

**UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI**  
**FACULTATEA DE INFORMATICĂ**



**LUCRARE DE LICENȚĂ**

Aplicație Web pentru planificarea resurselor în cadrul  
examenului de admitere

**propusă de**

***Sergiu Adrian Volocar***

**Sesiunea: *Februarie, 2020***

**Coordonator științific**

**Lect.dr. Frăsinaru Cristian**

**UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI**  
**FACULTATEA DE INFORMATICĂ**

**Aplicație Web pentru planificarea resurselor în  
cadrul examenului de admitere**

***Sergiu Adrian Volocar***

**Sesiunea: *Februarie, 2020***

**Coordonator științific**

**Lect.dr. Frăsinaru Cristian**

## DECLARAȚIE PRIVIND ORIGINALITATE ȘI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licență cu titlul “ *Aplicație Web pentru planificarea resurselor în cadrul examenului de admitere*” este scrisă de mine și nu a mai fost prezentată niciodată la o altă facultate sau instituție de învățământ superior din țară sau 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.

Iași, *Februarie 2020*

Absolvent *Volocaru Sergiu Adrian*

---

## DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul “*Aplicație Web pentru planificarea resurselor în cadrul examenului de admitere*”, 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 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.

Iași, *Februarie 2020*

Absolvent *Volocarui Sergiu Adrian*

---

## Cuprins

Cuprins .....	5
Introducere .....	7
1    Specificații funcționale .....	9
1.1    Descrierea problemei .....	9
1.2    Fluxuri de lucru .....	12
2    Arhitectura sistemului .....	14
2.1    Schema bazei de date .....	16
2.2    Diagrama de pachete.....	18
2.3    Diagrama de clase.....	19
3    Client .....	20
3.1    Ce este un client? .....	20
3.2    Paginile web ale aplicației .....	21
3.3    Cerere către API.....	23
3.4    Tehnologiile folosite .....	24
3.4.1    CSS.....	24
3.4.2    Bootstrap.....	24
3.4.3    NodeJS.....	24
3.4.4    React JS.....	25
3.4.5    Axios.....	25
4    API .....	26
4.1    Ce reprezintă un API? .....	26
4.2    Tipuri de API-uri.....	26
4.3    REST API .....	28
4.4    Configurări.....	31
4.4.1    Nivelul transferului de date.....	32
4.4.2    Nivelul de business .....	34
4.4.3    Nivelul de persistență .....	35
4.5    Tehnologiile folosite .....	37
4.5.1    Maven .....	37
4.5.2    Spring Boot .....	37
4.5.3    Hibernate.....	38

4.5.4	PostgresSQL.....	38
4.6	Algoritmi folosiți .....	39
4.6.1	Distribuirea .....	39
4.6.2	Alocarea.....	42
5	Manual de utilizare.....	43
5.1	Modul de instalare.....	43
5.2	Utilizare .....	44
6	Concluzii și direcții viitoare .....	47
6.1	Rezumat .....	47
6.2	Îmbunătățiri .....	47
	Bibliografie.....	48
	Anexe .....	49

# Introducere

## Contextul proiectului

Pentru organizarea examenului de admitere în cadrul Facultății de Informatică sunt implicate următoarele resurse: sălile, candidații, opțiunile candidaților, categoriile de candidați și supraveghetorii.

Prin disponibilizarea resurselor enumerate mai sus acest proces care are loc este un model ce poate fi automatizat.

Scopul aplicației este următorul:

- de a distribui candidații în săli folosind numărul minim de săli care să conțină un număr minim de locuri pentru a putea susține examenul de admitere;
- de a genera rapoarte în format pdf cu distribuția acestora;
- așezarea ierarhică a candidaților pe baza notelor;
- de a genera rapoarte în format pdf cu listele finale ce conțin rezultatele admiterii;

Așadar aplicația care va fi prezentată va încerca să abordeze această problemă într-o manieră cât mai simplă și va trebui să prezinte starea în care se află resursele folosite.

Aceste stări pot fi următoarele:

- câți candidați sunt momentan înscriși ?
- câte săli avem ?
- câte locuri disponibile într-o anumită sală?
- care este clasamentul curent al candidaților care au susținut examenul?
- care sunt opțiunile candidaților pentru examenul de admitere? ect..

## Motivația

Am ales această temă pentru că este una practică și poate fi folosită în administrarea examenului de admitere din cadrul Facultății de Informatică.

Metologia de lucru folosită este Kaban. Kanban este un tip specific de metodologie Agile. Kanban se străduiește să coordoneze și să echilibreze mai bine munca cu capacitatea și lățimea de bandă între lucrători. Utilizează principiile metodologiei Agile, dar le pune în aplicare într-un mod special.

Un alt factor decizional a reprezentat provocarea de folosi noi tehnologii precum ReactJS<sup>1</sup>, NodJS<sup>2</sup>, Postgres<sup>3</sup> și JasperReports<sup>4</sup> și aplicarea acestora într-un mod cât mai corect.

---

<sup>1</sup> <https://reactjs.org/>

<sup>2</sup> <https://nodejs.org/>

<sup>3</sup> <https://www.postgresql.org/>

<sup>4</sup> <https://community.jaspersoft.com/>



## 1 Specificații funcționale

În acest capitol voi prezenta specificațiile funcționale aplicației construite împreună cu prezentarea fluxurilor de lucru reprezentate prin intermediul diagramelor.

### 1.1 Descrierea problemei

La baza aplicației sunt folosite cele 3 resurse principale pentru a organiza sesiunea de admitere. Aceste resurse sunt candidații, sălile și supravegatorii. Plecând de la aceste 3 resurse voi enumera specificațiile funcționale în ordinea priorităților.

1. Aplicația trebuie să fie capabilă să gestioneze cele 3 resurse. Această gestionare este definită prin următoarele funcționalități:
  - a) afișarea;
  - b) sortarea;
  - c) modificarea;
  - d) import;
  - e) export: Se poate alege afișare/export pentru toate câmpurile sau doar pentru unele, selectate de către utilizator.
  
2. Din punctul de vedere a distribuirii acestor 3 resurse aplicația trebuie să dispună de următoarele funcționalități:
  - a) introducerea numărului maxim de locuri în fiecare sală;
  - b) stabilirea numărului de locuri utilizate din fiecare sală;
  - c) alocarea în săli a candidaților pe categorii.

O categorie este definită prin următoarele caracteristici:

- ✓ disciplina la care este susținut testul scris
  - ✓ limba în care este susținut testul scris
  - ✓ tipul de concurs (admitere/preadmitere)
- d) în cazul în care într-o sală sunt prezente mai multe categorii în acest caz candidații vor fi grupați în sală în funcție de categorie;
- e) atunci când are loc distribuția candidaților pentru sesiunea de admitere, candidații olimpici nu susțin testul scris. De asemenea, uni candidați la admitere care au susținut preadmiterea în anii anteriori pot alege să nu susțină testul scris. Aceste două tipuri de candidați nu trebuie să apară în listele de mai sus. În cazul elevilor care susțin preadmiterea nu există asemenea excepții;

3. Aplicația trebuie să fie capabilă să facă calculul mediei de admitere. Aceasta este media ponderată a mai multor câmpuri, între testul scris și notele de la bacalaureat și preadmitere.

Trebuie prevăzute situații speciale, precum:

- a) Unul din termenii formulei mediei este reprezentat de maximul dintre testul scris și nota obținută la bacalaureat la Matematică/Informatică.
- b) Nota la testul scris este de fapt maximul dintre nota la proba susținută acum și nota obținută în trecut la preadmitere. La majoritatea candidaților, una sau alta dintre cele două note poate lipsi.
- c) Candidații care nu au loc în nici una din categoriile de mai sus vor fi separați în două liste: cei cu media de admitere cel puțin 5.00 (care pot deveni admiși prin retragerea altor candidați deja admiși), respectiv cei cu media de admitere sub 5.00 (care sunt respinși definitiv).

d) Ierarhizarea trebuie să poată fi refăcută ulterior, în urma contestațiilor admise, dar mai ales a retragerilor. Deoarece candidații declarați admiși au obligația de a-și confirma locul într-o anumită perioadă de timp, trebuie ca informația privind confirmarea locului să poată fi introdusă în baza de date. La fiecare refacere a ierarhiei, candidații declarați admiși anterior, care nu și-au confirmat locul (aici ar intra și cei care și-au retras dosarul), sunt declarați respinși definitiv, similar celor cu media sub 5.00, iar locurile lor sunt ocupate de alți candidați.

4. Aplicația trebuie să fie capabilă să afișeze rezultatele în următoarele formate:

a) Funcționalitatea de afișare a listelor pentru fiecare categorie menționată la gestionarea rezultatelor la admitere, în ordinea descrescătoare a mediilor:

1. câte o listă pentru fiecare categorie dată de statutul buget/taxă și de limba de studii, precum și o listă pentru olimpici
2. o lista pentru candidații care nu sunt admiși, dar au posibilitatea avansării ulterioare (media cel puțin 5.00)
3. o lista pentru candidați respinși (media sub 5.00, admiși anterior care nu și-au confirmat locul)

b) funcționalitatea de afișare a listei generale a candidaților la admitere, în ordine alfabetică; pentru fiecare candidat este specificată lista din cele de mai sus pe care se regăsește.

## 1.2 Fluxuri de lucru

Mai jos avem prezentată o diagramă use-case a aplicației.

Această diagramă are rolul de a:

- oferi o descriere generală a modului în care va fi utilizat sistemul;
- furnizează o privire de ansamblu a funcționalităților ce se doresc a fi oferite de sistem;
- arată cum interacționează sistemul cu unul sau mai mulți actori;
- asigură faptul că sistemul va produce ceea ce s-a dorit;

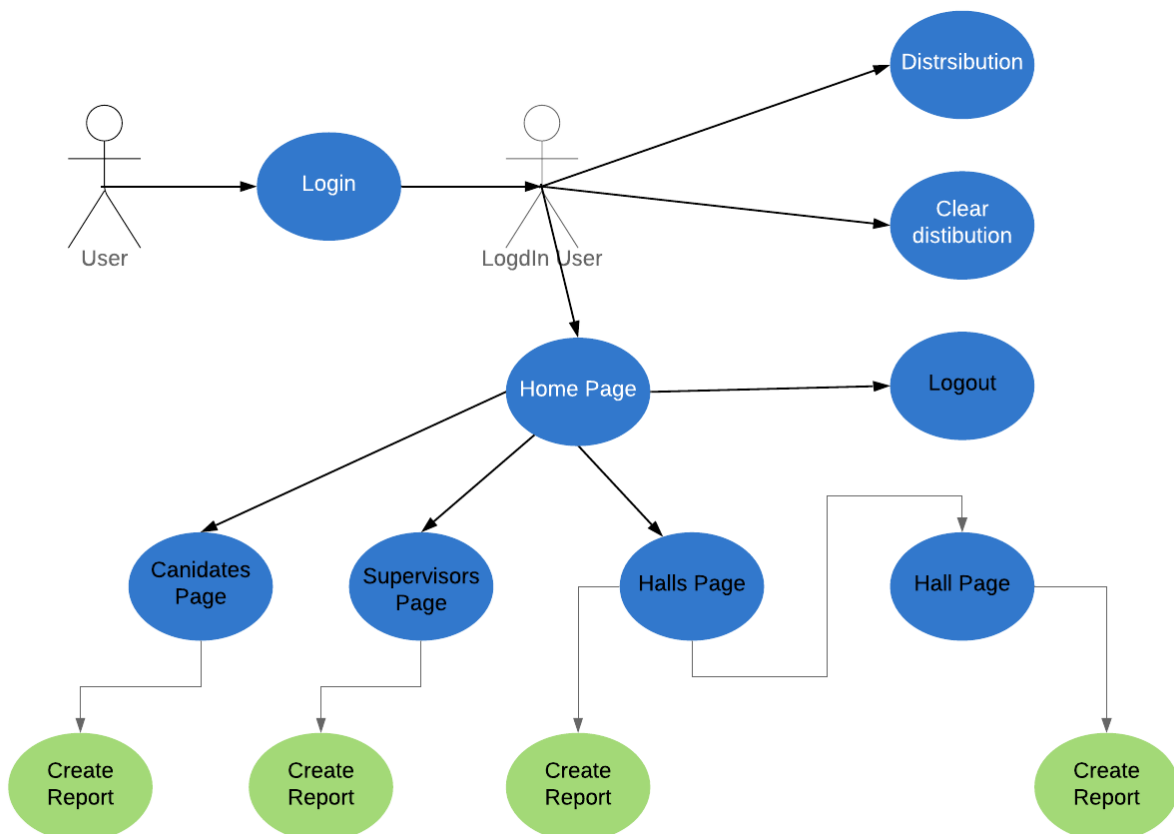
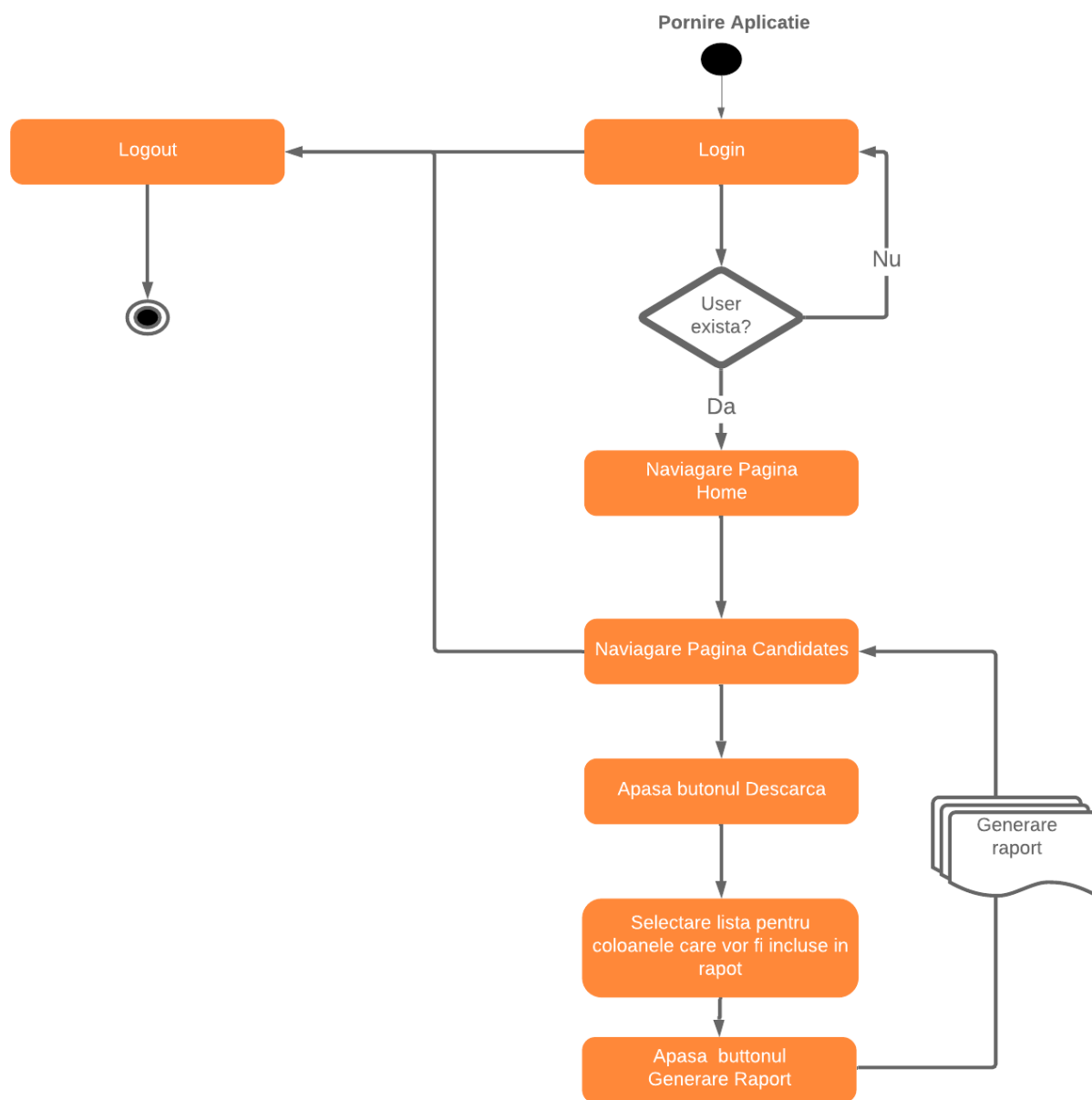


Figura 1: Digrama use-case a aplicației

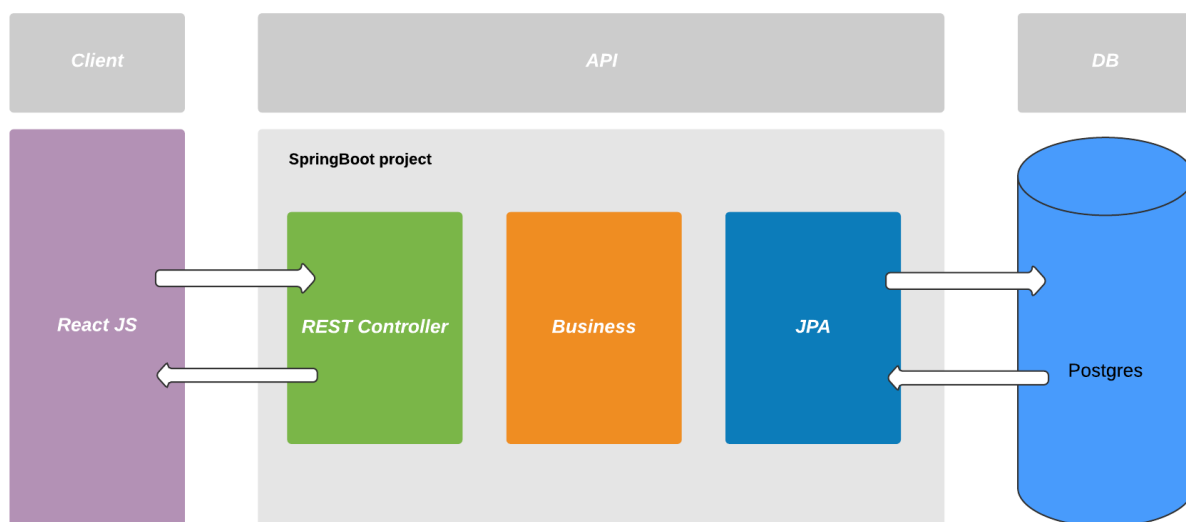
În figura de mai jos sunt prezentați pașii prin care un utilizator poate genera un raport din pagina Candidați.



**Figura 2** Diagrama de activitate pentru creerea unui raport

## 2 Arhitectura sistemului

În acest capitol voi prezenta arhitectura sistemului care va cuprinde: schema generală a aplicației, schemei bazei de date, diagramă de pachete și o diagramă de clase.



**Figura 3 Arhitectura generală a aplicației**

În Figura 3 Arhitectura generală a aplicației putem observa o separare între client și API.

REST API-ul este expus folosind SpringBoot. Acest API este consumat de client care este construit folosind ReactJS. Pentru baza de date s-a folosit Postgres.

În cazul în care dorim să dezvoltăm o aplicație mobilă pentru Android sau iOS putem să refolosim API-ul construit.

Această arhitectură este una flexibilă și poate fi extinsă pentru viitoare nevoi.

Însă o dată cu adoptarea unui anumit stil de arhitectură apar anumite constrângeri iar odată cu acestea și provocările.

Provocările întâlnite pot fi legate de:

- *Complexitate.* Trebuie să ținem cont dacă arhitectura aleasă este prea complexă. Este justificabilă complexitatea? Sau în cazul opus, este prea simplă? Aceste întrebări sunt importate pentru că stilul de arhitectură nu ne va ajuta să întreținem dependențele într-un mod cât mai ușor.
- *Mesagerie asincronă și consecvența eventuală.* Această poate fi folosită pentru a decupla servicii, pentru a crește fiabilitatea și scalabilitatea. Însă cu timpul va deveni o provocare din punctul de vedere a consistenței pentru că putem sfârși să avem mesaje duplicate.
- *Comunicarea între servicii.* Odată ce începem să decuplăm aplicația în servicii separate există riscul să îngreunăm comunicarea acestora. Această situație poate apărea prin faptul că ajungem să avem o latență inacceptabilă sau crearea congestiei rețelei (ca exemplu avem arhitectura bazată pe microservicii).
- *Gestionabilitatea.* Trebuie să ținem cont de cât de greu este de gestionat, monitorizat, deploy updates, ect..

După cum se poate observa în Figura 3 Arhitectura generală a aplicației am ales o structurare bazată pe layere. Acest tip de arhitectură este un standard folosit de majoritatea aplicațiilor Java EE (Enterprise Edition). Aceste straturi cuprind obiecte care sunt orientate către un anumit tip de responsabilitate.

Beneficiile arhitecturii bazată pe layere sunt următoarele:

- *Simplitate:* Este simplu de implementat și ușor de învățat.
- *Coerență:* Această modalitate de grupare a codului sursă oferă un plus din punct de vedere organizațional.
- *Navigabilitate:* Datorită restrângerilor oferite de aceste layere este foarte ușor de localizat un obiect.

## 2.1 Schema bazei de date

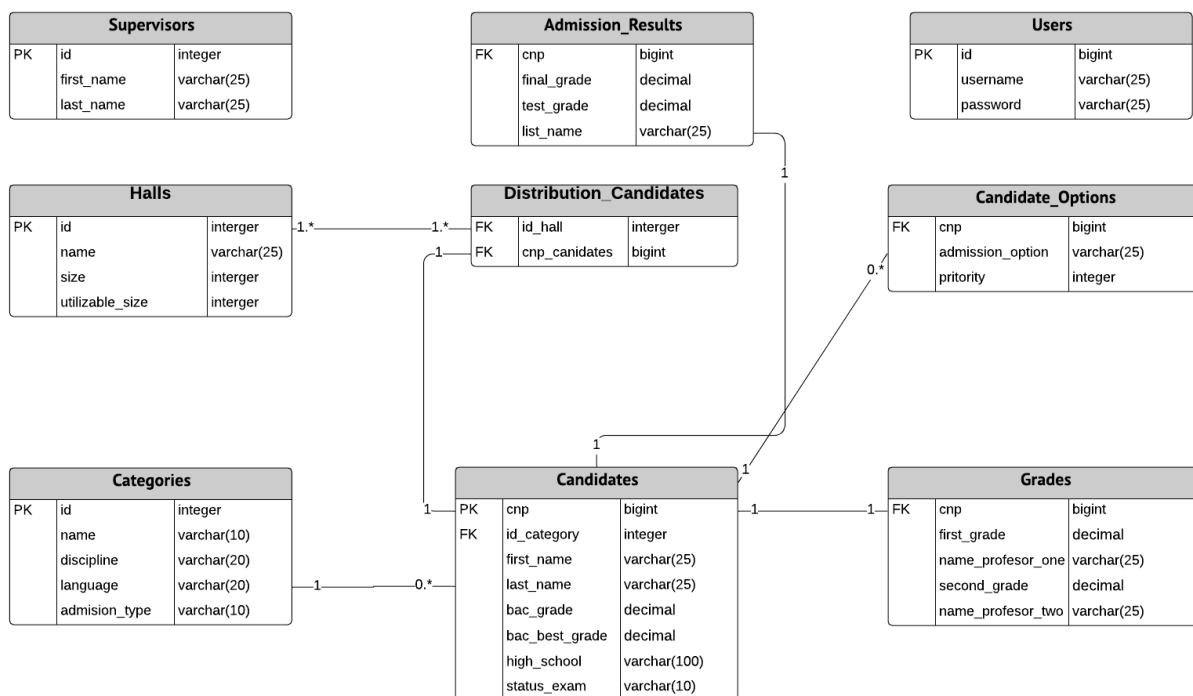


Figura 4 Schema bazei de date

Baza de date conține în total 9 tabele.

- USERS:** conține următoarele câmpuri: id , username și password. Cheia primară a acestei tabele este câmpul id. Această tabelă are rolul de a stoca informațiile necesare a unui utilizator pentru a se loga în aplicație.
- SUPERVISORS:** conține următoarele câmpuri: id , first\_name și last\_name. Cheia primară a aceste tabele este câmpul id și reprezintă tabela în care vor fi memorați supraveghetorii.
- HALLS:** conține următoarele câmpuri: id , name, size, utilizable\_size. Cheia primară a acestei tabele este câmpul id și reprezintă tabela în care vom memora sălile pentru examen.
- CANDIDATES:** conține următoarele câmpuri: cnp , id\_category, first\_name , last\_name, bac\_grade, bac\_best\_grade, high\_school, status\_exam . Cheia primară a acestei tabele este câmpul cnp însă conține și o cheie străină pentru a face legătură cu tabela *categories*. În această tabelă vom memora candidații care au aplicat pentru



admitere. În câmpul status\_exam va fi adugat la finalizarea examenului valoarea ADMIS sau RESPINS în funcție de rezultatul candidatului.

5. **CATEGORIES:** conține următoarele câmpuri: id , name , discipline, language and admission\_type. Cheia primară a acestei tabele este câmpul id. Această tabelă conține modalitățile în care un student poate susține examenul de admitere. Un exemplu ar putea fi “disciplina=Matematică, language=Engleză, admission\_type=Admitere”
6. **GRADES :** conține următoarele câmpuri: cnp , first\_grade, second\_grade. Cheia primară a aceste tabele este câmpul cnp. Aici vor fi memorate cele 2 note obținute din partea celor 2 profesori care vor corecta lucrarea candidatului.
7. **DISTRIBUTION\_CANDIDATES:** conține următoarele câmpuri: cnp și id\_hall. Cele 2 câmpuri sunt 2 chei străine și formează o cheie compusă. Acesta tabelă este folosită pentru a realiza asocierea dintr-un candidat și sala în care acesta a fost distribuit.
8. **CANDIDATE\_OPTIONS:** conține următoarele câmpuri: cnp, admission\_option și priority. Aici este folosită cheia străină cnp pentru a face legătura cu tabela candidates. În această tabelă vom găsi opțiunile pentru care a aplicat candidatul la examenul de admitere. Aceste opțiuni vor fi folosite pentru a crea listele finale.
9. **ADMISSION\_RESULT:** conține următoarele câmpuri: cnp, final\_grade, test\_grade, list\_name. În această tabelă vom reține media finală a candidatului, nota testului care este formată din media notelor primite de la cei 2 profesori și lista finală din care va face parte. Această tabelă împreună cu tabela *candidate\_options* vor fi folosite pentru rapoartele finale.

## 2.2 Diagrama de pachete

În Figura 5 Diagrama de pachete putem observa fluxul de date dintre clasele din cadrul aplicației care are loc la nivelul celor 9 pachete.



**Figura 5 Diagrama de pachete**

## 2.3 Diagrama de clase

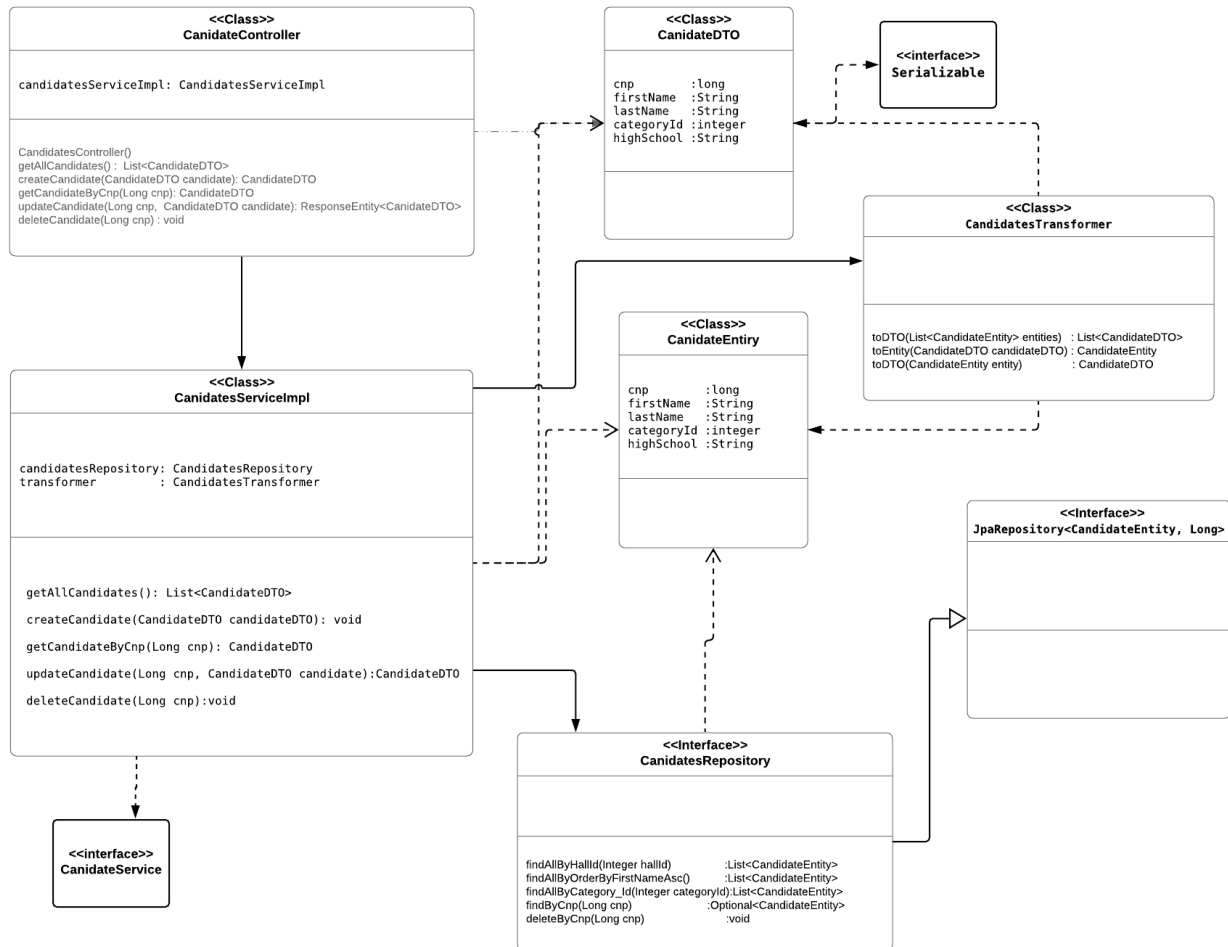


Figura 6 Diagrama de clase pentru Candidat

### 3 Client

În acest capitol voi prezenta diferite tipuri de client și motivația tipului ales. De altfel, voi prezenta și paginile prezente în aplicația web construită împreună cu rolurile acestora cât și tehnologiile folosite.

#### 3.1 Ce este un client?

Clientul reprezintă capătul primitor a unui serviciu sau solicitantul unui serviciu într-un tip de sistem model client-server. Clientul este cel mai adesea localizat pe un alt sistem sau computer care poate fi accesat prin rețea. Acest termen a apărut din prisma faptului că unele dispozitive nu își puteau rula propriile programe și erau conectate la computere la distanță.

Un client poate fi o simplă aplicație sau un întreg sistem care accesează servicii oferite de un server. Un client se poate conecta către un server prin diferite căi cum ar fi: socket, shared memory sau folosind protocoale de internet fiind chiar și cea mai comună metodă.

Clienții pot fi clasificați în trei tipuri:

- **Thin Client:** Reprezintă un client cu funcționalități minimale. Acesta preia informația deținută de un computer gazdă și treaba lui este să afișeze informația procesată de un server. El se bazează pe server să facă totul.
- **Thick/Fat Client:** Acesta este opusul tipul de client Thin Client. El poate face tot ce ține de procesarea informației și nu se bazează neapărat pe un server central, însă ar putea avea nevoie să se conecteze pentru informații, upload sau chiar update-uri pentru client în sine. Programele Anti-virus aparțin acestei categorii deoarece ele nu se bazează pe conectarea unui server pentru a le face treaba, însă acestea trebuie să se conecteze periodic pentru a primi noi informații despre noi viruși.
- **Hybrid:** Acesta este un mix între cele 2 tipuri de mai sus și reprezintă opțiunea aleasă pentru acest proiect. Clientul poate procesa o mare parte din funcționalități dar se bazează pe un server pentru informații critice sau pentru stocare informației.

## 3.2 Paginile web ale aplicației

Atunci când este proiectată grafica layout-ului pentru interfața utilizatorului trebuie să avem în vedere cele 2 concepte:

1. **UI** care se preocupă cu interfața utilizatorului ;
2. **UX** care se preocupă cu experiența utilizatorului.

**1.UI** (interfața utilizatorului) reprezintă tot ceea ce poate interacționa cu utilizatorul pentru a folosi o aplicație web sau un produs digital. Aici putem include de la tot ce vedem pe ecran sau ecrane tactile până la sunete, lumini și tastaturi.

**2.UX**(exeperiența utilizatorului) a apărut datorită progresului și evoluției UI-ului. În momentul în care un utilizator are o modalitate de interacționare experiența acestora va fi fie una negativă , una pozitivă sau chiar neutră. Aceste experiențe sau transformat în ce simț utilizatori despre aceste interacțiuni cu produsul cum ar fi de exemplu dacă produsul este: folositor, credibil, accesibil, folositor, ect. .

Cele 2 concepete au o legătură pentru că sunt centrate pe utilizator însă UX este focusat pe experiența utilizatorului de a rezolva o problemă pe când UI este focusat pe produsul la suprafață în modul în care arată și cum funcționează.

Atunci când am stabilit paginile web a aplicației am luat în calcul nevoie principale ale clientului. Toate paginile web împreună trebuie să cuprindă toate funcționalitățile și serviciile necesare și să fie la îndemâna utilizatorului. Practic această separare a paginilor web reprezintă o separare a responsabilităților și serviciilor pentru a fi mai ușor înțeles de către utilizator.

Paginile prezente în aplicație sunt următoarele:

- **Pagina de Home:** Acesta pagină pune la dispoziție utilizatorului diferite informații utile reprezentate fie prin chart-uri sau card-uri(aceaste fiind integrate folosind Bootstrap). Acesta pagină este responsabilă pentru a oferi statusul curent în cadrul susținerii examenului de admitere. Prin intermediul acestei paginii candidatul poate să gestioneze administrarea examenului și finalizarea acestuia.

- **Pagina de Candidați:** În cadrul acestei pagini utilizatorului poate vizualiza toți candidații, poate efectua operații precum: adăugare, ștergere, modificare a unui candidat și poate genera un raport cu toți candidații într-un format PDF. După ce examenul a fost finalizat în dreptul fiecărui candidat va fi o nouă funcționalitate disponibilă. Această funcționalitate constă în respingerea unui candidat.
- **Pagina de Supraveghetori:** În cadrul acestei pagini utilizatorului poate vizualiza toți supraveghetorii, poate efectua operații precum: adăugare, ștergere, modificare a unui supraveghetor și poate genera un raport cu toți supraveghetorii într-un format PDF.
- **Pagina de săli:** În cadrul acestei pagini utilizatorului poate vizualiza toate sălile, poate efectua operații precum: adăugare, ștergere, modificare a unei săli, poate genera un raport cu toate sălile într-un format PDF și poate accesa o anumită sală.
- **Pagina unei sale:** În cadrul acestei pagini vor fi prezenți doar candidații specifici acestei săli unde se vor putea efectua aceleași operații ca *în pagina de candidați* pentru acestea.
- **Pagina Categori:** În cadrul acestei pagini vom putea vedea categoriile disponibile pentru examen și numărul candidaților care fac parte din acea categorie.
- **Pagina Rapoarte:** În această pagină putem să descărcăm rapoartele cu rezultatele finale și cele referitoare la distribuirea candidaților.
- **Pagina Nothing:** Utilizatorul va fi redirecționat la această pagină în cazul în care a accesat un URL greșit.
- **Pagina Login:** Reprezintă pagina prin care se poate autentifica utilizatorul pentru a folosi aplicația.

### 3.3 Cerere către API

Componenta API este folosită pentru a accesa un API și este definită în felul următor:

```
5 export const API = axios.create({
6   baseURL: getBaseURL()
7 });
8
9 export function getBaseURL() {
10   if (process.env.REACT_APP_STAGE !== 'production') {
11     console.log("REACT_APP_STAGE:" + process.env.REACT_APP_STAGE);
12     return 'http://localhost:8080/';
13   }
14   return 'https://api-licenta.herokuapp.com/';
15 }
16
17 export const authService = {
18   login: (token) => {
19     cookie.save('bearer', token);
20
21     API.interceptors.request.use( onFulfilled: config => {
22       config.headers.Authorization = token;
23
24       return config
25     });
26
27     API_BLOB.interceptors.request.use( onFulfilled: config => {
28       config.headers.Authorization = token;
29
30       return config
31     })
32   },
33   logout: () => cookie.remove('bearer'),
34   isAuthenticated: () => cookie.load('bearer') !== undefined,
35   getToken: () => cookie.load('bearer')
36 };
```

**Figura 7 Componenta folosită pentru a creea cereri către API**

După cum putem observa în codul prezent de mai sus această componentă conține tipul de format în care se dorește să se primească un răspuns JSON și URL-ul de bază care va fi folosit pentru a crea de viitoare cereri.

Acest URL de bază va fi în funcție de mediul curent în care este pornită aplicația fie cel de producție sau de dezvoltare.

Pentru a construi o cerere către REST API va trebui să folosim sintaxa următoare:

```
function onBeforeSaveCell(row, cellName, cellValue) {
  API.put("/candidates/" + row.cnp, row).then((response) => {
    console.log(response);
  }, (error) => {
    console.log(error);
  });
}
```

În codul de mai sus avem prezentată funcția de modificare a unui candidat . Acesta folosește componenta API pentru a crea o cerere de modificare a unui candidat. Această componenta API are nevoie de următoarele informații:

- **tipul metodei:** “PUT”;
- **adresa API-ului:** “/candidates/+ cnpul candidatului pe care dorim sa îl modificăm;
- **informația pe care dorim să o transmitem:** în acest caz row;

## 3.4 Tehnologiile folosite

### 3.4.1 CSS

CSS este o abreviere pentru Cascading Style Sheet cu accentul pe Style. Ținând cont de faptul că HTML este folosit pentru a construi o structură a unui document web prin definirea elementelor precum titluri, paragrafuri, taguri pt imagini, ect.. . CSS vine pentru a oferi suport în stilizarea acestora. Aceste stilizări constă în culori, font-uri, scheme ect.. . Un avantaj este că aceste stiluri pot fi refolosite și ajustate pentru diverse rezoluții.

### 3.4.2 Bootstrap

Bootstrap este un proiect open source și reprezintă un framework pentru CSS, HTML și JavaScript pentru dezvoltare de tip responsiv. Acesta este orientat pentru a simplifica munca dezvoltatorului. Ce mai importantă componenta din cadrul acestui framework este aceea care se ocupă de aspectul întregii pagini. Toate elementele paginii web sunt așezate într-un container iar acestea vor fi ajustate dinamic în funcție de caracteristicile dispozitivelor folosite pentru deschiderea paginii precum tableta, telefon sau desktop.

### 3.4.3 NodeJS

Node.js este o platformă care execută cod JavaScript în afara unui browser. Acesta este folosit în principal pentru a construi programe în rețea precum web services. Dacă ar fi să comparăm Node.js și PHP putem spune ca majoritatea funcțiilor în PHP sunt blocate până la terminare pe când în Node.js funcțiile sunt non-blocante. Comenzile sunt executate concurrent sau chiar paralel și folosesc *callbacks* pentru a semnaliza finalizarea sau eșecul. Datorită acestui beneficiu acesta poate să suporte până la 10 000 de conexiuni concurente fără a suporta costul unui *thread*. Node.js folosește un *event loop* pentru scalabilitate în loc procese cu thread-uri. Acesta iese din event loop când nu mai sunt de realizat callbacks.



### 3.4.4 React JS

ReactJS este o librărie de JavaScript folosită pentru a construi UI în special pentru aplicațiile de o pagină. Un mare plus a acestei librării este faptul că poți refolosi codul prin componente UI. Am ales React pentru că îți permite să creezi aplicații mari care pot modifica informații fără să reîncarci pagina. Aceasta este rapid, scalabil și simplu.

### 3.4.5 Axios

Axios este un promise bazat pe un client HTTP pentru a face cereri către servere din afara browser-ului. Axios oferă asistență pentru interceptări de solicitare și răspuns, transformatoare și conversia automată la JSON. Din punct de vedere al securității oferă protecție în mod implicit unor cereri transversale straine.

## 4 API

În acest capitol voi prezenta API-ul. Voi trece prin ceea ce reprezintă un API, cum poate fi clasificat, regulile de construire a unui REST API, tehnologiile care stau la bază pentru a crea unui API și descrierea nivelurilor folosite prin prezentare de cod sursă.

### 4.1 Ce reprezintă un API?

API-ul este o abreviere care înseamnă o interfață de programare a aplicației. În termeni simpli acesta reprezintă un endpoint. Este mesagerul care primește o cerere pe care o livrează sistemului și se întoarce înapoi cu un răspuns. Acesta poate avea control la o resursă hardware și software. În termeni mai de bază, API-ul este un intermediar software care permite la două aplicații diferite să comunice între ele.

API-ul reprezintă o alternativă pentru servere de a comunica mai rapid și eficient. Atunci când avem un set de funcționalități sau proceduri care poate fi refolosit este indicat să se construiască un API.

Beneficiul major care îl poate oferi este faptul că este simplu de folosit.

### 4.2 Tipuri de API-uri

Din punct de vedere al accesibilității ele pot fi:

- **Open APIs** - cunoscute ca API-urile publice ce pot fi accesate de oricine.
- **Partner APIs**- pentru a fi accesate acestea este nevoie de drepturi sau licențe specifice. De obicei, aceste tipuri de API sunt asociate cu servicii plătite.
- **Internal APIs**- cunoscute ca API-urile private, acestea fiind expuse doar sistemelor interne
- **Composite APIs** - utilizarea sa principală este de a accelera procesul de execuție și de a îmbunătăți performanțele ascultătorilor în interfețele web.

Din punct de vedere al comunicării acestea se împart în:

- **High-Level APIs**- sunt cele care au un nivel ridicat de abstractizare și suntem ocupați doar de efectuarea unei funcționalități limitat precum în arhitectura REST.

- **Low-Level APIs**- sunt cele care au un nivel scăzut de abstractizare. Aici programatorul are posibilitatea să manipuleze funcții în cadrul hardware-ului sau în cadrul unui modul din aplicație. Aceste tipuri de API sunt în general folosite în media.

Am ales să construiesc un *Web API* care folosește protocolul HTTP. Un Web API este cunoscut ca un serviciu web care oferă o interfață pentru aplicațiile web ori pentru aplicațiile care trebuie să se conecteze între ele folosind comunicarea prin internet. Un serviciu web este o colecție de protocoale și standarde deschise, care sunt utilizate pe scară largă pentru schimbul de date între sisteme sau aplicații. Nu este o tehnologie ci un concept.

Acesta poate fi chemat sau consumat de diferite tipuri de aplicație precum aplicații web, aplicații desktop, aplicații mobile ect. Atunci când construim un web API nu suntem constrânși să folosim o anumită interfață sau tehnologie. De exemplu acesta poate fi construită folosind tehnologii precum Java, JavaScript, .NET ect.

Beneficiile tehnice care le poate aduce un web API sunt:

- ✓ Faptul că dezvoltăm un web API din start facem o separare de dezvoltarea pentru UI.
- ✓ Este simplu de creat și nu implică configurații complexe.
- ✓ Poate fi ușor de testat folosind tool-uri precum: Postman, PingAPI, ect..
- ✓ Are la baza protocolul HTTP asta îl face ușor de definit și simplu de folosit în metodologia REST.
- ✓ Suportă de asemenea și funcționalitățile MVC
- ✓ Este o arhitectură ușoară și este bună pentru dispozitive care au o lățime de bandă limitată, în această categorie ar putea fi incluse telefoanele.
- ✓ Folosește URI's pentru a identifica resurse.

Cele mai cunoscute servicii web sunt:

- **SOAP** (Simple Object Access Protocol) Acesta este un protocol. Folosește XML ca format pentru a transfera date și WSDL( Web Services Definition Language) pentru a expune interfața pentru serviciul oferit.
- **XML-RPC** Acesta este un protocol. Este mai vechi decât SOAP însă este mai simplu decât SOAP. Pentru a transfera date este folosit formatul XML(diferit de SOAP).
- **JSON-RPC** Acesta este un protocol. Este asemănător cu XML-RPC care folosește JSON ca format pentru transferul de date.

- **REST (Representational State Transfer).** Acesta nu este un protocol. Acesta este văzut ca un set de principii arhitecturale. Un web service pentru a fi considerat REST trebuie să aibă anumite caracteristici precum interfețe simple de identificat și folosit pentru cerere și manipularea resurselor.

### 4.3 REST API

Am ales ca serviciul web construit să fie REST. Acesta stil de arhitectură a fost inițial inventat de Roy Fielding care este și unul din principalii autori ai HTTP-ului. Roy Fielding împreună cu colegii lui a avut un singur obiectiv. Acel obiect a fost de a crea o standardizare astfel încât orice server să poată comunica cu alt server din lume.

REST prezintă o modalitate de comunicare/relaționare dintre server și client. Serverul deține informația și clientul poate cere informația însă clientul nu știe de exemplu unde este stocată informația sau cum este procesată. Clientul este interesat doar de tipul formatului în care va primi informația.

Spre deosebire de arhitecturile SOAP sau RPC care sunt modelate după metode și proceduri, arhitectura REST este centrată pe resurse. O resursă poate fi orice. O resursă poate fi o imagine, un document, o linie în baza de date, rezultatul unui algoritm etc ... În termeni practici o resursă poate reprezenta tot ce poate fi stocat pe un calculator.

În cadrul arhitecturii REST o resursă poate fi reprezentată în JSON, XML, HTML sau ceva cu totul diferit. Însă trebuie definită clară reprezentarea resurselor folosind deja caracteristicile protocolului HTTP. Pentru ca folositorii acestui API care cunosc protocolul HTTP să înțeleagă foarte ușor ce va face API-ul, utilizându-l doar uitându-se la metodele HTTP.

CERERE	Metoda	Path
Creeare un nou candidat	POST	/candidates
Șterge un candidat	DELETE	/candidates/{id}
Preia un candidat specific	GET	/candidates/{id}
Preia toți candidații	GET	/candidates/
Modifică un candidat	PUT	/candidates/{id}

Orice url pe care noi îl accesăm cum ar fi <https://api-licenta.herokuapp.com/candidates/> va trimite o cerere către un server identificat de aceasta adresă. Această cerere trimisă către server va avea un răspuns. Atât cererea cât și răspunsul sunt definite de protocolul HTTP(*Hyper Text Transfer Protocol*).

Aceste cereri sunt construite pe baza unor verbe, cunoscute ca verbele HTTP.

Cele mai comune verbe sunt:

- **POST** este folosit atunci pentru a crea resurse noi. În special este folosit pentru a crea resurse subordonate. Adică este aplicată pe resursa părinte și are grijă de a asocia noua resursă cu resursa părinte. La crearea cu succes se întoarce codul 201. POST nu este sigur deoarece atunci când facem două cereri POST mai mult ca sigur vom ajunge să avem 2 resurse cu aceleași informații.
- **GET** este folosit atunci când dorim să obținem o resursă. Aceasta este folosit în cazul în care dorim să citim informația. GET este sigur deoarece când facem multiple cereri indentice GET vom obține mereu aceeași informație. La primirea informației cu succes se va întoarce codul 200.
- **PUT** este folosit atunci când dorim să modificăm o resursă. Însă acesta poate fi folosit și pentru a crea noi resurse în cazul în care id-ul este ales de client și nu de server. La fel ca și POST nu este sigur deoarece modifică informația. PUT poate întoarce diferite coduri de succes. Va întoarce 200 dacă a modificat informația cu succes sau 204 dacă nu deține nici o informație în câmpul body și 201 dacă a creat cu succes o nouă resursă.
- **DELETE** este folosit atunci când dorim să ștergem o resursă. În cazul în care resursa a fost ștearsă cu succes va returna codul 200 sau 204.
- **PATCH** este folosit atunci când dorim să modificăm capabilități. O cerere de tip PATCH nu are nevoie de resursă întreaga ci doar de modificările ale acelei resurse.

Aceste verbe construiesc ca mai comună paradigmă a serviciilor web numita CRUD. CRUD provine de la create(creare), read(citire), update(modificare), delete(stergere).

O aplicație sau arhitectură considerată RESTFull sau modelul REST are următoarele caracteristici:

- Orice resursă trebuie să fie accesibilă folosind operațiile CRUD.
- Arhitectura este client/server și: fără stare, stratificată și suportă memoria în cache.

Atunci când construim un API trebuie să ținem cont de anumite standarde atunci când dorim să accesăm o anumită resursă. Mai jos voi exemplifica resursa referitoare la săli. Ținând cont de criteriile oferite la 4.3 REST API trebuie să:

- ne asigurăm că este cât mai simplu posibil și cât mai ușor de intuit, un exemplu este pentru toate sălile de curs folosim URL-ul /halls și pentru sala cu id=1 folosim URL-ul/halls/1
- folosim substantive precum /halls și nu verbe /getAllHalls
- folosim corect metodele HTTP enumerate la 3.3 REST API
- folosim pluralul /halls în loc de singularul /hall în acest fel se va evita confuzia
- ținem cont dacă în cazul în care avem un API care vrea să spună ceva mai mult ar trebui să alegem varianta /halls?category='ABC' în loc de /getHallsByCategory
- folosim corespunzător codurile HTTP enumerate la 3.3 REST API

Atunci când lucrăm cu arhitectura RESTFull sau REST trebuie să respectăm următoarele principii de dezvoltare:

1. **Client-server:** Folosind acest model este separată partea vizuală de ceea ce se ocupă de sarcini asupra datelor. Servărul va accepta sau respinge cererea primită de la client printr-un răspuns care poate fi interpretat de către client.
2. **Apatrid:** Este de datoria clientului de a se asigura că toate informațiile sunt furnizate de server. Ca regulă este foarte important ca servărul să nu dețină nici un fel de informație între cererile clientului. În acest fel se va elimina ambiguitatea. Mai exact servărul nu va ține cont de sarcinile oferite clientului înaintea unei noi sarcini. Relația de comunicare dintre client-server se va reduce la întrebare-răspuns.
3. **Cache:** Deoarece cererile clientului sunt independente, uneori clientul poate cere același lucru de mai multe ori. Acest lucru poate duce la intensificarea traficului de-a lungul rețelei. Așadar, acest concept este implementat de către client pentru a

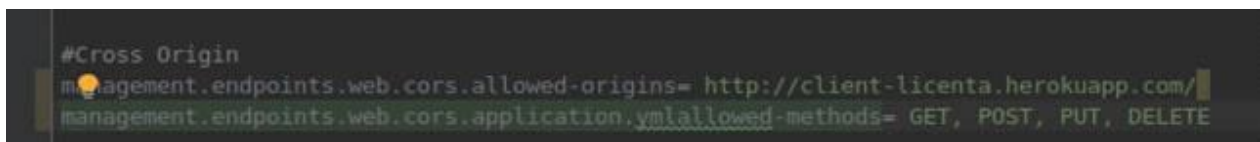
evita să fac o nouă cerere către server. Acest lucru poate duce la micșorarea performanței dintre cele 2 entități.

4. **Sistem stratificat:** Acesta este un strat între client și serviciul web. Acesta poate fi chiar un serviciu suplimentar cu care clientul interacționează înainte să facă o cerere către serviciul web. Acest strat va oferi transparență și nu va îngreuna comunicarea dintre client-server.
5. **Interfață/Contract uniform:** În cadrul REST acest concept reprezintă decuplarea clientului de implementarea unui serviciu REST. Pentru a defini o asemenea interfață trebuie să folosim anumite standarde:
  - Identificarea resurselor
  - Manipularea resurselor prin reprezentări
  - Mesaje auto-descriptive
  - Hypermedia ca motor de stare a aplicației

## 4.4 Configurări

În această relație dintre client și API bazată pe întrebare-răspuns Clientul folosește framework-ul Axios pentru a construi cereri. Din motive de securitate browser-urile interzic cererile către resurse care vin din afara originii curente. Această partajare a resurselor încrucișate(CORS) este o specificație W3C implementată de majoritate browser-urilor unde poți să specifice care domenii sunt autorizate pentru a face solicitări.

În Figura 8 Cross Origin avem prezentate cele 2 proprietăți din fișierul *application.properties* prezent în API referitoare la Cross Origin. În prima opțiune este specificată lista de domenii care are permisiunea de a face cereri către acesta iar în cea de a doua proprietate este specificată lista de metode permise.

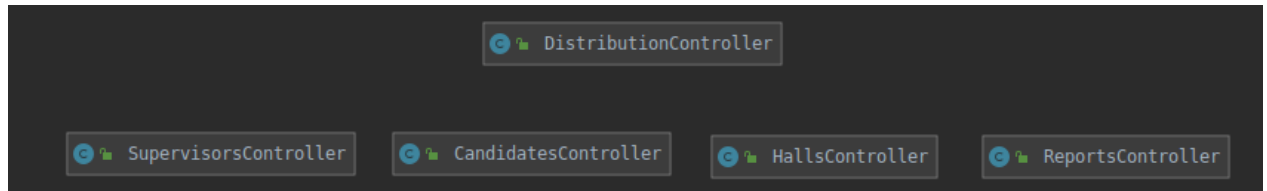


```
#Cross Origin
management.endpoints.web.cors.allowed-origins= http://client-licenta.herokuapp.com/
management.endpoints.web.cors.application.yml.allowed-methods= GET, POST, PUT, DELETE
```

Figura 8 Cross Origin

#### 4.4.1 Nivelul transferului de date

În cadrul acestui nivel are rolul de intercepta request-uri HTTP. Acesta le preia și le transmite mai departe către nivelul de business de unde va aștepta un răspuns. Răspunsul la rândului lui va avea acelaș traseu prin faptul ca va fi trimis înapoi către client folosind acelaș format prin protocolul HTTP.



**Figura 9 Clasele care se ocupă cu interceptările request-urilor**

Clasele din Figura 4 din cadrul API-ului comunică cu clientul folosind ca modalitate de comunicare obiecte DTO. **DTO** provine de la *Data Transfer Object*.

În aceste obiecte sunt stocate doar informațiile de care este interesat clientul folosind procesul de serializare si deserializare. Acesta poate avea un rol de a îmbunătăți performanța prin minizarea cererilor către server. De exemplu atunci când dorim să încărcăm pagina de Home avem 3 card-uri cu diferite informații precum: numărul de candidați înscriși, numărul de săli folosite și numărul de supraveghetori. Aceste informații pot fi considerate împreună un *view* și le integrăm pe toate împreună evitând să folosim 3 cereri separate.



```

@CrossOrigin
@RestController
public class CandidatesController {

    @Autowired
    private CandidatesService candidatesService;

    @GetMapping("/candidates")
    public List<CandidateDTO> getAllCandidates() { return candidatesService.getAllCandidates(); }

    @PostMapping("/candidates")
    public void createCandidate(@Valid @RequestBody CandidateDTO candidateDTO) {
        candidatesService.createCandidate(candidateDTO);
    }

    @GetMapping("/candidates/{cnp}")
    public CandidateDTO getCandidateByCnp(@PathVariable(value = "cnp") Long cnp) {
        return candidatesService.getCandidateByCnp(cnp);
    }

    @PutMapping("/candidates/{cnp}")
    public CandidateDTO updateCandidate(@PathVariable(value = "cnp") Long cnp,
                                         @Valid @RequestBody CandidateDTO candidateDTO) {
        return candidatesService.updateCandidate(cnp, candidateDTO);
    }

    @DeleteMapping("/candidates/{cnp}")
    public ResponseEntity<?> deleteCandidate(@PathVariable(value = "cnp") Long cnp) {
        candidatesService.deleteCandidate(cnp);
        return ResponseEntity.ok().build();
    }
}

```

**Figura 10 Clasa CandidatesController**

În cadrul claselor din acest nivel am folosit diverse adnotări pentru a minimiza timpul de lucru și de a fii cât mai ușor de citit și înțeles.

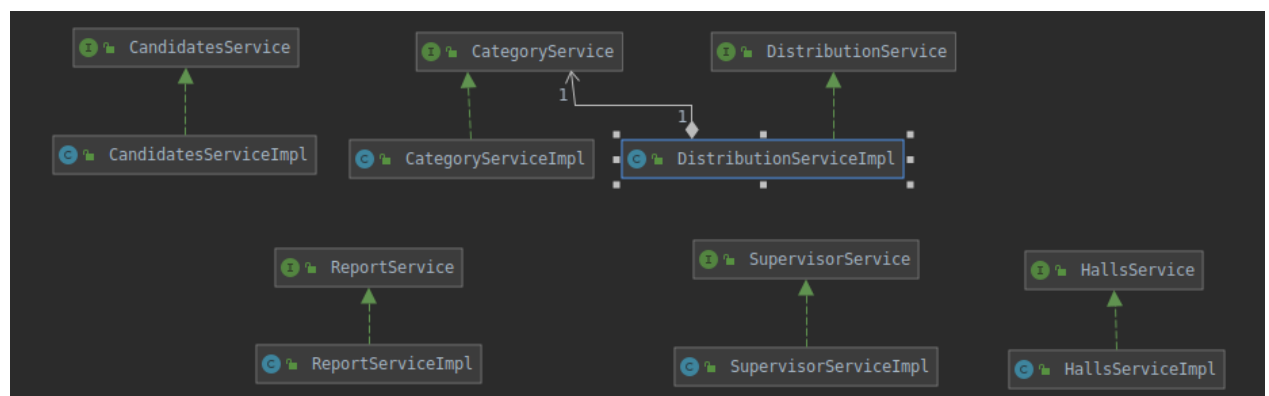
Aceste adnotări sunt următoarele:

- @RestController si @CrossOrigin aceste sunt aplicate la nivelul clasei. Prima are rolul de a specifica pentru fiecare metodă din cadrul clasei ca răspunsul acestora sa fie direct într-un răspuns HTTP ca JSON sau XML. Iar cea de a doua de a specifică care domenii au accesul către respectivele endpoint-uri din clasă.

- @GetMapping, @PostMapping, @DeleteMapping, @PutMapping aceste adnotări prezintă operațiile CRUD. Toate aceste adnotări au ca parametru o cale asociată.
- @PathVariable este folosită pentru a mapa o variabilă URI ca parametru în cadrul metodei folosite.
- @Valid are rolul de a valida informația
- @Autowired este folosită pentru a injecta în cazul de față serviciul CandidatesServices care se va ocupa de processarea datelor.

#### 4.4.2 Nivelul de business

În cadrul nivelului de business cunoscut ca și nivelul de serviciu este realizată translatarea operațiilor CRUD. Deși ajungem să avem mai multe clase acestea vor fi mai bine organizate și vom ajunge în a avea o separare a conceptelor mai bună.



**Figura 11 Interfețele și clasele folosite**

Aici vom verifica corectitudinea datelor prin toate validările și procesărilor necesare pentru a ne asigura de corectitudinea datelor. Acesta este reprezentată nivelul de legătură dintre cel de transfer de date și cel de persistență. Acesta nivel este folosit doar prin intermediul interfețelor pentru a nu expune implementarea, în acest fel la runtime utilizatorul va folosi doar interfețele.

În cadrul acest nivel sunt folosite multiple tipuri de clase POJO: DTO folosit în relația de comunicare cu nivelul de transfer de date, Model folosită la nivelul de serviciu, Entity atunci când comunică cu nivelul de persistență.

```

@Service
public class CategoryService {

    private static final Logger LOGGER = LoggerFactory.getLogger(CategoryService.class);

    @Autowired
    private CategoryRepository categoryRepository;

    @Autowired
    private CandidateRepository candidateRepository;

    public List<CategoryEntity> getAllCategoriesWithCandidates() {

        List<CategoryEntity> categories = categoryRepository.findAll();
        List<CategoryEntity> result = new ArrayList<>();
        for (CategoryEntity category : categories) {
            if (candidateRepository.findAllByCategory_Id(category.getId()).size() > 0) {
                category.setCandidateEntities(candidateRepository.findAllByCategory_Id(category.getId()));
                result.add(category);
            } else {

                LOGGER.info("Skipped the category with id [{}].", category.getId());
            }
        }
        return result;
    }
}

```

**Figura 12 Clasa CategoryServices**

În cadrul claselor din acest nivel cele mai comune adnotări vor fi `@Autowired` pentru a injecta repository din nivelul de persistență și `@Service`. `@Service` este doar o meta-annotation cu `@Component`, practic cele 2 vor fi tratate asemănător. Aceste meta-annotation permit dezvoltatorului să își customizeze codul.

#### 4.4.3 Nivelul de persistență

În acest nivel va conține toate interfețele adnotate cu `@Repository`. Precum și adnotarea `@Service` aceste două preia de la `@Component` posibilitatea acestora de scanare la rulare a claselor care implementează aceste interfețe. Pe lângă adnotarea `@Repository` aceste interfețe trebuie să extindă interfața **JpaRepository**. Folosind această interfață putem efectua interogări la baza de date.

Însă pentru a extinde acesta interfață sunt necesari 2 parametri, primul este o clasă entitate, această clasă mapează o tabelă, și ce al doilea reprezintă cheia pentru aceea tabelă care poate fi doar o primitivă(int, long, ect..) sau chiar un obiect în cazul în care avem o cheie compusă.

```
package com.sergiu.entity;

import javax.persistence.*;

@Entity
@Table(name = "candidates")
public class CandidateEntity {

    @Id
    @Column(name = "cnp")
    private Long cnp;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @OneToOne
    @JoinColumn(name = "category_id")
    private CategoryEntity category;

    @Column(name = "high_school")
    private String highSchool;

    @OneToOne
    @JoinTable(name = "distribution", joinColumns = {@JoinColumn(name = "cnp_candidate")}, inverseJoinColumns = {
        @JoinColumn(name = "id_hall")})
    private HallEntity hall;
```

**Figura 13 Clasa CandidatesEntity**

În Figura 7 avem prezentă entitatea pentru tabela Candidates. Această asociere cu tabela începe prin adnotarea @Entity. Acesta este folosită la nivelul de clasă și este obligatoriu. Această adnotare definește dacă o clasă poate fi mapată cu o tabelă, este precum un marker un exemplu ar fi interfața Serializable. @Table poate fi opțională fiind folosită pentru a specifica numele tablei.

@Column este folosită pentru a specifica numele tabele, @Id este pentru a identifica cheia primară. Însă pentru a identifica relații cu alte tabele sunt folosite următoarele adnotări:

@OneToMany, @OneToOne, @ManyToMany, care sunt puse deasupra obiectelor de instanță.

## 4.5 Tehnologiile folosite

### 4.5.1 Maven

Maven este un tool care faciliteaza administrarea proiectelor in Java. Acesta folosește un fișier XML numit POM sau Project Object Model. Acest fișier cuprinde diferite configurari ale proiectului, dependeșele din diferite proiecte externe si plugin-uri. Maven este responsabil pentru a descarca in mod dinamic toate dependențele(librariile) necesare.

### 4.5.2 Spring Boot

Spring Boot este un open source framework și are la bază framework-ul Spring care este orientat către micro servicii. El este dezvoltat de echipa Pivotal. Este folosit pentru a crea aplicații stand-alone și aplicații Spring care sunt gata de folosit. Dorită faptului ca framework-ul Spring a crescut în complexitate Spring Boot a fost inițiat ca o modalitate de a evita sa începi de la 0 și să salvezi timp. In timp ce framework-ul Spring este focusat pentru a oferi flexibilitate, Spring Boot țintește către o scurtare și o ușurare a căii pentru a dezvolta aplicații web.

Acesta oferă o modalitate mai simplă și mai rapidă de instalare, configurare și executare a aplicațiilor simple și bazate pe web.

Am decis să folosesc Spring Boot pentru următoarele beneficii:

- ✓ Este foarte ușor de dezvoltat aplicații bazate pe Spring cu Java.
- ✓ Reduce foarte mult din timpul de dezvoltare și îți crește timpul de productivitate.
- ✓ Totul este auto-configurat și nu este necesar nici o configurare manuală.
- ✓ Este foarte ușor de integrat cu ecosistemul oferit de Spring cum are fi Spring JDBC, Spring ORM, Spring Data, Spring Security ect.
- ✓ Acesta oferă un server HTTP incorporat precum Tomcat, Jetty, ect. .
- ✓ Oferă o mulțime de plugin-uri pentru a dezvolta și testa aplicația Spring Boot folosind instrumente precum Maven si Gradle.
- ✓ Oferă o mulțime de plugin-uri pentru a lucra cu baze de date încorporate foarte ușor.

### 4.5.3 Hibernate

Hibernate este un framework pentru limbajul java folosit pentru maparea unui model de domeniu orientat pe obiecte într-o bază de date pe relaționare. Acesta este un proiect open source și este foarte ușor de folosit. Beneficiul major care îl oferă acesta este faptul că simplifică relațiile cu baza de date ca simple obiecte in Java.

```
<dependency>

    <groupId>org.hibernate</groupId>

    <artifactId>hibernate-core</artifactId>

</dependency>
```

Fragment de cod: Dependță maven pentru hibernate

### 4.5.4 PostgreSQL

PostgreSQL - este un sistem de baze de date obiect-relațional puternic, care utilizează și extinde limbajul SQL combinat cu multe caracteristici care stochează în condiții de siguranță cele mai complicate sarcini de date.

```
<dependency>

    <groupId>postgresql</groupId>

    <artifactId>postgresql</artifactId>

</dependency>
```

Fragment de cod: Dependță maven pentru postgres

## 4.6 Algoritmi folosiți

### 4.6.1 Distribuirea

**Prima etapă** constă în selectarea numărului minim de săli care conțin numărul minim de locuri necesare pentru a susține examenul de admitere. Pentru a rezolva această problemă voi folosi backtracking care va selecta subsetul cel mai optim din punct de vedere a numărului de locuri. Acest subset va returna un set de săli care va fi folosit în cea de a doua etapă. Acest algoritm este cunoscut ca *Backtracking Algorithm for Subset Sum*.

*PSEUDOCOD:*

```
read -> halls; read-> nrCandidates; bestValue=Integer.MAX; solution = emptyList();
```

```
mapSolution =emptyMap(); //aici vom pastra toate soluțiile găsite
```

```
findAllSubset(halls, 0, 0, nrCandidates, solution, mapSolution, bestValue);
```

```
findAllSubset(hall, sum, startIndex, nrCandidates, solution, mapSolution, bestValue{
```

```
    if(nrCandidates == sum || sum> nrCandidates && sum < bestValue ){
```

```
        bestValue=sum;
```

```
        mapSolutions.put(bestValue, solution);
```

```
        solution.remove(lastElement);
```

```
        return;
```

```
    }
```

```
    else{
```

```
        for(i = startIndex; i<halls.size; i++){
```

```
            solution.add(halls[i]);
```

```
            findAllSubset(halls, sum+halls[i].space, i+1, nrCandidates, solution, mapSolutions ,bestValue);
```

```
        }
```

```
        solution = emptyList();
```

```
    }
```

```
}
```

A doua etapă constă în construirea unei tablele care va conține coeziunea dintre o categorie de candidați și o sală.

<div style="display: flex; align-items: center; justify-content: center;"> <div style="writing-mode: vertical-rl; transform: rotate(180deg);">C</div> <div>H</div> </div>		H1	H2	H3
		8	7	6
C1	10	-10/8	-10/7	-10/6
C2	5	5/8	5/7	5/6
C3	4	4/8	4/7	4/6

- H1, H2, H3 reprezintă sălile
- 8 7 6 reprezintă locurile disponibile din săli
- C1, C2, C3 reprezintă categoriile de candidați
- 10, 5, 4 reprezintă numărul candidaților din categorii

Tabelul de mai sus va fi completat folosind următoarea formulă stabilită:

$$\text{Cell}[C_i, H_j] = \begin{cases} x=0, & \text{dacă } H_i = 0 \text{ sau } C_j = 0 \\ x= C_j/H_i, & \text{dacă } H_i > C_j \\ x= 1, & \text{dacă } H_i = C_j \\ x= - (C_i \text{div } H_j), & \text{dacă } C_i > H_j \end{cases}$$

Următorul pas în algoritm este să selectăm câte o celulă în ordine descrescătoare a coeziunilor oferite de tabelă.



### *PSEUDOCOD:*

```
citeste -> setCells; //aceste celule vor fi ordonate dupa funcția de „coeziune”
cât timp(!setCells.empty()){
    cell =setCells.last();
    if(cell.getCohezion() >= 0){
        setCells.remove(cell);
    } else{
        if(cell.getChezion() ==1){
            listMove= cell.getCategory().getCandidates();
        }else{
            listMove=cell.getCategory().getCandidates().subList(0, cell.restOfCandidates());
        }
        size = listMove.size();
        cell.getHall().addCandidates(listMove);
        cell.getCanidates().remove(listMove);
    }
    refreshCells(setCells);//trebuie să restabilim coeziunile
}
```

Pentru a mă sigura că datele de intrare de la algoritmul de sus sunt corecte am folosit o clasă numită *Element* în cadrul aplicației care implementează interfața **Comparable**. Profitând de metoda publică `compareTo(Object o)` din această interfață am adăugat toate aceste elemente într-un `SortedSet`. Singurul lucru foarte important este acela de a cheama funcția ***refreshCells()*** pentru a actualiza elementele din cadrul `SortedSet`-ului.

### 4.6.2 Alocarea

Funcția prezenta mai jos parcurge rezultatele obținute de candidați. Candidați sunt așezați în ordine descrescătoare a notei finale obținute. Pentru fiecare candidate exceptând olimpicii va fi apletă funcția *getAllocationForCandidate()* care va trebuie să returnze lista în care va fi încadrat candidatul.

```
@Override
public void startAllocateCandidates() {
    if (gradeRepository.count() != candidateRepository.count()) {
        throw new FrameworkException("No toti candidatii au note!");
    }

    AllocationModel allocation = new AllocationModel(RO_BUGET, RO_TAXA, EN_BUGET, EN_TAXA, MD_RO_BUGET, MD_EN_TAXA);
    SortedSet<AdmissionResult> admissionResults = new TreeSet<>(admissionResultRepository.findAllByListNameIsNullOrListNameIsNot(ListAllocationType.L8));

    for (AdmissionResult admissionResult : admissionResults) {
        if (isOlympic(admissionResult)) {
            admissionResult.setFinalGrade(10.0);
            admissionResult.setListName(ListAllocationType.L1);
        } else {
            ListAllocationType listAllocationType = getAllocationListForCandidate(admissionResult, allocation);
            admissionResult.setListName(listAllocationType);
        }
        updateAdmissionResult(admissionResult);
        LOGGER.info("Candidatul:" + admissionResult.getCnp() + "a fost adaugat in lista" + admissionResult.getListName());
    }
}
```

**Figura 14** Functia principala de alocare a candidaților în săli

## 5 Manual de utilizare

### 5.1 Modul de instalare

Pentru a putea utiliza aplicația sunt necesare următoarele configurări:

#### 1. Java

- a) Trebuie să descărcăm JDK-ul de la linkul <https://java.com/en/download/>.
- b) Trebuie să rulăm executabilul.
- c) După finalizarea executabilului trebuie să adăugăm următoarele variabile de mediu JAVA\_HOME și JRE\_HOME.

În cazul utilizatorilor linux este necesară rularea comenzii: *sudo apt-get install default-jdk*

#### 2.PostgreSQL

- a) Trebuie să descărcăm versiunea dorită de la linkul: <https://www.postgresql.org/download/windows/>
- b) Trebuie să adăugăm variabila în mediu.(C:\Program Files\PostgreSQL\10\bin)

În cazul utilizatorilor linux este necesară rularea comenzii:

*sudo apt install postgresql postgresql-contrib*

#### 3.NodeJS

- a) Trebuie să descărcăm installerul de la linkul: <https://nodejs.org/en/>. Odată cu acesta va fi preinstalat și npm-ul.
- b) Trebuie să ne asigurăm că a fost instalat corect folosind comandă *node-version*.

În cazul utilizatorilor linux este necesară rularea comenzii: *sudo apt install nodejs*

#### 4.ReactJs

- a) Acesta vine direct la pachet cu *NodeJs*.

## 5.2 Utilizare

### 1.API

Pentru a putea porni API-ul avem nevoie de o bază de date. Configurațiile pentru acesta sunt specificate în fișierul *application.properties* din proiectul Maven.

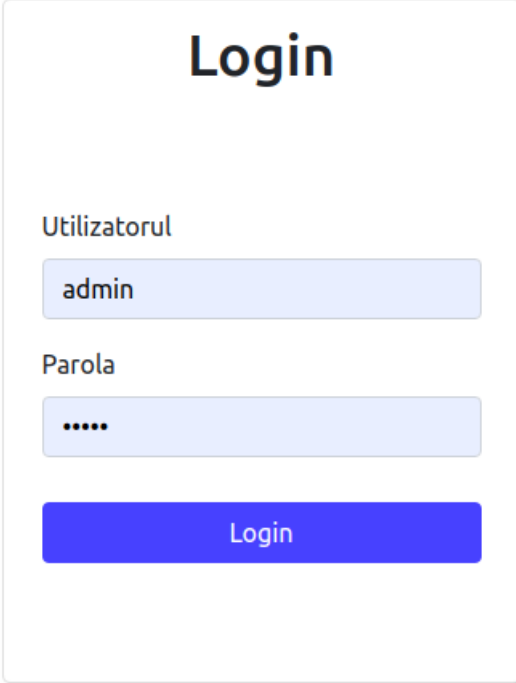
Acesta poate fi pornit din clasa *Application.java*. Când vom porni servărul acesta va folosi portul default 8080.

### 2.Clientul

Pentru a porni clientul este necesar să rulăm comanda:

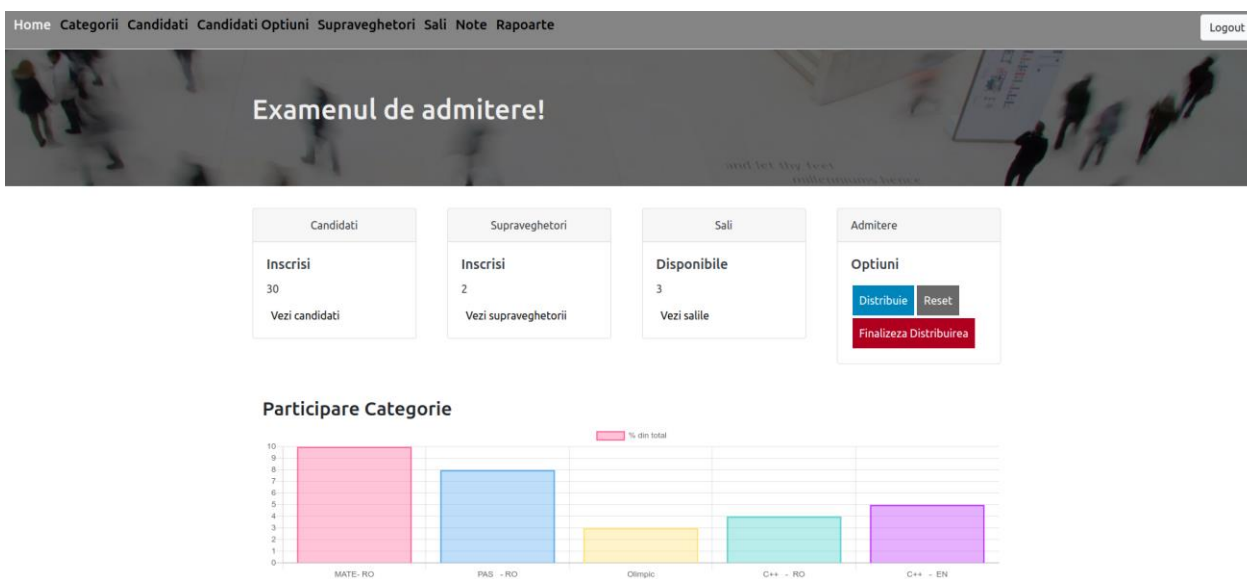
- *npm run start-local* dacă dorim să folosim versiunea locală a API-ului
- *npm start* dacă dorim să folosim versiunea din producție a API-ului.

### 3.Utilizarea

A screenshot of a web login form. The form is titled "Login" in a large, bold, black font. Below the title, there are two input fields. The first is labeled "Utilizatorul" (Username) and contains the text "admin". The second is labeled "Parola" (Password) and contains five dots, indicating a masked password. Below these fields is a blue button with the text "Login" in white.

**Figura 15 Pagina Login**

După ce utilizatorul s-a logat are control total asupra datelor. Acesta poate efectua operații de stergere, modificare și inserare a tuturor resurselor disponibile.



**Figura 16 Pagina Home**

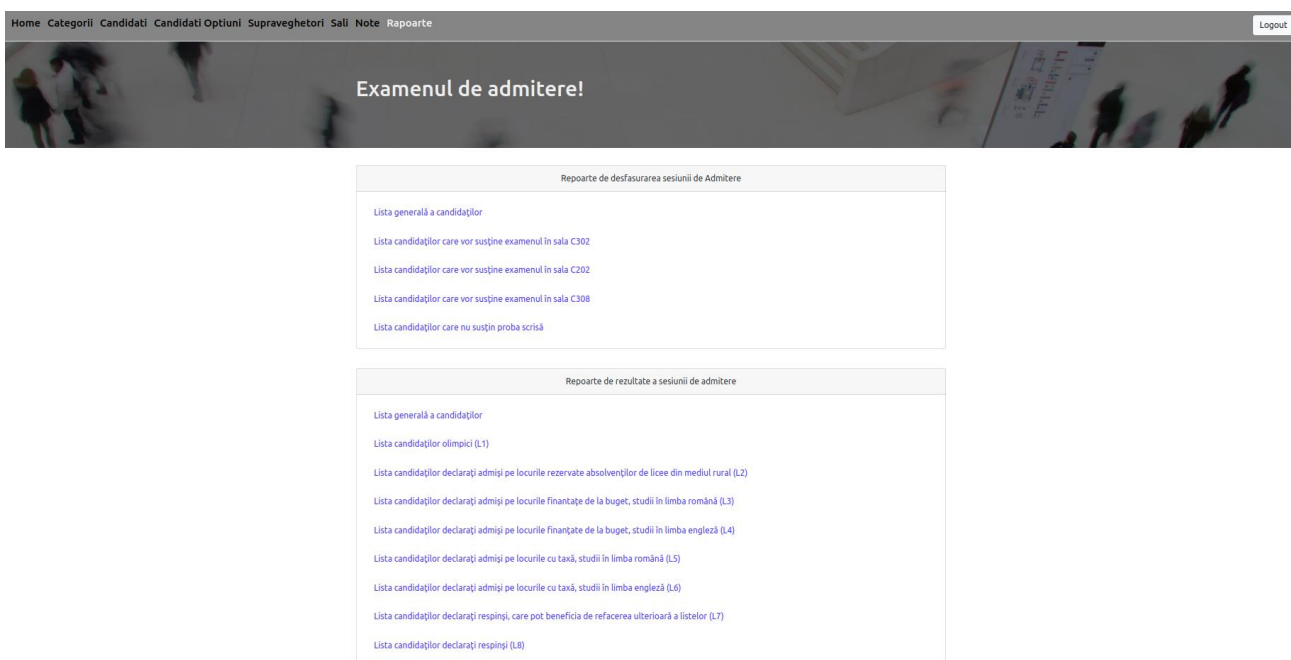
În cadrul paginii Home utilizatorul primește informații referitoare la situația curentă. Aceste informații sunt: numărul de candidați, numărul de supraveghetori, numărul de săli disponibile și un grafic care reprezintă opțiunea candidatului aleasă pentru a susține examenul de admitere.

**Examenul de admitere!**

CNP	Prenume	Nume	Mate/Info	Bac	Liceu	Categoria	Sala
3940122374...	Dan	Patrascu	9	8	Sportiv	MATE-RO	
3940122374...	Ghencea	Crinu	9.2	9.1	Sportiv	C++-EN	
3940122374...	Catalina-And...	Mihaila	10	9	Sportiv	PAS-RO	
3940122374...	Sergiu-Adrian	Samartean	8.8	6.4	Teoretic	C++-RO	
3940122374...	Catalina-And...	Volocaru	5.2	8.9	LKM	C++-EN	
3940122374...	Dorica	Volocaru	7.25	8.2	LKM	MATE-RO	
3940122374...	Mihai	Cretu	7.25	5.2	Teoretic	PAS-RO	
2940122374...	Catalina-And...	Apetrei	8	6.4	Racovita	PAS-RO	

**Figura 17 Pagina Candidați**

În pagina Canidați putem observa că, candidații nu a fost distribuiți neavând de altfel nici o sală în dreptul acestora. Aceștia vor fi așezați în săli după ce utilizatorul va apăsa buttonul de distribuie prezent în Figura 16 Pagina Home.



**Figura 18 Pagina Rapoarte**

În cadrul paginii de Raporte avem prezente 2 secțiuni de rapoarte. Prima secțiune va fi disponibilă după finalizarea distribuirii, mai exact prin apăsarea butonului Finalizare Distribuirea de la Figura 16 Pagina Home. Iar ce de a adoua secțiune va fi disponibilă după apăsarea butonului de Finalizare examen. Acest buton va apărea după ce s-a finalizat distribuirea și va putea fi folosit în cazul în care toți candidații care au participat la examen au notele înregistrate.

Home Categorii Candidati Candidati Optiuni Supraveghetori Sali Note Rapoarte									
Examenul de admitere!									
<div> <div> <div>Admitere</div> <div>Interiera</div> <div>Respinge</div> </div> <div>Search</div> </div>									
<input type="checkbox"/>	CNP	Prenume	Nume	Mate/Info	Bac	Liceu	Categoria	Sala	Respinge ...
<input type="checkbox"/>	19401223...	Sergiu-Adr...	Volocar	8	8.25	LKM	MATE-RO	C308	Respinge
<input type="checkbox"/>	29401223...	Catalina-A...	Apetrei	8	6.4	Racovita	PAS-RO	C302	Respinge

**Figura 19 Pagina Candidați Final**

După ce examenul a fost finalizat pe pagina Candidați avem posibilitatea de “Respinge” un candidat. Această funcționalitate este necesară de exemplu când un candidat își retrace dosarul sau alte motive.

## 6 Concluzii și direcții viitoare

### 6.1 Rezumat

Aplicația web pentru planificarea resurselor în cadrul examenului de admitere pune la dispoziție modalitate de distribuie a candidațiilor folosind cât mai puține resurse contribuind la o bună gestionare a examenului.

Aplicația este capabilă să producă rapoartele finale cu rezultate candidațiilor în aceeași manieră cu cele deja existente.

Aplicația este ușor de înțeles și ușor de folosit deoarece navigarea în cadrul acesteia este intuitivă.

### 6.2 Îmbunătățiri

Ar fi un plus pentru aplicație următoarele funcționalități:

1. Un sistem care oferă mesaje clare utilizatorului pentru a nu rămâne blocat.
2. Posibilitatea de a înregistra un utilizator.
3. Restricționarea unor anumite funcționalități într-un anumită stare a aplicației.
  - a. Să nu ai posibilitatea să introduci/ștergi: un candidat, o categorie, dacă deja s-a finalizat distribuirea.
  - b. Să nu fie afișate anumite pagini precum Note și Rapoarte dacă nu s-a finalizat distribuirea
4. Adăugarea mai multor rapoarte.

## Bibliografie

<https://reactjs.org/tutorial/tutorial.html>

<https://dzone.com/articles/what-is-spring-boot>

<https://www.kennethlange.com/books/The-Little-Book-on-REST-Services.pdf>

<https://www.guru99.com/api-vs-web-service-difference.html>

<https://medium.com/@cogziesys/web-api-why-to-choose-and-benefits-34139e84fd50>

<https://www.guru99.com/restful-web-services.html>

<https://www.petrikainulainen.net/software-development/design/understanding-spring-web-application-architecture-the-classic-way/>

<https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/>

<http://allenfang.github.io/react-bootstrap-table/start.html>



## Anexe

Figura 1: Digrama use-case a aplicației .....	12
Figura 2 Diagrama de activitate pentru creerea unui raport .....	13
Figura 3 Arhitectura generală a aplicației .....	14
Figura 4 Schema bazei de date .....	16
Figura 5 Diagrama de pachete .....	18
Figura 6 Diagrama de clase pentru Candidat.....	19
Figura 7 Componenta folosită pentru a crea cereri către API.....	23
Figura 8 Cross Origin.....	31
Figura 9 Clasele care se ocupă cu interceptările request-urilor .....	32
Figura 10 Clasa CandidatesController .....	33
Figura 11 Interfețele și clasele folosite .....	34
Figura 12 Clasa CategoryServices .....	35
Figura 13 Clasa CandidatesEntity .....	36
Figura 15 Funcția principală de alocare a candidaților în săli .....	42
Figura 16 Pagina Login.....	44
Figura 17 Pagina Home.....	45
Figura 18 Pagina Candidați .....	45
Figura 19 Pagina Rapoarte .....	46
Figura 20 Pagina Candidați Final .....	46