolean api, int queryTimeoutSeconds) {
        return ReportJpa.builder()
                .name(name)
                .tableName(tableName)
                .api(api)
                .queryTimeoutSeconds(queryTimeoutSeconds)
                .build();
    }

}
```

Kuva 50. Repository-luokan testaus

6 YHTEENVETO

Työn tarkoituksena oli kehittää Online Data raportointityökalu Wapice Oy:n asiakkaalle. Työkalun avulla käyttäjä voi hakea dataa kahdesta eri tietokannasta erilaisten suodattimien avulla. Raporttiin voidaan valita yksi tietokantataulu sekä suodattaa taulun sarakkeita tekstikentän avulla. Raportti hakee tietokannasta dataa valittujen sarakkeiden ja suodattimien perusteella ja palautta CSV-tiedoston, joka sisältää tietokannasta haetun datan.

Tämä oli ensimmäinen työprojekti, jota kehitin suurimmaksi osaksi itse. Esimies loi projektille pohjan, jonka avulla pääsin nopeasti alkuun. Minulla oli aikaisempaa kokemusta käytetyistä teknologioista, joten ymmärsin niiden perusteet. Taustatutkimuksen avulla sain teoreettisemman ymmärryksen esimerkiksi Spring Boot sovellusten rakenteesta sekä erilaisten annotaatioiden käytöstä. Esimerkiksi koodin määrää saatiin vähennettyä huomattavasti Lombok-kirjaston avulla. Angularia käyttäessä paransin TypeScript-osaamistani ja opin käyttämään esimerkiksi ”Lazy loading”-toimintoa. Yksi projektin haasteista oli monimutkaisten SQL-kyselyiden käyttö raporteissa. Tämä ongelma saatiin ratkottua käyttämällä Oraclen tietokantanäkymiä.

Projekti on tällä hetkellä toimintakunnossa ja asennettu testipalvelimelle asiakkaan testausta varten. Projektille tarvitaan vielä tulevaisuudessa uusia toimintoja kuten ulkoinen API, jonka avulla toiset sovellukset voivat hakea dataa raporteista suodattimien avulla. Raportin data halutaan myös JSON- ja CSV-formaatissa APIn kautta. Sovellukselle lisätään myös käyttäjäoikeuksia, joilla voidaan esimerkiksi rajoittaa tavallisten käyttäjien toimintaa sovelluksen sisällä.

LÄHTEET

Angular. Angular Components Overview. Viitattu 26.2.2022. https://angular.io/guide/component-overview

Angular. Built-in directives. Viitattu 5.3.2022. https://angular.io/guide/built-in-directives

Angular. CLI Overview and Command Reference. Viitattu 20.3.2022. https://angular.io/cli

Angular. Dependency injection. Viitattu 5.3.2022. https://angular.io/guide/dependency-injection

Angular. Intro to modules. Viitattu 5.3.2022. https://angular.io/guide/architecture-modules

Angular. Intro to services and DI. Viitattu 26.2.2022. https://angular.io/guide/architecture-services

Angular. Ng generate. Viitattu 20.3.2022. https://angular.io/cli/generate

Angular. What is Angular? Viitattu 25.2.2022. https://angular.io/guide/what-is-angular

Altexsoft. 2020. The Good and the Bad of Angular Development. Viitattu 25.2.2022. https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-angular-development/

Bigelow, S. 2020. Docker. Viitattu 24.4.2022. https://www.techtarget.com/searchitoperations/definition/Docker

Buttice, C. 2021. Oracle Database. Viitattu 24.4.2022. https://www.techopedia.com/definition/8711/oracle-database

Garg, B. 2019. Angular Architecture Overview. Viitattu 5.3.2022. https://medium.com/@bhavikagarg8/angular-architecture-overview-1e7cc7483a0

GeeksforGeeks. Introduction to Java. Viitattu 26.3.2022. https://www.geeksforgeeks.org/introduction-to-java/

Java67. Difference between View and Materialized View in Database or SQL? Viitattu 23.4.2022. https://www.java67.com/2012/11/what-is-difference-between-view-vs-materialized-view-database-sql.html

Matoso, D. 2017. HTTP primer for frontend developers. Viitattu 27.4.2022. https://www.webdevdrops.com/en/en/http-primer-for-frontend-developers-f091a2070637/

Mulders, M. 2019. What is Spring Boot? Viitattu 9.4.2022. https://stackify.com/what-is-spring-boot/

Oracle Tutorial. What Is Oracle Database. Viitattu 24.4.2022. https://www.oracletutorial.com/getting-started/what-is-oracle-database/

Pandey, A. 2021. Hexagonal Architecture. Viitattu 23.4.2022. https://dev.to/abh1navv/hexagonal-architecture-3ocl

Spring. Spring Initializr. Viitattu 9.4.2022. https://start.spring.io/

Vishnupriya. 2018. What is Typescript? Viitattu 26.2.2022. https://medium.com/@vishnupriya_web/what-is-typescript-faa0890b2baf

w3schools. Java Introduction. Viitattu 26.3.2022. https://www.w3schools.com/java/java_intro.asp