LAB University of Applied Sciences

Keräilykorttien kirjanpitosovellus

LAB-ammattikorkeakoulu
Insinööri (AMK)
2023
Rami Riekkinen

Tiivistelmä

Tekijä(t)	Julkaisun laji	Valmistumisaika
Rami Riekkinen	Opinnäytetyö, AMK	2023
	Sivumäärä 51	

Työn nimi

Keräilykorttien kirjanpitosovellus

Tutkinto ja koulutusala

Insinööri (AMK), Tieto- ja viestintätekniikka, Ohjelmistotekniikka

Toimeksiantajaorganisaatio (jos opinnäytetyöllä on toimeksiantaja)

Tiivistelmä

Opinnäytetyössä suunniteltiin ja toteutettiin avoimen lähdekoodin full-stack-websovellus Magic the Gathering -keräilykorttien kirjanpitoa varten. Opinnäytetyössä käydään läpi sovelluksen suunnittelu- ja toteutusvaiheet käyttöliittymän, palvelimen ja tietokannan osilta. Myös sovelluksen jatkokehitystä alustetaan.

Sovellus toteutettiin Angular-, Spring Boot- ja NoSQL-teknologiapinolla. Sovelluksen käyttöliittymä toteutettiin Angular-kehyksellä, palvelinsovellus Javan Spring Boot kehyksellä ja tietokanta MongoDB:llä. Sovellus laitettiin myös Docker-konttiin.

Tuloksena saatiin keräilynäkökulmaan painottuva kirjanpitosovellus, johon kirjautumalla käyttäjä voi tarkastella oman kokoelmansa arvoa ja seurata kokoelmansa edistymistä eri näkökulmista. Sovelluksessa käyttäjät pystyivät myös lisäämään tai poistamaan kortteja kokoelmastaan.

Asiasanat

Web-sovellus, Full-Stack, Angular, Spring Boot, MongoDB, Docker

Abstract

Author(s)	Type of Publication	Published
Rami Riekkinen	Thesis, UAS	2023
	Number of Pages 51	

Title of Publication

An accountant application for trading cards

Degree, Field of Study

Engineer (UAS), Information- and Communication Technology, Software Engineering

Organisation of the client (if the thesis work is commissioned by another party)

Abstract

Subject of the thesis was to plan and implement an open-source full-stack web application that works as an accounting application for Magic the Gathering trading cards.

The application was implemented with Angular, Spring Boot and NoSQL technology stack. The user interface of the application was implemented with Angular framework, the server application with Java's Spring Boot framework and the database with MongoDB. The Application was also made to run inside a Docker container.

In the thesis, the design and implementation phases of the application are reviewed form the user interface, server, and database parts. Further development of the application is also initiated.

As a result, an accounting application was made with an emphasis on collecting, whereby logging in, the user could view the value of their own collection and monitor the progress of their collection from different perspectives. In the app, users could also add or remove cards from their collection.

Keywords

Web application, Full-Stack, Angular, Spring Boot, MongoDB, Docker

Sisällys

1 Johdanto

Keräilykortit ovat lähtöisin 1860-luvulta, jolloin ne esiintyivät tupakka-askeihin sisällytettävinä mainoksina (Collectibles Insurance Services 2021). Keräilykortit ovat keränneet suosiotaan niiden alusta lähtien ja ovat tänäkin päivänä erittäin suosittuja. 1980-luvulla keräilykorttien arvo huomattiin ja korttien keräilynäkökulma muuttui, ja tämän takia kortteja alettiin säilöä siinä toivossa, että niiden arvo kasvaa (Collectibles Insurance Services 2021). Nykyhetkellä suosituimpiin keräilykorttisarjoihin kuuluvat 1990-luvulla perustetut Pokemon- sekä Magic the Gathering -pelisarjat. Keräilykorttien suosion jatkuvan kasvun takia korttien määrä kasvaa vuosi vuodelta, ja vuonna 2021 esimerkiksi Magic the Gathering -pelisarjaan kuului noin 23 000 korttia, joilla on uniikki nimi (Davis 2021). Koska korttien määrä on suuri, teknologian kehityksen ohessa pinnalle on nousseet erinäiset webpalvelut, jotka mahdollistavat keräilykorttisarjojen kirjanpidon sekä hintojen ja kokoelmien arvojen seurannan.

Opinnäytetyössä tuli toteuttaa avoimen lähdekoodin Full-Stack web-sovellus Magic the Gathering -keräilykorttien kirjanpitoa varten. Sovelluksen palvelinosuus tuli toteuttaa Javan Spring Boot kehystä käyttämällä ja sovelluksen käyttöliittymä tuli toteuttaa Angular kehyksellä. Sovelluksen tietokanta tuli myös toteuttaa NoSQL-tietokantaa hyödyntämällä. Sovelluksesta piti myös tehdä Docker image eli ”näköistiedosto” sovelluksen jatkokehitystä varten.

Opinnäytetyön suunnitteluvaiheessa asetettiin selkeät tavoitteet. Käyttäjän tuli pystyä rekisteröitymään ja kirjautumaan sovellukseen. Käyttäjän tuli pystyä selaamaan keräilykorttisarjan sisältöä seuraavista näkökulmista: käyttäjä voi selata kaikkia mahdollisia kortteja ja nähdä niiden perustiedot, käyttäjä voi selata eri pelisarjan lisäosia ja niiden kortteja ja käyttäjä voi tarkemmin nähdä yksittäisten korttien tiedot. Käyttäjän tuli myös pystyä lisäämään kortteja omaan kokoelmaan tai poistamaan niitä. Sovelluksessa oli myös oltava hakutoiminto, jolla voitiin hakea ja listata kortteja eri rajauksia käyttämällä, esimerkiksi hinnan mukaan. Käyttäjän piti myös pystyä näkemään oman profiilinsa tiedot ja myös tätä kautta vaihtaa oma salasana. Sovelluksella piti myös olla mahdollisuus muuttaa käyttäjiä admin-käyttäjiksi, jotka voivat hallinnoida oman paneelin kautta sovelluksen muita käyttäjiä. Sovelluksen suunnittelussa pidettiin mielessä myös sen jatkokehitys, joka on myös tärkeä osa-alue ohjelmistokehitystä ja tähän haluttiin myös panostaa. Sovellus pyrittiin toteuttamaan siten, että sitä on helppo jatkossa kehittää eteenpäin. Jatkokehitystä pohditaan tarkemmin opinnäytetyön lopussa. Opinnäytetyössä käydään myös lyhyesti läpi sovelluksen avoimen lähdekoodin merkitystä ja sen vaikutusta sovelluksen kehittämiseen jatkossa.

Opinnäytetyössä toteutetun sovelluksen kaltaisia sovelluksia on jo olemassa, mutta niistä usein puuttuu joitain haluttuja toiminnallisuuksia tai niiden toiminnot voisi tehdä paremmin. Samankaltaisissa sovelluksissa keräilynäkökulma on usein myös kadonnut, eli niihin on lisätty toiminnallisuuksia, jotka eivät enää liity keräilyyn. Nämä mainitut ongelmat valmiissa toteutuksissa toimivat myös motivaationa tälle opinnäytetyölle.

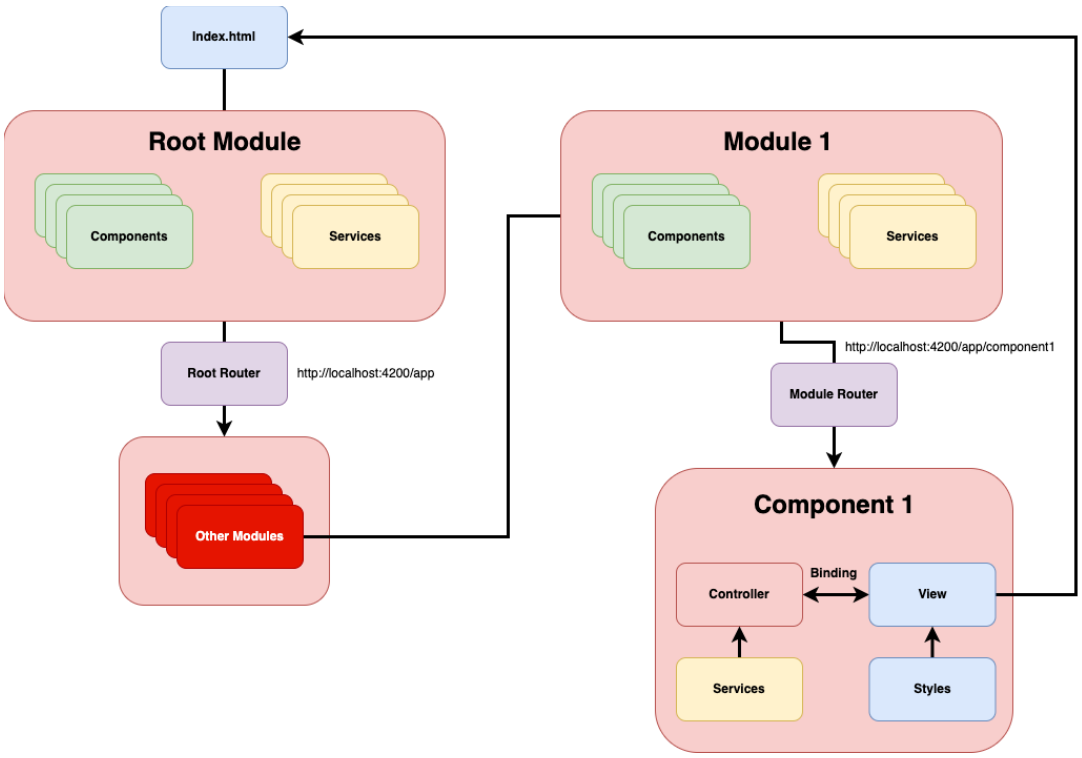
2 Teknologiapinon komponentit

2.1 Angular

Angular on JavaScript-ohjelmistokehys, mikä mahdollistaa dynaamisten yksisivuisten verkkosovellusten kehittämisen. Yksisivuinen sovellus tarkoittaa sitä, että verkkosovellus lataa vain yhden verkkosivun tai dokumentin ja päivittää sivujen sijaan sivun sisältöä eri komponenttien avulla.

Angularin arkkitehtuuri perustuu komponentteihin, jotka ovat järjestetty moduuleihin, jotka keräävät toisiinsa liittyvät komponentit ja muun koodin toiminnalliseksi kokonaisuudeksi. Angular sovellus määritellään joukoilla moduuleja, joista yhden on aina oltava päämoduuli, joka mahdollistaa sovelluksen käynnistymisen. (Angular 2022.) Alhaalla näkyvässä kuviossa (Kuvio 1) havainnollistetaan Angularin arkkitehtuuria.

Komponentit määrittelevät käyttäjälle tuodut näkymät. Komponentit koostuvat palveluista, ohjainluokasta sekä näkymään liitettävästä HTML-tiedostosta. Palvelut tarjoavat ohjainluokalle sen tarvitseman logiikan, jotta ohjainluokat pystyvät ohjelmallisesti muokkaamaan komponentin HTML-tiedostoa. Komponentteja ja palveluita voidaan käyttää myös muissa moduuleissa, mutta ne joudutaan sisällyttämään niihin moduuleihin, joissa niitä halutaan hyödyntää.



Kuvio 1. Angularin arkkitehtuuri

Angularin komponenteissa ohjainluokan data ja HTML-tiedosto yhdistetään erinäisin merkinnöin, jotta Angular voi muokata HTML-tiedostoa ennen, kun se näytetään käyttäjälle (Angular 2022). Tällä tavalla voidaan käyttäjälle esittää muuttuvaa dataa esimerkiksi, kuinka monta kertaa näytöllä näkyvää komponenttia on painettu. Tietojen sidontaa on Angularissa eri tyyppejä: Interpolaatiosidonta (eng. Interpolation Binding), ominaisuuksien sidonta (eng. property binding), luokkasidonta (eng. class binding), HTML-ominaisuuksien sidonta (eng. attribute binding) sekä tapahtumien sidonta (eng. event binding). Angular käyttää tietojen sitomisessa kahta eri tekniikkaa, jotka ovat yksisuuntainen- ja kaksisuuntainen sitominen. (Anad 2023.) Näistä sitomistyypeistä oleellisimmat ovat Interpolaatiosidonta, ominaisuuksien sidonta ja tapahtumien sidonta.

Interpolaatiosidonnalla saadaan HTML-näkymään näytettäväksi komponenttitiedoston ominaisuuksia. Tällä sitomistyylillä voidaan suorittaa erilaisia tehtäviä kuten esimerkiksi kutsua metodeja tai esittää taulukkojen sisältöjä. Interpolaatiosidonta merkitään HTML-tiedostoon kahdella peräkkäisillä aaltosulkeilla, joka on nähtävissä seuraavassa kuvassa (Kuva 1) rivillä 18. (Anad 2023.)

Ominaisuuksien sidonnalla voidaan määritellä HTML-tiedoston elementtien ominaisuuksia. Tällä tavalla voidaan sitoa komponenttien ominaisuuksia HTML-elementtien ominaisuuksiin. Tämä tarkoittaa sitä, että kun komponentin ominaisuuden arvo vaihtuu, se päivitetään myös HTML-elementtiin. Näin voidaan esimerkiksi vaihtaa HTML-elementin disabled-ominaisuuden arvoa. (Anad 2023.)

Tapahtumasidonnalla voidaan sitoa HTML_näkymässä tapahtuva tapahtuma, esimerkiksi elementin klikkaus, johonkin komponentin metodiin (Anad 2023). Tämän esimerkin mukaisesti, kun käyttäjä klikkaa jotain HTML-elementtiä, se käynnistää jonkin komponentin metodin.

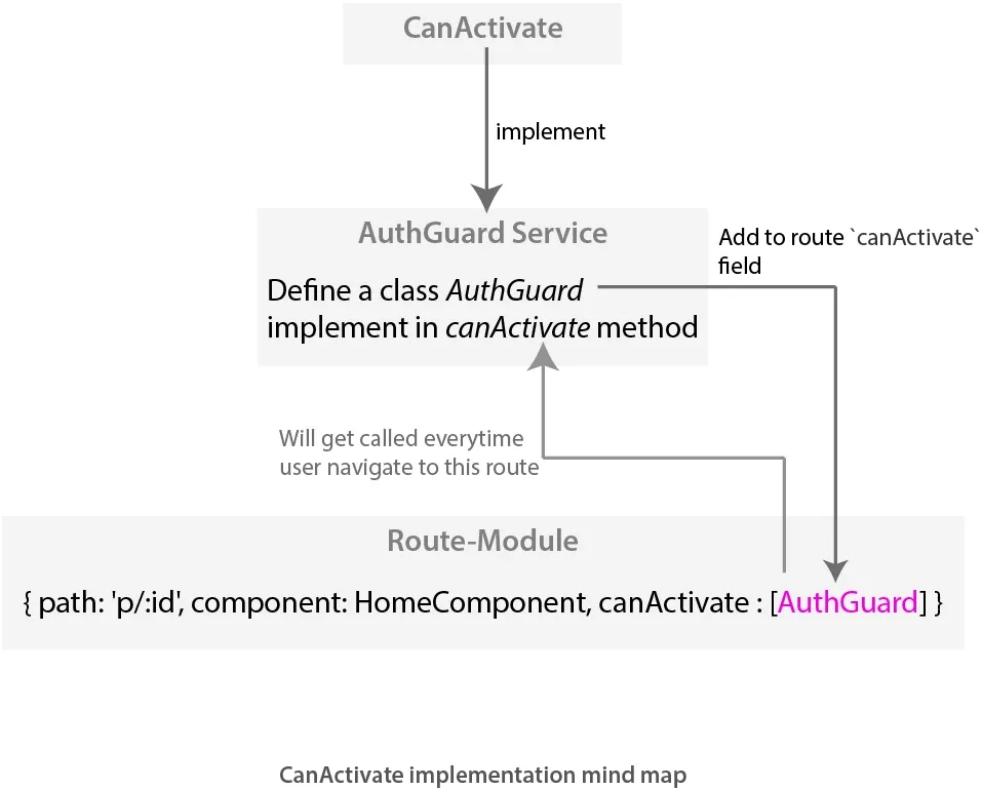
Yksisuuntainen sidonta on HTML-tiedoston ja Angular-komponentin välinen sidonta, jossa data liikkuu vain yhteen suuntaan. Data siis liikkuu vain näkymästä komponenttiin tai toisinpäin. Interpolaatiosidonta on esimerkki yksisuuntaisesta sidonnasta. (Anad 2023.)

Kaksisuuntaisessa sidonnassa data liikkuu komponentista näkymään ja takaisin. Jos esimerkiksi komponentin ominaisuudet muuttuvat, ne päivitetään suoraan HTML-tiedostoon tai jos käyttäjä esimerkiksi kirjoittaa tekstikenttään, sen arvo voidaan suoraan tallentaa komponentin muuttujaan. Kaksisuuntaista sidontaa käytetään yleisesti esimerkiksi lomakkeissa. (Anad 2023.) Seuraavassa kuvassa (Kuva 1) on nähtävissä esimerkki kaksisuuntaisen sidonnan syntaksista rivillä 2.

```
<div class="filter-container">
  <form [formGroup]="parentForm">
    <mat-button-toggle-group class="toggle-group" name="type" aria-label="Set Type" (change)="onSubmit($event.value)"
      <mat-button-toggle class="toggle" value="all">All</mat-button-toggle>
      <mat-button-toggle class="toggle" value="core">Core</mat-button-toggle>
      <mat-button-toggle class="toggle" value="expansion">Expansion</mat-button-toggle>
      <mat-button-toggle class="toggle" value="draft_innovation">Draft Innovation</mat-button-toggle>
      <mat-button-toggle class="toggle" value="commander">Commander</mat-button-toggle>
      <mat-button-toggle class="toggle" value="funny">Funny</mat-button-toggle>
      <mat-button-toggle class="toggle" value="masters">Masters</mat-button-toggle>
    </mat-button-toggle-group>
  </form>
</div>

<div class="sets-container" *ngIf="submittedValue == 'all' || submittedValue == ''">
  <ng-container *ngFor="let set of sets">
    <mat-card *ngIf="!set.parent_set_code" class="set-card" [routerLink]="['/dashboard/sets/', set.code]">
      <mat-toolbar color="accent"><p class="name">{{set.name}}</p></mat-toolbar>
      <img src="{{set.icon_svg_uri}}" alt="">
      <mat-card color="accent" class="info">
        <mat-progress-bar mode="determinate" value="{{getProgressValue(set.code)}}" class="progress-bar" color="accent"
        <h2 style="text-transform: capitalize;">{{set.set_type}}</h2>
        <h2>Cards: {{set.card_count}}</h2>
      </mat-card>
    </mat-card>      Rami Riekkinen, last month • set type filter
  </ng-container>
</div>
```

Kuva 1. Esimerkkejä Angular sidonnasta

Angularin Router (suom. reititin) moduuli tarjoaa kehittäjälle palvelun, jolla voidaan määritellä eri polkuja eri näkymämalleille. Reititin ohjaa polut näkymiin sen sijaan, että se lataisi uuden sivun. (Angular 2022.) Reitittimen polkuihin voidaan myös asettaa eri sääntöjä esimerkiksi vartijaluokkia, jotka päästävät käyttäjän näkemään polun sisällön vain, kun tietyt ehdot täyttyvät esimerkiksi vain kirjautunut käyttäjä voi nähdä sisällön. Seuraavassa kuviossa (Kuvio 2) näkyy Angularin Guard-luokan toimintaperiaate. Guard-luokan toiminta vaati CanActivate-luokan metodin canActivate implementoinnin.



Kuvio 2. Angular Guard-luokan implementointi (Sharma 2019)

Jotta käyttäjä voi edetä eri näkymiin, on aiemman kuvion canActivate-metodin palautettava arvo tosi. Tällä tavoin voidaan itse määritellä sääntöjä, missä tilanteissa käyttäjät päästetään mihinkin näkymään. Esimerkiksi seuraavassa kuvassa (Kuva 2) on asetettu sääntö, mikä päästää käyttäjän etenemään vain, jos käyttäjällä on JWT tallennettuna selaimen paikalliseen varastoon.

```
canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
    const token = localStorage.getItem('token')

    if(!token){
      this.router.navigate(['/'])
      return false
    } else {
      return true
    }
  }
}
```

Kuva 2. Guard-luokan canActivate-metodi

2.2 Spring Boot

Java Spring kehys (Spring Framework) on avoimen lähdekoodin ohjelmistokehys, jolla luodaan itsenäisiä tuotantotason sovelluksia. Spring Bootilla on monta eri määritelmää, mutta IBM:n määritelmän mukaan se on työkalulaajennus Java Spring kehykseen. Spring Boot mahdollistaa nopeamman ja helpomman websovellusten ja mikropalveluiden kehittämisen. (IBM 2023b.) Spring Frameworkin alustus on usein aikaa vievä prosessi,

mutta Spring Boot alustetaan minimaalisella konfiguraatiolla, eli se on heti valmis ajettavaksi (Tutorialspoint 2022).

Vaikka Spring Framework yksinään on hyvä ja kattava, se vaati paljon aikaa, jotta sen saa konfiguroitua. Spring Boot lieventää tätä kolmella tärkeällä ominaisuudella: autokonfiguraatiolla, mielipiteisellä lähestymistavalla (eng. opinionated approach) ja mahdollisuudella luoda itsenäisiä sovelluksia.

Automaattinen määritys tarkoittaa sitä, että sovellukset alustetaan ennalta määritellyillä riippuvuuksilla ja mahdollistaa siten sovellusten nopean kehityksen. Spring Boot määrittää automaattisesti sekä Spring Frameworkin, että kolmannen osapuolen paketit käyttäjän asetusten perusteella.

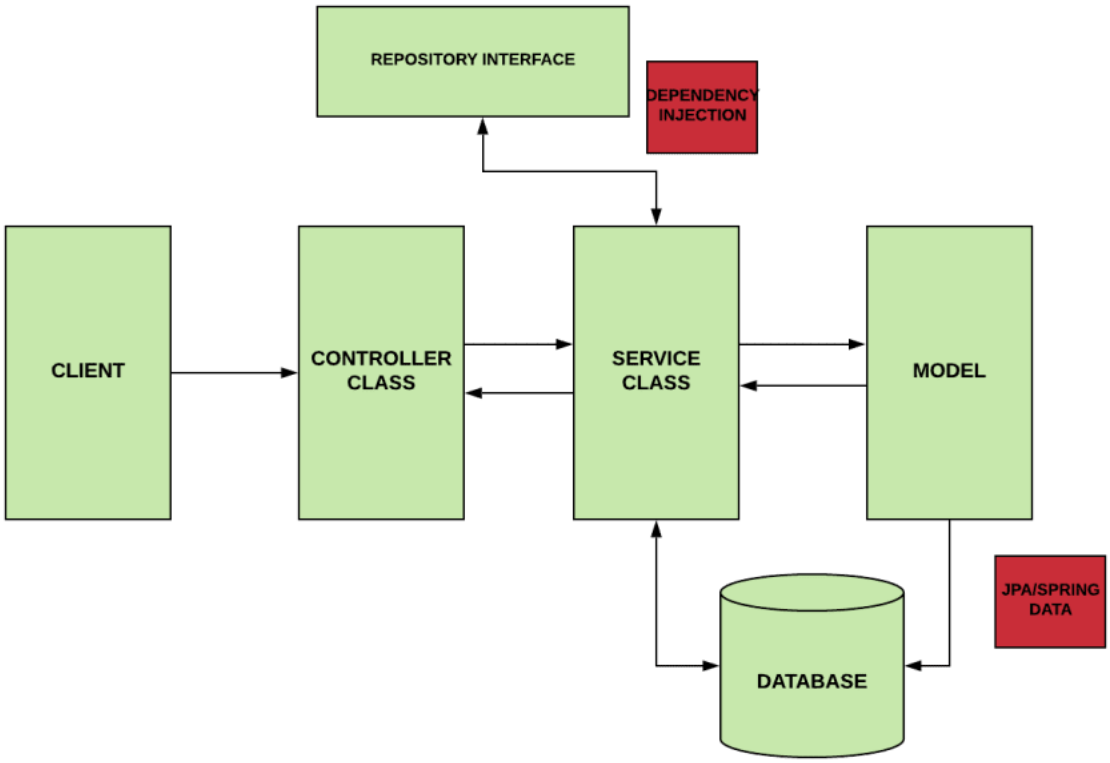
Spring Boot lisää ja määrittää projektin aloitusriippuvuudet projektin tarpeiden mukaan. Spring Boot valitsee, mitkä paketit asennetaan ja mitä arvoja käytetään. (IBM 2023b.) Projektin tarpeet määritellään projektin aloitusvaiheessa Spring Initializr työkalua käyttämällä, jossa voidaan määritellä projektin alkuasetukset, esimerkiksi tehdäänkö projekti Mavenilla vai Gradlella tai mitä Java versiota käytetään. Tämän jälkeen valitaan alustavat riippuvuudet. Näitä riippuvuuksia on tarjolla erityyppisiä ja eri tarkoitukseen sopivia. Esimerkiksi Lombok tai Spring Web.

Spring Boot mahdollistaa sovellusten luomisen, jotka toimivat itsenäisesti ilman ulkoista verkkopalvelinta upottamalla siihen verkkopalvelimen alustusvaiheessa, kuten Tomcat tai Netty. Tämä tarkoittaa sitä, että sovelluksen voi käynnistää millä tahansa alustalla. (IBM 2023b.)

Spring Bootin arkkitehtuuri koostuu neljästä tasosta: Presentation-, Business-, Persistence- ja Database-tasosta. Presentation-taso on Spring Boot-arkkitehtuurin ylin taso ja se esitetään usein Controller-luokkana. Tällä tasolla toteutetaan http-pyyntöjen käsittely ja autentikointi. Tällä tasolla käsitellään kaikki käyttöliittymältä saadut kutsut, jotka voivat olla tyyppiä: GET, POST, PUT, DELETE, PATCH. Business-tasolla käsitellään sovelluksen tarvitsema logiikka ja se tapahtuu usein Service-luokkaa käyttämällä. Persistence-taso hakee ja vie dataa tietokannasta palvelimelle. Tämä taso tehdään Repository-luokalla, joka vastaa kaikista tietokantakyselyistä. Database-taso käsittelee kaikki tietokantaoperaatiot. Database-tasoa voidaan ajatella itse tietokantana. (Dev Community 2022.)

Edellä esitettyjen tasojen perusteella Spring Boot:n työnkulkua voidaan kuvata seuraavanlaisesti: Käyttöliittymästä saapuu HTTP-pyyntö Controller-luokalle, joka tarkastaa pyynnön tyypin ja autentikoi sen. Tämän jälkeen pyyntö siirretään Service-

luokalle, joka tekee tarvittavat tietokantakyselyt ja logiikan ja lopuksi palauttaa käyttöliittymälle vastauksen. (Dev Community 2022.) Seuraavassa kuviossa (Kuvio 3) selvennetään Spring Bootin työnkulkua. Kuviosta voidaan nähdä, miten tässä kappaleessa selitetty työnkulku etenee.

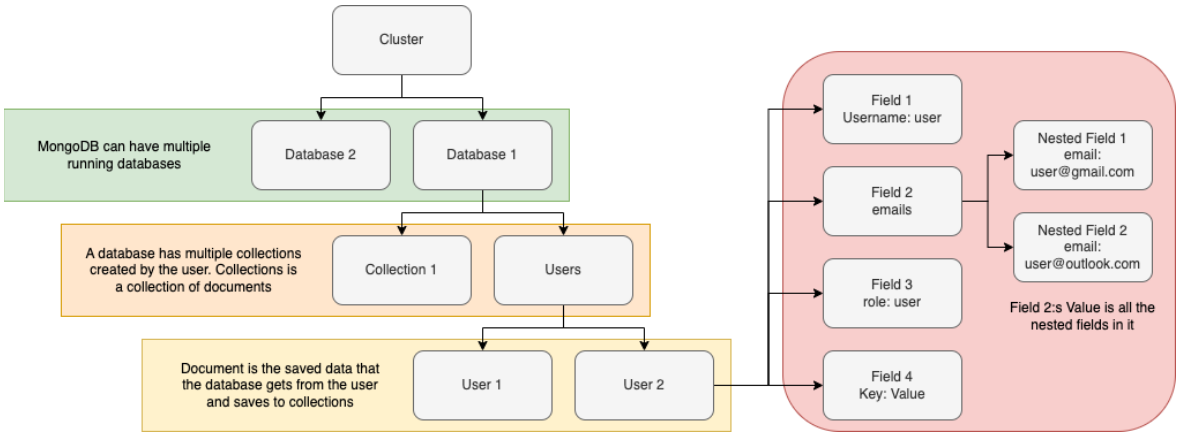


Kuvio 3. Spring Bootin työnkulku (Dev Community 2022)

2.3 MongoDB

MongoDB on avoimen lähdekoodin dokumenttipohjainen NoSQL-tietokanta, jossa tietokanta koostuu SQL-tietokantojen tauluista, sarakkeista ja riveistä poiketen JSON tyylisistä dokumenteista ja kokoelmista. MongoDB dokumentit koostuvat kentistä, jotka ovat avainarvo-pareja, jotka tukevat kaikkia BSON data tyyppejä. (GeeksforGeeks 2021.) MongoDB tukee myös useita ohjelmointikieliä ja -kehyksiä. Näihin kuuluvat: C, C++, C#, Java, Node.js, Perl, PHP, Python ja Ruby. (Saha 2022.)

MongoDB dokumentit tukevat niin sanottua sisäkkäistä datamallia (eng. nested model), jossa dataa voidaan tallentaa kenttiin hierarkkisessa muodossa, mikä tarkoittaa sitä, että kentän sisälle voidaan luoda uusia kenttiä. (Gillis & Botelho 2023.) Alhaalla kuvio (Kuvio 4) MongoDB:n tietokantarakenteesta, josta selkenee miten sisäkkäiset kentät voivat rakentua sisäkkäisistä kentistä. Esimerkiksi kuvion 4 emails-kenttä koostuu kahdesta sähköpostiosoitteesta.



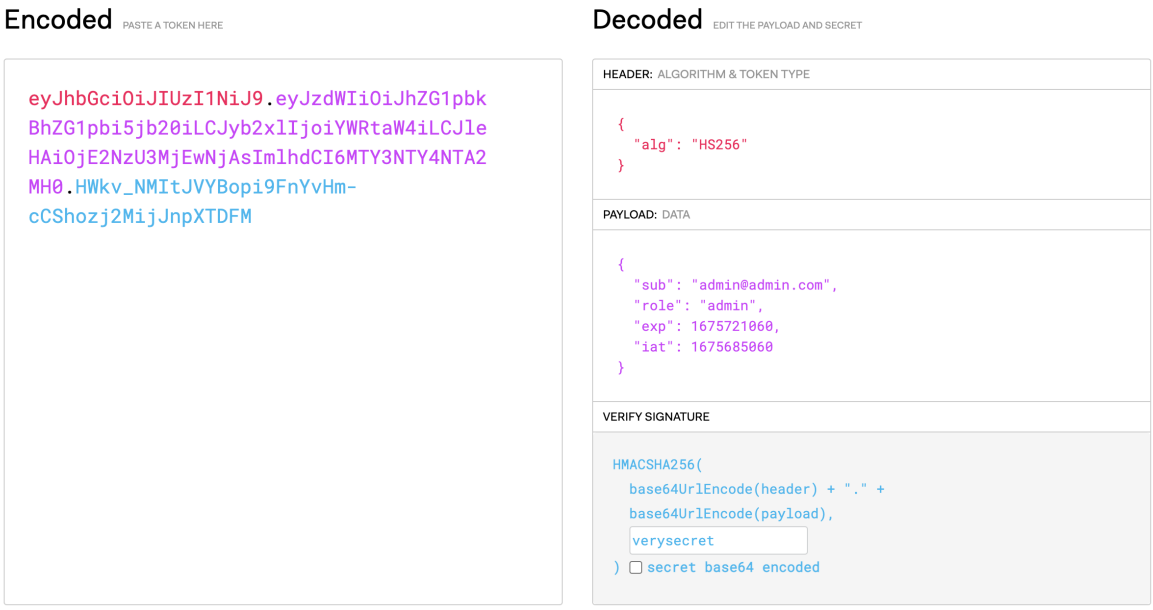
Kuvio 4. MongoDB:n tietokantarakenne

MongoDB sopii hyvin projekteihin, jotka käsittelevät paljon dataa esimerkiksi, jos halutaan lisätä tietokantaan tuhansia tietueita sekunnissa. MongoDB:n avulla tietokantarakennetta voidaan myös helposti muuttaa, koska sille ei määritetä etukäteen rakennetta. Näiden ominaisuuksien ansiosta MongoDB sopii hyvin esimerkiksi verkkokauppa-tyyppisiin tuotepohjaisiin sovelluksiin tai sovelluksiin, joiden rakenne voi muuttua milloin tahansa. (Saha 2022.)

2.4 JSON Web Token

JSON Web Token eli JWT-menetelmä on avoin standardi, joka määrittelee kompaktin tavan siirtää tietoa turvallisesti eri osapuolten välillä JSON-objektina. JWT:n voi allekirjoittaa erinäisillä algoritmeilla. (JWT 2023.)

JWT koostuu kolmesta osasta: otsikko, hyötykuorma ja allekirjoitus. Seuraavassa kuvassa (Kuva 3) voidaan nähdä esimerkki JWT:n muodosta kuvan vasemmalla puolella. Otsikko koostuu kahdesta osasta ja sisältää tiedon tokenin tyypistä ja sen käyttämästä salauksesta (kuvassa oikealla puolella Header-otsikon alla). Hyötykuorma on sisältöosuus ja se koostuu kehittäjän asettamista avainarvopareista, jotka voivat olla esimerkiksi käyttäjänimi tai sähköpostiosoite (Kuvassa Payload-otsikon alla). Allekirjoitusosuus (Kuvassa Verify Signature-otsikon alla) sisältää otsikon ja hyötykuorman koodatun muodon sekä salaisuuden yhdistelmän. (JWT 2023.)



Kuva 3. JWT:n rakenne

2.5 Docker

Docker on avoimen lähdekoodin alusta, jolla kehittäjät voivat rakentaa, ottaa käyttöön, suorittaa ja hallita kontteja (eng. containers). Nämä säilöt ovat standardoituja ja suoritettavia komponentteja, jotka yhdistävät sovelluksen lähdekoodin käyttöjärjestelmän kirjastoihin ja riippuvuuksiin, joita tarvitaan lähdekoodin suorittamiseen. (IBM 2023a.) Tämä mahdollistaa esimerkiksi Windows-käyttöjärjestelmälle rakennetun sovelluksen ajamisen Linux-käyttöjärjestelmässä.

Docker kontit ovat löyhästi eristettyjä ja suojattuja ympäristöjä. Useita kontteja voidaan ajaa samanaikaisesti. Kontit ovat kevyitä ja sisältävät kaiken sovelluksen suorittamiseen tarvitun sisällön, jonka takia käyttäjän ei tarvitse rajoittaa kehitystä tai ajoa oman ympäristön rajoihin. Kontteja voidaan myös helposti jakaa, jolloin kaikki saavat saman kontin, joka toimii samalla tavalla kaikkien ympäristöissä. (Docker Inc 2023.)

Docker käyttää asiakas-palvelin-arkkitehtuuria. Docker client (suom. asiakas) keskustelee Docker-daemonin kanssa, joka huolehtii konttien rakentamisesta, käyttämisestä ja jakelusta. Docker-asiakas ja -daemon kommunikoivat REST API:n avulla verkon välityksellä. Docker Compose on toinen Docker asiakas, joka mahdollistaa työskentelyn sovellusten kanssa, jotka koostuvat useammista konteista. (Docker Inc 2023.)

Docker Image on read-only-malli, joka sisältää ohjeet Docker-kontin luomiseen. Image pohjautuu valmiiseen imagepohjaan, jota muokataan omien tarpeiden mukaan (Docker

Inc 2023). Voidaan esimerkiksi rakentaa image, joka perustuu node 19 versioon. Tämä tarkoittaa, että konttia luodessa siinä on jo valmiina noden versio 19.

3 Suunnittelu

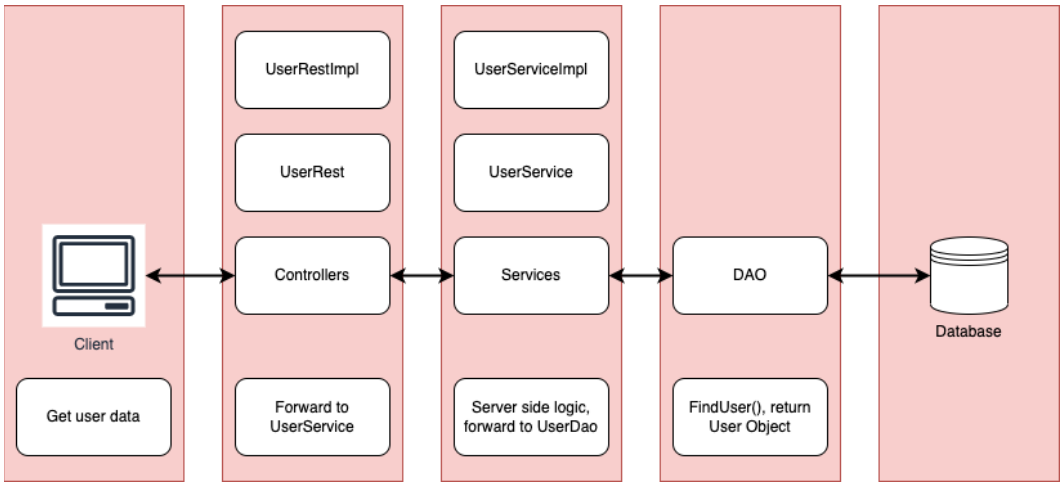
3.1 Tietokanta

Tietokantaan tallennettavan datan muoto oli etukäteen tiedossa. Data saatiin ulkoisesta API:sta JSON-muodossa, joten tietokannaksi valittiin MongoDB. MongoDB mahdollisti korttien tallentamisen tietokantaan lähes samassa muodossa, miten ne saatiinkin. Koska MongoDB ei tarvitse skeemaa toisin kuin MySQL, myös tietokantaan tarvittu työmäärä väheni. Spring Bootissa käytetty Data-paketti mahdollisti myös helpon ja nopean dokumenttien tallennuksen, hakemisen ja muokkaamisen.

Tietokannan suunnitellussa haluttiin etukäteen selvittää minkälaisia MongoDB-kokoelmia sovellus tarvitsisi ja miten ne mahdollisesti linkittyisivät toisiinsa. Asetettujen tavoitteiden perusteella nämä olivat kuitenkin helppo määritellä. Sovelluksiin haluttiin kokoelmat kaikille pelisarjan korteille (Cards), käyttäjien kokoelmille (Collections), käyttäjille (Users) ja lisäosille (Sets). Näistä mainituista kokoelmista Cards ja Collections ovat erikoisimmat. Cards-kokoelmaan päivitetään jatkuvasti uusien lisäosien varalta ja siihen tuodaan aina uusimmat kortit. Collections-kokoelma sisältää kenttinä käyttäjätiedot sekä kaikki pelisarjan kortit typistetyssä muodossa. Collections-kokoelman kortit-kenttään haluttiin korttien minimaaliset tiedot, jotka olivat kortin id, -nimi, -lisäosan koodi ja tieto siitä onko kortti kerätty.

3.2 Palvelinpuolen API

Palvelimen toiminta on kuvattu alhaalla olevassa kuviossa (Kuvio 5). Käyttäjältä lähetetty pyyntö ohjataan käyttöliittymän puolella oikeaan API:n controller-luokkaan, joka ohjaa pyynnön edelleen oikeaan service-luokkaan, jossa toteutetaan pyynnön vaatima logiikka. Service luokka parsii logiikan avulla käyttäjän pyynnöstä kaiken datan, jota tietokannan repository-luokka (DAO) tarvitsee ja lähettää sen eteenpäin. DAO-luokka käyttää saatua dataa hakeakseen tietokannasta kaikki dataa vastaavat dokumentit, jotka palautetaan sitten käyttöliittymän käyttöön, joka visualisoi datan käyttäjälle.



Kuvio 5. Käyttäjältä saadun pyynnön elinkaari palvelimessa

3.3 Käyttöliittymä

Sovelluksen käyttöliittymä suunniteltiin siten, että käyttäjä näkee aluksi vain kotinäkymän eikä pääse käsiksi muihin näkymiin ennen kuin käyttäjä rekisteröityy ja kirjautuu sovellukseen. Kirjautumisen jälkeen käyttäjälle avautuu dashboard-tyylinen näkymä, josta käyttäjä voi navigoida eri näkymiin, joissa voidaan tarkastella eri tietoja.

Käyttöliittymässä haluttiin hyödyntää suurimmaksi osaksi Angular Material-kirjastoa, josta saatiin valmiiksi tyyliteltyjä komponentteja. Näitä komponentteja voitiin käyttää kaikissa sovelluksen omissa komponenteissa (esim. hakutuloksissa).

4 Palvelimen ja API:n toteutus

4.1 Spring Data ja luokat

Spring Data on Spring kehyksen työkalupaketti, jolla voidaan tallentaa, muokata ja hakea dataa tietokannasta. Pakettiin löytyy tarkenne eri tietokannoille kuten esimerkiksi MongoDB- tai MySQL-tietokannoille. Jotta pakettia voidaan käyttää, tarvitaan rajapintaluokka, joka laajentaa MongoRepository-rajapintaa. Tälle luokalle määritetään tietotyyppi, jonka mukaisia dokumentteja luokka käyttää, esimerkiksi ”CardWrapper”, jota kuvan (Kuva 4) esimerkissä on käytetty. Sovelluksessa näitä Repository-luokkia tehtiin käyttäjille, korteille, lisäosille ja kokoelmille.

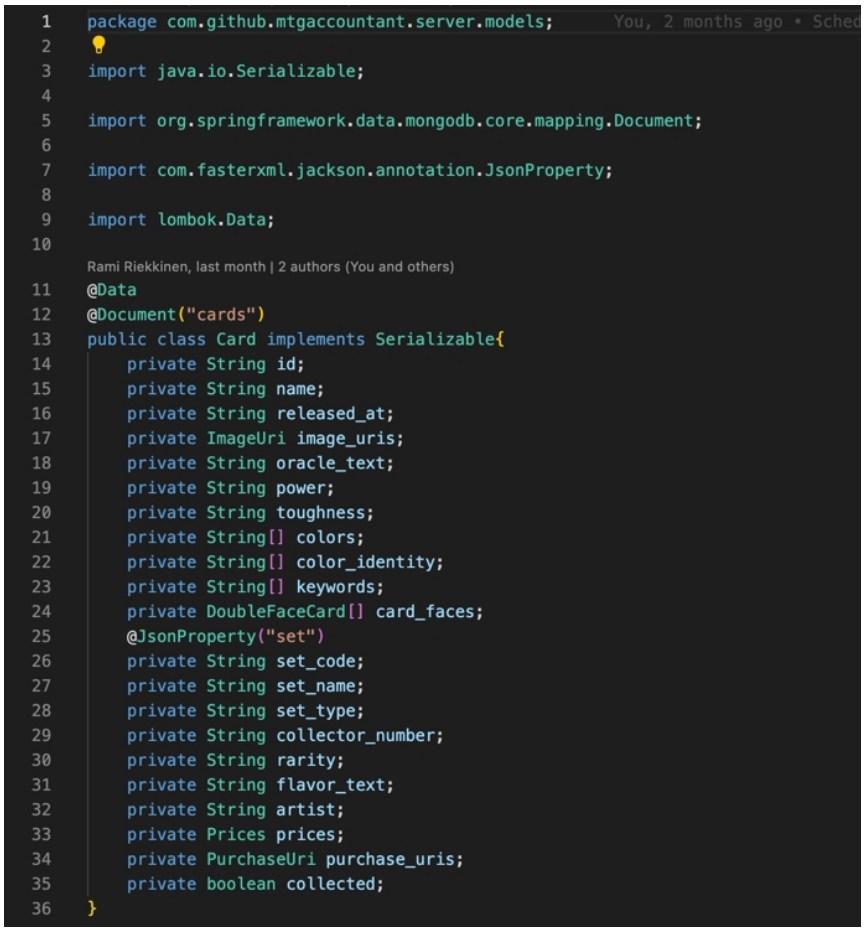
Repository-luokille on valmiita tietokannalle tehtäviä kyselyitä, joita voidaan käyttää ilman, että tätä luokkaa tarvitsee muuttaa. Query-annotaatiolla voidaan määritellä mukautettuja hakuja kuten esimerkiksi kuvan findAllCollectionCards(), joka hakee kaikki kortit, mutta palauttaa niistä vain kentät: id, name, set ja collected.

```
1   package com.github.mtgaccountant.server.dao;
2
3   import java.util.List;
4
5   import org.springframework.data.mongodb.repository.MongoRepository;
6   import org.springframework.data.mongodb.repository.Query;
7
8   import com.github.mtgaccountant.server.wrapper.CardSearchWrapper;
9   import com.github.mtgaccountant.server.wrapper.CardWrapper;
10  import com.github.mtgaccountant.server.wrapper.CollectionCardWrapper;
11
    Rami Riekkinen, last month | 2 authors (You and others)
12  public interface CardDao extends MongoRepository<CardWrapper, Integer>{
13      @Query(value="{}", fields="{'id' : 1, 'name' : 1, 'set' : 1, 'collected' : 1}")
14      List<CollectionCardWrapper> findAllCollectionCards();
15
16      @Query(value="{}")
17      List<CardWrapper> findAlCardWrappers();
18
19      @Query(value="{}")
20      List<CardSearchWrapper> findAllCardSearchWrappers();
21
22      @Query("{_id:'?0'}")
23      CardWrapper findCardById(String id);
24
25      @Query("{set:'?0'}")
26      List<CardWrapper> findSetCards(String code);
27  }
```

Kuva 4. Spring Datan Repository-luokka

Spring Data pakettiin kuuluu annotaatio ”@Document” joka sitoo Java-luokan MongoDB:n kokoelmaan. Tällä annotaatiolla voidaan ennalta määrittää tietokannan dokumentin kentät. (VMware Inc 2023.) Näin voidaan valmiiksi määritellä, minkä nimisiä kenttiä luodaan ja minkälaisia tietotyyppejä ne sisältävät. Seuraavassa kuvassa (Kuva 5) on

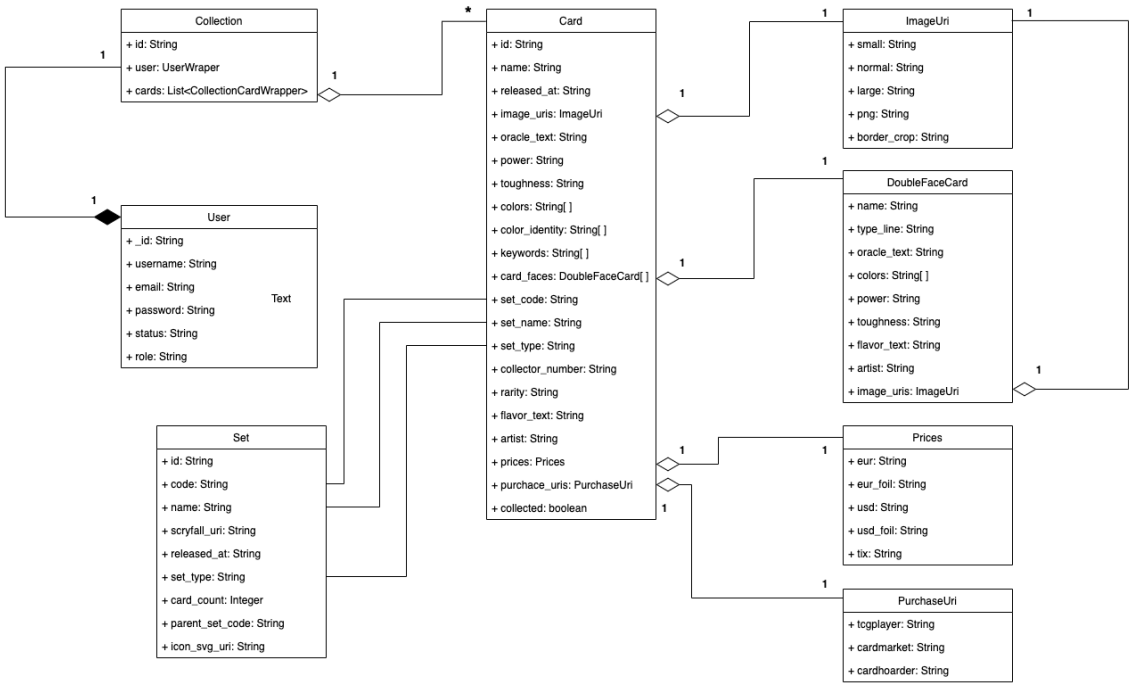
esimerkkinä Java-luokka, jossa käytetään Document-annotaatiota. Kuvasta voidaan myös huomata, että dokumenttiin sisällytetään dataa, joka on tietotyypiltään objekti. Esimerkiksi kenttä prices on tietotyypiltään Prices olio.



```java
package com.github.mtgaccountant.server.models;      You, 2 months ago • Sched

import java.io.Serializable;

import org.springframework.data.mongodb.core.mapping.Document;

import com.fasterxml.jackson.annotation.JsonProperty;

import lombok.Data;

Rami Riekkinen, last month | 2 authors (You and others)
@Data
@Document("cards")
public class Card implements Serializable{
    private String id;
    private String name;
    private String released_at;
    private ImageUri image_uris;
    private String oracle_text;
    private String power;
    private String toughness;
    private String[] colors;
    private String[] color_identity;
    private String[] keywords;
    private DoubleFaceCard[] card_faces;
    @JsonProperty("set")
    private String set_code;
    private String set_name;
    private String set_type;
    private String collector_number;
    private String rarity;
    private String flavor_text;
    private String artist;
    private Prices prices;
    private PurchaseUri purchase_uris;
    private boolean collected;
}
```

Kuva 5. Luokka, joka käyttää @Document- ja @Data-annotaatioita

Kuviossa 6 (Kuvio 6) selvennetään luokkien välisiä suhteita luokkakaavion avulla. Kuviosta voidaan nähdä esimerkiksi, että käyttäjällä voi olla vain yksi kokoelma, johon kuuluu useita kortteja. Kuviossa myös havainnollistetaan miten lisäosat ja niiden kortit linkitetään toisiinsa. Kortit linkittyvät lisäosiin lisäosien koodin, nimen ja tyypin perusteella.

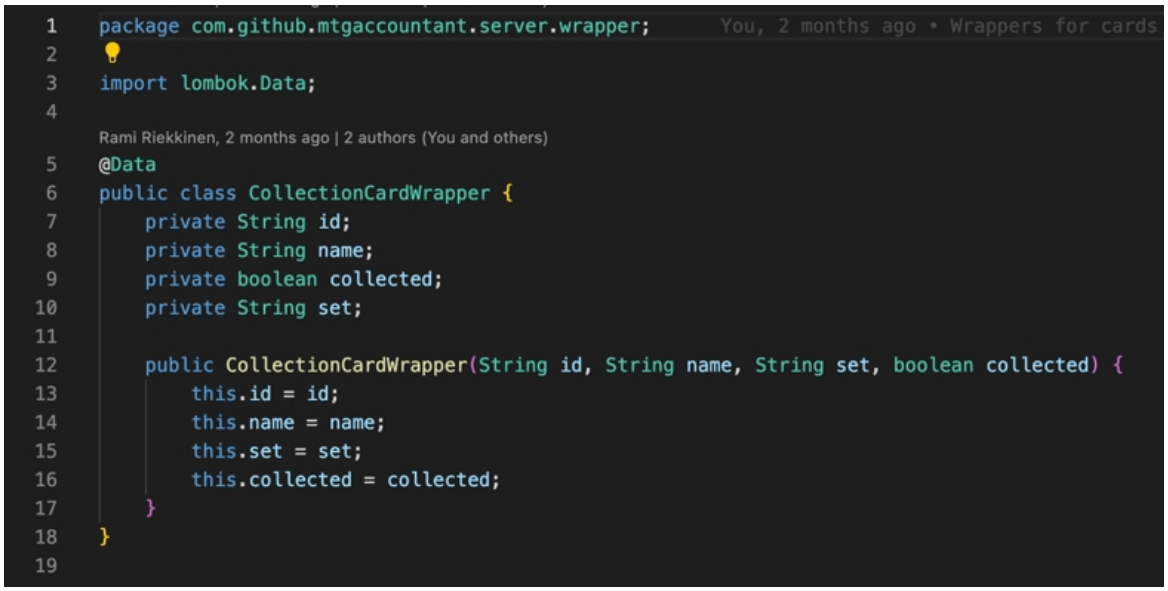


Kuvio 6. Palvelinsovelluksen luokkakaavio

4.1.1 Kääreluokat

Kääreluokat (eng. Wrapper Class) ovat perusluokkien typistettyjä versioita, jotka ovat rakenteeltaan muuten saman muotoisia, mutta niistä on usein jätetty joitain kenttiä pois. Nämä luokat ovat hyviä silloin kun liikutellaan esimerkiksi objekteja, jotka normaalisti sisältäisivät arkaluonteista dataa kuten esimerkiksi salasanoja tai objekteja, joissa on paljon ylimääräistä dataa, jota ei sen elinkaaren aikana käytetä ollenkaan. Aiempaa kuvaa (Kuva 5) voidaan verrata seuraavaan kuvaan (Kuva 6), joka on minimaalinen versio edellä olevasta luokasta.

Opinnäytetyössä Wrapper-luokkia hyödynnettiin esimerkiksi luokassa Collections, jossa dokumenttiin haluttiin sisällyttää User- ja Card-luokkien tietoja. User-luokasta haluttiin tietoturvasyistä jättää pois salasana kenttä ja Card-luokasta haluttiin vain kortin minimaalinen versio.



```
1  package com.github.mtgaccountant.server.wrapper;      You, 2 months ago • Wrappers for cards
2
3  import lombok.Data;
4
   Rami Riekkinen, 2 months ago | 2 authors (You and others)
5  @Data
6  public class CollectionCardWrapper {
7      private String id;
8      private String name;
9      private boolean collected;
10     private String set;
11
12     public CollectionCardWrapper(String id, String name, String set, boolean collected) {
13         this.id = id;
14         this.name = name;
15         this.set = set;
16         this.collected = collected;
17     }
18 }
19
```

Kuva 6. Wrapper-luokka

4.1.2 Lombok

Aiemmista kuvista, esimerkiksi kuvasta (Kuva 5) voidaan nähdä @Data-annotaatio, joka liittyy Lombok-kirjastoon. Tämä annotaatio mahdollistaa perinteisten asetusfunktioiden (getter ja setter) jättämisen pois. Annotaatiota käyttämällä nämä funktiot tehdään automaattisesti eikä niitä siten tarvitse kirjoittaa erikseen.

Lombok-kirjasto itsessään on annotaatiohin perustuva Java-kirjasto, jonka avulla voidaan vähentää koodin määrää. Lombokin annotaatiot ovat tarkoitettu korvaamaan Java-koodia, joka on osittain toistuvaa. Lombokin avulla voidaan esimerkiksi korvata konstruktoreita, getter-, setter- ja toString() metodeita vain muutamilla annotaatioilla. (Zanini 2021.)

4.2 Käyttäjät

Palvelimen API:lle luotiin päätepiste käyttäjille polkuun ”api/user”, josta johdettiin käyttäjän tietojen hakemiseen, luomiseen ja muokkaamiseen tarvittavat toiminnallisuudet. Näitä olivat esimerkiksi käyttäjän rekisteröinti, kirjautuminen, tietojen muokkaaminen tai JWT-tokenin tarkastaminen mistä puhutaan tarkemmin myöhemmässä luvussa.

Sovellukseen haluttiin kahdentyyppisiä käyttäjiä: peruskäyttäjä, joka käyttää sovelluksen toimintoja normaalisti sekä järjestelmänvalvoja eli admin-käyttäjä, joka sovelluksen toimintojen käytön lisäksi hallinnoi muita sovelluksen käyttäjiä. Nämä eri käyttäjätyypit eriteltiin tietokannassa rooli kentällä.

4.2.1 Rekisteröinti

Käyttäjien rekisteröinti tapahtuu palvelimen puolella siten, että käyttäjä täyttää rekisteröinti lomakkeen käyttöliittymässä ja lähettää sen palvelimen käsiteltäväksi. Palvelin tarkastaa lomakkeen tiedot ja luo uuden käyttäjän ja käyttäjäkohtaisen korttikokoelman, jos lomakkeella annetulla sähköpostiosoitteella ei löydy jo olemassa olevaa käyttäjää.

Jos käyttäjää ei ennestään jo löydy tietokannasta, palvelin tallentaa käyttäjän Users-kokoelmaan tietokantaan. Käyttäjää tallennettaessa salasana muutetaan hajautettuun muotoon, jotta se voidaan turvallisesti säilöä tietokantaan. Tämän jälkeen alustetaan käyttäjäkohtainen kokoelma, johon haetaan kortit Cards-kokoelmasta. Kokoelma olioon sitten lisätään haetut kortit ja luotu käyttäjä ja tallennetaan Collections-kokoelmaan tietokantaan. Lopuksi käyttäjälle palautetaan viesti pyynnön lopullisesta tilasta ja sen http-tilakoodista, joka voi olla yksi seuraavista: 200 - OK, 400 – BAD REQUEST tai 500 - INTERNAL SERVER ERROR. Alhaalla kuva rekisteröinti metodista (Kuva 7).

```
66      @Override
67      public ResponseEntity<String> signUp(Map<String, String> requestMap) {
68          try{
69              if(validateSignUpMap(requestMap)){
70                  User user = userDao.findUserByEmail(requestMap.get(key: "email"));
71                  Collection collection = new Collection();
72
73                  if(Objects.isNull(user)){
74                      userDao.save(getUserFromMap(requestMap));   // Save user document to database
75
76                      UserWrapper collectionUser = userDao.findUser(requestMap.get(key: "email"));
77
78                      collection.setCards(cardDao.findAllCollectionCards());
79                      collection.setUser(collectionUser);
80                      collection.setFinderID(collectionUser.getUsername() + collectionUser.getEmail());
81
82                      collectionDao.save(collection);     // Save user specific collection to database
83
84                      return MtgAccountantUtils.getResponseEntity(responseMessage: "Succesfully registered.", HttpStatus.OK);
85                  } else {
86                      return MtgAccountantUtils.getResponseEntity(responseMessage: "Email already exists.", HttpStatus.BAD_REQUEST);
87                  }
88              } else {
89                  return MtgAccountantUtils.getResponseEntity(MtgAccountantConstants.INVALID_DATA, HttpStatus.BAD_REQUEST);
90              }
91          } catch (Exception e){
92              e.printStackTrace();
93          }
94          return MtgAccountantUtils.getResponseEntity(MtgAccountantConstants.SOMETHING_WENT_WRONG, HttpStatus.INTERNAL_SERVER_ERROR);
95      }
```

Kuva 7. Käyttäjän rekisteröinti palvelimessa

4.2.2 Kirjautuminen

Sovellukseen kirjaudutaan sähköpostiosoitteen ja salasanan yhdistelmällä. Tiedot lähetetään palvelimelle, joka tarkastaa vastaako yhdistelmä mihinkään tietokannassa olevaan käyttäjään. Jos käyttäjän antamat tiedot ovat oikein, palautetaan käyttöliittymälle palvelimen luoma JWT-token. JWT-tokeniin sisällytetään käyttäjän sähköpostiosoite sekä rooli.

Kirjautumistilanteessa palvelin ensimmäisenä autentikoi käyttäjän. Jos autentikointi epäonnistuu, palautetaan käyttäjälle viesti, jossa kerrotaan salasanan tai sähköpostin olevan väärä. Seuraavaksi palvelin tarkastaa käyttäjän statuskentän, joka on tietokannasta saatava boolean-arvo. Jos tämä arvo on epätosi, palvelin lähettää vastauksen käyttöliittymälle viestin kanssa, jossa kerrotaan käyttäjän olevan tilapäisesti estetty kirjautumaan sovellukseen. Jos status arvo on tosi, palvelin luo käyttäjälle JWT-Tokenin ja palauttaa sen käyttöliittymälle.

4.2.3 Admin käyttäjät

Admin-käyttäjille tehtiin kaksi omaa päätepistettä, joita hyödynnetään admin-käyttäjien omassa hallintapaneelissa. Nämä olivat kaikkien käyttäjien tietojen hakeminen ja niiden muokkaaminen. Tällä hetkellä sovelluksessa ei ole mahdollista muuttaa peruskäyttäjää admin-käyttäjäksi, vaan muutos tehdään manuaalisesti tietokannan puolella.

Kaikkien käyttäjien tietojen hakeminen tapahtuu getAllUsers-metodilla, joka ensin tarkastaa käyttäjän roolin. Jos rooli on muotoa ”admin”, haetaan tietokannasta kaikkien käyttäjien tiedot ja palautetaan ne käyttäjälle http-tilakoodin 200-OK kanssa. Jos käyttäjän rooli on väärä, palautetaan käyttäjälle tyhjä käyttäjälista http-tilakoodin 401-UNAUTHORIZED kanssa. Muussa tapauksessa käyttäjälle palautetaan tilakoodi 500 – INTERNAL SERVER ERROR.

Admin-käyttäjä voi myös muokata muiden käyttäjien tilaa. Käyttäjien tila on tieto siitä pystyvätkö he kirjautumaan sovellukseen sisään, vaikka salasana ja sähköpostiosoite olisikin oikea. Palvelimelle lähetetään pyynnön mukana sähköpostiosoite, joka viittaa muutettavaan käyttäjään ja uusi tila, joka on muotoa tosi tai epätosi. Palvelin sitten hakee käyttäjän tiedot, jotka vastaavat annettua sähköpostiosoitetta ja muuttaa käyttäjän tilan pyynnön mukana annettuun tilaan. Uusi muokattu käyttäjä sitten tallennetaan tietokantaan vanhan käyttäjän tilalle.

4.3 Autentikointi

Sovelluksen autentikointi tapahtuu erinäisten suodattimien ja konfiguraatioiden avulla. Näillä voidaan määritellä eri sääntöjä siihen, minkä tyyppiset http-pyynnöt päästetään läpi. Itse tehtyjä suodattimia sovellukseen tehtiin kaksi. Toinen JWT:n autentikoimiseen ja toinen yleiseen tarkastukseen.

4.3.1 CORS Konfiguraatio

Sovelluksen turvallisuus konfiguraatiossa määritellään CORS-konfiguraatio sekä suodatinketju, joka suoritetaan jokaiselle tulevalle http-pyynnölle. Seuraavassa kuvassa (Kuva 8) nämä edellä mainitut osat ovat näkyvissä sovelluksen SecurityConfig-tiedostossa. Riviltä 81 alkaen määritellään sovelluksen CORS-asetukset. Näissä asetuksissa määritellään sallittaviksi http-metodeiksi POST, GET ja DELETE. Rivillä 86 asetetaan myös oletusarvoina kaikki originit ja headerit hyväksytyiksi.

Kuvan rivit väliltä 60–79 määrittelevät jokaiselle tulevalle http-pyynnölle tehdyn tarkastuksen. Tämä suoritetaan kaikille pyynnöille, jotka eivät tule palvelimen päätepisteisiin login, signup tai forgotPassword, koska näitä käytetään sovellukseen kirjautumiseen ja niiden on oltava kaikkien saatavilla.

```
60      @Bean
61      public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
62          //http.cors().configurationSource(request -> new CorsConfiguration().applyPermitDefaultValues())
63          http.cors().configurationSource(corsConfigurationSource())
64              .and()
65                  .csrf().disable()
66                  .authorizeHttpRequests()
67                  .requestMatchers(HttpMethod.OPTIONS, ...patterns: "/**").permitAll()
68                  .requestMatchers(...patterns: "/api/user/login", "/api/user/signup", "/api/user/forgotPassword").permitAll()
69                  .anyRequest().authenticated()
70              .and()
71                  .exceptionHandling()
72              .and()
73                  .sessionManagement()
74                  .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
75
76          http.addFilterBefore(jwtFilter, beforeFilter: UsernamePasswordAuthenticationFilter.class);
77
78          return http.build();
79      }
80
81      @Bean
82      CorsConfigurationSource corsConfigurationSource() {
83      CorsConfiguration cors = new CorsConfiguration();
84      cors.setAllowedMethods(Arrays.asList(...a: "POST", "GET", "DELETE"));
85      UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
86      source.registerCorsConfiguration(pattern: "/**", cors.applyPermitDefaultValues());
87      return source;
88  }
```

Kuva 8. Sovelluksen Security konfiguraatio

4.3.2 JWT-suodatin

Sovellukseen oli luotu myös JWT-suodatin, joka suoritetaan suodatinjonon ensimmäisenä. Tämä suodatin on nähtävissä seuraavassa kuvassa (Kuva 9). Myös tässä osoitteisiin login, signup tai forgotPassword tulevat pyynnöt päästetään läpi ilman JWT:n tarkastamista. Muissa tapauksissa tulevista pyynnöistä haetaan Authorization-otsikko ja jos se sisältää JWT:n, se validoidaan. Validointi voidaan nähdä riviväliltä 47–54.

Kuva 9. JWT-suodatin

4.4 Ulkoinen API

Korttien ja lisäosien tietojen hakemiseen käytettiin ulkoista Scryfall API:a. Tästä API:sta saaduista tiedoista karsittiin noin puolet pois sekä korttien, että lisäosien osalta. Myös haettujen korttien määrää rajattiin. Koska pelisarjassa on kymmeniä tuhansia kortteja, rajattiin haettu määrä noin 500 korttiin, jotta sovelluksen testaaminen olisi helpompaa ja nopeampaa.

Scryfall API:lle on laitettu rajoituksia pyyntöihin. Peräkkäisiin pyyntöihin pitäisi laittaa 50–100 millisekunnin viive tai noin 10 pyyntöä sekunnissa (Scryfall, LLC 2023). Tämä toteutettiin sovelluksessa siten, että kortit ja lisäosat haetaan kerran päivässä yhdellä pyynnöllä, tästä kerrotaan tarkemmin myöhemmässä kappaleessa. API:n dokumentaatiossa oli myös mainittu siitä, että hyvä käytäntö olisi hakea kortit ja tallentaa ne omaan lokaaliin tietokantaan, jotta kyselyiden tarve ja määrä vähenisi (Scryfall, LCC 2023).

4.5 Kortit ja lisäosat

Kortit ja lisäosat tallennetaan omiin käyttäjistä riippumattomiin tietokantakokoelmiin. Näille molemmille toteutettiin omat API:n päätepisteet. Päätepisteet eivät tarvinneet paljon toiminnallisuutta, joten vain muutamat http-GET menetelmät olivat tarpeen. Päätepisteet palauttavat käyttäjän pyytämät tiedot sekä palautuksen tilaa vastaavan http-tilakoodin.

Korttien osalta päätepisteet tehtiin kaikkien korttien-, yksittäisten korttien- sekä jonkin tietyn lisäosan korttien hakemiseen. Näiden toteutukset olivat yksinkertaisia eikä vaatineet juurikaan yhtään ohjelmallista logiikkaa. Seuraavassa kuvassa (Kuva 10) on esimerkkinä lisäosan korttien hakeminen. Tässä esimerkissä esiintyvä funktio hakee lisäosan uniikin koodin avulla Cards-kokoelmasta tietokannasta kaikki kortit, jotka sisältävät kyseisen koodin.

```
41      @Override
42      public ResponseEntity<List<CardWrapper>> getSetCards(String code) {
43          try {
44              return new ResponseEntity<>(cardDao.findSetCards(code), HttpStatus.OK);
45          } catch (Exception e) {
46              e.printStackTrace();
47              return new ResponseEntity<>(new ArrayList<>(), HttpStatus.INTERNAL_SERVER_ERROR);
48          }
49      }
50  }
51
```

Kuva 10. Lisäosan korttien hakeminen

Lisäosien päätepisteinä olivat kaikkien lisäosien tietojen- ja yksittäisen lisäosan tietojen hakeminen sekä kaikkien lisäosien uniikkien koodien hakeminen. Näiden päätepisteiden toiminnallisuus ja toteutus on lähes sama kuin korttien päätepisteissä, joten toiston välttämiseksi jätetään niiden selittäminen pois. Näiden lisäksi tehtiin vielä yksi päätepiste, joka tarkistaa onko annetulla lisäosalla siihen liittyviä lisäosia eli niin sanottuja lapsilisäosia. Tätä tarvittiin lisäosien näkymän yksinkertaistamiseksi.

4.6 Käyttäjäkohtaiset kokoelmat

Käyttäjien kokoelmien hallinnoimista varten tehtiin omat päätepisteet. Näitä olivat kokoelman hakeminen sähköpostiosoitteen avulla, kokoelmien muokkaaminen sekä lisäosakohtaisten keräilydatan hakeminen.

Kokoelmien hakemiseen tarvittiin pyynnön mukana annettu sähköpostiosoite, jota verrattiin kirjautuneen käyttäjän sähköpostiosoitteeseen. Tämä esti muita käyttäjiä pääsemästä käsiksi muiden käyttäjien kokoelmatietoihin. Tarkistuksen jälkeen palvelin hakee tietokannasta käyttäjän kokoelman finderID:tä käyttämällä. FinderID on kokoelmille annettu uniikki id, joka on käyttäjänimen ja sähköpostiosoitteen yhdistelmämerkkijono.

Metodi korttien muokkaamiseen ottaa parametreikseen sähköpostiosoitteen, jonka avulla tehdään samanlainen tarkistus kuin aiemmassa metodissa sekä requestMap nimisen Map-tyyppiä olevan parametrin. RequestMap sisältää kaksi listaa: lisättävät kortit sekä poistettavat kortit. Palvelin sitten muokkaa kokoelmien korttien collected-kenttää ja merkitsee ne kerätyiksi tai ei kerätyiksi. Seuraavassa kuvassa (Kuva 11) on nähtävissä pyynnön mukana saatujen listojen läpikäynti sekä listoissa olevien korttien keräilytilan muutos.

```
85          String[] idList = requestMap.get(key: "id_list");
86          String[] removeList = requestMap.get(key: "remove_list");
87
88          // Update cards property collected
89          for (CollectionCardWrapper card : collection.getCards()){
90              for (String id : idList){
91                  if (card.getId().equals(id)){
92                      //System.out.println("Found!");
93                      card.setCollected(collected: true);
94                  }
95              }
96
97              for (String id : removeList){
98                  if (card.getId().equals(id)){      You, 2 months ago • Co
99                      card.setCollected(collected: false);
100                 }
101             }
102         }
```

Kuva 11. Korttien keräilytilan muutos

Lisäosien keräilydatan hakemiseen luotu metodi ottaa parametreikseen sähköpostiosoitteen sekä lisäosan uniikin koodin, jotka annetaan http-pyynnön url:in mukana. Sähköpostiosoitteella tehdään taas tarkistus. Tämä metodi palauttaa käyttäjälle lisäosan korttien- sekä käyttäjän keräämien korttien lukumäärän. Lisäosan korttien lukumäärä haetaan tietokannasta lisäosan tiedoista. Kerättyjen korttien lukumäärä lasketaan tarkastelemalla käyttäjän kokoelmaa annetun lisäosakoodin perusteella ja lasketaan, kuinka monessa kortissa collected-arvo on tosi.

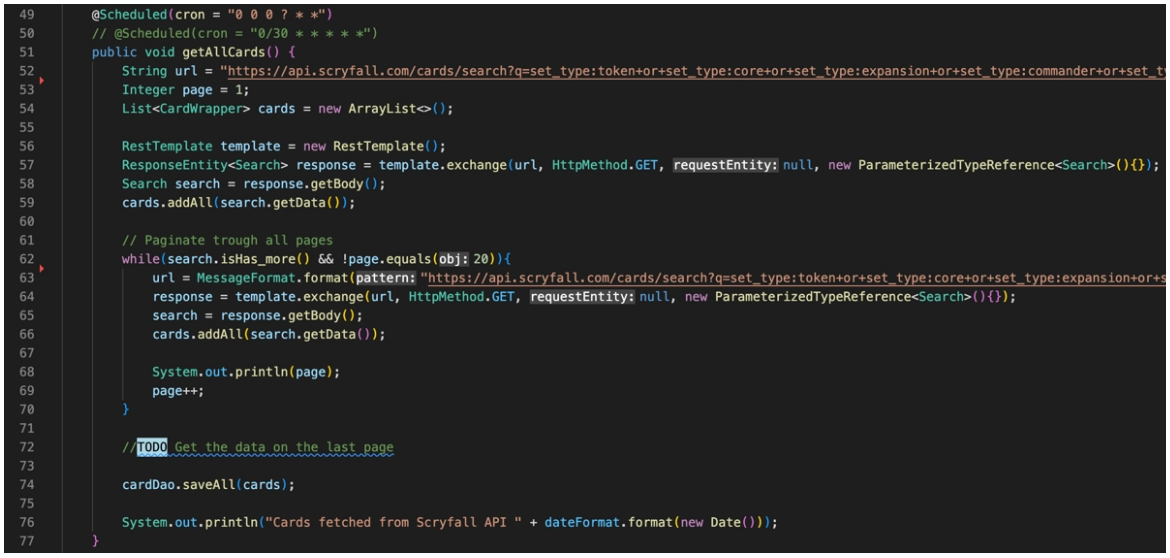
4.7 Ajoitetut tehtävät

Koska pelisarjaan tulee säännöllisin väliajoin uusia lisäosia ja niiden mukana lisää kortteja sekä myös korttien hinnat vaihtelevat päivittäin, tarvittiin jokin tapa päivittää tietokantaa tietyin väliajoin. Tämän ratkaisemiseksi tehtiin kolme ajoitettua tehtävää käyttäen @Scheduled-annotaatiota ja cron menetelmää.

4.7.1 Korttien päivittäminen

Ensimmäisen tehtävän tarkoituksena on etsiä uusia kortteja, joita ei tietokannassa vielä ole ja ennestään tietokannassa olevien korttien hintatietojen päivittäminen. Alhaalla olevassa kuvassa (Kuva 12) tehtävä hakee ulkoisesta API:sta kaikki pelisarjan kortit ja iteroi läpi kaikki sivut, jotka saadaan kyselyn vastauksessa. Jokaisen sivun kortit lisätään ArrayList-tyyppiseen varastoon. Ja kun kaikki sivut on käyty läpi, saatu valmis lista tallennetaan Cards-kokoelmaan tietokantaan. Jos listan korteissa on samat tiedot, eikä uusia kortteja ole, ei tapahdu mitään. Jos uusia kortteja löytyy tai korttien tiedot ovat

vaihtuneet, tietokantaa päivitetään. Tämä tehtävä tehdään aina ensimmäisenä ja se asetettiin tapahtumaan joka vuorokausi keskiyöllä.



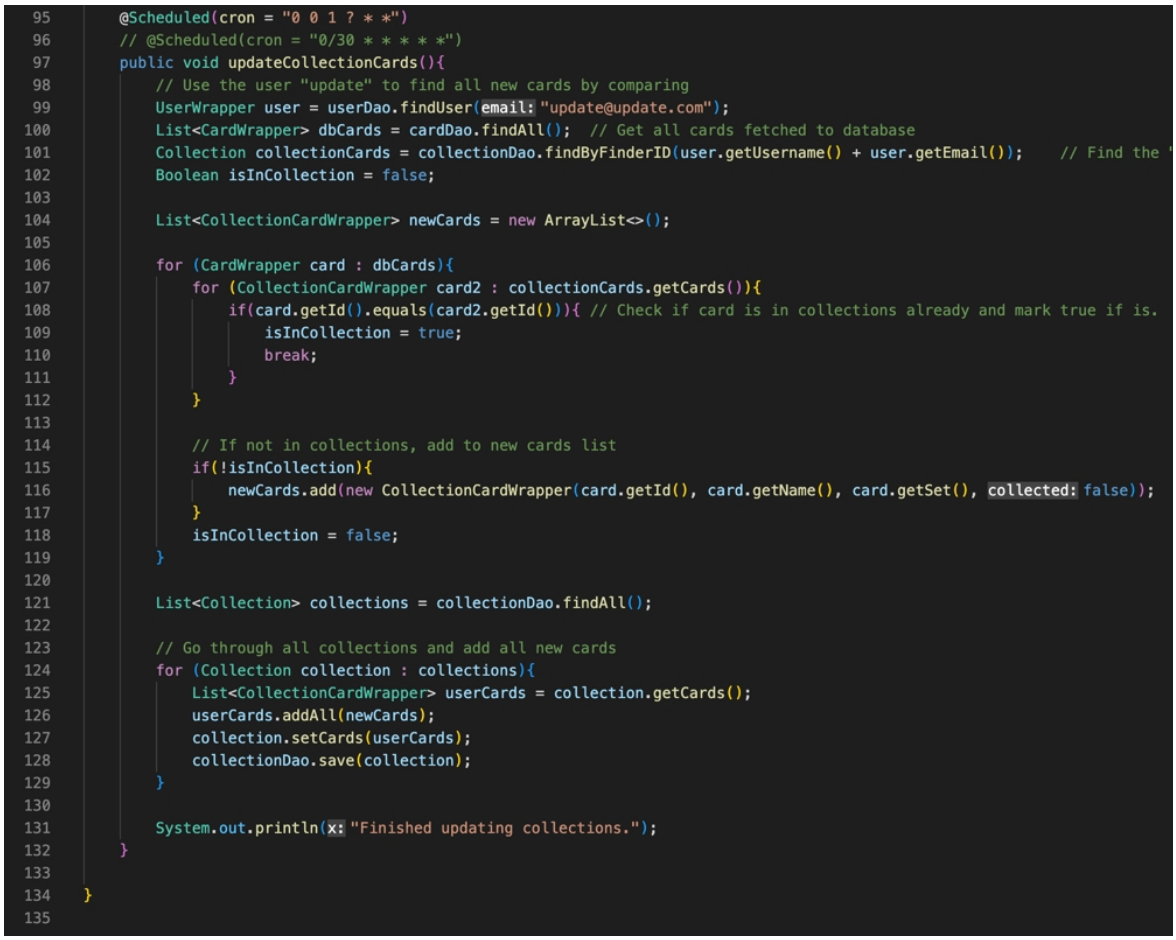
```
@Scheduled(cron = "0 0 0 ? * *")
// @Scheduled(cron = "0/30 * * * * *")
public void getAllCards() {
    String url = "https://api.scryfall.com/cards/search?q=set_type:token+or+set_type:core+or+set_type:expansion+or+set_type:commander+or+set_t
    Integer page = 1;
    List<CardWrapper> cards = new ArrayList<>();

    RestTemplate template = new RestTemplate();
    ResponseEntity<Search> response = template.exchange(url, HttpMethod.GET, requestEntity: null, new ParameterizedTypeReference<Search>(){});
    Search search = response.getBody();
    cards.addAll(search.getData());

    // Paginate trough all pages
    while(search.isHas_more() && !page.equals(obj: 20)){
        url = MessageFormat.format(pattern: "https://api.scryfall.com/cards/search?q=set_type:token+or+set_type:core+or+set_type:expansion+or+s
        response = template.exchange(url, HttpMethod.GET, requestEntity: null, new ParameterizedTypeReference<Search>(){});
        search = response.getBody();
        cards.addAll(search.getData());

        System.out.println(page);
        page++;
    }

    //TODO Get the data on the last page

    cardDao.saveAll(cards);

    System.out.println("Cards fetched from Scryfall API " + dateFormat.format(new Date()));
}
```

Kuva 12. Esimerkki ajoitetusta tehtävästä

4.7.2 Kokoelmien päivittäminen

Toinen tehtävä ajetaan tunti ensimmäisen jälkeen. Tämän tehtävän tarkoituksena on päivittää kaikkien olemassa olevien käyttäjien kokoelmien tiedot. Koska kokoelmissa on vain korttien minimaaliset tiedot niin riitti, että lisätään kaikki uudet kortit, jos niitä löytyy. Seuraavassa kuvassa (Kuva 13) nähdään tehtävän koodi.

Sovellukseen luotiin yksi käyttäjä tätä tehtävää varten. Tätä käyttäjää käytetään vain korttitietojen vertailuun. Tehtävä hakee sille tehdyn käyttäjän kokoelman tiedot ja vertaa niitä tietokannan Cards-kokoelmaan. Jos uusia kortteja ei löydy, tehtävä lopettaa toiminnan siihen pisteeseen. Jos uusia kortteja kuitenkin löytyy, tehtävä siirtyy käymään läpi kaikkien muiden käyttäjien kokoelmia ja päivittää korttien tiedot jokaiseen kokoelmaan.



```
95      @Scheduled(cron = "0 0 1 ? * *")
96      // @Scheduled(cron = "0/30 * * * * *")
97      public void updateCollectionCards(){
98          // Use the user "update" to find all new cards by comparing
99          UserWrapper user = userDao.findUser(email: "update@update.com");
100         List<CardWrapper> dbCards = cardDao.findAll();  // Get all cards fetched to database
101         Collection collectionCards = collectionDao.findByFinderID(user.getUsername() + user.getEmail());    // Find the '
102         Boolean isInCollection = false;
103
104         List<CollectionCardWrapper> newCards = new ArrayList<>();
105
106         for (CardWrapper card : dbCards){
107             for (CollectionCardWrapper card2 : collectionCards.getCards()){
108                 if(card.getId().equals(card2.getId())){ // Check if card is in collections already and mark true if is.
109                     isInCollection = true;
110                     break;
111                 }
112             }
113
114             // If not in collections, add to new cards list
115             if(!isInCollection){
116                 newCards.add(new CollectionCardWrapper(card.getId(), card.getName(), card.getSet(), collected: false));
117             }
118             isInCollection = false;
119         }
120
121         List<Collection> collections = collectionDao.findAll();
122
123         // Go through all collections and add all new cards
124         for (Collection collection : collections){
125             List<CollectionCardWrapper> userCards = collection.getCards();
126             userCards.addAll(newCards);
127             collection.setCards(userCards);
128             collectionDao.save(collection);
129         }
130
131         System.out.println(x: "Finished updating collections.");
132     }
133
134 }
135
```

Kuva 13. Kokoelmien päivittäminen

4.7.3 Lisäosien hakeminen

Lisäosien hakeminen tehdään kolmannessa tehtävässä tunti toisen tehtävän jälkeen. Tämän tehtävän tarkoitus on tarkistaa, löytyykö ulkoisesta API:sta uusia lisäosia. Tehtävä hakee API:sta kaikki lisäosat ja tallentaa ne tietokantaan. Jos uusia ei löydy, ei tapahdu mitään, mutta jos uusia lisäosia on saatavilla, ne lisätään tietokantaan.

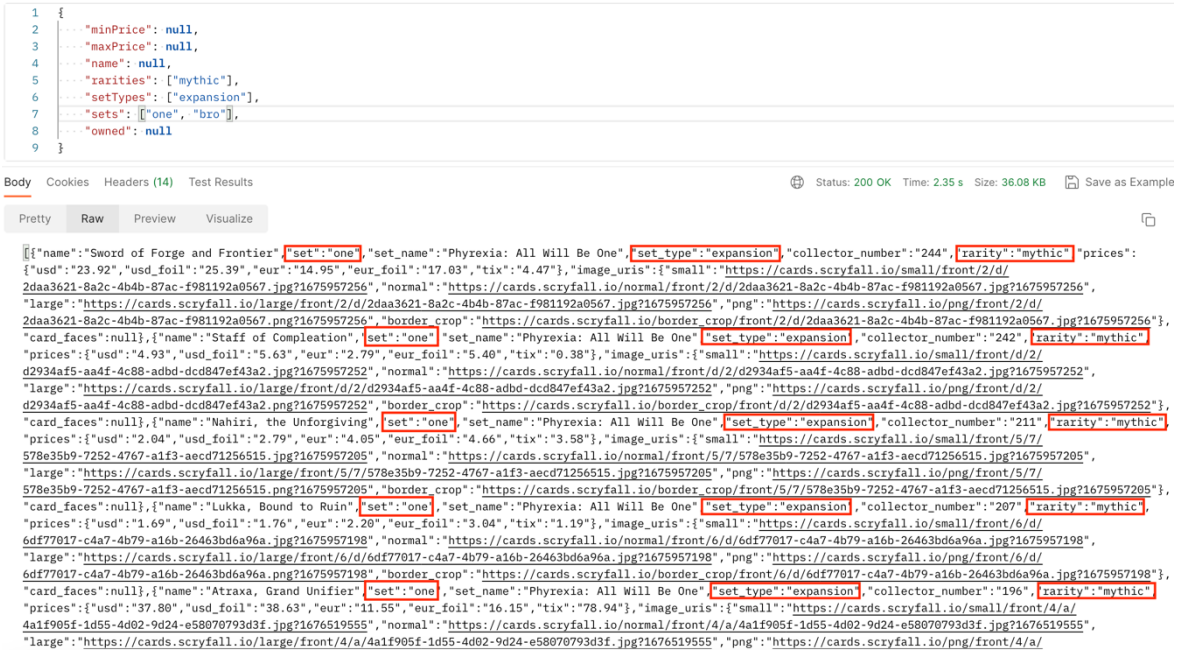
Koska uusia lisäosia tulee harvemmin, eikä niiden tiedot voi jälkikäteen muuttua, tämän tehtävän ajastuksen voisi muuttaa pidemmäksi aikaväliksi esimerkiksi kerran kuussa, mutta koska kyselyn vastaus on todella lyhyt, ei nykyisellä ajastuksella ole merkitystä sovelluksen toimintaan.

4.8 Korttien hakutoiminto

Sovellukseen toteutettiin myös korttien hakutoiminto, jolla voidaan hakea kortteja tietokannasta, jotka vastaavat annettuja rajauksia. Tätä toimintoa varten luotiin hakutoiminnolle oma itsenäinen päätepiste. Tämä toiminnallisuus oli vaikea toteuttaa ja tuloksena tästä tuli toimiva kokonaisuus, mutta tässä on vielä paljon parannettavaa.

Tällä palvelulla on kaksi metodia. SearchCards palauttaa listan korteista, jotka vastaavat annettuja rajauksia. convertCardWrapperToSearchWrapper on metodi, joka muuttaa tietokannasta löydettyjen korttien muodon SearchWrapper-tyyppisiksi.

SearchCards-metodi saa parametreikseen http-pyynnölle annetut rajaukset. Metodin saamat rajaukset tulevat ClientSearch-oliona, joka sisältää kentät korttien nimelle, harvinaisuuksille, lisäosan tyypeille, minimi- ja maksimihinnoille ja lisäosille. Rajausten perusteella etsitään tietokannasta kortteja, jotka vastaavat kaikkia annettuja kenttiä. Seuraavassa kuvassa (Kuva 14) on nähtävissä esimerkkihaku Postman-työkalua hyöndyntämällä. Tässä esimerkissä haetaan kortteja, jotka ovat harvinaisuudeltaan ”mythic”, kuuluvat lisäosatyyppiin ”expansion” sekä kuuluvat lisäosiin ”Phyrexia: All Will Be One (ONE)” ja ”Brother’s War (BRO)”.



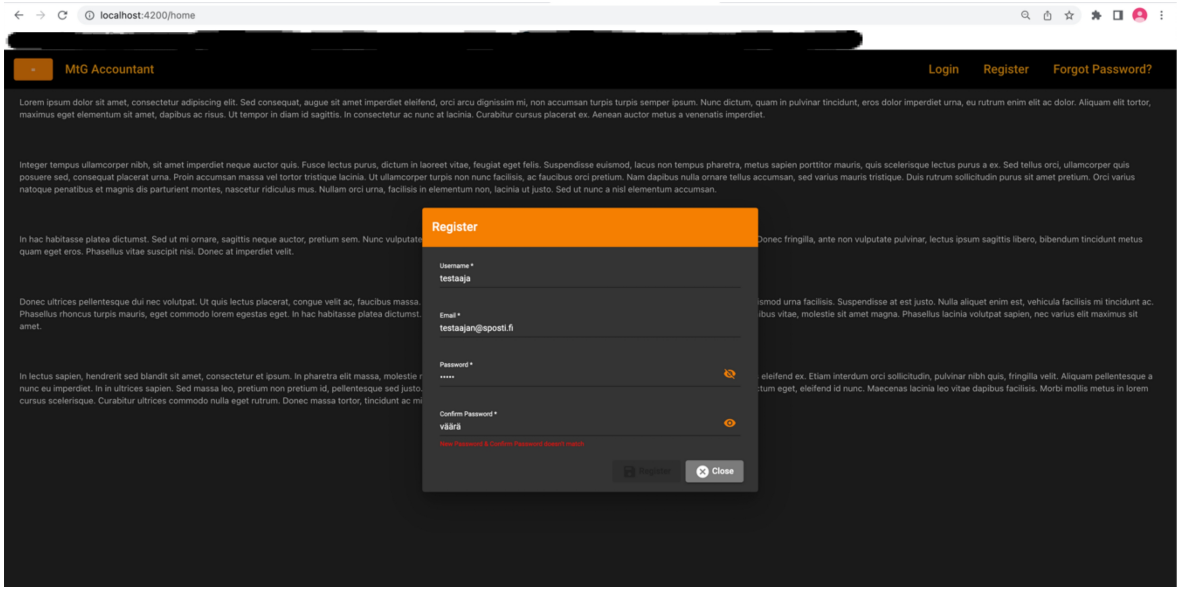
Kuva 14. Esimerkkihaku ja sen tulokset

5 Käyttöliittymä

5.1 Rekisteröityminen, kirjautuminen sekä unohtunut salasana

Sovelluksen kotinäkymässä, joka on saatavilla käyttäjille, jotka eivät ole kirjautuneet sovellukseen on nähtävissä kolme linkkiä: Login, Register ja Forgot Password. Nämä linkit avaavat dialog-tyyppisen ikkunan, jossa on niiden toimintaa vastaava lomake.

Jotta käyttäjä voi kirjautua sovellukseen, on hänen rekisteröidyttävä ensin. Rekisteröintilomake sisältää kentät sähköpostiosoitteelle, käyttäjänimelle, salasanalle ja salasanan vahvistamiselle. Jotta lomake voidaan lähettää palvelimelle, on kaikki kentät täytettävä. Sähköpostiosoitteen on oltava oikeassa muodossa sekä salasana kenttiin asetettujen salasanojen on vastattava toisiaan. Seuraavassa kuvassa (Kuva 15) on rekisteröitymisdialogi, johon on testimielessä annettu väärä salasana salasanan vahvistus kenttään.



Kuva 15. Rekisteröintivaiheen Dialog-ikkuna

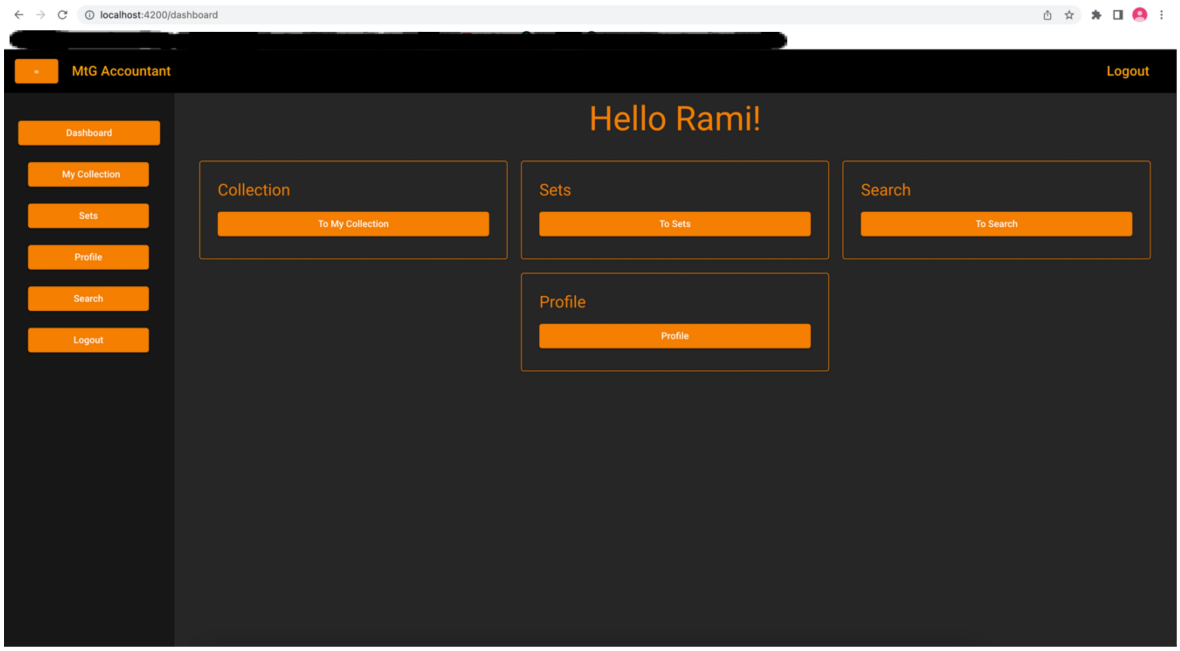
Kirjautumislomakkeessa sovellukseen pääsee käsiksi oikealla sähköpostiosoitteen ja salasanan yhdistelmällä. Lomake lähetetään palvelimelle, joka palauttaa JWT-tokenin. Saatu JWT sekä siitä eritelty sähköpostiosoite tallennetaan selaimen paikalliseen varastoon (eng. local storage), jonka avulla pystytään tarkastelemaan käyttäjän kirjautumistilannetta.

5.2 Navigointi

Sovelluksen syvempi navigointi on rajattu vain kirjautuneille käyttäjille. Kirjautumisen jälkeen sovelluksessa navigoimiseen on kaksi tapaa: Dashboard näkymässä olevien painikkeiden avulla tai pikavalikosta löytyvien painikkeiden avulla. Jälkimmäinen tapa toteutettiin sen takia, ettei käyttäjän tarvitse aina palata dashboard-näkymään halutessaan siirtyä näkymään, joka sijaitsee eri polussa.

Kun käyttäjä on kirjautunut sovellukseen, hänelle aukeaa dashboard-näkymä, joka on sovelluksen keskus. Tästä näkymästä käyttäjälle avautuu polut kaikkiin sovelluksen toimintoihin. Tässä käyttäjällä on mahdollisuus siirtyä kokoelman tai lisäosien tarkasteluun, hakutoimintoon tai käyttäjän profiiliin. Käyttäjän ollessa admin-käyttäjä, hänelle aukeaa myös käyttäjien hallintapaneeli tähän näkymään.

Pikavalikon komponentti sisällytettiin sovelluksen päänäkymään, jotta se on kaikissa näkymissä saatavilla. Pikavalikon voi avata ja sulkea mieltymyksen mukaan ja se skaalaa näkymän muut komponentin sen mukaan. Seuraavassa kuvassa (Kuva 16) on nähtävissä molemmat navigointitavat.



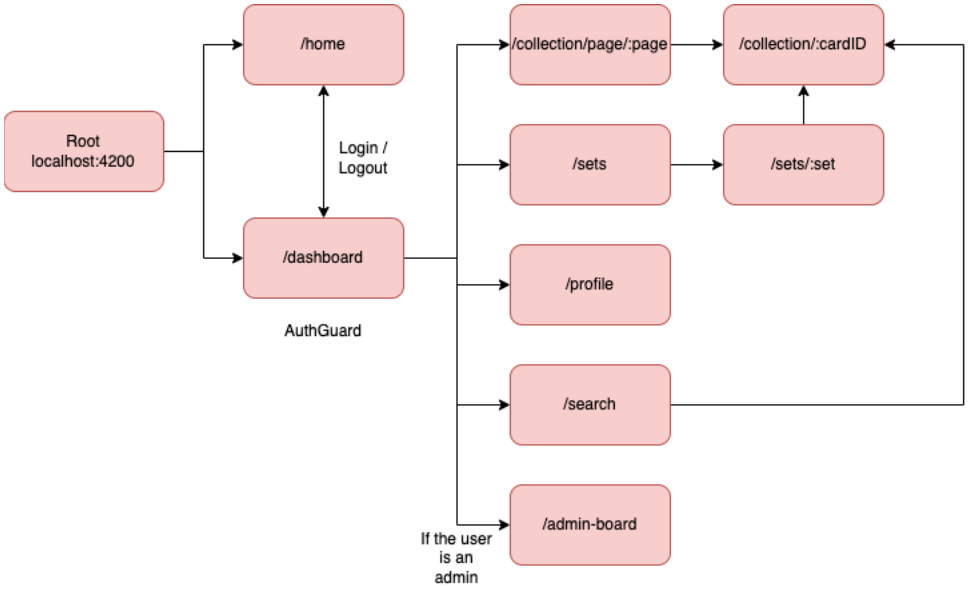
Kuva 16. Dashboard-näkymän navigointi sekä sivupalkki

5.2.1 Reititys

Sovelluksen reititys on jaettu kahteen moduuliin: root ja dashboard. Sovellukseen tultaessa käyttäjälle avautuu kotinäkymä, joka on sovelluksen esittelynäkymä. Kirjautumisen jälkeen sovellus navigoi dashboard-näkymään ja lukitsee käyttäjän pois

kotinäkymästä. Tämä tarkoittaa sitä, että jos käyttäjä pyrkii palaamaan kotinäkymään kirjautumatta ulos, sovellus uudelleenohjaa käyttäjän takaisin dashboardiin.

Dashboard näkymässä käyttäjälle avautuu monipuolisemmat reitit, joita havainnollistetaan seuraavassa kuviossa (Kuvio 7). Tästä näkymästä käyttäjä voi selata oman kokoelmansa tietoja eri näkökulmista esimerkiksi sarjan mukaan. Käyttäjä pystyy navigoimaan yksittäisen kortin tietoihin monen eri reitin kautta. Dashboard näkymässä reitti admin-paneeliin on saatavilla vain admin käyttäjille.



Kuvio 7. Angular sovelluksen reititys

5.2.2 Reittien suojaus

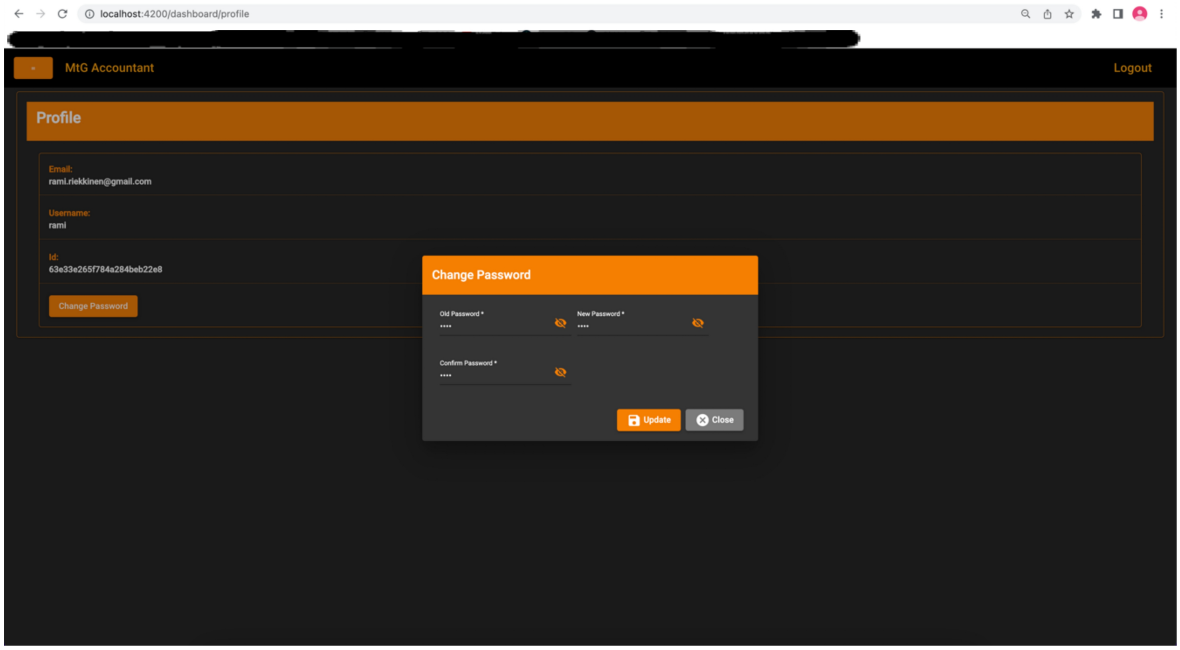
Sovelluksen dashboard-moduuli on suojattu AuthGuard-luokalla, joka implementoi Angularin omaa CanActivate-luokkaa. Tämä tarkastaa löytyykö selaimen muistista sinne kirjautumisvaiheessa tallennettua JWT-tokenia. Jos JWT löytyy, sovellus sallii käyttäjän etenemisen dashboard-näkymään, jos ei, sovellus ohjaa käyttäjän takaisin kotinäkymään.

Sovelluksessa myös implementoitiin Angularin HttpInterceptor-luokkaa, joka keskeyttää http-pyynnöt ja suorittaa niille omat käsittelyt. Tällä tavalla lisätään pyyntöihin palvelimen tarvitsema JWT-Token Authorization header. Jos pyyntö palauttaa virheen, sovellus tarkastaa virheen mukana saadun http-statuskoodin. Jos status on 401 tai 403, sovellus uudelleen reitittää käyttäjän kotisivulle. Muussa tapauksessa selaimen paikallinen varasto tyhjennetään ja käyttäjä kirjataan ulos.

5.3 Profiilinäkymä ja salasanan vaihtaminen

Käyttäjän profiilinäkymässä voidaan tarkastella käyttäjän tietoja. Tällä hetkellä tästä näkymästä voidaan nähdä käyttäjän sähköpostiosoite, käyttäjänimi sekä käyttäjien uniikki id. Näitä tietoja voidaan tarpeen vaatiessa lisätä, mutta tämänhetkisessä versiossa ei ollut tarpeen näyttää muita tietoja.

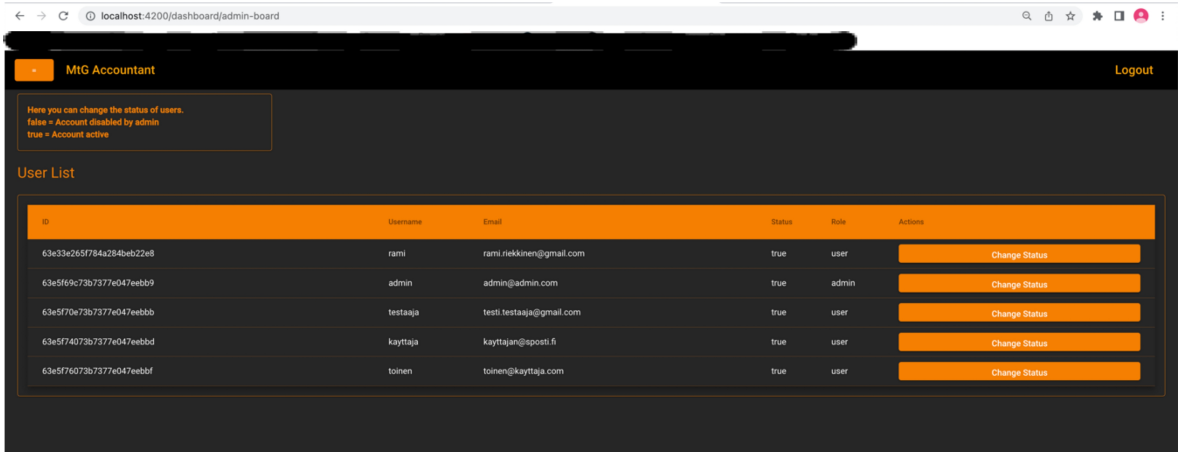
Tässä näkymässä on myös mahdollista vaihtaa käyttäjän salasana. Näkymässä on painike, jota painamalla käyttäjälle aukeaa Angular Material-kirjaston Dialog-komponentti, joka sisältää salasanan vaihtolomakkeen. Tähän lomakkeeseen käyttäjän on annettava vanha salasanansa sekä uusi salasana, joka pitää kirjoittaa kaksi kertaa eri kenttiin. Seuraavassa kuvassa (Kuva 17) nähdään kyseinen dialogi-ikkuna sekä taustalla profiilinäkymä.



Kuva 17. Profiilinäkymä sekä salasanan vaihtaminen

5.4 Admin-paneeli

Admin-roolin omaavilla käyttäjillä on mahdollisuus päästä käyttäjien hallintapaneeliin. Tässä näkymässä admin voi tarkastella kaikkien sovelluksen käyttäjien tietoja sekä vaihtaa eri käyttäjien tilan. Nykyisessä versiossa tämän näkymän toiminnallisuus on rajoitettu vain käyttäjien tilan muuttamiseen, mutta tätä on suunniteltu laajennettavan sovelluksen seuraavissa versioissa. Seuraavassa kuvassa (Kuva 18) on nähtävissä admin-paneeli, jota varten tehtiin muutamia testi käyttäjiä.



Kuva 18. Admin-paneelin hallintanäkymä

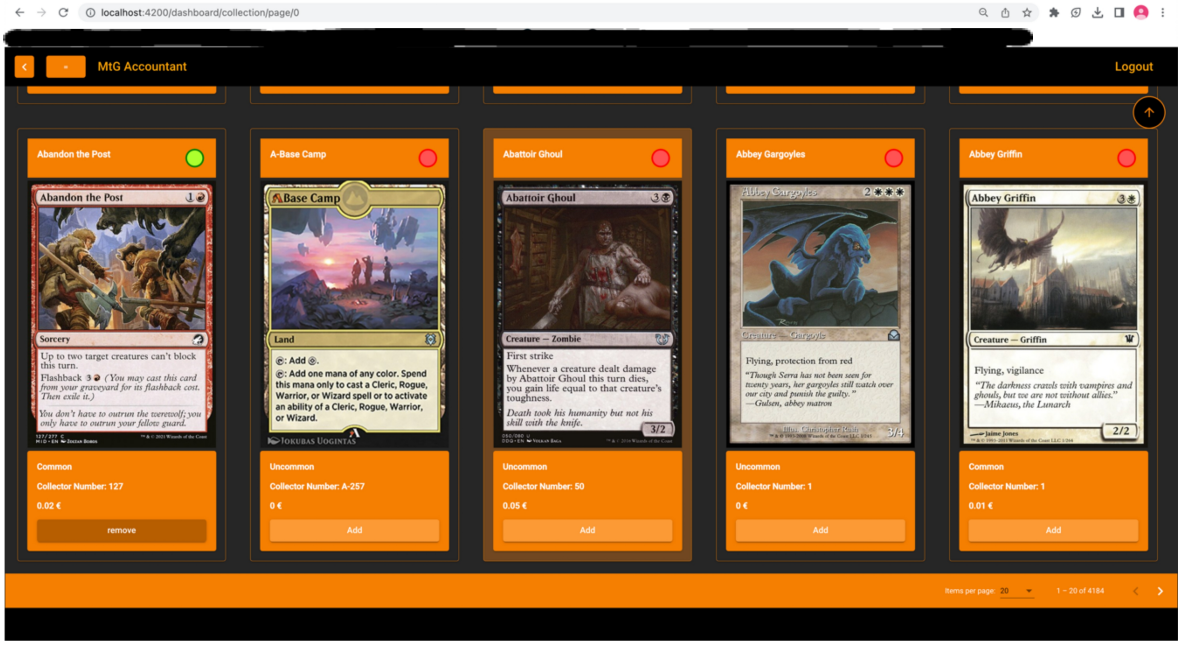
Sovelluksen jatkokehitystä varten toteutettiin toiminto, jolla admin käyttäjät voivat poistaa käyttäjiä väliaikaisesti käytöstä. Tätä toimintoa voidaan mahdollisesti käyttää, jos sovellukseen toteutetaan chat-ominaisuus tai muu vastaava toiminto, jossa käyttäjät ovat yhteydessä toisiinsa. Jos admin vaihtaa jonkin käyttäjän tilan muotoon epätosi, kyseinen käyttäjä ei voi kirjautua sovellukseen ennen kuin tila on admin-käyttäjien toimesta asetettu takaisin arvoon tosi. Käyttäjän tilan muutos tehdään painiketta painamalla, joka lähettää palvelimelle pyynnön, joka vaihtaa käyttäjän tilan.

5.5 Kokoelman ja korttien tarkastelu

Kokoelman ja korttien tarkastelu ja edistymisen seuraaminen olivat sovelluksen päätavoitteita. Näitä varten luotiin viisi eri näkymää. Collection-näkymästä on mahdollista selata kaikkia pelisarjan kortteja. Sets-näkymässä voidaan selata eri lisäosia ja valita niistä jokin tarkempaan selaamiseen Set-näkymässä. Yksittäisen kortin näkymässä voidaan nähdä jonkin tietyn kortin tarkemmat tiedot. Sovellukseen tehtiin myös hakunäkymä, josta voidaan lähettää palvelimelle lomaketyyppinen kysely, jonka avulla voidaan etsiä kortteja eri rajauksilla.

5.5.1 Kaikki kortit

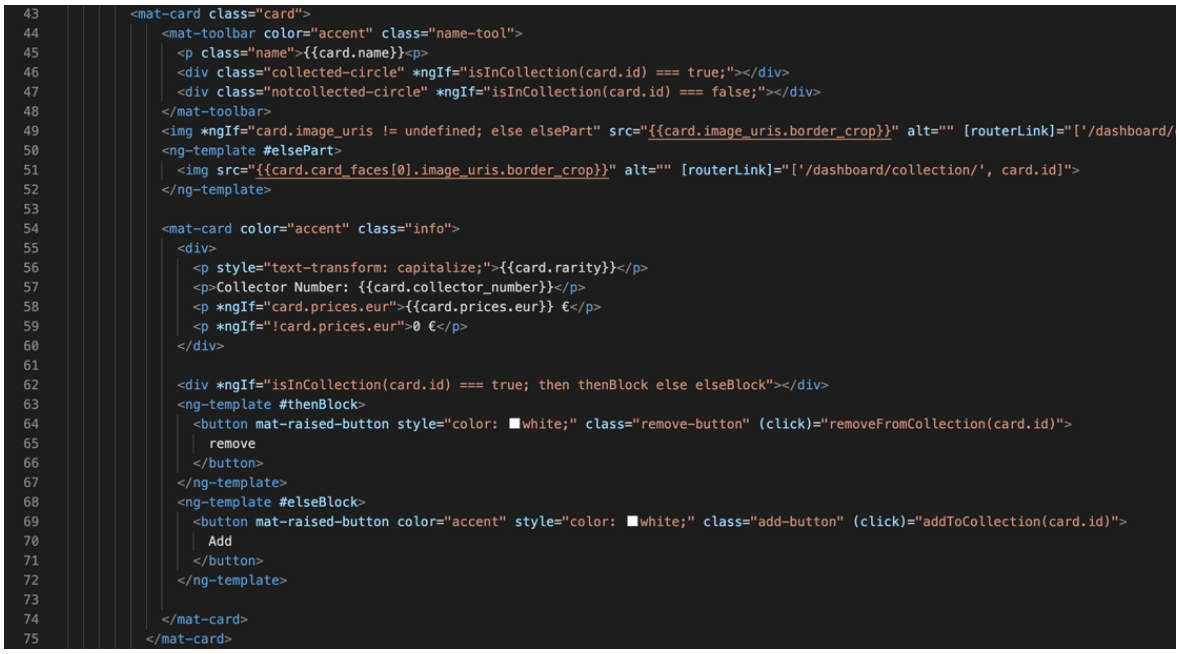
Kaikkia pelisarjan kortteja voidaan selata polun ”/dashboard/collection” alta. Tässä näkymässä käyttäjä voi selata kaikkia pelisarjan kortteja. Koska kortteja on pelisarjassa kymmeniä tuhansia, näkymä toteutettiin siten, että käyttäjä näkee 20 tai 40 korttia kerrallaan. Seuraavassa kuvassa (Kuva 19) esimerkki Collections-näkymästä. Tässä esimerkkikuvassa on navigoitu toiselle sivulle ja valittu näytettäväksi 20 korttia.



Kuva 19. Collections-näkymä sivutuksella

Tässä näkymässä käyttäjä voi halutessaan lisätä tai poistaa kortteja omasta kokoelmastaan. Käyttäjä voi myös siirtyä tarkastelemaan haluamaansa korttia tarkemmin painamalla hiirellä jonkin kortin kuvaa, joka siirtää käyttäjän kyseisen kortin tietoihin.

Kortit haetaan käyttöliittymän käyttöön tekemällä palvelimelle http-pyyntö, joka palauttaa tietokannasta löytyvät kortit taulukkona. Käyttöliittymä sitten asettaa kaikki taulukon kortit omaan Angular Materialin Card-komponenttiin, jotka voi aiemmasta kuvasta (Kuva 19) tunnistaa kortteja ympäröivistä oransseista viivoista. Seuraavassa kuvassa (Kuva 20) voidaan nähdä, kuinka tämä on toteutettu käyttöliittymän koodissa.



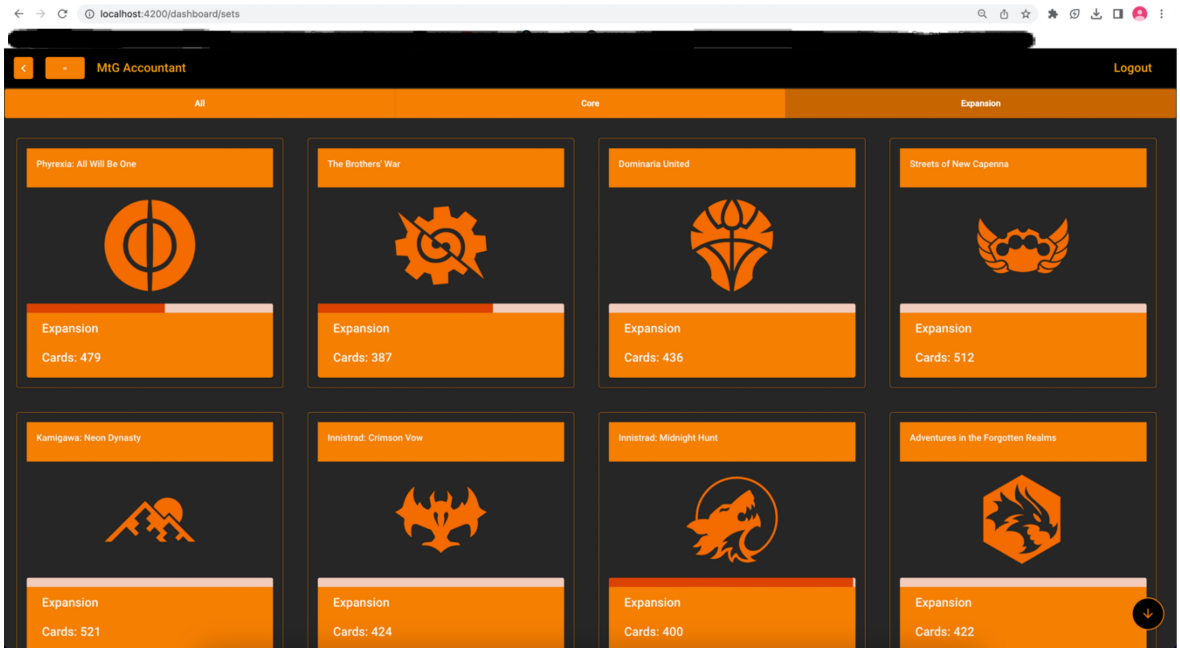
```
<mat-card class="card">
    <mat-toolbar color="accent" class="name-tool">
      <p class="name">{{card.name}}<p>
      <div class="collected-circle" *ngIf="isInCollection(card.id) === true;"></div>
      <div class="notcollected-circle" *ngIf="isInCollection(card.id) === false;"></div>
    </mat-toolbar>
    <img *ngIf="card.image_uris != undefined; else elsePart" src="{{card.image_uris.border_crop}}" alt="" [routerLink]="['/dashboard/
    <ng-template #elsePart>
      <img src="{{card.card_faces[0].image_uris.border_crop}}" alt="" [routerLink]="['/dashboard/collection/', card.id]">
    </ng-template>

    <mat-card color="accent" class="info">
      <div>
        <p style="text-transform: capitalize;">{{card.rarity}}</p>
        <p>Collector Number: {{card.collector_number}}</p>
        <p *ngIf="card.prices.eur">{{card.prices.eur}} €</p>
        <p *ngIf="!card.prices.eur">0 €</p>
      </div>

      <div *ngIf="isInCollection(card.id) === true; then thenBlock else elseBlock"></div>
      <ng-template #thenBlock>
        <button mat-raised-button style="color: white;" class="remove-button" (click)="removeFromCollection(card.id)">
          remove
        </button>
      </ng-template>
      <ng-template #elseBlock>
        <button mat-raised-button color="accent" style="color: white;" class="add-button" (click)="addToCollection(card.id)">
          Add
        </button>
      </ng-template>

    </mat-card>
  </mat-card>
```

Kuva 20. Korttien template-muoto

5.5.2 Lisäosat

Lisäosien tarkastelu jaettiin sovelluksessa kahteen näkymään. Ensimmäinen on listatyyppinen näkymä eri lisäosista (Sets). Tässä näkymässä käyttäjä voi valita tarkempaan tarkasteluun haluamansa lisäosan, joka näytetään omassa näkymässään (Set).

Sets-näkymässä käyttäjälle näytetään pelisarjan päälisäosat, joka tarkoittaa sitä, että lisäosien lapsilisäosat sisällytetään niiden parent-lisäosaan. Lisäosat esitetään Angular Materialin Card-komponentteina. Näissä komponenteissa voidaan nähdä lisäosan ikoni, seurantapalkki sekä perustiedot, joiksi valittiin lisäosan nimi, -tyyppi sekä -korttien lukumäärä. Näitä komponentteja painamalla käyttäjä siirtyy kyseisen lisäosan tarkempaan esittelyyn. Sets-näkymässä voidaan myös rajata näytettäviä lisäosia tyypin mukaan. Käyttäjä voi valita näytetäänkö kaikki lisäosat vai vain ”core”- tai ”expansion”-tyyppiä olevat lisäosat. Seuraavassa kuvassa (Kuva 21) näkyy Sets-näkymä, johon on rajattu vain ”expansion”-tyyppiä olevat lisäosat.



Kuva 21. Sets-näkymä

Lisäosien tarkemmassa esityksessä käyttäjälle näytetään ensin lisäosan ikoni ja tarkemmat tiedot sekä kokoelman tilan seurantapalkki näkymän yläosassa (Katso Kuva 22). Edellä mainitut tiedot on pakattu omaan Angular Material Card-komponenttiin, jonka sisällä ne ovat omissa korteissaan. Kuvassa näkyvä seurantapalkki seuraa kerättyjen korttien määrää, jonka käyttöliittymä saa palvelimelta. Tällä tavoin käyttäjä pystyy seuraamaan omaa edistymistään eri lisäosissa. Näiden tarkempien tietojen jälkeen käyttäjä pystyy selaamaan kaikkia kortteja, mitä valittu lisäosa sisältää. Käyttäjä voi tässä näkymässä lisätä tai poistaa kortteja omasta kokoelmastaan. Koska tässä näkymässä korttien määrä pysyy melko pienenä, ei sivutukselle ollut tarvetta.

Kuva 22. Lisäosan tarkempi näkymä

Seuraavassa kuvassa (Kuva 23) on nähtävissä koodiesimerkki, miten seurantapalkki toimii, kun kokoelmaan lisätään kortti. GetCollectedCountFromSet-metodi tekee palvelimelle kyselyn, jolla se hakee lisäosan keräilytiedot, jonka jälkeen se palauttaa ne käyttöliittymän käsittelyyn, joka alustaa eri muuttujat saaduilla arvoilla. AddToCollection-metodia kutsutaan aina, kun käyttäjä lisää kortin kokoelmaan. Seurantapalkin osalta se kasvattaa kerättyjen korttien määrää ja laskee uudelleen seurantapalkin leveyden.

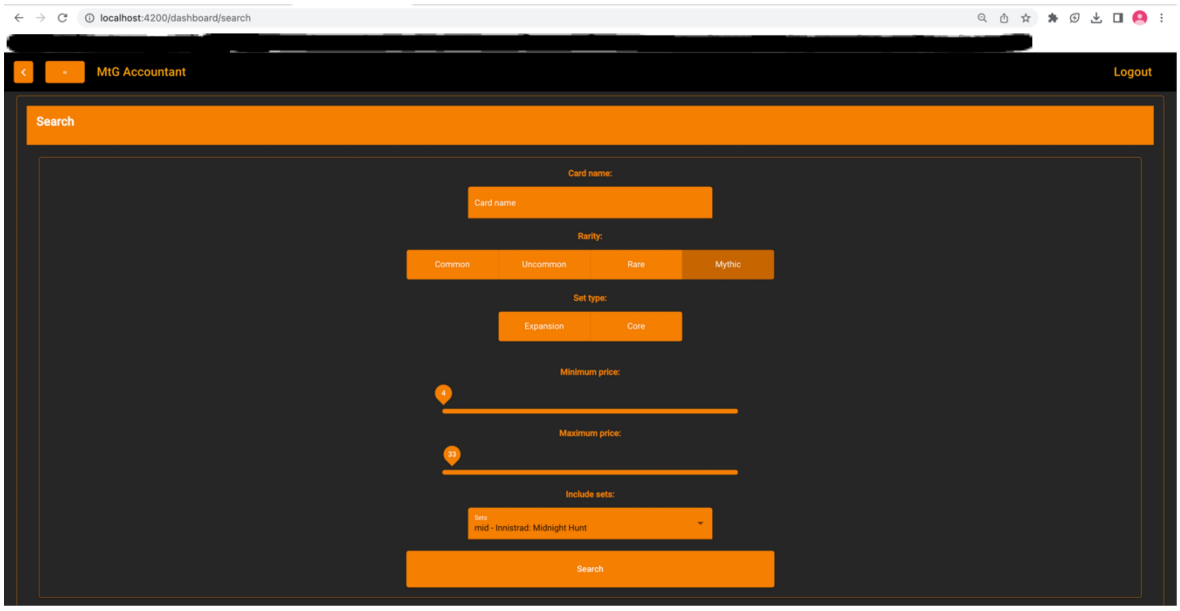
Lisäosan tietojen lisäksi näkymässä on mahdollista pika liikkua eri lapsilisäosiin sekä lajitella kortteja eri määritelmin esimerkiksi nimen tai hinnan mukaan. Näkymässä on myös mahdollista kelata sivun alkuun ja loppuun. Tässä näkymässä on myös mahdollista lisätä tai poistaa kaikki kortit kyseisestä lisäosasta. Tämä toiminnallisuus on sovelluksen tämänhetkisessä versiossa vain mahdollista päälisäosan korteille.

```
// ------------ Irrelevant Code Here--------------

ngOnInit(): void {
    this.getCollection()
    this.getCards()
    this.getSetData(this.code)
    this.getCollectedCountFromSet(this.code)
    this.progressWidth = (this.collectedData.collected / this.collectedData.totalCount)*100
}

getCollectedCountFromSet(code: string){
    this.cardsService.getCollectedCountFromSet(code).subscribe(collected => this.collectedData = collected)
    this.cardsService.getCollectedCountFromSet(code).subscribe(collected => this.collected = collected.collected)
    this.cardsService.getCollectedCountFromSet(code).subscribe(collected => this.progressWidth = (collected.collected /
}ollected.totalCount)*100)

// ------------ Irrelevant Code Here--------------

addToCollection(id: string): void {
  console.log(id);

  // This is needed for updating the view
  for (const element of this.collection) {
    if(element.id == id){
      element.collected = true
    }
  }

  this.collected++
  this.progressWidth = (this.collected / this.set.card_count) * 100

  // update to db happens here
  this.cardsService.updateCollection([""], [id]);
  this.cdr.detectChanges()
}

// ------------ Irrelevant Code Here--------------
```

Kuva 23. Seurantapalkin toimintaa

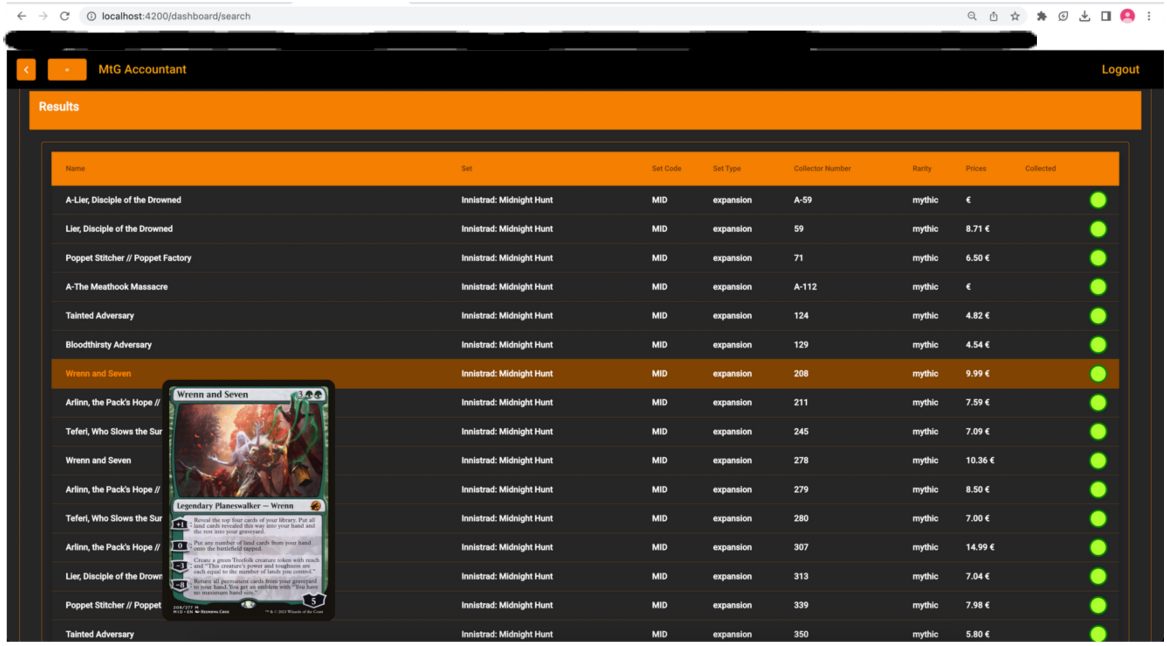
5.5.3 Haku

Sovelluksessa kortteja voidaan myös etsiä hakutoiminolla. Hakunäkymässä käyttäjälle avautuu lomaketyyppinen näkymä, jossa käyttäjä voi etsiä kortteja eri rajauksin. Käyttäjä voi etsiä kortteja nimen, harvinaisuuden, tyypin, hinnan, lisäosan tai näiden yhdistelmän perusteella. Seuraavassa kuvassa (Kuva 24) on näytettynä hakulomake esimerkkirajauksin.



Kuva 24. Hakunäkymä esimerkkirajauksin

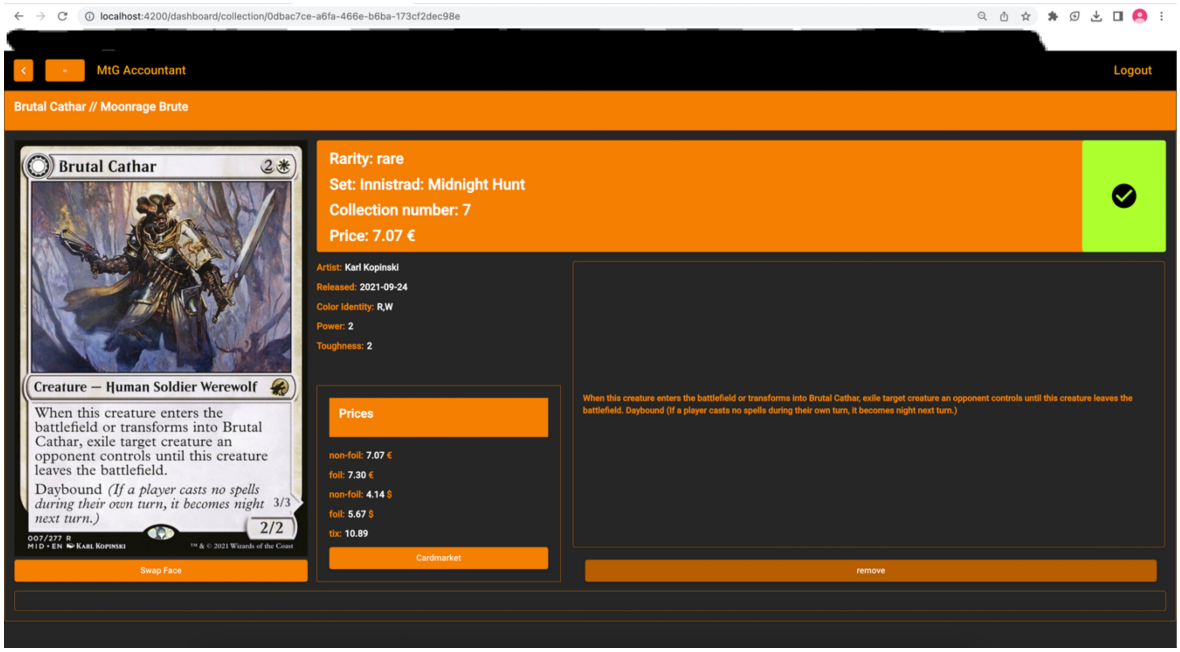
Kun käyttäjä painaa search-painiketta, käyttöliittymästä lähetetään pyyntö palvelimelle, joka koostuu käyttäjän määrittelemistä rajauksista. Palvelin sitten palauttaa käyttöliittymälle listan korteista, jotka vastaavat käyttäjän antamia rajauksia. Tämän jälkeen ruudulle aukeaa uusi komponentti, jossa tuloksena saadut kortit ovat listattuna taulukkoon. Seuraavassa kuvassa (Kuva 25) on aiemman kuvan (Kuva 24) rajauksia vastaavat tulokset. Tulosten nimi kenttää painamalla käyttäjä voi myös liikkua kyseisen kortin tarkempaan tarkasteluun. Tuloksia selaamalla käyttäjän on mahdollista nähdä esikatselukuva haluamastaan kortista osoittamalla hiiri sen nimeen.



Kuva 25. Hakutulosten näyttäminen

5.5.4 Yksittäisen kortin tarkastelu

Yksittäisen kortin tarkastelunäkymään on mahdollista päästä monia eri reittejä (Katso Kuvio). Kun käyttäjän on jonkin reitin kautta valinnut tarkasteltavan kortin, hänelle aukeaa tässä luvussa käsiteltävä näkymä. Tässä näkymässä käyttäjä näkee valitun kortin tarkemmat tiedot. Näitä tietoja ovat tällä hetkellä nimi, harvinaisuus, lisäosa, keräilynumero, hintatiedot, artistin nimi, julkaisuvuosi, väritiedot ja toimita- sekä kuvaustekstit. Tässä näkymässä käyttäjä voi myös lisätä tai poistaa kyseisen kortin kokoelmastaan. Seuraavassa kuvassa (Kuva 26) on esimerkkinäkymä kortille Brutal Cathar.



Kuva 26. Yksittäisen kortin näkymä

Magic the Gathering -pelisarjassa on myös kaksipuoleisia kortteja. Näiden korttien tiedot voivat vaihdella riippuen siitä kumpi puoli esitetään esimerkiksi kortin power tai toughness arvot voivat muuttua. Tätä varten tehtiin painike, jota painamalla vaihdetaan näytettävät tiedot ja kuva.

Yksittäisen kortin näkymässä näytetään myös kortin hintatiedot euroina ja dollareina. Kortin hintaan voi vaikuttaa onko kyseessä kiiltokortti vai ei, joten myös näille tapauksille lisättiin omat kentät. Hintatietojen alapuolella on myös linkki sivustolle, josta kortin voi ostaa.

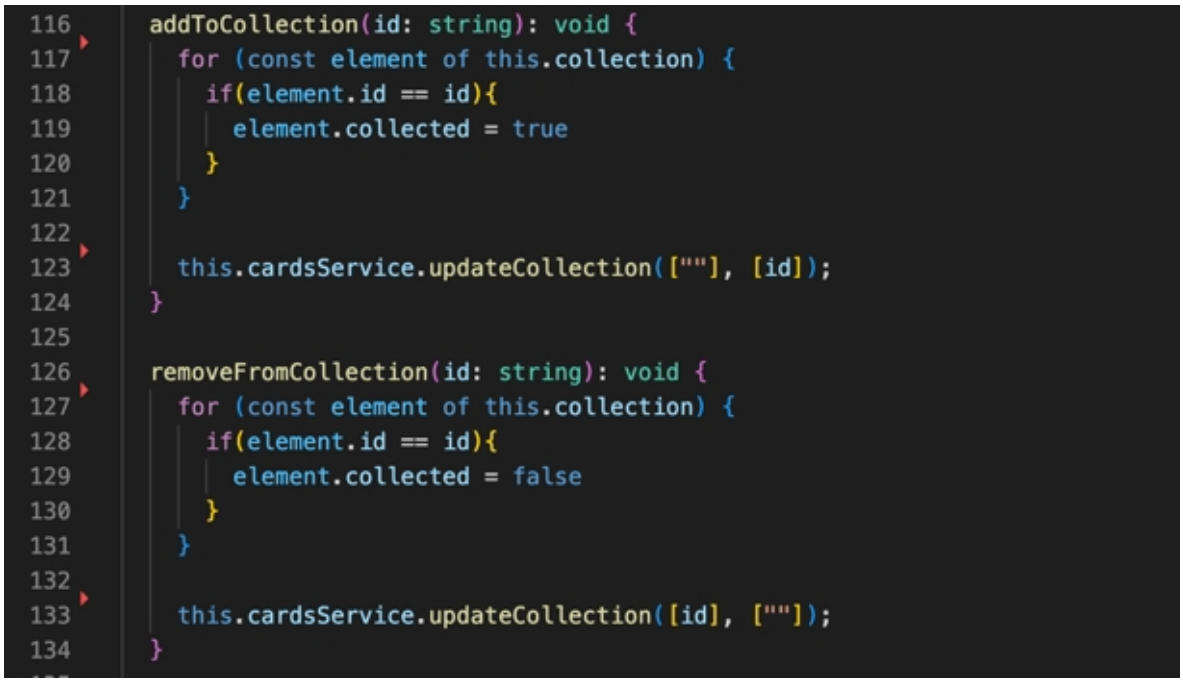
5.6 Kokoelmaan lisääminen ja poistaminen

Korttien lisääminen tai poistaminen on sovelluksen yksi päätavoitteista ja nämä toiminnot löytyvät Set- ja Collection-näkymistä. Kortteja poistetaan tai lisätään painamalla painikekomponenttia, joka löytyy jokaisen kortin alta aiemmin mainituissa näkymissä. Sen sijaan, että sovellukseen olisi tehty molemmille toiminnoille omat painikkeensa, tämä toteutettiin yhdellä painikkeella, jonka tila riippuu siitä, onko kortti merkitty kerätyksi vai ei.

Sovelluksen aiemmissa versioissa käyttäjä ensin merkitsi haluamansa kortit lisätyiksi tai poistetuiksi, jonka jälkeen palvelimelle lähettiin pyyntö, jonka mukana tuli useita kortteja listattuna. Tätä tapaa käytetään myös muissa samankaltaisissa sovelluksissa. Tämä toteutustyyppi on kuitenkin ongelmallinen ja siitä voi koitua suurta harmia käyttäjälle. Yleisenä esimerkkinä: jos käyttäjä lisäilee kortteja ja siinä vaiheessa, kun käyttäjä on lähettämässä lomaketta palvelimelle, tapahtuu jokin verkkovirhe. Tässä tilanteessa käyttäjän muokkaamat tiedot häviävät ja hän joutuu tekemään kaiken uudelleen. Tämän ongelman ratkaisuksi annettiin käyttäjälle painike jokaisen kortin alle, jota painamalla keräilytieto tallennetaan heti eikä aiemmin mainittua ongelmaa tapahdu.

Näkymän alustuksessa kaikkien korttien tila tarkistetaan ja käyttäjälle näytetään tilaa vastaava painike. Painikkeiden toiminnot eroavat hieman aiemmin mainituissa näkymissä, mutta niiden päärakenne on sama. Tämä johtuu Set-näkymässä näytettävässä seurantapalkista.

Painikkeita painamalla palvelimelle lähetetään POST-pyyntö, jonka bodyn mukana annetaan kaksi taulukkoa, jotka sisältävät poistettavat ja lisättävät kortit. Tämä tapa on jäännös aiemmista prototyypeistä, mikä selvennettiin aiemmassa luvussa. Tässä sovelluksen versiossa näistä taulukoista vain toiseen lisätään yksi kortin id-kenttä ja toiseen sisällytetään tyhjä merkkijono, jotta palvelin pystyy erottamaan kumman toimenpiteen se suorittaa. Alhaalla olevassa kuvassa (Kuva 27) nähdään korttien lisääminen ja poistaminen Angularin TypeScript-tiedostossa.



```
116   addToCollection(id: string): void {
117     for (const element of this.collection) {
118       if(element.id == id){
119         element.collected = true
120       }
121     }
122
123     this.cardsService.updateCollection([""], [id]);
124   }
125
126   removeFromCollection(id: string): void {
127     for (const element of this.collection) {
128       if(element.id == id){
129         element.collected = false
130       }
131     }
132
133     this.cardsService.updateCollection([id], [""]);
134   }
```

Kuva 27. Korttien lisääminen ja poistaminen

6 Docker-konttien luominen

Kun sovelluksen toteutusosa saatiin valmiiksi, siirryttiin tästä tekemään Docker konttia. Sovelluksen projektirakenteen takia tarvitsi luoda kaksi Docker Imagea, joista ensimmäinen tehtiin sovelluksen käyttöliittymän www-palvelimelle ja toinen sovelluksen palvelimelle.

Käyttöliittymän Docker image pohjautuu noden 19.5 versioon. Tälle imagelle luodaan oma hakemistorakenne ja sinne kopioidaan sovelluksessa luotu package.json tiedosto, jonka mukaan asennetaan käyttöliittymän tarvitsemat riippuvuudet. Riippuvuuksien asentamisen jälkeen kopioidaan käyttöliittymän rakenne ja annetaan oman ympäristön portti 4200 imagen käyttöön. Lopuksi ajetaan komento ”npm start”, joka käynnistää käyttöliittymän. Seuraavassa kuvassa (Kuva 28) on nähtävissä käyttöliittymän Dockerfile josta on nähtävissä edellä mainitut vaiheet.

```
FROM node:19.5.0-alpine

RUN mkdir -p /usr/src/mtgacc
RUN mkdir -p /usr/src/mtgacc/client

WORKDIR /usr/src/mtgacc/client

COPY package*.json ./
RUN npm install

COPY . .

EXPOSE 4200

CMD [ "npm", "start" ]
```

Kuva 28. Käyttöliittymän Dockerfile

Palvelinsovelluksen Docker imagen luominen koostuu kahdesta osasta: sovelluksen rakentamisesta ja sen käynnistämisestä. Molemmat vaiheet pohjautuvat Oraclen OpenJDK:n versioon 17. Sovelluksen rakentamisvaiheessa kopioidaan sovelluksen tarvitsemat tiedostot ja asennetaan sen tarvitsemat riippuvuudet. Tämän jälkeen sovelluksesta luodaan ajettava tiedosto. Käynnistysvaiheessa kopioidaan riippuvuudet ja

ajetaan sovelluksen pääohjelma. Seuraavassa kuvassa (Kuva 29) nähdään palvelinsovelluksen Dockerfile.

```
FROM openjdk:17-oracle as build
WORKDIR /mtgacc/server

COPY mvnw .
COPY .mvn .mvn
COPY pom.xml .
COPY src src
RUN chmod +x mvnw

RUN ./mvnw install -DskipTests
RUN mkdir -p target/dependency && (cd target/dependency; jar -xf ../*.jar)

FROM openjdk:17-oracle
VOLUME /tmp
ARG DEPENDENCY=/mtgacc/server/target/dependency
COPY --from=build ${DEPENDENCY}/BOOT-INF/lib /server/lib
COPY --from=build ${DEPENDENCY}/META-INF /app/META-INF
COPY --from=build ${DEPENDENCY}/BOOT-INF/classes /server
ENTRYPOINT ["java","-
cp","server:server/lib/*","com.github.mtgaccountant.server.ServerApplication"]
```

Kuva 29. Palvelinsovelluksen Dockerfile

Koska sovellus koostuu kahdesta Docker Imagesta, luotiin sovellukselle vielä docker-compose.yml tiedosto. Tämä tiedosto linkittää aiemmin luodut imaget toisiinsa ja käynnistää ne. Seuraavassa kuvassa (Kuva 30) on nähtävissä kyseinen compose-tiedosto. Tässä tiedostossa sovelluksen käyttöliittymän image linkitetään palvelinsovelluksen imageen. Compose-tiedostossa asetetaan myös luotavien konttien käyttöön niille määritetyt portit. Palvelinsovellus tarvitsi myös käyttöönsä ympäristömuuttujatiedoston, joka asetetaan kuvassa kohdassa ”env_file”.

```
version: '3'
services:
  client:
    build: ./client
    ports:
      - "4200:4200"
    container_name: mtgacc-client
    restart: always
    links:
      - server

  server:
    build: ./server
    ports:
      - "8080:8080"
    container_name: mtgacc-server
    restart: always
    env_file:
      - server/.env
```

Kuva 30. Docker-compose.yml tiedosto

7 Avoin lähdekoodi

7.1 Avoimen lähdekoodin merkitys sovelluksen kehityksessä

Opinnäytetyössä toteutettu sovellus haluttiin tehdä avoimen lähdekoodin projektina. Koska keräilykorttisarjoja on lukuisia, jää sovelluksen täysi potentiaali hyödyntämättä. Avoimen lähdekoodin ansiosta keräilykorttiyhteisö voisi jatkokehittää sovellusta tukemaan useampia keräilykorttisarjoja esimerkiksi Pokemon- tai Yu-Gi-Oh-sarjoja. Sovelluksen versionhallinnassa on käytetty Githubia.

Yhteisön mielipiteiden ja kehityksen avulla sovellukseen saataisiin kaikille mieleisiä ominaisuuksia. Tällä hetkellä sovellus soveltuu opiskelijan omiin tarpeisiin, mutta muut keräilijät saattavat keskittyä omissa kokoelmissaan eri alueisiin. Tällä tavoin sovelluksesta saataisiin mahdollisimman kattava kokonaisuus. Eri pelisarjoista kiinnostuneet voivat esimerkiksi kehittää sovellusta tukemaan muitakin pelisarjoja kuin Magic the Gathering. Kehityksessä pitäisi kuitenkin aina pitää mielessä keräilynäkökulma, joka on haluttu pitää sovelluksen keskiössä. Esimerkiksi pelattavuuteen ei haluta ottaa kantaa, eikä siihen haluta lisätä siihen liittyviä toiminnallisuuksia. Yhteisön osallistuminen projektiin helpottaisi myös siihen käytettävää työmäärää, mikä tarkoittaisi sitä, ettei kaikkea tarvitsisi tehdä itse.

Avoimen lähdekoodin ansiosta projektiin on mahdollista saada monipuolisempaa osaamista, mikä helpottaa isompien muutosten toteuttamisessa. Esimerkiksi mobiilisovelluksen luonti.

Koska sovelluksen lähdekoodi on avointa, voidaan sen avulla myös tehdä itselle räätälöityjä versioita. Yksittäiset kehittäjät voivat siis jatkokehittää sovellusta haluamaansa suuntaan. Tällä tavoin voidaan saada mahdollisesti sama pohja useille eri keräilykorttisarjoille. Tämän ansiosta sovelluksen näkökulmaa voidaan myös muuttaa ilman, että siitä koituu haittaa sen alkuperäiselle ratkaisulle.

Useiden kehittäjien avulla saataisiin myös useamman henkilön näkökulma sovelluksen kehitykseen. Pitkäaikaisen kehityksen yksi huomattavista ongelmista on niin sanottu ”sokeutuminen”. Tämä tarkoittaa sitä, että mahdollisia ongelmia, puutteita tai kehittävää on välillä vaikea enää löytää. Useiden kehittäjien ja näkökulmien ansiosta vikoja ja puutteita olisi helpompi löytää.

7.2 Lisenssi

Ohjelmistolisenssi on laillinen sopimus, joka määrittelee, kuinka ohjelmistoa voidaan käyttää. Näin voidaan asettaa rajoituksia siihen, miten sovellusta voidaan muokata,

käyttää tai jakaa. Avoimen lähdekoodin projekteissa voidaan lisenssien avulla saada projekti muiden saataville. (Horcasitas 2021.) Jos avoimen lähdekoodin projektiin ei sisällytetä lisenssiä, jossa toisin määrätään, kukaan muu ei voi kopioida, jakaa tai muokata projektia ilman laillisten toimenpiteiden vaaraa (Github 2023).

Avoimen lähdekoodin projekteille on olemassa olevia lisenssipohjia. Nämä pohjat voidaan jakaa kahteen osa-alueeseen: sallivat lisenssit ja copyleft-lisenssit. Sallivat lisenssit antavat käyttäjille luvan käyttää, muokata ja jakaa lähdekoodia, mutta käyttäjillä on myös mahdollisuus muuttaa jakelun ehtoja. Tämä tarkoittaa sitä, että jos projektista johdetaan oma projekti, voidaan tämä jakaa eri ehdoilla kuin mitä alkuperäisen teoksen lisenssi olisi saattanut vaatia. (Horcasitas 2021.) MIT-lisenssi on esimerkki sallivasta lisenssistä (Fossa Inc 2023).

Copyleft-lisenssi antaa myös luvan käyttää, muokata ja jakaa lähdekoodia, mutta se tarjoaa suojan uudelleenlisensointia vastaan. Tämä tarkoittaa sitä, että alkuperäisestä teoksesta johdettu tuotos on julkaistava alkuperäisen teoksen lisenssiehdoilla. (Horcasitas 2021.) GPL-lisenssit ovat esimerkkejä copyleft-lisensseistä (Fossa Inc 2023).

Sovelluksen lisenssin valinnassa vertailtiin näitä edellä mainittuja kategorioita. Seuraavassa kuvassa (Kuva 31) on vertailu taulukko suosittujen avoimen lähdekoodin lisensseistä ja niiden rajoituksista. Sovellus halutaan olevan kaikkien saatavilla, joten vertailun tuloksena voidaan asettaa sovellus copyleft-lisenssin alle. Copyleft-lisensseistä vertailtiin GNU GPLv3- (kuvassa ensimmäinen sarake) sekä MPL 2.0-lisenssejä (kuvassa viides sarake). Huomattavin ero näillä kahdella oli muutosten ilmoittaminen. GNU GPLv3-lisenssi vaatii käyttäjää ilmoittamaan tehdyistä muutoksista lisenssin alle sijoittuvasta materiaalista. MPL 2.0-lisenssit eivät tätä vaadi. Vertailujen tuloksena hyvänä mahdollisena lisenssinä sovellukselle pidetään MPL 2.0-lisenssiä. Tällä tavoin käyttäjät voivat pitää sovelluksen omassa käytössä ja muokata sitä tarpeidensa mukaan. Myös sovelluksesta periytyvät julkaisut pysyvät kaikkien saatavilla.

snyk	Copyleft					Permissive			
	GPL	AGPL	LGPL		MPL	Apache	MIT	BSD	Unlicense
Permissions in addition to commercial use, distribution, modification:									
Patent use	●	●	●	●	●	●	●	●	●
Patent use	●	●	●	●	●	●	●	●	●
Conditions									
Disclose source	●	●	●	●	●	●	●	●	●
License & copyright notice	●	●	●	●	●	●	●	Source	●
Network use is distribution	●	●	●	●	●	●	●	●	●
Same license	●	●	Library	●	File	●	●	●	●
State changes	●	●	●	Some	●	●	●	●	●
Limitations/Disclaimers									
Liability	●	●	●	●	●	●	●	●	●
Warranty	●	●	●	●	●	●	●	●	●
Trademark use	No explicit limitation				●	●	●	●	●

Kuva 31. Lisenssien vertailutaulukko (Snyk Limited 2023)

8 Jatkokehitys

8.1 Käyttäjien välinen interaktio ja admin-paneelin monipuolisempi hallinnointi

Sovelluksen kehitysvaiheessa yksi isompia rakenteellisia muutoksia vaativista ideoista oli chat-toiminto käyttäjien välille. Tämä mahdollistaisi käyttäjien välisen interaktion, jonka kautta käyttäjät voisivat sopia keskenään esimerkiksi korttien vaihdoista tai muusta sen kaltaisesta. Myös jonkinlainen foorumi olisi myös vaihtoehto, mikä mahdollistaisi saman sekä myös antaisi käyttäjille mahdollisuuden keskustella pelisarjaan liittyvistä asioista esimerkiksi uusista lisäosista.

Interaktioon liittyen myös admin-käyttäjien toiminnallisuuksia halutaan lisätä. Tällä hetkellä admin-käyttäjät voivat vain hallinnoida käyttäjien kirjautumista, mutta tähän haluttaisiin enemmän mahdollisuuksia. Näitä voisivat olla esimerkiksi käyttäjien manuaalinen salasanan vaihto ja keskustelujen valvominen.

8.2 Mobiilisovellus

Sovelluksen mobiiliversio olisi yksi isoimmista askeleista eteenpäin sovelluksen kehittämisessä. Tämä toisi paljon lisää mahdollisuuksia, joiden avulla sovelluksesta tulisi houkuttelevampi useammille käyttäjille. Tämä toisi myös lisää toimintoja, joita ei tietokoneella voida toteuttaa. Näistä yksi huomattava esimerkki olisi korttien lisääminen kameran avulla. Näin käyttäjä voisi uutta pakkaa avatessa suoraan tarkistaa onko kortit jo merkitty kerätyiksi ja lisätä niitä kokoelmaan sitä mukaan, kun uusia kortteja löytyy.

Mobiiliversio tarkoittaisi sitä, että sovelluksen käyttöliittymä jouduttaisiin tekemään kokonaan uudelleen. Tämä tarkoittaa sitä, että sovelluksen siirtäminen mobiiliin pitäisi ajoittaa hyvään väliin. Sovelluksen mobiiliversio toteutettaisiin suurella todennäköisyydellä React Native teknologialla.

8.3 Web-sovellus vai työpöytäsovellus

Koska käyttäjillä on eri mieltymyksiä sovellusten käytettävyydestä etenkin web-sovellusten ja työpöytäsovellusten välillä, nousi esille idea sovelluksen työpöytäversiosta. Jatkoa varten haluttiin miettiä tämän toteuttamista. Jatkokehitystä varten haluttiin pohtia alustavasti, tehdäänkö sovellus kokonaan uudestaan työpöytäsovelluksena, tehdäänkö molemmat vai jätetään sovellus nykyiseen tilaan.

9 Yhteenveto ja pohdinta

Yhteenvetona opinnäytetyön tavoitteena oli suunnitella ja toteuttaa full-stack-websovellus Magic the Gathering -keräilykorttien kirjanpitoa varten. Lisäksi tavoitteina oli siistin ja helppokäyttöisen käyttöliittymän tekeminen Angular-kehystä käyttämällä, palvelimen ja API:n toteutus Javan Spring-kehyksen Spring Boot-lisäosaa käyttämällä, tietokannan toteuttaminen NoSQL-tietokantaa käyttäen sekä jatkokehityksen pohtiminen. Sovelluksen tuli olla toimiva kokonaisuus, jota on helppo jatkossa kehittää eteenpäin. Yleisellä tasolla kaikkiin tavoitteisiin päästiin sekä kaikki mainittavat ongelmat ratkaistiin.

Sovelluksesta saatiin jonkin verran käyttäjäpalautetta. Yleisellä tasolla palaute oli positiivista. Sovelluksen toiminnallisuudessa ei ollut moitteita, mutta käyttöliittymän suunnittelusta saatiin rakentavaa palautetta.

Palautteissa käyttöliittymän isoimmat ongelmat liittyivät navigoimiseen. Nämä ongelmat johtuivat osin siitä, että testikäyttäjät eivät olleet tuttuja sovellukseen sisällytetyn korttisarjan kanssa. Koska sovellus on suunniteltu käyttäjille, jotka tuntevat pelisarjan entuudestaan, itse pelisarjaan liittyviä ongelmia ei oteta huomioon sovelluksen jatkokehityksessä.

Palautteiden perusteella käyttöliittymässä liikkuminen tuntui alussa sekavalta sekä jotkin merkinnät olivat hankalia ymmärtää. Näitä olivat esimerkiksi korttien kerättymerkintä, joka näkyy esimerkiksi kuvassa 25 (Kuva) vihreällä pallolla merkittynä. Tähän olikin toiveena saada jokin selitys merkinnöiden merkityksistä.

Palautteissa oli myös toiveena saada jokin tieto, että jotain tapahtuu. Tämä on huomattavissa esimerkiksi hakunäkymässä, jossa haun jälkeen kestää hetken ennen kuin tulokset tulevat näkyviin. Tähän toivottiin jonkin näköinen indikaattori siitä, että sovellus tekee jotain, eikä käyttäjälle tulisi mieleen, että sovellus on rikki. Tämä on toteutettavissa esimerkiksi pyörivän latausindikaattorin avulla, jota sovelluksessa on jo käytetty jossain määrin.

Sovelluksen värityylistä annettiin myös palautetta ja tähän toivottiin väripaletin laajempaa käyttöä, mikä tällä hetkellä on hieman vähäistä. Epämukavuutta aiheutti esimerkiksi mustan ja kirkkaan oranssin väriyhdistelmä. Tätä aiotaan jatkossa parantaa laajentamalla paletin käyttöä, mutta värityyli halutaan silti pitää pelisarjan teemaa mukailevana.

Sovelluksen nykyisessä versiossa on toteutettu sen kaikki perustoiminnot. Käyttäjä voi rekisteröitymisen ja kirjautumisen jälkeen hallinnoida omaa kokoelmaa ja tarkastella sitä eri näkökulmista. Käyttäjä voi myös tarkastella omaa profiilia ja sen kautta vaihtaa oman

salasanan. Kortteja voi myös hakea eri rajauksin esimerkiksi hinnan mukaan. Admin-käyttäjät pystyvät hallinnoimaan muiden käyttäjien kirjautumismahdollisuuksia ja tarkastelemaan muiden käyttäjien tietoja.

Keräilykorttien suosio jatkaa kasvuaan ja ne jatkavat yhä kehittymistään. Magic the Gathering -pelisarja saa myös säännöllisesti lisää sisältöä. Samaan aikaan opinnäytetyössä toteutetun sovelluksen kaltaisien työkalujen tarve kasvaa. Keräilykorttisarjojen tuomiin muutoksiin sekä lisäyksiin sopeutuvat sovellukset tulevat tulevaisuudessakin pärjäämään pitkälle.

Lähteet

Anad, B. 2023. Understanding Data Binding in Angular. Knowledgehut Solutions. Viitattu 20.2.2023. Saatavissa https://www.knowledgehut.com/blog/web-development/data-binding-in-angular

Angular. 2022. Introduction to Angular concepts. Viitattu 1.2.2023. Saatavissa https://angular.io/guide/architecture

Collectibles Insurance Services. 2021. The History of Collectible Trading Cards. Viitattu 30.3.2023. Saatavissa https://collectinsure.com/2021/07/09/the-history-of-collectible-trading-cards/

Davis, J. 2021. How many total Magic: the Gathering cards have ever been printed? Medium. Viitattu 30.3.2023. Saatavissa https://medium.com/@ErrorJustin/how-many-total-magic-the-gathering-cards-have-ever-been-printed-a8eb4328575b

Dev Community. 2022. Spring Boot Architecture. Viitattu 20.2.2023. Saatavissa https://dev.to/maddy/spring-boot-architecture-547i

Docker Inc. 2023. Docker overview. Viitattu 14.2.2023. Saatavissa https://docs.docker.com/get-started/overview/

Fossa Inc. 2023. The Developer's Guide to Open Source Software Licenses. Viitattu 3.4.2023. Saatavissa https://fossa.com/developers-guide-open-source-software-licenses

GeeksforGeeks. 2021. What is MongoDB – Working and Features. Viitattu: 2.2.2023. Saatavissa https://www.geeksforgeeks.org/what-is-mongodb-working-and-features/

Gillis, A., Botelho, B. 2023. Definition MongoDB. TechTarget. Viitattu 19.4.2023. Saatavissa https://www.techtarget.com/searchdatamanagement/definition/MongoDB

Github. 2023. No Licence. Viitattu 3.4.2023. Saatavissa https://choosealicense.com/no-permission/

Horcasitas, J. 2021. Understanding Open-Source Software Licenses. Digital Ocean. Viitattu 3.4.2023. Saatavissa https://www.digitalocean.com/community/tutorials/understanding-open-source-software-licenses

IBM. 2023a. What is Docker? Viitattu 14.2.2023. Saatavissa https://www.ibm.com/topics/docker

IBM. 2023b. What is Java Spring Boot? Viitattu 2.2.2023. Saatavissa https://www.ibm.com/topics/java-spring-boot

JWT. 2023. Introduction to JSON Web Tokens. Okta. Viitattu 6.2.2023. Saatavissa https://jwt.io/introduction

WMware. 2023. Accessing Data with MongoDB. Viitattu 6.2.2023. Saatavissa https://spring.io/guides/gs/accessing-data-mongodb/

Saha, D. 2022. What is MongoDB and Why to use it? Dot Net Tricks Innovation. Viitattu 27.2.2023. Saatavissa https://www.dotnettricks.com/learn/mongodb/what-is-mongodb-and-why-to-use-it

Scryfall, LLC. 2023. API Documentation. Viitattu 7.2.2023. Saatavissa https://scryfall.com/docs/api

Sharma, P. 2019. Understanding Angular Guards. Codeburst. Viitattu. 16.3.2023. Saatavissa https://codeburst.io/understanding-angular-guards-347b452e1892

Snyk Limited. 2023. GNU General Public License: GPLv3 explained. Viitattu 3.4.2023. Saatavissa https://snyk.io/learn/what-is-gpl-license-gplv3-explained/

Tutorialspoint. 2022. Spring Boot – Introduction. Viitattu 2.2.2023. Saatavissa https://www.tutorialspoint.com/spring_boot/spring_boot_introduction.htm