

UNIVERSITATEA ALEXANDRU IOAN CUZA IAȘI
FACULTATEA DE INFORMATICĂ



LUCRARE DE LICENȚĂ

Platformă destinată interacțiunii dintre doctor și pacient : DOC Web

propusă de

Plugariu Raluca Nicoleta

Sesiunea: Iulie, 2019

Coordonator științific

Lect. Dr. Frasinaru Cristian

DECLARAȚIE PRIVIND ORIGINALITATEA ȘI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licență cu titlul „DOC Web” 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:

- toate fragmentele de text reproduse exact, chiar și 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 autor și dețin referințe precise;
- rezumarea ideilor altor autori precizează referința precisă la textul original.

Iași,

Absolvent Plugariu Raluca Nicoleta

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „DOC Web”, 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,

Absolvent Plugariu Raluca Nicoleta

Acord privind proprietatea dreptului de autor

Facultatea de Informatică este de acord ca drepturile de autor asupra programele-calculator, format executabil și sursă, să aparțină autorului prezentei lucrări, Plugariu Raluca Nicoleta
Închierea acestui acord este necesară din următoarele motive:

Decan Adrian Iftene

Absolvent Plugariu Raluca Nicoleta

Cuprins

1.Introducere	6
2.Contribuții.....	7
3. Aplicații similare.....	7
4.Tehnologii utilizate	8
4.1 Java	8
4.2 Spring.....	9
4.3 Java Server Faces (JSF)	10
4.4 H2 Database	11
4.5 Maven	11
5. Analiză și proiectare.....	12
5.1 Diagrama Use Case.....	12
5.2 Diagrama de entități.....	14
5.3 Arhitectura.....	15
5.5 Structura bazei de date	16
5.4 Structura Aplicației	20
5.5 Detalii de implementare	23
6. Modul de utilizare al aplicației	31
6.1 Rolul Doctor.....	32
6.2 Rolul Pacient.....	37
7. Concluzii și direcții viitoare.....	41
Bibliografie	42

1.Introducere

Lucrarea de față prezintă crearea unei aplicații menită să ușureze obținerea interacțiunii dintre un pacient și un doctor, precum și organizarea și monitorizarea pacienților și a programărilor, din perspectiva unui doctor.

Ideea proiectului a venit odată cu necesitatea menținerii unei bune conexiuni între aceste două entități, doctor și pacient. Din experiența personală, am observat că necesita mult timp investit în găsirea unui doctor de care noi am avea nevoie, care să îndeplinească anumite criterii după care noi căutăm, și nu în ultimul rând, disponibilitatea acestuia în vederea programării unei vizite. Desigur că aceste funcționalități descrise mai sus, pot fi găsite deja în diferite platforme, cum ar fi cele oferite de instituțiile medicale, dar ideea acestui proiect este să le aducă împreună într-o singură platformă.

Este o aplicație care se axează mai mult pe nevoile unui pacient, deoarece acesta poate găsi un doctor în funcție de recenziile pe care acesta le deține, provenind de la alți utilizatori, după locația în care se află, iar acesta este un lucru foarte important pentru persoanele care se află într-o zonă sau oraș în care nu dețin informații legate de rețeaua de medici, și nu în ultimul rând, pacienții primesc informații legate de programul doctorilor și intervalele libere pentru consultații.

Din perspectiva unui doctor, cel mai mare avantaj pe care îl aduce această aplicație este că nu trebuie să aparțină unei anumite instituții pentru a avea un profil aici, ceea ce ajută la o vizibilitate mult mai bună pe piață. Cum spuneam, platformele deja existente, sunt create de instituții medicale, în mare parte din mediul privat. Exemplul concret din Iași este Arcadia, care pune la dispoziție profilul doctorilor din cadrul instituției. Dar nu multe sunt acestea, drept urmare, există în Iași doctori foarte bine pregătiți dar care nu sunt vizibili publicului larg. Pe lângă acest aspect, în cadrul aplicației doctorul își poate gestiona lista pacienților și a calendarului și poate răspunde la interacțiunile cu pacientul : acceptă sau refuză cereri de programari.

2. Contribuții

Platforma este un produs destinat doctorilor din orice domeniu, dar și publicului larg care are nevoie de a interacționa cu aceștia.

Logica aplicației este concepută în totalitate de mine, ea oferă câte o interfață pentru fiecare tip de user – doctor , pacient. Acestea au fost create utilizând framework-uri precum Primefaces și Bootfaces, iar template-ul general este inspirat din produsul gratuit AdminLTE 2, o interfață bazată în mare parte pe Bootstrap, ce oferă utilizatorilor o experiență plăcută.

3. Aplicații similare

Acest capitol propune o analiză sumară a unor soluții similare existente pe piață, scoțând în evidență diferite abordări pentru acest tip de aplicație.

3.1 Clickdoc

3.2 Doctor On Demand

3.3 Online Doctor

4. Tehnologii utilizate

4.1 Java

“Java este o tehnologie inovatoare lansată de compania Sun Microsystems în 1995, care a avut un impact remarcabil asupra întregii comunități, a dezvoltatorilor de software, impunându-se prin calități deosebite cum ar fi simplitate, robustețe și nu în ultimul rând portabilitate. Denumită inițial OAK, tehnologia Java este formată dintr-un limbaj de programare de nivel înalt pe baza căruia sunt construite o serie de platforme destinate implementării de aplicații pentru toate segmentele industriei software.”

API (Application Programming Interface)

Un API este un set de reguli folosit pentru comunicarea dintre o aplicație cu o altă aplicație. API este de multe ori corelat cu o librărie software, primul descrie specificațiile, iar cel din urmă descrie implementarea acestora. Un lucru important de menționat este că atunci când trebuie să folosim un API, nu trebuie să cunoaștem nici un detaliu tehnic cu privire la modul în care a fost scris.

J2EE (Enterprise Edition)

J2EE este o platformă Java ce pune la dispoziție o colecție de API-uri, care vine în ajutorul dezvoltatorilor pentru a crea aplicații server-side, deținută în momentul actual de Oracle. Această platformă conține elemente ce pot fi împărțite în 2 categorii:

1. Componente :
 - Servlets
 - JSP (Java Server Pages)
 - EJB (Enterprise Java Beans)
2. Servicii/API
 - JMS (Java Message Service)
 - JTA (Java Transaction API)
 - JAAS (Java Authentication and Authentication Service)
 - JNDI (Java Naming and Directory Interface)
 - Java Mail API
 - JAX-WS (Java API for XML Web Services)
 - JAXB (Java Architecture for XML Binding)
 - SOAP
 - JPA (Java Persistence API)
 - JSF (Java Server Faces)

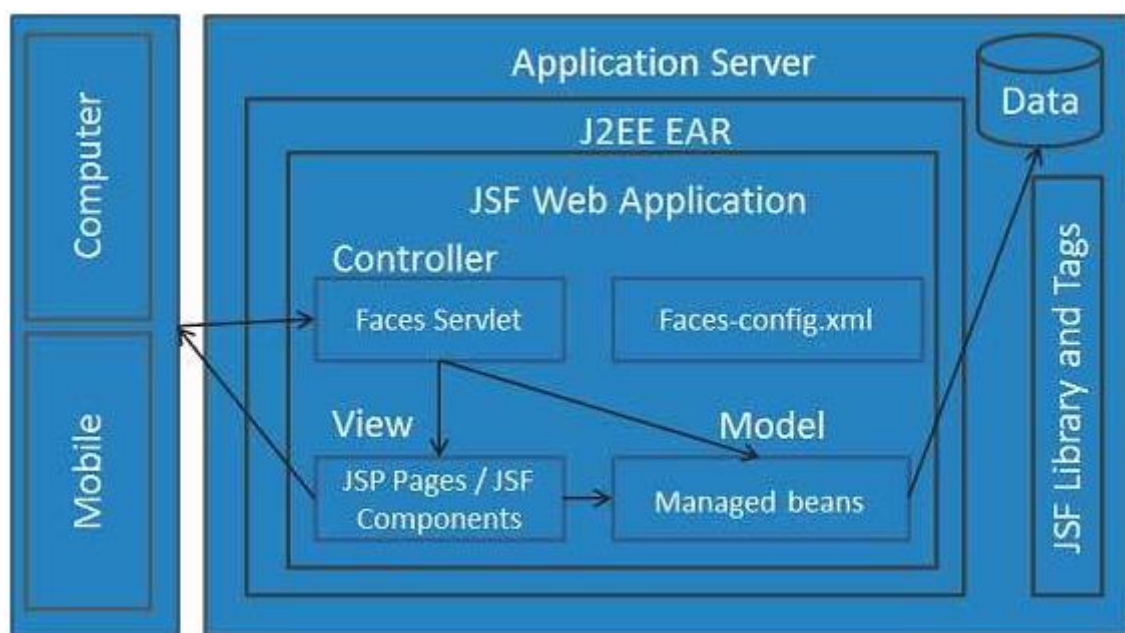
4.2 Java Server Faces (JSF)

JSF este un framework pentru aplicațiile Web ce intenționează să simplifice dezvoltarea interfețelor utilizatorului. Specificațiile JSF definesc un set standard de componente UI (User Interface) și oferă un API (Application Programming Interface) pentru dezvoltarea lor.

JSF reduce efortul pentru crearea și mentenanța aplicațiilor ce vor rula pe partea de server. Câteva dintre facilitățile pe care această tehnologie le oferă sunt:

- Furnizează componente User Interface reutilizabile.
- Face posibil transferul de date rapid între componentele User Interface.
- Manageriază stările User Interface între multiple cereri către server.
- Permite implementarea componentelor proprii.

Tehnologia JSF este bazată pe arhitectura MVC (Model View Controller) cu scopul de a separa partea logică a aplicației de partea de prezentare.



4.3 Spring Framework

Spring este un framework pentru dezvoltarea aplicațiilor Java, mult mai ușor de folosit în comparație cu modul de dezvoltare prevăzut în folosirea Java2EE. Acest framework utilizează diverse tehnici noi precum Aspect-Oriented-Programming (AOP), injecția de dependență (DI) sau Plain-Old-Java-Object (POJO). Se axează în principal pe furnizarea de diverse modalități de a ajuta la gestiunea business object-urilor. Dezvoltarea aplicațiilor Web este mult mai ușoară în comparație cu cadrul clasic Java și interfețele de programare a aplicațiilor (API), cum ar fi conectivitatea bazei de date Java (JDBC), paginile JavaServer (JSP) și Java Servlet.

Poate fi considerat ca o colecție de submodule sau layere, cum ar fi Spring AOP, Spring Object-Relational Mapping (Spring ORM), Spring Web Flow și Spring Web MVC, care pot fi folosite împreună pentru a furniza toate funcționalitățile necesare în dezvoltarea unei aplicații web.

Cele mai importante elemente din cadrul Spring sunt:

- **Containerul IoC:**
Este containerul de bază care utilizează modelul DI sau IoC pentru a furniza implicit o referință de obiect într-o clasă în timpul rulării. Acest model acționează ca o alternativă la modelul de localizare a serviciului.
- **Data access:**
Permite dezvoltatorilor să utilizeze API-uri persistente, cum ar fi JDBC și Hibernate, pentru stocarea datelor de persistență în baza de date.
- **Spring MVC:**
Are la bază arhitectura MVC. Toate solicitările făcute de un utilizator parcurg mai întâi controlerul și apoi sunt expediate către diferite vizualizări, adică către diferite pagini JSP sau servlets.
- **Gestionarea tranzacțiilor:**
Ajută la gestionarea tranzacțiilor unei aplicații fără a afecta codul acesteia. Are la bază Java Transaction API (JTA) pentru tranzacțiile globale gestionate de un server de aplicații și tranzacțiile locale gestionate utilizând JDBC Hibernate, Java Data Objects (JDO) sau alte API-uri de acces la date.
- **Serviciul Web:**
Generează endpointurile și definițiile serviciului web bazate pe clase Java, dar este dificil să le gestionezi într-o aplicație. Pentru a rezolva această problemă, serviciul Spring Web oferă abordări bazate pe straturi care sunt gestionate separat de parsarea Extensible Markup Language (XML) (tehnica de citire și manipulare a XML-ului).
- **Nivelul de abstractizare JDBC:**
Ajută utilizatorii să gestioneze erorile într-un mod ușor și eficient. Codul de programare JDBC poate fi redus când acest strat de abstractizare este implementat într-o aplicație Web. Acest strat gestionează excepții, cum ar fi DriverNotFound.

4.4 H2 Database

H2 este o bază de date open source bazată pe Java. Acesta poate fi încorporat în aplicații Java sau poate fi executat în modul client-server. În principal, baza de date H2 poate fi configurată să funcționeze ca bază de date în memory sau pe disc.

Această bază de date poate fi utilizată în modul embedded sau în modul server.

Principalele caracteristici ale bazei de date H2 :

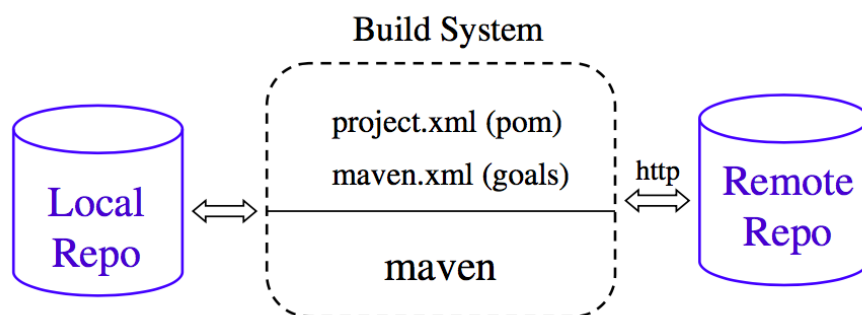
- Extrem de rapid, open source, JDBC API
- Disponibil în modurile embedded și server; în baze de date în memorie
- Consola bazată pe browser
- Amprentă mică - dimensiune de fișier de aproximativ 1,5 MB

4.5 Maven

Maven este o aplicație de tip open-source oferită de Apache menită să ușureze management-ul proiectului, automatizând infrastructura de compilare. Timpul de lucru scade, iar odată cu acesta costul de producție al unui produs software se micșorează. Acest lucru este realizat prin crearea unei structuri standard de directoare și un ciclu de viață prestabilit pentru compilare.

În esență, Maven este util și adeseori folosit în proiecte de dimensiuni mari cu multiple submodule, deoarece reușește să compileze proiectul ca un tot unitar, oferind în mod automat informații despre proiect și ajutând echipele de programatori să colaboreze la același proiect.

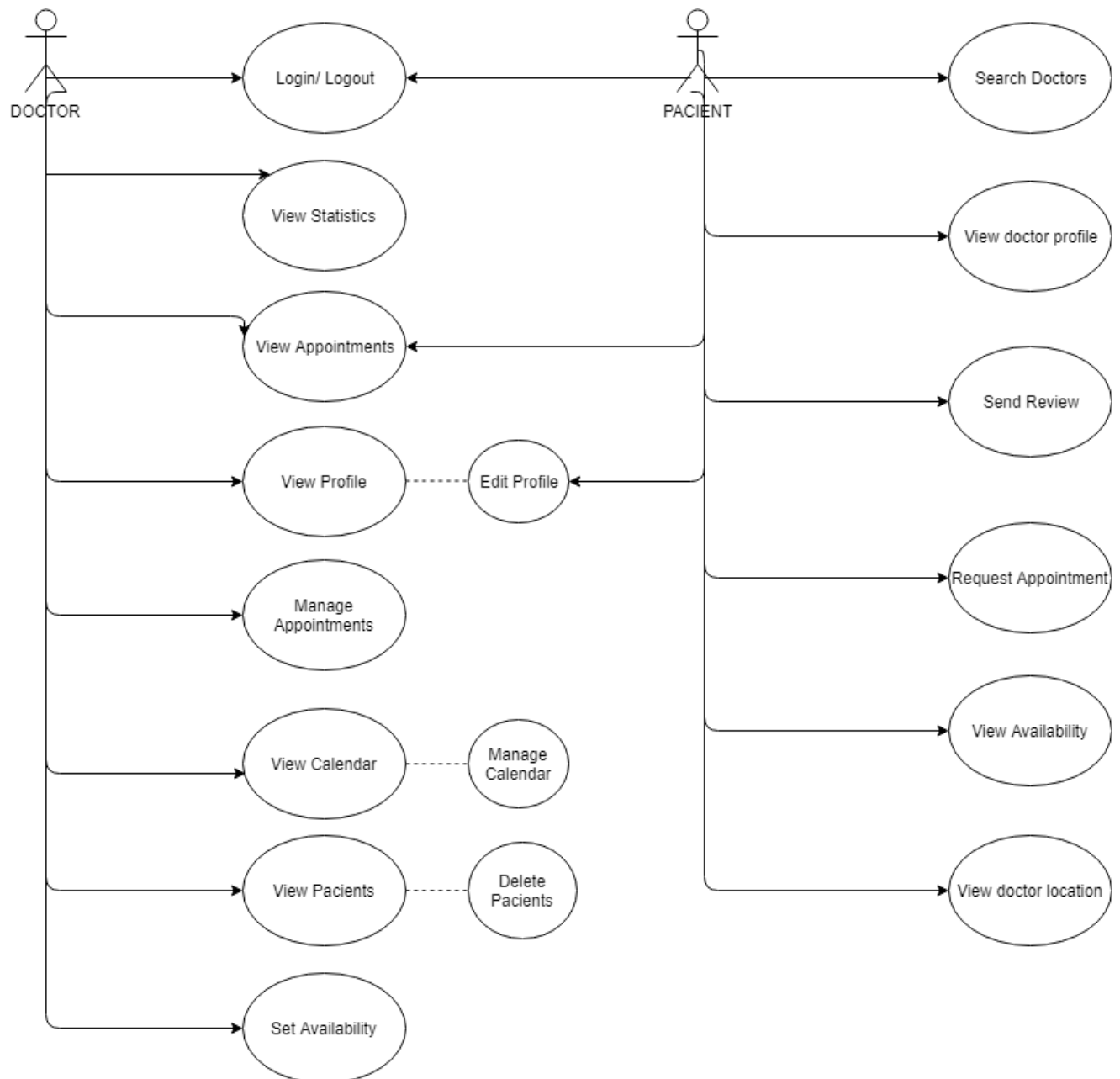
Structura proiectului este definită într-un fișier descriptiv unic, numit pom.xml, denumirea sa făcând referire la Project Object Model (POM). Acesta conține diferite informații, printre care detaliile ce țin de configurare, dependențe, informații despre resursele de testare, membrii echipei, etc. Dependențele sunt folosite pentru a importa automat librăriile necesare rulării.



5. Analiza si proiectare

Aplicația DOCWeb își propune să ofere utilizatorilor o platforma cat mai simpla și usor de accesat, în vederea interacțiunii dintre doctori și pacienți, precum și sisteme de gestionare a unor diferite entități.

5.1 Diagrama Use Case



Conform diagramei putem deduce că aplicația ar trebui să ofere:

- Autentificarea și autorizarea în funcție de rolul utilizatorului în aplicație

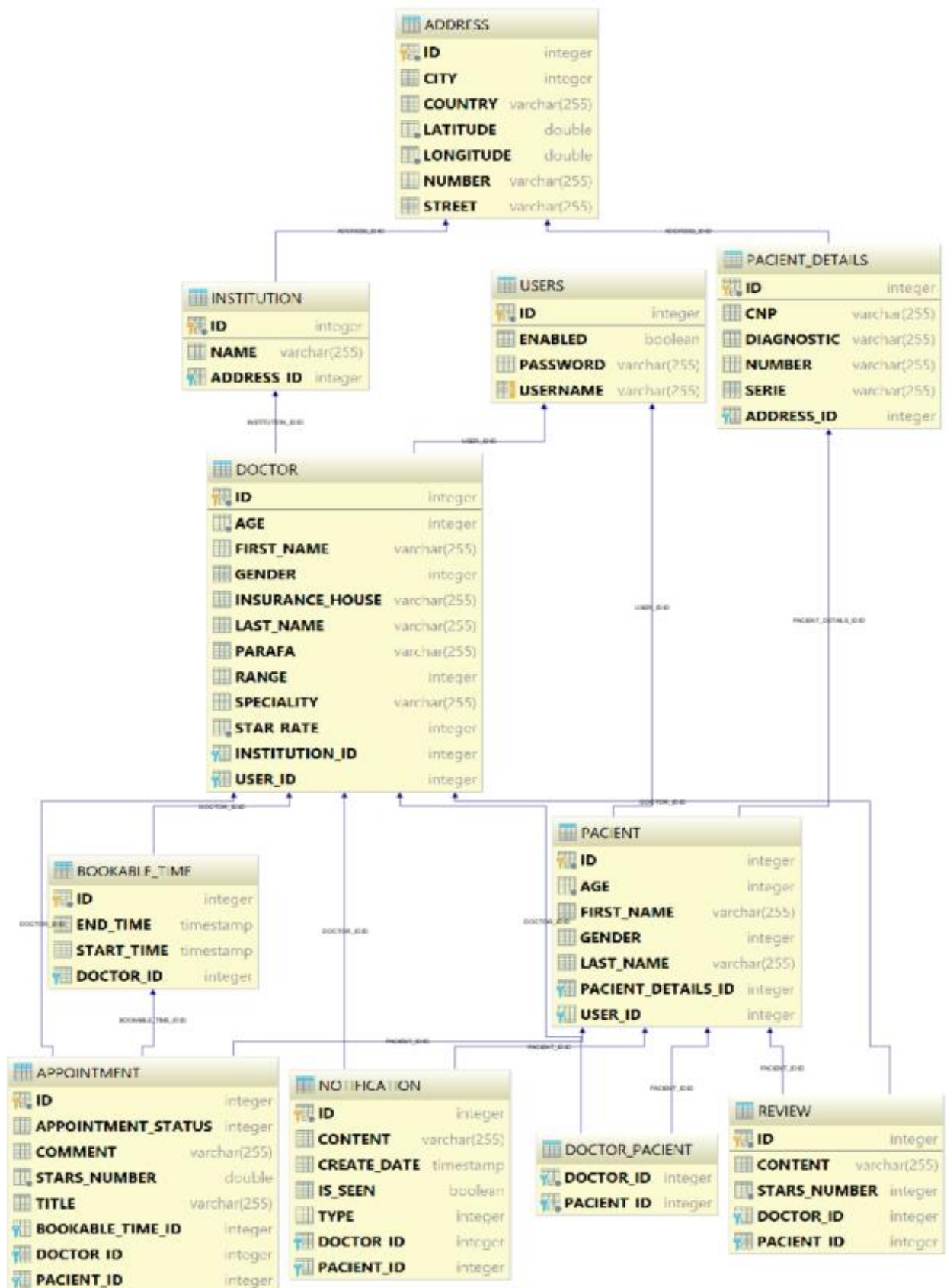
În cazul în care utilizatorul logat are rolul de doctor poate să:

- Afișeze/Editeze programările
- Steargă o programare
- Afișeze/Editeze profilul
- Afișeze calendarul
- Seteze orele și zilele la care este disponibil pentru programări
- Afișeze/Șteargă pacienții
- Editeze calendarul
- Accepte/Refuze o programare
- Primească notificari în privința cererilor de programare dar și pentru adaugarea unui nou review pe profilul său

În cazul în care utilizatorul logat are rolul pacient poate să:

- Afiseze programările sale
- Caută un doctor în funcție de locație, scor și specializare
- Afișeze profilul doctorului
- Afișeze orele la care doctorul este disponibil pentru o programare
- Adaugă un review pe profilul doctorului
- Trimite o cerere pentru o programare la o anumită oră
- Afișeze/Editeze profilul
- Primească notificari în privința editării unei programări deja stabilite

5.2 Diagrama de entități



Conform diagramei putem deduce că:

- Un doctor are mai mulți pacienti
- Un pacient are mai mulți doctori
- Un doctor are mai multe notificări
- Un doctor are mai multe programări
- Un doctor are mai multe review-uri
- Un doctor are mai multe intervale de timp destinate programărilor
- Un pacient are mai multe programări
- Un pacient are mai multe review-uri trimise doctorilor
- O programare are asignat un doctor și un pacient

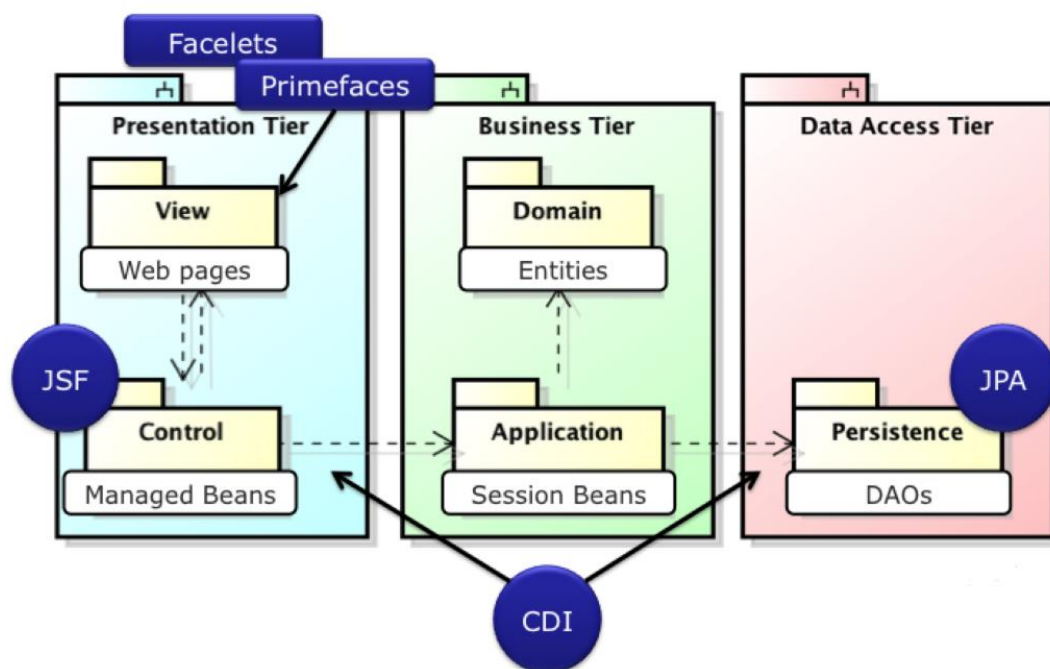
5.3 Arhitectura

DOCWeb este o aplicație bazată pe o arhitectură structurată pe 3 nivele :

1. Nivelul de prezentare sau client- conține elemente JSP, Servlet și framework-ul JSF, un framework bazat pe design patternul MVC – Model View Controller, modelul fiind constituit din Managed Beans, view-ul îl reprezintă pagini xhtml cu componente Primefaces, iar controller-ul este reprezentat de Faces Servlet, ce se ocupă de preluarea cererilor utilizatorului și rezolvarea acestora.

2. Nivelul de mijloc denumit și nivelul logic – este alcătuit din servicii EJB ce mențin logica aplicației și entități ce constituie modelul aplicației.

3. Nivelul de date sau EIS (Enterprise Information Systems) – gestionează conexiunea dintre entități și baza de date. Persistența datelor este făcută de Hibernate - o implementare a interfețelor oferite de JPA.



5.4 Structura bazei de date

Baza de date utilizată de aplicația OEC conține 12 tabele conform diagramei de entități. Tabele sunt utilizate pentru a păstra date despre doctori, pacienti, locatii, institutii, notificări, conturi , roluri dar și pentru a păstra legăturile @ManyToMany.

Punctul de start al aceste aplicații este autentificarea care, din punct de vedere al bazei de date, a necesitat utilizarea a doua tabele: USERS si AUTHORITIES;

USERS	
ID	integer
ENABLED	boolean
PASSWORD	varchar(255)
USERNAME	varchar(255)

Tabelul USERS are ca și cheie primară id-ul care va fi incrementat automat la crearea unei noi entități, cu ajutorul Hibernate-ului prin secvente Oracle, lucru care se va întâmpla în cazul tuturor entităților din cadrul aplicației. Celelalte 3 proprietăți ale acestei tabele sunt folosite în cadrul validării autentificării de către Spring Security.

AUTHORITIES	
ID	integer
AUTHORITY	varchar(255)
USERNAME	varchar(255)

Tabelul AUTHORITIES are ca scop atribuirea unui rol fiecărui user, recunoscut prin proprietatea de USERNAME.








Proprietatea AUTHORITY indică tipul de rol pe care userul îl deține, iar acesta poate avea valoarea "ROLE_DOCTOR" sau "ROLE_PACIENT" și în funcție de această valoare se va face accesul la view-urile din aplicație, corespunzătoare fiecărui rol.

DOCTOR	
ID	integer
AGE	integer
FIRST_NAME	varchar(255)
GENDER	integer
INSURANCE_HOUSE	varchar(255)
LAST_NAME	varchar(255)
PARAFA	varchar(255)
RANGE	integer
SPECIALITY	varchar(255)
STAR_RATE	integer
INSTITUTION_ID	integer
USER_ID	integer

Tabelul DOCTOR este reprezentarea uneia dintre cele două entități căiera i se adresează aplicația. Așadar, pe lângă cheia primară și identificatorul unic al tabelului :id-ul, acesta mai stochează diferite informații precum : nume, prenume, vârstă, gen, nivel de senioritate, sau scorul obținut din review-urile pe care acesta le primește. Proprietatea de specializare are valori predefinite, stocate într-un enum, pentru a păstra o consistență în valorile stocate în această coloană, dar și pentru a evita încărcarea bazei de date cu noi tabele și legături.







Un doctor aparține unei instituții iar această relație este reprezentată de cheia străină atribuită coloanei institution_id , iar cum entitatea DOCTOR este un user în această aplicație, implică faptul că este nevoie de o cheie străină care să facă legatura cu tablul USERS, iar tocmai acest lucru îl reprezintă

coloana user_id.








PACIENT	
 ID	integer
 AGE	integer
 FIRST_NAME	varchar(255)
 GENDER	integer
 LAST_NAME	varchar(255)
 PACIENT_DETAILS_ID	integer
 USER_ID	integer

Tabelul PACIENT reprezintă cea de-a doua entitate importantă în cadrul acestei aplicații, având ca identificator unic, id-ul ce este cheie primară. Acest tabel conține principalele informații legate de o entitate de tip pacient și anume: vârsta, numele, prenumele și genul. Restul informațiilor ce vor alcătui profilul unui pacient se regăsesc în tabelul PACIENT_DETAILS, iar legătura cu acest tabel se realizează prin cheia străină atribuită coloanei pacient_details_id, care reprezintă id-ul intrării din acel tabel, corespunzător pacientului curent.

La fel ca și în cazul doctorului, entitatea pacient este tot un tip de user în cadrul aplicației așa că acesta deține o proprietate de tip USER, iar legătura cu acest tabel este mapată cu ajutorul coloanei user_id.



PACIENT_DETAILS	
 ID	integer
 CNP	varchar(255)
 DIAGNOSTIC	varchar(255)
 NUMBER	varchar(255)
 SERIE	varchar(255)
 ADDRESS_ID	integer

Așa cum spuneam, tabelul PACIENT_DETAILS a fost creat cu scopul de a organiza mai bine datele ce țin de un pacient. Așadar el conține informații ce vin și în folosul doctorului precum: cnp, serie, număr, diagnostic. Un pacient are asignată o adresă, iar coloana address_id reprezintă cheia străină care face legătura cu tabelul ADDRESS în care vor fi stocate informațiile legate de locația pacientului în funcție de care se va face căutarea doctorilor.




ADDRESS	
 ID	integer
 CITY	integer
 COUNTRY	varchar(255)
 LATITUDE	double
 LONGITUDE	double
 NUMBER	varchar(255)
 STREET	varchar(255)

Tabelul ADDRESS conține intrări ce reprezintă informații legate de locația unei entități, aceasta fiind atribuită fie unei instituții, fie unui pacient, așadar identificatorul unic al acestui tabel este id-ul.

Restul atributelor: city, country, latitude, longitude, number, street, sunt menite să ofere toate informațiile necesare în vederea orientării și localizării unei entități.









DOCTOR_PACIENT	
 DOCTOR_ID	integer
 PACIENT_ID	integer

Tabelul DOCTOR_PACIENT este un tabel de legătură între entitatea pacient - doctor, ce reprezintă relația de @ManyToMany, în sensul în care un doctor poate avea mai mulți pacienți, iar un pacient poate aparține mai multor doctori. Așadar el conține cheile străine reprezentate de id-ul doctorului, respectiv id-ul pacientului.

INSTITUTION	
 ID	integer
 NAME	varchar(255)
 ADDRESS_ID	integer

Tabelul INSTITUTION conține câteva informații legate de locul de muncă asignat unui doctor. Această legătură cu doctorul am vazut-o mai sus cum s-a mapat cu ajutorul unei cheie străine în cadrul tabelului DOCTOR ce reprezintă id-ul instituției.





Nu în ultimul rând, o instituție are o adresă, iar aceasta este reprezentată de către coloana address_id.

APPOINTMENT	
 ID	integer
 APPOINTMENT_STATUS	integer
 COMMENT	varchar(255)
 STARS_NUMBER	double
 TITLE	varchar(255)
 BOOKABLE_TIME_ID	integer
 DOCTOR_ID	integer
 PACIENT_ID	integer

Tabelul APPOINTMENT conține informații ce descriu o programare, precum : titlu, status, comentariu din partea pacientului, data la care va avea loc și bineînțeles, doctorul care la care se face programarea și pacientul care o solicită.

Așadar putem spune ca un doctor poate avea mai multe programări cu diferiți pacienți, iar un pacient poate avea mai multe programări la diferiți doctori. Aceste relații de @OneToMany (one doctor - many appointments, one pacient - many appointments) sunt formate cu ajutorul cheilor străine : doctor_id și pacient_id, care indică id-urile entităților corespunzătoare.






Informațiile legate de data la care are loc programarea sunt structurate în tabelul BOOKABLE_TIME, iar legatura cu acele date se realizează prin coloana bookable_time_id.

BOOKABLE_TIME	
 ID	integer
 END_TIME	timestamp
 START_TIME	timestamp
 DOCTOR_ID	integer

Tabelul BOOKABLE_TIME a fost creat pentru o mai bună reprezentare a unei date la care este creată o programare, sau care este disponibilă pentru o programare. Legatura cu tabelul ce reprezintă o programare : APPOINTMENT am descris-o puțin mai sus.

Tabelul conține informații ce descriu un interval de timp : start_time și end_time. Deasemenea, aici este regăsim și reprezentarea relației de *@OneToMany* cu tabelul DOCTOR (un doctor poate avea mai multe bookable time-uri) prin coloana doctor_id, cheie străină ce indica id-ul doctorului ce

deține o intrare de tipul bookable_time.








REVIEW	
 ID	integer
 CONTENT	varchar(255)
 STARS_NUMBER	integer
 DOCTOR_ID	integer
 PACIENT_ID	integer

Tabelul REVIEW conține informații legate de review-ul pe care un pacient are oportunitatea de a-l crea pe pagina unui doctor și anume : conținutul sau mesajul review-uli și nota pe care acesta dorește să o atribuie, pe o scara de la 0 la 5.

Un doctor poate avea mai multe review-uri, așadar coloana doctor_id conține o cheie străină care realizează relația de *@OneToMany*.

Iar un pacient poate trimite review-uri mai multor doctori, iar pentru a putea ține o evidență asupra autorului unui review, trebuie create legatura între tabelul REVIEW și PACIENT, aceasta fiind definită cu ajutorul coloanei

pacient_id.

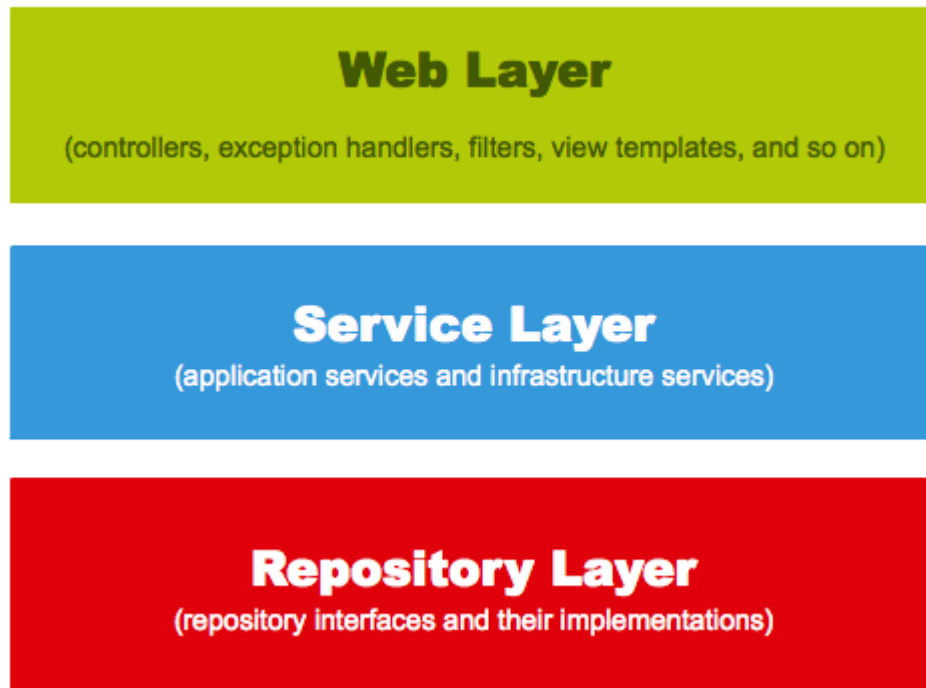
NOTIFICATION	
 ID	integer
 CONTENT	varchar(255)
 CREATE_DATE	timestamp
 IS_SEEN	boolean
 TYPE	varchar(255)
 DOCTOR_ID	integer
 PACIENT_ID	integer

Tabelul NOTIFICATION conține date necesare în construirea unei alerte destinate unui doctor sau unui pacient : conținutul notificării, data la care a fost declanșată, tipul notificării care precizează contextul din care este alerta și un câmp de tip boolean , is_seen, care precizează dacă notificarea a fost verificată de carte doctor sau nu.

Un doctor poate avea mai multe notificări, această relație fiind rezolvată de cheia străină din coloana doctor_id, respectiv, un pacient poate avea mai multe notificări.

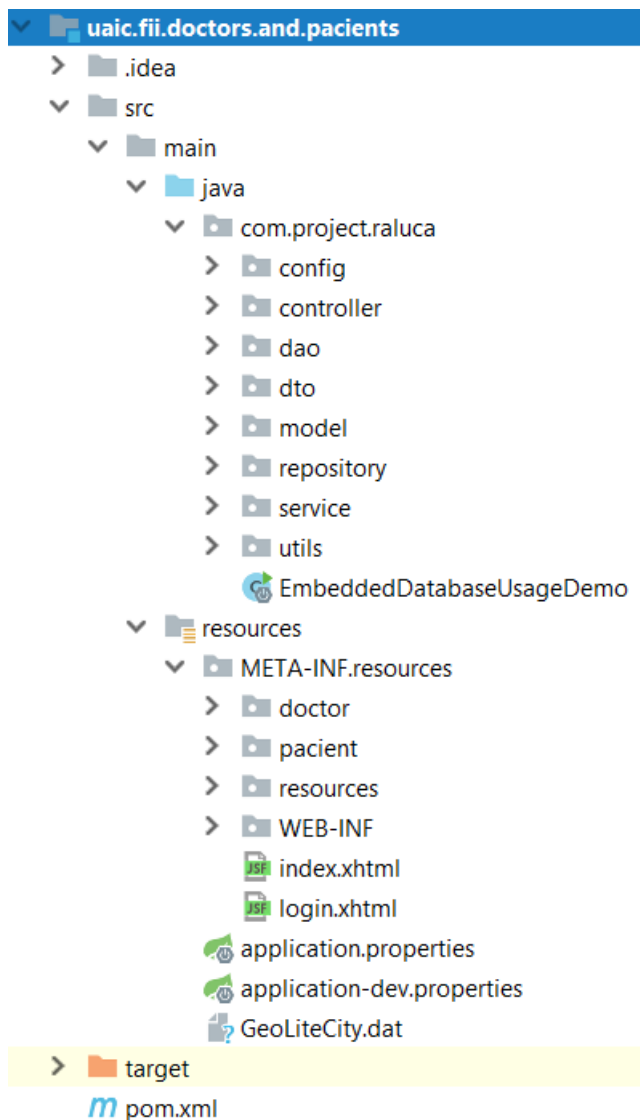
5.5 Structura Aplicației

Aplicația are la bază un singur modul care conține mai multe pachete împărțite în funcție de responsabilitatea pe care acestea o poartă, dar și după modelul celor 3 nivele regăsite în arhitectura aplicațiilor Spring.



În cadrul acestei structuri am respectat câteva principii de dezvoltare :

- Separation of Concern (SoC) Principle care se referă la faptul că o aplicație ar trebui să fie divizată în secțiuni, iar fiecare secțiune să trateze anumite informații care afectează code-ul aplicației.
- Single Responsibility Principle care susține că fiecare modul sau clasă trebuie să aibă responsabilitate asupra unei singure părți ale funcționalității furnizate de un sistem software.
- Keep it Simple Stupid (KISS) care susține că trebuie să păstrăm codul simplu și clean pentru a fi ușor de înțeles.



Așadar, în pachetul `com.project.raluca` regăsim grupate subpachetele după principiile descrise mai sus.

Din cadrul web layer-ului face parte pachetul **`com.project.raluca.controller`** care este responsabil cu procesarea input-ului primit de la user și de a returna un răspuns. Acest nivel este punctul de intrare în aplicație, așa că este responsabil și de autentificare.

Pachetul **`com.project.raluca.service`** conține business logic-ul aplicației și funcționează ca o bariera a tranzacțiilor, prin care manageriază entitățile și comunică cu alte servicii sau cu nivelul repository.

Pachetul **`com.project.raluca.repository`** este responsabil pentru comunicarea cu datele stocate folosite.

Pachetul **`com.project.raluca.model`** conține reprezentările tabelor din baza de date sub formă de entități, dar și relațiile dintre acestea. Tot odată, aici se află și clasele de tip enum.

Pachetul **`com.project.raluca.dto`** conține clase ce reprezintă obiecte de transfer (Data Transfer Object) care înglobează date ce sunt folosite între procese cu scopul de a reduce numărul de call-uri al metodelor și de a furniza anumite date mult mai ușor și accesibil.

Pachetul **`com.project.raluca.config`** conține clase responsabile cu realizarea configurărilor legate de baza de date sau securitate.

În pachetul **`resources`** se află paginile `.xhtml` și resursele multimedia.

Și nu în ultimul rând, în cadrul modulului `uaic.fii.doctors.and.pacients` se regăsește documentul `.xml` numit POM (Project Object Model) care este conceput central în Maven și care descrie întregul proiect.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>uaic.fii.doctors.and.pacients</artifactId>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.0.RELEASE</version>
  </parent>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>1.8</java.version>
    <joinsfaces.version>3.3.0-rc2</joinsfaces.version>
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.joinsfaces</groupId>
        <artifactId>joinsfaces-dependencies</artifactId>
        <version>${joinsfaces.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>org.joinsfaces</groupId>
      <artifactId>primefaces-spring-boot-starter</artifactId>
    </dependency>
  </dependencies>
</project>
```

Maven este unul dintre cele mai cunoscute *project builder*-e. Pe lângă capabilitățile de build, Maven poate rula rapoarte, genera rapoarte sau facilita comunicarea între membrii unei echipe de lucru.

În mod minimal, pom-ul definește o denumire unică (`groupId` și `artifactId`) și o versiune, precum și detalii de configurare astfel încât Maven poate builda proiectul fără alte elemente de bază.

Tot în cadrul acestui fișier regăsim o listă de dependențe— care reprezintă librării java împachetate sub forma unui `.jar` pe care Maven le gestionează și le adaugă în cadrul proiectului.

5.5 Detalii de implementare

Conform capitolului precedent, aplicația este structurată pe nivele de responsabilități, așa că voi continua să prezint cum a fost construită aplicația începând de la cel mai de jos nivel.

Primul pas a fost construirea structurii bazei de date, entitățile și legăturile dintre ele. Un exemplu de entitate este clasa Doctor :

```
@Entity
@Table(name = "Doctor")
public class Doctor extends AbstractEntity implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "user_id")
    private Users user;
    private String firstName;
    private String lastName;

    @Enumerated(EnumType.STRING)
    private InsuranceHouse insuranceHouse;

    @Enumerated(EnumType.STRING)
    private Speciality speciality;

    @OneToMany(mappedBy="doctor", cascade =
CascadeType.ALL, fetch=FetchType.EAGER)
    @Fetch(value = FetchMode.SUBSELECT)
    private List<Review> reviewList;

    @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    @JoinTable(
        name = "Doctor_Pacient",
        joinColumns = { @JoinColumn(name = "doctor_id") },
        inverseJoinColumns = { @JoinColumn(name = "pacient_id") }
    )
    @Fetch(value = FetchMode.SUBSELECT)
    private List<Pacient> patientsList;
```

Aici specificăm faptul că obiectul descris este o entitate ce persistă în baza de date cu ajutorul adnotării `@Entity` și specificăm numele tabelului corespunzător din baza de date `@Table(name = "Doctor")`. Relațiile dintre doctor și restul entităților sunt marcate cu ajutorul adnotărilor `@OneToOne`, `@OneToMany` și `@ManyToMany`. Aceste adnotări conțin parametri precum `fetch`, `cascade` sau `orphanRemoval`.

Parametrul *fetch* poate avea mai multe valori, printre care cele mai utilizate *Lazy* și *Eager*, reprezentând modalitatea de încărcare a datelor. Parametrul *orphanRemoval*, ce poate avea valoarea *true* sau *false*, este utilizat pentru prevenirea păstrării obiectelor fără referință, atunci când un obiect este șters, toate obiectele ce sunt legate prin cheie străină de obiectul șters vor fi șterse. Adnotarea `@Enumerated` este folosită în corelație cu obiectele de tip `enum`, astfel încât valorile unei proprietăți ale entității pot fi doar valorile specificate în `enum`.

Odată ce am construit clasele de entități, urmatorul pas a fost să avem reprezentarea lor în baza de date. Cel mai important plus pe care îl aduce H2 Database este că, odată configurată în cadrul proiectului, aceasta generează toate tabelele și relațiile dintre ele, fara a fi nevoie de un script de inițializare. Așadar, conexiunea la consola oferită de baza de date H2 este specificată în fișierul de configurare application.properties:

```
spring.datasource.url=jdbc:h2:file:~/test;DB_CLOSE_DELAY=-1;
spring.datasource.username=sa
spring.datasource.password=admin
spring.datasource.driver-class-name= org.h2.Driver
spring.h2.console.enabled=true
```

Având deja structura datelor ce vor fi manipulate în cadrul aplicației, putem vedea cum se realizează acest lucru începând cu cel mai de jos nivel: REPOSITORY.

În cadrul acestui nivel implementat de interfețele aflate în pachetul **com.project.raluca.repository** sunt furnizate funcționalități de tipul CRUD (Create, Read, Update, Delete) prin extinderea interfeței JpaRepository pusă la dispoziție de Spring Data.

```
@Repository
public interface UserDoctorRepository extends JpaRepository<Doctor,
Integer> {
    Doctor findByUserUsername(String username);
}
```

Adnotarea @Repository indică faptul ca interfața UserDoctorRepository este responsabilă cu persistența datelor. Metodele care implementează această responsabilitate se află în cadrul JpaRepository și sunt de tipul : findAll(), save(entity), findById(id), delete(entity). Pe lângă aceste metode pe care interfața noastră le moștenește prin extinderea interfeței JpaRepository, ne putem defini alte metode specifice nevoilor noastre în cadrul dezvoltării.

Urmatorul nivel al aplicației, cel care se ocupa de toata logica aplicației este nivelul Service, care este implementat de clasele din pachetul **com.project.raluca.service**.

```
@Service
public class UserDoctorService {

    @Autowired
    private UserDoctorRepository userDoctorRepository

}
```

Adnotarea @Service indică faptul că este o componentă ce implementează diferite funcționalități ce țin de comportamentul entității Doctor dar și de relaționarea cu restul entităților.

În cadrul unui serviciu regasim conceptul de Dependency Injection din Spring indicat de adnotarea @Autowired. Acesta ajută la rezolvarea și injectarea altor bean-uri, cum este UserDoctorRepository în cazul nostru.


```

@Transactional(
    readOnly = false, propagation = Propagation.REQUIRED,
    rollbackFor = { IllegalArgumentException.class,
        IllegalAccessException.class})

public UserDoctorDTO create(final UserDoctorDTO doctor) {
    Doctor userDoctor = (Doctor) dtoHelper.convertToEntity(doctor,
        Doctor.class);
    return (UserDoctorDTO) dtoHelper.convertToDTO(
        userDoctorRepository.save(userDoctor), UserDoctorDTO.class);
}

```

Un exemplu de funcționalitate pe care o poate conține un serviciu este metoda de create() care primește ca parametru un obiect de tipul UserDoctorDTO și cu ajutorul interfeței repository, poate invoca metoda de save() care face ca obiectul să persiste în baza de date.

@Transactional are la bază două concepte importante : contextul persistenței și tranzacția bazei de date. Annotarea tranzacțională definește sfera unei tranzacții dintr-o singură bază de date. Tranzacția bazei de date se întâmplă în interiorul unui *context de persistență*.

Contextul de persistență este în JPA EntityManager, implementat intern utilizând Hibernate Session (când se utilizează Hibernate ca furnizor de persistență).

Contextul de persistență este doar un obiect de sincronizare care urmărește starea unui set limitat de obiecte Java și asigură că schimbările asupra acelor obiecte sunt în cele din urmă persistente înapoi în baza de date.

Aceasta este o noțiune foarte diferită de cea a unei tranzacții de bază de date. Un Entity Manager poate fi utilizat în mai multe tranzacții de baze de date , de fapt, de multe ori.

O altă funcționalitate ce poate fi regăsită într-un serviciu este metoda de inițializare care se execută după ce bean-urile aplicației au fost inițializate și este adnotată cu @PostConstruct

```

@PostConstruct
public void bootstrap() throws ParseException {

    Doctor userDoctor = new Doctor();
    userDoctor.setFirstName("test");
    userDoctor.setLastName("plugariu");
    userDoctor.setGender(Gender.FEMALE);
    userDoctor.setUser(user);
    userDoctor.setRange(Range.PROFESOR);
    userDoctor.setSpeciality(Speciality.NEUROLOG);
    userDoctor.setInstitution(institution);
    review.setDoctor(userDoctor);
    review2.setDoctor(userDoctor);
    userDoctor.setReviewList(reviewList);

    userDoctorRepository.save(userDoctor);
}

```

Am folosit această metodă pentru a inițializa și a salva anumite entități în baza de date, fără a crea vreun script sql.

Așa cum spuneam, aici găsim și metode care conțin un flow logic al unui anumit context. Să luăm spre exemplu cazul în care un doctor acceptă o programare iar asta presupune update-ul obiectului ce reprezintă programarea și anume Appointment, trimiterea unei notificări, dar și ștergerea din calendar a slot-ului de timp înlocuit de programare.

```
public UserDoctorDTO acceptAppointment(int userPacientID, int doctorID,
AppointmentDTO appointmentDTO) {
    appointmentDTO.setAppointmentStatus(AppointmentStatus.ACCEPTED);
    appointmentService.update(appointmentDTO);
    UserDoctorDTO doctor = this.get(doctorID);
    Notification notification = new Notification();
    notification.setType(NotificationType.APPOINTMENT_REJECTED);
    notification.setContent("You appointment request for doctor " +
doctor.getUser().getUsername() + " has been accepted");
    notification.setCreateDate(new Date());
    notification.setSeen(Boolean.FALSE);
    UserPacientDTO userPacientDTO = userPacientService.get(userPacientID);
    userPacientDTO.getNotificationList().add(new NotificationDTO());

    //delete from bookableTimes

    userPacientService.update(userPacientDTO);
    return this.update(doctor);
}
```

O funcționalitate importantă în această aplicație este cea de localizare a unui user. Pașii în implementarea acesteia au fost următorii:

1. Obținerea adresei IP publice a device-ului de pe care este conectat userul

```
String ip = "";
try {
    URL whatismyip;

    whatismyip = new URL("http://checkip.amazonaws.com");

    BufferedReader in = null;

    in = new BufferedReader(new
InputStreamReader(whatismyip.openStream()));

    ip = in.readLine();

} catch (UnknownHostException e) {
    e.printStackTrace();
}
```

După cum de poate vedea, acest serviciu public furnizează adresa ip a computerului, iar cu ajutorul metodei `openStream()` am putut citi din acest URL.

2. obținerea coordonatelor de longitudine și latitudine în funcție de adresa IP

```
private static LookupService lookup;

static {
    try {
        lookup = new LookupService(
            GeoIPv4.class.getResource("/GeoLiteCity.dat").getFile(),
            LookupService.GEOIP_MEMORY_CACHE);

        System.out.println("GeoIP Database loaded: " +
            lookup.getDatabaseInfo());
    } catch (IOException e) {
        System.out.println("Could not load geo ip database: " +
            e.getMessage());
    }
}

public static GeoLocation getLocation(String ipAddress) {
    return GeoLocation.map(lookup.getLocation(ipAddress));
}
```

Pentru acest lucru am folosit o librărie open-source GeoIP, și un fișier `GeoLiteCity.dat` ce reprezintă o bază de date. Metoda `getLocation()` returnează un obiect de tipul `GeoLocation` care arată astfel :

```
public class GeoLocation {

    private String countryCode;
    private String countryName;
    private String postalCode;
    private String city;
    private String region;
    private int areaCode;
    private int dmaCode;
    private int metroCode;
    public float latitude;
    public float longitude;
}
```

Acest obiect conține toate informațiile necesare în vederea localizării pe harta a unui punct.

Ultimul nivel al aplicației, nivelul Web conține clase ce țin modelul și comportamentul unei pagini web. Aici se regăsesc clase de control cu adnotări specifice JSF: @ManagedBean cu diferite adnotări ce definesc ciclul de viață al clasei : @ViewScoped, @SessionScoped, @RequestScoped.

Managed Bean este o clasă Java Bean ce este procesată de framework-ul JSF. Aceasta conține metodele *getter* și *setter*, partea logică a aplicației și toate valorile formularelor HTML. Managed Bean poate fi accesată dintr-o pagină JSF și lucrează ca un model pentru componentele User Interface.

Adnotările de tip *Scope* specifică ciclul de viață al unui Managed Bean. Acestea pot fi de tipul:

- RequestScoped – Bean-urile își încheie ciclul de viață odată cu cererea/ răspunsul HTTP.
- NoneScoped – Bean-urile sunt create în cadrul unei evaluări EL (Expression Language) și sunt distruse după finalizarea acesteia.
- ViewScoped – Bean-urile sunt menținute cât timp utilizatorul interacționează cu același view JSF în browser. Sunt create odată cu cererea HTTP și distruse atunci când este accesat un alt view.
- SessionScoped – Bean-urile își mențin ciclul de viață odată cu sesiunea. Sunt create odată cu prima cerere HTTP care face referire la acestea și sunt distruse atunci când sesiunea este invalidată.
- ApplicationScoped – Bean-urile își desfășoară ciclul de viață odată cu aplicația în sine.
Sunt create în momentul în care prima cerere HTTP face referire la acestea sau la pornirea aplicației, fiind distruse la închiderea ei.
- CustomScoped – Ciclul de viață al Bean-urile poate fi definit de către programator.

Un exemplu este clasa `UserDoctorController`, o clasă ce își păstrează datele pe durata întregii sesiuni.

```
@SessionScope
@Named("userDoctorController")
public class UserDoctorController {

    @Autowired
    private UserDoctorService userDoctorService;

    public UserDoctorDTO userDoctorDTO;

    private static final Logger log =
        LoggerFactory.getLogger(UserDoctorService.class);

    public List<UserDoctorDTO> getAll() {
        List<UserDoctorDTO> list = userDoctorService.getAll();
        return list;
    }

    public void create(UserDoctorDTO doctorDTO) {
        userDoctorService.create(doctorDTO);
    }

    public UserDoctorDTO getCurrentUserDoctor() {
        Authentication authentication =
            SecurityContextHolder.getContext().getAuthentication();
        return userDoctorService.findDoctorByUsername(authentication.getName())
    }
}
```

Clasa `UserDoctorController` furnizează informații legate de userul care este logat precum și alte date pe care le conține doctorul în baza de date care vor fi folosite în completarea view-urilor xhtml.

```
<h3 class="profile-username text-center">#{userDoctorController.userDoctorDTO.firstName}
#{userDoctorController.userDoctorDTO.lastName}
</h3>

<p class="text-muted text-center">
#{userDoctorController.userDoctorDTO.range}</p>

<ul class="list-group list-group-unbordered">
    <li class="list-group-item">
        <b>Institution</b> <a class="pull-right">
#{userDoctorController.userDoctorDTO.institution.name}</a>
    </li>
    <li class="list-group-item">
        <b>Specialisation</b> <a class="pull-right">
#{userDoctorController.userDoctorDTO.speciality}</a>
    </li>
</ul>
```

Pentru partea de securitate a aplicației am folosit Spring Security care este un framework pentru autentificare și control al accesului.

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // require all requests to be authenticated except for the
        resources

        http.authorizeRequests().antMatchers("images/**", "/webjars/**", "/login.jsp"
        , "/META-INF/resources/index.xhtml").permitAll();

        // login
        http.formLogin().loginPage("/login.xhtml")
            .successHandler(successHandler)
            .failureUrl("/login.xhtml?error=true")
            .permitAll();

        http.authorizeRequests().antMatchers("/javax.faces.resource/**")
            .permitAll().anyRequest().authenticated();

        // logout
        http.logout().logoutSuccessUrl("/login.xhtml");
        http.csrf().disable();
    }

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth)
        throws Exception {

        auth.jdbcAuthentication().dataSource(dataSource);
    }
}
```

Clasa SecurityConfig conține configurările pentru partea de autentificare care se folosește de baza de date prin jdbcAuthentication().

```
if(authority.getAuthority().equals("ROLE_DOCTOR")) {
    try {
        redirectStrategy.sendRedirect(arg0, arg1,
        "/doctor/homeD.xhtml");
    } catch (Exception e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
} else if(authority.getAuthority().equals("ROLE_PACIENT")) {
    try {
        redirectStrategy.sendRedirect(arg0, arg1,
        "/pacient/homeP.xhtml");
    }
}
```

Iar controlul accesului este realizat în clasa SimpleAuthenticationSuccessHandler în funcție de rolul fiecărui user.

6. Modul de utilizare al aplicației

În acest capitol voi prezenta pas cu pas modul de utilizare al aplicației împreună cu toate funcționalitățile și serviciile oferite. Datorită faptului că există 2 roluri de utilizator, aceștia vor avea acces la anumite pagini destinate lor, astfel încât doctorul va putea accesa paginile ce conțin în url-ul lor /doctor/*, iar pacientul /pacient/*. Dar cum am precizat în capitolul precedent, aceștia vor fi redirectionați în mod corespunzător rolurilor lor.

Prima pagină și primul contact cu această aplicație este reprezentat de pagina index, care poate fi accesată de orice utilizator chiar dacă nu este logat.



Ea conține 2 elemente intuitive, corelate cu rolurile aplicației, doctor și pacient. Astfel utilizatorul alege tipul de login pe care și-l dorește, urmând să fie redirectat către pagina de login.

Please login

User name

Password

Login

După selectarea elementului dorit, se va încărca pagina de login care conține 2 câmpuri: username și password. Aici credențialele sale sunt validate, iar dacă acestea nu sunt corecte, utilizatorul va rămâne pe pagina de login care va conține un mesaj de eroare, având șansa să încerce încă o dată cu alte credențiale. Dacă acestea vor fi validate ca fiind corecte, utilizatorul va fi redirectat către paginile de home corespunzătoare rolului său.

6.1 Rolul Doctor

Odată logat, angajatul are parte de o interfață intuitivă ce oferă o experiență plăcută utilizatorului. Pagina homeDoctor conține un Dashboard unde se regăsesc majoritatea funcționalităților pe care la oferă aplicația.

DOCWEB

test plugariu

Welcome test plugariu !

0 Patients

1 Specializations

2 Reviews

3 Score

Appointments for today

Patients

Name: Ion Dumbrava
Title: Consultation
Start time: 2019-06-25 08:00:00.0

Calendar

Set up your availability

Current Date June 25, 2019

Month Week Day

Tuesday

All Day

6am

7am 7:00 - 8:00 Free slot

8am 8:00 - 9:00 Free slot

9am 9:00 - 10:00 Free slot

10am

11am

12pm

1pm

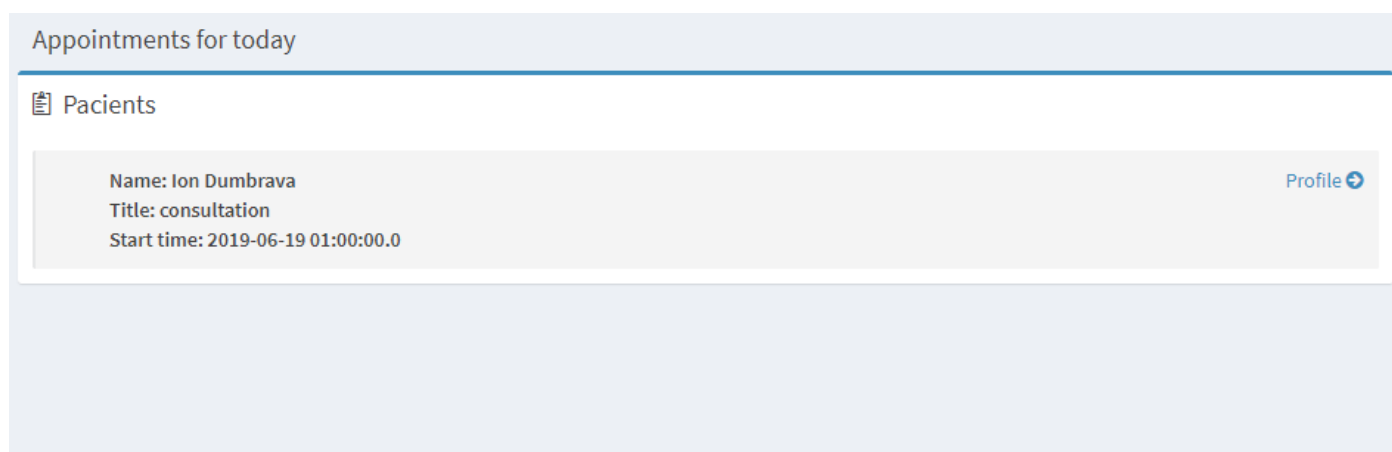
2pm

3pm

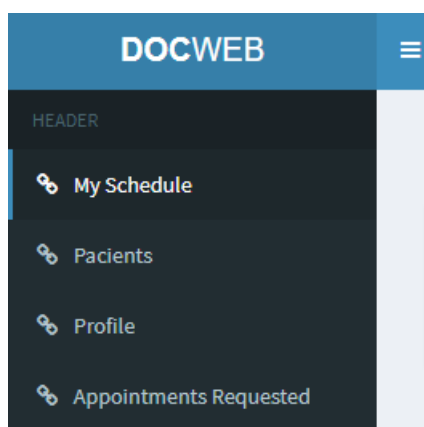
Dashboard-ul conține elemente informative legate de numărul de pacienți, specializările, numărul de review-uri și scorul obținut în urma evaluărilor din partea pacienților.

Rolul principal al acestei pagini este să informeze doctorul cu privire la programările din ziua respectivă așadar în centru este afișată lista pacienților din acea zi.

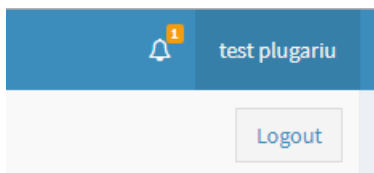
Calendarul din partea stângă oferă posibilitatea doctorului de a seta orele la care acesta este disponibil pentru programări.



Sunt afișate numele pacientului, tipul programării și ora la care va avea loc. În partea din dreapta se află un link care redirecționează către profilul pacientului respectiv pentru a avea o imagine mai amplă asupra lui.



În partea din stânga a paginii se află meniul ce conține link-uri către toate paginile pe care doctorul le poate accesa : orarul, lista pacientilor, profilul doctorului sau lista programărilor cerute de către pacienți.



Iar în partea din dreapta paginii se află o iconiță ce indică numărul de notificări pe care doctorul le-a primit, iar langa acesta se află numele doctorului care este un buton ce ascunde funcția de logout. Odată apăsând butonul de logout, doctorul va fi redirecționat pe pagina de index.

Așadar, navigând prin meniu și alegând pagina My Schedule, vom fi redirecționați către pagina în care doctorul își poate manageria calendarul.

Schedule

Current Date June 2019 Month Week Day

Sun	Mon	Tue	Wed	Thu	Fri	Sat
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13		
16	17	18	19 12a consultation	20		
23	24	25	26	27		
30	1	2	3	4	5	6

Event Details

Titles: * Operation

From: 21/06/2019 00:00

To: 21/06/2019 02:00

Participants: Ion Dumbrava

Reset

☒ Ion Dumbrava

În cadrul acestei pagini doctorul poate adăuga diferite evenimente apăsând în dreptul unei anumite zile și i se va deschide un pop-up în care acesta poate completa informațiile legate de evenimentul pe care îl va crea : titlul, intervalul de timp în care va avea loc și participanții, aceștia fiind aleși dintr-un drop-down ce conține lista tuturor pacienților doctorului.

Aceste evenimente pot fi afișate în intervalul unei luni, cum este în figura de mai sus, pe intervalul unei săptămâni sau unei zile, doar apăsând pe unul din cele 3 butoane din partea dreapta : Month, Week, Day.

Pagina de profil a doctorului conține toate informațiile personale ale acestuia pe care le poate și edita. Deasemenea acesta poate vedea și lista review-urilor lăsate de pacienți.

test plugariu
PROFESOR

Institution
arcadiatest

Specialisation
NEUROLOG

About Me

Education

Facultatea de Informatica Iasi

Location

RO Iasi str 2

Phone

0751897931

Edit Profile

First Name

test

Last Name

plugariu

Email

raluca.pluginu

Phone

0751897931

Range

PROFESOR

Submit

Reviews

!!!!!!

Writed by: Ion Dumbrava

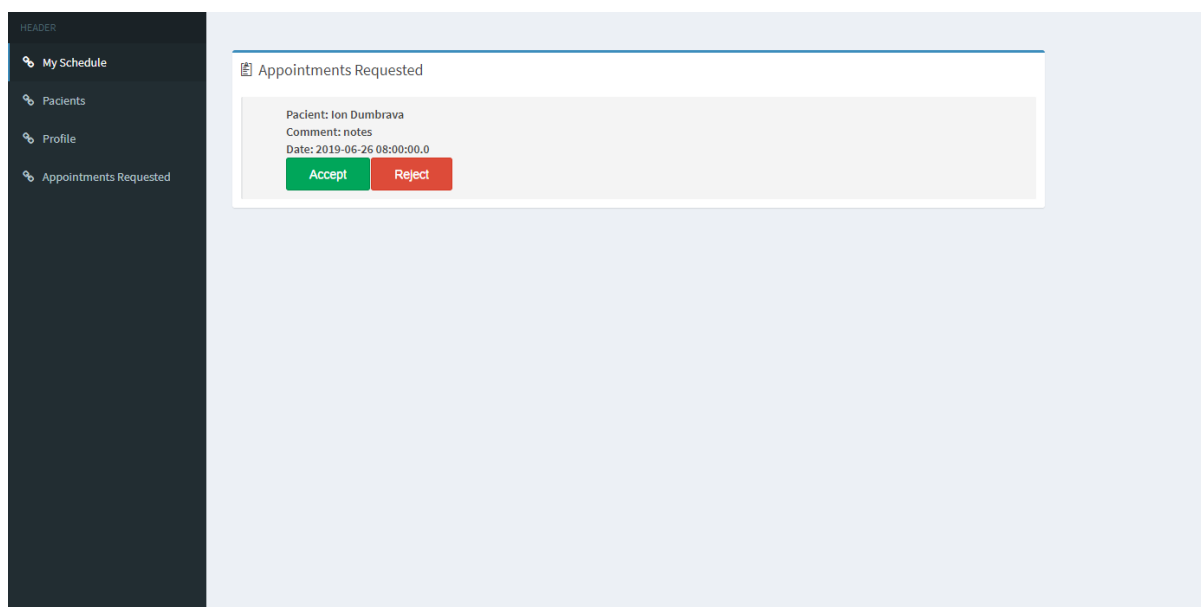
★★★★☆

??????

Writed by: Ion Dumbrava

★★★★☆

Pagina Appointments Requests prezintă doctorului lista de programări solicitate de către pacienți.

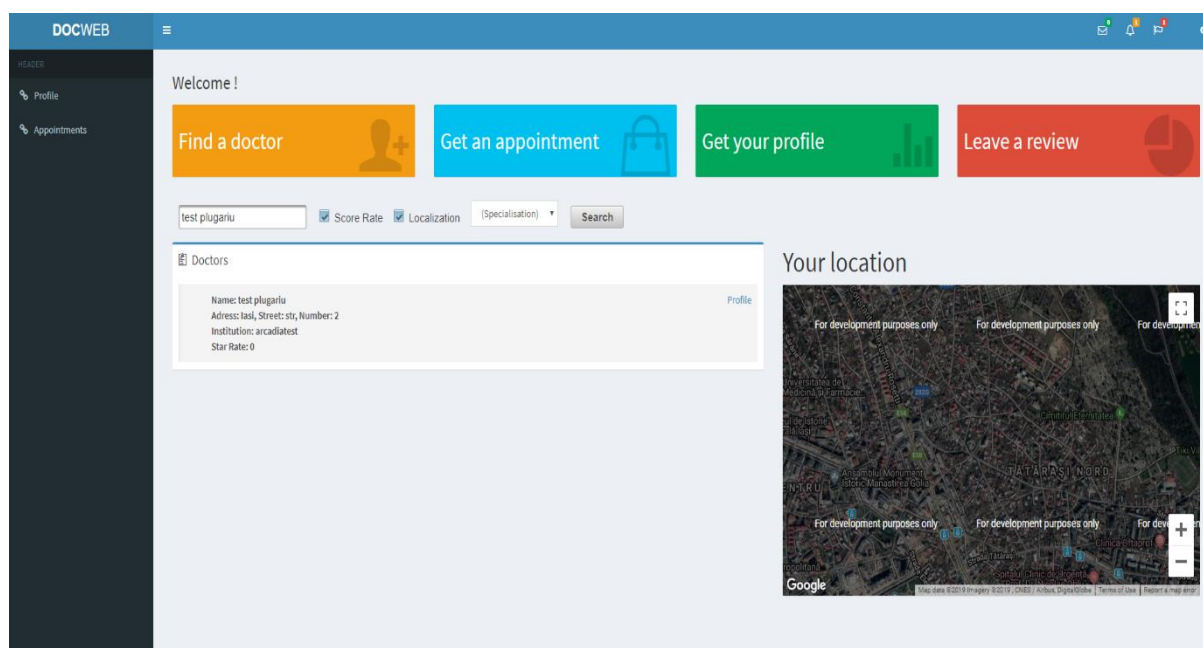


Doctorul are la dispoziție două butoane prin care poate accepta sau refuza cererea pacinetului.

În ambele cazuri, pacinetul va primi o notificare prin care este informat cu privire la statusul cererii de programare: acceptată sau refuzată.

6.2 Rolul Pacient

Asa cum am prezentat în introducere, aplicația încearcă să rezolve problema interacțiunii dintre pacient și doctor. Așa că am ales ca pe pagina pe care este redirecționat pacientul după login, acesta să găsească motorul de căutare doctori, unul din motivele principale pentru care un pacient folosește această aplicație.



Această pagină conține sugestii cu privire la funcționalitățile pe care aplicația le furnizează, astfel încât orice utilizator sau tip de pacient să poată lua la cunoștință lucrurile care îi stau în putere să le facă în această aplicație: Căutarea unui doctor, Cererea unei programări, Formarea unui profil și Trimiterea unui review pe pagina doctorului.

Am ales să introduc și acea hartă în pagina de home pentru a confirma pacienților faptul că aplicația a reușit să îi localizeze iar dacă aceștia vor dori să caute doctori în funcție de locație, reperul față de care se caută este unul corect.

plu ☐ Score Rate ☐ Localization (Specialisation)

test plugariu

DOCTORS

Name: test plugariu [Profile](#)
 Address: Iasi, Street: str, Number: 2
 Institution: arcadiatest
 Star Rate: 0

Căutarea doctorilor se poate face după nume, acesta fiind introdus într-un input box de tipul autocomplete, astfel că pacientul primește sugestii de doctori din baza de date care conțin în cadrul numelor lor, caracterele introduse de pacienți. De asemenea, ei pot căuta după o specializare anume, după nota pe care o deține doctorul pe profilul lui sau după locația în care acesta lucrează, sortându-se astfel lista în funcție de distanța dintre pacient și doctor.

În partea stângă a unui rezultat se află un link care trimite către profilul doctorului care arate în felul următor:

<p>test plugariu PROFESOR</p> <p>Institution arcadiatest</p> <p>Specialisation NEUROLOG</p> <p>Location Iasi, Street: str, Number: 2</p> <p>Star rate ★★★★★</p> <p>Availability</p> <p>Pick a date for your appointment</p> <p><input type="button" value="Check Availability"/></p>	<p>Profile</p> <p>First Name <input type="text" value="test"/></p> <p>Last Name <input type="text" value="plugariu"/></p> <p>Email <input type="text" value="raluca.plugariu"/></p> <p>Range <input type="text" value="PROFESOR"/></p> <p>Insurance House <input type="text"/></p> <p>Reviews</p> <p>!!!!!!! Writed by: ★★★★★</p> <p>??????? Writed by: ★★★★★</p> <p>Leave a review</p> <p>★★★★★</p> <p><input type="text"/></p> <p><input type="button" value="Save"/></p>
--	---

Odată redirectat pe profilul doctorului vor fi afișate date de contact, informații generale, dar și lista review-urilor pe care doctorul le deține.

Reviews

!!!!!!??
Writed by:
☆☆☆☆☆

??????
Writed by:
☆☆☆☆☆

Leave a review

☆☆☆☆☆

Save

Fiecare review are afisat conținutul, autorul și numarul de stele atașate.

Același lucru îl poate face pacientul completând input box-ul cu feedback-ul pe care dorește să îl ofere, alegând un numar de stele și apăsând pe butonul send, review-ul va apărea în lista de mai sus, iar doctorul va primi o notificare în această privință.

Availability

Pick a date for your appoinment

06/25/2019

Calendar icon

Check

7:00

8:00

9:00

Acest panou ofera încă o funcționalitate foarte importantă pentru pacienți și anume aceea de a verifica intervalele de timp în care doctorul este disponibili pentru programări. Pacientul trebuie să aleagă o zi din calendar, iar apăsând pe butonul Check Availability, i se vor afișa o listă de ore disponibile, dacă acestea există. Daca nu există, pacientul poate cauta o altă dată.

Odată ce pacientul primește o listă cu intervalele disponibile, acesta își poate alege o anumită oră pentru o programare doar apăsând pe ora respectivă, iar acestuia i se va afișa un dialog prin care trebuie să confirme că își dorește o programare la acel doctor, în data respectivă. Dacă pacientul confirmă, cererea va fi trimisă de către doctor, iar pacientul o va putea urmări în pagina de Appointments dacă va fi acceptată sau nu.

7. Concluzii si directii viitoare

În această lucrare a fost prezentată evoluția aplicației DOCWeb, de la stadiul de concept până la implementare și funcționare. Consider că aplicația ar putea fi de ajutor atât doctorilor cât și pacienților acestora. Pe viitor s-ar putea încerca perfecționarea ei astfel:

1. Realizarea unei versiuni pentru dispozitive mobile pentru ca utilizatorii să aibă acces la aplicație chiar și atunci când nu sunt la birou.
2. O integrare a serviciilor oferite de Google Maps astfel încât să poată fi calculată ruta dintre două locații
3. Posibilitatea comunicării prin video call între doctor și pacient

Bibliografie

1. Curs practic de Java, Cristian Frăsinaru
2. andrei.clubcisco.ro/cursuri/f/f-sym/5master/aac-dai/Curs10.doc
3. https://www.tutorialspoint.com/jsf/jsf_tutorial.pdf
4. <http://enos.itcollege.ee/~jpoial/allalaadimised/lugemist/JSF-2.0-Programming-Cookbook.pdf>
5. Java Persistence with Hibernate, Christian Bauer
6. <https://spring.io/>
7. Head First EJB, Kathy Sierra