UNIVERSITATEA „ALEXANDRU IOAN CUZA” IAȘI
FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Aplicații ale problemelor de tip Stable Matching

propusă de:

Student: Boca Ioan-Bogdan
Coordonator științific: Lect. dr. Frăsinaru Cristian

Sesiunea: iulie 2017

Declarație privind originalitatea și respectarea drepturilor de autor

Prin prezenta declar că Lucrarea de licenţă cu titlul "Aplicații ale problemelor de tip Stable Matching" este scrisă de mine și nu a mai fost prezentată niciodată la o altă facultate sau instituție de învățământ superior din țară sau din străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- toate fragmentele de text reproduse exact, chiar în traducere proprie din altă limbă, sunt scrise între ghilimele și dețin referința precisă a sursei;
- reformularea în cuvinte proprii a textelor scrise de către alți autori deține referința precisă;
- codul sursă, imagini etc. preluate din proiecte open-source sau alte surse sunt utilizate cu respectarea drepturilor de auton și dețin referințe precise;
- rezumarea ideilor altor autori precizează referința precisă la textul original.

Iaşi, 3 iulie 2018 Absolvent Boca Ioan-Bogdan

(semnătură în original)

Declarație de consimţământ

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul "Aplicații ale problemelor de tip Stable-Matching", 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 sa utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iași, 3 iulie

Absolvent Boca Ioan-Bogdan

_____

(semnătură în original)

Cuprins

Introducere

1.1 Motivație

Problemele de tip *Stable-Matching* și-au găsit aplicări în numeroase domenii, pentru a automatiza probleme reale. Problema de construire a unei distribuții stabile apare in diferite forme: repartizarea medicilor rezidenți către spitale, a copiilor către școli, distribuirea studenților la opționale etc. În lucrarea de față se va prezenta o aplicație a unui algoritm de tip *Stable-Matching*, cadrul de lucru fiind modelarea și rezolvarea problemei înscrierii studenților la proiectele de licență.

O problemă de tip *Stable-Matching* este definită în mod clasic folosind două grupuri de entități: bărbați (**M**) și femei (**W**), fiecare individ având o listă de preferințe cu corespondenți din setul opus. Se pune astfel problema găsirii unei repartiții stabile între cele două grupuri. Stabilitatea este dată de inexistența a două elemente care s-ar prefera reciproc față de cei cu care au fost asignați.

În lucrarea "College admissions and the stability of marriage" apărută în 1962, Gale și Shapley au demonstrat că având listele de preferințe ale fiecărui element, se poate construi o repartiție stabilă, ba mai mult, au oferit și un algoritm care realizează acest lucru.

Prezenta lucrare este structurată în trei capitole după cum urmează:

- primul capitol conține noțiuni teoretice despre problemele de tip *Stable-Matching*, precum și descrierea algoritmului utilizat în rezolvarea problemei
- al doilea capitol descrie arhitectura aplicației realizate și felul în care au fost alese și utilizate tehnologiile utilizate

- ultimul capitol conține o perspectivă asupra aplicației, precum și extensiile ce ar putea constitui o direcție pe viitor pentru imbunătățirea procesul de repartiție al studenților către proiecte de licență

1.2 Contribuții

În multe universități, studenții în an terminal trebuie să realizeze un proiect sub coordonarea unui profesor pentru a-și finaliza studiile. Fiecare student are preferințe asupra proiectelor pe care le găsește ca fiind acceptabile, dar și profesorii au preferințe asupra studenților (de exemplu: o medie a studentului cât mai mare).

De la această idee a pornit și acest proiect. Se dorește a se automatiza procesul de înscriere și repartiție a studenților la proiecte de licență. În acest scop s-a realizat o aplicație Web modernă, scalabilă.

Stable-Matching

Varianta inițială a problemei propuse de Gale și Shapley, numită si *Stable-Marriage* se caracterizează prin aceeași dimensiune a celor două seturi de entități. Datorită faptului că aceasta problemă a castigat popularitate, au apărut numeroase variații ale versiunii inițiale. Astfel, problemele de tip *Stable Matching* se împart in trei categorii:

- *one-to-one:* (ex: Stable-Marriage)
- *one-to-many:* (ex: Student-Project Allocation problem, Stable-Admissions problem, Hospital-Residents problem)
- *many-to-many:* (ex: Two-sided Market)

2.1 Preliminarii

Pentru a introduce noțiunile ce stau la baza algoritmilor de tip *Stable-Matching*, pentru început, mă voi folosi de modelul clasic *stable matching*, numit in continuare **SM**. O instanță a **SM** constă în același număr n de bărbați (**M**) și femei (**W**).
M=$\{m_1, m_2, m_3 \ldots, m_n\}$, **W**=$\{w_1, w_2, w_3 \ldots, w_n\}$
Fiecare persoană are o listă de *preferințe* care ordonează strict toți membrii din setul opus. Dacă în lista de preferințe a unui bărbat *m w1* se află înaintea lui *w2*, vom nota acest lucru cu *w1* $\succ_m$ *w2*. O notație similară este folosită și în cazul preferințelor femeilor. O repartiție M este un set disjunct de perechi bărbat-femeie. Dacă într-o repartiție M un bărbat m este asociat unei femei w scriem că *M(m)* $= w$ și *M(w)* $= m$. Spunem că un bărbat m și o femeie w formează o pereche *blocantă* dacă următoarele condiții sunt îndeplinite:

- *M(m)* $\neq w$
- $w \succ_m$ *M(m)*

FIGURE 2.1: O instanță a **SM** cu n = 4

Bărbați	**Preferințe**
m_1	m_2 m_1 m_4 m_3
m_2	m_4 m_3 m_1 m_2
m_3	m_1 m_4 m_3 m_2
m_4	m_2 m_1 m_4 m_3

Femei	**Preferințe**
w_1	w_2 w_4 w_1 w_3
w_2	w_3 w_1 w_4 w_2
w_3	w_2 w_3 w_1 w_4
w_4	w_4 w_1 w_3 w_2

- $m \succ_w M(w)$

O repartiție este numită instabilă dacă există o pereche blocantă, iar in caz contrar este stabilă.

Exemplu: Considerăm o instanță a problemei **SM** cu $n = 4$ specificată prin listele de preferințe din *Figura 2.1.* Repartiția $\{(m_1, w_4), (m_2, w_3), (m_3, w_2), (m_4, w_1)\}$ este stabilă. Stabilitatea poate fi verificată considerând pe rând fiecare bărbat ca fiind un potențial membru al unei perechi blocante. m_1 ar putea forma o pereche blocantă cu w_2, însă $m_3 \succ_{w_2} m_1$. Fiecare din m_1 și m_2 sunt repartizați către prima lor preferință, drept urmare nu pot forma o pereche blocantă. În final, m_4 poate forma o pereche blocantă doar cu w_4, însă $m_1 \succ_{w_4} m_4$.

Această instanță a **SM** admite două repartizări stabile, cea de-a doua fiind $\{(m_1, w_4)$, $(m_2, w_1), (m_3, w_2), (m_4, w_3)\}$.Stabilitatea poate fi verificată într-o manieră asemănătoare cu cea de mai sus. Pe de altă parte, repartiția $\{(m_1, w_1), (m_2, w_3), (m_3, w_2), (m_4, w_4)\}$ este instabilă deoarece (m_1, w_4) formează o pereche blocantă: $w_4 \succ_{m_1} w_1$ și $m_1 \succ_{w_4}$ m_4. Alte repartiții instabile pot avea mai multe perechi blocante: de exemplu, repartiția $\{(m_1, w_1), (m_2, w_2), (m_3, w_4), (m_4, w_3)\}$ are șase perechi blocante.

2.2 Algoritmul lui Gale Shapley

În 1962, Gale și Shapley au prezentat un algoritm care rezolvă problema **SM** demonstrând că orice pentru orice instanță a problemei se poate construi o repartiție stabilă. Algoritmul utilizează trei variabile, k, X, si x și două constante n și Ω.

- n: numărul de femei = numărul de bărbați
- k: numărul de cupluri deja formate
- X: pretendentul
- x: femeia căruia pretendentul ii face propunerea
- Ω: bărbat imaginar (va fi adăugat la finalul listei de preferințe al fiecărei femei)

Algorithm 1 Gale Shapley Algorithm

```
1: k ← 0; all the women are (temporarily) engaged to Ω;
2: while k < n do
3:     X ← (k+1)-st man;
4:     while X ≠ Ω do
5:         x ← best choice remaining on X's list;
6:         if x prefers X to her fiance then
7:             engage X and x
8:             X ← preceding fiance of x
9:         if X ≠ Ω then
10:            widthdraw x from X's list
11:    k ← k + 1
12: celebrate n weddings
```

Descrierea algoritmului:

- La începutul algoritmului numărul de cupluri deja formate este inițializat cu 0, iar fiecare femeie este asociată temporar cu Ω
- Cât timp numărul de cupluri formate este mai mic decât n se fac următoarele operații:
  - Se alege X ca fiind al *k+1*-lea bărbat
  - Cât timp $x \neq \Omega$
    - Se alege x ca fiind prima prima preferință din lista lui X
    - În cazul în care x preferă pe X mai mult decât pe partenerul curent atunci:
      - *(X, x)* vor forma un cuplu
      - X devine fostul partener al lui X
      - Se șterge x din lista de preferințe a lui X
      - Se reintră în bucla *while* interioară, urmând ca fostul partener al lui x să își găsească cu nouă parteneră
  - Se trece la următorul bărbat
- La finalul buclei while exterioare vom avea formate n cupluri.

Algoritmul are ca și principiu adresarea propunerilor în funcție de ordinea elementelor în listele de preferință. Aceste propuneri se fac dintr-un singur sens; spre exemplu dacă din setul **M** se fac propuneri către setul **W** spunem că M este *men-optional*, însemnând că orice bărbat va forma o pereche cu femeia aflată cea mai sus în lista sa de preferințe pe care ar putea sa o obțină din **W**.

Corectitudinea algoritmului:
Observația 1: Odată ce *(m,w)* formează un cuplu w poate forma un cuplu cu m' doar dacă $m' \succ_w m$, altfel rămâne cu m. Cu alte cuvinte, odată ce o femeie e căsătorită aceasta nu va mai fi liberă.
Observația 2: Bărbații propun femeilor aflate din ce în ce mai jos în lista lor de preferințe.
Vom demonstra, prin *reducere la absurd* că algoritmul lui Gale Shapley produce o repartiție M perfectă [1]

• **Presupunere** Presupunem că M nu este o repartiție perfectă.

M nefiind o repartiție perfectă, înseamnă că există un bărbat m ce rămâne nerepartizat la finalul algoritmului. Deoarece algoritmul ia bărbații necăsătoriți în ordine, iar aceștia propun femeilor din lista lor de preferințe cărora nu le-au propus încă, rezultă că m a propus tuturor femeilor și a rămas fără parteneră. Acest lucru implică faptul că m a fost refuzat sau părăsit de toate cele n femei. O femeie w poate refuza un bărbat m_1 doar dacă este căsătorită cu m_2 și $m_2 \succ m_1$ sau dacă m_2 îi propune lui w și are loc aceeași relație. Din acest lucru rezultă că fiecare femeie a fost asociată cu un bărbat și conform ***Observației 1***, acestea nu mai sunt libere. În concluzie, avem *(n-1)* bărbați căsătoriți cu n femei.

• **Contradicție** Avem n femei căsătorite cu *n-1* bărbați.

Vom demonstra că algoritmul **GS** produce o repartiție M stabilă.

• **Presupunere** Presupunem că avem o pereche blocantă *(m, w)* în *M*. Fie $M(w)=m_1$ și $M(m)=w_1$.

Folosind presupunerea, observăm că m i-a propus lui w. Acest lucru rezultă din faptul că m formează un cuplu cu w_1, care se află mai jos în lista sa de preferințe. Singura cale prin care m ar fi putut să îi propună lui w_1 ar fi fost să propună tuturor femeilor aflate înaintea acesteia, deci și lui w. O propunere poate avea două răspunsuri: **da** sau **nu**. Vom arăta că ambele răspunsuri duc la contradicție.
Da: În acest caz, perechea *(m, w)* a fost în *M*. Dar cum $(m, w) \notin M$, rezultă că w a părăsit m pentru un partener mai favorabil. Folosind presupunerea și definiția unei perechi blocante rezultă că, la un moment dat, în timpul rulării algoritmului w s-a asociat cu un partener mai puțin favorabil decât partenerul curent. (*Contradicție*)

[1] de dimensiune n

Nu: Pentru ca w să refuze pe m, w ar fi trebuit să fie asignată cu un m_2 mai favorabil. Ținând cont de faptul că w formează o pereche cu m_1 în M și $m \succ_w m_1$, prin tranzitivitate rezultă și că $m \succ_w m_2$. (*Contradicție*, explicația fiind aceeași ca și în cazul anterior)

- **Contradicție** Fie m nu i-a propus lui w, fie w a părăsit m pentru un partener mai nefavorabil, ambele contrazicând algoritmul.

Complexitatea timp/spațiu a algoritmului este ordinul $0(n^2)$.

2.3 Student-Project Allocation

În multe universități din lume, studenții în an terminal sunt nevoiți să realizeze un proiect dintr-un anumit domeniu pentru a-și finaliza studiile. De obicei sunt propuse o gamă largă de proiecte, din care studenții au de ales. De multe ori, un profesor poate propune mai multe proiecte, cu toate că este probabil ca unele proiecte să nu fie alese.

Fiecare student are preferințe asupra proiectelor pe care dorește să le realizeze (*ex:* un student dorește o colaborare cu un anumit profesor, sau dorește să realizeze un proiect dintr-un domeniu ce l-a atras în timpul facultății). De asemenea, și profesorii au preferințe asupra studenților pe care îi coordonează (*ex:* medie a studentului cât mai mare).

Există anumite limite superioare: fiecare proiect are o anumită capacitate, care reprezintă numărul maxim de studenți ce pot realiza acel proiect, și fiecare profesor poate avea un număr maxim de studenți pe care îi poate superviza într-o sesiune de licență.

Ne vom referi în continuare la problema asignării studenților la proiecte, prescurtată **SPA**. Datorită faptului că în acest proces sunt implicați numeroși studenți există un interes mare în a automatiza acest proces folosindu-se scheme centralizate de repartiție. Asemenea scheme de repartiție se folosesc deja în următoarele instituții de învățământ: University of York, University of Southampton, etc.

SPA este o generalizare a clasicii probleme a repartizării medicilor rezidenți către spitale, care are aplicații într-una din cele mai mare scheme de repartiție din lume: *National Resident Matching Program* (**NRMP**). **NRMP** este dat în funcțiune încă din 1962, și în momentul de față se ocupă anual de repartiția a aproximativ 30000 de studenți absolvenți către spitale. Repartiția se face pe baza preferințelor medicilor rezidenți, dar și pe baza locurilor disponibile în spitale și a preferințelor spitalelor. Algoritmul găsește o repartizare stabilă, care este *resident-optimal*.

2.3.1 Definirea problemei

O instanță a **SPA** se definește prin **S**=$\{s_1, s_2, s_3 \ldots, s_n\}$ o mulțime de studenți, **P**=$\{p_1, p_2, p_3 \ldots, p_m\}$ o mulțime de proiecte și **L**=$\{l_1, l_2, l_3 \ldots, l_k\}$ o mulțime de profesori. Fiecare student s_i își formează o listă de preferințe, ordonând o submulțime a **P** în ordine strictă. Dacă proiectul p_j apare în lista de preferințe a lui s_i, spunem că proiectul p_j este acceptabil pentru studentul s_i. Vom nota cu A_i mulțimea proiectelor pe care studentul s_i le găsește ca acceptabile.

Fiecare profesor l_k oferă un set nevid de proiecte P_k. l_k are asociată o listă de preferințe în care sunt ordonați strict studenții care găsesc acceptabil un proiect propus de profesorul l_k. De asemenea l_k are asociat un număr întreg pozitiv, numit *capacitate*, reprezentând numărul de studenți pe care dorește să îi coordoneze. Similar, fiecare proiect are asociată o constrângere de *capacitate* ce indică numărul maxim de studenți ce pot realiza acel proiect.

O repartiție M este o submulțime a S × P astfel încât:

- $(s_i, p_j) \in M$ a.î. $p_j \in A_i$ (s_i este asignat unui proiect pe care îl găsește acceptabil)
- Pentru fiecare student $s_i \in S$ este îndeplinită relația $|\{(s_i, p_j) \in M : p_j \in P\}| \leq 1$. (un student este asignat maxim unui proiect)

Notații:

- $\forall s_i \in S$ prin $M(s_i)$ notăm proiectul p_j cu care acesta a fost asignat. O notație asemănătoare va fi folosită și pentru proiecte.
- Spunem despre un proiect p_j că este under-subscribed, full sau over-subscribed dacă $|\ M(p_j)\ |$ este mai mic, mai mare sau egal în raport cu capacitatea acestuia.
- Prin $M(l_k)$ indicăm mulțimea studenților care au fost asignați unui proiect propus de l_k.
- Un profesor este under-subscribed, full sau over-subscribed considerând $|\ M(l_k)\ |$ în raport cu capacitatea acestuia.

Spre deosebire de *Stable-Marriage*, în *SPA*, listele de preferință nu trebuie să conțină toate elementele din setul opus, iar la terminarea algoritmului nu este garantat că fiecare student/proiect va avea un corespondent din setul opus.

Vom defini și în cazul **SPA** noțiunea de *pereche blocantă*. O pereche $(s_i, p_j) \in (S \times P) \setminus M$ este blocantă dacă:

FIGURE 2.2: O instanță **SPA**

Studenți	**Preferințe**
s_1	p_1 p_3
s_2	p_1 p_2 p_3
s_3	p_1 p_4
s_4	p_1

Profesori	**Preferințe**
l_1	s_1 s_2 s_3 s_4
l_2	s_2 s_1 s_3

l_1 oferă p_1, p_2. l_2 oferă p_3, p_4
Capacitatea proiectelor: c_i=1
Capacitatea profesorilor: d_1=1, d_2=2

- $p_j \in A_i$.
- Fie s_i nu este asignat în M, fie s_i preferă p_j în detrimentul proiectului cu care este deja asignat.
- Fie
  - p_j este under-subscribed si l_k este under-subscribed, sau
  - p_j este under-subscribed, l_k este full, și fie $s_i \in M(l_k)$ fie l_k preferă s_i în detrimentul ultimului student din $M(l_k)$, sau
  - p_j este full și l_k preferă pe s_i în detrimentul ultimului student din $M(p_j)$

2.3.2 Algoritmul utilizat

Dat fiind faptul că există un interes ridicat pentru această problemă, ea a fost studiată de-a lungul timpului, și s-a găsit un algoritm care să o rezolve. În această secțiune voi prezenta algoritmul utilizat în rezolvarea problemei și cum a fost aplicat în cazul concret.

Algoritmul are la bază două operații fundamentale: cea de aplicare și cea de ștergere. Operația de aplicare este asemanatoare cu operația de propunere din cadrul Stable-Marriage. Prin operația de ștergere se înțelege ștergerea unei preferințe p_j din lista de preferințe a lui s_i.

Inițial, studenții nu au nici un proiect asignat. Cât timp există un student s_i care nu are un proiect asignat și are o listă de preferințe de lungime mai mare decât zero se va intra în bucla *while* interioară. Se va crea o asociere temporară între studentul s_i și primul proiect din lista sa de preferințe, p_j. Fie l_k profesorul care a propus acel proiect.

Dacă p_j este *over-subscribed*, se va rupe asocierea temporară dintre p_j și ultimul student student asociat la acest proiect în raport cu lista de preferințe a lui l_k. În cazul în care l_k este over-subscribed se va căuta studentul s_r printre studenții asignați la un proiect propus de l_k. s_r va fi ultimul student în raport cu lista de preferințe a lui l_k. Se va rupe asocierea dintre s_r și proiectul său asociat.

Algorithm 2 Project Allocation Algorithm

```
1: assign each student to be free;
2: assign each project and lecturer to be totally unsubscribed ;
3: while some student s_i is free and s_i has non-empty list do
4:     p_j ← first project on s_i's list;
5:     l_k ← lecturer who offers p_j;
6:     M = M ∪ (s_i, p_j)
7:     if p_j is over-subscribed then
8:         s_r ← worst student assigned to p_j
9:         M = M \ (s_r, p_j);
10:    else if l_k is over-subscribed then
11:        s_r ← worst student assigned to l_k;
12:        p_t ← project assigned to s_r;
13:        M = M \ (s_r, p_t);
14:    if p_j is full then
15:        s_r ← worst student assigned to l_k;
16:        for each succesor s_t of s_r on L_k do
17:            for each project p_u ∈ P_k ∩ A_t do
18:                delete (s_t, p_u);
19: return M
```

În cazul în care proiectul p_j este full, se va șterge proiectul p_j din lista de preferințe a studenților aflați după ultimul student asignat la p_j în raport cu lista de preferințe a lui l_k. Astfel, acești studenți nu vor mai avea oportunitatea de a aplica la acest proiect.

În cazul în care l_k este full, se va aplica un procedeu similar de ștergere.

Complexitatea timp a algoritmului depinde de cum sunt implementate operațiile de aplicare și de ștergere. Aceste operații pot fi implementate în timp constant, astfel complexitatea SPA devine $\Theta(\lambda)$, unde λ este lungimea totală a listelor de preferințe. Complexitatea spațiu este de ordinul a $0(mn)$, unde n este numărul de studenți, iar m este numărul de proiecte dintr-o instanță SPA.

Construcția lui M pentru instanța **SPA** din ***Figura 2.2***

Etapa de inițializare:

- $M = \emptyset$
- s_1, s_2, s_3, s_4 = free
- $l_1, l_2, p_1, p_2, p_3, p_4$ = *under-subscribed*

Bucla *while* în care se vor realiza asocierile:

- s_1 aplică la prima sa preferință, p_1. $M = \{(s_1, p_1)\}$

- p_1 este *full*. Ștergem din lista lui s_2, s_3, s_4 pe p_1 deoarece aceștia se află după s_1 în lista de preferințe a lui l_1.
- l_1 este full. Ștergem din lista lui s_2 proiectul p_2.
- s_2 aplică la p_3. $M = \{(s_1, p_1), (s_2, p_2)\}$
- p_3 este full. Se aplică operația de ștergere pe (s_1, p_3).
- s_3 aplică la p_4. $M = \{(s_1, p_1), (s_2, p_2), (s_3, p_4)\}$
- Atât p_4 cât si l_2 sunt full, însă nu se mai operații de ștergere.

În final, se va returna M ca fiind $M = \{(s_1, p_1), (s_2, p_2), (s_3, p_4)\}$. Se observă că s_4 nu a fost repartizat nici unui proiect.

În contextul aplicației Web realizate, algoritmul folosit este SPA, cu mici modificări. În primul rând s-a eliminat constrângerea de capacitate pentru profesori. Astfel, numărul maxim de studenți pe care un profesor dorește să îi coordoneze este egal cu suma capacităților proiectelor propuse de el. Listele de preferință din partea profesorilor s-au mutat către proiecte. Astfel, fiecare proiect are propria sa listă de preferințe, cu studenții ce au aplicat la acest proiect. În acest mod, un profesor poate favoriza un student doar în contextul unui anumit proiect, mutându-l mai sus în lista de preferințe.

Arhitectură și implementare

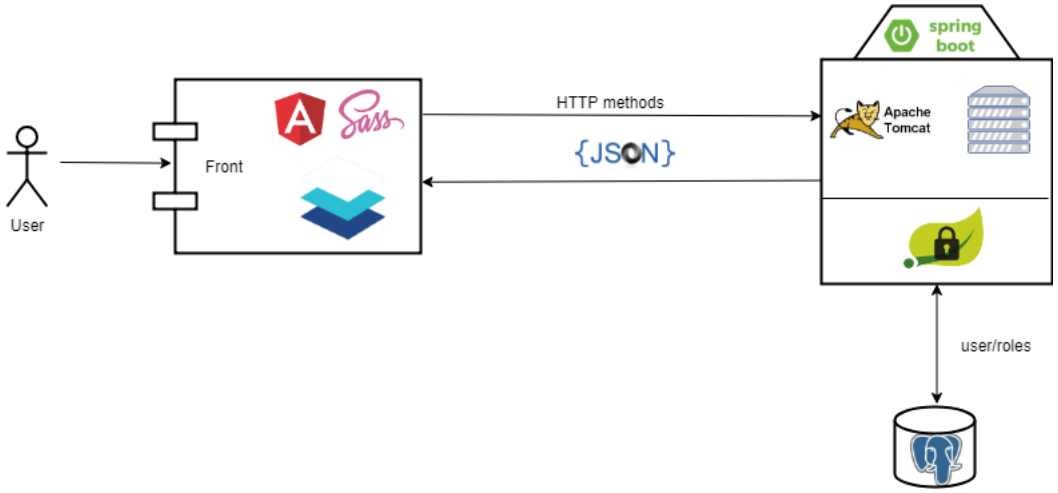
În acest capitol se va discuta despre arhitectura proiectului cât și despre detaliile de implementare ale aplicației realizate.

Aplicația este separată în două proiecte:

- cel de front-end, construit cu Angular 5
- cel de back-end, realizat în Java

Pentru a oferi o imagine de ansamblu a aplicației realizate, în figura 4.1 este prezentată arhitectura generală a proiectului.

FIGURE 3.1: Arhitectura generală a proiectului



3.1 Specificații funcționale Aplicații ale problemelor de tip Stable Matching

Nume alternativ: **FII License Matcher**
Tip Aplicație: **Web**
Domeniu de activitate: **Educație**

Descriere generala
Pentru a-și finaliza studiile, studenții în an terminal sunt nevoiți să realizeze un proiect sub îndrumarea unui profesor. Datorită faptului că în acest proces este implicat un număr mare de studenți, se pune problema automatizării acestuia. În acest scop, s-a realizat o aplicație Web. Aplicaţia va înlesni procesul de înscriere a studenților către proiecte de licență și repartizarea acestora.

Obiectivele aplicației:

- Managementul înscrierilor la licență: importarea studenților, a notelor, a cursurilor și a profesorilor, stabilirea perioadelor în care este posibilă aplicarea la proiecte
- Definirea proiectelor: Baze de date de tip graf, Algoritmi evolutivi pentru rețele de sortare, etc.
- Definirea preferințelor: Se vor forma liste de preferințe pentru fiecare student/proiect
- Vizualizarea unor statistici
- Vizualizarea situatiei anilor trecuți
- Repartizarea proiectelor către studenți: se va realiza o potrivire între studenți și proiecte

Scenarii de utilizare:
Pentru a utiliza aplicația, utilizatorul va trebui să introducă e-mailul si parola. În cazul în care există un cont asociat cu informațiile introduse, acesta va putea accesa funcționalitățile aplicației. Fiecare rol are asociat un set de acțiuni pe care le poate efectua in aplicație. În continuare, vor fi descrise aceste roluri și acțiunile posibile.

Profesor: Acțiunea fundamentală pe care un profesor o poate realiza este cea de a adăuga proiecte. Pentru a adăuga un proiect, un profesor va trebui sa completeze următoarele câmpuri: titlul proiectului, descrierea și capacitatea acestuia. Fără aceste

informații completate, proiectul nu va putea fi adăugat. Pe lângă acestea, se vor putea introduce și următoarele informații suplimentare: etichete (ce vor fi folosite pentru căutarea proiectelor), referințe bibliografice, un set de întrebări (la care studentul va trebui să răspundă în momentul aplicării la proiect) și este posibilă și încărcarea unui fișier.

De asemenea, un profesor poate edita sau șterge proiecte propuse de el și poate vizualiza anumite statistici (numărul de studenți înscriși la un proiect, media studenților care au aplicat la un proiect, și media poziției unui proiect în lista de preferințe a studenților).

Pentru fiecare proiect propus, profesorul va putea vizualiza lista de studenți care au aplicat pentru acel proiect. Această listă este sortată inițial după media generală a notelor studenților. Este posibil ca prin drag and drop să se reordoneze lista, astfel unii studenți putând fi favorizați. Pentru fiecare student care a aplicat, se vor putea vizualiza informații suplimentare, cum ar fi notele studentului, răspunsurile pe care le-a oferit în momentul aplicării și nota personală.

Proiectele propuse de către profesori în anii trecuți vor putea fi importate și în anul curent, evitând astfel completarea formularului în cazul în care se dorește refolosirea unui proiect.

Student: Un student are la dispoziție o listă de proiecte din care își poate forma o listă cu preferințe aplicând la acestea. Un student poate aplica la un proiect o singură dată. În cazul în care un proiect are asociate anumite întrebări, studentul va fi nevoit să răspundă la ele în momentul aplicării. De asemenea, studentul poate atașa și o notă personală în momentul aplicării.

Aplicând la diverse proiecte, studentul își va forma o listă de preferințe. Această lista va fi sortată inițial după data aplicării, însă va putea fi reordonată prin drag and drop. De asemenea, un student își poate retrage aplicarea la proiectele din lista sa.

Pentru a facilita căutarea de proiecte, acestea pot fi căutate după titlul proiectului, după etichete sau dupa numele profesorului.

Administrator: Un administrator se ocupă de buna funcționare a aplicației. Acțiunile pe care le poate realiza sunt organizatorice. Astfel, un admin poate stabili perioadele în care înscrierile la proiecte sunt posibile, poate pregăti aplicația pentru a nouă sesiune de înscriere la licență prin: importarea în aplicație a profesorilor, a cursurilor, a studenților și a notelor studenților.

Tot un administrator poate vizualiza lista tuturor utilizatorilor aplicației și poate șterge conturi dacă e necesar.

Istoricul repartizărilor din fiecare an este salvat, și va putea fi vizualizat. Se va alege un an, și un fișier de tip pdf cu numele studenților și ce proiecte au realizat va fi descărcat. Această funcționalitate este disponibilă doar administratorilor.

Preluarea datelor

Datele vor fi încărcate de către un admin în aplicație din panoul de admin. Fișierele încarcăte vor trebui sa fie în formatul .csv și vor trebui sa respecte un anumit format. În continuare voi prezenta formatul acestor fișiere.

Fișierul ce conține informații despre studenți va trebui să aibă următoarea structură:

- pe prima coloană anul studentului
- pe a doua coloană numele și prenumele acestuia
- urmată de numărul matricol
- iar la final adresa de e-mail a acestuia

Exemplu minimal de asemenea fișier:

```
2,"ION V. POPESCU","30000004SL000000","ion.popescu@info.uaic.ro"
1,"MARCELONA A. PASTRAMA","30000004SL000001","marcelona.pastrama@info.uaic.
    ro"
```

Fișierul ce conține informații despre profesori va trebui să aibă următoarea structură:

- pe prima coloană numele şi prenumele acestuia
- pe a doua coloană adresa de e-mail a acestuia

Un exemplu minimal de fișier:

```
"Brancoveanu Costel","c.brancoveanu@info.uaic.ro"
"Albu Viorel","a.viorel@info.uaic.ro"
```

În cazul celor două fișiere, pentru fiecare rând din fișier se va crea câte un cont în mod automat. Acesta va avea asociată o parolă random, care va fi trimisă pe adresa de e-mail asociată contului nou creat. Utilizatorul va avea posibilitatea de a-și schimba parola.

```
"CS1101","SD","Structuri de date",1,1
"CS1102","ACSO","Arhitectura calculatoarelor si sisteme de operare",1,1
```

Mai sus, am prezentat un scurt exemplu de fișier ce conține informații despre cursuri. Coloanele au următoarea semnificație:

- prima coloană reprezintă codul asociat cursului
- a doua coloană reprezintă abrevierea cursului (va fi folosită la importul notelor)
- penultima coloană reprezintă anul în care e ținut cursul
- ultima coloană reprezintă semestrul

Nu în ultimul rând, pentru a putea forma listele de preferință inițiale ale proiectelor, avem nevoie de notele studenților. Acestea vor trebui să fie importate printr-un fișier având următoarea structură:

- pe prima coloană numărul matricol al studentului
- pe a doua coloană prescurtarea materiei la care s-a obținut nota
- pe ultima coloană valoarea notei

```
1 "30000004SL000000","SD",10
2 "30000004SL000000","ACSO",9
```

Alte observații
Aplicația este internaționalizată, cu două localizări: română și engleză.

3.2 Arhitectura aplicației

Aplicația este separată în două proiecte (unul de back-end și unul de front-end). S-a ales această separare deoarece cele două proiecte pot fi dezvoltate independent, astfel am putut lucra doar la o singură parte a aplicației la un moment dat. Atât timp cât timp fiecare parte știe ce format de mesaje să trimită celuilalt, acestea pot fi păstrate modulare și separate. Separând preocupările legate de interfața cu utilizatorul de preocupările legate de lucrul cu datele, îmbunătățim flexibilitatea interfeței între platforme și îmbunătățim scalabilitatea serverului prin simplificarea acestuia.

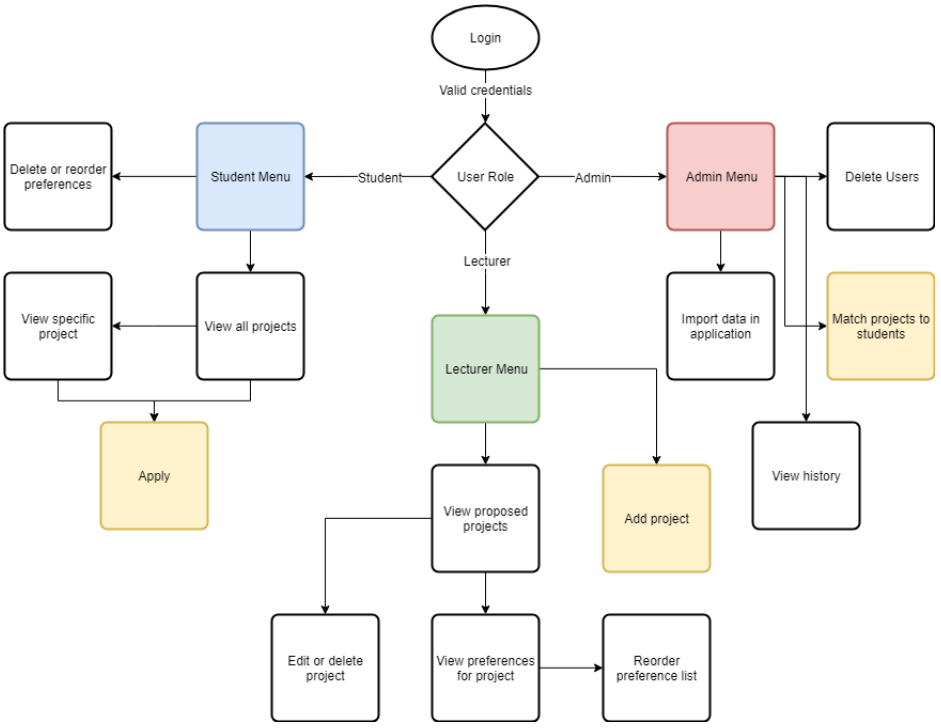


FIGURE 3.2: Flow-ul aplicației

3.2.1 Front-end

Partea de front-end a fost realizată în principal folosind framework-ul Angular.

O aplicație GUI rulează întotdeauna ca o singură aplicație separată pentru fiecare utilizator, astfel încât nu există nici o provocare de tip "număr mare de utilizatori" care este tipică pentru aplicațiile de back-end. În schimb, o aplicație GUI trebuie să se ocupe de următorii factori de scalabilitate: creșterea datelor încărcate în aplicație, creșterea complexității și mărimii proiectului, de obicei urmate de timpi mai mari de încărcare a aplicației.

Pentru a avea o calitate cât mai bună a aplicației, în primul rând avem nevoie de o structură a proiectului cât mai organizată.

În figura de mai sus sunt prezentate principalele componente ale aplicației.

AppModule este modulul de bootstrapping, responsabil pentru lansarea aplicației și combinarea împreuna a altora module.

MaterialModule este creat în scopul organizării și pentru a nu încărca foarte mult *AppModule*. În acest modul vor fi importate toate componentele aparținând librariei **Angular Material** care sunt utilizate în această aplicație.

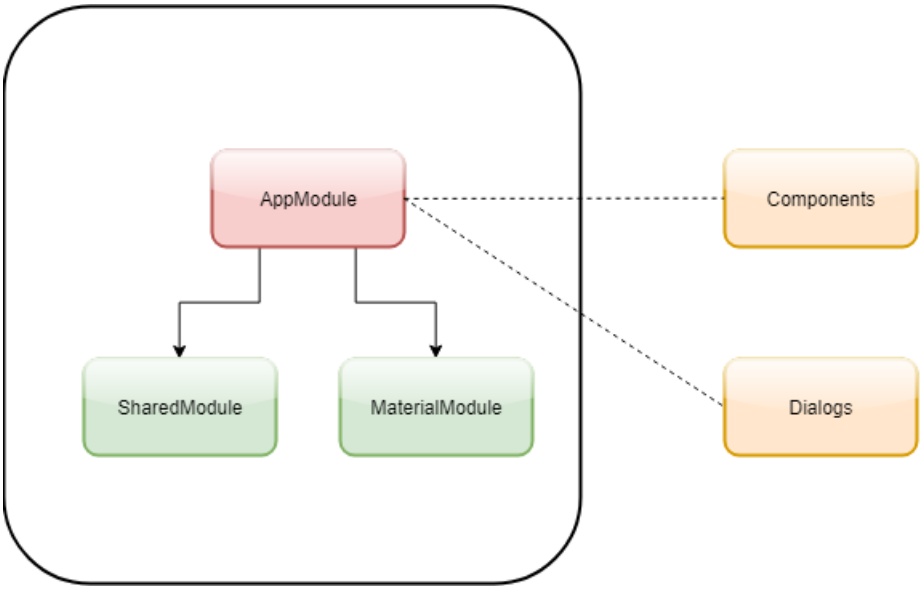


FIGURE 3.3: Structură front-end

SharedModule reprezintă un set de servicii care vor fi refolosite in alte module din aplicație. Printre acestea se numără servicii de autentificare, pipe-uri, interceptori, etc.

Components și *Dialogs* reprezintă elemente vizuale ce vor crea interacțiunea cu utilizatorul.

3.2.2 Back-end

Proiectul de pe partea de back-end respectă pattern-ul Entity-Control-Boundry (**ECB**). **ECB** este o variație a pattern-ului Model-View-Controller. Entity, Control și Boundry sunt stereotipuri de clasă, dar UML are câteva icoane speciale pentru a le reprezenta.

Entity reprezintă obiecte care reprezintă datele sistemului: utilizatori, proiecte, întrebări etc.

Boundry sunt obiectele cu care interfațează cu actorii sistemului: interfețe utilizator, gateway-uri, proxy-uri etc.

Controller sunt obiectele care mediază comunicarea între boundry și entity. Ele orchestrează execuția comenzilor ce vin de la boundry.

De asemena, aplicația este organizată pe mai multe niveluri, respectându-se în același timp si pattern-ul Entity-Control-Boundry. O arhitectură pe mai multe nivele oferă un model prin care dezvoltatorii pot crea aplicații flexibile și reutilizabile. Prin separarea unei aplicații în mai multe niveluri, dezvoltatorii dobândesc opțiunea de a modifica sau adăuga un anumit strat, în loc să refacă întreaga aplicație. O arhitectură este compusă în

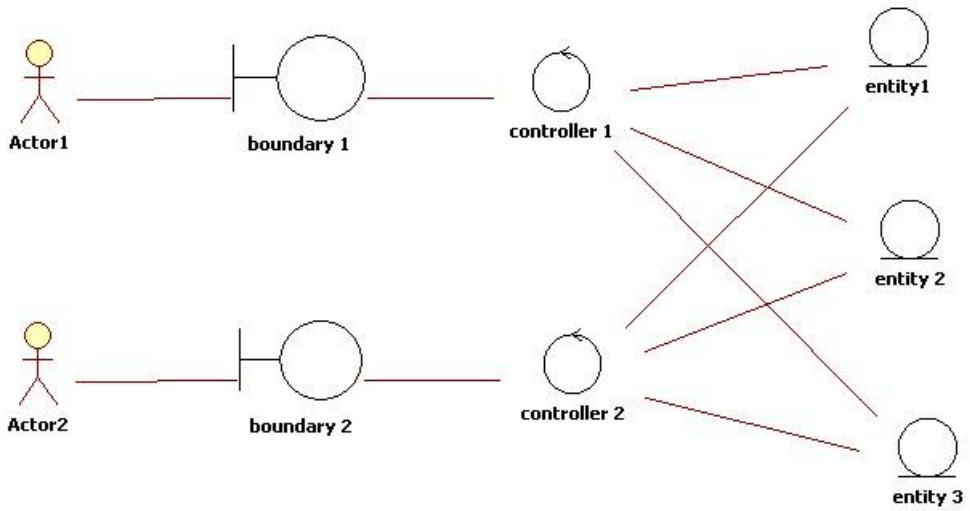


FIGURE 3.4: Entity-Control-Boundry pattern

general din trei niveluri: unul de prezentare, un nivel în care se realizează logica asupra datelor și un nivel de stocare a datelor.

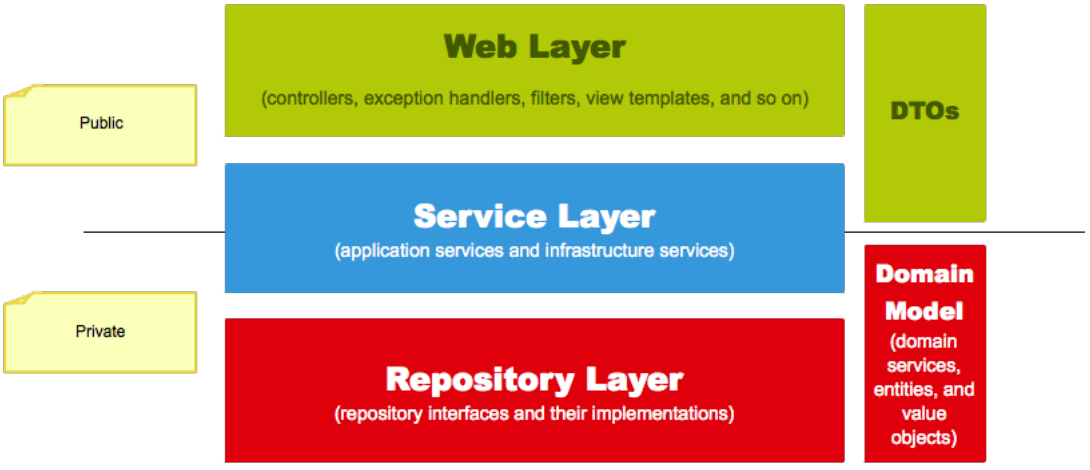


FIGURE 3.5: Arhitectura back-end

Stratul web este stratul superior al aplicației. Acest strat răspunde de preluarea datelor introduse de către utilizator și de returnarea răspunsului către acesta. Stratul superior se ocupă, de asemenea, și de excepțiile apărute la acest nivel sau în nivelele inferioare. Deoarece nivelul web este punctul de intrare în aplicație, tot aici se realizează și procesul de autentificare și autorizare.

Tot la acest nivel apar și așa numitele "obiecte de transfer". Acestea sunt simple obiecte care acționează ca un container de date simplu. Un dto este un obiect care deține doar proprietăți și are getteri și setteri, dar nu are logică de nici o semnificație. Motivația utilizării acestora este aceea că reduce numărul de apeluri (ce sunt costisitoare din punct

de vedere al timpului) ce trebuiesc efectuate la comunicarea între sisteme. Pentru a realiza maparea automata a entităților către dto-uri am utilizat librăria **ModelMapper**.

Stratul de serviciu este următorul strat sub stratul de web. Acționează ca o frontieră între nivelul de web și nivelele inferioare.

Nivelul repository este un nivel superior peste entități. Un repository este un mecanism care încapsulează comportamentul de stocare, recuperare și căutare peste o colecție de obiecte.

Pe stratul cel mai jos vom găsi entitățile. Acestea sunt mapări ale obiectelor din baza de date către obiecte din limbajul de programare folosit. Maparea se realizează de obicei folosinf un framework. Astfel dezvoltatorul este scutit de manipularea manuală și de convesia de obiecte a setului rezultat. În prezenta aplicație, pentru mapare s-a utilizat framework-ul Hibernate.

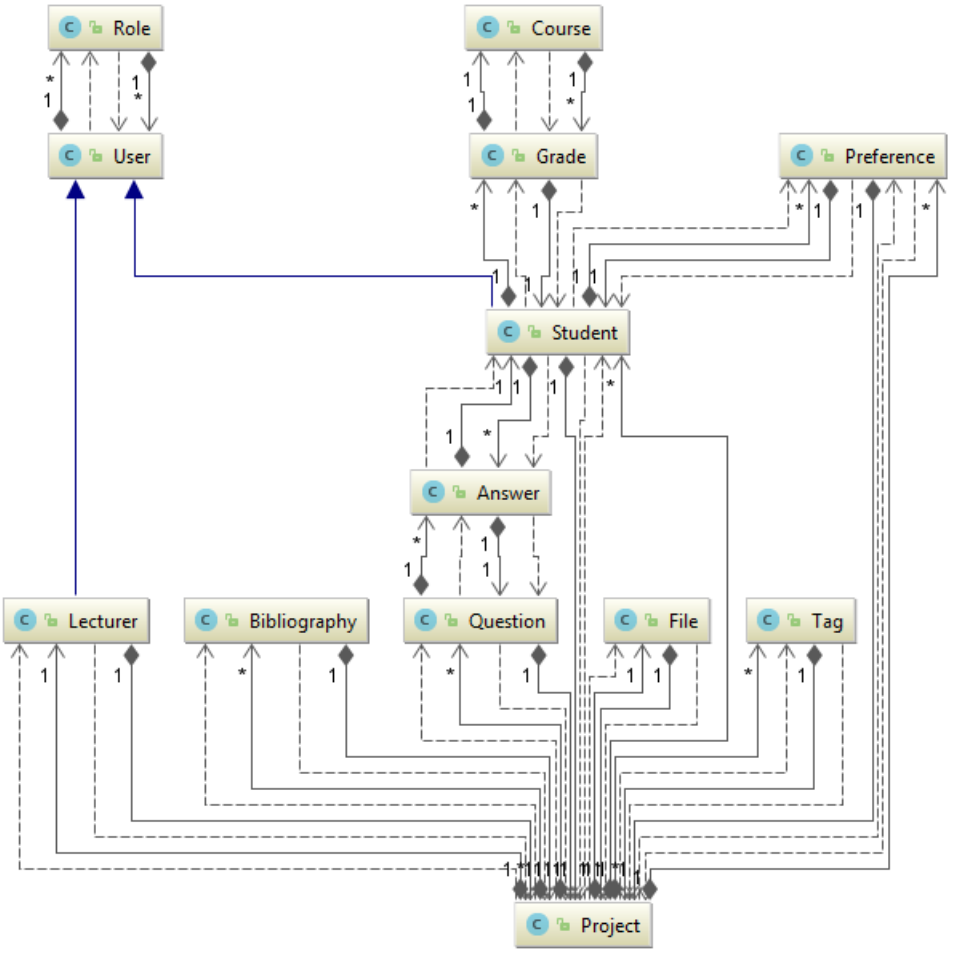


FIGURE 3.6: Diagrama nivelului entităților

Datorită faptului că cele două proiecte sunt independente, este necesar un mod de comunicare între ele. REST vine în ajutor cu acest lucru. Sistemele ce adoptă acest stil sunt numite deseori RESTful, și se caracterizează prin faptul că sunt fără stare și separă preocupările clientului și ale serverului.

```java
    /**
     * Find a project by id
     * @param id the Id to search by
     * @return The requested project
     * @throws ResourceNotFoundException when the requested project doesn't
    exist in the database
     */
     @RequestMapping(value = "/{id}", method = RequestMethod.GET)
     public ResponseEntity<?> getProjectById(@PathVariable Long id) throws
    ResourceNotFoundException {
       Project project = projectService.findById(id);

       if (project == null) {
         throw new ResourceNotFoundException(String.format("Proiectul cu id-ul
    =%s nu a fost gasit!", id));
       }
       return ResponseEntity.ok(modelMapper.map(project, ProjectDto.class)); //
     return 200 with json body

     }

```

(A) Spring Boot REST endpoint

```typescript
    public getProjectById(id: number): Observable<Project> {
      return this.http.get(`${APP_CONSTANTS.ENDPOINT}/projects/${id}`);
    }

```

(B) Client Angular

FIGURE 3.7: Exemplu comunicare back-end cu front-end

Spring oferă un modul care facilitează crearea de servicii REST, reducând drastic cantitatea de cod scris necesar expunerii unui asemenea serviciu.

REST este acronimul pentru Representational State Transfer si reprezintă un model arhitectural pentru crearea serviciilor Web. REST descrie o arhitectură orientată pe resurse. Într-o arhitectură REST se deosebesc două trăsături importante:

- datele asupra cărora clientul îi spune serverului să opereze se află în URI
- operația pe care serverul o face asupra datelor este descrisă direct de metoda HTTP

În figura 3.7 am prezentat un model de comunicare între cele două componente ale proiectului. Serverul expune o anumită informație la o adresă, iar clientul face o cerere la acea adresă folosind o metodă HTTP adecvată.

3.2.3 Persistența datelor

Drept bază de date am ales să folosesc PostgreSQL (versiunea 9.4).

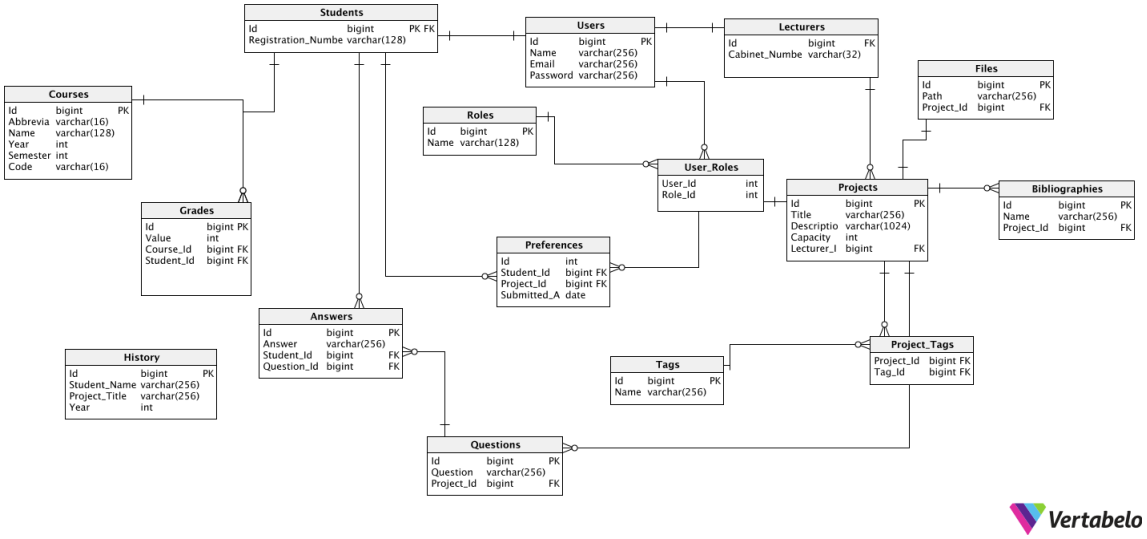


FIGURE 3.8: Arhitectura bazei de date

3.2.4 Repartizarea proiectelor

În urma aplicării studenților la proiecte se formează listele de preferințe. Acestea vor putea fi modificate, pe deoparte de studenți dar și de profesori, formându-se astfel liste de preferințe și din partea studenților dar și a proiectelor.

Cu ajutorul acestor liste și cu ajutorul alogoritmului prezentat în capitolul 1 (modificat, astfel încât să îndeplinească noile cerințele) se vor realiza repartizările. Un administrator va fi responsabil de acest lucru. El are la dispoziție un buton de "previzualizare". Apăsând pe acel buton, un fișier de tip .pdf ce va conține repartizările va fi descărcat. În cazul în care situația este mulțumitoare, acesta va putea apăsa butonul de "salvare" care va repartiza propriu-zis proiectele și va arhiva situația pe anul curent. În urma acestui procedeu, studenții vor putea vedea proiectele la care au fost repartizați, iar profesorii vor vedea studenții cu care vor colabora.

În continuare voi prezenta implementarea, în Java, a algoritmului ce realizează repartizarea proiectelor către studenți.

```
1 public Map<Student, Project> solve() {
2   while (problem.thereIsAFreeStudentWithNoEmptyList()) {
3     Student si = problem.getFreeStudentWithNoEmptyList();
4     Project pj = si.getFirstProject();
5     // provisionally assign si to pj
```

```
6       si.setProject(pj);
7       pj.addStudent(si);
8       if (pj.isOverSubscribed()) {
9         Student sr = pj.getWorstAssignedStudent();
10        // break provisional assignment between sr and pj
11        pj.removeStudent(sr);
12        sr.setProject(null);
13      }
14      if (pj.isFull()) {
15        Student sr = pj.getWorstAssignedStudent();
16            pj.deleteSuccessors(sr);
17      }
18      return problem.getSolution();
19    }
20  }
```

Se observă că s-a renunțat la constrângerea de capacitate pentru profesori. Aceasta devine suma capacităților proiectelor propuse de el. Preferințele din partea profesorilor au fost mutate către proiecte. Lista de preferințe a unui proiect este formată din lista studenților care au aplicat la acel proiect, ordonată în funcție de media generală a studenților, și eventual, modificată manual de către profesor.

Tehnologii utilizate

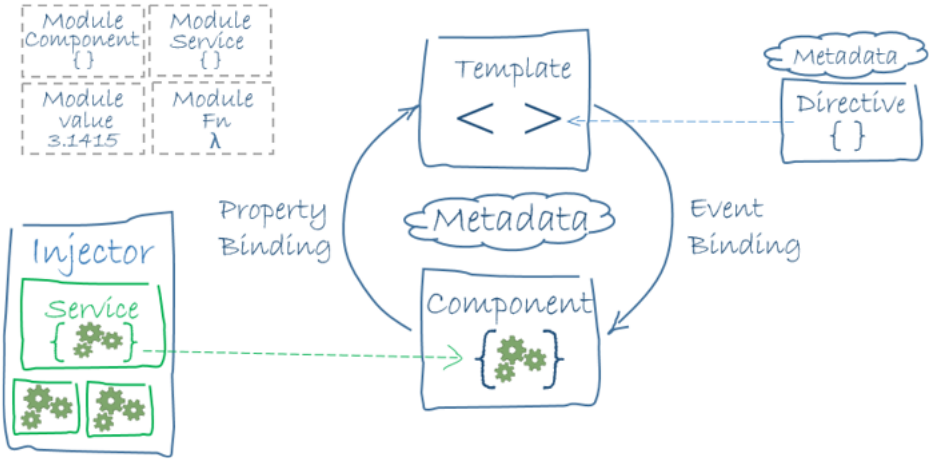
Angular este unul dintre cele mai populare framework-uri în ziua de azi ce ne permite să dezvoltăm aplicații de tip single-page. În aplicațiile web tradiționale, serverul răspunde la fiecare cerere a utilizatorilor prin pagini HTML. În single-page applications, întregul site este încărcat doar la prima cerere, cererile următoare urmând să modifice diferite porțiuni ale acestuia. Astfel dezvoltăm aplicații web mai rapide, oferind utilizatorilor o experiență mai fluidă. Angular este disponibil atât pentru limbajul de scripting TypeScript, cât și pentru JavaScript. În cazul soluției pentru TypeScript, programarea este bazată pe un meta-limbaj oferit de către aceștia.

Încă un aspect cheie al acestui framwerok este dezvoltarea unei aplicații bazată pe componente. O componentă este un bloc independent care are o logică, un view și un model care să o reprezinte. Template-ul ce reprezintă felul în care va fi vizualizată componenta, metdodele care controlează interacțiunea cu acea componentă și atributele propriu-zise (modelul) permit implementarea pattern-ului Model-View-Controller la nivel de componentă. Componentele sunt încapsulate și pot fi folosite în diferite părți ale aplicației, evitând astfel scrierea de cod duplicat. Angular are la bază un mecanism de dependency injection foarte bine pus la punct. Scopul acestuia este de a simplifica management-ul dependințelor dintre componente. Astfel componentele sunt mai decuplate, pot fi testate mult mai ușor si codul este mai flexibil.

Versiunea de Angular folosită în cadrul proiectului este 5, fiind ultima versiune stabilă în momentul realizării proiectului.

Pentru a ușura munca, am ales ca partea de front-end să fie bazată pe componente. Astfel am folosit Angular Material. Angular Material este un set de componente moderne realizate de către echipa Angular, bazate pe specificația de material design introdusă de Google. Această specificație este descrisă ca un set de instrucțiuni, pictograme și componente care se combină pentru a crea o experiență unificată a utilizatorilor pe telefon, calculator sau tabletă.

FIGURE 4.1: Arhitectura Angular



Această specificație este utilizată de Google în sistemul de operare Android și este, de asemenea, foarte populară pe web datorită utilităților sale UI.

Astfel a rezultat o bibliotecă de componente care sintetizează principiile bunului design. Componentele sunt modulare și ușor de integrat în aplicație. Componentele din această bibliotecă ne ajută să realizăm rapid design-ul aplicației, având un prototip funcțional într-un timp foarte scurt.

Angular Material nu vine cu suport direct pentru responsivitate. Pentru ca aplicația să poată fi folosită pe dispozitive de diferite dimensiuni a fost necesară utilizarea **FxLayout**. **FxLayout** este un pachet suplimentar pentru **Angular** ce ne pune la dispoziție un set de directive pe care le putem folosi direct în template. Nu avem nevoie de un fișier separat pentru **CSS**. **FxLayout** automatizează în mod inteligent procesul de aplicare a Flexbox CSS corespunzător în ierarhiile de vizualizare a browserului.

În Figura 3.3 am dat un exemplu de utilizare a Angular Material. Se observă utilizarea componentei mat-card care este oferită de către bibliotecă. Mat-card este un container de conținut pentru text, fotografii și acțiuni în contextul unui singur subiect. Angular Material oferă o serie de secțiuni prestabilite pe care le putem folosi în interiorul containerului, cum ar fi: mat-card-title, mat-card-content, mat-card-actions, etc. Demn de menționat este si modul de a instanția multiple componente folosind directiva ngFor.

Rezultă astfel componenta din figura de mai jos, fără a fi nevoie de cod CSS. Acesta este avantajul utilizării acestei biblioteci de componente. Aplicația fiind bazată în totalitate pe aceste componente, design-ul este uniform.

Pentru a prezenta statisticile am folosit librăria **chart.js**.

```
<mat-card class="project-card" fxFlex>
  <mat-card-header>
    <mat-card-title>{{project.title}}</mat-card-title>
    <mat-card-subtitle> {{'PROJECTS.PROPOSER' | translate}}:{{project.
  lecturer.name}} </mat-card-subtitle>
  </mat-card-header>
  <hr/>

  <mat-card-content>
    <p [innerHTML]="div.innerText"> </p>
    <mat-chip-list class="project-tags">
      <mat-chip *ngFor="let tag of project.tags" selected="true">
        {{ tag.name }}
      </mat-chip>
    </mat-chip-list>
  </mat-card-content>

  <mat-card-actions>
    <button *ngIf="(authService.role | async) === 'ROLE_STUDENT'" mat-
  button (click)="apply()">{{'PROJECTS.APPLY' | translate}}</button>
    <button mat-button [routerLink]="['/projects', project.id]"> {{'
  PROJECTS.DETAILS' | translate}} </button>
  </mat-card-actions>

</mat-card>

```

FIGURE 4.2: Exemplu utilizare Angular Material

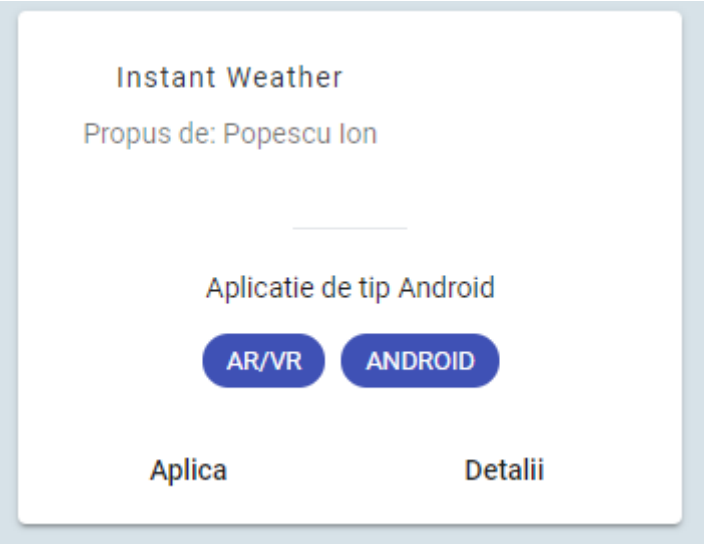


FIGURE 4.3: Exemplu componentă Angular Material

Pe partea de back-end am ales să folosesc drept limbaj de programare Java, iar ca framework Spring. Spring este un framework care asigură un model de configurare și de programare inteligent pentru aplicații moderne scrise în Java, indiferent de platforma folosită. Spring se bazează pe principiul de *convenție*. Datorită acestui lucru programatorul trebuie să facă mai puține *configurații*, care de cele mai multe ori sunt sursa erorilor, astfel putându-se concentra asupra codului scris.

Câteva dintre caracteristicile framework-ului sunt: dependency injection, programare orientată pe aspecte (AOP), suport fundamental pentru JDBC, JPA, etc.

Datorită faptului că cele două proiecte sunt separate, este nevoie de o metodă de comunicare între ele. Am ales ca cele două componente să comunice prin REST, server-ul expunând un astfel de serviciu, iar de pe front-end este consumat.

Spring Security este un modul oferit de Spring ce asigură servicii de securitate pentru aplicații dezvoltate pe platforma Java EE. Cele două mari arii în ceea ce privește securitatea sunt autentificarea si autorizarea. Autorizarea se referă la procesul de a decide dacă unui utilizator îi este permis să execute o acțiune într-o aplicație, iar autorizarea este procesul prin care identitatea utilizatorului este recunoscută. Spring Security oferă un set de capacități de autorizare foarte bine puse la punct. Există trei domenii principale de interes: autorizarea solicitărilor web, autorizarea metodelor (dacă pot fi invocate sau nu) și autorizarea accesului la instanțele de obiecte ale domeniului.

Având în vedere cele prezentate, acest modul a fost folosit în aplicația realizată. Ținând cont de natura aplicației, se identifică trei tipuri de utilizatori: studenți, profesori și administratori. Astfel, a fost nevoie de un mecanism de autorizare bazat pe roluri (RBAC) în cadrul căruia utilizatorii aparținând unei anumite categorii pot apela diverse metode. Acesta s-a implementat cu ajutorul acestei librării, folosind JSON Web Token(JWT).

JSON Web Token este un standard deschis, bazat pe JSON (RFC 7519) pentru crearea de tokeni de acces cu anumite revendicări. De exemplu, un server ar putea genera un token care are revendicarea de "logat ca administrator" și să îl furnizeze unui client. Clientul ar putea folosi acel token pentru a dovedi ca este logat ca administrator. Tokenii sunt semnați de cheia privată a unei părți (de obice serverul), astfel încât ambele părți pot verifica dacă tokenul este legitim. Acesta este un mecanism de autorizare fără stare, deoarece serverul nu va stoca informații despre utilizator într-o sesiune, astfel se eliberează memoria serverului.

După ce se realizează procesul de autentificare, clientul va primi un token pe care îl va salva în local-storage. Utilizatorul va trebui să trimită JWT la fiecare cerere facută către server, de obicei în antetul Authorization, folosind schema Bearer. Conținutul antetului ar trebui să arate dupa cum urmează:

Authorization: Bearer <token>

Spring oferă un mod facil de externalizare a proprietăților, în fișiere de configurare ce vor fi citite automat de către server. Astfel, în acest fișier au fost incluse și anumite proprietăți utilizate în procesul de autentificare și autorizare.

```
jwt.header = Authorization
jwt.expires_in = 300
jwt.mobile_expires_in = 600
jwt.secret = smapp
```

FIGURE 4.4: Constante JWT

În figura 3.4 sunt prezentate aceste proprietăți. Tokenii au o anumită durată de viață, reprezentată de câmpul *jwt.expires_ in*. Durata de viață a acestora este de 300 secunde. În cazul în care utilizatorul se conectează de pe un dispozitiv mobil, durata de viață a token-ului va fi de 600 secunde. Termenul de expirare identifică timpul după care token-ul nu va mai fi acceptat pentru procesare. Este util în cazul în care un token-ul este furat de către o terță parte. În cazul în care token-ul este furat, utilizatorul rău-voitor nu va putea accesa resursele protejate decât în perioada în care token-ul este valid.

Aplicația dispune de un mecanism de reîmprospătare a tokenilor, pentru a nu fi necesară autentificarea utilizatorilor de fiecare dată când token-ul expiră.

Câmpul *jwt.secret* reprezintă cheia cu care se realizează semnătura digitală a token-ului. Algoritmul de semnare folosit este **HS512**.

PostgreSQL este un sistem de baze de date relaționale. Este gratuit sub o licență open source de tip BSD. Aceasta nu este controlată de nici o companie, își bazează dezvoltarea pe o comunitate răspândită la nivel global, precum și câteva companii dezvoltatoare. Pentru a folosi PostgreSQL împreună cu Spring, trebuie să includem următoarea dependință în fișierul pom.xml aflat în rădăcina proiectului. Project Object Model sau POM este unitatea fundamentală de lucru în Maven. Este un fișier XML care conține informații despre detaliile de proiect și de configurare utilizate de Maven pentru a construi proiectul. Dependința postgresql este pentru driver-ul bazei de date PostgreSQL.

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

După cum am mai specificat, în fișierul application.properties scriem diferite setări de configurae ale unei aplicații Spring. Tot în acest fișier setăm si conexiunea către baza de date, numele și parola. Mai jos sunt prezentate setările din proiectul realizat, iar dacă se dorește a modifica vreun parametru se poate realiza foarte ușor.

```
spring.datasource.platform=postgres
spring.datasource.url=jdbc:postgresql://localhost:5433/smapp
spring.datasource.username=smapp
spring.datasource.password=$mapp
```

În figura 3.8 este prezentată arhitectura bazei de date. Pentru a modela aceste entități într-un mod cât mai orientat-obiect am folosit framework-ul Hibernate. Hibernate este o unealtă ORM (Object-Relational Mapping) pentru limbajul de programare Java care furnizează un framework pentru asocierea unui model creat folosind paradigma obiectuală la o bază de date relațională.

Proprietatea de bază pentru Hibernate este asocierea dintre clase definite în Java și tabele din baza de date, dar și asocierea dintre tipurile de date din Java și tipurile de date SQL. Acest framework oferă și anumite facilități precum interogări sau regăsire de datem generând interogări. Astfel dezvoltatorul este scutit de manipularea manuală și de convesia de obiecte a setului rezultat. Asocierea se poate realiza în două moduri: folosind un fișier de configurare XML sau folosind adnotări Java. În proiectul realizat, s-a folosind a doua variantă și anume lucrul cu adnotări.

```
@Entity
@Table(name = "projects")
public class Project {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String title;
  private String description;
  private Integer capacity;

  @ManyToOne
  @JoinColumn(name = "lecturer_id")
  private Lecturer lecturer;

  @OneToMany(mappedBy = "project", cascade = CascadeType.ALL)
  private List<Bibliography> bibliographies;

  @ManyToMany(cascade = CascadeType.ALL)
  @JoinTable(
    name = "project_tags",
    joinColumns = @JoinColumn(name = "project_id"),
    inverseJoinColumns = @JoinColumn(name = "tag_id")
  )
  private List<Tag> tags;

```

```
27   @OneToMany(mappedBy="project", cascade = CascadeType.ALL, fetch =
       FetchType.EAGER)
28   private SortedSet<Preferences> preferences;
29
30   /*
31    * other properties, getters, setters
32    */
33
34 }
```

În exemplul de mai sus am arătat un exemplu de utilizare a adnotărilor oferte de **Hibernate** pentru a modela entitatea "proiect". Demn de precizat este și faptul că se pot defini relații de tipul *one-to-many* sau *many-to-many* între clase, lucru care se răsfrânge automat și asupra schemelor bazelor de date (independent de dialectul folosit). Fiecare proprietate din clasă va fi tratată ca o coloană din tabel, iar numele din baza de date trebuie să coincidă cu numele proprietății. În cazul în care cele două nume diferă, se poate folosi adnotarea *@Column* împreună cu numele coloanei din tabel pentru a se realiza maparea.

Implementarea unui repository care folosește Java Persistence API a fost până acum ceva timp un proces care necesita mult timp și care presupune scriere de cod foarte încărcat. Prea mult cod duplicat trebuie scris pentru a executa interogări simple, precum și pentru a adăuga paginare. Spring Data JPA urmărește să îmbunătățească în mod semnificativ implementarea acestor repository-uri. În calitate de programator, vei scrie doar interfețe și metode care respectă anumite convenții, urmând ca Spring Data JPA să ofere implementarea în mod automat.

```
1 /**
2  * User Entity Repository
3  */
4 @Repository
5 public interface UserRepository extends JpaRepository<User, Long> {
6   User findByEmail(String email);
7   Page<User> findAllByEmailStartingWith(String email, Pageable pageable);
8 }
```

Prin implementarea extinderea interfeței JpaRepository, Spring Data JPA ne oferă implementarea metodelor CRUD și acces facil la paginare.