UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Thesis Matcher

Aplicație de repartizare a studenților la profesori coordonatori de licență

propusă de

Andrei Dascălu

Sesiunea: februarie, 2023

Coordonator științific

Lect. Dr. Frăsinaru Cristian

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

Thesis Matcher

Andrei Dascălu

Sesiunea: februarie, 2023

Coordonator științific

Lect. Dr. Frăsinaru Cristian

Avizat,

Îndrumător lucrare de licență,

Lect. Dr. Frăsinaru Cristian.

Data: ..... Semnătura: .....

Declarație privind originalitatea conținutului lucrării de licență

Subsemnatul **Dascălu Andrei** domiciliat în **România, jud. Iași, mun. Iași, strada Sf. Teodor nr. 14**, născut la data de **03 noiembrie 2000**, identificat prin CNP **5001103226752**, absolvent al Universității "Alexandru-Ioan Cuza" din Iași, **Facultatea de informatică** specializarea **informatică**, promoția 2022, declar pe propria răspundere cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art. 143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul **Thesis Matcher** elaborată sub îndrumarea domnului **Lect. Dr. Frăsinaru Cristian**, pe care urmează să o susțin în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului ei într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diplomă sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data: ..... Semnătura: .....

Declarație de consimțământ

Prin prezenta declar că sunt de acord ca lucrarea de licență cu titlul **Thesis Matcher**, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test, etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de informatică.

De asemenea, sunt de acord ca Facultatea de informatică de la Universitatea ”Alexandru-Ioan Cuza” din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent **Andrei Dascălu**

Data: ..... Semnătura: .....

Cuprins

Motivație

Lucrarea de licență reprezintă culminarea anilor de facultate, o dovadă a studentului de deprindere a unor noțiuni, de specializare într-un anumit domeniu și de capabilitatea a acestuia de a contribui cu soluții proprii la problemele curente sau viitoare ale societății. În consecință, alegerea unei tematici adecvate pentru lucrarea de licență este un pas de bază datorită unor motive pertinente. În primul rând, abordarea unei tematici cât mai aproape de domeniile sau subiectele de interes ale studentului rezultă într-o atenție mai mare și o implicare corespunzătoare ale acestuia în realizarea în sine a lucrării. În al doilea rând, există o legătură directă între eficiența și temeinicia realizări lucrării de licență și colaborarea dintre profesorul coordonator și student, fiind necesară o comunicare constantă și productivă prin feedback-ul în ambele sensuri, dar și prin resurse, materiale sau idei.

Profesorii coordonatori au un număr de locuri limitat, prin urmare este necesară o repartizare eficientă și mai ales optimă a studenților în funcție de preferințe, luând în același timp în calcul și punctele de vedere ale profesorilor. De asemenea, studenții ar trebui să aibă posibilitatea să cunoască dinainte tematicile propuse de profesori și eventualele condiții prealabile de realizare a respectivelor teme. O aplicație centralizată este așadar întru totul necesară optimizării și îmbunătățirii în general ale acestui proces organizatoric din cadrul facultății.

Introducere

0.0.1 Scopul documentului

Documentul de față are ca scop prezentarea problemei și a unei soluții propuse adecvate acesteia. În acest fel, se urmărește descompunerea problemei în subprobleme și detalierea acestora, ilustrarea unor soluții deja existente, analizarea aplicației web propuse, în mod sistematic, evidențiind atât detalii de abordare, arhitectură, tehnologii utilizate, implementare.

0.0.2 Scopul aplicației

Aplicația *Thesis Matcher* este o soluție web ce urmărește să rezolve problema alocării studenților din anii terminali la profesorii coordonatori.

Unul dintre principalele obiective este centralizarea întregului proces, de la repartizare până la alte colaborare și alte aspecte organizatorice, asigurând astfel o desfășurare bună și mai sigură. Fiind un proces anual, cu un număr semnificativ de părți implicate, este inevitabilă necesitatea de a automatiza într-o anumită măsură desfășurarea acestuia.

Un obiectiv de asemenea important este simplificarea obținerii informaților de interes, atât de către studenți, cât și de profesori. Studenții au posibilitatea să afle tematicile propuse de fiecare coordonator, eventual să propună o idee proprie. Participanții pot urmări în timp real situația locurilor disponibile, precum și alte statistici.

Cel de al treilea obiectiv este optimizarea repartizării prin implementarea ierarhizării de către student a opțiunilor alese sub formă de preferințe.

0.0.3 Modul de implementare

Soluția se prezintă sub formă de aplicație web ce se împarte în două ”sub-aplicații”.

Partea de *Front-End* este implementată în Angular. Partea de *Back-End* este implementată în Java Spring Boot. Această parte cuprinde atât comunicarea cu Front-End-ul prin intermediul request-urilor, dar și procesare sub forma implementării unui algoritm de licitație cu *double-push* pentru rezolvarea unei probleme de *asymmetric assignment*. Acest lucru presupune o optimizare a soluției deoarece permite părților să efectueze alegeri cu același grad de importanță. Mai precis, în contextul de față, un student poate realiza o ordine a preferințelor profesorilor, în unele cazuri cu o anumită indiferență în ideea că studentul poate prefera atât o opțiune, cât și alta cu același grad.

Capitolul 1

Descrierea problemei

Lucrarea de față propune să rezolve o instanță a unei probleme des întâlnite în cotidian și descrise în literatura de specialitate, problema *Asignării optime*. În cazul de față, există două mulțimi participante. În primiul rând profesorii, mai precis propunerile individuale ale acestora pentru proiectul final de licență, și studenții. Fiecare student stabilește o ierarhie a preferințelor (unde sunt ordonate propunerile alese de ei; precizări și detalieri ale acestora urmează a fi făcute).

De asemenea, profesorii au opțiunea de a încheia un acord cu un student sau mai mulți pentru realizarea unui anumit proiect fără a participa la etapa de stabilire a unei repartizări pe cele două mulțimi.

Într-un final, după efectuarea algoritmului, fiecărui student îi este atribuit în mod optim, cu alte cuvinte cu o satisfacere cât mai mare a preferințelor, un proiect pentru lucrarea sa de licență.

1.1 Scopul aplicației

În consecință, aplicația propusă are atât scopul de a eficientiza propcesul de repartizare a studenților la profesorii de licență (respectiv tezele propuse de aceștia), un proces complex și periodic specific oricărei facultăți, dar și scopul de a optimiza această repartiție, în ideea de a distribui proiectele studenților după abilitățile și preferințele lor, încurajând astfel o dedicare și elaborare cât mai adecvate din partea studenților în realizarea lucrării finale.

1.2 Scopul documentului

Documentul prezent descrie o soluție a problemei evidențiate anterior sub forma unei aplicații web ce utilizează o formă a algoritmului de *licitație* (*Auction algorithm*) adaptat contextului, ce utilizează o strategie *double-push* pentru pasul de licitare. Sunt ilustrate părțile componente ale aplicației și justificată alegerea anumitor implementări și tehnologii. De asemenea, acest document urmărește și familiarizarea eventualilor utilizatori cu aplicația.

1.3 Structura documentului

Pentru început, a fost realizată o descriere generală a aplicației subliniind câteva particularități în al doilea capitol, **Arhitectura aplicației. Detalii**. Capitolul ulterior, **Tehnologii utilizate**, explică alegerea anumitor framework-uri precum și avantajele acestora (un scurt istoric a fost realizate pentru familiarizarea cu aceste tehnologii). Urmează patru capitole care surprind în detaliu părțile componente ale aplicației, **Baza de date**, **Front-end-ul**, **Back-end-ul**, **Algoritmului de repartizare**. În final, capitolul **Scenarii de utilizare** prezintă sugestii în legătură cu navigarea și folosirea aplicației, nu fără a sublinia în cele din urmă câteva **Concluzii** și posibile idei de dezvoltare și îmbunătățire a aplicației în viitor.

1.4 Contribuții

Aplicația **Thesis Matcher** este compusă după cum a fost precizat din două părți principale.

Front-end-ul realizat în Angular a fost adaptat pentru o cât mai facilă utilizare atât de către profesori, cât și de către studenți, urmărindu-se un aspect cât de cât minimalist și clar.

Back-end-ul în Spring Boot conține o parte de autentificare și autorizare simplă, dar eficientă și suficientă prin intermediul JWT (JSON Web Token).

O contribuție majoră o reprezintă însă algoritmul de determinare a soluției, componentă tot a back-end-ului. În cazul de față, a fost utilizat un algoritm de licitație (*Auction algorithm*) pentru o problemă de asignare asimetrică în ideea că cele două mulțimi participante cel mai probabil nu sunt de aceeași dimensiune.

Capitolul 2

Arhitectura aplicației. Detalii

În primul rând, **Thesis Matcher** este o aplicație de tipul utilitar, scopul acesteia este unul de uz intern, în cadrul facultății. Prin urmare, traficul efectuat de către utilizatori este unul scăzut, lucru ce permite o arhitectură monolitică.

În al doilea rând, arhitectura urmează un model *MVC* (Model View Controller) caracteristic unei aplicații web de acest tip, unde clientul trimite anumite request-uri prin interacțiunea cu GUI-ul, iar serverul la rândul său transmite anumite informații. Acest model arhitectural permite o organizare și modularizare riguroasă a aplicației [11].

În al treilea rând, baza de date este una de tipul SQL deoarece este necesară stocarea unui număr relativ mare de relații între entități, reprezentate de către preferințele studenților. De aceea, a fost ales MySQL pentru modelarea relațiilor și datelor din aplicație.

Astfel, aplicația este pe trei niveluri (*3-tier architecture*). Un nivel al clientului, *Presentation Layer* ce facilitează utilizarea, un nivel de business, *Application Layer*, ce tratează request-uri, efectuează operații de autentificare și autorizare, precum și algoritmice, iar în final un nivel de prelucrare al datelor, *Data Layer*.

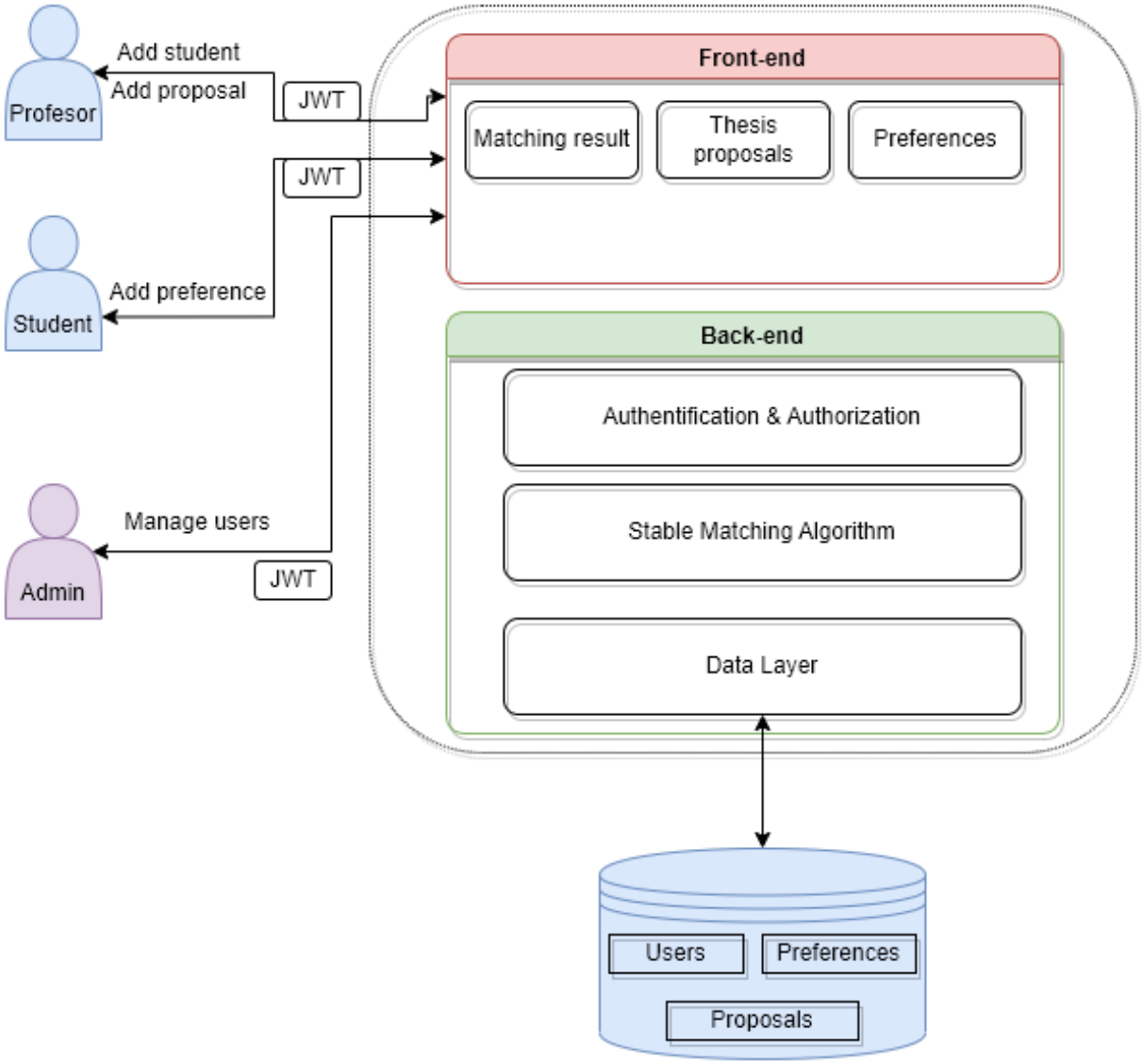

Figura 2.1: Arhitectura aplicației

Capitolul 3

Tehnologii utilizate

În acest capitol sunt prezentate în primul rând tehnologiile utilizate în implementarea aplicației.

3.1 Angular

Angular este un framework de JavaScript scris în TypeScript și menținut de Google. Framework-ul a fost dezvoltat în principal pentru crearea aplicațiilor web *single-page*, într-o manieră ce ușurează mentenanța și dezvoltarea ulterioară.

3.1.1 Scurt istoric

În 2010, Miško Hevery, un angajat la Google la acel timp, a lansat un proiect cu numele *AngularJS* care a fost apreciat în mare măsură de comunitate. Între anii 2014-2015 a avut loc o reîmprospătare majoră a framework-ului însemnând de fapt o rescriere majoră a acestuia. Noua versiune avea să fie numită simplu Angular. Au urmat câțiva ani de tranziție deoarece multe proiecte deja în producție erau utilizau AngularJS și trebuiau refactorizate. În acest moment, Angular este cel mai folosit framework de front-end, în specialde dezvoltatorii de la Google și de către start-up-uri. Exemple de companii recunoscute ce folosesc Angular sunt Microsoft, Gmail, PayPal, Forbes [10].

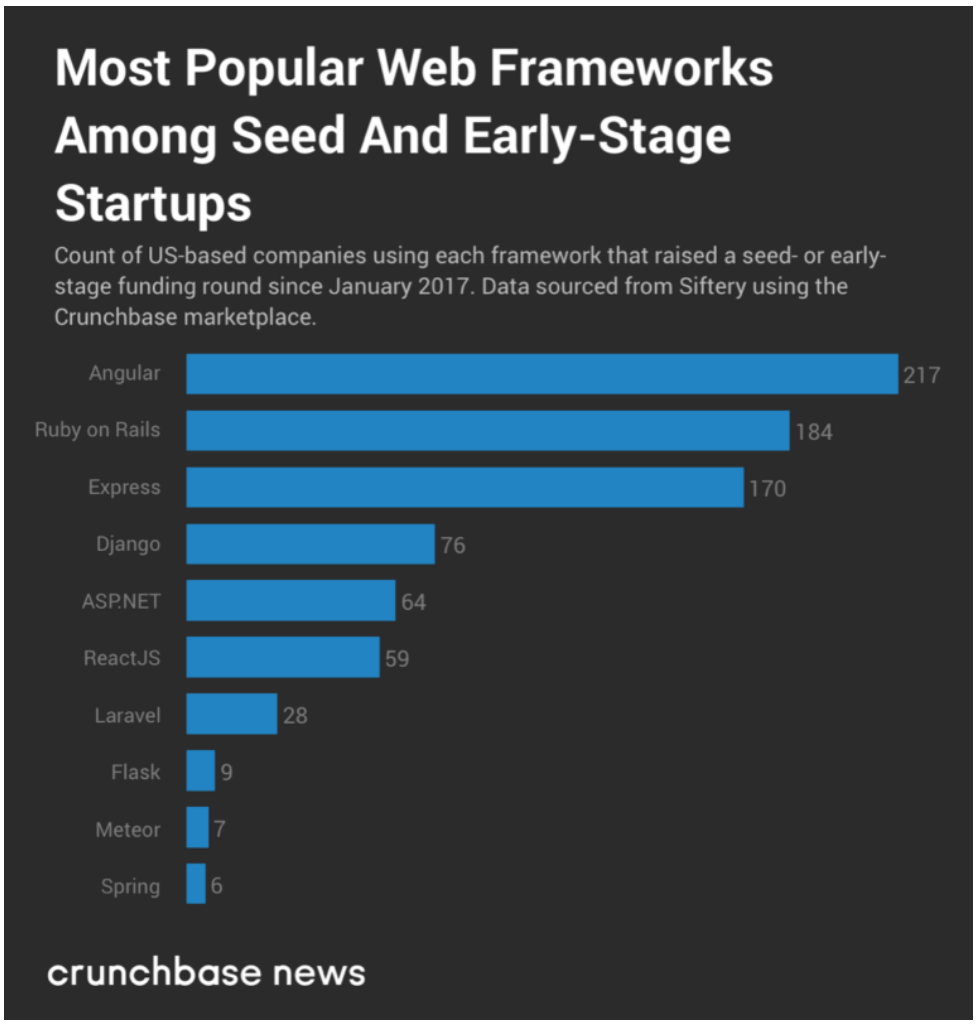

Figura 3.1: [12]

3.1.2 Particularități

Motivul alegerii Angular pe partea de front-end este reprezentat de o serie de caracteristici.

Un astfel de motiv este *arhitectura bazată pe componente*, fiind astfel o formă de programare orientată pe obiect. Utilizatorul crează în mod uzual clase corespunzătoare componentelor ce conțin și un șablon HTML (*eng.* HTML template). Pentru simplificare, Angular oferă și opțiunea de ”injectare” a serviciilor *custom* sau *in-built* într-o componentă ce utilizează aceste funcționalități. În acest fel, utilizatorul poate reutiliza, înlocui, modifica componente în diverse locuri, obținându-se astfel un UI (User Interface) modularizat.

Un alt motiv este modul de încărcare a paginii web. Angular folosește *lazy loading* ce permite încărcarea instantanee a website-urilor, prin afișarea doar a componentelor cerute și necesare utilizatorului, în timp ce celelalte sunt pregătite în fundal pentru alte

eventualități.

Dependency injection reprezintă un al treilea motiv, un design pattern ce permite împărțirea lucrului între diferite servicii, distribuind în mod eficient sarcinile. Prin inițializarea dependențelor, Angular reușește să reducă în mod considerabil codul de tip *boilerplate* (fragmente similare de cod des utilizat între care există mici diferențe) și să extindă mai ușor o astfel de aplicație.

Framework-ul are trei tipuri de *dependency injections*:

1. Constructor injection
2. Setter injection
3. Interface injection

Din punct de vedere al arhitecturii, Angular poate fi considerat în general un framework MVVM (Model-View-ViewModel), după cum se observă în figura următoare.

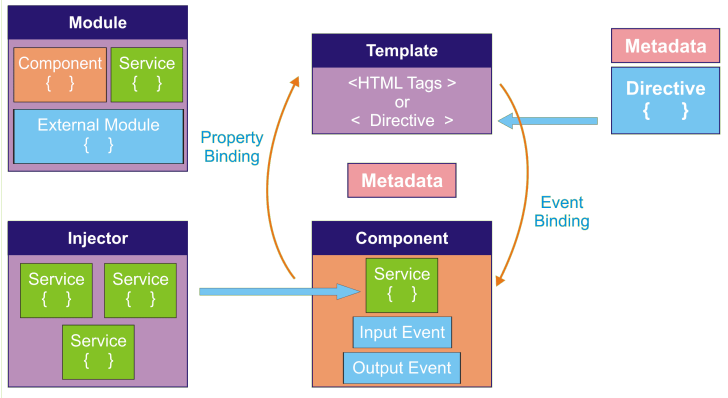

Figura 3.2: Arhitectura Angular [1]

3.2 Spring Boot

Spring Boot este un micro-framework open-source folosit pentru a crea aplicații Spring cu microservicii. Spre deosebire de alte framework-uri de Java, acesta oferă configurări XML flexibile, procesare în loturi puternică, tranzacții cu baza de date și o varietate de instrumente de dezvoltare.

3.2.1 Scurt istoric

Framework-ul *Spring* a fost creat în 2004 pentru a simplifica dezvoltarea programelor pe partea de server. În aprilie 2014 a fost lansat Spring boot 1.0.0 în urma unor cereri din partea programatorilor de a configura serviciile de web container într-un container spring din metoda principală. În decembrie 2016 a fost lansat Spring Boot 1.3 ulterior trecerii framework-ului Spring de la versiunea 4.1 la 4.2 și includea sprinjin pentru fisiere JAR complet executabile, noi utilitare spring-boot-dev și auto-configurare pentru tehnologii de caching.

3.2.2 Motivație

Spring Boot este bazat pe Java, unul dintre cele mai populare limbaje de programare. Framework-ul are o comunitate vastă de utilizatori cu diverse materiale și cursuri.

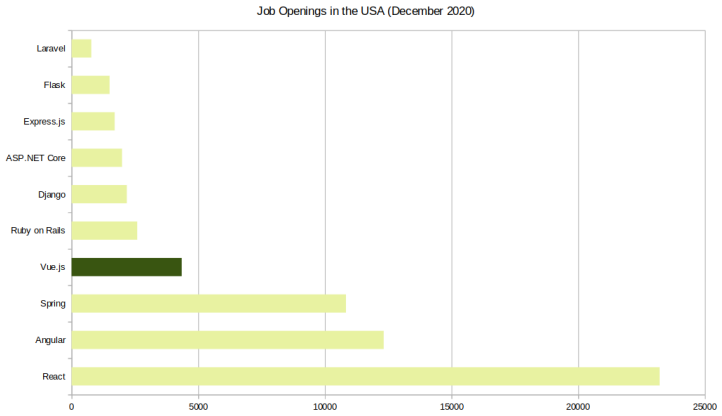

Figura 3.3: [2]

Avantajele prinicipale sunt după cum urmează. Spring Boot este *multi-threaded*, util pentru operații repetitive și de durată. Facilitează crearea și testarea aplicațiilor Java oferind un setup default pentru unit și integration testing. Există de asemenea posibilitatea de a integra Spring Boot cu ecosistemul Spring ce include Spring Data, Spring Security, Spring ORM și Spring JDBC într-un mod simplificat.

3.2.3 Particularități

Principala particularitate a Spring Boot o reprezintă adnotările (*Spring Boot annotations*) utilizate în auto configurare. De exemplu, **@SpringBootApplication** marchează metoda principală a aplicației și este obligatorie. **@EnableAutoConfiguration**

oferă oricărei clase pe care o marchează cu opțiunea de Automatic Configuration. **@ComponentScan** scanează la inițializare toate declarările de *beans* și pachete.

Un alt detaliu este utilizarea *Spring Starter Dependencies* ce facilitează gestionarea dependențelor unei aplicații în continuă dezvoltare.

Spring Boot Actuator oferă utilitare de producție aplicației. Actuator este utilizat în mare parte pentru a obține informații de funcționare despre o aplicație în rulare (metrice, info, dump, env, etc.), cu ajutorul HTTp endpoints și JMX beans. Ultima versiune, Spring Boot 2.x Actuator, suportă și modele CRUD.

3.2.4 Detalii

Versiunea de Java este 11.

Versiunea de Spring Boot este 2.7.6.

3.3 MySQL

MySQL este cea mai populară bază de date open-source, fiind o soluție adoptată de aplicații des folosite precum Facebook, Twitter, Netflix [24]. Este un sistem de gestionare a bazelor de date relaționale (RDMS) de tipul client/server care include un server SQL cu multiple fire de execuție (multi-threaded), diverse librării și API-uri.

3.3.1 Avantaje

MySQL este tehnologia aleasă pentru stocarea datelor datorită fiabilității și scalabilității sale, precum și funcționalităților variate. De asemenea, integrarea cu alte tehnologii precum Spring Boot are loc ușor datorită librăriilor și soluțiilor unei comunități extinse de utilizatori.

A fost aleasă o bază de date relațională pentru a asigura consistența datelor, luând în calcul numărul mare de preferințe al studenților ce trebuie procesate.

Capitolul 4

Baza de date

În acest capitol este argumentată alegerea unui tip de baze de date SQL, mai exact MySQL, pentru a reține informații importante aplicației. A fost preferată o abordare clasică asupra unei baze de date relaționale în special datorită numărului mare de tranzacții ce necesită o asigurare mai mare a integrității datelor [25].

Aplicația Thesis Matchereste de tip utilitar, aceasta are scopul de a fi folosită într-un context restrâns, în cadrul facultății. Prin urmare numărul de utilizatori este unul relativ mic, reprezentat de către studenți și profesori. De asemenea, poate fi observată o relație strânsă între profesori și propunerile acestora, dar și între studenți și preferințele acestora.

Relațiile între entități sunt de mai multe tipuri, în special *One-to-one* și *One-to-many*. Acest lucru se poate observa în diagrama următoare a bazei de date utilizate.

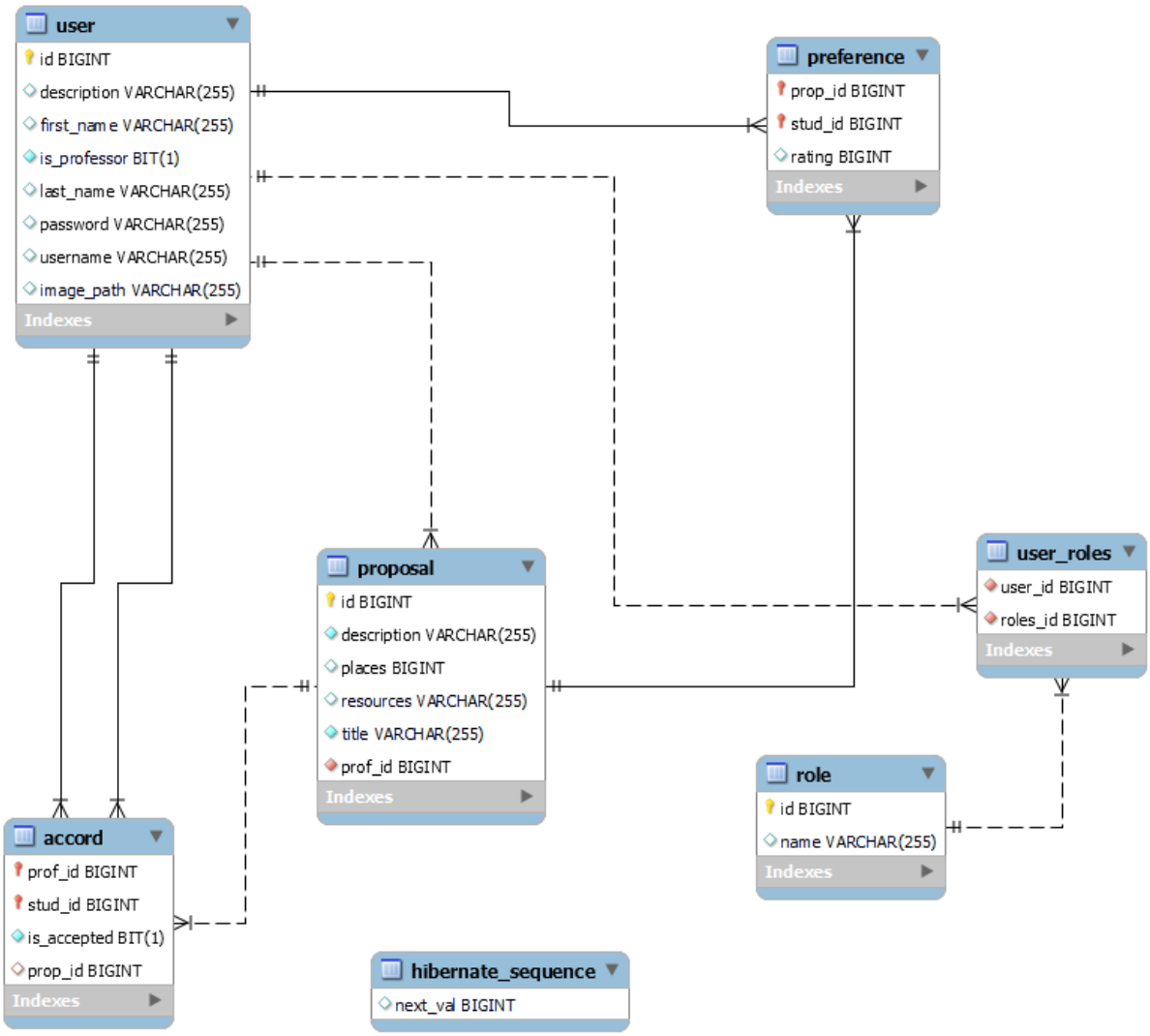

Figura 4.1: Baza de date

În următoarele subcapitole sunt detaliate modelele utilizate pentru a reprezenta aceste date.

4.1 Modelarea utilizatorilor

Utilizatorii au fost modelați într-un mod clasic. Aceștia au la rândul lor roluri în funcție de drepturile pe care le au ei în cadrul aplicației. Astfel, sunt identificate două astfel de roluri: **ROLE_USER** și **ROLE_ADMIN**. Pe lângă acest lucru, utilizatorii sunt ori profesori, ori studenți, particularitate ilustrată de câmpul boolean *is_professor* al entității **`user`**.

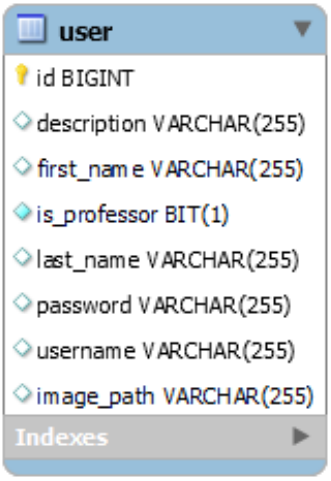

Figura 4.2: User

Fiecare utilizator este identificat unic printr-un `id` de tipul de date BIGINT, care este și cheie primară a entității. Ca în majoritatea cazurilor, aceștia au un nume de familie (`last_name`) și un prenume (`first_name`). Câmpul `username` este de tipul VARCHAR(255) și este reprezentat de un email valid, lucru asigurată pe partea de front-end. Câmpul `password` este tot de tipul VARCHAR(255) și reprezintă parola utilizatorului, însă criptată pe partea de back-end cu un algoritm clasic. Aceste două câmpuri sunt primite de către utilizator din partea unui administrator și sunt utilizate pentru autentificarea în aplicație. A fost preferată această abordare pentru a împiedica crearea de conturi din partea unor persoane din afara contextului.

4.1.1 ROLE_USER

Cei mai mulți utilizatori au rolul **ROLE_USER**, un rol default. Acest lucru le autorizează accesul la aplicație, odată ce aceștia sunt autentificați. Interfața diferă în funcție de tipul utilizatorului.

Tipuri

Profesorii pot crea și actualiza propuneri (proposals) pentru teza de licență. Aceste propuneri pot fi *project* sau *topic*, detalierea acestora urmând a fi făcută ulterior. De asemenea, aceștia pot încheia acorduri (accords) cu anumiți studenți pentru unele dintre propunerile lor. Acest lucru le permite studenților să nu mai participe la algoritmul de asignare, fiind deja repartizați profesorilor respectivi.

Studenții pot crea și actualiza preferințe (preferences) într-o anumită ierarhie a lor. Așadar, categorisirea utilizatorilor în profesori și studenți are unicul scop de a diferenția interfața în funcție de funcționalitățile specifice tipurilor acestora.

4.1.2 ROLE_ADMIN

Există în plus un număr restrâns de utilizatori care au rolul de administrator, **ROLE_ADMIN**. Aceștia au dreptul de a gestiona conturile participanților, mai precis de a crea sau elimina utilizatori.

4.2 Preferințele

Preferințele sunt modelate simplu, reprezentând în sine o relație între entitățile studenți și propuneri.

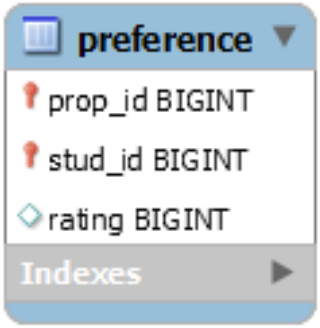

Figura 4.3: Preference

După cum se poate observa în tabela **`preference`**, acestea cuprind un `id_stud` ce face legătura cu tabela **`student`** și un `id_prop` ce face legătura cu tabela **`proposals`**.

În plus, există un câmp `rating` care este de tipul BIGINT și indică proporțional importanța preferinței, mai exact o preferință cu un `rating` $= 100$ este pe primul loc în ierarhie. Este de asemenea important de precizat că mai multe preferințe ale unui student pot avea același rating, lucru ce permite o mai bună satisfiabilitate.

Aceste relații între studenți și tezele propuse vor constitui datele de intrare pentru algoritm, fiind astfel de o importanță semnificativă în cadrul aplicației.

4.3 Propunerile

În cazul propunerilor, la nivel teoretic acestea pot fi de două tipuri, proiect (project) sau temă (topic). Diferența constă în faptul că un proiect este o propunere mai concisă, cu o descriere exactă a lucrării așteptate, fie ea una de cercetare sau o aplicație propriu-zisă, iar un proiect nu poate fi ales și realizat decât de către o singură persoană. Un astfel de exemplu ar putea fi *„Realizare unei aplicații de reconoaștere facială utilizând metode de machine learning"*, cu precizările și restricțiile aferente.

Pe de altă parte, o temă este o descriere pe larg a anumitor particularități ale unui proiect de cele mai multe ori dintr-un anumit domeniu de specialitate. La fel ca un proiect specific, o temă poate cuprinde atât lucrări general teoretice, cât și practice, studentul fiind nevoit să stabilească împreună cu profesorul său coordonator o teză concretă. Însă o propunere de acest fel poate avea un număr de locuri disponibile mai mare de 1, deoarece mai mulți studenți pot urma direcții diferite asupra tematicii propuse. Spre exemplu, profesorul poate propune o temă de „Algoritmi genetici și optimizarea acestora" care poate fi repartizată unor 3 studenți care stabilesc în mod distinct cu profesorul de licență lucrările lor.

Cu toate acestea, din motive de simplificare și optimizare, la nivelul de schemei de baze de date, a preferată modelarea propunerilor într-o singură tabelă intitulată sugestiv **`proposal`**, diferența dintre cele două tipuri fiind făcută de câmpul `places`.

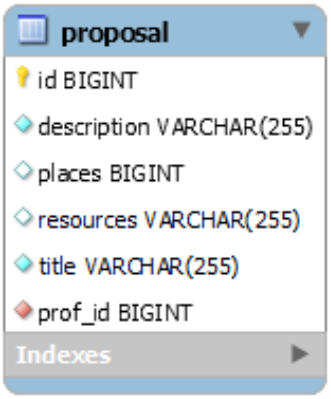

Figura 4.4: Proposal

În cazul proiectelor (projects) câmpul acesta rămâne `null`, spre deosebire de teme (topics). Fiecare propunere are un titlu (`title`) reprezentativ și o descriere (`description`) adecvată de tipul VARCHAR(255), precum și o listă de referințe pentru a ajuta studentul. Ultimul câmp este `id_prof` care indică autorul propunerii.

4.4 Acordurile

În cadrul aplicației Thesis Matcher, fiecare profesor poate stabili împreună cu un student un acord, reprezentat de tabela cu același nume **`accord`**, în ideea că acest student și propunerea aleasă nu mai sunt luate în calcul la rularea algoritmului de repartizare. Cu alte cuvinte, acest detaliu le permite studenților posibilitatea să realizeze un proiect preferat, adecvat abilităților și înclinațiilor sale, desigur cu aprobarea profesorului corespunzător tezei alese.

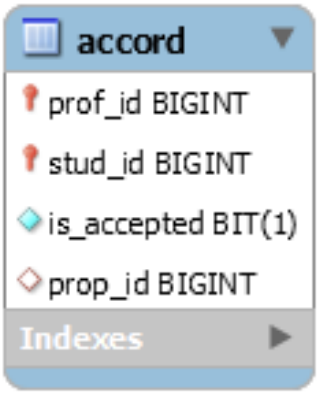

Figura 4.5: Accord

4.5 Lucrul pe partea de back-end cu baza de date

Partea de back-end a aplicației, fiind implementată cu ajutorul Spring Boot, interacționează cu baza de date prin intermediul Hibernate, alegere făcută luând în considerare o serie de avantaje.

4.5.1 Hibernate

Figura 4.6: `https://hibernate.org/images/hibernate-logo.svg`

Ce este Hibernate

Hibernate este un framework, mai precis un ORM (object relational mapping) creat pentru a facilita maparea modelelor orientate-obiect la baze de date relaționale

pentru aplicații web [23]. În mod intern, acest ORM utilizează JDBC API pentru a interacționa cu baza de date [16].

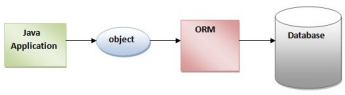

Figura 4.7: ORM

Hibernate este cu alte cuvinte o implementare a specificațiilor JPA (Java Persistence API) pentru persistența datelor. JPA este un set de reguli care oferă un standard definit și o anumită funcționalitate acestor tehnologii ORM [16].

Funcționalitatea Hibernate

Framework-ul precizat corelează clase din limbajul Java tabelelor din serverul de baze de date, dar și tipurile de date din Java cu cele din SQL pentru a asigura persistența obiectelor. Hibernate returnează astfel aplicației enitățile sub formă de obiecte Java și reduce astfel timpul de procesare a rezultatelor și codul necesar implementat de programator.

Avantaje ale Hibernate

Hibernate este în primul rând un framework *open source* și *lightweight*, lucru ce a permis familiarizare cu acesta și ușoara configurare a lui.

În al doilea rând, Hibernate este un framework relativ rapid deoarece beneficiază de un cache intern, structurat pe două niveluri, primul nivel fiind folosit în mod normal [16].

De asemenea, tehnologia optată utilizează o versiune orientată obiect a SQL-ului, HQL (Hibernate Query Language), care permite generează query-uri independente de baza de date, fără a scrie în mod explicit query-uri în SQL [16].

În final, Hibernate oferă posibilitatea de a crea și popula tabelele din baza de date în mod automat, direct din cod.

Imaginea următoare ilustrează specificațiile din fișierul `application.properties` specific oricărui proiect Maven, în care este configurată utilizarea framework-ului Hibernate. Acest lucru include date de conectare la baza de date, modul de conectare, dialectul SQL folosit etc.

```
spring.datasource.url=jdbc:mysql://localhost:3306/thesis_matcher?useSSL=false&serverTimezone=UTC
spring.datasource.username=[redacted]
spring.datasource.password=[redacted]
spring.datasource.platform=mysql
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.properties.hibernate.format_sql=true
```

Figura 4.8: application.properties

Pentru a putea folosi Hibernate într-un proiect de Spring Boot este necesară adăugarea dependenței următoare în fișierul `pom.xml`.

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

Figura 4.9: Dependența din pom.xml

4.6 Generarea datelor

În vederea generării unor date reprezentative problemei actuale, a fost creată o clasă separată numită `Populator` ce introduce înregistrări (records) în baza de date. De aceea au fost stabilite în primul rând o serie de parametri care reglează dimensiunile populațiilor și proporțiile dintre diferitele tipuri de entități. În continuare vor fi prezentați pe scurt acești parametri și configurarea lor, urmând a descrie maniera de generare propriu-zisă a utilizatorilor, propunerilor profesorilor și nu în ultimul rând, preferințelor studenților.

Configurarea parametrilor necesari generării datelor este realizată prin intermediul unei clase `PopulatingConfiguration`.

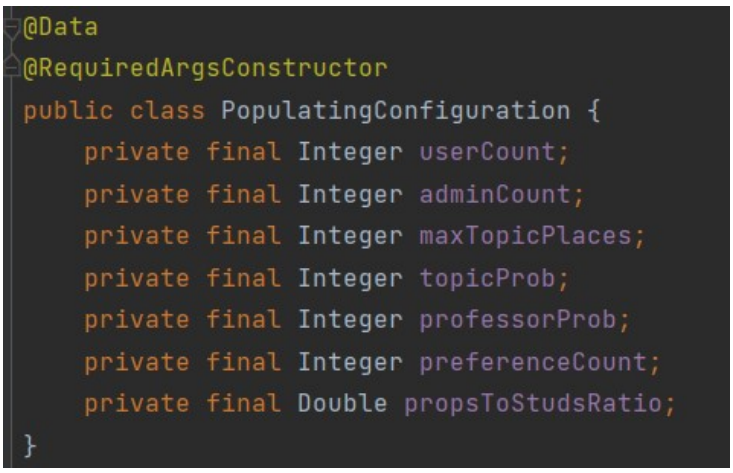

Figura 4.10: Clasa de parametri de configurare

Aceasta conține parametri precum numărul de utilizatori (`userCount`), numărul de administratori (`adminCount`), probabilitatea ca un utilizator să fie profesor (`professorProb`) etc.

Primul pas este generarea utilizatorilor. Pentru această etapă a fost folosită clasa `Faker` în obținerea unor nume aleatorii. Parola tuturor a fost setată aceeași pentru simplificare, iar primii `adminCount` utilizatori primesc `ROLE_ADMIN` pe lângă `ROLE_USER`.

Cel de al doilea pas este generarea propunerilor pentru fiecare profesor ales dintre utilizatori conform unei probabilități. Se calculează întâi un număr de locuri în corelație cu numărul de studenți ce trebuie acoperit de fiecare profesor pentru a putea da șansa unei repartizări a tuturor studenților. De menționat că tipul propunerilor este tot ales aleatoriu.

Ultimul pas este crearea preferințelor și a acordurilor utilizatorilor.

În cazul de față, numărul total de utilizatori a fost ales **500**, cu o probabilitate de **10%** ca un utilizator să fie profesor. Fiecare student are **20** de preferințe, iar numărul total de propuneri este într-un raport de **1.2** cu numărul total de studenți.

Capitolul 5

Front-end

Pentru partea de front-end a aplicației a fost ales framework-ul de Typescript Angular datorită unor mai multe avantaje. O particularitate importantă și principală a acestui framework este faptul că utilizează componente pentru crearea unei aplicații *single-page*.

Astfel, posibilitatea de creare a componentelor permite reutilizarea acestora și economisirea codului, aplicația devenind una orientată-obiect datorită și faptului că aceste componente sunt implementate în Typescript, un limbaj *strongly typed* dezvoltat din Javascript.

Nu în ultimul rând, familiarizarea cu această tehnologie are loc relativ repede, iar documentația vastă alături de diversele cursuri și tutoriale permit o învățare adecvată și dezvoltarea unor soluții optime.

5.1 Interfața

Interfața este adaptată în funcție de tipul de utilizator autentificat. Există elemente comune în general, dar și anumite funcționalități specifice.

5.1.1 Pagina principală

Prima pagină a aplicației Thesis Matcher este cea de **Matching**, vizibilă oricărui utilizator, indiferent de tipul său sau dacă este autentificat sau nu.

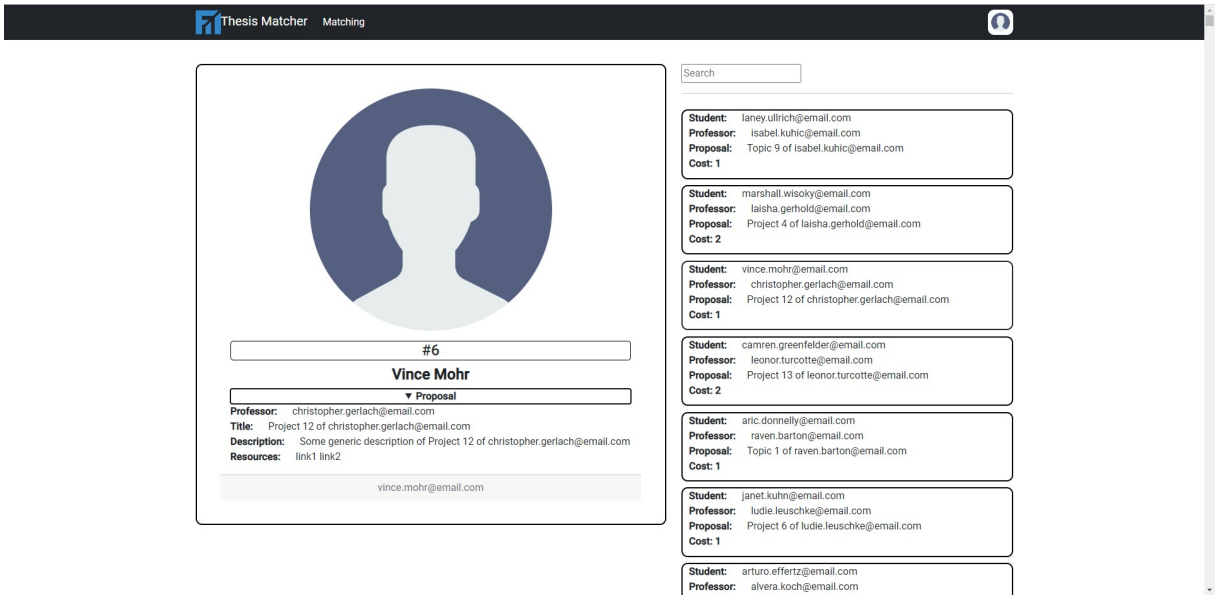

Figura 5.1: Matching (repartiția)

Aici se poate observa lista tuturor studenților și propunerea (proiectul sau tematica) asignată fiecăruia în funcție de preferințele sale. De asemenea, pentru fiecare asignare este înregistrat un *cost* care indică locul propunerii primite în ordinea preferințelor sale (de exemplu, o asignare cu costul 1 înseamnă că propunerea a fost pe primul loc în lista sa, o alta cu costul 2 a fost pe al doilea loc etc.). În plus, există o opțiune de a căuta prin rezultat după numele studentului, al profesorului sau titlul lucrării.

Totuși, opțiunile utilizatorului sunt mult restrânse, lucru ce se poate observa și din bara de navigare, el fiind nevoit să se autentifice.

Pagina de autentificare este simplă, utilizatorul fiind nevoit să introducă email-ul și parola unui cont existent după cum se poate obeserva în figura următoare.

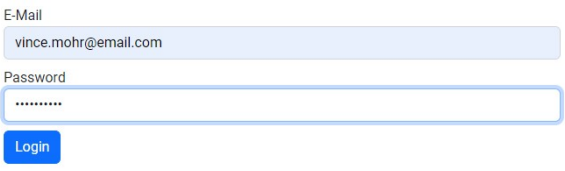

Figura 5.2: Pagina de autentificare

Odată autentificat, utilizatorul își poate schimba fotografia de profil, descrierea în care poate specifica domeniile de interes. În plus, acesta își poate schimba parola de la cont. Toate aceste setări pot fi efectuate din pagina **My Profile**.

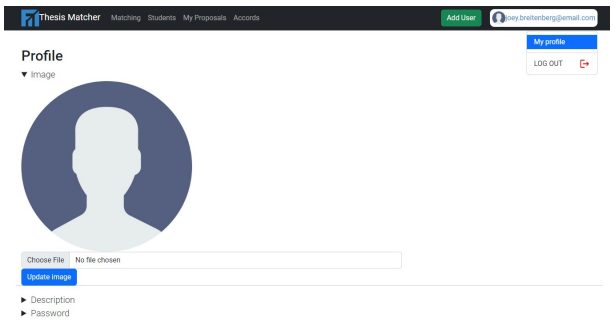

Figura 5.3: Profilul utilizatorului

5.1.2 Interfața studentului

Un utilizator de tip *student* poate accesa pagina **Professors** unde poate vedea detalii despre fiecare profesor, însă principalul scop al acestei pagini este de a accesa propunerile fiecărui profesor în parte.

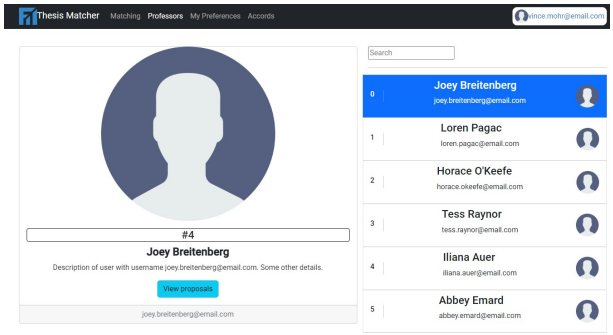

Figura 5.4: Profesorii

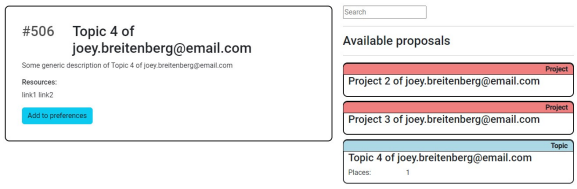

Figura 5.5: Propunerile unui profesor

Acest lucru permite adăugarea unei propuneri la lista sa de preferințe. El poate căuta în lista de propuneri a profesorului respectiv după titlu sau descriere. Pentru a marca și diferența opțiunile între ele după tip, proiectele au fost colorate cu roșu, iar tematicile (topics) în albastru, acestea având specificat și un număr limită de locuri disponibile, spre deosebire de proiecte care au câte unul singur.

Pagina preferințelor (**Preferences**) conține toate propunerile apreciate de către student. Acesta poate elimina de exemplu un proiect din listă sau poate modifica rating-ul, un număr întreg între 1 și 100. După cum a fost deja menționat, două preferințe pot avea același rating, iar acest număr este utilizat strict pentru ordonarea preferințelor. Atunci când este adăugată o preferință, aceasta are inițializat rating-ul cu 1, fiind la finalul ierarhiei.

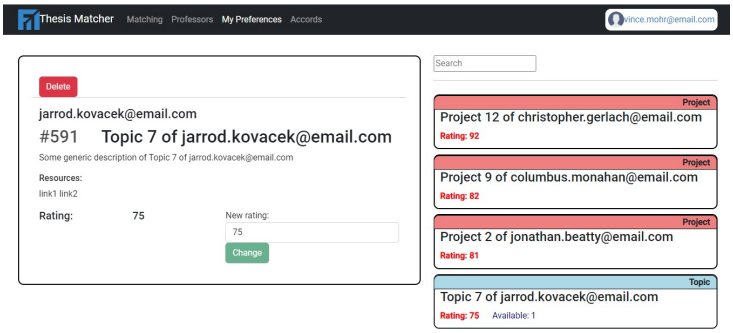

Figura 5.6: Preferințele

5.1.3 Interfața profesorului

Interfața unui profesor este în mare măsură similară. În pagina **Students** el poate vedea o listă de studenți și iniția un acord pentru o anumită lucrare, adică să asigneze un proiect sau o tematică unui student.

Pagina **My proposals** cuprinde propunerile create de profesor. Acesta poate adăuga noi proiecte, modifica elemente deja existente sau să le elimine.

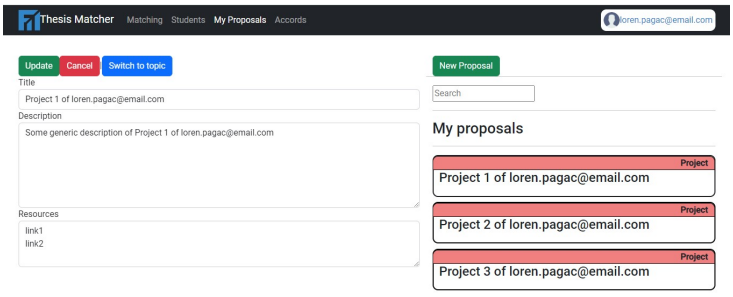

Figura 5.7: Propunerile

În secțiunea **Accords** există toate acordurile inițiate de către profesor. Pentru a fi luat în calcul, un acord trebuie însa acceptat de către student. În acest fel, studentului i se asignează deja un proiect, iar în acest fel atât acesta, cât și propunerea, nu mai intră în pasul de determinare a unei repartizări. Acordurile acceptate sunt marcate cu verde,

în caz contrar, cu roșu.

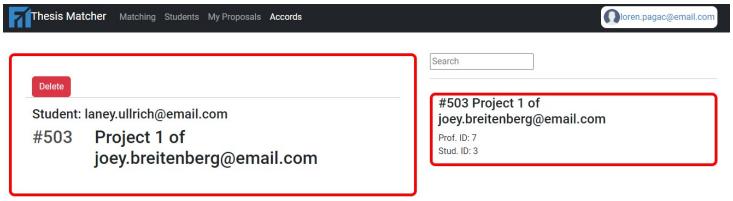

Figura 5.8: Acordurile

5.1.4 Administratorul

În cadrul acestei aplicații, fiind proiectată ca utilitar de uz intern, doar utilizatorii cu drepturi de administrator (ROLE_ADMIN) pot adăuga noi utilizatori. Prin urmare, doar aceștia au acces la pagina **Add User** de unde pot realiza acest lucru.

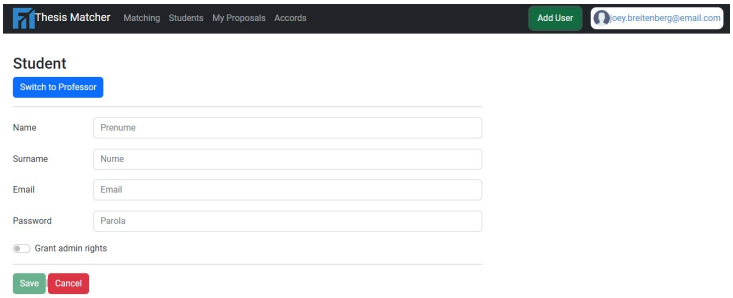

Figura 5.9: Adăugarea unui utilizator

5.2 Detalii de implementare

Înainte de a prezenta implementarea generală a părții de front-end, este necesară explicarea alcătuirii unei componente Angular.

Componenetele sunt elementele de bază ale interfeței unei aplicații Angular, o astfel de aplicație conține un arbore de componente [5]. Acestea sunt de fapt din punct de vedere tehnic un tip de directive, întotdeauna asociate cu un template.

O componentă este alcătuită dintr-un fișier Typescript ce modelează componenta prin intermediul unei clase, un *template* (șablon) ce descrie cum este construită (*rendered*) aceasta și eventual un fișiser CSS care indică detalii de stilizare [8].

5.2.1 Lifecylce hooks

Fiecare componentă are un ciclu de viață (lifecycle) care începe odată cu instanțierea și afișarea acesteia, continuă cu detectarea schimbărilor în view și în proprietățile instanței și se termină cu distrugerea componentei și eliminarea acesteia din DOM [6].

Există posibilitatea de a răspunde la astfel de evenimente din ciclul de viață al unei componente prin intermediul unor interfețe precum OnInit sau OnDestroy.

Figura 5.10: Metodele ngOnInit și ngOnDestroy în componenta UserList

Se poate observa cum în metoda `ngOnInit` a componentei `UserList` sunt efectuate operații necesare instanțierii și afișării listei de utilizatori. Utilizând serviciul `userService` injectat în constructor, se crează un obiect de tipul `Subscription` pentru a obține un tablou de utilizatori. În metoda `ngOnDestroy`, acest obiect este înlăturat pentru a optimiza folosirea memoriei.

5.2.2 Accesarea unei rute

Angular este utilizat în principal pentru crearea de aplicații *single-page*, adică toate funcționalitățile există într-o singură pagină HTML. Browser-ul încarcă doar părțile (componentele) necesare utilizatorului, fără a încărca o nouă pagină. De aceea, utilizatorul navighează prin intermediul rutelor predeterminate [7]. Acestea permit afișarea unor view-uri specifice în funcție de calea URL. Pentru a dispune de această funcționalitate, trebuie importat modulul `RouterModule` din pachetul `@angular/router`.

Rutele sunt stabilite în acest caz în modulul `AppRoutingModule`, iar definiția unei rute este din punct de vedere tehnic un obiect JavaScript [7]. Fiecare are măcar proprietățile `path` ce indică calea URL și `component`, numele componentei afișate. Definirea este realizată în manieră ierarhică, fiecare rută putând avea o listă de descendenți.

Spre exemplu, în imaginea următoare se pot vedea rutele necesare vizualizării propunerilor, a detaliilor unui anumit element, adăugarea și modificarea unei propuneri.

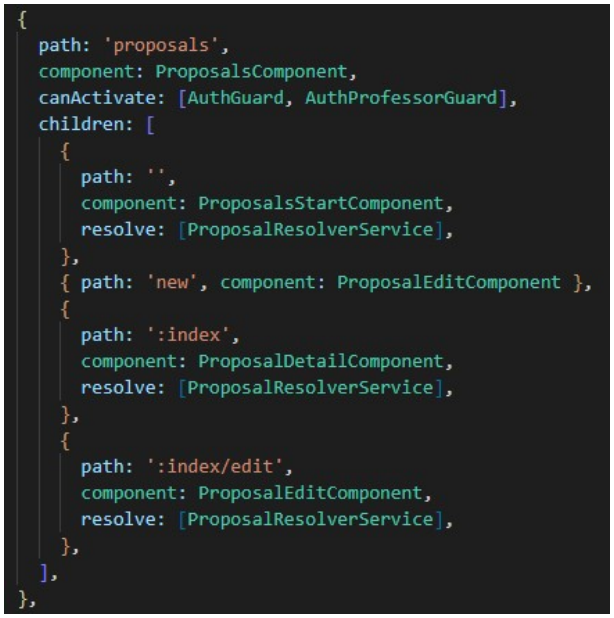

Figura 5.11: Rutele pentru propuneri

5.2.3 Serviciile

În cadrul acestei aplicații, serviciile sunt o parte esențială în special în comunicarea dintre componente și transmiterea și primirea de informații către/de la back-end. Fiecare serviciu este adnotat cu `Injectable` pentru a permite injectarea acestora în alte obiecte.

În figura următoare este prezentată parțial implementarea serviciului de autentificare.

Figura 5.12: Serviciul de autentificare

Acesta utilizează la rândul său serviciile `HttpClient` pentru comunicarea cu back-end-ul prin request-uri HTML și `Router` pentru navigare. Astfel, în metoda `login` este efectuat un request `POST`, cu *body* de tipul `x-www-form-urlencoded` (specificat în *headers*) conținând câmpurile *username* și *password* cu valorile aferente. Rezultatul acestui request este un obiect de tipul *AuthResponseData* ce conține în special câmpul *accessToken*. Acest token este un șir de caracter reprezentând un JWT (JSON Web Token) și este unic fiecărui utilizator și necesar pentru efectuarea cu succes a oricărui request ulterior.

Un alt detaliu este reprezentat de variabila `userData`, de tipul *BehaviourSubject<UserData>*, care permite emiterea unor evenimente pentru a indica autentificarea și dezautentificarea utilizatorului.

Resolver

O categorie particulară a serviciilor este cea de **Resolvers**. Un astfel de serviciu implementează interfața *Resolve* și este utilizat pentru operații premergătoare încărcării unei componente, iar specificarea acestui are loc în `AppRoutingModule`. În cazul propunerilor, prin intermediu serviciului `ProposalService` este obținută o listă completă a propunerilor profesorului autentificat.

```
@Injectable({
  providedIn: 'root',
})
export class ProposalResolverService implements Resolve<Proposal[]> {
  constructor(private proposalService: ProposalService) {}

  resolve(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Proposal[] | Observable<Proposal[]> | Promise<Proposal[]> {
    return this.proposalService.fetchUserProposals();
  }
}
```

Figura 5.13: Resolver pentru propuneri

Guard

O altă categorie este cea a gărzilor sau **Guards** în engleză, care autorizează accesul utilizatorilor la diversele rute în conformitate cu drepturile acestora, de student și profesor sau administator. În `AuthGuard` este detaliată verificarea dacă utlizatorul este autentificat. În cazul favorabil, acestuia îi este permis accesul la ruta respectivă prin returnarea valorii *true*, altfel este returnat un obiect de tipul textitUrlTree pentru a îl redirecționa spre autentificare.

```
@Injectable({
  providedIn: 'root',
})
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ):
    | Observable<boolean | UrlTree>
    | Promise<boolean | UrlTree>
    | boolean
    | UrlTree {
    return this.authService.userData.pipe(
      map((userDate) => {
        const isAuth = !!userDate;
        if (isAuth) {
          return true;
        }
        return this.router.createUrlTree(['/auth']);
      })
    );
  }
}
```

Figura 5.14: Guard pentru autentificare

5.2.4 Formularele

Obținerea, modificarea și ștergerea datelor nu ar fi posibilă fără formularele utilizate. În cadrul acestei aplicații, pe parte de front-end a fost aleasă metoda *template-driven* de implementare a formularelor (forms). Un astfel de tip folosește *two-way data binding* pentru a actualiza modelul de date din componentă în timp ce au loc modificări și vice-versa [4].

```
<form #authForm="ngForm" (ngSubmit)="onSubmit(authForm)" *ngIf="!isLoading">
  <div class="form-group">
    <label for="email">E-Mail</label>
    <input type="email" id="email" class="form-control" ngModel name="email"
      required email>
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input type="password" id="password" class="form-control" ngModel
      name="password" required minlength="6">
  </div>
  <div class="form-group">
    <button class="btn btn-primary" type="submit"
      [disabled]="!authForm.valid">Login</button>
  </div>
</form>
```

Figura 5.15: Formular de autentificare

Se poate observa cum formularul de autentificare este compus din mai multe input-uri, cel pentru email și cel pentru parolă, incluse în div-uri marcate de clasa *form-group*. Fiecare input are o serie de directive specifice ce asigură restricții necesare pentru o bună experiență a utlizatorului. De exemplu, directiva `required` indică obligativitatea completării input-ului, iar `email` verifică forma corectă a email-ului.

Procesarea datelor are loc pe în codul de Typescript odată ce formularul este trimis, în acel moment fiind apelată metoda `OnSubmit` ce primește ca argument un obiect de tipul *Form*.

5.3 Precizări

Versiunea de Angular utilizată este 13.

În primul rând, pentru stilizarea șabloanelor (templates) componentelor a fost folosit Bootstrap v5.2. Acest framework a permis facilitarea poziționării elementelor HTML, modificarea stilului și dimensiunilor acestora.

Căile URL necesare request-urilor către back-end au fost construite utilizând enum *ApiPaths* definit în fișiserul `environment.ts`.

Pentru modelarea ușoară a obiectelor JSON în comunicarea cu back-end au fost create clase *model* precum *ProposalModel*, *PreferenceModel* etc.

Capitolul 6

Back-end

Partea de back-end a aplicației Thesis Matcher a fost implementată în framework-ul de Java Spring Boot. Această tehnologie în primul rând oferă o varietate de utilități pentru dezvoltarea web, procesare în paralel, tranazacții cu baza de date [15]. Acesta beneficiază de un server integrat, în cazul de față este vorba de Tomcat, permițând un eventual *deployment* mai ușor al aplicației. De asemenea, spre deosebire de Spring, Spring Boot nu are nevoie de o configurare XML.

În al doilea rând, acest framework permite accesul la o serie utilă de plugin-uri care permit dezvoltarea unei securități adecvate, comunicare facilă cu baza de date și simplificarea codului

Ultimul argument pentru alegerea acestei tehnologii este familiaritatea cu acesta și comunitatea extinsă de utilizatori și tutoriale.

Codul aplicației de back-end a Thesis Matcher este structurat pe straturi (layers) ce permite localizarea ușoară a claselor în funcție de rolurile acestora [17]. Există astfel module pentru *controllers*, *models*, *repository*, *service*, *algorithm* etc.

6.1 Inițializarea proiectului

Datorită modului simplu și convenabil, proiectul a fost inițializat cu Spring Boot Initializr [20]. Tipul proiectului este *Maven*, cu versiunile de Java 11 și Spring Boot 2.7.6 pentru a fi compatibil cu versiunea de Java.

6.1.1 Dependențele

O dependență în Spring Boot poate fi comparată cu o librărie deoarece oferă anumite funcționalități. Gestionarea acestora de către programator este prin intermedul fișierului `pom.xml` (pentru un proiect Maven), odată adăugate, acestea sunt descărcate din *Maven Central* (un *repository* oferit de către comunitatea Maven) și stocate local în directorul `.m2` [18].

Partea de back-end a proiectului Thesis Matcher utilizează în primul rând dependențe necesare lucrului cu baza de date. `Spring Data JPA` permite persistența datelor cu ajutorul Spring Data și Hibernate, pe când `MySQL Driver` asigură conexiunea cu baza de date MySQL. Pe lângă `Spring Web` necesară creării de servicii pentru o aplicație web, back-end-ul beneficiază și de `Spring Boot DevTools` care facilitează dezvoltarea și automatizează restart-ul aplicației. Pentru partea de securitate a aplicației a fost utilizată atât `Spring Security` pentru a crea o autentificare și o autorizare personalizate, cât și `Java JWT` pentru procesarea cheilor JWT.

Pe lângă aceste dependețe, există și altele precum `Lombok` pentru generarea codului de tipul *boilerplate* (getters, setters etc.) sau `Java Faker` pentru generarea datelor entităților.

6.2 Configurarea

Fișierul `application.properties` menționat în secțiunea **4.6 Generarea datelor** conține elemente de configurare a aplicației, în special referitor la conexiunea cu baza de date. Este setat tipul bazei de date, în acest caz este MySQL, adresa la care rulează baza de date, username-ul și parola pentru conectare, precum și dialectul.

Server-ul de Tomcat este activ la adresa `http://localhost:8080` unde 8080 este portul.

Clasa `BackEndAppApplication` este cea care declanșează auto-configurarea și scanarea componentelor, pornind aplicația [19], motiv pentru care este adnotată cu `SpringBootApplication`. Tot aici este definit și metoda `passwordEncoder` adnotată cu `@Bean` (@Bean method) care descrie metoda de criptare a parolei unui utilizator cu funcția BCrypt.

6.3 Arhitectura Spring Boot

Imaginea următoare evidențiază maniera de tratare a interacțiunii cu clientul și tratarea operațiilor efectuate asupra bazei de date.

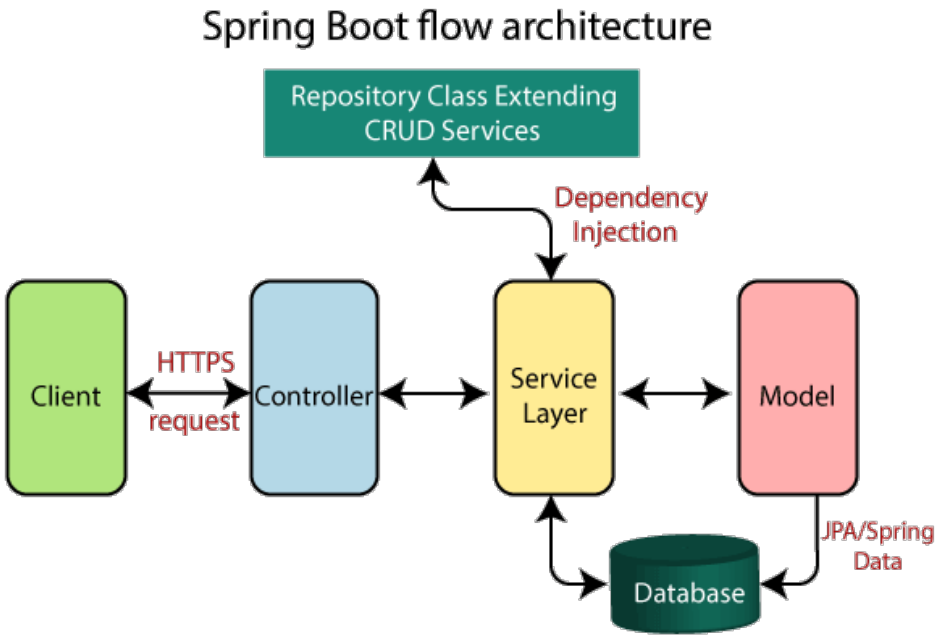

Figura 6.1: Arhitectura de flux a Spring Boot

Clientul efectuează un request HTTP care este gestionat în primă instanță de **controller** care utilizează **servicii** necesare tratării request-ului. În acestea are loc logica operațiilor asupra înregistrărilor din baza de date cu ajutorul claselor **model** și prin intermediul unor clase **repository**. Informația de la client și către acesta are loc de cele mai multe ori sub forma unor obiecte DTO (Data Transfer Object) convertite în JSON în body-ul request-ului. Un exemplu de clasă DTO este cea a propunerilor, `ProposalDto`, ce conține strict informațiile de interes.

Figura 6.2: Clasa ProposalDto

Aplicația de back-end poate fi considerată un *REST API*. Un API este un set de definiții și protocoale pentru dezvoltarea unei aplicații de software, constituind în fapt un mediator între utilizator (în cazul de fată fiind aplicația de front-end) și resursele sau serviciile cerute într-un mod controlat și sigur [21]. REST (Representational State Transfer) este la rândul său un set de constrângeri arhitecturale ce trebuie respectate pentru ca un API să poată fi considerat RESTful.

Un astfel de API trebuie să urmeză în primul rând o arhitectură client-server, unde request-urile sunt gestionate prin intermediul protocolului HTTP. API-ul transferă o reprezentare a instanței unei resurse spre un anumit *endpoint* (acesta include o adresă URL și identifică un canal de comunicare a resursei dintre back-end și front-end [22]), în format JSON de cele mai multe ori. De asemenea, această comunicare trebuie să fie *stateless*, adică fiecare request este separat de restul și nu sunt reținute informații referitor la tranzacțiile anterioare. În al doilea rând, transferul de resurse este standardizat în ideea că acestea sunt separate de reprezentările trimise către client și manipulate de către acesta tot prin intermediul reprezentărilor [21].

6.4 Modelele

Pentru a simula înregistrările dintr-o bază de date, Spring Boot dispune de posibilitatea de a defini clase ce identifică anumite entități cu ajutorul unor adnotări specifice. Spring Boot utilizează *Entity Scanning* pentru a le identifica în loc de un fișier special cum ar fi `persistence.xml` în Spring. Clasele luate în considerare sunt cele adnotate cu `Entity`, `Embeddable` sau `MappedSuperclass` [3].

Se observă de exemplu clasa `User`.

Figura 6.3: Clasa User

Proprietățile acestei clase identifică câmpurile, identificate de numele lor, ale tabelei `user` din baza de date, iar tipul lor este conform reprezentării lor. Id-ul este generat automat cu ajutorul unei valori reținute în tabela `hibernate_sequence` creată de framework-ul Hibernate, fiind de asemenea adnotată cu `@Id` pentru a sublinia că această proprietate reprezintă cheie primară.

Un utilizator de tipul profesor are o listă de propuneri (proposals, care sunt proiecte ori tematici), relația dintre tabela `user` și `proposal` prin intermediul cheilor străine (foregin keys) fiind specificată prin adnotarea `@OneToMany`. Opțiunea de `FetchType.LAZY` permite încărcarea mai rapidă a utilizatorului deoarece propunerile sale sunt instanțiate numai în cazul în care este nevoie. Fiecare utilizator are de altfel o colecție de roluri care pot fi `ROLE_USER` în mod normal sau `ROLE_ADMIN` și indică drepturile sale în aplicație.

Există și situația ca id-ul unei entități, precum cea a acordului, să fie compus, deoarece la nivelul reprezentării în tabela `accord`, o înregistrare este identificată prin id-urile studentului și profesorului care au hotărât colaborarea în realizarea unui anumit proiect. În consecință, a fost necesară crearea unei noi clase `AccordKey` care să simuleze cheia primară compusă și adnotarea proprietății cu `@EmbeddedId`.

6.5 Clasele *repository*

Pentru fiecare entitate modelată în cadrul aplicației există o clasă *repository* care reprezintă un mecanism de stocare, căutare și modificare a informațiilor din baza de date. Aceasta implementează interfața generică `JpaRepository` (Java Persistence API) care furnizează o serie de metode CRUD (Create Read Update Delete) pentru accesarea datelor. Există posibilitatea de definirea a unor metode particulare prin cuvinte cheie care ajută la generarea automată a interogărilor SQL, fiind un mod clar și facil de efectuare a operațiilor [13]. Astfel de clase descriu nivelul de acces al datelor (Data Acces Layer)

```
public interface AccordRepository extends JpaRepository<Accord, AccordKey> {
    List<Accord> findByIsAccepted(boolean isAccepted);
    List<Accord> findByStudent_Id(Long studId);
    List<Accord> findByStudent_IdAndIsAccepted(Long studId, boolean isAccepted);
    List<Accord> findByProfessor_Id(Long profId);
    Long countByProposal_Id(Long propId);
    boolean existsByStudent_IdAndIsAccepted(Long studId, boolean isAccepted);
}
```

Figura 6.4: Repository pentru un acord

Clasa `AccordRepository` conține de exemplu patru metode de căutare a acordurilor, în funcție de diferiți parametri. Metodele pot returna și dacă există o numită înregistrare, precum și numărul de anumite înregistrări. Datele sunt salvate sau actualizate cu metoda `save()` ce primește ca argument un obiect entitate, în cazul în care id-ul este *null*, este introdusă o nouă înregistrare, altfel este modificată cea cu id-ul respectiv.

6.6 Serviciile

Aplicația dispune pentru entități de interfețe servicii corespunzătoare ce descriu metode de manipulare a datelor respective, implementate mai departe de clase ce trebuie adnotate cu `@Service` și care crează funcționalitățile necesare, reprezintă astfel teoretic nivelul serviciilor.

Serviciile utilizează clase *repository* după cum este prezentat în figura următoare.

```java
@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
    User user = userRepo.findByUsername(username);
    if (user == null) {
        log.error("User not found in the database");
        throw new UsernameNotFoundException("User not found in the database");
    } else {
        // log.info("User {} found in the database", username);
    }
    Collection<SimpleGrantedAuthority> authorities = new ArrayList<>();
    user.getRoles().forEach(role -> {
        authorities.add(new SimpleGrantedAuthority(role.getName()));
    });
    return new CustomUser(user.getUsername(), user.getPassword(), authorities, user.getId(), user.isProfessor());
}
```

Figura 6.5: Metodă de obținere a utilizatorului după username

Aceasta obține un obiect de tipul `User` cu un username dat și în plus rolurile acestuia, fiind convertit ulterior în `CustomUser` pentru verificarea și autorizarea utilizatorului autentificat în contextul aplicației.

6.7 Controlerele (Controllers)

Într-un final, cel de-al treilea nivel, de prezentare (presentation layer), este constituit de controlere care efectuează comunicarea cu partea de front-end și ale căror clase sunt adnotate cu `RestController`. Metodele descriu tratarea request-urilor HTTP cu ajutorul diferitelor servicii.

```java
@GetMapping("/professors")
public ResponseEntity<List<UserDto>> getProfessor() {
    List<User> users = userService.getProfessors();
    List<UserDto> usersDto = users.stream().map(this::convertToDto).collect(Collectors.toList());
    return ResponseEntity.ok().body(usersDto);
}
```

Figura 6.6: Metodă de trimitere a tuturor profesorilor

```java
@PostMapping("/user/save")
public ResponseEntity<User> saveUser(@RequestBody User user) {
    URI uri = URI.create(ServletUriComponentsBuilder.fromCurrentContextPath().path("/api/user/save").toUriString());
    return ResponseEntity.created(uri).body(userService.saveUser(user));
}
```

Figura 6.7: Metodă de salvare a unui utilizator

În cazul clasei `UserController`, metoda `getProfessors` este apelată de fiecare dată când este primit un request de tipul GET la endpoint-ul `"/professors"`,

lucru evidențiat de adnotarea premergătoare definiției. Răspunsul HTTP este reprezentat de un obiect de tipul `ResponseEntity` cu *status code* 200 (cerere efectuată cu succes) și având în *body* o listă de obiecte de tipul `UserDto`, după cum a fost precizat anterior.

Metoda `saveUser` primește ca parametru un utilizator transmis prin body-ul request-ului POST, lucru ce trebuie specificat prin adnotarea `@RequestBody`. Răspunsul este obiectul salvat, cu *status code* 201 (created).

6.8 Securitatea

Securitatea aplicației este dezvoltată de pachetul `security` care conține în principal clasa `SecurityConfig` ce extinde clasa abstractă `WebSecurityConfigurerAdapter` și este adnotată cu `Configuration` și `EnableWebSecurity`. Aceasta suprascrie metoda `configure` care indică maniera de tratare a utilizatorului ce interacționează cu aplicația.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    CustomAuthenticationFilter customAuthenticationFilter = new CustomAuthenticationFilter(authenticationManagerBean());
    customAuthenticationFilter.setFilterProcessesUrl("/api/login");
    http.cors().and().csrf().disable();
    http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    http.authorizeRequests().antMatchers("/api/login/**", "/api/token/refresh/**", "/api/matchings").permitAll();
    http.authorizeRequests().antMatchers(HttpMethod.GET, "/api/user/**").hasAnyAuthority("ROLE_USER");
    http.authorizeRequests().antMatchers(HttpMethod.POST, "/api/user/save/**").hasAnyAuthority("ROLE_ADMIN");
    http.authorizeRequests().antMatchers(HttpMethod.POST, "/api/proposal/save/**").hasAnyAuthority("ROLE_USER");
    http.authorizeRequests().anyRequest().authenticated();
    http.addFilter(customAuthenticationFilter);
    http.addFilterBefore(new CustomAuthorizationFilter(), UsernamePasswordAuthenticationFilter.class);
}
```

Figura 6.8: Metodă de configurare a securității

Metoda primește ca parametru o variabilă `http` de tipul `HttpSecurity` și care permite configurarea contextului abordat. Utilizând un sistem bazat pe chei JWT (JSON Web Token), aplicația de back-end nu trebuie să creeze o sesiune pentru fiecare utilizator.

JWT este un standard care definește un mod compact de transmitere sigură a informațiilor între aplicații sub forma unor obiecte JSON, putând fi considerată o semnătură digitală a utilizatorului. Un astfel de token are trei părți separate de caracterul punct: header, payload și semnătură. Header-ul conține tipul token-ului, adică JWT, și algoritmul folosit pentru semnarea acestuia, aplicația Thesis Matcher utilizează *HMAC256*.

Payload-ul conține pretențiile (claims) cum ar fi rolurile deținute, dacă utilizatorul este profesor sau nu și altele. Semnătura este obținută prin criptarea primelor două părți cu algoritmul specificat și un șir de caractere secret știut doar de către back-end [14]. Fiecare token are un termen de validitate pentru a indica perioada în care utilizatorul este autentificat.

De asemenea, request-urile din partea clientului sunt autorizate în funcție de anumite filtre, implementându-se în acest scop unul pentru autentificare în primă instanță și unul pentru autorizare.

6.8.1 Autentificarea

Autentificarea este verificarea identității unui utilizator, în acest caz find realizată într-un mod clasic cu ajutorul username-ului și parolei. Utilizatorul furnizează credențialele sale care sunt verificate pe partea de back-end și sunt luate măsuri în funcție de validitatea acestora [9].

Verificarea este asigurată de un filtru de autentificare.

```
@Override
public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) throws AuthenticationException {
    String username = request.getParameter("username");
    String password = request.getParameter("password");
    log.info("Username: {}\nPassword: {}", username, password);
    UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(username,
            password);
    return authenticationManager.authenticate(authenticationToken);
}
```

Figura 6.9: Tentativă de autentificare

Atunci când un utilizator dorește acces la aplicație, în metoda descrisă mai sus este informată proporietatea `authenticationManager` de datele utilizatorului și încercă autentificarea acestuia.

```
// access jwt token
String accessToken = JWT.create()
        .withSubject(user.getUsername())
        .withExpiresAt(expirationDate)
        .withIssuer(request.getRequestURL().toString())
        .withClaim("roles",
                user.getAuthorities().stream().map(GrantedAuthority::getAuthority).collect(Collectors.toList()))
        .withClaim("isProfessor", user.isProfessor())
        .sign(algorithm);
```

Figura 6.10: Autentificare cu succes

În cazul în care este autentificat cu succes, este creată cheia de acces, denumită

`accessToken`, în metoda suprascrisă `successfulAuthentication`. Token-ul va conține informații despre username-ul persoanei, termenul de valabilitate, rolurile și tipul acesteia. Rezultatul este trimis către client sub formă de JSON. Această cheie JWT va fi folosită în toate request-urile viitoare ale utilizatorului pentru confirmare identității acesteia.

6.8.2 Autorizarea

Autorizarea este deciderea dacă o anumită persoană are permisiunea de a accesa o anumită resursă și este realizată în funcție de rolurile acesteia. Un administrator va avea de aceea un număr extins de drepturi cum ar fi adăugarea unui nou utilizator, acțiune pe care un cont doar cu ROLE_USER nu o poate face. Modalitatea de autorizarea este restricționarea anumitor URL-uri în funcție de permisiuni [9].

În consecință, fiecare utilizator este obligat ca, odată ce este autentificat, să poată fi identificat și autorizat înaintea de executarea request-urilor efectuate. După autentificarea cu succes a sa, utilizatorul primește un JWT (JSON Web Token), un șir de caractere ce conține criptat informații specifice cum ar fi username-ul, tipul de utilizator (profesor sau student) și permisiunile sale (ROLE_USER, ROLE_ADMIN).

```
String authorizationHeader = request.getHeader(AUTHORIZATION);
if (authorizationHeader != null && authorizationHeader.startsWith("Bearer ")) {
    try {
        String token = authorizationHeader.substring("Bearer " .length());
        Algorithm algorithm = Algorithm.HMAC256(SECRET.getBytes());
        JWTVerifier verifier = JWT.require(algorithm).build();
        DecodedJWT decodedJWT = verifier.verify(token);
        String username = decodedJWT.getSubject();
        String[] roles = decodedJWT.getClaim("roles").asArray(String.class);
        Collection<SimpleGrantedAuthority> authorities = new ArrayList<>();
        stream(roles).forEach(role -> {
            authorities.add(new SimpleGrantedAuthority(role));
        });
        UsernamePasswordAuthenticationToken authenticationToken =
                new UsernamePasswordAuthenticationToken(username, null, authorities);
        SecurityContextHolder.getContext().setAuthentication(authenticationToken);
```

Figura 6.11: Autorizarea în `CustomAuthorizationFilter`

În filtrul de autorizare, este verificată validitatea requset-ului primit. Acesta trebuie să conțină un header *Authorization* cu valoarea token-ului unic precedat de șirul de caractere ”Bearer ”. Cheia JWT este decriptată și este decisă acordarea permisiunii

pentru reqest-ul respectiv. Dacă nu sunt îndeplinite condițiile adecvate, este trimis un răspun cu status code 401, *Unauthorized*, sau 403, *Forbidden*.

După definirea celor două filtre cu responsabilitățile specifice, acestea sunt ordonate în mod logic pantru verificarea datelor și sunt descrise condițiile de validitate pentru rutele corespunzătoare request-urilor.

Există pe lângă cele două mecanisme menționate anterior un filtru care gestionează verificările CORS (Cross-Origin Resource Sharing) în care este specificată adresa de origine a aplicației de front-end, a clientului. Server-ul primește din partea browser-ului înainte de procesarea fiecărui request un *preflight request* care verifică permisiunile pentru anumite metode HTTP. În consecință, metoda @Bean `corsConfigurationSource` din clasa `SecurityConfig` marchează adresa de origine `http://localhost:4200` a front-end-ului ca fiind de încredere și permite efectuarea request-urilor provenind de la acesta.

Capitolul 7

Algoritmul de repartizare

Problema pe care își propune această aplicație să o rezolve este o instanță a problemei asignării sau problema atribuirii (*assignment problem*). Este de precizat că cele două nume vor fi folosite interschimbabil în secțiunile următoare. În cazul de față, instanța este reprezentată de o mulțime de studenți, fiecare cu o ordonare a unor preferințe, și o mulțime a propunerilor profesorilor (proiecte sau tematici generale). Cerința este repartizarea optimă a propunerilor către studenți în funcție de preferințele acestora.

În literatura de specialitatea au fost descriși și dezvoltați diverși algoritmi de rezolvare a unei astfel de instanțe a problemei atribuirii. Algoritmul implementat de aplicația Thesis Matcher este unul din clasa de algoritmi de licitație (*auction algorithms*), descris în lucrarea "Assignment Problem with Constraints" scrisă de către Ulrich Bauer, Facultatea de Informatică a Universității Tehcnice din München, 2005. Pentru a descrie coerent și cât mai clar logica acestui algoritm, în secțiunile următoare vor fi descrise problema atribuirii și similitudinile acesteia cu o problemă de flux, conceptele matematice ale algoritmilor propuși pentru rezolvare și pașii algoritmului implementat pentru instanța prezentă.

7.1 Problema asignării (atribuirii)

Problema simetrică a asignării constă în două mulțimi X și Y de dimensiuni egale, o mulțime $E \subseteq X \times Y$ și o funcție de cost c_{xy} pentru oricare pereche posibilă $(x, y) \in E$. Scopul este cuplarea oricărui element din X cu un element din Y ca la final costul total să fie minim [26].

În contextul grafurilor, această problemă poate fi redusă la problema fluxului de cost minim într-un graf bipartit $G = ((X \bigcup Y,\ E)$ cu o funcție de cost $c_{xy},\ (x, y) \in E$, și capacitatea $u_{xy} = 1, \forall (x, y) \in E$.

Expresia matematică a problemei [26, p. 6] este

$$minimizarea \sum_{(i,j)\in E} c_{ij}x_{ij}$$

astfel încât

$$\sum_{j:(i,j)\in E} x_{ij} = 1, \forall i \in X,$$

$$\sum_{i:(i,j)\in E} x_{ij} = 1, \forall j \in Y,$$

$$x_{ij} \geq 0, \forall (i, j) \in E$$

.

Variabila x_{ij} indică câtă unități de flux sunt trimise pe muchia (i, j).

Cu toate acestea, problema repartizării fiecărui student o teză de licență în funcție de preferințele sale este o problemă asimetrică deoarece numărul de propuneri, $|Y|$, este mai mare sau egal decât numărul de studenți (trebuie să fie măcar egal pentru a putea atribui fiecărui student o lucrare). La final, o soluția determinată se va afla în variabila x_{ij} unde dacă $x_{ij} = 1$, atunci studentului i îi este repartizată propunerea j din totalul de propuneri al tuturor profesorilor.

Important de precizat este că în acest caz, rating-ul total al preferințelor satisfăcute trebuie maximizat. De aceea, este realizată o normalizare în primă instanță pentru a transforma rating-urile în costuri, prin calcularea $c_{ij} = 101 - rating_{ij},\ rating_{ij} \in \mathbb{N} \cap [1, 100],\ \forall (i, j) \in E$.

7.2 Algoritmul de licitație

Algoritmul de licitație este o metodă intuitivă ce rezolvă problema clasică a atribuirii (asignării), reușind de asemenea să depășească în performanță algoritmi similari, fiind în același timp potrivit pentru calculul în paralel [27, p. 1]. Acest algoritm a fost propus de matematicianul Dimitri Bertsekas și este în sine o metodă euristică aplicată algoritmului *push/relabel* pentru a îmbunătăți rezultatele practice [26, p. 24], având în mod similar o complexitate de $\mathcal{O}(nm \log(nC))$ unde n este numărul total de noduri ($|X|+|Y|$), m este numărul de muchii $|E|$, iar $C = max_{(i,j)}\{c_{ij}\}$ este maximul costurilor.

Algoritmul de licitație poate fi descris ca o licitație în viața reală. Elementele din mulțimea X, studenții, licitează pentru elementele din Y, lucrările propuse de profesori. Pentru un student i, propunerea j este cea mai benfică dacă suma costului c_{ij} și prețul $\pi(j)$ este minim, iar o propunere este atribuită studentului cu cea mai bună ofertă[26]. Funcția π este utilizată în pasul de reetichetare (*relabel*) a unei propuneri.

Pentru a permite finalizarea algoritmului, prețurile π trebuie să crească cu o valoare ϵ. Mai mult decât atât, Bertsekas a fost primul care a propus scalarea prin ϵ (ϵ-*scaling*) pentru a îmbunătăți performanța algoritmului în cazul instanțelor unde numărul de obiecte din Y licitate este mai mic. În cazul în care ϵ este prea mic, poate apărea un "război al prețurilor" când este nevoie de o creștere mai substanțială a prețurilor, deci un număr mare de iterații, până anumiți studenți găsesc propuneri mai profitabile decât cele licitate. Soluția este execuția algoritmului începând cu ϵ inițializat cu cel mai mare cost al preferințelor (C) și scăderea treptată a acestuia de la o iterație la alta.

Algoritmul de licitație poate fi corelat cu unul de calculare a unui flux de cost minim, cu un graf $G = (X \cup Y, E)$ unde X reprezintă studenții, iar Y propunerile. Este definită o nouă variabilă c_{ij}^{π} numită costul redus al muchiei (i, j) din graful $G = (V, E)$ în raport cu funcția de preț $\pi : V \to \mathbb{R}_+$, $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$.

Un pas de licitație este numit *double-push* deoarece este echivalent cu operațiile *push* și *relabel*. Când un student x licitează pentru un proiect y, este efectuat un flux pe muchia (x, y). Dacă y a fost deja asigant unui x', y transmite flux către x' deoarece are un exces pozitiv.

7.2.1 Algoritmul de licitație pentru problema asimetrică a asignării

În cazul problemei asimetrice de asignare, $|X| \leq |Y|$ și trebuie determinat care dintre muchii trebuie luate în calcul. Se poate întâmpla ca după o iterație, prețul unui element din Y poate deveni prea mare și poate rămâne neasignat în iterația următoare, deși ar putea fi preferat de către un student. În cazul problemei simetrice, fiecare $y \in Y$ este asignat.

O primă abordare este transformarea problemei asimetrice într-una simetrică prin introducerea unor $|Y| - |X|$ noduri "virtuale" în mulțimea X, fiecare dintre ele conectat la toate nodurile din Y cu câte un cost 0. Acest lucru poate rezulta însă la un graf dens din punct de vedere al muchiilor.

O altă abordare, utilizată și în proiectul de față, este utilizarea unui arbore binar

(*binary heap*). De fiecare dată când este reetichetat (*relabeled*) un nod care nu se află în heap, este introdus în heap și eliminat din acesta nodul cu cel mai mic preț. Dacă acest nod este atribuit, fluxul pe muchia respectivă devine 0. Utilizând un arbore binar, reetichetarea nodului are loc cu o complexitate de $\mathcal{O}(\log(|X|)$, iar accesarea nodului cu cel mai mic preț este imediată deoarece se află în vârf. Complexitatea generală a algoritmului crește la $\mathcal{O}(nm \log(nC) \log(|X|))$, care nu afectează mult însă performanța algoritmului.

7.3 Pseudocod

Algoritmii următori scriși în pseudocod și utilizați de aplicație au fost descriși în lucrarea menționată anterior, *Constrained Assignment* [26].

7.3.1 Funcția *doublePush* cu arbore binar (heap)

Algorithm 1 $doublePush(i)$

Fie (i, j) și (i, k) muchii astfel încât c_{ij}^{π} să fie minim și c_{ik}^{π} al doilea cel mai mic
$x_{ij} \leftarrow 1$ //push
if $j \notin H$ **then** // H este arborele binar (heap)
    $H.put(\pi(j),\ j)$ // j este introdus în heap cu cheia prețul său actual
    if $|X| > H.size()$ **then**
        $g \leftarrow H.min()$
        $H.removeMin()$
        if $\exists f \in X, x_{fg} = 1$ **then** // propunerea g este deja asignată unui student f
            $x_{fg} \leftarrow 0$ // push
        end if
    end if
end if
if $\exists h \in X, h \neq i$ și $x_{hj} = 1$ **then** // propunerea j era deja atribuită unui alt student h
    $x_{hj} \leftarrow 0$ // push
end if
$\pi(j) \leftarrow \pi(k) + c_{ik} - c_{ij} + \epsilon$ // relabel

7.3.2 Funcția *refine* de îmbunătățire a rezultatului

Algorithm 2 $refine()$

1: $\epsilon \leftarrow \epsilon/\alpha$ // α este o constantă aleasă în mod euristic
2: **for all** $(i, j) \in E$ **do**
3: $\quad x_{ij} \leftarrow 0$
4: **end for**
5: **while** $\exists i \in X$, i neasignat **do**
6: $\quad doublePush(i)$
7: **end while**

7.3.3 Algoritmul de licitație pentru asignare

Algorithm 3 $solve()$

1: $\epsilon \leftarrow max_{(i,j)\in E}\{c_{ij}\}$
2: **for all** $j \in Y$ **do** //inițializează prețul tuturor propunerilor cu 0
3: $\quad \pi(j) \leftarrow 0$
4: **end for**
5: **while** $\epsilon \geq 1/n$ **do**
6: $\quad refine()$
7: **end while**
8: return x

7.4 Implementare

Implementarea algoritmului prezentat anterior a avut loc în limbajul Java pentru a putea fi folosit în cadrul unei aplicații de Spring Boot. A fost creată în mod special clasa *Edge* pentru a modela muchiile dintre cele două mulțimi X a studenților și Y a propunerilor, mai precis preferințele studenților. Câmpul **start** indică indicele studentului și câmpul **end** indică indicele propunerii preferate.

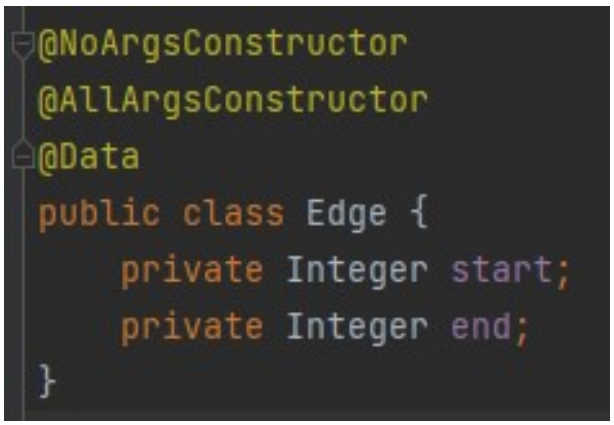

Figura 7.1: Clasa Edge

7.4.1 Preprocesarea datelor

Înainte de efectuarea algoritmului, este nevoie de operații de preprocesare pentru a converti obiectele primite de la baza de date (studenții, propunerile, preferințele) în variabile de input adecvate. Pentru acest lucru a fost creată clasa *Convertor* care utilizează patru servicii, `accordService`, `preferenceService`, `proposalService` și `userService`, pentru a accesa înregistrările din baza de date, reținându-se în două liste `studIds` și `propIds` id-urile studenților (students) și propunerilor (proposals). Sunt eliminate id-urile care se află într-un acorduri acceptate, adică cele care au câmpul `isAccepted` $= true$.

Algoritmul este nevoit să primească listele X și Y sub formă de indecși ale elementelor în listele `studIds` `propIds`, iar nu id-urile în mod direct, deoarece propunerile pot fi de două feluri: proiect (project) sau tematică (topic). Tematicile pot avea un număr de locuri disponibile mai mare de 1, de aceea în `propIds` se adaugă duplicate ale aceluiași id câte locuri sunt disponibile.

Indecșii id-urilor studenților sunt reținute într-o variabilă `studIndices` de tipul `HashMap<Long, Integer>` unde cheile sunt id-urile, iar valorile sunt indecșii în lista de id-uri. Similar, indecșii propunerilor sunt reținuții în variabila `propIndices` de tipul `HashMap<Long, List<Integer>>` de această dată deoarece un id de propunere poate avea mai multe duplicate în listă.

După aceste convertiri, este calculată o variabilă c a costurilor (similară modelului teoretic) de tipul `HashMap<Edge, Double>` unde cheile sunt muchiile (preferințele), iar valorile acestora sunt costurile. Un cost se calculează astfel: preferințele unui student sunt ordonate descrescător, teza de licență cea mai favorabilă primind costul 1.0,

iar două propuneri cu același rating primind același cost.

7.4.2 Algoritmul

Algoritmul este modelat de clasa *AssignAlgorithm* ce este inițializat cu variabilele n numărul de studenți, m numărul de propuneri, c variabila costurilor. Clasa dispune de metodele similare algoritmilor prezentați anterior în pseudocod.

În metoda *solve()* este inițializată o listă `unassigned` cu toți studenții neatribuiți. Se inițializează variabila eps cu cel mai mare cost din c și variabila pi este setată pe valoarea 0 pentru toate propunerile. Cât timp eps nu depășește pragul $alpha/(n+m)$ se execută metoda *refine()*.

Metoda *refine()* reinițializează variabila soluție x cu 0 pe fiecare muchie. Se alege în mod aleatoriu un student neasignat și se execută metoda *doublePushHeap(i)*.

```
private void doublePushHeap(Integer i) {
    Edge edge;
    Integer[] result = getEndsWithMinReducedCost(i);
    Integer j = result[0], k = result[1];
    if (j != null && k != null) {
        x.put(new Edge(i, j), 1); // push
        solution.put(i, new Assignation(j, c.get(new Edge(i, j))));
        unassigned.remove(new Integer(i));
        if (!heap.containsValue(j)) {
            heap.put(pi[j], j);
            if (heap.size() > n) {
                Integer g = heap.pollFirstEntry().getValue();
                for (Integer f = 0; f < n; ++f) {
                    edge = new Edge(f, g);
                    if (x.containsKey(edge)) {
                        x.put(edge, 0); // push
                        solution.put(i, null);
                        unassigned.add(f);
                    }
                }
            }
        }
        for (Integer h = 0; h < n; ++h) {
            edge = new Edge(h, j);
            if (!h.equals(i) && x.get(edge) != null && x.get(edge) == 1) {
                x.put(edge, 0); //push
                unassigned.add(h);
            }
        }
        pi[j] = pi[k] + c.get(new Edge(i, k)) - c.get(new Edge(i, j)) + eps;
        pi[j] = round(pi[j]);
    }
}
```

Figura 7.2: Metoda *doublePushHeap*

Este definită variabila locală `result` ca un tablou de două poziții. Poziția `result[0]` conține j, iar, `result[1]` conține k, unde (i, j) este muchia cu cel mai mic cost redus

și (i, k) muchia cu al doilea cel mai mic cost redus. Variabila `heap` a fost declarată de tipul `TreeMap<Double, Integer>` unde cheile sunt prețurile propunerilor ($\pi[j]$) și valorile sunt indecșii. Acest lucru permite ordonarea propunerilor asignate în funcție de prețul acestora, de asemenea accesarea propunerii (elementului $j \in Y$) cu prețul $\pi(j)$ minim are loc în $\mathbb{O}(1)$.

Orice alt student h asignat anterior propunerii j este marcat ca fiind neasignat (push step). În contextul problemei fluxului într-un graf bipartit, fluxul dinspre studentul i spre propunerea j este setat pe 0.

```
for (Integer h = 0; h < n; ++h) {
    edge = new Edge(h, j);
    if (!h.equals(i) && x.get(edge) != null && x.get(edge) == 1) {
        x.put(edge, 0); //push
        unassigned.add(h);
        break;
    }
}
```

Figura 7.3: Pasul de push

Pasul final este cel de reetichetare (relabel step) observat în imaginea următoare.

```
pi[j] = pi[k] + c.get(new Edge(i, k)) - c.get(new Edge(i, j)) + eps;
pi[j] = round(pi[j]);
```

Figura 7.4: Pasul de relabel

7.4.3 Procesarea rezultatului

Soluția constă într-o variabilă de tipul `Map<Integer, Assignation>` cu cheile reprezentând indecșii studenților în lista `studIds`. Valorile sunt de tipul `Assignation` ce conține câmpul `end` ce reprezintă indexul propunerii atribuite în lista `propIds` și câmpul `cost` ce are calculat calitatea preferinței satisfăcute. Acest lucru permite accesarea rapidă a soluției pentru un student i și construirii rezultatului total.

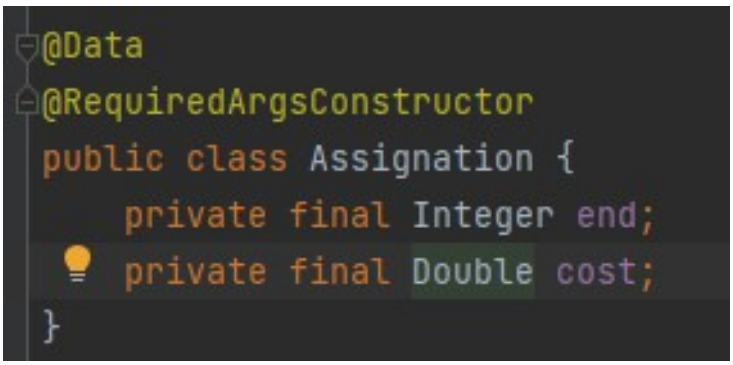

Figura 7.5: Tipul Assignation

La finalul execuției algoritmului, există posibilitatea ca un număr de studenți să nu aibă proiecte atribuite în cazul în care aceștia nu au avut un număr de preferințe îndeajuns de mare. Dacă există o astfel de situație, se alege în mod aleatoriu un student dintre aceștia și îi este asignată o propunere dintre cele care nu au fost încă luate până când toți studenții sunt repartizați.

Rezultatul final este alcătuit în primul rând din studenții care au stabilit acorduri cu profesori pentru anumite proiecte, repartizările calculate de algoritm și repartizările ulterioare ale studenților neatribuiți.

Din punct de vedere tehnic, rezultatul transmis către front-end prin intermediul unui request de tipul GET este o listă de obiecte de tipul `MatchingDto` (DTO = Data Transfer Object).

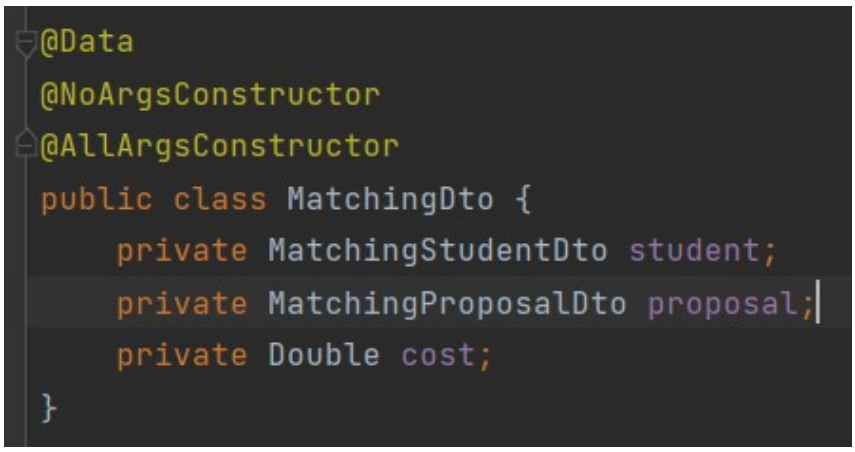

Figura 7.6: Clasa MatchingDto

Figura 7.7: Clasa MatchingStudentDto

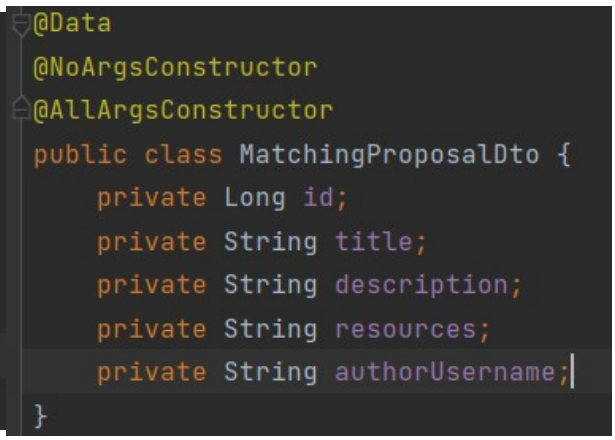

Figura 7.8: Clasa MatchingProposalDto

7.5 Precizări

Variabila `pi` conține valori reale, dar a fost considerat că o precizie de două variabile este de ajuns pentru efectuarea algoritmului. Pentru trunchierea valorilor, a fost implementată o metodă statică `round(double value)` care aproximează superior prețurile calculate.

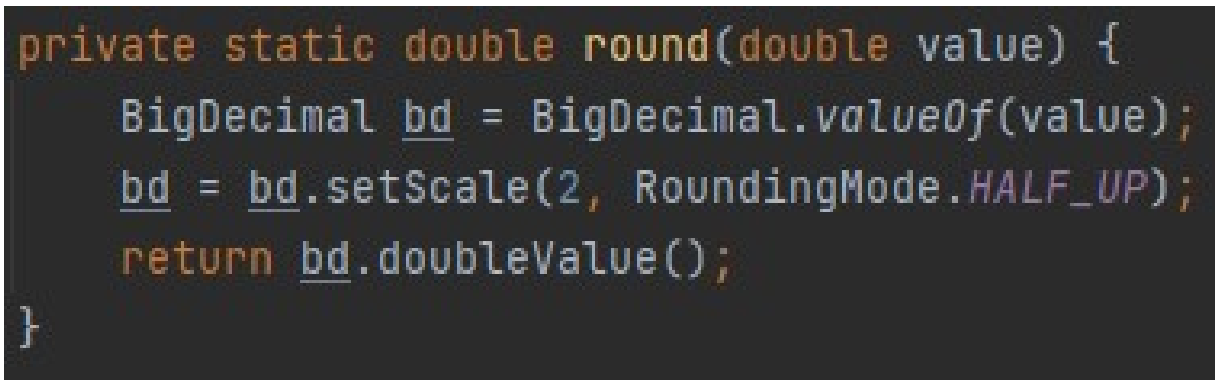

Figura 7.9: Rotunjirea prețurilor

Constanta `alpha` a fost inițializată cu valoarea 10, urmând recomandările menționate în lucrarea *Constrained Assignment* [26, p. 46].

Variabila `eps` are valorea inițială egală cu cel mai mare cost dintre muchiile existente.

Capitolul 8

Scenarii de utilizare

În mod natural, persoanele dintr-o facultate utilizează aplicația Thesis Matcher în scopuri diferite în funcție de tipul lor. Fiecare este întâmpinat de pagina principală **Matching** care prezintă repartizarea tezelor de licență studenților în funcție de preferințele lor. Însă odată autentificat, interfața este adaptată nevoilor specifice utilizaotrului.

Un student poate vizualiza lista tuturor profesorilor și poate accesa propunerile unui anumit profesor. Acesta are opțiunea de a adăuga anumite proiecte sau tematici în lista proprie de preferințe, dacă nu au fost încă convenite în acorduri profesor-student. Preferințele urmează a fi ordonate în funcție de rating.

Un profesor care este are administrator, după autentificare, poate adăuga un cont pentru un nou student. Ulterior, profesorul poate încheia un acord cu acest student asupra unui proiect, dacă studentul acceptă o astfel de înțelegere, oferindu-i astfel șansa acestuia de a nu mai participa în etapa de reaprtizare algoritmică, fiindu-i deja atribuită o lucrare convenită.

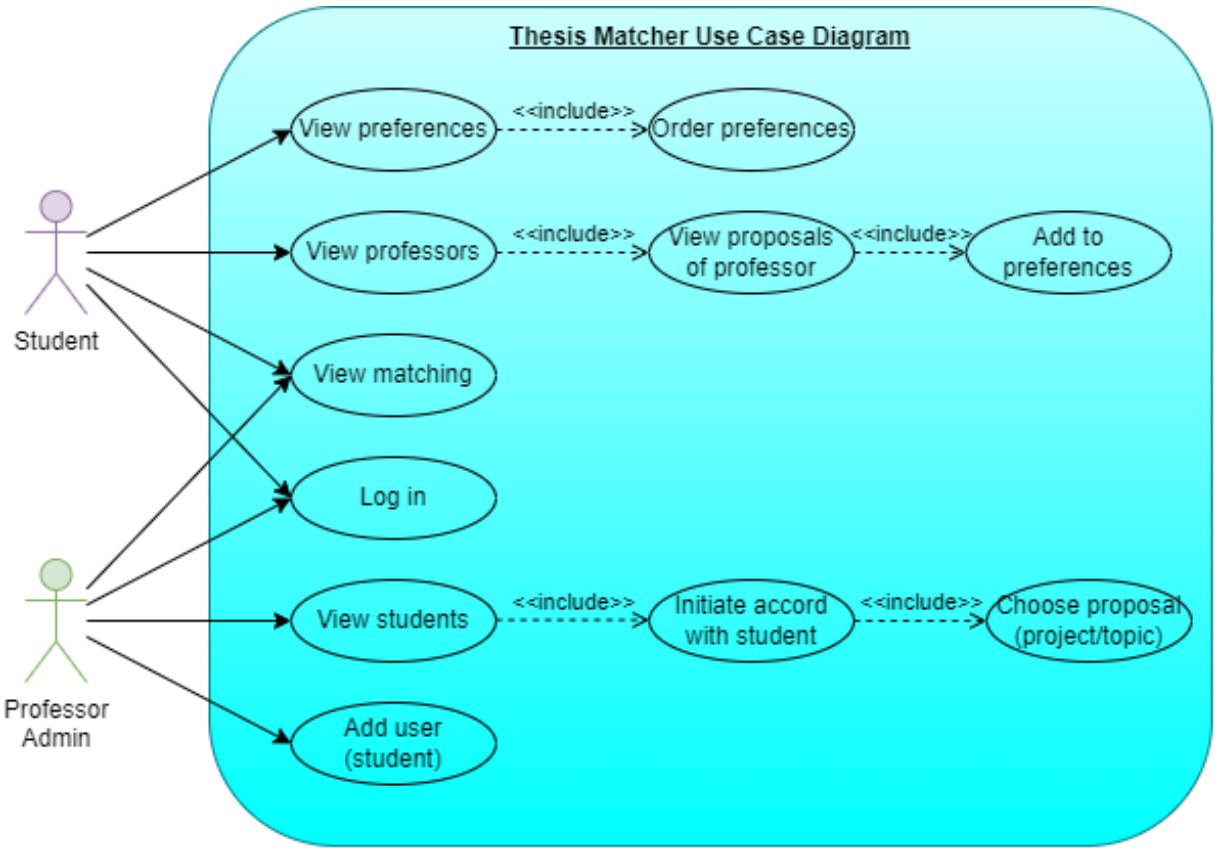

Figura 8.1: Diagramă Use Case

Concluzii

Aplicația Thesis Matcher propune să rezolve problema des întâlnită în facultăți de a repartiza studenții la profesorii de licență, mai precis de a atribui tezele propuse studenților ținând cont de preferințele lor. Într-un final, scopul principal este de a oferi șansa viitorilor absolvenți de a realiza o lucrare finală pe măsura aptitudinilor cumulate pe parcursul anilor de studenție, dar și în funcție de înclinațiile lor spre o anumită specializare. Mai mult decât atât, aplicația oferă posibilitatea unei cercetări riguroase de către student a proiectelor propuse prin centralizarea acestora și opțiunea de a conveni cu un profesor asupra unei propuneri favorabile, sub forma unui acord.

Dezvoltarea acestei aplicații a oferit prilejul cercetării mai amănunțite a tehnologiilor utilizate și dezvoltării abilităților în crearea de soluții software. Aplicația prezintă astfel interes în modul de adaptare a interfeței în funcție de utilizator și în același timp adoptarea unui design simplist și intuitiv. Partea de back-end prezintă o implementare personalizată a mecanismului de securitate adaptat unui context special, de uz intern.

Pe lângă aceste aspecte, urmând a rezolva problema de față, au fost cercetați mai mulți algoritmi potențiali, alegându-se într-un final un algoritm de atribuire prin double-push cu heap. Una dintre provocări a fost adaptarea datelor problemei la cele de intrare (input) și optimizarea prelucrării informațiilor de la baza de date. De asemenea, implementarea conceptelor teoretice descrise în Java constă un alt interes prin modularizarea și particularitățile codului.

Posibile dezvoltări viitoare

Aplicația prezintă o serie de direcții de dezvoltare în viitor, prezentând un potențial de a deveni după anumite îmbunătățiri și modificări o aplicație utilitară robustă în contextul intern al facultății.

O primă idee de dezvoltare este implementarea unor noi algoritmi adaptați da-

telor de intrare în urma cercetării performanțelor acestora. O altă idee este implementarea unui sistem de preferințe și pentru profesori care vor putea nu numai încheia acorduri pentru anumite lucrări propuse, ci și de a crea o listă ierarhizată de studenți potențial coordonați. Studenți la rândul lor ar putea avea opțiunea de a propune o lucrare. De asemenea, introducerea de noi utilizatori ar putea fi mult simplificată prin intermediul încărcării și procesării unui fișier `xml` sau `csv`. În final, hostarea acestei aplicații utlizând servicii cloud precum cele ale Google sau Amazon conferă scalabilitate și accesibilitate.

Bibliografie

[1] https://www.ngdevelop.tech/wp-content/uploads/2017/12/Angular_Architecture.png.

[2] https://miro.medium.com/max/1400/1*Y6A_rS5IdRDUG4KRIFP_fA.png.

[3] 31. working with sql databases. https://docs.spring.io/spring-boot/docs/2.1.13.RELEASE/reference/html/boot-features-sql.html.

[4] Angular - building a template-driven form. https://angular.io/guide/forms.

[5] Angular - component. https://angular.io/api/core/Component.

[6] Angular - lifecycle hooks. https://angular.io/guide/lifecycle-hooks.

[7] Angular - using angular routes in a single-page application. https://angular.io/guide/router-tutorial.

[8] Angular - what is angular? https://angular.io/guide/what-is-angular.

[9] Getting started — spring security and angular. https://spring.io/guides/tutorials/spring-security-and-angular-js/.

[10] The history of angular. the past, present, and future of... — by dave gavigan — the startup lab — medium. https://medium.com/the-startup-lab-blog/the-history-of-angular-3e36f7e828c7.

[11] How to design a web application: Software architecture 101. https://www.educative.io/blog/how-to-design-a-web-application-software-architecture-101.

[12] How to use angular development in 2022. https://www.moveoapps.com/blog/how-to-use-angular-development/.

[13] Introduction to spring data jpa — baeldung. https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa.

[14] Json web token introduction - jwt.io. https://jwt.io/introduction.

[15] Key advantages and disadvantages of using spring boot. https://scand.com/company/blog/pros-and-cons-of-using-spring-boot/.

[16] Learn hibernate tutorial - javatpoint. https://www.javatpoint.com/hibernate-tutorial.

[17] Spring boot - code structure - geeksforgeeks. https://www.geeksforgeeks.org/spring-boot-code-structure/.

[18] Spring boot - dependency management - geeksforgeeks. https://www.geeksforgeeks.org/spring-boot-dependency-management/.

[19] Spring boot @springbootapplication, springapplication class — digitalocean. https://www.digitalocean.com/community/tutorials/springbootapplication-springapplication.

[20] Spring initializr. https://start.spring.io/.

[21] What is a rest api? https://www.redhat.com/en/topics/api/what-is-a-rest-api.

[22] What is an api endpoint? — smartbear software resources. https://smartbear.com/learn/performance-monitoring/api-endpoints/.

[23] What is hibernate? definition from theserverside. https://www.theserverside.com/definition/Hibernate.

[24] What is mysql? — oracle. https://www.oracle.com/mysql/what-is-mysql/.

[25] When to use sql vs. nosql. https://integrant.com/blog/when-to-use-sql-vs-nosql/.

[26] Ulrich Bauer. Assignment problem with constraints. Master's thesis, Technische Universität München Fakultät für Informatik, 2005.

[27] Dimitri P. Bertsekas. Auction algorithms. https://web.mit.edu/dimitrib/www/Auction_Encycl.pdf.